Distributed Shared Memory for Large-Scale Dynamic Systems

Vincent Gramoli

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE | INRIA

UNIVERSITÉ DE RENNES 1

INRIA Research Centre Rennes
INRIA Futurs Saclay

Université de Rennes 1

Distributed Shared Memory for Large-Scale Dynamic Systems

A dissertation presented to

Université de Rennes 1

in partial fulfillment of the requirements

for the degree of

Doctor of Philosophy

in the subject of

Computer Science

by

Vincent Gramoli

defended on Thursday, November 22nd 2007 in front of the jury composed of:

Antonio Fernández	Professor	Examiner
Roy Friedman	Professor	Referee
Anne-Marie Kermarrec	Senior Researcher	Examiner
Michel Raynal	Professor	Advisor
Pierre Sens	Professor	President
Alex Shvartsman	Professor	Referee

Year 2007
Numéro d'Ordre : 3637

École Doctorale Matisse

This thesis focuses on newly arising challenges in the context of data sharing due to the recent scale shift of distributed systems. Distributed systems keep enlarging very rapidly. Not only more persons use their computer to communicate all over the world, but the amount of individual objects that get connected is increasing. Such large scale systems experience an inherent dynamism due to the unpredictability of the users behaviors. This drawback prevents traditional solutions from being adapted to this challenging context. More basically, this affects the communication among distinct computing entities. This thesis investigates the existing research work and proposes research suggestions to solve a fundamental issue, the distributed shared memory problem, in such a large scale and inherently dynamic environment.

Keywords: Distributed shared memory, quorum, atomicity, consistency, reconfiguration, dynamic, large-scale.

To my wife and my family

Acknowledgments

I wish to thank especially professor Michel Raynal who accepted to become my advisor while my subject was partially defined. I wish to thank Anne-Marie Kermarrec that encouraged my research at any time, I owe her a lot. Thanks to Alex Shvartsman for having accepted to review my thesis and with whom I lived a great research experience at very stimulating locations. Thanks to Roy Friedman to review my thesis and to Antonio Fernández and Pierre Sens, who accepted to examined my PhD work. I am grateful to Carole Delporte, Hugues Fauconnier, Matthieu Latapy, and Pierre Fraigniaud for having taught me very interesting fields of computer science and helping me to find my path into research. Also, I would like to thank Emmanuelle Anceaume, Maria Gradinariu, and Michel Hurfin with whom I discovered Peer-to-Peer systems, and those who enlightened me about the French doctoral status. Thanks to my friends Nic, Andri, Piotr, Monika, Alexis, and Kishori from Storrs; Achour, Aline, Corentin, Erwan, Etienne, Fabrice, François, Gilles, Marin, and Yann, the task force of the ASAP group; and Julien, Ludovic, Jean-Marie, Audrey, and Jean-Pierre from Brittany. Last but not least, thanks to all my co-authors who accepted working with me including Seth, Grisha, Toni, Achour, Ernesto, Bruno, Márk.

Citations to Previously Published Work

The essential of Chapter 2 appeared in the following papers:

> ”Reconfigurable Distributed Storage for Dynamic Networks”, G. Chockler, S. Gilbert, V. Gramoli, P. M. Musial, and A. A. Shvartsman, Proceedings of the OPODIS, 2005.
>
> ”Mémoire partagées distribuées pour systèmes dynamiques à grande échelle”, V. Gramoli, Proceedings of Journées Doctorales en Informatique et Réseaux, 2007.

The most important part of Chapter 3 appeared in the following papers:

> ”Peer-to-Peer Architecture for Self-* Atomic Memory”, E. Anceaume, M. Gradinariu, V. Gramoli, and A. Virgillito, Proceedings of ISPAN, 2005.
>
> ”SQUARE: A Scalable Quorum-based Atomic Memory with Local Reconfiguration”, V. Gramoli, E. Anceaume, A. Virgillito, Proceedings of the ACM SAC, 2007 and IRISA technical report number 1805.

Portions of Chapter 4 appeared or will appear in the following papers:

> ”Core Persistence in Peer-to-Peer Systems: Relating Size to Lifetime”, V. Gramoli, A.-M. Kermarrec, A. Mostéfaoui, M. Raynal, and B. Sericola, Proceedings of the On The Move Workshops, 2006 and IRISA technical report number 1799.
>
> ”Timed Quorum Systems for Large-Scale and Dynamic Environments”, V. Gramoli and M. Raynal, Proceedings of the OPODIS, 2007.

Portions of Chapter 4 and almost the whole Appendix A appeared in the following paper:

> ”Distributed Slicing in Dynamic Systems”, A. Fernández, V. Gramoli, E. Jiménez, A.-M. Kermarrec, M. Raynal, Proceedings of the ICDCS, 2007 and INRIA technical report number 6051.

Electronic preprints are available on the Internet at the following URL:

> `http://www.irisa.fr/prive/vgramoli/php/pub_irisa_year.php`

Contents

Introduction

The Scale-shift of Distributed Systems

Distributed systems is now a major research topic in the field of computer science. The interest and research efforts devoted to this topic have continuously increased since the late seventies. Nowadays, distributed systems have drastically changed and new fundamental issues have arisen. We believe that the major cause for these topical issues is the recent scale-shift that distributed systems experience.

The need of resources has been one of the main motivations behind distributed systems. Indeed, take computational resources as an example. Multiprocessor systems allow to share the computational tasks on different processors while distributed machines can execute multiple computational tasks at the same time, one on each machine. Nowadays, while powerful multiprocessor systems remain expensive a novel form of computational collaboration arises due to the enhancement of the Internet and various networks. The collaboration of distributed machines produces more resources than any single machine can afford. First observations of this phenomenon appeared with the SETI@home project that was launched on 1999 and that provides an average of 264 TeraFlops thanks to the collaboration of millions of machines.[1] Governed by the will of users to obtain more resources, distributed systems keep enlarging.

A peer-to-peer (p2p) system is a distributed system that has no centralized control. Actually, the p2p paradigm relies on the fact that distributed entities not only benefit from resources but also provide resources. In traditional distributed systems services are hosted by servers and accessed by clients, thus, in p2p systems every entity acts as a client and as a server at the same time. Peer-to-peer systems have gained in popularity with the massive utilization of file-sharing applications over the Internet, since 2000. These systems propose a tremendous amount of file resources. Beyond file sharing applications many research efforts have been devoted to assign content to nodes so that the content can be retrieved easily. These efforts result mainly in the definition of distributed hash tables [MKKB01, RFH+01, RD01] mapping each stored data to a key used to retrieve the corresponding data in the system.

Nowadays, there is an increasing amount of various computing devices surrounding us. First,

[1] SETI@home provides more than 264.10^{12} floating point operations per second on average as reported by BoincStats on July, 17th 2007.

the addressing space of the original Internet protocol, IPv4, becomes insufficient to face the demand of new connected entities and leads to the standardization of a new protocol, IPv6, to tolerate the growing amount of entities. Second, each person tends to use various computing devices that all communicate together through different wireless networks. Third, the decrease of the cost of sensors (like RFID) allows their production to enlarge and their deployment for monitoring applications. Finally, IDC predicts that there will be 17 billions of traditional network devices by 2012. In such contexts, it is common knowledge that scalability has become one of the most important challenges of today's distributed systems.

The Future of Communication between Numerous Entities

The scale-shift of distributed systems modifies the way computational entities communicate. In this context, connecting multiple and various entities together led to other interesting challenges due to their independent behavior. Energy dependence, disconnection, malfunctioning, and other environmental factors affect the availability of each computational entity independently. This translates into irregular periods of activity during which an entity can receive messages or compute tasks. As a result of this independent and periodic behaviors guided by environmental variations, these systems are inherently highly dynamic. Moreover, large-scale prevents a single entity from maintaining any global information about the system. Consequently, dynamism is hardly measurable and completely unpredictable. As a major drawback of scalability, dynamism strengthens the difficulty for entities to communicate.

Distributed systems can be differentiated along two major paradigms: shared-memory and message-passing. In shared-memory, entities communicate by reading from and writing to a single memory. In message-passing, entities communicate by sending and receiving messages whose transmission time is, in general, arbitrarily long. Traditionally, tightly-coupled distributed architectures, as multi-processor machines use the shared-memory model for communication among processors. The motivation for shared-memory stems from the simplicity with which algorithms can be designed and with which programs can be written compared to the message-passing model. Conversely, the prevalent model for loosely-coupled distributed architecture as network of workstations is message-passing. The motivation for message-passing stems from the ability to replicate on several workstations so that each workstation can fail independently without affecting the performance of the system. Despite the complex programming tasks message-passing requires, this model is more appropriate when message delays are arbitrarily long. This motivates the need for emulating shared-memory in message-passing model. This emulation, also known as distributed shared memory, is the subject of this thesis.

The objective of Distributed Shared Memory (DSM) is to emulate the functioning of shared-memory in message-passing model. From the standpoint of clients, the DSM must transparently appear as if it was a single shared-memory. Consequently, the DSM must provide clients with read and write primitives that, when executed, result in the reading and writing of data. The difficulty

comes from the fact that operations requested by distinct clients must be ordered: a client reads the value written by another client depending on the time the read and the write occurred. Ideally this ordering should respect real-time, however, distant clients are not synchronized and operations may execute concurrently. As a result, we need a set of properties on operations that must be satisfied by any execution of our DSM, in order to express formally how to emulate a shared-memory. This set of properties is called a consistency criterion. During the last decades, DSM for failure-prone static systems have been studied. In such context, a bounded number of entities may crash. Recently, DSM have been studied in a dynamic model where an unbounded amount of failures may occur. Now, we strongly believe that DSM for large-scale dynamic systems is of crucial interest for communication in distributed systems.

Thesis Content

This thesis investigates emulations of shared-memory for loosely-coupled distributed systems. To this end, several distributed shared memories (DSMs) are described. Each of these DSMs provides transparently the clients with read and write operations translating into emulating a shared-memory while operations are specified in the message-passing model. In shared-memory, a write operation consists in modifying the value of an object by writing it into the memory while a read operation consists in returning the value of an object stored in the memory. We focus on DSMs defined over quorum systems.

The first major issue addressed in this thesis is dynamism. Dynamism induces a potentially unbounded amount of failures and makes unusable the solutions suited for static systems, even with a bounded amount of failures. Consequently, recoveries that occur frequently must be integrated into the DSM to cope with continuous failures. Otherwise after some time, either the memory becomes inconsistent, operations are no longer ensured to terminate, or objects may disappear from the system. Among those repeated recoveries is the reconfiguration mechanism. Chapter 2 presents DSMs that tolerate dynamism. First, it presents a general DSM that can be used in static system. Second, it presents several problems that are related to dynamism and the solutions provided by the literature. Third, it presents a DSM whose reconfiguration relies on a consensus algorithm. This consensus is made fast so that the DSM tolerates a larger rate of failures. Moreover it is independent of operations so that operation termination does not depend on reconfiguration termination.

The second major issue addressed in this thesis is scalability. First, large-scale systems can not afford a global reconfiguration. That is the reconfiguration cost must be minimized for the system to scale well. Second, large-scale systems may experience a high variation in the number of read and write requests: some object may interest a large part of the system at some time while its interest may decrease suddenly, later on. This variation introduces a problem of load management that may induce request loss and non-termination of some operations. Chapter 3 compares different structure of quorum systems that appeared so far. It also points out that communication among quorum members impacts on the overall performance of the DSM. Furthermore, it presents several

dynamic quorum systems especially suited for dynamism. Lastly, it presents a DSM that scales well and self-adapts in face of dynamic events and load variation.

Chapters 2 and 3 remedy the problem of dynamism and scalability, respectively. Practically speaking, these solutions may experience limitations to face both dynamism and scalability. Chapter 4 relaxes the consistency requirement used in previous chapters to achieve better efficiency. That is, it gives the definition of probabilistic quorums systems. Existing solutions are given and a new definition of quorum system is proposed to tolerate dynamism: Timed Quorum Systems (TQS). TQS relies on timely and highly probabilistic requirements that makes it very efficient despite dynamism. For the sake of scalability, the TQS implementation proposed does not rely on any structure, thus avoiding reconfiguration while replicating during a read or a write operation execution. Finally, for practical needs we give valuable measures of the guarantee achieved by this solution. These measures boil down to promising results for other quorum-based applications in large-scale dynamic systems.

Roadmap of the thesis. Chapters 2 and 3 of this thesis guarantee deterministic consistency while Chapter 4 guarantees probabilistic consistency. Lastly, this thesis argues that probabilistic solutions to the large-scale dynamic DSM promise competitive result at the cost of weak relaxations. The results, obtained for loosely-coupled distributed systems, are confirmed by case studies on peer-to-peer systems, yet we claim that most of the results apply to other large-scale dynamic systems.

This thesis investigates the distributed shared memory problem in dynamic and large-scale system. Chapter 2 explains how to implement distributed read and write operations using quorums and introduces reconfiguration of quorum systems as a key requirement for coping with dynamism. Chapter 3 proposes an alternative reconfiguration method that minimizes the amount of information each node has to maintain in order to achieve scalability. Chapter 4 defines timed quorum systems that intersect each other during their bounded lifetime with high probability and that do not rely on any structural requirements, avoiding reconfiguration to achieve better performance.

Finally, the Appendix A proposes a solution to an interesting problem of large-scale and dynamic systems that is discussed in the Section 2.4. Moreover the detailed Input/Output Automata [Lyn96] specification of two algorithms proposed in Chapters 2 and 3 are given in Appendices B and C.

Chapter 1

Preliminaries

This Chapter presents some definitions and the general model that is used in this thesis. Section 1.1 introduces the vocabulary and definitions related to quorum systems. Section 1.2 introduces the distinct models of shared memory while Section 1.3 gives the general model used in this thesis.

1.1 Quorums at the Heart of Consistency

Quorums are at the heart of the emulation of distributed shared memory proposed in this document. Here, we define the vocabulary related to quorum systems that is used along this thesis.

Weighted voting. Systems experiencing isolated failures require replication of the object at distant locations with independent failure probability. If distinct nodes are allowed to modify the same object, then a synchronization process must be used to ensure consistency when concurrency occurs. A mechanism that provides nodes with an indication on whether consistency might suffer from their modification have been defined by Gifford [Gif79] and Thomas [Tho79] the same year of 1979. In these mechanisms, before executing an operation, a node asks for some permission of other nodes, typically the ones owning a copy of the object. A permission is granted depending on the set of nodes that have responded and on the answer of each responder: one granted permission prevents another permission from being granted.

Gifford considers a system of voters where a global weight W is shared among all voters so that their vote has a corresponding weight in granting the permission. The permission is granted for a read (resp. write) operation if the sum of weights of the received votes is r (resp. w), such that $r+w>W$. Thomas assumes a distributed database where the same copy of the data is replicated at distant locations. Multiple nodes can run an operation on this database by sending a corresponding message to one database copy. Then this copy tries to gather the permission of a majority of databases to execute the operation.

Generalization of quorum systems. Despite the appealing aspect of weighted voting and majorities as intuitive paradigms, they are not the ultimate solution to consistency in distributed system. Indeed, a more general approach exists as shown by Garcia-Molina and Barbara [GMB85].

Definition 1.1.1 (Set System) *A* set system $\mathcal{S}$ *over a universe* U *is a set of subsets of* U.

Originally, a *quorum system* over a universe U is defined simply as a set of subsets of U so that any couple of sets have a nonempty intersection. A *coterie* $\mathcal{C}$ is a quorum system whose quorums satisfy *minimality*: for any $Q_1, Q_2 \in \mathcal{C}$, we have $Q_1 \not\subset Q_2$. In this sense, quorum systems generalize coteries by relaxing minimality.

Definition 1.1.2 (Quorum System) *A* quorum system $\mathcal{Q}$ *over a universe* U *is a set system over* U *such that for any* $Q_1, Q_2 \in \mathcal{Q}$*, the intersection property* $Q_1 \cap Q_2 \neq \emptyset$ *holds.*

A *bicoterie* [MMR92] under U has been defined as two set systems $B = \langle \mathcal{S}_1, \mathcal{S}_2 \rangle$, each satisfying minimality and such that for any $S_1 \in \mathcal{S}_1$, and any $S_2 \in \mathcal{S}_2$, we have $S_1 \cap S_2 \neq \emptyset$. Building upon the bicoterie definition, we define naturally a *biquorum system* as a bicoterie whose set systems do not verify minimality. In other words, a biquorum system is a couple of set systems $\langle \mathcal{S}_1, \mathcal{S}_2 \rangle$, such that for any $S_1 \in \mathcal{S}_1$ and any $S_2 \in \mathcal{S}_2$, we have $S_1 \cap S_2 \neq \emptyset$.

Definition 1.1.3 (Biquorum System) *A* biquorum system $\mathcal{Q}$ *over a universe* U *is a couple of set systems* $\langle \mathcal{Q}_1, \mathcal{Q}_2 \rangle$ *over* U *such that for any* $Q_1 \in \mathcal{Q}_1$*, and for any* $Q_2 \in \mathcal{Q}_2$*, the intersection property,* $Q_1 \cap Q_2 \neq \emptyset$*, holds.*

Biquorum systems have been already used in the literature. First, Gifford [Gif79] defines read and write quorums of nodes, where each node is assigned a specific number of votes. The sum of votes in any read quorum and in any write quorum must be larger than the total number of votes, which translates into the intersection between any read and any write quorum. Second, Herlihy [Her86] uses read and write quorums for objects whose specification satisfies FIFO—First in, First Out—policy. This policy provides an object with enqueue and dequeue primitives such that enqueue stores a value and dequeue returns the value that has not been dequeued yet and that has been enqueued first. Finally, an informal definition of biquorum system appeared in [LWV03].

Here, we consider dynamic systems, that is, we say that an intersection is non-empty if and only if it contains at least one active node that has not failed or left the system. By abuse of notation, we say that two sets (or more particularly two quorums) intersect if and only if they have a non-empty intersection.

1.2 Shared Memory Models

Shared-memory is used for communication among participants of a distributed system. As said previously, it appears to be one of the two major communication paradigms for distributed systems,

the other being the message-passing model. Shared memory simplifies algorithm formalization as well as code programming. The shared memory is accessed by some nodes through read and write operations. The read operation simply returns to the requester a piece of information contained in the memory without modifying it. The write operation modifies the information stored in the shared memory. Distinct shared memories exist, each shared memory depending on the number of nodes allowed to read it and the number of nodes allowed to write it. First observe that a model in which a single node is allowed to read the memory and write it, is equivalent to a single node with exclusive access to its memory. In contrast, we are interested by the ability for nodes to share a memory.

- **MWSR:** In the multi-writer/single-reader (MWSR) model, multiple nodes can execute write operations whereas only one node can execute read operations.
- **SWMR:** In the single-writer/multi-reader (SWMR) model, multiple nodes can execute read operations whereas only a single node can execute write operations.
- **MWMR:** In the multi-writer/multi-reader (MWMR) model, multiple nodes can execute the write operations and the read operations.

It is noteworthy that the models presented above do not differentiate the number of operations executed simultaneously. Actually, ensuring consistency while many operations are concurrent can be reduced to solving concurrency when only two operations are concurrent: hardware synchronization might be required at some nodes in both cases. However, complexity relies tightly on the type of concurrent operations: if two write operations are concurrent, the result of one operation overwrites the result of the other, thus, those operations must be ordered; if two read operations occur concurrently, then both may return the same value, thus, there is no need to order them.

In the remaining of this document, we focus on the strongest of these models, the MWMR model. We also consider a single object to which operations apply. The memory is constructed by means of the composition of multiple objects in a straightforward manner.

1.3 General System Model

The system consists of a distributed set of interconnected nodes. We assume that a node simply needs to know the identifier of another node to communicate with it and that each node knows the identifier of a subset of nodes that are its *neighbors* in the communication graph. Every node has a unique identifier, and the set of identifiers is a totally ordered set, denoted I. [1] The communication is asynchronous in the sense that messages can be arbitrarily delayed. Communication is unreliable in that message can be reordered and even lost, however, if a message is delivered, then it has been sent and not altered; and no messages are duplicated. The system is dynamic, meaning that nodes

[1] Thereafter, the set of identifiers I is assumed to be a subset of $\mathbb{N}$.

that are already in the system may leave at any time while new node can join the system at any time. A node failure is considered as a leave and a recovery is considered as a new join event. This means that when a node rejoins the system, it obtains a new identifier and loses all its previous knowledge of the system. We refer to a node that belongs to the system as an *active* node, while a node that has left or failed is referred to as a *failed* node.

Any object in the system is accessed through read and write operations initiated by any node. When a node initiates an operation we refer to it as a *client*. A read operation aims at returning the current value of the object while a write aims at modifying the current value of the object. Object values are replicated at a set of nodes we refer to as *servers*. Observe that any node can be client and server. Each value v has an associated tag t. The tag has a counter that indicates the version of the associated value and a node identifier that is used as a tie-breaker: tag t is lower than tag t' if the counter of t is lower than the counter of t' or if the counters are the same whereas the identifier of t is lower than the identifier of t'. We consider local consistency conditions, and thus, only a single object so that the memory is obtained by composition of multiple object.

Chapter 2

Facing Dynamism: Reconfigurable Quorum System

The availability of the Distributed Shared Memories presented here depends on the availability of a quorum system. Since failure may accumulate in our dynamic model, the quorum system must readapt to face long-term or permanent failures of its quorum members. In the following, we refer to a quorum system as a *configuration*. The mechanism needed to replace the nodes of a quorum system to cope with dynamism is the *reconfiguration*. The following chapter focuses essentially on a single kind of reconfiguration that replaces the whole quorum system by another one.

Reconfiguring a distributed shared memory without altering ongoing operations is a difficult task. Reconfiguration must not violate consistency and must allow operation termination. First, the new configuration must be uniquely chosen by all participants. Second, the object states must be conveyed from the previous configurations to the new one before any of those previous configurations could be removed. Finally, to guarantee that the operations eventually terminate, operations must execute independently from the reconfiguration. Otherwise, a delayed reconfiguration may block some operations.

This chapter presents an emulation of shared memory in message-passing model when the environment is dynamic. To allow dynamism the quorum system at the heart of our emulation is regularly reconfigured. The reconfiguration mechanism investigated in this chapter is the quorum system replacement, i.e., here, reconfiguration aims at replacing the current quorum system by a new one.

Guaranteeing consistency. Originally, consistency aims at providing guidelines for shared memory to emulate as accurately as possible the behavior of a single memory. A consistency criterion can be seen as a contract between an application and a memory: provided that some properties are verified by the application, the memory ensures desirable guarantees. These guarantees must always hold and defined the safety properties of our system. As opposed to *liveness* that ensures that eventually something good happens, *safety* guarantees that nothing bad can happens.

Here, something bad is for the consistency contract not be respected.

Various consistency criteria have been defined in the literature. A strength scale is generally used to compare those criteria with each other and *atomicity* appears to be the strongest. Atomicity [Lam86, Lyn96] allows clients to access data (object) through read and write operations such that operations are ordered in a sequential manner satisfying the specification of the object and the real-time precedence. *Linearizability* is equivalent to atomicity and has been defined for variously typed objects [HW90]. Another widely investigated consistency criterion, *sequentiality*, impose that operations ordering respects only the local operation ordering of each node. Atomicity is stronger than sequentiality because any atomic execution satisfies also sequentiality while the reverse is not true.

Why choosing atomicity? Despite the appealing realistic emulation of shared memory provided by atomic DSM, implementing atomicity in asynchronous settings remains a topical issue. As a consequence, weaker consistency criteria have been proposed: weak-atomicity [Vid96], causality [Lam78, HA90], PRAM [LS88], processor consistency [Goo89], and others. The drawback of such consistency criteria is to allow multiple different views of the system at any locations, violating *one-copy serializability* [BG83]. Some consistency criteria sharing performance of strong and weak consistency appeared: mixed consistency [ACLS94] and hybrid consistency [AF92], using weak ordering [DSB86].

Locality characteristic has been defined by Herlihy and Wing as the ability for a consistency criterion to be preserved under object composition. That is, if the executions restricted to each object verify independently the *local* criterion then the whole execution applied to the composition of these objects verifies the same criterion. The authors showed in [HW90] that, knowing if a given criterion is local is not a trivial problem. To illustrate this idea they showed that atomicity is local while sequentiality is not. The compositionality caused by the locality property has been efficiently used in the design of numerous proofs of atomicity [LS02, DGL+05, CGG+05, CLMT05]. In the following, we focus on the atomicity as the consistency criteria that must satisfy the distributed shared memories we propose.

Efficiency. A major aim is to provide efficient operations in distributed shared memory. It is well-known that most operations applied to a file system are read operations [BHK+91], hence minimizing read latency improves on average operations efficiency. For this purpose, Dutta et al. [DGLC04] proposed to upper bound the latency of read operations satisfying atomic consistency, namely *fast reads*. A recent achievement has also proposed a *semi-fast* atomic read operations [GNS06] requiring from one to two gossip rounds.

2.1 All-to-all Operation and Reconfiguration

This Section outlines the problem of emulating a DSM using quorum-based operations. First, it formally defines atomicity for a better understanding of the problem of implementing atomic operations. Second, it indicates simply what is a quorum-based atomic operations. All operations presented in this document rely on similar quorum-based phases. Third, we emphasize the various problems related to reconfiguring an atomic memory that consists in successively replacing the memory participants by new ones. Finally, it emphasizes the importance of operation and reconfiguration independence for reconfigurable atomic memory.

Atomicity. Atomicity is the property of an object x if in any system execution, the operations applied to x are partially ordered using relation $\prec$ such that the conjunction of the following properties holds:

1. **Serial Specification Requirement:** if a read operation occurs on object x then it returns the last value v, with respect to $\prec$, written on object x; or the default value if no values have been written so far.
2. **Real-Time Requirement:** if an operation π_1 ends before an operation π_2 starts then $\pi_2 \not\prec \pi_1$.
3. **Serialization Requirement:** for any operation it exists a unique serialization point and the serialization point ordering of the operations satisfies the ordering of those operations. (Serialization point order is a total order while the operation order is a partial order.)

As indicated by Property 1, the ordering of operations on object x determines the value that is read or the value that is written at any point in time on object x. This states that an object respects its specification returning its single current value. Property 2 ensures that this ordering cannot contradict the real-time precedence. Hence, no stale values can be returned since staleness is especially expressed by real-time precedence. Finally, Property 3 implies that write operation are totally ordered while read operations are ordered with respect to the write operations. That is, even though two write operations are concurrent, only the effect of one of the two will be noticeable after both operations finish.

Concurrency problem. Properties 1 and 2 are of special interest. On the one hand, assume that two write operations run concurrently. Since they all have a distinct serialization point (Property 3), they cannot be executed partially but only one can determine the up-to-date value. On the other hand, assume that a write operation w is concurrent with two read operations, r_1 and r_2, while none of these read operations is concurrent with the other read operation. That is, if the serialization point of the write operation precedes the serialization point of the first read operation $w \prec r_1$, then $w \prec r_2$ by the transitivity of $\prec$. Similarly, if $r_2 \prec w$, then $r_1 \prec w$ too. However, atomicity is violated if the operation r_1 returns the value written by w while r_2 returns a value that has been

overwritten by operation *w*. This problem of reading a value while a more up-to-date has already been read, called the new/old inversion problem, has been firstly outlined by Lamport [Lam86], and is presented in Figure 2.1.

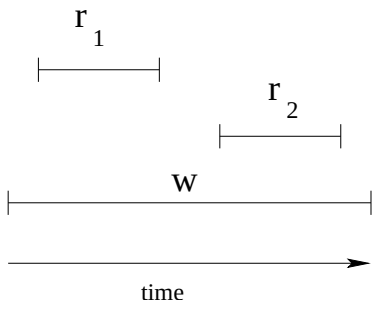


Figure 2.1: The new/old inversion problem occurs if read operation r_1 returns the value written by write operation *w* while read r_2 returns a value that has been written before *w* occurs.

To solve this problem, read operations must ensure that further read operations never return less up-to-date value. In some sense, it is similar to the fact stated in Property 1 where a write must ensure that no less up-to-date value will ever be read. This is the reason why in emulation of atomic registers in message passing system, "read must write". The read-must-write problem of distributed atomic memory is well-known. In [AW98], a theorem states that *In any wait-free simulation of a single-writer single-reader register, at least one read must write*, and in [Lam86] it is said that *there exists no algorithm to implement an atomic register using a finite number of regular registers that can be written only by the writer (of the atomic register)*.

The three properties presented above lead to the atomic object defined in Theorem 13.16 of [Lyn96]. The first point of the original Theorem is deduced from the other, hence, it is omitted here.

Definition 2.1.1 (Atomic Object) *Let x be a read/write atomic object. Let H be a complete sequence of invocations responses of read and write operations applied to object x. The sequence H satisfies atomicity if and only if there is a partial ordering $\prec$ on the operations such that the following properties hold:*

1. *if the response event of operation π_1 precedes the invocation event of operation π_2, then it is not possible to have $\pi_2 \prec \pi_1$;*
2. *if π_1 is a write operation and π_2 is any operation, then either $\pi_2 \prec \pi_1$ or $\pi_1 \prec \pi_2$;*
3. *the value returned by a read operation π_2 is the value written by the last preceding write operation π_1 regarding to $\prec$ (in case no such write operation exists, this value returned is the default value).*

Domain	Description
$I \subset \mathbb{N}$	the set of node identifiers
V	the set of all possible object values
$T \in \mathbb{N} \times I$	the set of all possible tags

Table 2.1: Algorithm Domain

2.1.1 Quorum-based Read/Write Operations

In the following, read and write operations are presented at a high level. As a first attempt, we propose a simple atomic memory that does not tolerate dynamism. The memory is specified at a high level for the sake of simplicity. The major goal is, here, to explain how an operation can use quorum round trips to ensure consistency. Then, we describe informally faster operations for static settings.

Generic atomic object implementation. For the sake of genericness, this algorithm lacks from explicit specification of communication procedure (send and recv) so as the termination identification test (is-quorum-contacted). For the sake of generality, these procedures that rely on a lower level of specification (depending on the type of solution that aimed to be provided) are omitted here. For example, a client *i* might know a complete quorum and wait until all quorum members respond before deciding to end the procedure **Propagate** and **Consult**. Conversely, node *i* might wait for the message of the last quorum member without knowing every element.

Algorithm 1 implements an atomic object which supports read/write operations performed by multiple readers and multiple writers. the variable domain used in this algorithm is presented in Table 2.1. The pseudocode is high level and describes a generic solution using quorum access. This algorithm has been inspired by the work of Attiya, Bar-Noy, and Dolev [ABND95], and Lynch and Shvartsman [LS02].

A client executes an operation by running the **Read** or **Write** procedure. An object value *v* and a value version (or *tag*) *t* are maintained at each server. This tag is incremented each time a new value is written. The node id of the writer is added to the tag as a lower weight integer to ensure uniqueness of tag.

Read and **Write** procedures are similarly divided in two phases: the first phase **Consults** the value and the associated tag of an object by querying a quorum. The second phase **Propagates** the up-to-date value *vnew* and tag *tnew* to a whole quorum. When the consultation ends, the client gets the lastly written value back (with its tag). In case of a write operation, the client increments the tag and propagates this new tag with the new value it wants to write. In case of read operation, the client simply propagates the up-to-date value (with its tag) it has just consulted.

The sequence number *s* serves as a phase counter to encompass asynchrony and thus preventing that a quorum takes in account a stale message. More precisely, when **Consult** or **Propagate** starts, the sequence number is incremented indicating a new phase. When a node receives a message with

Algorithm 1 Generic Atomic Object Algorithm

1: **State of i:**
2: $Q_1, ..., Q_k \subset I$ the quorums
3: $\mathcal{Q} = \{Q_1, ..., Q_k\}$, the quorum system
4: $\mathcal{M}$, a message containing:
5: $type \in \{\mathsf{GET}, \mathsf{SET}, \mathsf{ACK}\}$, the message type,
6: $\langle val, tag \rangle \in V \times T \cup \{\perp\}$, the value and tag,
7: $seq \in \mathbb{N}$, the sequence number of the message.
8: $v \in V, v = v_0$, the object value
9: $t \in T, t = \langle 0, i \rangle$, the tag used containing:
10: $compteur \in \mathbb{N}$, the write operations counter
11: $id \in I$, the identifier of the writer
12: $s \in \mathbb{N}, s = 0$, the current sequence number
13: $tmax \in T$, the highest tag encountered
14: $vlast \in V$, the most up-to-date encountered value
15: $tnew \in T$, the new tag to write
16: $vnew \in V$, the new value to write

17: **Read()$_i$:**
18: $\langle v, t \rangle \leftarrow$ **Consult**()$_i$
19: **Propagate**($\langle v, t \rangle$)$_i$
20: **Return** $\langle v, t \rangle$

21: **Write($vnew$)$_i$:**
22: $\langle v, t \rangle \leftarrow$ **Consult**()$_i$
23: $tnew \leftarrow \langle t.compteur + 1, i \rangle$
24: **Propagate**($\langle vnew, tnew \rangle$)$_i$

sequence number s querying it to participate, it **Participates** by sending the same sequence number. That is, the phase a message belongs to is clearly identified, and might be ignored.

Fast quorum-based atomic operations. In [DGLC04], Dutta et al. investigate how fast can a distributed atomic read be where the number of failures is upper bounded. This result applies to failure-prone static system but does not consider accumulating failures.

They define a fast implementation of a SWMR atomic memory as an implementation in which any read operation of all possible executions ends after a single round trip. They prove that no such implementations are possible if there are more than $S/t - 2$ readers executing operations in a row, where S is the total number of servers and t is an upper bound on the number of failures.

```
25: Consult()_i:
26:     Let Q be a quorum of 𝒬
27:     s ← s + 1
28:     send(GET, ⊥, s) to nodes of Q
29:     Repeat:
30:         if recv(ACK, ⟨v,t⟩, s) from j then
31:             rcvd-from ← rcvd-from ∪ {j}
32:             if t > tmax then
33:                 tmax ← t
34:                 vlast ← v
35:     Until is-quorum-contacted(rcvd-from)
36:     rcvd-from ← ∅
37:     Return ⟨vlast, tmax⟩

38: Propagate(⟨v,t⟩)_i:
39:     Let Q be a quorum of 𝒬
40:     s ← s + 1
41:     send(SET, ⟨v,t⟩, s) to nodes of Q
42:     Repeat:
43:         if recv(ACK, ⊥, s) from j then
44:             rcvd-from ← rcvd-from ∪ {j}
45:     Until is-quorum-contacted(rcvd-from)
46:     rcvd-from ← ∅

47: Participate()_i (activated upon reception of 𝓜):
48:     if recv(𝓜) from node j then
49:         if 𝓜.type = GET then
50:             send(ACK, ⟨v,t⟩, 𝓜.seq) to j
51:         if 𝓜.type = SET then
52:             if 𝓜.tag > t then
53:                 ⟨v,t⟩ = 𝓜.⟨val, tag⟩
54:             send(ACK, ⊥, 𝓜.seq) to j
```

More precisely, they show that it is possible only if there is less than $S/t-2$ readers and $t < S/2$. They also show that it exists a fast implementation when this does not hold (i.e., there are less subsequent and distant reads than $S/t-2$).

Their implementation proposes one phase read operation. Write operations contact as many

servers as possible such that it remains wait-free, i.e. $S' = S - t$. To convey information from a write operation to a read operation, thus ensuring ordering of read operations with respect to write operations (cf. Property 3 of Definition 2.1.1), read operations contact as many servers that can testify of the previous write operation, as possible. By assumption, the number of faulty servers is less than or equal to t, thus the number of non-faulty servers that have been contacted by the previous write operation is $S'' = S - 2t$. To prevent new/old inversion from happening, i.e., that two read operations violate Definition 2.1.1, each read operation contacts as many servers whose value has been read by the previous operation, as possible. This number is $S''' = S - 3t$. To generalize, after a new value has been written on $S - t$ servers, the $i - 1^{st}$ following read operation must contact $S'' = S - it$ servers. Since a client ignores the index of its read operation in the sequence of read operations following the last write, each server counts the times its value has been read and send this counter to the client when accessed.

Because the number of servers distinct readers must access decreases while no write occurs, at some point the number becomes 0, preventing the read operations to return the last written value. Thus, the number of distinct readers is bounded.

2.1.2 Quorum System Reconfiguration

The type of reconfiguration we consider aims at moving an object from one collection of sites to another collection of sites. Such a mechanism is motivated in dynamic systems by the malfunctioning of some sites or in large-scale environments by adaptation to load variations.

Reconfiguration evolves the quorum configurations. This mechanism serves two purposes. First, it aims at installing a new quorum configuration, informing participants of the new configuration that should be used. Second, this mechanism is in charge of removing an obsolete configuration. The key challenges are thus to inform nodes of the new configuration before they fail and to remove the previous configuration while it is safe to do so.

This reconfiguration process replicates the object value from an initial collection of sites to a final collection of sites and multiple executions of such mechanism may follow. That is, a reconfigurable object must be implemented using two data types:

1. **Configuration sequence:** the configuration sequence is a record containing some configurations. This record is used to keep track of some configurations that have been used and indicates which configuration is the current one.

2. **Object state:** the object replica records the value of the object and the associated tag, the version of this value. During a reconfiguration execution, object state must be conveyed from the initial configuration to the final one, thus mapping the current configuration with the up-to-date value of the object.

The configuration sequence keeps track of the evolution of the system. At the beginning the system uses a default configuration and this configuration is stored as the first index in the configuration sequence. At the end of the first reconfiguration execution, a second configuration is

introduced in the sequence and the system uses this configuration until another reconfiguration instance installs a new configuration in the sequence, and so on.

The system must use the right configuration. At any time, the current configuration is the configuration with the largest index in the configuration sequence. In the current configuration, the up-to-date object state must be available. To ensure that the system uses the up-to-date object state the reconfiguration mechanism has to ensure two properties:

1. **Configuration consistency:** the current configuration owns the up-to-date object replica.

2. **Configuration currency:** the system uses the current configuration.

For the sake of configuration consistency, the reconfiguration replicates the up-to-date object replica from the initial configuration to the final one. Consequently, by iteration on the length of the configuration sequence, if the first current configuration owned the up-to-date object replica, then the configuration consistency holds.

For the configuration currency, the reconfiguration has to inform every participant that a new configuration is installed and that this new configuration is the current one. For instance, all participants may maintain its own configuration sequence such that any new configuration is mapped to the largest index. Observe that a responsible node *i* might take the responsibility of archiving the set of all configurations but this would require all participants to access node *i* and the system would suffer from single-point of failure and congestion.

Ensuring consistency and currency of configurations. An interesting measure is the time a new configuration needs to become current. Indeed, this duration defines the minimal uptime of node maintaining the object state. As soon as all nodes maintaining the object state fail, the object state is lost. Since the last configuration cannot be added to all configuration sequences simultaneously, when should a configuration be considered as current? This question arises the problem that from two distinct standpoints there may coexist two distinct configurations.

In [Her86] Herlihy proposes a four step reconfiguration mechanism. These step are as follows:

- **Gather the current object state:** this step aims at contacting the initial configuration to ensure that the most up-to-date object state is obtained.

- **Storing the object state:** this step aims at contacting a new quorum to store the up-to-date object state safely.

- **Initialize a new configuration:** this step aims at initializing the new configuration members by informing that they are part of the new configuration.

- **Update the configuration:** This step aims at notifying the participants that a new configuration is installed, that is, the old configuration that are no longer current can be discarded.

In these steps, an object state is conveyed to reliable storage and the current configuration changes from the used one to a new one. In the following, we only consider a single instance of reconfiguration, thus we refer to these configurations as, respectively, the *initial configuration* and the *final configuration*.

In this mechanism, operations cannot occur during the reconfiguration step without potentially deprecating the object state present in the current configuration. Even though the up-to-date object state is gathered at the first step, by the time the initial configuration is discarded, the object state might be updated so that the final configuration maintains a non up-to-date object state.

A widely adopted solution aims at locking operations during the time the system reconfigures. That is, the replica mapped to the final configuration cannot be staled by a concurrent operation. Virtually synchronous services [BJ87, Fri95], and group communication services in general [1996] can be used to implement a totally ordered broadcast. Totally ordered broadcast is a useful abstraction for consistency and currency maintenance. First, the up-to-date object state can be broadcast to the new set of participants, called a *view*, for consistency, then a message can trig the use of this new view for currency purpose. In general, group communication method requires that view-formation is executed while operations stop, so that no object modification is received in between affecting the object state at a stale configuration. That is, operations are delayed at least the time the view-formation occurs.

In [MA04], the authors propose a storage supporting reconfiguration to tolerate a large spectrum of faults, including malicious ones. Their approach proposes a reconfiguration mechanism to overcome the potentially accumulating failures that may occur, and when such a mechanism occurs the pending operations are stopped. At the time the reconfiguration ends, the previously pending operations resume from the execution point they were stopped. Similarly to group communication services, [MA04] aims at replacing views. An initial view becomes inactive before a final view becomes active. During the time none of these views are active, clients cannot gather responses from a quorum, that is, operations cannot be performed. This period must be sufficiently long for the value of the initial view to be conveyed to the final one. When this is done, the final view may become active. This process makes the operation execution depend on the view replacement (i.e. reconfiguration) process. Therefore, reconfiguration performance impacts directly on operation performance. These kind of approaches trades the independence of operations against consistency. In contrast, the following algorithms make independence and consistency cohabit.

2.1.3 Independence of Read/Write Operations

To ensure independence of operations the key idea is to notify the system about the time to upgrade its configuration. As explained previously, installing a new configuration is not sufficient to ensure that the system will access the up-to-date object state. In other words, the four steps are not sufficient to discard previous reconfiguration if operations and reconfiguration can occur concurrently.

Conversely, to allow operations independence so that they can progress even if reconfiguration

is pending, the system needs to use distinct configurations at the same time during a transition period. This period ensures that the result of all previous operations have been conveyed to the new configuration object state and that all new operations will be applied to the newly installed configuration.

In RAMBO [LS02], Lynch and Shvartsman present independent operations that may run concurrently to reconfiguration. In this case, the reconfiguration may delay some operations but operation progress does not depend on reconfiguration progress. As explained previously, the reconfiguration does not install the new configuration while discarding the old ones. Instead the reconfiguration is divided into two major phases:

1. **Configuration installation:** the reconfiguration installs a new configuration. From this point on, the system may use multiple configuration at the same time: the initial configuration that belongs to the configuration sequence since the beginning of the reconfiguration execution and the final configuration that is freshly installed.

2. **Configuration upgrade:** the reconfiguration ensures that enough nodes are aware of the current configuration, so that any operation will use it before ending. This allows participants to discard the other configuration(s) that have thus become obsolete.

Single reconfigurer. Build on the robust shared-memory emulation of [ABND95], Lynch and Shvartsman proposed an extension using dynamic quorums [LS97] to provide multi-writer/multi-reader MWMR shared memory emulation. This emulation is specified into independent Input/Output Automata [Lyn96], the reader/writer and the reconfigurer. The reader/writer provides the client with read and write primitives to access the emulated shared memory. The reconfigurer, as in [ES00], is executed at some node in the network to replace a quorum configuration by another. Thus it may suffer from some single point of failure.

2.2 Decentralized Reconfiguration

Later on, Reconfigurable Atomic Memory for Basic Objects (RAMBO [LS02]) appeared. RAMBO emulates a MWMR shared memory in dynamic distributed systems. It provides a quorum-based reconfiguration mechanism that can be triggered by any participant. The participants keep sending messages among each other, we call this action gossiping among nodes. An improvement of RAMBO appeared in [GMS05] where the number of messages is importantly reduced by determining a subset of nodes that are allowed to communicate. While the result of [GMS05] still satisfies safety, it may delay, in some cases, the execution of RAMBO.

The RAMBO service is divided in several automata. The Reader/Writer automaton executes the operations when requested by the clients. This operation are divided into two phases: the first phase consults the up-to-date object state by gathering the tags and value of all the nodes of at least one quorum. Then, the second phase propagates the tags and values of the object. In comparison with

previous approaches, RAMBO uses biquorum system as defined in Definition 1.1.3. for operation and reconfiguration. We refer to the first type as *consultation quorums* and to the second type as *propagation quorums*.

The Reconfigurer automaton installs a final configuration using an external consensus algorithm. As a result multiple configurations can cohabit at the same time. For limiting the number of active configurations, the reconfigurer upgrades periodically to remove obsolete configurations from the configuration sequence of all participants. This upgrade is carried out locally by each node *i* in two cases. First, if node *i* discovers that some of the configurations recorded in its configuration sequence have already been removed from the configuration sequence of other participants. Second, node *i* might garbage collect the obsolete configuration itself. In this case, node *i* informs a consultation quorum and a propagation quorum of the obsolete configurations and gathers the up-to-date value and tag of the object. This ensures that any concurrent or later operation requesting an obsolete configuration will learn that a more up-to-date configuration should be used. Then, node *i* informs a propagation quorum of the new configuration about the up-to-date object value and tag. This ensures that later operations accessing the new configuration will have the up-to-date object value and tag. The upgrade mechanism must be executed once for each of the configurations to remove.

2.2.1 Coupling Installation and Upgrade of Configurations

A first improvement on the RAMBO algorithm aimed at speeding up the upgrade process and minimizing the configuration sequence [GLS03]. This upgrade improvement led to a new algorithm called RAMBO II. The RAMBO II algorithm upgrades many configurations in the meantime. If multiple configurations are present in the system, then the algorithm upgrades all configurations in a row. That is, after a single configuration upgrade mechanism, RAMBO II ensures that all but the current configuration can be removed. The two phases of the upgrade process are depicted on Figure 2.2. Each phase corresponds to the delay of a one message exchange. In phase 1, node *i* contacts a consultation quorum and a propagation quorum of each obsolete configurations. By doing so, node *i* tells all obsolete configuration members about the new configuration and a consultation quorum of each obsolete configurations tells *i* about the up-to-date object state. In phase 2, node *i* propagates the up-to-date object state to a consultation quorum and a propagation quorum of the new configuration.

Issue related to decentralization. Decentralized reconfiguration execution presents some drawbacks. When reconfiguration is executed at a single node, then this node can decide the new configuration to install. However, when the reconfiguration is decentralized then a consensus must be reached to prevent distinct current configurations from cohabiting.

Upgrade Phase 1

- Node *i* consults the latest *object state* of any previous configuration.
- Node *i* propagates the information that there is a new current configuration to any previous configuration.

Upgrade Phase 2

- Node *i* propagates the latest *object state* to the new configuration.

Figure 2.2: The two phases of the upgrade process of RAMBO II [GLS03].

2.2.2 Deciding upon the Final Configuration using Paxos

In RAMBO, the reconfiguration assumed the presence of an external consensus algorithm to ensure that new configurations are installed one after the other such that there is a totally ordered sequence of configurations in time in the system. The Recon automaton of RAMBO responsible of the reconfiguration process is triggered from any nodes of the system. Then, Recon executes an instance of Paxos [Lam98]. Here we propose a reconfigurable distributed storage, called RDS, that integrates Paxos into RAMBO. RDS improves on RAMBO by coupling the configuration installation—consensus instance—with the configuration removal. This coupling speeds up the reconfiguration process for better fault-tolerance. To strengthen this result, RDS specifies a fast version of Paxos. RDS is detailed in Section 2.3.

The Paxos algorithm is a famous asynchronous consensus algorithm that achieves optimal delay in practical circumstances.[1] Paxos uses quorums to ensure agreement among participants. In recent years, Paxos experienced an important research interest among the distributed system community. Particularly, Paxos has been the subject of at least eleven papers that appeared over the last seven years.[2]

The key idea of Paxos is as follows. The participants of Paxos share three roles: *proposers* that propose values, *acceptors* that choose a value, and *learners* that learn the chosen value. A single node can play multiple roles. One Paxos instance is divided into ballots each representing a new attempt to reach consensus. (Ballots can occur concurrently, are not necessarily ordered, and may interfere with each other, delaying Paxos termination but not violating safety.) Paxos is a

[1] Traditional consensus algorithms save one message delay in comparison with Paxos by assuming that the set of participants that propose a value is the same as the set of participants that decide upon a value. In practice, it might not be the case.

[2] Results obtained from the DBLP (Digital Bibliography & Library Project [dbl]): eleven papers whose title explicitly mention the term Paxos have been published in a journal or a conference between 2000 and 2006. Three of them appeared in 2006.

Phase 1a. A coordinator sends a new ballot to a quorum of proposers.

Phase 1b. Upon reception of the message, any proposer responds by sending the value of the highest ballot (if any) they have already voted for. Each proposer that learns about a more up-to-date ballot abstained from any earlier ballot.

Phase 2a. After having received the response of at least a quorum of acceptors, the coordinator chooses a new value for its ballot and informs a quorum of acceptors of the value it has chosen for its ballot.

Phase 2b. Any acceptor, that learns about this value and that did not abstain from that ballot, may vote for it and tells the learners (and the coordinator) about this vote. When the learners (and the coordinator) hear that a quorum of acceptors has voted for it, they decide this value.

Figure 2.3: Informal description of Paxos

leader-based algorithm in the sense that a single *coordinator* is responsible of one ballot.

The Part-Time Parliament. Paxos has been firstly presented as a set of informal rules in a technical report of Lamport [Lam89], published nine years later in [Lam98]. This presentation describes the functioning of an ancient Part-Time Parliament of a Greek island named Paxos. More precisely, this paper explains how decrees can be passed while legislators may not be present in the Chamber at the same time and may experience difficulties to communicate. Interestingly, this part-time parliament boils down to an algorithm providing consistency despite any number of failures and resuming properly after more than a half of the nodes recover.

Figure 2.3 describes the normal sequence of message sent in a ballot of Paxos, where a message exchange is called a phase. Acceptors abstain from a ballot b if they discover another ballot b' with a larger identifier (**Phase 1b**). Acceptors may vote for a ballot if they did not abstain from it earlier. A ballot succeed only if a quorum of acceptors vote for its value, however, acceptors of a single quorum may vote for concurrent ballots. If this happens, a new ballot with a larger identifier must be started to try reaching consensus again. The consensus is reached when learners receive a message informing them of the chosen value; this reception (**Phase 2b**) terminates the ballot.

Fast Paxos. A fast consensus algorithm is an algorithm in which a node learns the chosen value two message delays of when the value has been proposed. Fast Paxos [Lam06b] is a fast consensus algorithm that is a variant of Paxos. That is, in Fast Paxos, a node may learn about the decided value at the latest 2δ after the value has been proposed, where δ is an upper bound on the transmission delay of messages. This result is achieved when no collisions occurs between concurrent distinct

ballots. A collision occurs when any quorum of acceptors vote for distinct values of concurrent ballots. Another consensus algorithm [BGMR01] can be easily modified using the same basic idea of Fast Paxos to achieve fast consensus.

The main improvement over classic Paxos is to avoid the two first message delays of Paxos in case no collision occurs. The first phase of the original algorithm aims at renewing the ballot number. This renewal might be avoided under specific circumstances:

- At the beginning of the first ballot execution: since no value has been proposed yet, the coordinator might first send message **2a** proposing any value.
- If the same coordinator runs multiple consensus instances: the coordinator who successfully led the ballot of the previous consensus instance, may reuse the same ballot number for the following consensus instance proposing directly any value in message **2a**.
- If the coordinator sends a message to acceptors for them to treat any proposer message like if it were a **2a** message. In this specific case an additional collision recovery must be implemented to ensure progress.

Avoiding first phase speeds up Paxos. Without this phase, the algorithm becomes a fast consensus algorithm, i.e., a value is chosen 2 message delays after it has been proposed. Figure 2.4 outlines message exchanges used in Classic Paxos and in Fast Paxos when no failures occur. In some cases, where Fast Paxos can avoid the first phase, Fast Paxos lasts 2 message delays less than Classic Paxos. The time taken to solve the consensus, i.e., the time taken for a proposed value to be accepted is 3 message delays long in Fast Paxos while it is 4 message delays long in Classic Paxos.

Next, we use Fast Paxos to decide upon a new configuration. That is, we propose a dynamic memory, RDS, that integrates Fast Paxos for the sake of reconfiguration installation.

Modification to fit reconfiguration needs. Our goal is to benefit from Fast Paxos to determine rapidly the next configuration to install. To this end, we integrate Fast Paxos into our configuration installation mechanism. Hence, each configuration installation has its own consensus instance that aims at deciding upon a unique value: the new configuration to install. Following Fast Paxos execution, configurations are proposed, voted, and one among those is decided. Then, we couple this installation mechanism with the configuration upgrade mechanism. The result is an all-in-one algorithm that provides a rapid reconfiguration: it parallelizes the installation and the upgrade to save time and strengthen fault-tolerance.

Coupling installation and upgrade relaxes a strong requirements for consistency: the activity of obsolete configurations. Recall first that the upgrade process, as investigated above, consults the object state, informs the previous configurations about the new configuration, and propagates the consulted object state to the new configuration before old configurations can be safely removed. That is, between the time of an installation and the time of an upgrade, at least one consultation

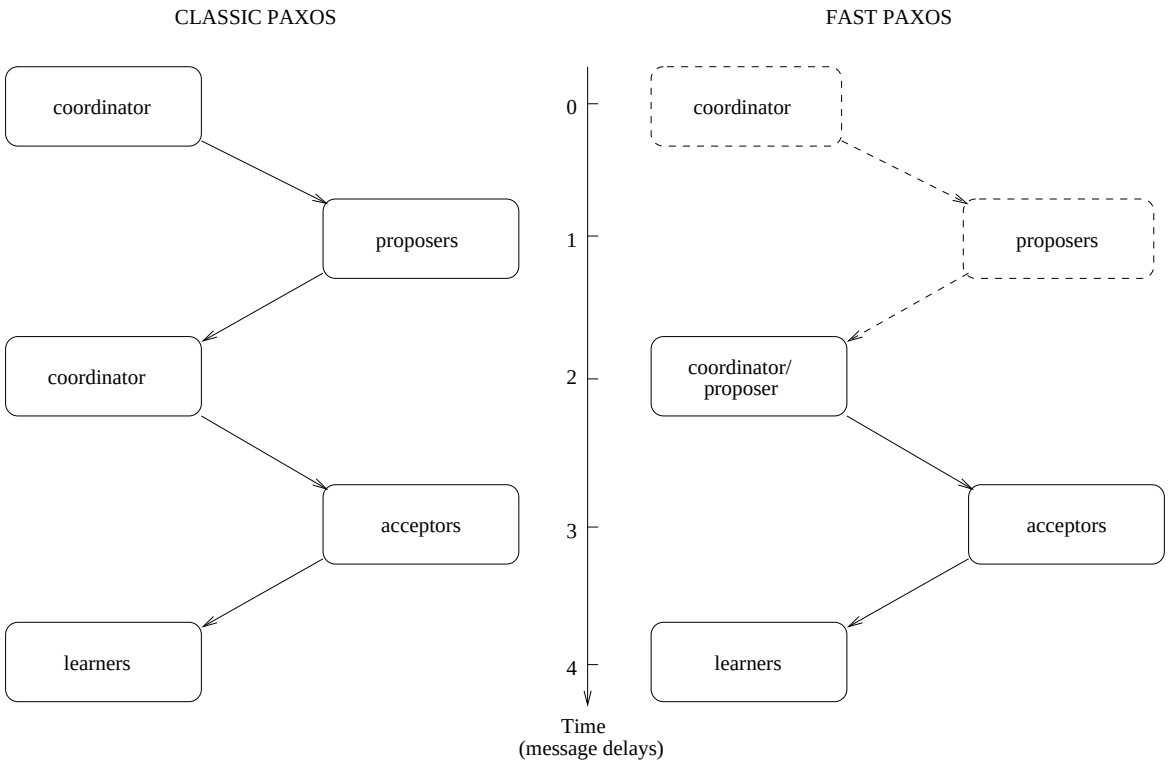


Figure 2.4: Classic Paxos and Fast Paxos. Classic Paxos takes 4 messages delays to complete. (Classic Paxos solves consensus in 3 message delays after values are proposed.) Fast Paxos takes 3 message delays to complete in some cases. (Fast Paxos can solve consensus in 2 message delays after values are proposed.)

quorum and one propagation quorum of each old configuration must remain active. Conversely, coupling installation and upgrade translates into removing the old configuration as soon as the new configuration is installed. That is, only one configuration is active at a time and much less nodes need to be active in such circumstances, strengthening fault-tolerance.

Although configuration upgrade and Fast Paxos are two quorum based algorithms, Fast Paxos must be modified for parallelizing the executions of the two algorithms. Fast Paxos uses standard quorum system [Lam06b] whereas upgrade uses biquorum systems. More precisely, during an upgrade there is no need for all quorums to intersect each other. Instead, upgrade uses two types of quorum, and any quorum of the first type must intersect any quorum of the second type while quorums of the same type do not need to intersect. In the following we refer to a quorum of proposers, which a coordinator may contact during the first phase of Fast Paxos, as a *proposer-quorum*. Similarly, we refer to a quorum of acceptors, which a coordinator might contact in the second phase, as a *acceptor-quorum*.

In fact, it is noteworthy that two proposer-quorums do not need to intersect in Fast Paxos. This comes directly from the fact that nodes can learn of a new ballot at any time, abstaining from earlier ballots, yet there is no need to abstain during the first phase. But for progress of the algorithm, the coordinator needs to learn about the voted values of the largest ballot of the consensus instance. This translates into the need for any proposer-quorum to intersect any acceptor-quorum. Furthermore, any acceptor-quorums must intersect each other, otherwise two acceptor-quorums would possibly vote for distinct values. To summarize, in a single consensus instance Fast Paxos requires only the following intersections: For any proposer-quorum P, and any acceptor-quorum A: $P \cap A \neq \emptyset$ and $A \cap A \neq \emptyset$.

Built upon these observations, Fast Paxos is modified to use the biquorum systems of the upgrade. First, proposer-quorums need not intersect, thus it can be simply set to either any consultation quorum or any propagation quorum. Since the role of the proposer-quorum is to return the value of the largest ballot, we set proposer-quorums to consultation quorum. Second, every acceptor-quorum must intersect other acceptor-quorums but also all proposer-quorums, thus it is set to the union of a consultation quorum and a propagation quorum.

For the sake of communication efficiency, we piggyback messages of Fast Paxos and messages of upgrade, all together leading to the sequence of phases depicted in Figure 2.5. The first phase of Fast Paxos is unnecessary in many circumstances as previously mentioned. The second phase of Fast Paxos ensures that enough nodes have voted for the same configuration in the current ballot (decision) so that no different configurations can be decided later on. The first phase of upgrade consults the current object state and propagates the newly decided configuration so that any later operations will apply to it. Object state is easily piggybacked in the Fast Paxos second phase, however, this phase is insufficient to ensure that enough nodes are aware of the new configuration. This phase makes a learner decide only if an acceptor-quorum has voted for the right configuration whereas not enough learners may be aware of this configuration. To make sure that all operations will apply to the new configuration, an additional propagation message must be received from enough learners. Next Section describes in details a dynamic memory including this reconfiguration mechanism.

2.3 Dynamic Distributed Shared Memory (benefiting from Consensus)

This Section describes a distributed shared memory that tolerates dynamism, called RDS. We present the algorithm for a single object; atomicity is preserved under composition and the complete shared memory is obtained by composing multiple objects.

Reconfigurable Distributed Storage for dynamic networks. The Reconfigurable Distributed Storage (RDS) integrates the Paxos algorithm for participants to decide upon a new configuration.

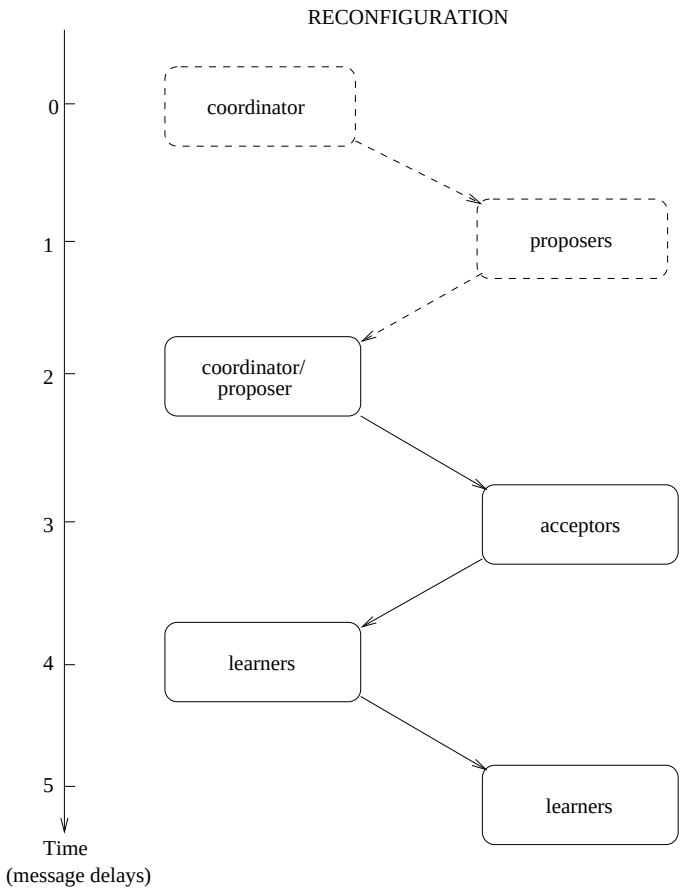


Figure 2.5: The reconfiguration protocol of RDS takes at the maximum 5 message delays and may take 3 message delays. After consensus is reached a single additional message delay is necessary for informing enough nodes about the newly decided configuration.

During the reconfiguration mechanism, each node considering itself to be the leader can start executing a consensus instance. If no too many instances are conflicting, then the participants install the new configuration and remove the obsolete one immediately after in a safe manner. The major advantage over the previous reconfigurable mechanisms is to speed-up operations and to speed-up reconfigurations. This directly translates into better fault-tolerance and higher quality of service. The RDS algorithm is described in the next subsection as a sequence of phases. Each of these phases can run concurrently with each other. The detailed algorithm is specified in Input/Output Automata in Appendix B.

2.3.1 Read and Write Operations using Global Knowledge

Algorithm 2 Read and **Write** protocols, requiring up to two phases (four message delays).

```
 1: State of node i:
 2:     s, a sequence number;
 3:     configs, the set of all known active configurations;
 4:     op-configs, all known active configurations at the start of the current phase;
 5:     op-type ∈ {read, write}, the operation type;
 6:     message-type ∈ {RW1a, RW1b, RW2a, RW2b}, the type of the message.

 7: RW-Phase-1a:
 8:     Choose a unique sequence number, s
 9:     for every c ∈ configs do
10:         Send ⟨RW1a, s⟩ to R ∈ c
11:     op-configs ← configs

12: RW-Phase-1b:
13:     Upon reception of ⟨RW1a, s⟩ from j:
14:     Sends ⟨RW1b, s, tag, value⟩ to j
```

Read and write operations proceed by accessing the currently active configurations. Each replica maintains a *tag* and a *value* for the data being replicated. Tag is a counter-id couple used as a write operation version number where its node id serves as a tiebreaker. Each read or write operation potentially requires two phases: the *consultation* phase (**RW-Phase-1**) to consult some replicas, learning the most up-to-date tag and value, and the *propagation* phase (**RW-Phase-2**) to propagate the tag and value to the replicas. In a consultation phase, the initiator contacts one consultation quorum from each active configuration, and remembers the largest tag and its associated value. In a propagation phase, read operations and write operations behave differently: a write operation chooses a new tag that is strictly larger than the one discovered in the consultation phase, and sends the new tag and new value to a write quorum; a read operation simply sends the tag and value discovered in the consultation phase to a propagation quorum.

Single-phase read operations. Sometimes, a read operation can avoid performing the propagation phase, if some prior read or write operation has already propagated that particular tag and value. Once a tag and value has been propagated, be it by a read or a write operation, it is marked *confirmed* (Line 39). If a read operation discovers that a tag has been confirmed, it can skip the second phase (Lines 24–25).

One complication arises when during a phase, a new configuration becomes active. In this case, the read or write operation must access the new configuration as well as the old one. In order to accomplish this, read or write operations save the set of currently active configurations, *op.configs*, when a phase begins (Lines 11, 28); a reconfiguration can only add configurations to this set—

15: **RW-Phase-2a:**
16: Upon reception of $\langle \text{RW1b}, s, tag, value \rangle$ from j:
17: **if** $tag > tag_i$ **then**
18: $\langle tag_i, val_i \rangle \leftarrow \langle tag, value \rangle$
19: *rvcd-from-2a* $\leftarrow$ *rvcd-from-2a* $\cup \{j\}$
20: **if** for any $c \in$ *configs*, there exists $R \in c$ such that $R \subseteq$ *rcvd-from-2a* **then**
21: **if** *op-type* $=$ write **then**
22: $\langle tag.counter_i, tag.id_i \rangle \leftarrow \langle tag.counter_i + 1, i \rangle$
23: $val_i \leftarrow v$
24: **else if** *tag* is marked as confirmed **then**
25: return val_i
26: **for** every $c \in configs$ **do**
27: Send $\langle \text{RW2a}, s, tag_i, val_i \rangle$ to $W \in c$
28: *op-configs* $\leftarrow$ *configs*

29: **RW-Phase-2b:**
30: Upon reception of $\langle \text{RW2a}, s, tag, value \rangle$ from j:
31: **if** $tag > tag_i$ **then**
32: $\langle tag_i, val_i \rangle \leftarrow \langle tag, value \rangle$
33: Sends $\langle \text{RW2b}, s, tag_i, val_i, configs \rangle$ to j

34: **RW-Done:**
35: Upon reception of $\langle \text{RW2b}, s, tag, value, configs \rangle$ from j:
36: *op-configs* $\leftarrow$ *op-configs* $\cup$ *configs*
37: *rvcd-from-done* $\leftarrow$ *rvcd-from-done* $\cup \{j\}$
38: **if** for every $c \in$ *configs*, there exists $W \in c$ such that $W \subset$ *rcvd-from-done* **then**
39: Mark *tag* as confirmed
40: **if** *op-type* $=$ read **then**
41: return val_i

none are removed during the phase. Even if a reconfiguration finishes with a configuration, the consultation or propagation phase must continue to use it.

2.3.2 Reconfiguration by Replacement

This section presents a periodic and global reconfiguration protocol. Quorum members run a Fast Paxos consensus instance to decide upon a new configuration to install, then they install this configuration by notifying the nodes.

When a client wants to change the set of replicas, it initiates a reconfiguration, specifying a new configuration. The nodes then initiate a consensus protocol, ensuring that everyone agrees on the active configuration, and that there is a total ordering on configurations. The resulting protocol is somewhat more complicated than typical consensus, since at the same time, the reconfiguration operation propagates information from the old configuration to the new configuration.

Algorithm 3 Recon protocol requiring up to three phases (five message delays).

State of node i: c, the current configuration to replace; c', the configuration competing for installation in the current reconfiguration; *leader*, a boolean indicating if the current node considers itself as a coordinator; *proposal*, the configuration proposed by the leader, ℓ; *tag*, *value*, the tag and value of the object.

Recon-Phase-Init i: If c is the only configuration in the set of active configurations, then the reconfiguration can begin. The request is forwarded to the putative leader. If the leader has already completed Phase 1 for some ballot b, then it can skip Phase 1, and use this ballot in Phase 2. Otherwise, the leader performs Phase 1.

Recon-Phase-1a:
  if *leader* **then**
    Chooses a unique larger ballot $b.num$
    Send $\langle \mathsf{Recon1a}, b.num \rangle$ to $R \in c$
    $proposal \leftarrow c'$

Recon-Phase-1b:
  Upon reception of $\langle \mathsf{Recon1a}, b.num \rangle$ from ℓ:
  if there is no $b' \in known\text{-}ballot : b'.num > b.num$ **then**
    Let b'' be such that $b''.num = \max\{b.num\} : b \in voted\text{-}ballot[c]$
    Let c'' be $b''.conf$
    Send $\langle \mathsf{Recon1b}, b.num, configs, b''.num, c'' \rangle$ to ℓ

Recon-Phase-2a:
  if *leader* **then**
    Upon reception of $\langle \mathsf{Recon1b}, b.num, configs, b''.num, c'' \rangle$ from j:
    if $(b''.num \neq \perp)$ **then** $S \leftarrow S \cup \{< b''.num, c'' >\}$
    $rvcd\text{-}from\text{-}2a \leftarrow rvcd\text{-}from\text{-}2a \cup \{j\}$
    $op\text{-}configs \leftarrow op\text{-}configs \cup \{configs\}$
    if there exists $R \in c$ such that $R \subset rcvd\text{-}from\text{-}2a$ **then**
      if $S \neq \emptyset$ **then**
        Let $proposal$ be such that $\langle b, proposal \rangle \in S$ and $b = max_{\langle bn, * \rangle \in S}\{bn\}$
      Send $\langle Recon2a, b.num, c, v \rangle$ to $R \in c$ and to $W \in c$
      $S \leftarrow \emptyset$
      $rvcd\text{-}from\text{-}2a \leftarrow \emptyset$

Recon-Phase-2b:
  Upon reception of $\langle \mathsf{Recon2b}, b.num, c, c' \rangle$ from ℓ:
  if c is the only active configuration **then**
    if there is no $b' \in known\text{-}ballot : b'.num > b.num$ **then**
      Send $\langle \mathsf{Recon2b}, b.num, c, c', tag, value \rangle$ to $R \in c$ and to $W \in c$

The reconfiguration protocol uses the Fast Paxos algorithm [KR02, Lam05, Lam06b]. Fast Paxos is detailed above. The reconfiguration request is forwarded to a coordinator, which coordinates the reconfiguration, consisting of three phases: a *prepare* phase, **Recon-Phase-1**, in which a

ballot is made ready, a *propose* phase, **Recon-Phase-2**, in which the new configuration is proposed, and a *propagate* phase, **Recon-Phase-3**, in which the results are distributed.

```
31: Recon-Phase-3:
32:     Upon reception of ⟨Recon2b, b.num, c, c′, tag, value⟩ from j:
33:     rvcd-from-3a ← rvcd-from-3a ∪ {j}
34:     if c is the only active configuration then
35:         if there exists W ∈ c and R ∈ c such that R ∪ W ⊂ rcvd-from-3a then
36:             configs ← configs ∪ {c′}
37:             op-configs ← op-configs ∪ {c′}
38:             if tag > tag_i then
39:                 ⟨tag_i, val_i⟩ ← ⟨tag, value⟩
40:             Send ⟨Recon3a, c, c′, tag, value⟩ to R ∈ c and to W ∈ c
41:             rvcd-from-3a ← ∅

42: Recon-Phase-Done:
43:     Upon reception of ⟨Recon3a, c, c′, tag, value⟩ from j:
44:     rvcd-from-done ← rvcd-from-done ∪ {j}
45:     if there exists W ∈ c and R ∈ c such that R ∪ W ⊂ rcvd-from-done then
46:         configs ← configs \ {c}
47:         rvcd-from-done ← ∅
48:         if tag > tag_i then
49:             ⟨tag_i, val_i⟩ ← ⟨tag, value⟩
```

The prepare phase accesses a consultation quorum of the old configuration, and thus learns about any earlier ballots (Line 20). When the coordinator concludes the prepare phase, it chooses a configuration to propose: if no new configurations have yet been proposed to replace the current old configuration, the coordinator can propose its own preferred new configurations (Line 17); otherwise, the coordinator must choose the previously proposed configuration with the largest ballot (Line 7). The propose phase then begins, accessing both a read and a write quorum of the old configuration (Line 35). This serves two purposes: it requires that the nodes in the old configuration cast a vote for the new configuration, and it collects information on the tag and value from the old configuration. Finally, the propagate phase accesses both a read and a write quorum from the old configuration (Line 45); this ensures that enough nodes are aware of the new configuration to ensure that any concurrent reconfiguration request obtains the desired result.

There are two optimizations included in the protocol. First, if a node has already prepared a ballot as part of a prior reconfiguration, it can continue to use the same ballot for the new reconfiguration, without redoing the prepare phase. This means that if the same node initiates multiple reconfigurations, only the first reconfiguration has to perform the prepare phase. Second, the propose phase can terminate when *any* node, even if it is not the coordinator, discovers that an appropriate set of quorums has voted for the new configuration. If all the nodes in a quorum send their responses to the propose phase to all the nodes in the old configuration, then all the replicas can terminate the propose phase at the same time, immediately sending out propagate messages.

Again, when any node receives a propagate response from enough nodes, it can terminate the propagate phase. This saves the reconfiguration one message delay. Together, these optimizations mean that when the same node is performing repeated reconfigurations, it only requires three message delays: the coordinator sending the propose message to the old configuration, the nodes in the old configuration sending the responses to the nodes in the old configuration, and the nodes in the old configuration sending a propagate message to the initiator, which can then terminate the reconfiguration.

2.3.3 Safety Proof: Implementing Atomicity

In this section, we show that the read and write operations are atomic (linearizable). We depend on two lemmas commonly used to show linearizability: Lemmas 13.10 and 13.16 in [Lyn96].

We use the tag of the operations to induce a partial ordering on operations, which then allows us to prove the key property necessary to guarantee atomicity: if π_1 is an operation that completes before π_2 begins, then the tag of π_1 is no larger than the tag of π_2; if π_2 is a write operation, the inequality is strict.

Ordering configurations. Before we can reason about the consistency of read and write operations, however, we must show that nodes agree on the active configurations. For a reconfiguration replacing configuration c, we say that reconfiguration $\langle c, c' \rangle$ is *well-defined* if no node replaces configuration c with any configuration except c'. This is, essentially, showing that the consensus protocol successfully achieves agreement.

The proof is an extension of the proof in [Lam98] which shows that Paxos guarantees agreement, modified to incorporate optimizations in our algorithm and reconfiguration.

Theorem 2.3.1 *For all executions, there exists a sequence of configurations, $c_1, c_2, \ldots$, such that reconfiguration $\langle c_i, c_{i+1} \rangle$ is well-defined for all i.*

Proof. We proceed by induction: assume that for all $\ell' < \ell$, $\langle c_{\ell'}, c_{\ell'+1} \rangle$ is well-defined. If configuration c_ℓ is ever replaced in the set of active configurations with configuration $c_{\ell+1}$, we show that $\langle c_\ell, c_{\ell+1} \rangle$ is also well-defined. Assume, by contradiction, that this is not the case. Then there exist two nodes, say i and j, that complete the propose phase (**Recon-Phase-2**) and replace c_ℓ with two different configurations, c and c'. These two nodes must have different ballots, b and b', respectively, at the end of the propose phase (**Recon-Phase-2**). Without loss of generality, assume that $b < b'$.

At some point, ballot b' must have completed a prepare phase (**Recon-Phase-1**). First, consider the case where b' was prepared as part of a recon operation installing configuration $c_{\ell+1}$. Since by induction all smaller reconfigurations are well-defined, we can conclude that the read quorum associated with preparing b' must intersect the write quorum associated with proposing b; let i' be a node in the intersection. If i' received the prepare message from b' prior to the propose message

for b, then ballot b could not be smaller than ballot b'. Therefore, we can conclude that node i' received the propose message for b, along with the proposed configuration, c, prior to responding to the prepare message for b'. Hence when some node proposed ballot b', it must have been aware of ballot b and configuration c, leading to the conclusion that $c = c'$, contradicting our initial assumption. (It is possible to show using the same argument that no configuration with a larger ballot can be available to the proposer of b'.)

Second, consider the case, then, where b' was prepared as part of a recon operation installing a configuration $c_{\ell'} < c_\ell$. In this case, we can show that $b \geq b'$, contradicting our assumption. In particular, some recon for $c_{\ell'}$ must terminate prior to the proposals of b and b'. By examining the quorum intersections, we can show that the ballot b' must have been discovered by the proposal for this earlier recon installing $c_{\ell'}$; from there it must have been discovered by the proposal for the recon installing $c_{\ell'+1}$, and so on, until it was discovered by the proposal for b, from which we can conclude that $b \geq b'$.

We can therefore conclude from these two cases that reconfiguration $\langle c_\ell, c_{\ell+1} \rangle$ is well-defined. □

Ordering operations. We now proceed to show that tags induce a valid ordering on the operations. If both operations "use" the same configuration, then this property is easy to see: operation π_1 propagates its tag to a propagation quorum, and π_2 discovers the tag when reading from a consultation quorum. The difficult case occurs when π_1 and π_2 use differing configurations. In this case, the reconfigurations propagate the tag from one configuration to the next.

We refer to the smallest tag at a node that replaces configuration c_ℓ with configuration $c_{\ell+1}$ as the "tag for configuration $c_{\ell+1}$." We can then easily conclude from this definition, along with a simple induction argument, that:

Invariant 2.3.2 *If some node i has configuration $c_{\ell+1}$ in its set of active configurations, then its tag is at least as large as the tag for configuration $c_{\ell+1}$.*

This invariant allows us to conclude two facts about how information is propagated by reconfiguration operations: the tag of each configuration is no larger than the tag of the following configuration, and the tag of a read/write operation is no larger than the tag of a configuration in its set of active configurations.

The next lemma requires showing how read and write operations propagate information *to* a reconfiguration operation:

Lemma 2.3.3 *If c_ℓ is the largest configuration in i's op-config set of operational configurations when **RW-Phase-2** completes, then the tag of the operation is no larger than the tag of configuration $c_{\ell+1}$.*

Proof. During the **RW-Phase-2**, the tag of the read or write operation is sent to a propagation quorum of the configuration c_ℓ. This quorum must intersect the consultation quorum during the

Recon-Phase-2 propagate phase of the reconfiguration that installs $c_{\ell+1}$. Let i' be a node in the intersection of the two quorums. If i' received the reconfiguration message prior to the read/write message, then node i would learn about configuration $c_{\ell+1}$. However we assumed that c_ℓ was the largest configuration in *op-config* at i at the end of the phase. Therefore we can conclude that the read/write message to i preceded the reconfiguration message, ensuring that the tag was transfered as required. □

Theorem 2.3.4 *For any execution, α, it is possible to determine a linearization of the operations.*

Proof. As discussed previously, we need to show that if operation π_1 precedes operation π_2, then the tag of π_1 is no larger than the tag of π_2, and if π_1 is a write operation, then the inequality is strict.

There are three cases to consider. First, assume π_1 and π_2 use the same configuration. Then the write quorum accessed during the propagate phase of π_1 intersects the consultation quorum accessed during the consultation phase of π_2, ensuring that the tag is propagated.

Second, assume that the *smallest* configuration accessed by π_1 in the propagate phase is larger than the *largest* configuration accessed by π_2 in the consultation phase. This case cannot occur. Let c_ℓ be the largest configuration accessed by π_2. Prior to π_1, some configuration installing configuration $c_{\ell+1}$ must occur. During the final phase **Recon-Phase-2** of the reconfiguration, a consultation quorum of configuration c_ℓ is notified of the new configuration. Therefore, during the consultation phase of π_2, the new configuration for $c_{\ell+1}$ would be discovered, contradicting our assumption.

Third, assume that the *largest* configuration c_ℓ accessed by π_1 in the propagate phase **RW-Phase-2** is smaller than the *smallest* configuration $c_{\ell'}$ accessed by π_2 in the consultation phase **RW-Phase-1**. Then, Lemma 2.3.3 shows that the tag of π_1 is no larger than the tag of c_ℓ; Invariant 2.3.2 shows that the tag of c_ℓ is no larger than the tag of $c_{\ell'}$ and that the tag of $c_{\ell'}$ is no larger than the tag of π_2. Together, these show the required relationship of the tags.

If π_1 skips the second phase, **RW-Phase-2**, then some earlier read or write operation must have performed a **RW-Phase-2** for the same tag, and hence the proof follows as before. □

2.3.4 Liveness Proof: Speeding up Reconfiguration to strengthen Fault Tolerance

In this section we examine the performance of RDS, focusing on the efficiency of reconfiguration and how the algorithm responds to instability in the network. In order for the algorithm to make progress in an otherwise asynchronous system, we need to make a series of assumptions about the network delays, the connectivity, and the failure patterns. In particular, we assume that, eventually, the network stabilizes and delivers messages with a delay of d. The main results in this section are then as follows. First, we show that the algorithm "stabilizes" within $e+2d$ time after the network stabilizes, where e is the time required for new nodes to fully join the system and notify old

nodes about their existence. Second, we show that after the algorithm stabilizes, reconfiguration completes in $5d$ time; if a single node performs repeated reconfigurations, then after the first, each subsequent reconfiguration completes in $3d$ time. Finally, we show that after the algorithm stabilizes, reads and writes complete in $8d$ time, reads complete in $4d$ time if there is no interference from ongoing writes, and in $2d$ if no reconfiguration is pending.

Eventual synchrony as a requirement. Asynchrony makes it impossible to guarantee that consensus will terminate in a system where continual node failures occur as it has been proved by Fisher, Lynch, and Paterson [FLP85]. In order to ensure that the algorithm progresses, we assume, among others, eventual synchrony, meaning that the communication between nodes becomes eventually synchronous. More precisely, we assume that eventually (at some unknown point) the network stabilizes, becoming synchronous and delivering messages in bounded (but unknown) time.

Our goal is to model a system that becomes stable at some (unknown) point during the execution. Formally, let α be a (timed) execution and α' a finite prefix of α during which the network may be unreliable and unstable. After α' the network is reliable and delivers messages in a timely fashion. We refer to $\ell time(\alpha')$ as the time of the last event of α'. In particular, we assume that following $\ell time(\alpha')$: (i) all local clocks progress at the same rate, (ii) messages are not lost and are received in at most d time, where d is a constant unknown to the algorithm, (iii) nodes respond to protocol messages as soon as they receive them and they broadcast messages every d time to all service participants, (iv) all enabled actions are processed with zero time passing on the local clock.

Other assumptions. Additionally, we restrict the rate of reconfiguration after stabilization, and limit node failures such that some quorum remains available in an active configuration. (For example, in majority quorums, this means that only a minority of nodes in a configuration fail between reconfigurations.) We present a more detailed explanation in Section 2.3.5.

Generally, in quorum-based algorithms, the operations are guaranteed to terminate provided that at least one quorum does not fail. In contrast, for a reconfigurable quorum system we assume that at least one quorum does not fail prior to a successful reconfiguration replacing it. For example, in the case of majority quorums, this means that only a minority of nodes fail in between reconfigurations. Formally, we refer to this as *configuration-viability*: at least one read quorum and one write quorum from each installed configuration survive $4d$ after (i) the network stabilizes and (ii) a following successful reconfiguration operation.

We place some easily satisfied restrictions on reconfiguration. First, we assume that each node in a new configuration has completed the joining protocol at least time e prior to the configuration being proposed, for a fixed constant e. We call this *recon-readiness*. Second, we assume that after stabilization, reconfigurations are not too frequent: $5d$*-recon-spacing* implies that two reconfiguration termination are at least $5d$ apart.

Also, after stabilization, we assume that nodes, once they have joined, learn about each other

quickly, within time e. We refer to this as *join-connectivity*.

Finally, we assume that a leader election service chooses a single coordinator, namely the *leader*, at time $\ell time(\alpha') + e$ and that it remains alive until the next leader is chosen and for a sufficiently long time for a reconfiguration to complete. For example, a leader may be chosen among the members of a configuration based on the value of an identifier.

Bounding reconfiguration delays. We now show that reconfiguration attempts complete within at most five message delays after the system stabilizes. Let ℓ be the node identified as the leader when the reconfiguration begins.

The following lemma describes a preliminary delay in reconfiguration when a non-leader node forwards the reconfiguration request to the leader.

Lemma 2.3.5 *Let the first* $\mathsf{recon}(c,c')$ *event at some active node i, where $i \neq \ell$, occur at time t and let t' be* $\max(\ell time(\alpha'),t)+e$. *Then, the leader ℓ starts the reconfiguration process at time $t'+2d$.*

Proof. When the $\mathsf{recon}(c,c')$ occurs at time t, one of two things happen: either the reconfiguration fails immediately, if c is not the current, unique, active configuration, or the recon request to is forwarded to the leader. Observe that *join-connectivity* ensures that i knows the identity of the leader at time t', so no later than time $t'+d$, i sends a message to ℓ that includes reconfiguration request information. By time $t'+2d$ the leader receives message from i and starts the reconfiguration process. □

The next lemma implies that after some time following reconfiguration request, there is a communication round where all messages include the same ballot.

Lemma 2.3.6 *After time $\ell time(\alpha')+e+2d$, ℓ knows about the largest ballot in the system.*

Proof. We know that after $\ell time(\alpha')$, only ℓ can create a new ballot. Therefore ballot b must have been created before $\ell time(\alpha')$. Since ℓ is the leader at time $\ell time(\alpha')+e$, we know that ℓ has joined before time $\ell time(\alpha')$.

If ballot b still exists after $\ell time(\alpha')$ (the case we are interested in), then there are two possible scenarios. Either ballot b is conveyed by an in transit message or it exists an active node i aware of it at time $\ell time(\alpha')+e$. In the former case, gossip policy implies that the in transit message is received at time t, such that $\ell time(\alpha')+e < t < \ell time(\alpha')+e+d$. However, it might happen that ℓ does not receive it, if the sender ignored its identity at the time the send event occurred. Thus, at this time one of the receiver sends a message containing b to ℓ. Its receipt occurs before time $\ell time(\alpha')+e+2d$ and ℓ learns about b. In the latter case, by join-connectivity assumption at time $\ell time(\alpha')+e$, i knows about ℓ. Gossip policy implies i sends a message to ℓ before $\ell time(\alpha')+e+d$ and this message is received by ℓ before $\ell time(\alpha')+e+2d$, informing it of ballot b. □

Next theorem shows that any reconfiguration completes in at most $5d$ time, following $\ell time(\alpha')$. In Theorem 2.3.8 we show that when the leader node has successfully completed the

previous reconfiguration request then it is possible for the subsequent reconfiguration to complete in at most $3d$.

Theorem 2.3.7 *Assume that ℓ starts the reconfiguration process initiated by $\mathsf{recon}(c,c')_*$ at time $t \geq \ell time(\alpha') + e + 2d$. Then the corresponding reconfiguration completes no later than $t + 5d$.*

Proof. By Lemma 2.3.6, ℓ knows about the largest ballot in the system after $\ell time(\alpha') + e + 2d$. By *configuration-viability*, we know that at least one consultation quorum and at least one propagation quorum of configuration c are active during the reconfiguration, and *recon-readiness* implies that nodes have joined the service and are aware of each other. The phases of Paxos imply that there are two message exchanges followed by a broadcast in phase **Recon-Phase-3a**. Since the message delay is bounded by d, each message exchange requires $2d$ time and an additional d for the broadcast. From this we conclude that the reconfiguration process terminates in phase **Recon-Phase-3b** at time $t + 5d$. □

Theorem 2.3.8 *Let ℓ be the leader node that successfully conducted the reconfiguration process from c to c'. Assume that ℓ starts a new reconfiguration process from c' to c'' at time $t \geq \ell time(\alpha') + e + 2d$. Then the corresponding reconfiguration from c' to c'' completes at the latest at time $t + 3d$.*

Proof. The proof is analogous to the proof of Theorem 2.3.7. Observe that at the beginning of the new reconfiguration process ℓ has the highest ballot. This means that ℓ may keeps its ballot and starts from **Recon-Phase-2a** (since it has previously executed **Recon-Phase-1b**). Hence only a single message exchange in **Recon-Phase-2a**/**Recon-Phase-2b** and a single broadcast following **Recon-Phase-3a** takes place. Therefore, the last phase of Paxos occurs at time $t + 3d$. □

Bounding read-write delays. This Section presents bounds on the duration of read/write operations under assumptions stated in the previous section. Recall from Section 2.3 that both the read and the write operations are conducted in two phases, first the query phase and second the propagate phase. We begin by first showing that each phase requires at least $4d$ time. However, if the operation is a read operation and no reconfiguration and no write propagation phase is concurrent, then it is possible for this operation to terminate in only $2d$ – see proof of Lemma 2.3.9. The final result is a general bound of $8d$ on the duration of any read/write operation.

Lemma 2.3.9 *Consider a single phase of a read or a write operation initiated at node i at time t, where i is a node that joined the system at time $\max(t - e - 2d, \ell time(\alpha'))$. Then this phase completes at the latest at time $\max(t, \ell time(\alpha') + e + 2d) + 4d$.*

Proof. Let c_k be the largest configuration in any active node's *op-configs* set, at time $t - 2d$. By the *configuration-viability* assumption, at least one consultation quorum and at least one propagation quorum of c_k are active for the interval of $4d$ after c_{k+1} is installed. By the *join-connectivity* and

the fact that i has joined at time $\max(t-e-2d, \ell time(\alpha'))$, i is aware of all active members of c_k by the time $\max(t-2d, \ell time(\alpha')+e)$.

Next, by the timing of messages we know that within d time a message is sent from each active members of c_k to i. Hence, at time $\max(t, \ell time(\alpha')+e+2d)$ node i becomes aware of c_k, i.e. $c_k \in$ *op-configs*.

At d time later, messages from phase **RW-Phase-1a** or **RW-Phase-2a** are received and **RW-Phase-1b** or **RW-Phase-2b** starts. Consequently, no later than $\max(t, \ell time(\alpha')+e+2d)+2d$, the second message of **RW-Phase-1** or **RW-Phase-2** is received.

Now observe that configuration might occur in parallel, therefore it is possible that a new configuration is added to the *op-configs* set during **RW-Phase-1** or **RW-Phase-2**. Discovery of new configurations results in the phase being restarted, hence completing at time $\max(t, \ell time(\alpha')+e+2d)+4d$. By *recon-spacing* assumption no more than one configuration is discovered before the phase completes. □

Theorem 2.3.10 *Consider a read operation that starts at node i at time t:*

1. *If no write propagation phase is pending at any node and no reconfiguration is ongoing, then it completes at time* $\max(t, \ell time(\alpha')+e+2d)+2d$.
2. *If no write propagation phase is pending, then it completes at time* $\max(t, \ell time(\alpha')+e+2d)+8d$.

Consider a write operation that starts at node i at time t. Then it completes at time $\max(t, \ell time(\alpha')+e+2d)+8d$.

Proof. At the end of the **RW-Phase-1**, if the operation is a write, then a new non confirmed tag is set. If the operation is a read, the tag is the highest received one. This tag was maintained by a member of the contacted consultation quorum, and it is confirmed only if the phase that propagated it to this member has completed. From this point, if the tag is not confirmed, then in any operation the fix-point of propagation phase **RW-Phase-2** has to be reached. But, if the tag is already confirmed then the read operation can terminate directly at the end of the first phase. By Lemma 2.3.9, this occurs at the latest at time $\max(t, \ell time(\alpha')+e+2d)+4d$; or at time $\max(t, \ell time(\alpha')+e+2d)+2d$ if no reconfiguration is concurrent. Likewise by Lemma 2.3.9, the **RW-Phase-2** fix-point is reached in at most $4d$ time. That is, any operation terminates by confirming its tag no later than $\max(t, \ell time(\alpha')+e+2d)+8d$. □

2.3.5 Experimental Analysis of Reconfiguration

Musial and Shvartsman [MS04] developed a prototype distributed implementation that incorporate both RAMBO and RAMBO II. The system was developed by manually translating the Input/Output Automata specification to Java code. To mitigate the introduction of errors during translation,

the implementers followed a set of precise rules, similar to [CS98], that guided the derivation of Java code from Input/Output Automata notation. The implementers developed RDS based on the existing RAMBO codebase [GMS04] on a network of workstations. The primary goal of our experiments was to gauge the cost introduced by reconfiguration. When reconfiguration is unnecessary, there are simpler and more efficient algorithms to implement a replicated DSM. Our goal is to achieve performance similar to the simpler algorithms while using reconfiguration to tolerate dynamic changes.

To this end, we designed three series of experiments where the performance of RDS is compared against the performance of an atomic memory service which has no reconfiguration capability — essentially the algorithm of Attiya, Bar Noy, and Dolev [ABND95] (the "ABD protocol"). In this section we briefly describe these implementations and present our initial experimental results. The results primarily illustrate the impact of reconfiguration on the performance of read and write operations.

For the implementation we manually translated the IOA specification (from the appendix) into Java code. The target platform is a cluster of eleven machines running Linux. The machines are various Pentium processors up to 900 MHz interconnected via a 100 Mbps Ethernet switch.

Each instance of the algorithm uses a single socket to receive messages over TCP/IP, and maintains a list of open, outgoing connections to the other participants of the service. The nondeterminism of the I/O Automata model is resolved by scheduling locally controlled actions in a round-robin fashion. The ABD and RDS algorithm share parts of the code unrelated to reconfiguration, in particular that related to joining the system and accessing quorums. As a result, performance differences directly indicate the costs of reconfiguration. While these experiments are effective at demonstrating comparative costs, actual latencies most likely have little reflection on the operation costs in a fully-optimized implementation.

Experiment (a). In the first experiment, we examine how the RDS algorithm responds to different size configurations (and hence different levels of fault-tolerance). We measure the average operation latency while varying the size of the configurations. Results are depicted in Figure 2.3.5.

In all experiments, we use configurations with majority quorums. We designate a single machine to continuously perform read and write operations and compute average operation latency for different size configurations, ranging from 1 to 5. In the tests involving the RDS algorithm, we chose a separate machine to continuously perform reconfiguration of the system – when one reconfiguration request successfully terminates another is immediately submitted.

Experiment (b). In the second set of experiments, we test how the RDS algorithm responds to varying load. Figure 2.3.5 presents results of the second experiment, where we compute the average operation latency for a fixed-size configuration of five members, varying the number of nodes performing read/write operations changes from 1 to 10. Again, in the experiments involving RDS algorithm a single machine is designated to reconfigure the system. Since we only have

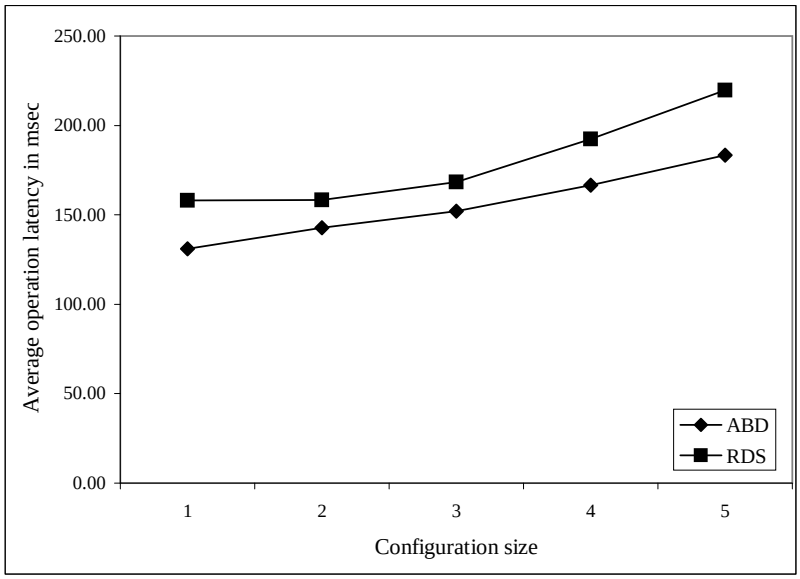


Figure 2.6: Average operation latency: as size of configurations changes

eleven machines to our disposal, nodes that are members of configurations also perform read/write operations.

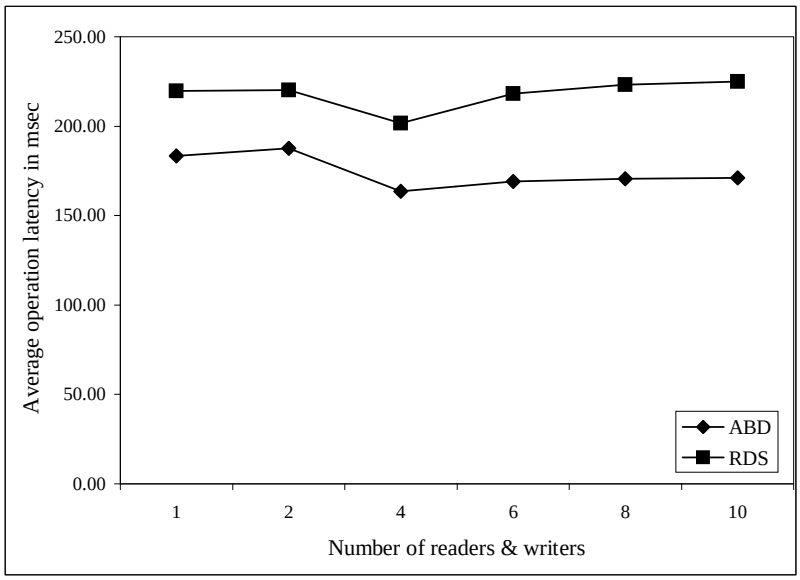


Figure 2.7: Average operation latency as number of nodes performing read/write operations changes.

Experiment (c). In the last experiment we test the effects of reconfiguration frequency. Two nodes continuously perform read and write operations, and the experiments were run varying the number of instances of the algorithm. Results of this test are depicted in Figure 2.3.5. For each of the sample points on the x-axis, the size of configuration used is half of the algorithm instances. As in the previous experiments, a single node is dedicated to reconfigure the system. However, here we insert a delay between the successful termination of a reconfiguration request and the submission of another. The delays used are 0, 500, 1000, and 2000 milliseconds. Since we only have eleven machines to our disposal, in the experiment involving 16 algorithm instances, some of the machines run two instances of the algorithm.

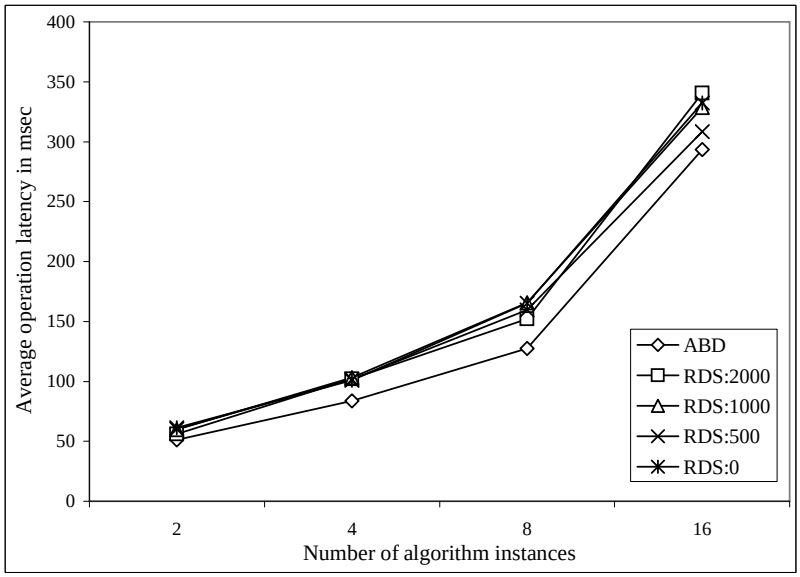


Figure 2.8: Average operation latency as the reconfiguration and the number of participants changes.

Interpretation. We begin with the obvious. In all three series of experiments, the latency of read/write operations for RDS is competitive with that of the simpler ABD algorithm. Also, the frequency of reconfiguration has little effect on the operation latency. These observations lead us to conclude that the increased cost of reconfiguration is only modest.

This is consistent with the theoretical operation of the algorithm. It is only when a reconfiguration exactly intersects an operation in a particularly bad way that operations are delayed. This is unlikely to occur, and hence most read/write operations suffer only a modest delay.

Also, note that the messages that are generated during reconfiguration, and read and write operations, include replica information as well as the reconfiguration information. Since the actions are scheduled using a round-robin method, it is likely that in some instances a single communication

phase might contribute to the termination of both the read/write and the reconfiguration operation. Hence, we suspect that the dual functionality of messages helps to keep the system latency low.

A final observation is that the latency does grow with the size of the configuration and the number of participating nodes. Both of these require increased communication, and result in larger delays in the underlying network when many nodes try simultaneously to broadcast data to all others. Some of this increase can be mitigated by using an improved multicast implementation; some can be mitigated by choosing quorums optimized specifically for read or write operations.

2.4 Discussion and Conclusion

2.4.1 Communication Overhead

As pointed out by the experimentations, communication impacts much more on the operation delay than reconfiguration. In networks where the bandwidth is limited, mechanisms that needs a lot of bandwidth may provoke communication bottleneck at some point of the network. Recent scale-shift of distributed systems strengthen this observation. If the communication complexity of the algorithm depends on the amount of participants, bandwidth limitation prevents the system from scaling. Next, we propose two directions of remedying this problem.

This scalability issue has already been experienced in some peer-to-peer file-sharing applications like Gnutella [gnua], where all peers participate equally while some of them have drastically low bandwidth capabilities. As a result of utilizing peers on an equality-based policy while resources are heterogeneously scattered among peers, the lowest capable peers limit the overall performance of the system. To circumvent this well-known issue, peer-to-peer applications tend now to use a peer depending on the amount of resource it offers. More particularly, Kazaa [kaz]-like file-sharing applications elect supernodes which are peers with extra-capabilities to handle most of the requests. More generally, differing peer-to-peer applications, like Skype, demonstrate the need of using specific nodes to handle firewall/NAT[3] by-passing. To conclude, large-scale systems grow unboundedly while the amount of resources is bounded. This changes the peer-to-peer paradigm of ”all peers must act equally” into ”all peers must act proportionally to the resources it has”. Consequently, the problem for a peer to determine how it must participate depends on the relative amount of resource it has compared to other peers. The problem of determining the relative amount of resource a peer has has been identified as the *distributed slicing*. Dedicated solutions are described in Appendix A.

2.4.2 Conclusion

This chapter addresses the problem of emulating a distributed shared memory in a dynamic system. Dynamism induces an unbounded amount of dynamic events including failures. These dynamic

[3]NAT: Network Address Translation.

events are coped with using a periodic mechanism, called reconfiguration, that continually reinstall an active quorum system. Assuming that no too many failures occur during a bounded period of time, this reconfiguration mechanism ensures the availability of the memory at any time.

The reconfiguration relies on a variant of a consensus algorithm that executes very rapidly. The speed of reconfiguration is of primary importance in dynamic systems, since tolerance of failures diminishes as time elapses. Time must be taken into account and will be at the heart of the quorum systems proposed in Chapter 4. Because of the speed of reconfiguration, our memory tolerates high dynamism.

Moreover, reconfiguration is periodic and must happen with a frequency that depends on the inherent fault-tolerance of the quorum system. However, studying the number of quorums and their size is out of the scope of this chapter and those parameters are studied in Chapter 3.

As discussed above an open issue is to minimize the communication complexity induced by reconfiguration so that the system growth does not produce important overhead. The sake of scalability implies each node communication complexity to be non-proportional to the number of participants. The sequel of this document investigates scalability.

Chapter 3

Facing Scalability: Quorum System with Local Reconfiguration

This chapter focuses on the scalability issue in distributed shared memory. In large scale systems, the number of participants is potentially unbounded. The paradox between the bounded resources each node has and the unbounded participation experienced by the system arises several problems. This paradox prevents each node from maintaining global information. First, nodes cannot record information about every other nodes because the number of nodes is too high regarding to the amount of memory each node possesses. Second, nodes cannot even determine global characteristics such as exact system size. Indeed, the inherent dynamism continuously changes these characteristics that may not reflect the current system state at the time they are observed. Thus, the node bandwidth cannot afford the amount of messages induced like previously observed in Section 2.3.5.

In the context of distributed shared memory, we focus on the scale-shift of the number of clients. All clients have an independent behavior and may request the memory at any time. Moreover, because of the lack of global information their behavior is unpredictable. The growth of requests directly produces a load increase that may affect the memory performance: if the memory gets overloaded, then quality of service may be dramatically affected, treatments may be delayed and requests may be lost. This chapter does not investigate the scale-shift in the number of requested objects, but only the scale-shift in the number of participants.

Structural properties of the memory. For the purpose of scalability, we outline the predisposition of some memory to handle load. Recall that we consider only quorum-based memories where not all replicas are accessed during a single operations. This makes distributed file systems like Pastis [BPS05] out of the scope of this thesis. Depending on the role of its quorum members, each quorum system reacts differently to load. Since read and write operations always consist in accessing quorums one or two times, this may affect operations performance. More precisely, the set of nodes that are in a quorum and the way those nodes are accessed have an important impact

on the operation complexity. To better understand what is an efficient quorum system or a scalable quorum system, we investigate the way quorum members are scattered into quorums and the way all nodes communicate with quorum members. This translates into investigating the structural properties and communication structure of quorum systems. Furthermore, we discuss properties like *nondominance* [GMB85] and stronger ones that cope with a bounded number of faults and allow optimality in terms of load.

Scalable distributed shared memory. We propose an emulation of a distributed shared memory especially suited for large-scale systems. Since reconfiguration cost depends on the amount of information that has to be maintained, we restrict communication to logical vicinities. As a result, we present a local reconfiguration mechanism that involves only a small set of logical neighbors. Finally, we provide read and write primitives that satisfy atomic consistency. This memory adapts dynamically in face of load variation and scales well due to the limited amount of information each node maintains.

Roadmap. The following chapter is divided into four sections. Section 1 investigates the structure of quorum systems and describes properties for scalability. Section 2 focuses on communication between nodes and quorum members. Section 3 describes a complete scalable DSM. Finally Section 4 concludes the chapter by discussing some aspects of the chosen scalable DSM.

3.1 Structure of Quorum Systems

As observed in the previous chapter, progress requires that at least two quorums of the current quorum system remain active. Fault tolerance property of quorum systems resides in their structural properties: the number of quorums, the size of each quorum, the number of nodes at the intersection of two quorums... Compare, for example, a quorum system that contains all majority sets. As soon as a majority of nodes fail, the whole system fails. Intuitively, reducing the size of quorum systems may thus strengthen fault tolerance. This section investigates the structure of quorum systems and outlines structural requirements to tolerate failures and load.

3.1.1 Single-Point of Failure

Some quorum systems suffer from single point of failures, i.e., they may fail if a single member fails. A crucial property of quorum system is their availability [NW98]. The *availability* of a quorum system is the probability that all its quorum fail given the probability that each element fails. If the intersection no longer holds between two quorums, then these quorums are no longer *active*. For example, if all the nodes at the intersection of two quorums Q_1 and Q_2 fail, then quorums Q_1 and Q_2 are considered as *failed*. Now, if all quorums intersect at exactly the same node i, then failure of node i makes all quorums inactive, leading to the failure of the whole quorum

system. In this case we say that the quorum system suffers a single point of failure, meaning that the failure of a single node makes the whole quorum system fail.

A more difficult problem of interest is the inconsistency of quorum system. Beyond providing data availability in face of failures, quorum systems must provide consistency in distributed shared memory. Although quorums are replicated sets of data used to ensure fault-tolerance, quorum system may experience inconsistency. Next, we illustrate this idea presenting two quorum systems that experience single point of failure.

A star-like quorum system with single-point of failure. To better understand how a quorum system may suffer from a single point of failure, we take the star quorum system as an example. The star quorum system is defined in [HMP95] as a quorum system whose quorums have the same node in common. All of its quorums intersect at a single node, one might think of as the center of a star. Despite the replication of quorum members, the intersection among quorums is not replicated. The star quorum system, thus, ensures data availability but not data consistency despite the failure of a single node. The definition of the star quorum system is stated as follows:

Definition 3.1.1 (Star Quorum System) *Let $U = \{u_1, u_2, ..., u_n\}$ and let Q consist of the $n-1$ quorums $\{u_1, u_2\}, \{u_1, u_3\}, ..., \{u_1, u_n\}$. Then Q is a* star quorum system.

It is straightforward from the above definition that the star quorum suffers from a single point of failure. Assume for example that the single failure occurs at node u_1. In this case, there is no active node between any couple of quorums. Since no quorum intersect, all quorums are failed, hence the quorum system fails.

A tree-based quorum system with single-point of failure. Another quorum system that might experience single point of failure is the tree-structured quorum system presented in [AE90]. Actually the tree-based quorum system is a biquorum system (as defined in Definition 1.1.3) since it uses two types of quorums, the read and the write quorums (as in [Her86]). Such quorums intersect if their type is different, thus there is no need for intersection between all quorums. A read quorum is given by a recursive function returning the current node or the majority of its children, while a write quorum function picks the union of the majority of nodes at any height of the tree.

Since all write quorums contain the root node of the tree, all write quorums fail after a single node, the root, fails. If this happen, then no write quorums intersect all read quorums. Even though some read quorums may still intersect a non complete write quorum it is difficult to identify which one intersect the remaining active members of a write quorum. Because of that, the failure of the root prevents any couple of read quorum and write quorum from being active, leading to the failure of the whole quorum system.

To circumvent this issue, quorum systems must tolerate the failure of multiple nodes. To this end, either a quorum must tolerate the failure of some of its members or a quorum system must tolerate the failure of some of its quorums. In the former case, the intersection must be large enough

to persist despite some node failures whereas in the latter case there must be enough intersections to ensure that the quorum system persists despite some intersection failures. These two candidate solutions are denoted *large intersections* and *numerous intersections* and are described below.

3.1.2 Replicating the Intersection

Since intersection is the weak point of quorum system, the nodes at the intersection between quorums must be replicated. As previously said, for a quorum system to remain active, either its quorums must tolerate member failures or it must tolerate quorum failures. This is achieved using replication of intersection in two ways.

1. **Large intersections:** First, the intersection among any couple of quorum contains more than a single node. Even though a member of the intersection fails, the intersection still holds.

2. **Numerous intersections:** Second, quorums are replicated such that many intersections exist. If the common members of two quorums fail an intersection remains between another couple of quorums.

In the latter case (the numerous intersections) quorum system fault tolerance is ensured by quorum replication while a single quorum may fail. In the former case, however, quorum system fault tolerance is ensured by intersection replication, and the whole quorum system is as fault-tolerant as any single quorum.

Depending on the type of failure we consider, one of these approaches is better suited than the other.

Applications of large intersections. When considering byzantine failures, i.e., where nodes might disrespect their specification during failure instead of simply stopping, large intersections are required. In [MR04], Malkhi and Reiter define the *masking quorum systems* class including quorum systems where each couple of quorums have $2f+1$ elements in common. This quorums are used to cope with f byzantine faults. In order to obtain the accurate information that has been stored into quorum Q, one must request at least $2f+1$ elements of quorum Q to ensure that the majority of answers ($f+1$ answers) contain the right value. That is, requesting the value rely on contacting a quorum that have $2f+1$ elements in common with other quorums.

Applications of numerous intersections. Unlike large intersections that require larger quorums to handle faults, numerous intersections require multiple quorums. Since the complexity of accessing a quorum depends on its size, it is more complex to access quorums in a quorum system with large intersections than quorums in a quorum system with numerous intersections. Differently, the crash failure model we use, which allows a node to stop at any time but not to disrespect arbitrarily its specification, does not require heavy large intersections but rather numerous intersection. Due

to the complexity of accessing large intersections, numerous intersections will be preferred for the quorum system we consider in the following.

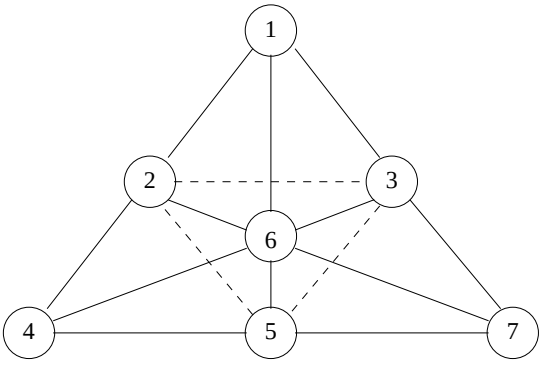


Figure 3.1: A finite-projective-plane-based quorum system with numerous intersections. The quorum system contains 7 quorums of size 3, where each element belongs to exactly 3 quorums: $\{\{1,2,4\},\{1,3,7\},\{1,5,6\},\{2,3,5\},\{2,6,7\},\{3,4,6\},\{4,5,7\}\}$.

In [Mae85], Maekawa suggests quorum system where quorums intersect at exactly one element. This property naturally excludes large intersection property while it allows numerous intersections property. The quorum system consists of m^2+m+1 nodes where $m=p^k$ where p is a prime and k a constant. This quorum systems has $m+1$ quorums, each of size $m+1$ and each pair of quorum having exactly one element in common. Figure 3.1 presents such a quorum system.

Disjoint intersections. A quorum system may satisfy numerous intersections if there are multiple intersections. Since intersections are simply sets, numerous intersections implies that the number of intersecting sets are different. Disjoint intersection is a stronger property in the sense that a quorum system satisfying numerous intersections may not satisfy disjoint intersections and a quorum system satisfying disjoint intersections satisfies numerous intersections. A quorum system that satisfies disjoint intersection contains quorums that intersect each other on distinct nodes. More formally, we have the following definition:

Definition 3.1.2 (Disjointness Property) *A quorum system Q over U satisfies* disjointness *if and only if, for any three distinct quorums $Q_1,Q_2,Q_3 \in Q$, $Q_1 \cap Q_2 \cap Q_3 = \emptyset$.*

Figure 3.2 depicts an example of a 3-gon quorum system that have disjoint intersection property. The *n*-gon quorum systems have been proposed in [KH97] for the purpose of coteries construction. Observe that some *n*-gon quorum systems do not satisfy disjoint intersection, the 3-gon quorum system used here is a special case described for the sake of illustration of disjoint intersection property.

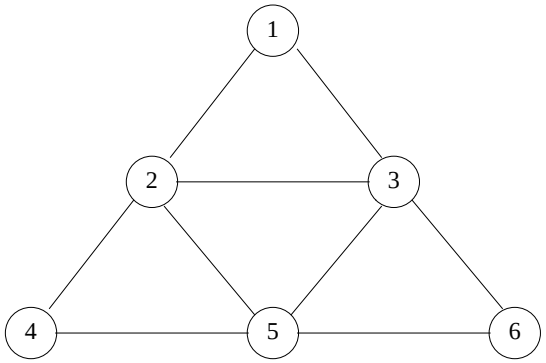


Figure 3.2: A 3-gon-based quorum system. The quorum system contains 4 quorums of size 3, where each element belongs to at most two quorums: $\{\{1,2,4\},\{1,3,6\},\{2,3,5\},\{4,5,6\}\}$.

About the fault-tolerance of quorum system, the fact that these intersections are disjoint implies that there must be at least $|Q|$ nodes that fail for all quorums to fail. A biquorum system of the grid structure has been presented in [LWV03] where lines and columns represent two types of quorums. This biquorum system satisfies disjointness, all intersections being disjoints. In other words, provided a grid quorum system Q with $Q_1, Q_2, Q_3 \in Q$ we have $Q_1 \cap Q_2 \cap Q_3 = \emptyset$ despite every two quorums intersect. If all of the $|Q|$ nodes located on the diagonal of this grid biquorum system fail then no quorums are active anymore. More generally, the fact that these intersections are disjoint implies that intersection failures are not correlated. Despite the failure of nodes at one intersection, the quorums these nodes were part of are not active anymore, but still it may exist (independently) some quorums that intersect.

3.2 Communication Structure of Quorum Systems

Beyond the structure of quorum system, an important aspect is the way the nodes of the quorum system communicate with each other. The stronger assumption, yet more powerful one, is that every node can communicate directly with any other node. In this case, the communication structure, also called *overlay*, represents a complete graph where each node has a degree of $n-1$, where n is the system size. Specific settings make this assumption unreasonable. First, in large-scale systems with few millions of participants, the memory of a single node may not afford recording the address of the $n-1$ other nodes. Second, if the system is also dynamic then simultaneous join/leave may produce a large amount of messages, notifying new node addresses, that limited bandwidth capacity cannot handle. Finally, in wireless sensor networks, communication is physically restricted to geographical proximity, so that a node is able to communicate with other only if all nodes are located in the same transmission range.

Along with this Section we consider logical communication overlay. More precisely, we as-

sume that any node has the ability to communicate with another if it has enough information to send message that will be routed to him through the use of an underlying network layer. In other words, we assume that a given network layer is provided so that address can be resolved and information can be routed as it is done through the OSI network layer, on the Internet. Consequently, a node i simply needs the address of node j to communicate with him. Shortly, in this case we say that node i knows about node j. Section 3.4 discusses some alternative for specific settings like wireless networks.

3.2.1 Non-Adaptive Quorum Probe vs. Adaptive Quorum Probe

A *non-adaptive quorum probe* is the action of probing a quorum while all the members of this quorum are known at the beginning of the probe. The only way to probe a quorum in a single round-trip requires that the client knows all the quorum members at the time it starts to probe, as presented in Chapter 2. In contrast, an *adaptive quorum probe* is the action of probing a quorum without knowing all of its members. For example, a client may initiate an adaptive quorum probe by contacting a member of this quorum that, in turn, contacts another member of this quorum, and so on. From a different point of view, one might think at a non-adaptive probe as contacting nodes using a reactive routing whose initiating node i ignores some nodes that it has to contact: for instance, i may contact indirectly a node j it does not know about by contacting other nodes until reaching someone that knows j.

Adaptive quorum probe helps finding an active quorum in a lightweight manner. While contacting all nodes of the quorum system to probe a single quorum results a fortiori in message waste, adaptive quorum probe can progressively contact each member of an active quorum with few wasted messages. Next paragraphs present solutions that use adaptive quorum probe over a replicated set of quorums.

A tree-based quorum system with numerous intersections. Some quorum systems benefit from the power of replication to tolerate failures. The binary-tree protocol of Agrawal and El Abbadi [AE89] proposes the use of a recursive function returning either the current node plus the result of the same function applied to one child, or the result of the function applied to both children. Applying such function at the root of a k-equilibrated tree returns quorums. A quorum is thus built following a single path from the root either to the leaves or interrupting this path at some node i and duplicating it in the two subtrees rooted at node i. Any quorum intersect any other quorum in this approach. The motivation behind this quorum system is to allow an adaptive quorum probe in the presence of failure. During a quorum probe, if a node of the tree has failed, then the probe replaces the failed node by its two children in the tree: intersection is still guaranteed.

Path quorum system. The Path quorum system appeared in [NW98]. This protocol uses a grid-like structure where each cell of the grid corresponds to a node. The structure is static and a

dynamic version of this quorum system is described later on, in Subsection 3.2.2. A quorum is defined as a path traversing the grid from left to right and from the bottom to the top. Quorum probes are adaptive to remedy failures without sounding all quorums. When the client initiates the probe, it only knows a single access point of the grid. This node access point, say node i, contacts in turn a member of one of the quorums i also belongs to. If this node is failed, then node i tries contacting another member of one of the quorums it belongs to, and so on until node i finds an active node j. From this point on, node j acts similarly and so on until a whole quorum is contacted.

3.2.2 Reparation of Accumulating Failures

As previously explained, structural properties may strengthen fault-tolerance of quorum systems. However, when failures accumulate, a reconfiguration must occur to refresh the set of nodes composing the quorum system, whatever the quorum system is. The type of reconfiguration we consider in this section relies on the structure of the quorum system in use and aims at reparing the system rather than replacing it. In the following, we present several reparation techniques used in the literature.

Dynamic quorum adjustment. Herlihy presents, in [Her87], dynamic quorums to handle failures. The chief motivations for these quorums rely on transactions of distinct levels. Each level corresponds to a dedicated quorum configuration where read and write quorums intersect. Differently from the quorum reconfiguration presented in Chapter 2, configuration are dependent since write quorums of a given level must intersect all read quorums of any higher level. An inflation mechanism handles quorum failures or partitions of the quorum system in the following way: If failures disable all write quorums of level ℓ, then the transaction contacts a write quorum of the smallest level higher than ℓ that owns active quorums.

Dynamic path quorum system. The dynamic path quorum system [NW03] is a dynamic adaptation of the path quorum system in which the grid cells are replaced by Voronoi cells. The quorum system is dynamic and uses adaptive quorum probe so that quorum member can be added or removed dynamically while every point of the coordinate space must have an active responsible node. The quorum consists in traversing vertically and horizontally the coordinate space.

Dynamic and/or quorum system. The dynamic and/or quorum system [NN05] uses a tree-based structure. Similarly to the tree quorum protocol [AE90], a recursive function is applied on the height of the tree. As entitled, this function, starting from the root of the tree, consists in alternatively choosing the left child **and** the right child of the current node, or the left child **or** the right child of the current node. Conversely to other tree-based quorum protocols, inner nodes are virtual and are simply used for choosing members of a quorum, only leaves represent system nodes.

For the same tree quorum system, two alternated sequences of **and** and **or**, define the biquorum system. More precisely, the sequences starting with an **and** at the root contain all quorums of the first type whereas the sequences starting with an **or** at the root contain all quorums of the second type. This quorum system, originated in [Val84] has been made dynamic in [NN05]. It is dynamic in the sense that after a leaf fails, the tree is re-balanced so that old quorums and new ones still intersect.

3.3 Scalable Distributed Shared Memory (benefiting from Locality)

This Section defines a scalable distributed shared memory, called *Square*[1]. We focus here on the scale-shift of the number of clients in the system. Square does not contain all the system nodes but rather uses a dynamic quorum system that represents a subset of the system nodes, that is Square is able to grow depending on the request rate, by making active nodes enter the quorum system. The core structure of the quorum system provides optimal load when not altered by dynamic events. This structure adapts to failures using local reconfiguration that involves a constant number of message exchanges. The structure also adapts to load variation by shrinking and expanding. The low load, reconfigurable, dynamism, and self-adaptation makes Square a distributed shared memory especially suited for large-scale systems.

3.3.1 Trading Availability with Dynamism

Since pioneered work of [AE89], many researches focused on the robustness and availability of quorum system. The *robustness* of quorum system expresses the ability for the quorum system not to be affected by isolated failure.

While these properties were of crucial importance in failure-prone static quorum systems, availability and robustness are mostly provided by the dynamism of quorum systems. As already mentioned in Subsection 3.2.2, in dynamic quorum systems, the failures are coped with using reparation mechanisms so that failed members are replaced by new active ones.

Dynamic grid quorum system. At the core of Square, a new quorum system called the dynamic grid quorum system lies. This quorum system is similar to the grid protocol [Mae85, AV86] and the path quorum system [NW98] in that it uses a biquorum system where quorums represent lines and columns of a grid. The grid corresponds to a two-dimensional coordinate space $[0,1) \times [0,1)$ divided into cells/subregions, as rectangles in the plane. Each subregion of the grid is mapped to a node that is responsible of. In the following, we refer to the quorum system as the *memory* and to the quorum members as the *replicas*.

[1] SQUARE stands for Scalable QUorum-based Atomic memory with local REconfiguration.

The communication structure forms a torus: (i) inside the grid, nodes responsible of two abutting regions are neighbors; (ii) at the edges of the grid, nodes responsible of the two opposite regions of the same abscissae/ordinate are neighbors too. Figure 3.3 draws the torus communication graph for a dynamic grid quorum system divided into 16 subregions. The quorum system contains only a subset of the system nodes. Each client knows at least one access node inside the quorum system to which it can request an operation, and every member of the quorum system can communicate trough neighborhood, i.e., two quorum members communicate if and only if they are neighbors.

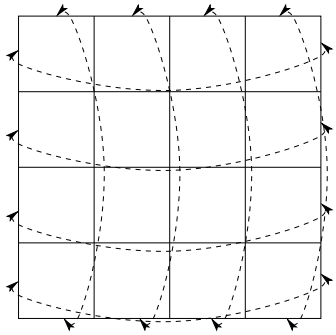


Figure 3.3: The torus communication structure of the dynamic grid quorum system. Neighbors are responsible of two abutting regions inside the grid or two opposite regions at the edge of the grid. (This later neighboring relation is depicted with dashed arrows.)

The size of the structure, i.e., the number of subregions it contains, is dynamic. Initially, only one replica is responsible for the whole space. The bootstrapping process pushes a finite, bounded set of replicas in the quorum system. Since the system is dynamic, nodes may join and leave (or fail) at arbitrary time. In order to maintain the communication overlay some adjustments follows a dynamic event using a technique used in distributed hash table maintenance [RFH+01] as described below.

- **Join event:** the joining node *i* contacts a responsible *j* of the grid. This responsible *j* splits its subregions in two halves. This split is made alternatively horizontally and vertically to keep the subregions as square as possible. Hence, if *j* has lastly split its zone vertically then it splits its zone horizontally and vice-versa. The subregion halves form two new subregions. The one with the lowest abscissae and ordinates is mapped to *j* while the other is mapped to the joining node *i*.

- **Leave event:** when a node *i* leaves the overlay, its neighbors detect the failure and one of its neighbors *i* decides, based on its unique identifier, to repair the overlay. The reparation is made so that each subregion keeps its rectangle shape and one active responsible node.

We refer to a region (or simply to a replica) r as the product of two intervals: $I_x^r = [r.xmin, r.xmax)$ and $I_y^r = [r.ymin, r.ymax)$, where *r.xmin* (resp. *r.xmax*) is the left-most (resp. right-most) abscissa of zone r, and *r.ymin* (resp. *r.ymax*) is the bottom-most (resp. top-most) abscissa of zone r. Intuitively, we define dynamic quorum sets as dynamic tiling sets, that is sets of replicas whose zones are pairwise independent and totally cover the abscissa and ordinate of the coordinate space shared by the replicas.

Definition 3.3.1 (Dynamic Quorum) *Let c be a real constant with* $0 \leq c < 1$. *The consultation quorum* $Q_{h,c}$ *is defined as the set of replicas satisfying* $\{r.ymax > c \geq r.ymin\}$. *The propagation quorum* $Q_{v,c}$ *is defined as the set of replicas satisfying* $\{r \in I \mid r.xmax > c \geq r.xmin\}$.

Theorem 3.3.1 *For any consultation quorum* $Q_{h,c}$ *and any propagation quorum* $Q_{v,c'}$, *the intersection holds:* $Q_{h,c} \cap Q_{v,c'} \neq \emptyset$.

The proof follows from the fact that it exists a node responsible for point (c, c') in the space.

Data consistency characteristic. For quorum systems whose application is data storage, some data maintenance procedures must be added to the join/leave procedure. Actually, it might happen that a joining node is not aware of the data maintained by the quorum it joined. Therefore, quorum-based read and write operations may no longer satisfy consistency. In addition to updating its set of neighbors, a joining node has also to update its state according to the state of its neighbors.

Note that if the joining node acquires a data that is not yet replicated at all nodes of a single quorum, validity is not violated. Moreover, the newly arrived node does not need to maintain a replica of the data owned by each quorum, but simply the most up-to-date data of the object it encounters. More technically, the join event in the dynamic grid quorum system requires one message round containing the current data between the joining node and its neighbors on the overlay. Later on, we detail this procedure as part of the Square algorithm specification.

3.3.2 Congestion Avoidance

A crucial metrics of quorum system is load. (Load is formally defined below.) This metrics is responsible of two issues that may result in important delays and message loss: congestion and overload. Congestion happens when load is applied at some specific locations of the memory whereas overload happens when the system receives too many requests that cannot be handled. More specifically, we consider that the memory is not *overloaded* if at least one member is not overloaded and that a quorum is overloaded when at least one of its replica is overloaded. (Note that if at least one node is overloaded, then other members of the two quorums it belongs to are often overloaded too.) Coping with congestion requires load-balancing while coping with overload requires more capacity. Here, we especially focus on load-balancing to face congestion. Overload

issue is studied in Subsection 3.3.4. To better understand how congestion apply to a quorum system, we first have to define the access strategy over a quorum system and its resulting load.

An *access strategy* over a quorum system is a probability distribution function that maps each quorum with its probability of being accessed such that the sum of probabilities over all quorums equals 1.

Definition 3.3.2 (Access Strategy) *An* access strategy ω *for a set system* $\mathcal{S}$ *is a probability distribution on the elements of* $\mathcal{S}$ *at time* t*. That is,* $\omega : \mathcal{S} \times T \rightarrow [0,1]$ *satisfies:* $\sum_{s \in \mathcal{S}} \omega(s) = 1$.

Next, we restate the definition of load that appeared in [NW98]. We first introduce the load of an element i as the sum of the access probabilities of the quorums i belongs to. For instance, if node i belongs to two quorum Q_1 and Q_2 and the access probability is ω, then the load of i is $\mathcal{L}_\omega(i) = \omega(Q_1) + \omega(Q_2)$.

Definition 3.3.3 (Load of an element) *Given an access strategy* ω *for a quorum system* Q *over a universe* U*, the* load *of an element* $i \in U$ *is*

$$\mathcal{L}_\omega(i) = \sum_{Q \in Q : i \in Q} \omega(Q).$$

Second, we restate the definition of the load of a quorum system as the load of the maximally loaded element minimized over all the possible access strategies.

Definition 3.3.4 (Load of a quorum system) *The* load *of a quorum system* Q *is*

$$\mathcal{L}(Q) = \min_{\forall \omega} \left\{ \max_{i \in Q} \{\mathcal{L}_\omega(i)\} \right\}.$$

In some sense the load of a quorum system indicates how this quorum system balances the load over its distinct members. For example, a highly loaded quorum system does not balance the load well, since it means that even for the best strategy there still exists a highly loaded element. In contrast, a poorly loaded quorum system would accept a strategy where all elements can be not much loaded. To illustrate the former example, one can refer to the star quorum system presented in Subsection 3.1.1: Despite the access strategy, there exists a highly loaded element, which translates into a highly loaded quorum system. For the latter example, we explain how the dynamic grid quorum system achieves optimal load hereafter.

Avoiding congestion in the dynamic grid quorum system. The dynamic grid quorum system presents desirable features to face congestion. First, if any operation accesses both a consultation quorum and a propagation quorum, then the load is well-balanced over the quorum system. This is due to the fact that the quorum system resulting from the union of any consultation quorum and

any propagation quorum has optimal load.[2] Second, each element of the dynamic quorum system participates equally in handling the load when all quorums are uniformly accessed. As a result, balancing the load among quorums of the system balances the load among all members. First, we describe these two features in detail. Then, we present how to balance the load over the dynamic grid quorum system to avoid congestions.

The dynamic grid quorum system is similar to the grid quorum system. In case enough nodes join the structure by requesting uniformly the nodes present, it is easy to see that all subregions have equal size. In this case, the unions of any consultation quorum and any propagation quorum of the dynamic grid quorum system forms a grid quorum system. More formally, for any $DG = \langle \mathcal{S}_1, \mathcal{S}_2 \rangle$, the quorum system $\mathcal{Q} = \{Q = Q_1 \cup Q_2 : \forall Q_1 \in \mathcal{S}_1, \forall Q_2 \in \mathcal{S}_2\}$ is a grid quorum system. The following analysis shows that the grid quorum system has optimal load.

Lemma 3.3.2 *If $\mathcal{Q}$ is a grid quorum system over the universe U, then $\mathcal{L}(\mathcal{Q}) = \frac{2\sqrt{n}-1}{n}$ and it is optimal.*

Proof. Let n be the number of nodes in U and let y be a mapping of any node of U to $\frac{1}{n}$. By the quorum definition of $\mathcal{Q}$, $|Q| = n$ and any $u \in U$ belongs to $2\sqrt{n}-1$ quorums. Consequently, $\mathcal{L}(\mathcal{Q}) = \frac{2\sqrt{n}-1}{n}$. Observe that $\sum_{u \in U} y(u) = 1$ and for any $Q \in \mathcal{Q}$ we have $\sum_{u \in Q} y(u) = \mathcal{L}(\mathcal{Q})$. Following this observation, Proposition 4.7 of [NW98] states that this load is optimal. □

Homogenizing quorum systems. The disjointness property previously defined (Definition 3.1.2) is not sufficient to ensure that the load can be well balanced among participants. For instance, when most of the nodes of the quorum system are not part of any intersection, these nodes do not participate in handling the load of the system: removing or adding these nodes to the quorum system does not change the load of the quorum system, like if those nodes were useless.

Garcia-Molina et al. [GMB85] defined a relation between coteries to identify which coterie of the two is more general than the other coterie. If for any quorum H of coterie C_2 there exists a quorum G of another coterie C_1 that is included in H, then C_1 *dominates* C_2. Built upon this relation, a *nondominated coterie* is a coterie that is dominated by no coteries. This definition allows to identify a coterie in which each quorum element is necessary for intersection with other quorums and where each intersection contains a sufficient amount of nodes too. As an extension of the domination relation, a bicoterie is dominated by another bicoterie or it is nondominated as defined in [MMR92]. We directly apply the definition of [MMR92] to biquorum systems. (Observe that a nondominated biquorum system is also a nondominated coterie.)

Definition 3.3.5 (Biquorum Dominance) *A biquorum system $B_1 = \langle C_1, C_1' \rangle$ under U dominates another biquorum system $B_2 = \langle C_2, C_2' \rangle$ under U if and only if:*

[2] Here load stands for the Definition 3.3.4.

1. $B_1 \neq B_2$.
2. *For any* $H \in C_2$, *there exists* $G \in C_1$ *such that* $G \subseteq H$.
3. *For any* $H \in C'_2$, *there exists* $G \in C'_1$ *such that* $G \subseteq H$.

Nondominance does not provide us with sufficient guarantee to minimize load. It is noteworthy that all quorums of a nondominating biquorum system have a minimal number of elements: removing one element would violate intersection property with another quorum. However, this property does not provide us with sufficient guarantee to balance the load. Take as an example the following nondominated biquorum system $\langle\{\{1\},\{2,3\}\},\{\{1,2\},\{1,3\}\}\rangle$ where quorums of the first type $\{1\}$ and $\{2,3\}$ are used as consultation quorums and quorums of the second type $\{1,2\}$ and $\{1,3\}$ are used as propagation quorums. Then an ideal access strategy that minimizes load maps quorum $\{1\}$ to 0, and quorums $\{2,3\}$, $\{1,2\}$, and $\{1,3\}$ to $\frac{1}{3}$, meaning that propagations would occur two times more frequently than consultation. However, frequency of consultation and propagation are ruled by the operation specification and the operation frequency imposed by the environment. To encompass the fact that quorum members play different role, thus, having unequal responsibilities, we propose a new constraint on the quorum system. This constraint, *completeness*, forces any element to have the same responsibility in quorums.

Definition 3.3.6 (Completeness) *A quorum system Q over U satisfies* completeness *if and only if, for any $u \in U$, $|\{Q \in Q : u \in Q\}| = 2$.*

An interesting fact is the following: despite how constraining they are, disjointness and completeness together provide more scalable coteries than nondominance. First-of-all, observe that a quorum system that satisfies completeness is also a coterie, since completeness implies minimality. (Recall that a quorum system Q verifies minimality if and only if, for all $Q_1, Q_2 \in Q$, $Q_1 \not\subset Q_2$.) An interesting characteristic of complete and disjoint biquorum systems $B = \langle Q_1, Q_2\rangle$ is that each quorum has size $|Q_1| + |Q_2| - 1$. This is straightforward from the fact that all intersections contain a single element and are disjoint. The drawback is that they may accept a lower number of quorums than quorum systems that do not satisfy these constraints. Comparing with the path quorum system that does not verify completeness, an adaptive probing of the path quorum system can switch from probing one quorum to probing another quorum (without backtracking). This is due to the fact that there are many quorums. Alternatively, dynamic quorum systems simply replace failed nodes by active ones without modifying the quorum probe.

The dynamic grid biquorum system guarantees completeness property under some circumstances. Since this system is dynamic, the structure may transiently not be equilibrated. However, in case the grid structure is equilibrated the dynamic grid biquorum system verifies both completeness and disjointness.

A remaining issue is to force the load to apply equally to all quorums. As previously investigated in Definition 3.3.4, the load of a quorum system is computed independently of any access

strategy. In fact, the result is the load of the busiest element induced by the best possible strategy that minimizes it among all possible access strategies. However, from a practical standpoint, a quorum system is always accessed following a strategy depending on clients behaviors but rarely the best one. Moreover, in general, determining the best possible strategy to access the quorum system is a difficult task.

In order to consider the variation of load due to specific distribution of requests applied to quorums and to replicas, we propose a more practical definition of load, called the *workload* of a quorum member.

Definition 3.3.7 (Workload of a quorum element) *Let Q be a quorum system over universe U receiving operation requests at any time from the environment. The* workload w_i *of a given member i of any quorum of Q is the number of requests i has received but has not treated yet.*

For the purpose of remedying workload increase due to skewed distributions applied to the dynamic grid quorum system, we propose a mechanism to balance the workload over all replicas. Assume a large-scale system where numerous clients try to access a large dynamic grid quorum system by executing operations on some nodes without knowing each other, and without knowing replicas. Assume also that every quorum member treats the requests at the same rate. It is highly probable that requests of clients are heterogeneously scattered over the replicas. To circumvent the resulting unbalanced workload, we propose that in any quorum, overloaded replicas share their workload with replicas of distinct quorums, to improve workload-balancing. Next paragraph details this mechanism.

Thwarting the overlay to balance the workload. The chief aim of *Square* is to provide a shared memory for dynamic environments. As said before, clients can access an atomic object of *Square* by invoking a read or a write operation on any replica this client knows in *Square*. This invocation is done through the **Operation** procedure. Pseudocode of this procedure is shown in Algorithm 4. All the information related to this request are described in parameter $\mathcal{R}$. For instance, if the client requests a read operation then $\mathcal{R}.type$ is set to read, and value $\mathcal{R}.value$ is the default value v_0. For a write operation, *type* is set to write and *value* is the value to be written. The other subfields of $\mathcal{R}$ are discussed below.

When such a request $\mathcal{R}$ is received by a replica, say i, i first checks whether it is currently *overloaded* or not. Recall that a replica is overloaded if and only if it receives more requests than it can currently treat. If i is overloaded then it conveys the read/write operation request to a less loaded replica. This is accomplished by the **Thwart** process (cf. Line 20). Conversely, if i is not overloaded then the execution of the requested operation can start and i becomes the $\mathcal{R}.initiator$ of this operation. Thus, i starts the traversal process: First, i **Consults** a consultation quorum to learn about the most up-to-date value of the object and an associated tag (Line 24). As explained later, this results in updating the local value-tag pair. From this point on, if the operation is a write then the counter of the request tag, $\mathcal{R}.tag$, is set to the incremented local one (Line 27) and the request

tag identifier is set to i to break possible tie. Second, i **Propagates** in the propagation quorum starting at i the new value and its associated tag to ensure this value will be taken into account in later operation executions. In case of write, the $\mathcal{R}.value$ propagated is the value to write, initialized by the client; while in case of a read, it is the *value* previously consulted (Line 32). Finally, this consulted value is returned to conclude the read operation, as shown at Line 35.

Observe that, if the operation is a read and the consulted value has already been propagated twice at this replica, then the operation completes just after the **Consult** without requiring a **Propagate** phase (see the paragraph on the improvement on the probe complexity of a read operation hereafter).

The Thwart aims at balancing the workload among participants. This mechanism, as depicted in Algorithm 5, relies essentially on two procedures, called **Thwart** and **Forward**. The **Thwart** is executed if i receives an operation request while it is overloaded (cf. Line 20 of Algorithm 4). This mechanism checks the workload of each quorum until it finds a non-overloaded one. For this purpose a sequence of quorum representatives, and located on the same diagonal axis, are contacted in turn, as shown in Figure 3.4. Each of these representatives is a replica subsequently denoted the target of the request $\mathcal{R}.target$.

It is noteworthy that contacting subsequent replicas located on a diagonal axis leads to contacting all quorums. Furthermore, contacting only one representative per quorum is sufficient to declare that this quorum is overloaded or not. By definition, these replicas are not necessarily neighbors, and thus, an intermediary replica j is simply asked to **Forward** the thwart to $\mathcal{R}.target$ without checking its workload. Because of asynchrony, although a replica i sends a message to its neighbor j, at the time j receives the message j might have modified its state and might no longer be the neighbor of i. (Because a new zone may have been created between nodes i and j.) To encompass this, a $\mathcal{R}.point$ indicates the final destination in the overlay coordinate space and a replica **Forwarding** or **Thwarting** first checks whether it is still responsible of this point, as expressed Lines 13 and 3.

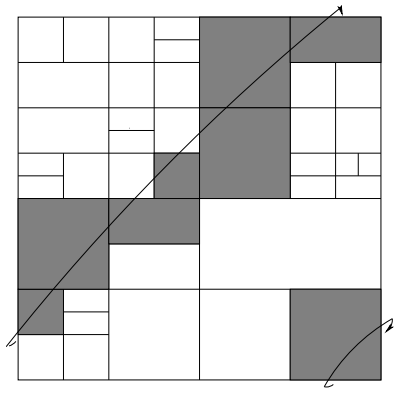


Figure 3.4: The thwart mechanism.

Algorithm 4 Read/Write Operation

```
 1: State of node i:
 2:     status ∈ {node, replica}, the status of the node initially replica
 3:     available ∈ {true, false}, a boolean initially true
 4:     tag, the tag with fields
 5:         counter, the number of write operations that precedes this writing
 6:         id, the identifier of the writer that wrote this tag
 7:     R, the request with fields
 8:         starter, the node that started to forward the request by thwarting
 9:         value, the value to write or ⊥
10:         tag, the tag encountered so far
11:         initiator, the node that started treating the request by traversing
12:         target, the node targeted by the request
13:         point, the next point (in the overlay) where the request is forwarded

14: Prerequisite Functions:
      first-time-traversal() indicates whether i is the starting point of the traversal.

15: Operation(R):
16:     if available then
17:         if overloaded then
18:             if first-time-thwart(R) then
19:                 R.starter ← i
20:             Thwart(R, i)
21:         else
22:             if first-time-traversal(R) then
23:                 R.initiator ← i
24:             Consult(R, i)
25:             if R.type = write then
26:                 R.tag ←
27:                     ⟨tag.counter + 1, i⟩
28:                 Propagate(R, i)
29:                 Acknowledge(R)
30:             else
31:                 R.tag ← tag
32:                 R.value ← value
33:                 if R.value has not been propagated twice then
34:                     Propagate(R, i)
35:                 Return(value)
```

3.3.3 Read and Write Operation using Local Knowledge

Probe complexity. Probe complexity is defined alternatively with distinct failure models. Peleg and Wool define the probe complexity of quorum systems in [PW02] as the complexity to contact either an active quorum or to obtain sufficient hints proving the lack of such an active quorum.

Algorithm 5 The Thwart Protocol invoked by the Read/Write Operation

Prerequisite Functions:
next-point-on-diagonal() returns the replica identifier responsible of the extreme north-east point of the zone of i.
closest-neighbor-of($\mathcal{R}$.*point*) returns the neighbor that is responsible of the coordinate point given as an argument.

Thwart($\mathcal{R}$, i):
  if $\mathcal{R}$.*target* $= i \wedge \mathcal{R}$.*point* $\in$ *zone* **then**
    if $\mathcal{R}$.*starter* $= i$ **then**
      Expand()
    else if overloaded **then**
      $\mathcal{R}$.*point* $\leftarrow$ *next-point-on-diagonal*()
      $j \leftarrow$ *closest-neighbor-of*($\mathcal{R}$.*point*)
      Forward($\mathcal{R}$, j)
    else
      Operation($\mathcal{R}$)

Forward($\mathcal{R}$, i):
  if $\mathcal{R}$.*point* $\in$ *zone* **then**
    for $j \in$ *neighbors* **do**
      if $\mathcal{R}$.*point* $\in j$.*zone* **then**
        $\mathcal{R}$.*target* $\leftarrow j$
    Thwart($\mathcal{R}$, $\mathcal{R}$.*target*)

Their definition relies on the quorum system structure and assume an adversarial failure model. Later on, Nadav and Naor [NN05] define the probe complexity as the communication and time complexity of accessing a quorum. This definition relies on the communication overlay in use and assumes a randomized failure model.

Failure detection was the responsibility of the quorum probe mechanism in static quorum system while it becomes the responsibility of quorum system adjustment in dynamic quorum systems. Consequently, while probe complexity as defined by Peleg et al. [PW02] relies on the cost of finding a quorum of failed nodes in static systems, probe complexity relies only on the cost to find an active quorum in dynamic systems. Indeed, dynamic quorum systems [NW05, AM05, NN05] aim at reparing the quorum system locally after failure detection. In dynamic quorum systems, a leave is handled similarly as a failure and such an event is detected locally by the neighbors of the node involved. Consequently, failure detection is no longer part of the quorum probe mechanism.

As presented in Chapter 2, the elementary phase of a read or write operation consists in contacting a quorum. Hence, quorum probe is of crucial importance in the design of distributed shared memory.

However, there is a tradeoff between time complexity and communication complexity. The optimal time complexity to probe a quorum is two message delays since the client needs at least

one round-trip to contact a quorum. The solution proposed in Chapter 2 presents already optimal time complexity. The tricky point relies on the message complexity associated with such result. Indeed to provide a fast probe, the client needs to know an active quorum. This induces that either every client is notified as soon as any single failure occurs or it contacts all quorums to ensure that at least one is active. In both cases, the message complexity is unaffordable in a large-scale system where bandwidth resource is limited. Next, we investigate the best compromise between communication and time complexity in a large-scale environment.

Algorithm 6 The Traversal Protocol invoked by the Read/Write Operation

Prerequisite Functions:
next-vert-nbr() returns the next vertical neighbor in the sense depending on the last message receipt, to continue the propagation. If it received south-directed (resp. north-directed) message, it sends it in the south (resp. north) sense.
other-vert-nbr() returns the next vertical neighbor in the opposite sense of the last message sending.
next-hor-nbr() returns the next horizontal neighbor in the sense depending on the last message receipt, to continue the consultation.

```
Consult(R, i):
    if available then
        R.tag ← max(tag,
            R.tag)
        R.value ← max(value, R.value)
        if ¬(R.initiator = i) then
            Consult(R, next-hor-nbr())
        else if i has already consulted then
            End()

Propagate(R, i):
    if available then
        tag ← max(tag,
            R.tag)
        value ← max(value, R.value)
        if ¬(R.initiator = i) then
            Propagate(R, next-vert-nbr())
        else if i has already propagated then
            End()
        else
            Propagate(R, other-vert-nbr())
```

Reading and writing by traversing the overlay. The Traversal, presented in Algorithm 6, consists in two procedures as shown in Figure 3.5(a), called respectively **Consult** and **Propagate**; the former consults the value and tag of a whole consultation quorum whereas the latter one propagates a value and a tag to a whole propagation quorum. Each of these procedures is executed (only if *i*

is *available*, i.e., *i* is not involved in a dynamic event) from neighbor to neighbor by forwarding the information about the request $\mathcal{R}$, until both quorums (i.e., the consultation quorum and propagation quorum) have been traversed. The traversal ends once the initiator of the traversal receives from its neighbor the forwarding request it initially sent (i.e., the "loop" is complete). When **Consult** or **Propagate** completes, the initiator *i* gets back the message (Lines 10 and 19), knowing that a whole quorum has participated. From this point on, *i* can continue the operation execution. That is, by directly sending the response to the requesting client if operation $\mathcal{R}$ is complete otherwise by starting a **Propagate** phase.

There are two differences between **Consult** and **Propagate**. First, the **Consult** gathers the most up-to-date value-tag pair of all the consultation quorum replicas (Line 6) whereas the **Propagate** updates the value-tag pair at all replicas of the propagation quorum (Line 15). Second, the **Consult** contacts each member of the quorum once following a single direction (Line 8), while the **Propagate** contacts each member of the quorum twice with messages sent in both directions (Lines 17 and 21). Consequently, if the value has been propagated twice at node *i*, then *i* knows that the value has been propagated at least once to every other replica of its propagation quorum. This permits later read operation to complete without propagating this value once again.

Improvement on the probe complexity of a read operation. Not only, the traversal is lock-free compared to [AGGV05], but it does not require the confirmation phase of [DGL+05, CGG+05], while proposing fast read operations. This results directly from the adaptiveness of our traversal mechanism. Minimizing atomic read operation latency suffers some limitations. Indeed, to guarantee atomicity two subsequent read operations must return values in a specific order. This problem has been firstly explained in [Lam86] as the new/old inversion problem. That is, when a read operation returns value *v*, any later (non-concurrent) read operation must return *v* or a more up-to-date value. *Square* proposes read operations that may terminate after a single phase, solving the aforementioned problem without requiring locks or additional external phase. For this purpose, the **Consult** phase of the read operation identifies if the consulted value has been propagated at enough locations. If the value *v* has not been propagated at all members of a propagation quorum, a **Propagate** phase is required after the end of the **Consult** phase and before the read can return *v*, otherwise a later read might not **Consult** the value. Conversely, if a value *v* has been propagated at a whole propagation quorum, then any later **Consult** phase will discover *v* or a more up-to-date value, thus the read can return *v* with no risk of atomicity violation.

The solution is presented in Figure 3.5(b) and relies on interleaving messages during the **Propagate** phase. This phase is executed from neighbor-to-neighbor. Figure 3.5(b) presents a propagation quorum of the torus grid as a ring where each circle models a replica and a link models a relation between two neighbors. The black circle represents the initiator of the **Propagate** phase. Unlike the **Consult** phase, the **Propagate** phase starts by two message sendings: one message in each of the two senses (north and south senses in the torus). Those messages are conveyed in parallel from neighbors to neighbors until the initiator receives them back.

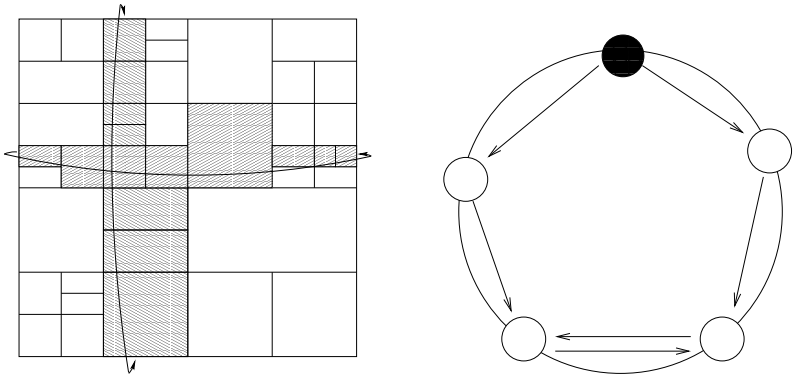


Figure 3.5: (a) The traversal mechanism traverses the overlay either horizontally following a consultation quorum in one sense or vertically following a propagation quorum in both senses. (b) The **Propagate** Phase consists in following the propagation quorum, traversing the overlay, in the two senses. Traversing the torus is represented here by a ring.

The idea is simple: when a replica of the ring receives a first message it simply updates its local value-tag pair with the one of the message; when the replica receives a second message it deduces that all the members of a propagation quorum have updated their local pair to the propagated one. During a **Consult** phase of a read operation, if the (most up-to-date) consulted pair $\langle v', t' \rangle$ has been found at a replica r that has received only one message containing $\langle v', t' \rangle$, then a **Propagate** phase must occur before the end of the read operation. If replica r has received two messages propagating $\langle v', t' \rangle$, then the read can terminate immediately after the **Consult** phase. For instance, in Figure 3.5(b) propagation is ongoing: if r is one of the two bottom replicas, then the read operation can return immediately, otherwise the read must **Propagate**.

3.3.4 Self-Adaptiveness to Unpredictability

Generally, large scale system includes individual participants that act on their own. This behavior is unpredictable from the system standpoint. Sometimes, to face unpredictability, the system must adapt. For instance, consider that many participants request in the meantime the same object, to handle this workload burst the system should increase the amount of capacity. Conversely, if the workload drops down because of participant inactivity, a large capacity is useless resulting in resource waste.

Adaptiveness is thus a desirable feature especially for large-scale system where individual behaviors are unpredictable. Another approach would have been to use self-stabilizing quorum system [Bel99], however, if dynamic events of load bursts occur continuously, then the quorum system may never self-stabilize. An interesting research work in this context though, would be to investigate local self-stabilization of quorum systems. In the following, we propose a quorum system that adapts its amount of resources depending on its workload.

Adapting the quorum system structure. Here, we present self-adaptive mechanisms of *Square*. If a burst of requests occurs on the whole overlay the system needs to **Expand** by finding additional resources to satisfy the requests. Conversely, if some replicas of the overlay are rarely requested, then the overlay **Shrink**s to speed up rare operation executions. Finally, when some replicas leave the system or crash, then a **FailureDetection** requires some of the replicas around the failure to reconfigure. Those three procedures appear in Algorithm 7.

For some reasons (e.g., failure) a replica might leave the memory without notification. Despite the fact that safety (atomicity) is still guaranteed when failures occur, it is important that the system reconfigures. To this end, we assume a periodic gossip between replicas that are direct neighbors. This gossip serves a heartbeat protocol to monitor replica vivacity. Based on this protocol, the failure detector identifies failures after a period of inactivity. When a failure occurs the system self-heals by executing the **FailureDetection** procedure: a takeover node is deterministically identified among active replicas according to their join ordering, as explained in [RFH+01]. This replica takes over the responsibility region that has been left, it reassigns a constant number of responsibility zones to make sure that each region has its responsible replica, and it notifies its neighborhood before becoming newly *available*.

Algorithm 7 The Adjusting Primitives invoked locally or by the Thwart Protocol

```
 1: Expand:
 2:     available ← false
 3:     j ← FindExternalNode()
 4:     ActiveReplication(j)
 5:     ShareLoad(j)
 6:     NotifyNeighbor(i)
 7:     NotifyNeighbor(j)
 8:     availabale ← true

 9: Shrink:
10:     NotifyNeighbor(i)
11:     status ← node

12: FailureDetection(j):
13:     available ← false
14:     TakeOver(j)
15:     NotifyNeighbor(j)
16:     availabale ← true
```

Two other procedures, namely **Expand** and **Shrink** are used to keep a desired tradeoff between workload and operation complexity. When the number of replicas in the memory diminishes, fault tolerance is weakened and the overlay is more likely overloaded. Conversely, if the overlay quorum size increases, then the operation latency raises accordingly. Therefore, it is necessary to provide adaptation primitives to maintain a desired overlay size. The **Shrink** procedure occurs when a node i is underloaded (i.e., i does not receive enough requests since a sufficiently long period of

time). If this occurs, *i* locally decides to give up its responsibility, to leave the overlay, and to become a common node (i.e., a node that does not belong to the memory). Conversely, an **Expand** procedure occurs at replica *i* that experienced an unsuccessful thwart. In other words, when the thwart mechanism started at *i* fails in finding a non-overloaded replica (i.e., the thwart turns around the memory without finding a non-overloaded replica), then *i* decides to expand the overlay. From this point on, initiator *i* becomes *unavailable* (preventing itself from participating in traversals), chooses a common node *j* (i.e., a node which does not belong to *Square*), and actively replicates its tag and value at *j*. From this point on, *j* becomes a replica, *i* shares a part of its own workload and responsibility zone, and *j* and *i* notify their neighbors before they become newly *available*.

3.3.5 Correctness Proof of a Scalable DSM

To show that Square emulates a distributed shared memory, we first show, that Square implements an atomic object. Then, we show that the algorithm terminates under reasonable assumptions.

Safety proof. The following theorem shows the safety property (i.e., atomicity) of our system. The proof relies essentially on the fact that tags monotonically increase and on quorum intersection property.

Theorem 3.3.3 *Square implements an atomic object.*

Proof. First, we show that the tag used in a successful operation is monotonically increased at some location. In absence of failures, it is straightforward. Assume now that replica *i* leaves the memory and that a replica *j* takes over *i*'s zone after a **FailureDetection** event or *j* receives an **Expand** order: *j* stop being *available* until it exchanges messages with its new neighbors (by NotifyNeighbors event), catching up with the most up-to-date value.

Second, we show that operation ordering implied by tags respects real-time precedence. A write operation **Propagates** its tag in any case while a read **Propagates** it if it has not been propagated yet. That is, a whole quorum-column is aware of the tags of ended operation. All operations contain a **Consult** phase, and by quorum intersection (cf. Theorem 3.3.1), discover the latest tag. Because each written tag is unique and monotonically increased, writes are totally ordered and since the value is always associated with its tag object specification is not violated. □

Liveness. Here we show that our algorithm terminates under sufficient conditions. In order to allow the algorithm to progress, we first make a series of assumptions.

1. First, we assume that a local perfect failure detector [CHT96] is available at each replica. Such a low level mechanism, available in CAN, enables a replica to determine whether one of its neighbors has failed (i.e., crashed) by periodically sending heartbeat messages to all its neighbors. Here, we assume a perfect failure detector but we claim that a weaker failure detector can achieve the same result (e.g., trusting failure detector [DGFGK05]).

2. Furthermore, as far as liveness is concerned, we are primarily interested in the behavior of *Square* when communication is reliable and failures are not concentrated on a same neighborhood. This leads to the following environmental properties: *i) neighbor-failure*: between the time a replica fails and the time it is replaced, none of its neighbors fail; and *ii) failure-spacing*: there is a minimal delay between two failures occurring at the same point in the memory.

3. Finally, we assume that clients can act infinitely often, and concurrently. However during a finite period of time, the level of concurrency is finite. This model is often referenced in the literature as the *infinite arrival process with finite concurrency* model [MT00]. This model limits the number of expand that may occur in a bounded period of time. With no such an assumption, continuous expansions of the overlay would make overlay thwart or traversal impossible.

Theorem 3.3.4 *Eventually, every operation completes.*

Proof. First, we show that a sent message is eventually received. Let i and j be two neighbors and j fails while i sends a message. Using its failure detector i will discover j's failure and a replica j' will take over j's zone. By *neighbor-failure* and *failure-spacing* assumptions, the next message from i to j' will be successfully received.

Now, we show that the traversal and the thwart mechanisms terminate. We consider the worst case scenario of the thwart: the thwart wraps around the entire torus. First, observe that the overlay is a torus and the sense of subsequent messages does not change: east, north, south or diagonal. Second, by the *infinite arrival with finite concurrency* model we know that the number of **Expand** events during a finite period of time is finite. This implies that the number of replicas to contact during a traversal or a thwart is finite and both mechanisms converge successfully. □

Theorem 3.3.5 *Infinitely often the memory is not overloaded.*

Proof. By the *infinite arrival with finite concurrency* model, the level of concurrency is bounded during a period of time sufficiently long. From the above theorem, operations terminate. Thus eventually, the workload on each replica i does not increase, i.e., i is not overloaded, which makes the atomic memory not overloaded. From the *infinite arrival with finite concurrency* model, these periods of time occur infinitely often. Thus infinitely often the memory is not overloaded. This completes the proof. □

3.3.6 Peer-to-Peer Simulation Study

This section presents the results of a simulation study performed through a prototype implementation of *Square*. The aim of simulations is to show *Square* properties: self-adaptiveness, scalability,

workload-balancing, and fault-tolerance. The prototype is implemented on top of a peer-to-peer (p2p) simulator, namely Peersim [JMB04]. Peersim is a simulator especially suited for large-scale systems. We used its event-based simulation mode in order to simulate asynchronous communication and independent node activities.

Environment. We simulate a p2p system containing 30,000 nodes. We recall that this is the maximum number of nodes that can be potentially added to the overlay/memory. As we show, the actual number of nodes in the memory during simulation is much lower. Here we describe the parameters of the simulator:

- We lower bound the message delay between nodes to 100 time units (i.e., simulation cycles) and we upper bound it to 200 time units.
- Any replica has to wait 1500 time units without receiving any request before deciding to leave the memory (**Shrink**).
- Once in every period of 2000 time units, replicas look at their buffer and treat the buffered requests, deciding to forward them (**Thwart**) or to execute them (**Traversal**).
- We send from 500 to 1000 operation requests onto the memory every 50 time units. The exact number of operation requests chosen depends on each of the following experiments.
- Each of the requested operations is a read operation with probability 0.9 and a write operation with probability 0.1.
- The request distribution can be uniform or skewed (i.e., normal). Since the results obtained with the two distributions do not present significant differences we present only those obtained with uniform distribution.
- We observe the memory evolution every period of 50 time units starting from time 0 up to $70,000$. Each curve presented below results, when unspecified, from an average measurement of 10 identically-tuned simulations.

In all experiments, except otherwise mentioned, requests are issued at some rate during a fixed period, after which the requests stop. To absorb the workload induced by the requests, the overlay replicates the object at nodes of the system that are not yet in the memory, as specified in the protocol. This self-adaptiveness occurs until the memory reaches an acceptable configuration satisfying the tradeoff between capacity and latency. An *acceptable configuration* is a configuration where the memory is neither overloaded, nor underloaded. This happens when some replicas of the overlay shrink while other expand. More specifically, this occurs between the first time the memory size decreases and the last time the memory size increases for a given fixed rate.

Self-adaptiveness. Figure 3.6 reports the number of nodes in the memory versus time. In particular, the solid line indicates the evolution of the memory size along time, showing the adaptiveness of Square to a constant requests rate. In this figure, the memory reaches the acceptable configuration at time 9350, while the memory leaves the acceptable configuration at time $49,200$. Let us

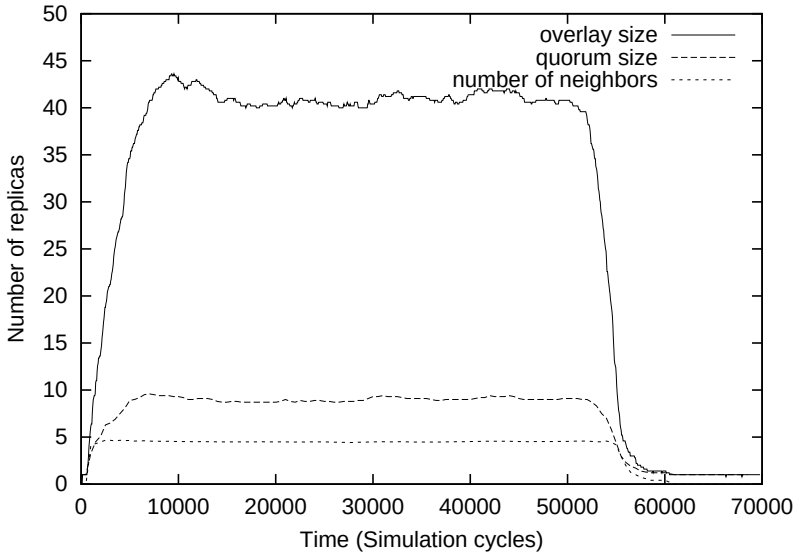


Figure 3.6: Memory size, quorum size, and number of neighbors.

focus on the three resulting time intervals. Before time 9350, the memory grows quickly and its growth slows down while converging to the acceptable configuration. Then, the small oscillation in the acceptable configuration is due to few nodes either leaving the memory (**Shrink**) or joining it as replicas (**Expand**). This shows how *Square* is able to tune the capacity with respect to the request workload. After time 49,200, the memory stops growing and when the last operations are executed, workload decreases drastically causing a series of memory shrinks until one node remains. Recall that, during all three phases, although operation requests can be forwarded to other replicas, every operation is successfully executed by the memory, thus preserving atomicity.

Figure 3.7 shows the adaptiveness of the memory to abrupt changes in workload. The vertical intervals indicate the error margin at some points of the curve. We simulate a burst of workload at time 23,000 where the request rate is multiplied by 2. Then requests are stopped at time 46,000. We clearly see that the memory is reactive and quickly self-adapts to face workload variation: the memory size grows right after the burst (i.e. it is multiplied by 1.4) and shrinks right after requests stop (i.e. divided by 1.2), while recovering a steady progress.

Scalability. The dotted line in Figure 3.6 plots the evolution of the average number of neighbors of each node along time and depicts an interesting result. We recall that two replicas are neighbors if they are responsible of two abutting zones and notice that when the memory contains a single node then the number of neighbors is 0. Even though the number of zones keeps evolving, the average number of neighbors per replica remains constant over time. Comparing to an optimal grid containing equally sized zones, the result obtained is similar: we can see that the number of

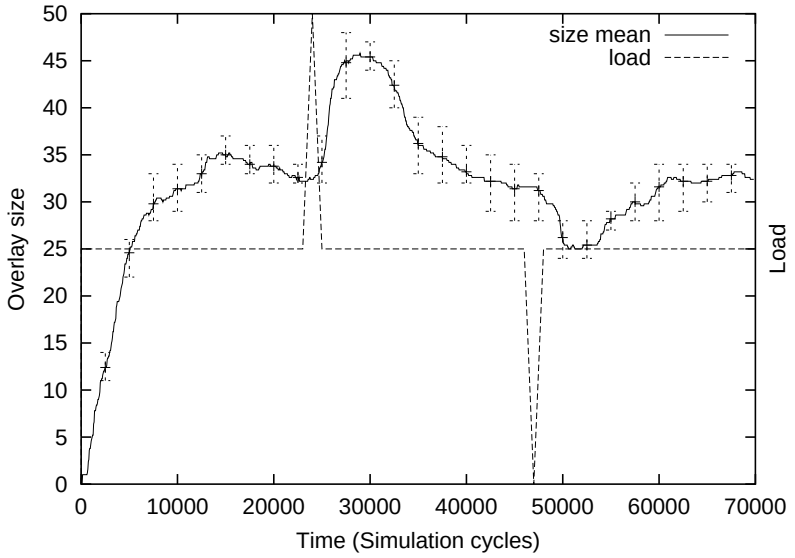


Figure 3.7: Self-adaptiveness in face of bursts of workload.

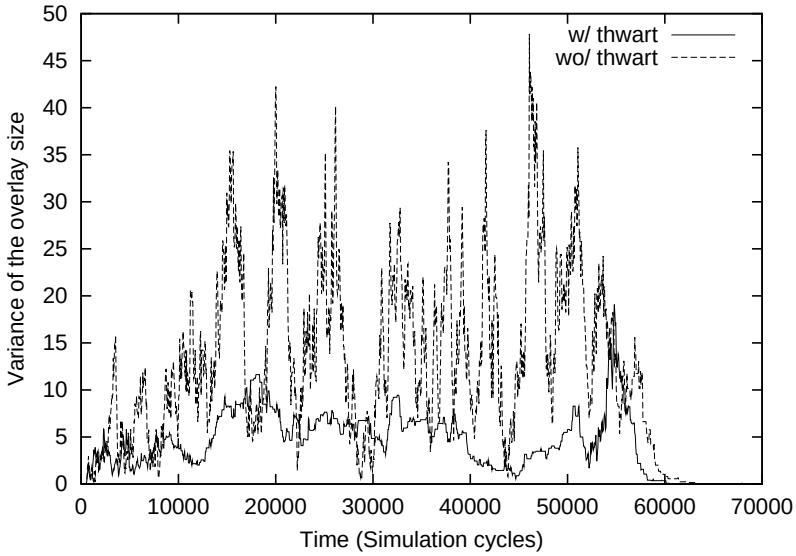


Figure 3.8: Thwart impact.

neighbors is less than 5 while in the optimal case it would be exactly 4. We point out again that this behavior is not exclusively due to the uniform distribution of requests but it is also obtained with a skewed distribution because the thwart balances the load. Since only a local neighborhood

of limited-size has to be maintained, the reconfiguration needed to face dynamism is scalable.

Load-balancing. The main objective of the thwart mechanism is to balance the workload among nodes. In order to highlight the effects of the thwart, we ran 5 different executions of the simulations, and computed the variance of the memory size. Results are reported in Figure 3.8. The dashed curve refers to executions where we disabled the thwart process (i.e., when a node is overloaded while it receives requests it directly expands the memory without trying to find a less-loaded replica of the memory), while the solid curve refers to executions with the thwart enabled. This simulation shows that the variance of the memory size is strongly reduced by the thwart mechanism. Without the thwart, expansion might occur while a part of the memory is not overloaded, that is, the replicas become rapidly heterogeneously loaded. This phenomenon produces a strong variation in the memory size: many underloaded replicas of the memory shrink while many overloaded replicas expand. Conversely, with the thwart mechanism any replica tries to balance the workload over the whole memory, verifying that the memory is globally overloaded before triggering an expansion. This makes the memory more stable.

Fault-tolerance. In order to show that our system adapts well in face of crash failures, we injected two bursts of failures, while maintaining a constant request rate, and observed the reaction of the memory. Figure 3.9 shows the evolution of memory size as time elapses and as failures are injected. The first burst of failures occurs at the $20,000^{th}$ simulation cycle and involves 20% of the memory replicas drawn uniformly at random.

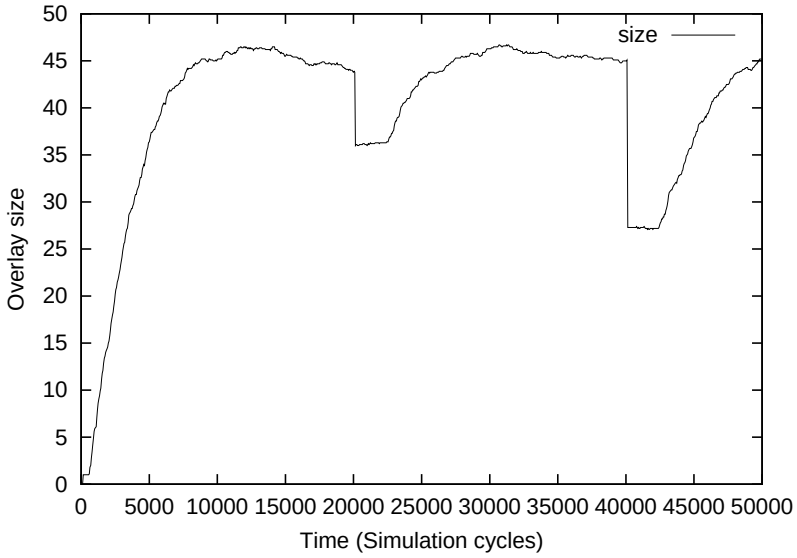


Figure 3.9: Self-adaptiveness in face of important failures.

request rate	read latency (in avg)	write latency (in avg)	max. memory size	max. consultation quorum size	max. propagation quorum size
100	478.6	733.3	10	5	6
125	621.8	812.5	14	4	8
250	1131.8	1395.8	24	3	14
500	1500.7	2173.5	46	8	23
1000	2407.9	3500.9	98	11	51

Figure 3.10: Trade-off between response time and memory size.

The second one occurs 20,000 cycles later (at simulation cycle 40,000) and involves 50% of the memory replicas. At simulation cycle 20,000, we clearly observe that the overall number of replicas drastically diminishes. Then, few cycles later, the number of replicas starts increasing again, trying to newly face the constant request rate. This phenomenon is even more pronounced at time 40,000 when 50% of the replicas fail. In both cases the system is able to completely return to an acceptable configuration without blocking, even after a large amount of failures have occurred.

Operation latency. Experiment of Figure 3.10 is composed of 5 simulations with different request rates and indicates how *Square* minimizes read operation latency. First, recall that the fast adaptive read operation contains only a **Consult** phase, thus the consultation quorum size impacts more on read operation latency than propagation quorum size does. We tuned *Square* such that a replica that receives more read requests than write requests tends to split horizontally its responsibility zone, when an expansion occurs. Since an operation is of type read with probability 0.9, replicas choose more frequently (in average) to split horizontally than vertically, consequently consultation quorums are smaller than propagation quorums, as depicted in the 5^{th} and 6^{th} columns of Figure 3.10. An increase in the requests rate—indicated in column 1—strengthens this difference: it enlarges the amount of operations, thus the phenomenon becomes more evident. Furthermore, the 2^{nd} and 3^{rd} columns confirm our thought: read operation latency is far lower than write operation latency. To conclude, even though self-adaptiveness implies that latency increases when workload increases, *Square* minimizes efficiently read operation latency.

3.4 Discussion and Conclusion

3.4.1 Quorum Access in Ad Hoc Networks

Square builds and maintains a logical communication overlay. Logical communication overlay is a very powerful tool that allows all nodes to communicate with each other, provided they know each other. In Internet-like applications, including p2p applications, participants can use a logical overlay to communicate with each other, since node i can communicate with node j if i possesses the IP address of j.

Differently, communication in ad hoc networks is often constrained by geographical locations.

That is, a node i communicates directly with node j only if i and j are geographical neighbors. For example, in wireless sensor networks, node j needs to be in the transmission range of node i. In this case, it is impossible to obtain a torus communication overlay as proposed in the dynamic grid quorum system: the furthest node from i cannot be the neighbor of i due to geographical constraints. That is, each quorum node contacted by i and that is not a neighbor of i have to communicate with intermediary nodes for information to be routed back to i.

In order to reduce quorum access cost, recent research achievements in the context of ad hoc networks tend to define quorums as clusters of nodes that are geographically close [DGL+05, CDHP+05]. For example, GeoQuorums [DGL+05] is made of mobile entities, called *mobile hosts*, and fixed entities, called *focal points*. A focal point is a fixed region in the plane with some mobile hosts that can communicate through atomic broadcast. A focal point is considered as *failed* if no mobile hosts is in it. Focal points are grouped into clusters using an independent algorithm. The introduced biquorum system contains two types of quorums such that any quorum of the first type intersects any quorum of the second type. The first typed quorums contain all focal points of a cluster while the second typed quorums contain one focal point of each cluster. Intersection of quorums is guaranteed because any quorum of the first type has at least one focal point in common with any quorum of the second type.

Another approach [LWV03] uses a dynamic quorum strategy that consists in contacting k nodes: for instance, if $k \geq \lfloor \frac{n}{2} + 1 \rfloor$, then intersection is ensured deterministically. Chapter 4 presents much lower threshold k to obtain intersection with the desired probability. The same paper [LWV03] presents a grid quorum system similar to Square, however, their approach require that each client knows exactly all quorum members. That is, in settings where storage space is limited such solutions do not scale with the size of the memory. Square is a scalable solution that do not require large storage space and is thus promising for ad-hoc networks settings. A routing protocol such as the one presented in [MKB05] could complete the Square protocol to provide a scalable DSM for ad-hoc network.

3.4.2 Limitations of Square

First, the local reconfiguration executes frequently. To prevent all-to-all exchange of global reconfiguration, Square proposes a local reconfiguration mechanism. Even if the number of message that is required during a local reconfiguration is very low compared to a global reconfiguration, the local reconfiguration is far more frequently executed than global reconfiguration. While global reconfiguration is executed unfrequently, a local reconfiguration must occur after each dynamic event. In fact, quorum replication presented in Chapter 2 minimizes the impact of dynamism, whereas quorum replication of Square only balances the load. As a result, while DSM with global reconfiguration produces more congestion when scaling, the overall communication cost of DSM with local reconfiguration may not be lower.

There exists an important tradeoff between communication complexity and operation latency that delays operation of Square. This tradeoff is due to the local reconfiguration mechanism at

the heart of Square. As seen in the Section 3.3.6, the memory grows when the system scales and the request rate increases. A serious drawback is the overhead on the operation latencies. Since quorum probes are non-adaptive from neighbor to neighbor, the operation latency is linear in the quorum size. Since, in the best case, the quorum size is $\sqrt{n}$, the operation latency may be dramatically high when the request rate is very important ($O(\sqrt{n})$). Hence, even if Square tolerates high load, its time complexity is affected by load. Some of the quorum systems proposed in the literature, whose memory size never changes, have adopted a different compromise between operation latency and reconfiguration complexity. These quorum systems have smaller operation latency but larger reconfiguration complexity. Finding the right compromise, although very interesting, is not the subject of this thesis: the goal of Square was to present very low reconfiguration complexity for efficient adaptation in face of load bursts, leave events, and join events.

3.4.3 Conclusion

This chapter addresses the problem of distributed shared memory that achieves scalability in a dynamic context. Dynamism requires reconfiguration while scalability requires the communication overhead to be handled by the underlying network, even when the system enlarges. A solution for minimizing communication cost while coping with accumulating failures seems to be local reconfiguration; locality relies on proximity in the communication graph. This reconfiguration is executed by a node and its neighbors if a failure is detected locally. By restricting the number of neighbors in the communication graph, the communication complexity of local reconfiguration can be very low.

Other issues related to scale-shift have been identified. If the system grows, then the potential number of clients requesting the memory may increase accordingly. This scale-shift produces high bursts of load. To cope with congestion that may appear at some nodes of the memory or the overload that may happen if the memory has not enough resources, the proposed memory adapts dynamically its resources in face of load variation. First, our solution balances the load over the distributed participants to prevent congestion. Then, our solution expands if the whole memory is overloaded and shrinks as soon as bursts of load stop.

Furthermore, all these scalability-related issues led us to investigate the structure of quorum system so as the way quorum members communicate with each other. Specific properties on the structure of quorum systems have been compared along with this chapter. Our solution takes benefit of our observations on quorum system structure for inherent load-balancing and communication structure to minimize reconfiguration cost.

Finally, we identified an important tradeoff between message complexity and operation latency. In fact, minimizing message complexity of local reconfiguration increases the operation latency. Tuning the degree of the logical communication graph to diminishes operation latency while increasing the reconfiguration complexity is of significant interest. While our solution scales well by moderating the use of limited bandwidth resource, induced message latency may become too large. Next chapter overcomes this issue by tolerating dynamism, achieving scalability, and minimizing

latency at the cost of relaxed consistency guarantees.

Chapter 4

Facing Scalability and Dynamism Probabilistically: Timed Quorum System

This chapter focuses on the problem of emulating a distributed shared memory in a large-scale and dynamic context. Previous chapters have enlightened the complexity of implementing atomicity in message-passing model. More specifically, Chapter 2 proposed a memory for dynamic systems using a periodic reconfiguration that is not suited for large-scale systems. In contrast, Chapter 3 proposes a memory that scales well but whose local reconfiguration is built upon failure detectors. Here, we try to avoid the use of reconfiguration to obtain a lightweight distributed shared memory emulation, at the price of relaxing deterministic guarantees.

This chapter relaxes the strong atomic consistency criterion chosen so far for emulating distributed shared memory. Instead, the aim of this chapter is to propose a memory with high quality of service in large-scale dynamic systems. This translates into ensuring that clients, which execute operations, are satisfied with high probability.

Timely guarantees to cope with dynamism. For this purpose we define *Timed Quorum System* (TQS), a new quorum system that introduces the notion of time and probabilistic guarantees into quorums. More precisely, TQS relies on quorums with a bounded lifetime and aims at providing consistency guarantees in an unbounded lifetime. The intersection property between two quorums depends not only on probability but also on time. (Recall that the intersection property of two quorums holds if and only if their intersection contains at least one active node.) The notion of time is of great interest in dynamic systems because of the difficulty of ensuring invariants. Because of its timely characteristics, TQS is easily implementable in dynamic systems.

Probabilistic guarantees to cope with scaling. For the sake of addressing both dynamism and scalability issues while emulating distributed shared memory, this chapter presents an implementation of TQS that avoids any complex structural requirement (for intersection among quorums

or communication among quorum members). Instead, the memory presented here trades simply structural requirements with connectivity requirement and atomic object emulation with probabilistic atomic object emulation. The resulting algorithm provides probabilistic guarantees while not requiring costly reconfiguration or failure detection.

Specifically, the quorum intersection is provided with high probability. That is, it may happen, though very unlikely, that two quorums do not intersect. Due to this lack of intersection, operation may be affected and cannot ensure atomic consistency. To cope with this consistency violation we propose a new consistency criterion that is a probabilistic variant of atomicity. This new consistency criterion allows some operations to fail in some exceptional cases. Interestingly, this consistency criterion requires that each operation has the same probability of failing, i.e., no failed operation affect the success of any other operations.

To summary, the problem addressed in this chapter is the emulation of DSM in large-scale and dynamic context. First, a new consistency criterion is compared to existing consistency criteria. Then, various probabilistic quorum systems are presented and a new one based on a timely intersection property is defined. Finally, we give an implementation of this timed quorum system and we prove that this emulates a DSM that respects probabilistic atomicity.

Roadmap. Section 4.1 focused on probabilistic guarantees, by presenting probabilistic consistency criteria and the probabilistic quorum systems. Section 4.2 gives hints on how to avoid costly tasks that prevent the system from scaling or tolerating dynamism. Section 4.3 presents our solution and show that it implements probabilistic atomicity. Finally, Section 4.4 discusses the resulting solution and concludes the chapter.

4.1 Probabilistic Guarantees

Probabilistic guarantees are weaker than deterministic ones. Indeed, a property that holds deterministically holds also with probability 1 and a property that holds with some probability $p < 1$ does not hold deterministically. As a result, it is easier to ensure probabilistic guarantees than deterministic ones. Here, the aim is to circumvent the problems encountered when guaranteeing atomicity by guaranteeing probabilistic atomicity while improving scalability and dynamism tolerance of DSM. First, we investigate existing consistency relaxation and define probabilistic atomicity. Then, we present interesting probabilistic quorum system implementations.

4.1.1 Probabilistic Consistency

In the following we present some consistency criteria that motivated us to define probabilistic atomicity.

Weakening atomic consistency. Many efforts have been devoted to express formally consistency criteria. Thus, there are various consistency criteria among which atomicity [Lam86, Lyn96] (a.k.a. linearizability [HW90]) is the strongest one. This means that any implementation that satisfies atomicity implements, for sure, any other consistency criterion. Atomicity presents desirable features like strength and locality [HW90] that motivates its use in the previous chapters of this thesis. Nonetheless, atomicity is difficult to ensure due to its strength. As a result of this difficulty, numerous consistency criteria weaker than atomicity appeared in the literature. Safety and regularity [Lam86] both require that a read operation that is not concurrent with any write returns the last written value. Weak atomicity states that if two reads return the same value then any read ordered after must return either the same value or a more up-to-date value.

An interesting consistency class, called hybrid consistency [AF98], benefits from both weak and strong consistency criteria. This class defines strong and weak operations orderings: strong operations ordering implies that there exists a consistent ordering on operations from any node standpoint while weak one allows different nodes to consider different operation orderings. A consistency criterion providing weak or strong ordering, depending on the object that is accessed, appeared in [DS90, DSB88]. In some sense, hybrid consistency allows some weak operations to cohabit with strong operations. All these consistency criteria rely on deterministic requirements that are heavy to implement in large-scale dynamic systems. For instance, as part of the specification of the implementation of hybrid consistency, each operation must be defined as either strong or weak but not both. That is, all operations must be predefined as strong or weak and during any execution, the requirement of strong operations is never relaxed while weak operations never provide strong guarantee.

In contrast, randomness has been introduced in other consistency criteria. For instance in [AGMT95] the shared memory model states that in some cases stale values may be returned depending on the type of shared objects that is read. Moreover, other work uses probabilistic guarantee on latency, preventing any wrong value from being returned after a read operation [SZ98]. Finally, randomized registers [LW05] allow read to return old values as long as any value written is eventually read or overwritten. As far as we know, none of the consistency criteria that relies on probability allow operations to occur successfully with high probability. In this chapter, we propose a new consistency criterion based on atomic consistency guaranteeing that any operation is atomic (or succeeds) with high probability. An algorithm using TQS is proposed as an implementation of a probabilistic atomic object and we show that this implementation also satisfies randomized registers.

Randomized object. Randomized registers appeared in [LW05] as a register abstraction that can be used for implementation of iterative algorithms that converges after a finite number of steps. Before defining random register, we have to define what means that a read operation reads from a write operation. A read operation r is said to *read from* some write w if w begins before r ends; the value returned by r is the same value as that written by w; w is the latest write operation satisfying

the two previous conditions. Next, we define a randomized object equivalently to the definition of randomized register of [LW05] by assuming the seminal first rule and restating the two others.

Definition 4.1.1 (Randomized Object) *Let x be a randomized object. Let H be a complete sequence of invocations responses of read and write operations applied to object x. The sequence H satisfies the randomized object abstraction if and only if the following properties hold:*

1. *Every read in every complete execution reads from some write.*

2. *Let e be any finite execution and let inv(π) be an invocation event of a write operation π that belongs to e. The probability that this write is read from infinitely often is 0, if an infinite number of writes are performed after inv(π).*

A randomized object is an abstraction that defines two rules satisfied by read and write operations. The first rule states that a read operation must return a value written by a previous write operation. Note that this rule includes the serial specification of object as mentioned in Chapter 2. Moreover, this rule alone relaxes the property of safe register given in [Lam86]. Nevertheless, the third rule requires that, provided an infinite number of write operations, at some point in time, read operations will stop reading from the same write. This ensures that the information that is read is either getting more up-to-date as long as read operation occurs or it is already up-to-date.

Probabilistic atomic object. Probabilistic atomic object is a new abstraction providing distributed shared memory emulation with high quality of service despite large scale and dynamism. For the sake of tolerating scale-shift and dynamism, we aim at relaxing some properties. However, our goal is to provide each client with a distributed shared memory emulation that provides satisfying quality of service. Quality of service must be formally stated by defining a consistency criterion that defines the guarantees the application can expect from the memory emulation. We aim at providing quality of service in terms of accuracy of read and write operations. In other words, our goal is to provide the clients with a memory that guarantees that each read or write operation will be successfully executed (thus, verifying atomicity) with high probability. This notion of quality relaxes previous deterministic constraints as atomic consistency and allows the emulation of distributed shared memory through less complex mechanisms. It is noteworthy that the expression *with high probability* means a probability $1-e^{\eta^2}$, where η represents a constant, larger than 1, that is tuned by the application designer.

In order to formalize the notion of quality of service while reusing formalism chosen so far, we define the probabilistic atomic object as an atomic object where operation accuracy is provided with high probability. Let us first recall properties 1 and 3 of atomicity (Definition 2.1.1):

- $(\pi_1,\pi_2)-$*ordering*: if the response event of operation π_1 precedes the invocation event of operation π_2, then it is not possible to have $\pi_2 \prec \pi_1$;

- $(\pi_1, \pi_2)-$*return*: the value returned by a read operation π_2 is the value written by the last preceding write operation π_1 regarding to $\prec$.

The definition of probabilistic atomicity is as follows.

Definition 4.1.2 (Probabilistic Atomic Object) *Let x be a read/write probabilistic atomic object. Let H be a complete sequence of invocations responses of read and write operations applied to object x. The sequence H satisfies probabilistic atomicity if and only if there is a partial ordering $\prec$ on the successful operations such that the following properties hold:*

1. *Let π_1 be a successful operation. Any operation π_2 satisfies $(\pi_1, \pi_2)-$ordering with high probability. (If π_2 does not satisfy it, then π_2 is considered as unsuccessful.)*
2. *if π_1 is a write operation and π_2 is any operation, then either $\pi_2 \prec \pi_1$ or $\pi_1 \prec \pi_2$;*
3. *Let π_1 be a successful operation. Any operation π_2 satisfies $(\pi_1, \pi_2)-$return with high probability. (If π_2 does not satisfy it, then π_2 is considered as unsuccessful.)*

Observe that the partial ordering is defined on successful operations. That is, either an operation π fails and this operation is considered as unordered or the operation succeeds and is ordered with respect to other successful operations.

Even though an operation succeeds with high probability, in an infinite execution it is very likely that at least one operation fails. However, our goal is to provide the operation requester (client) with high guarantee of success at each of its operation request.

4.1.2 Probabilistic Quorum System

Seminal probabilistic quorum systems have been firstly defined by Malkhi et al. [MRW97], as quorum systems whose quorums intersect with high probability. An implementation has been given in [MRW97] for static settings. The relaxation of the deterministic intersection guarantee permits to achieve low load and high fault-tolerance at the same time.

Application of probabilistic quorum systems. A first application of probabilistic quorum systems is distributed electronic voting systems [MRWW01, MR98]. Voters of a country are uniquely identified and can vote only once. That is, as soon as any voter *v* casts a vote on a voting-machine, its identifier must be locked country-wide for preventing any further vote from voter *v*. If a quorum knows that voter *v* has already voted, a second vote contacting a second quorum will prevent with high probability voter *v* from voting twice. Even though a particular voter might not be locked with some small probability, it remains unlikely that this particular voter tries to vote several times.

File-sharing applications in peer-to-peer systems is another application of probabilistic quorums systems [MTK06]. The goal is to inform a sufficient amount of nodes that a data is hosted at

some specific nodes of the system. If this information is sufficiently replicated among the system nodes, then the information is easily found by any requester. The solution of [MTK06] is for each replica to inform k quorums of nodes that it hosts the data. Later on, any node can access the data by contacting at least one quorum that intersects at least one of the k other with high probability.

Definition of probabilistic quorum systems. Probabilistic quorum systems are defined by a set system and an access strategy that maps each quorum of the set system with a probability of being accessed. This probability of being accessed and the quorum definition determine the intersection probability. Recall first, as mentioned in Definition 1.1.1, that a set system over a universe U is a set of subsets of U. Next, let an *access strategy* be a probability distribution function defined over a set system.

Definition 4.1.3 (Access Strategy) *An* access strategy ω *for a set system $\mathcal{S}$ is a probability distribution on the elements of $\mathcal{S}$. That is, $\omega : \mathcal{S} \to [0,1]$ satisfies $\sum_{s\in\mathcal{S}} \omega(s) = 1$.*

A probabilistic quorum system is a quorum system whose intersection property holds with high probability.

Definition 4.1.4 (Probabilistic Quorum System) *Let Q be a set system, let ω be an access strategy for Q, and let $0 < \varepsilon < 1$ be given. The tuple $\langle Q, \omega\rangle$ is a* probabilistic quorum system *if for any quorums $Q_1 \in \langle Q, \omega_1\rangle$ and $Q_2 \in \langle Q, \omega_2\rangle$, we have:*

$$\Pr[Q_1 \cap Q_2 \neq \emptyset] \geq 1 - \varepsilon.$$

Similarly to the biquorum system (Definition 1.1.3), the probabilistic biquorum system relaxes the intersection property to quorums of distinct types.

Definition 4.1.5 (Probabilistic Biquorum System) *Let Q_1 and Q_2 be two set systems over a universe U, let ω_1 (resp. ω_2) be an access strategy for Q_1 (resp. Q_2), and let $0 < \varepsilon < 1$ be given. The tuple $\langle Q_1, Q_2, \omega_1, \omega_2\rangle$ is a* probabilistic biquorum system *if for any $Q_1 \in \langle Q_1, \omega_1\rangle$ and $Q_2 \in \langle Q_2, \omega_2\rangle$:*

$$\Pr[Q_1 \cap Q_2 \neq \emptyset] \geq 1 - \varepsilon.$$

Probabilistic quorum systems are originally presented for static systems where the number of failures considered is upper bounded. In [MRW97], the authors give an implementation of a probabilistic quorum system for a static system. The quorum size is $\ell\sqrt{n}$, with ℓ a constant and n is the number of active members in the quorum system, and the access strategy is the uniform access strategy among all these quorums.

4.1.3 Probabilistic Weak Quorum System

Miura et al. [MTK06] benefit from the relaxed intersection requirements of probabilistic quorum systems to contact nodes uniformly at random. The quorum system has no predefined structure, since quorums are constructed uniformly at random. Moreover, they define *probabilistic weak quorum systems* as a set of quorums, each being accessed through a distinct access strategy and such that each quorum intersects at least one of k other quorums with high probability. This definition weaken the intersection requirement given in the seminal Definition 4.1.4.

Looking for an object in a peer-to-peer networks. This probabilistic weak quorum system is used for the purpose of object searching application in peer-to-peer systems. Some nodes post an information about an object location in the network. Later on, a node i searches for the object by trying to contact a node that owns the information about the object location. Each of those nodes contact a distinct quorum. If the quorum contacted by the searching node i intersects one of the quorum contacted by the posting nodes, then i obtain the location information and can access the file. A probabilistic weak quorum system is depicted on Figure 4.1.

More technically, for one quorum to intersect at least one among k quorums with high probability, the quorum size must be $\Omega(\sqrt{\frac{n}{k}})$ where n is the total number of nodes and $k \in [\frac{n}{1000}, \frac{n}{100}]$ is the number of nodes posting the information about the object location. (Actually, these k nodes own the data initially.) During the protocol, each of these k nodes posts to $O(\sqrt{\frac{n}{k}})$ nodes chosen uniformly at random, i.e., a quorum of nodes, the information that it owns the data. After this, a node can retrieve with high probability the data location by contacting $O(\sqrt{\frac{n}{k}})$ nodes chosen uniformly at random, i.e., another quorum of nodes.

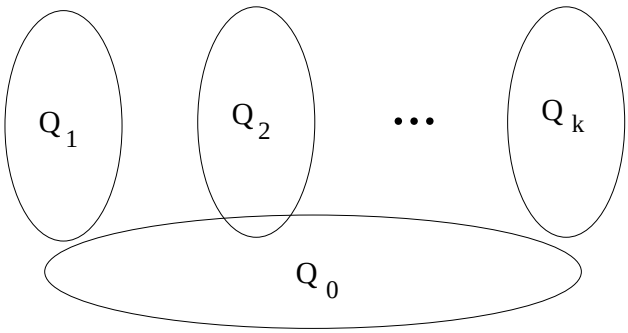


Figure 4.1: A probabilistic weak quorum system. Quorum Q_0 intersects at least one of the k quorums Q_1, Q_2, ..., Q_k with high probability.

In some sense probabilistic weak quorum system is similar to a traditional probabilistic quorum system. First, multiple nodes contact a large quorum in a collaborative way for posting some information. Second, a single node contacts a small quorum for searching the information. The

collaboration is motivated by the fact that many more messages are used for posting the information than for searching it. From a single node standpoint the cost of posting or searching is equal in terms of number of messages involved.

Discussion on adapting the protocol for read/write in dynamic systems. One could think of a generalization of this algorithm for read/write operations in a dynamic systems, where a write operation is made more costly than a read operation since write operations are less frequent than read operations. However, for consistency purpose such distributed shared memory requires all posting nodes to share a consistent view of the object before propagating the same value. That is one solution would be to artificially update the view of all posting nodes before they post. This is easily achieved using a consultation before propagating and tagging like explained in the two-phase write operations presented in previous chapters.

The proposed protocol, the name-thread protocol, aims at providing each node with a global knowledge of the system in order to search for an object later on. This protocol is proved to be self-stabilizing. That is, when the environment stabilizes and no failure occurs during a sufficient amount of time, then the algorithm succeeds. However, in large-scale systems where dynamism is frequent, it is unreasonable to assume stabilization of the whole system during a long period of time. Interestingly, despite high dynamism, an object value replicated at numerous k locations is expected to persist a large period of time [GKM+06]. While it is impossible to assume system stabilization, an interesting aspect is thus to consider dynamism as part of the model: quorums intersect even if some nodes keep failing. The solution presented in Section 4.2 stems from this idea.

4.1.4 Probabilistic Quorum Systems for Dynamic Settings

In [AM05], Abraham and Malkhi proposed an implementation of a structured probabilistic quorum system. As far as we know it is the first time, dynamism is introduced into probabilistic quorum system definition. The resulting structured quorums systems Q, called *dynamic* ε*-intersecting*, consider quorums whose structure evolves without violating the probabilistic intersection. To handle dynamism, a logical structure is re-adapted each time a node joins, fails, or leaves.

A structured quorum system. The structure of the quorum system is a dynamic approximation of a De Bruijn graph that serves two major purposes. First the De Bruijn graph limits the cost of reparation of the structure needed when a failure is detected. Second, this graph maps each node to a specific level depending on its location on the structure such that it can roughly determine the size of the whole structure. Based on these facts, the structure needs a logarithmic number of steps[1] to readapt each time a join or a leave occurs in the system, and can determine the number and the size of random walks to run in order to achieve good probability of intersecting.

[1] This number is logarithmic in the system size

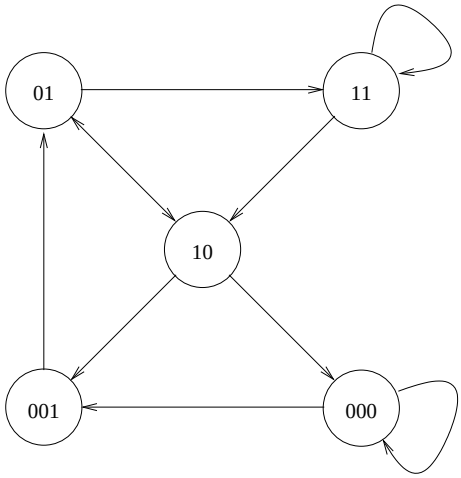


Figure 4.2: An example of dynamic approximation of the De Bruijn graph with five nodes.

Formally, each node is identified by an ordered sequence of bits 0 or 1. A node i identified by $\langle a_1, ..., a_k \rangle$ has an edge to node j if and only if:

- $\langle a_2, ..., a_k \rangle$ is a prefix of j identifier, or
- j identifier is a prefix of $\langle a_2, ..., a_k \rangle$, or
- j identifier is $\langle a_2, ..., a_k \rangle$.

An example of a De Bruijn graph with 5 nodes identified by 000, 001, 01, 10, and 11, is depicted on Figure 4.2.

Since the structure is continuously re-ordered to face dynamism, each node can approximate the number of nodes present in the structure by looking at its level, i.e., the number of digits d that identifies its location on the structure. The approximation obtained is 2^d. Because of dynamism, the structure might not be equilibrated at the time a node needs to approximate its size, that is, a gap factor G is taken into account as a margin error on the size approximation. Based on the structure size approximation and an upper bound on G, nodes can determine the number of random walks needed to contact a quorum with high probability. The authors show that $O(\sqrt{n})$ random walks, each being of length $O(\log n)$ are sufficient to achieve intersection with high probability. (More precisely, the number of random walks required is $\rho\sqrt{2^d + 2\log_2 G}$, with ρ a constant.)

Quorum probe in structured quorum system. A quorum is obtained, using $O(\sqrt{n})$ random walks of length $O(\log n)$. More precisely, a quorum contains the last nodes contacted by each of the random walks, thus leading to quorums of size $O(\sqrt{n})$.

A random walk is run by a source node of the approximation of the De Bruijn graph. The message corresponding to this random walk is forwarded from neighbor to neighbor $O(\log n)$ times, using a decreasing time-to-live value piggybacked in the message. During the random walk, each neighbor is chosen depending on the identifier it owns.

Depending on the identifier of the current node and the current time-to-live value, an index i is chosen. A coin is flipped so that value 0 or 1 is uniformly drawn at random. Given the value v obtained, the next neighbors is chosen such that its identifiers has v at index i. If more than a single neighbor identifier has the corresponding value v at index i, the next neighbor is chosen uniformly at random among those ones.

Observe that, even though each identifier is chosen uniformly between 0 and 1, the neighbors are not necessarily chosen uniformly among all possible neighbors. This results from the fact that among three neighbors, it might happen that two neighbors have value $v = 0$ while one neighbor has value $v = 1$, so that the first two are drawn with probability $\frac{1}{4}$ while the last one is drawn with probability $\frac{1}{2}$. Anyway, at the end of all random walks, each contacted node has an identifier that contains a sequence of bits, each uniformly drawn at random among 0 and 1.

Reconfiguration by replication and maintaining the structure. Key requirements of this quorum systems rely on the invariant that should not be affected by dynamism. The Invariant is necessary to ensure that each node has a right identifier and that the structure is balanced. For example, random walks assume that each node has at least two neighbors so that the message can be forwarded with the same probability to a node having a 1 or a 0 at index i of its identifier.

Invariant 4.1.1 *The De Bruijn approximation graph is balanced and the identifier of any node indicates its position in the graph.*

To guarantee this Invariant, each dynamic event involves communication cost to update the identifiers and re-balance the structure. Observe that the structure is dynamic in the sense that the number of identifiers is potentially infinite: we can add a digit at the end of the current identifiers to obtain new identifiers. Despite this flexibility, the node identifiers are used to equilibrate the structure. A node identified with less digits has a more important role and is more likely loaded, while a node identified with more digits participates less likely in the protocol. Moreover, a node determine the level of the structure based on its number of digits and the largest difference in the number of digits between any two identifiers define the mistake margin that impacts on the probability of intersection of two random walks. Consequently, the identifier needs to be updated as fast as possible, regarding to the number of nodes in the system. Due to this update, the worst case time complexity required to handle a dynamic event in this structure is $O(\log n)$ time.

Overcoming invariant related issues. In Section 4.2, we overcome these problems. We benefit from the Timed Quorum System (TQS) definition to implement a structureless quorum system. First, this quorum system do not need any structural property, thus, no failure detection is required

and the system self-adapts naturally in face of dynamism. Second, because of the absence of structure, the quorum system does not need to update any node identifier when dynamic events occur. This is fundamental in large-scale dynamic systems since dynamic events may be very frequent.

4.2 Avoiding Quorum System Reconfiguration

Reconfiguration is necessary to provide consistency maintenance deterministically in a dynamic systems. Reconfiguration repairs a quorum system when failures accumulate. More precisely, reconfiguration replaces a node, a quorum, or the whole quorum system by a new one. To ensure consistency, reconfiguration is not finished before the state of the new entity is updated with respect to the state of the other participants. In the quorum system, the state of each node must reflect its quorum belongingness and reconfiguration must guarantee this despite dynamism.

Strict quorum systems require to be structured while probabilistic quorum systems do not. While a strict quorum must provide a client with the status of at least one element of all quorums of the system, the probabilistic quorum must not. Because of this requirement, the communication among elements of a strict quorum has some structural constraints. During the probe of a strict quorum, a client must know all quorum members (non-adaptive probe) or a quorum member such that this member will contact another member in turn, and so on (adaptive probe) as explained in Section 3.3.4. This communication implies a dedicated communication structure among nodes. In contrast, given an approximation of the system size (approximating the system size is explained in Section 4.4) and the number of replicas in the system, a node might simply contact randomly a certain amount of nodes in the system to ensure that at least one replica is contacted with high probability, whatever nodes are connected too.

Next, we explain how to obtain randomness without structured communication overlays. This translates into the fact that probabilistic quorum systems do not need each node be mapped to a specific role in the communication structure. Neither do node states need to reflect any structural positioning. Without these structural requirements in probabilistic quorum system, we propose a structureless quorum system that do not need reconfiguration.

4.2.1 Structureless Quorum System

Structured quorum system requires a costly reconfiguration mechanism to cope with dynamism. As said before, it is natural to think of a quorum system as a structure. For instance, Chapter 3 presents a grid-like topology to provide optimal load. Nevertheless, each dynamic event needs a reconfiguration mechanism to maintain the structure of the quorum system. This reconfiguration induces a costly communication overhead and is difficult to implement.

Structured communication overlays. The communication structure of quorum system is used to associate some participants together so that they can communicate. Communication is a well-studied subject in large-scale dynamic networks. Two major solutions for mapping a node to neighbors appeared in such context. First, structured solutions provide each node with a logical location in a graph, that represents for example a ring [MKKB01], a grid [RFH+01], a butterfly network [MNR02]. These solutions aim at maintaining the structure despite dynamism, triggering a dedicated handling mechanism each time a dynamic event occurs. The use of such structures gained popularity with the achievements of distributed hash table (DHT) where a set of object keys is mapped to nodes for the purpose of routing.

Unstructured communication overlay. Another type of systems called unstructured systems, have gained consideration for large-scale dynamic networks these recent years. This communication model used in such systems has at heart a gossip mechanism that first appeared in [DGH+87] and benefits from randomness, periodicity, and locality. Despite the fact that gossip-based systems share many similarities with systems described in various fields [CGJ+07], a unified abstraction for this communication model has been proposed as the peer-sampling service [JGKvS04], which allows nodes to exchange periodically a random subset of their neighbor identifiers resulting in a dynamic and unstructured overlay.

Unstructured communication overlay copes inherently with dynamism. First, reconfiguration process is useless, since no structures need to be maintained. Second, as assumed in structured overlay, a joining node sends a join message to a contact node already in the system. This contact node adds the joining node identifier to its own list of neighbors. The periodic exchange of a random set of neighbors disseminates this identifier in the network. Conversely, the identifier of a failed node progressively disappear from the system. This is due to a priority mechanism that privileges the exchange of identifiers of the most recently gossiping nodes.

Building quorums on-the-fly. Benefiting from unstructured communication overlay, unstructured quorum systems are very efficient. First, they tolerate inherently dynamism by removing (adding) identifiers of failed (joining) nodes from (to) the overlay. Second, they do not need reconfiguration. Instead of relying on a specific structure, each quorum is built on-the-fly so that a client that executes a request at a given time creates a specific quorum. This can be done by contacting a sufficiently large set of nodes, each node being accessed following a dedicated strategy.

In the context of DSM, a read operation must access the last written value, that is, values must be replicated regularly in the system to cope with dynamism. This replication mechanism does not need a costly reconfiguration but can be done though write operation executions without additional overhead. (Later on, we present a replication as part of any operation.) Observe that nodes where a value is written/replicated at a certain time, might leave the system after some time elapses. This observation leads us to define Timed Quorum System that provides timely quorums whose intersection depends on the time quorums are built.

4.2.2 Timed Quorum System

This section defines Timed Quorum Systems (TQS) as quorum systems whose intersection among quorums is probabilistic and depends on the time a quorum is probed. Each quorum is mapped to a specific instant and has a bounded lifetime due to the dynamism intensity (i.e., *churn*) of the system. Before being created of after its lifetime elapses, a quorum is not guaranteed to intersect with any other quorums, however, during its lifetime a quorum is considered as *reachable*: two quorums that are reachable at the same time intersect with high probability. In dynamic systems nodes may leave at any time, but this probability is bounded, thus it is possible to determine the intersection probability of two quorums.

Most of the dynamic models assume that dynamic events are dependent from each other: only a limited number of nodes leave and join the system during a bounded period of time. For instance in Chapter 2, it is assumed that nodes departures are dependent: quorum replication ensures that all nodes of at least any two quorums remain active between two reconfigurations occur. However, in a real dynamic system, nodes act independently. Due to this independence, even with a precise knowledge of the past dynamic events, one cannot predict the future behavior of a node. Putting this observation into quorum system context, it translates into the impossibility of predicting deterministically whether quorums intersect.

In contrast, here, we measure how likely two quorums intersect. More precisely, our goal is to measure the probability that two quorums intersect depending on time. In the following, we present a dynamic system in which nodes join and leave the system at any time and independently so that it is impossible to predict when. However, the probability that k nodes leave the system increases as time elapses. As a result, the probability that a quorum $Q(t)$ probed at time t and that a quorum $Q(t')$ probed at time t' intersect decreases as the period $|t'-t|$ increases.

Building onto this observation, we propose a TQS where each quorum is defined for a given time t. Each quorum $Q(t)$ has a lifetime Δ that represents a period during which the quorum is reachable. Differently to availability defined in the previous chapter, reachability does not depend on the number of nodes that are failed in a quorum system because this number is unpredictable in dynamic systems. Instead, a $Q(t)$ quorum is reachable if at least one node of quorum $Q(t)$ is reached with high probability: if two quorums are reachable at the same time, they intersect with high probability. More generally, let two quorums $Q(t)$ and $Q(t')$ of a TQS be reachable during Δ time (their lifetime is Δ); if $|t-t'| \leq \Delta$ then $Q(t)$ and $Q(t')$ intersect with high probability.

Dynamic memory and quorum abstraction. In [LS02, CGG+05] as in Chapter 2, a memory is emulated using biquorum systems that are subsequently replaced over time. This replacement introduces new quorums that are decided upon with a reconfiguration mechanism. Subsequent quorums do not need to intersect provided that information is conveyed from ones to others using a reconfiguration mechanism. Similarly in [FRT05], a quorum abstraction is defined as subsequent biquorum systems. This abstraction requires two properties: (i) intersection and (ii) progress, in which the notion of time is introduced. First, a propagation quorum intersects the consultation

quorum contacted subsequently. Second, each node of a quorum remains active between the time the quorum starts being probed and the time the quorum stopped being probed. Observe that the intersection property allows some consultation quorums and some propagation quorums to be disjoint. However, the fact that clients convey information from a consultation quorum to a propagation quorum can be viewed as an intersection requirement.

TQS differs from above solutions in two points. First, the intersection between quorums depends only on the time a quorum is probed rather depending on the type or the rank of the phase it corresponds to. Second, TQS especially relaxes the intersection property by requiring probabilistic guarantees instead of deterministic ones. That is, the probability of intersection evolves with time, thus, even though some nodes of quorum $Q(t)$ leave the system, the intersection probability may be sufficient (i.e., $Q(t)$ is reachable). These two points make TQS especially suited for dynamic systems where deterministic properties and properties that hold at any time are difficult to implement.

In [AM05], a probabilistic quorum system implementation is designed for quorum systems that adapt in face of changes. In this approach, the quorum system is a structure such that quorum probes are given by the structure. That is, a single quorum system can evolve (grows and shrinks) over time but remains the same quorum, because of the location of its nodes on the structure. Even though this quorum changes, the way it is probed never changes. Moreover, the implementation given tolerate dynamism, by re-adapting the topology (and quorums) when a failure occurs. Conversely, a TQS experiences quorum death and quorum birth: any quorum can disappear after some time and a new one can appear after some time. The key point is that the way a quorum is probed changes over time and is only dictated by node failure and node activity.

Since TQS relies on time, quorums are accessed differently depending on the time: if a node i fails at time t, then before time t, node i may be accessed while after time t, node i will never be accessed. Hence, its access probability varies over time. The key advantage of TQS is that they can be implemented with no failure detection and no reconfiguration mechanism. (Such an implementation is given below.) For some applications, the drawback of such an implementation is the operation regularity requirement. We discuss this drawback in Section 4.4.

Definition of Timed Quorum System (TQS). Next, we formally define TQS whose quorums intersection is probabilistic and depends on the time quorums are created. TQS are especially suited for dynamic systems where the behavior of nodes is unpredictable, since they simply require probabilistic intersection and no deterministic intersection. Moreover, quorums experience a bounded lifetime so that their intersection guarantees are timely. Recall that the universe contains the set of all possible nodes, including the one that have not join the system yet.

We first define the timed access strategy as an access strategy over a set system that may vary over time. This definition is motivated by the fact that an access strategy defined over a set $\mathcal{S}$ can evolve. To compare with the existing probabilistic dynamic quorums, in [AM05] the authors defined a dynamic quorum system using an evolving strategy that might replace some nodes among

a quorum while its access strategy remains identical despite this evolution. Unlike the dynamic quorum approach, we need a more general framework to consider quorums that are different not only because of their structure but also because of how likely they can be accessed. The timely access strategy adds a time parameter to the access strategy defined in Subsection 4.1.2 allowing a timed access strategy to vary over time.

Definition 4.2.1 (Timed Access Strategy) *A* timed access strategy $\omega(t)$ *for a set system* $\mathcal{S}$ *at time* $t \in T$ *is a probability distribution on the elements of* $\mathcal{S}$ *at time* t*. That is,* $\omega : \mathcal{S} \times T \rightarrow [0,1]$ *satisfies at any time* $t \in T$*:* $\sum_{s \in \mathcal{S}} \omega(s,t) = 1$.

Informally, at two distinct instants $t_1 \in T$ and $t_2 \in T$, an access strategy might be different for any reason. For instance, consider that some node i is active at time t_1 while the same node i is failed at time t_2, hence it is likely that if $i \in s$, then $\omega(s,t_1) \neq 0$ while $\omega(s,t_2) = 0$. This is due to the fact that a node is accessible only when it is active.

Definition 4.2.2 (Δ-Timed Quorum System) *Let* $\mathcal{Q}$ *be a set system, let* $\omega(t)$ *be a timed access strategy for* $\mathcal{Q}$ *at time* t*, and let* $0 < \varepsilon < 1$ *be given.*

The tuple $\langle \mathcal{Q}, \omega(t) \rangle$ *is a* Δ-timed quorum system *if for any quorums* $Q(t_1) \in \mathcal{Q}$ *accessed with strategy* $\omega(t_1)$ *and* $Q(t_2) \in \mathcal{Q}$ *accessed with strategy* $\omega(t_2)$*, we have:*

$$\Delta \geq |t_1 - t_2| \quad \Rightarrow \quad \Pr[Q(t_1) \cap Q(t_2) \neq \emptyset] \geq 1 - \varepsilon.$$

Implementing a randomized object. Next, we give a simple Timed Quorum System algorithm that implements a randomized object. Since randomized object are used to implement iterative algorithm and to solve a large class of problems [UD90], this implementation gives an overview of the strength of Timed Quorum system. Moreover, it enlightens the fact that Timed Quorum System may be used for various kind of applications. We assume that the system set contains only all sets containing $q \neq 0$ active nodes in the system and that the access strategy $\omega(t)$ is the uniform access strategy over all possible active quorums at time t.

Theorem 4.2.1 *The Timed Quorum System Algorithm implements a randomized object.*

Proof. Property 1 holds because when a node is locally read it returns the last value it has written locally.

For the second property, we need to show that no value written at a node is infinitely read. For this purpose, we show that any written node either fails or is eventually over-written. Assume that there is a series ℓ of subsequent write operations and let Q_k be the quorum contacted by the k^{th} of these write operations. Without loss of generality, we refer to Q_v as the quorum with the most up-to-date value v and we bound the probability $\mathcal{P}_1$ that v is infinitely read.

Observe first that any given node i of Q_v might fail before all write execution. By assumption when i recovers it no longer hosts value v. In this case, value v cannot be infinitely read at node i.

Next, we upper bound the probability $\mathcal{P}_1$ by considering a worst case scenario in which no nodes of Q_v fail. At any time t, the timed access strategy $w(t)$ draws uniformly at random any quorum containing q active nodes at time t. The upper bound of $\mathcal{P}_1$ is the probability $\mathcal{P}_2$ that at least one node i of Q_v survives the series of write operations, while no node of Q_v fails:

$$\begin{aligned}\mathcal{P}_2 &= \sum_{i \in Q_v} \Pr[i \notin Q_1 \cup \ldots \cup Q_\ell], \\ &= \sum_{i \in Q_v} \prod_{j=1}^{\ell} \Pr[i \notin Q_j], \\ &= |Q_v| \prod_{j=1}^{\ell} (1 - \frac{|Q_j|}{n}).\end{aligned}$$

Since $|Q_j| = q > 0$, and $\mathcal{P}_1 \leq \mathcal{P}_2$, we have $\lim_{\ell \to \infty} \mathcal{P}_1 = 0$ and the result follows. □

4.3 Scalable Dynamic Distributed Shared Memory (benefiting from Prototypical Gossip)

The goal is to define a structureless memory that provides probabilistic atomic operations in a large-scale dynamic system. To this end, we need an implementation of a Timed Quorum System (TQS). In other words, we are interested in answering the following question: Using quorums whose lifetime Δ depends on their size q due to dynamism, how can we implement a memory that ensures operation success with high probability in a large system? Or, more technically, if an object value v is written at a quorum $Q(t)$ at time t and if a value v' is consulted at a quorum $Q(t')$ at time $t' = t + \Delta$, how can we ensure to have $v = v'$ with high probability? In the remaining of this paper, we assume that all quorums contain q nodes, since an interesting goal is to minimize both the size of quorums where the value is propagated and the size of quorums a client has to probe to find the right value.

The solution we present here works roughly as follows. A two-phase operation is executed frequently by any client in the system. The first phase consults the newest value of the object at q nodes in the system while the second phase propagates the newest value to q nodes of the system. A tag associates each value so that the newest value is with the largest tag and is easily identifiable. During a phase, q' nodes are contacted uniformly at random by dissemination—the underlying communication graph ensures uniformity of disseminated node drawings—such that $q = O(\sqrt{nD})$ different nodes are contacted, where n is the system size and D is a values depending on the dynamism intensity (churn). (Remark that it has been proved in [MRWW01] that $q = O(\sqrt{n})$ is sufficient if the system is static.) The phase dissemination that contacts q' nodes takes $log(q')$ message delays. The sets of q nodes contacted during each phase form a TQS: any two sets intersect with high probability depending on the time they are probed.

4.3.1 Model and Definitions

Model of the dynamic system. The computation model is very simple. The system consists of n nodes. It is dynamic in the following sense. Every time unit, cn nodes leave the system and cn nodes enter the system, where c is an upper bound on the percentage of nodes that enter/leave the system per time unit; this can be seen as new nodes "replacing" leaving nodes. A node leaves the system either voluntarily or because it crashes. A node that leaves the system does not enter it later. (Practically, this means that, to re-enter the system, a node that has left is considered as a new node; all its previous knowledge of the system state is lost.)

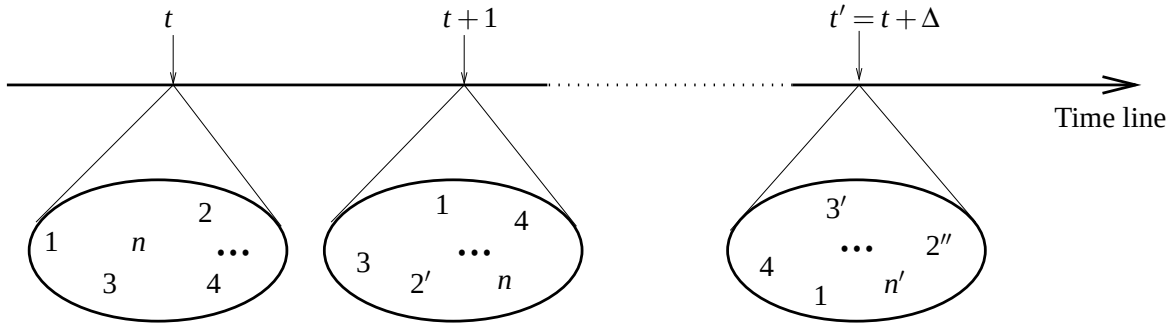


Figure 4.3: System evolution

Figure 4.3 describes a possible system evolution. Initially (time t), there are n nodes (identified from 1 to n; let us take $n = 5$ to simplify). Let $c = 0.2$, which means that, every time unit, $nc = 1$ node changes (a node disappears and a new node replaces it). Then, at time $t+1$ the node 2 is replaced by the node $2'$. Let $\Delta = 4$. At time $t+4$, we see that the nodes 3 and n have been replaced by the nodes $3'$ and n', respectively, while the new node $2'$ has in turn been replaced by the node $2''$ and the nodes 1 and 4 still belong to the system. The important point here is that a new node can in turn be replaced at a later time. More particularly, we assume that the churn rate c applies equally to any subset of the system: any subset S of the system experiences $|S|c$ joins and leaves by time unit. This model is extended to a more realistic model in Subsection 4.3.4.

Preliminary notations and definitions. This paragraph defines several terms that are used in the algorithm description. First, recall that a shared object is accessed through read operations, which return the current value of the object, and write operations, which modify the current value of the object. To clarify the notion of currency when concurrency happens, it is important to explain what are the up-to-date values that could be considered as current. We refer to the *last value* as the value associated with the largest *tag* among all values whose propagation is complete. We refer to the *up-to-date values* at time t as all values v that satisfies one of the following properties:

- Value v is the last value.

- Value v is a value whose propagation is ongoing and whose associated tag is at least equal or larger to the tag associated with the last value.

In the remaining of this chapter, we refer to $val(\phi)$ and $tag(\phi)$ as, respectively, the value and tag consulted/propagated by phase ϕ.

Second, it is important to understand what is a successful phase. The goal of a consultation phase is to return an up-to-date value, whereas the goal of the propagation phase is to propagate an up-to-date value v so that v can be identified as an up-to-date value. Thus, we refer to a *successful phase* as a phase that achieves its goal. Observe that, if the consultation of an operation is unsuccessful, then the subsequent propagation phase of the same operation might propagate a new value with a small tag so that this value will not be identifiable as an up-to-date value. In this case, we say that both the consultation and propagation are unsuccessful phases. A more formal definition of the successful/unsuccessful phase follows.

Definition 4.3.1 (Successful Phase) *A consultation phase ϕ is* successful *if and only if it returns an up-to-date value $val(\phi)$. A propagation phase ρ is* successful *if and only if it propagates a tag $tag(\rho)$ largest than any of the tags that were in the system when ρ started. A phase is* unsuccessful *if it is not successful.*

We refer to successful operations as operations whose consultation phase and propagation phase are successful. Observe that this corresponds to the notion of successful operation previously given in Definition 4.1.2.

TQS ensures that two active quorums will intersect with high probability, however, if no quorum is active, then the value of an object does no longer persist. To ensure that new operations replicate the object value sufficiently, we assume that at last one operation is executed every period Δ. As previously explained this mechanism serves as a continuous replication and replaces the traditional reconfiguration mechanism to cope with accumulated failures.

4.3.2 Disseminating Memory using Underlying Gossip

In the following, we present a completely structureless memory. The quorum systems this memory uses does not rely on any structure which makes it flexible. In contrast to using a logical structured overlay for communication among members, we use an unstructured communication overlay. The lack of structure presents several benefits. First, there is no need to readapt the structure at each dynamic event. Second, there is no need for detecting failure. Since failure detectors are impossible to achieve in asynchronous settings [FLP85], the absence of failure detector strengthens the feasibility of our solution. Moreover, traditional solution based on failure-detector needs heart-beat message to try detecting failures. Here, no such costly mechanisms are needed.

Our solution proposes a periodic replication. Even though reconfiguration is useless in an unstructured system, some dynamism-related issues have to be addressed. For instance, to ensure

the persistence of an object value despite unbounded leaves, the value must be replicated an unbounded number of times. The solution we propose relies on periodic operations. This periodicity avoids failure detector but requires to be carefully tuned. If replication is not frequent enough, then the value might be lost. Conversely, if replication is too frequent then too many messages would result in a bandwidth resource waste.

Replicating during client operations. Benefiting from the natural primitive of the distributed shared memory, the object value is replicated using operations. The distributed shared object is accessed by client through read and write operation. Like explained along with this document, any operation has at its heart a quorum-probe that replicates value. It has been largely observed that replication is needed to tolerate bounded amount of failures. In large-scale systems, it is also reasonable to assume that shared objects are frequently accessed because of the large number of participants. The replication mechanism for structureless memory has been motivated by these observations. Since operations provide replication and shared objects experience frequent operation requests in large-scale systems, frequent replications are mainly ensured by client operations.

Consequently, replication does not produce a significant communication overhead regarding to the communication complexity of operations. More precisely, as long as operations are frequent enough, replication is not required. When the communication complexity is high due to the numerous participants, then there is no additional replication mechanism and additional complexity is null. However, at some time when operations frequency decreases, the object value must be replicated to prevent unavailability. Observe that communication complexity induced by this replication compensates the absence of operation communication.

Quorum probe. The algorithm is divided in three distinct parts that represent the state of the algorithm (Lines 1–12), the actions initiated by a client (Lines 13–40), and the actions taken upon reception of messages by a node (Lines 41–66), respectively. Each node i has its own copy of the object called its value val_i and an associated tag tag_i. Field *tag* is a couple of a counter and a node identifier and represents, at any time, the version number of the value *val*. We assume that initially, there are q nodes that own the default value of the object, the others nodes have their values *val* set to $\perp$ and all their *tag*s are set to $\langle 0,0\rangle$.

Each read and write operation is executed by client i in two subsequent phases, each disseminating a message to $q = O(\sqrt{nD})$ nodes, where $D = 1/(1-c)^{\Delta}$ is required to handle churn c during period Δ.[2] The two subsequent phases are called the *consultation phase* and the *propagation phase*. The consultation phase aims at consulting the up-to-date value of the object that is present in the system. (This value is identifiable since it associates the largest tag present in the system.) More precisely, client i disseminates a consultation message to q nodes so that each receiver j responds with a message containing value val_j and tag tag_j so that client i can update val_i and tag_i. In fact, i updates val_i and tag_i if and only if the tag_i has either a smaller counter than tag_j or it has an

[2] In [MRWW01], it has been showed that $q = O(\sqrt{n})$ is sufficient in static systems.

Algorithm 8 Disseminating Memory at node i

State of node i:
  $q = \frac{\beta\sqrt{n}}{(1-c)^{\frac{\Delta}{2}}}$, the quorum size
  $\ell, k \in \mathbb{N}$ the disseminating parameters taken such that $\frac{k^{l+1}-1}{k-1} \geq q$
  $val \in V$, the value of the object, initially $\perp$
  tag, a couple of fields:
    $counter \in \mathbb{N}$, initially 0
    $id \in I$, an identifier initially i
  marked, an array of boolean initially false at all indices
  sent-to-nbrs1, *sent-to-nbrs2* two sets of node identifiers, initially $\emptyset$
  rcvd-from-qnodes, an infinite array of identifier sets, initially $\emptyset$ at all indices
  $sn \in \mathbb{N}$, the sequence number of the current phase, initially 0
  $father \in I$, the id of the node that disseminated a message to i; initially i

Read$_i$:
  $\langle val, tag \rangle \leftarrow$ **Consult**()
  Propagate($\langle val, tag \rangle$)

Write(v)$_i$:
  $\langle *, tag \rangle \leftarrow$**Consult**()
  $tag.counter \leftarrow tag.counter + 1$
  $tag.id \leftarrow i$
  $val \leftarrow v$
  Propagate($\langle val, tag \rangle$)

Consult$_i$:
  $ttl \leftarrow \ell$
  $sn \leftarrow sn + 1$
  while ($|$*sent-to-nbrs1*$| < k$) **do**
    send$\langle$CONS$, val, tag, ttl, i, sn\rangle$ to a set J of $(k - |$*sent-from-nbrs1*$|)$ neighbors $\neq$ *father*
    sent-to-nbrs1 $\leftarrow$ *sent-to-nbrs1* $\cup J$
  end while
  sent-to-nbrs1 $\leftarrow \emptyset$
  wait until $|$*rcvd-from-qnodes*$[sn]| \geq q$
  return ($\langle val, tag \rangle$)

equal counter but a smaller identifiers $i < j$ (node identifiers are always distinct); in this case we say $tag_i < tag_j$ for short (cf. Lines 51 and 55). Ideally, at the end of the consultation phase client i has set its value val_i to the up-to-date value. Read and write operations differ from the value and tag that are propagated by the client i. Specifically, in case of a read, client i propagates the value and tag pair freshly consulted, while in the case of write, client i propagates the new value to write associated with a strictly larger tag than the largest tag that has been consulted so far. The propagation phase propagates the corresponding value and tag by dissemination among nodes.

32: **Propagate(**$\langle$ *val*,*t* $\rangle$**)**$_i$**:**
33: $ttl \leftarrow \ell$
34: $sn \leftarrow sn+1$
35: **while** $(|\textit{sent-to-nbrs1}| < k)$ **do**
36: send$\langle$PROP, *val*, *tag*, *ttl*, *i*, *sn*$\rangle$ to a set J of $(k-|\textit{sent-to-nbrs1}|)$ neighbors $\neq$ *father*
37: $\textit{sent-to-nbrs1} \leftarrow \textit{sent-to-nbrs1} \cup J$
38: **end while**
39: $\textit{sent-to-nbrs1} \leftarrow \emptyset$
40: **wait until** $|\textit{rcvd-from-qnodes}[sn]| \geq q$

41: **Participate**$_i$**:** // activated upon reception of a message.
42: recv$\langle$*type*, *v*, *t*, *ttl*, *client-id*, *sn*$\rangle$ from j
43: **if** $(\textit{marked}[sn])$ **then**
44: send$\langle$*type*, *v*, *t*, *ttl*, *client-id*, *sn*$\rangle$ to a neighbor $\neq j$
45: **else** // this sequence is not marked yet
46: $\textit{marked}[sn] \leftarrow$ true
47: **if** $(\textit{type} =$ CONS$)$ **then**
48: $v \leftarrow val$
49: $t \leftarrow tag$
50: **else if** $(\textit{type} =$ PROP$)$ **then**
51: **if** $tag < t$ **then**
52: $val \leftarrow v$
53: $tag \leftarrow t$
54: **else if** $(\textit{type} =$ RESP$)$ **then**
55: **if** $tag < t$ **then**
56: $val \leftarrow v$
57: $tag \leftarrow t$
58: $\textit{rcvd-from-qnodes}[sn] \leftarrow \textit{rcvd-from-qnodes}[sn] \cup \{j\}$
59: $ttl \leftarrow ttl - 1$
60: **if** $(ttl > 0)$ **then**
61: **while** $(|\textit{sent-to-nbrs2}| < k)$ **do**
62: send$\langle$*type*, *v*, *t*, *ttl*, *client-id*, *sn*$\rangle$ to a set J of $(k-|\textit{sent-to-nbrs2}|)$ neighbors $\neq$ *father*
63: $\textit{sent-to-nbrs2} \leftarrow \textit{sent-to-nbrs2} \cup J$
64: **end while**
65: $\textit{sent-to-nbrs2} \leftarrow \emptyset$
66: send $\langle$RESP, *val*, *tag*, *ttl*, $\perp$, *sn*$\rangle$ to *client-id*

Next, we focus on the dissemination procedure that is at the heart of the consultation and propagation phases. There are two parameters, ℓ, k, that define the way all consultation or propagation messages are disseminated. Parameter ℓ indicates the depth of the dissemination, it is used to set a time-to-live field *ttl* that is decremented at each intermediary node that participates in the dissemination; if $ttl = 0$, then dissemination is complete. Parameter k represents the number of neighbors that are contacted by each intermediary participating node. Together, parameters ℓ and k define the number of nodes that are contacted during a dissemination. This number is $\frac{k^{\ell+1}-1}{k-1}$ (Line 3) and

represents the number of nodes in a balanced tree of depth ℓ and width k. (This value is provable by recurrence on the depth ℓ of the tree.) Observe that the number of nodes that are contacted during a dissemination must be larger than q as written Line 3.

There are three kind of messages denoted by message *type*: CONS, PROP, RESP indicating if the message is a consultation message, a propagation message, or a response to any of the two other messages. When a new phase starts at client i, a time-to-live field *ttl* is set to ℓ and a sequence number *sn* is incremented. This number is used in message exchanges to indicate whether a message corresponds to the right phase. Then the phase proceeds in sending continuously messages to k neighbors waiting for their answer (Lines 25–28 and Lines 35–38). When the k neighbors answer, client i knows that the dissemination is ongoing. Then client i receives all messages until a large enough number q of nodes have responded in this phase with the right sequence number (Lines 30, 40). If so, then the phase is complete.

Observe that during the dissemination, messages are simply marked (if not so), responded (to client i), and refowrarded to other neighbors (until *ttl* is null). Messages are marked by the node i that participates into a dissemination for preventing node i from participating multiple times in the same dissemination (Line: 43). As a result, if node i is asked several times to participate, it first participates (Lines 46–66) and then it asks another node to participate (Lines 43–45). More precisely, if $marked[sn]$ is true, then node i re-forwards messages of sequence number *sn* without decrementing the *ttl*. Observe that phase termination and dissemination termination depends on the number of participants rather than the number of responses: it is important that enough participants participate in each dissemination for the phase to eventually end.

Preventing stale value propagation. It is interesting to understand how a value can be read and written using timed quorum system. First, observe that some quorum might not intersect, though this is very unlikely. There is an intersection between any two quorums with high probability, thus, there might exist a quorum that does intersect any other. The goal of the read operation is to return the most up-to-date value of the object, while the goal of the write is to propagate a new value that must appear as more up-to-date than any other.

Due to probabilistic guarantees, each operation might not satisfy its goal. Indeed, a consultation might fail in contacting any node that has the largest tag and up-to-date value. The subsequent propagation phase tries, in case of a read operation, to propagate a stale value, or, in case of a write operation, to propagate a value with a potentially non-adequate tag. Remark that a write operation whose consultation failed might still associate its value the largest tag. Propagating low tag or stale value may have dramatical consequence on further operations. Since intersection probability depends on the number of nodes that own up-to-date value and largest tag, it is crucial that no stale value overwrite an up-to-date value so as no low tag overwrite the largest tag. To remedy this problem, each node contacted during a propagation updates its current tag-value pair only if the propagation informs it about a more up-to-date value associated a larger tag (cf. Lines 51 and 55).

Contacting participants randomly. In order to contact the participants randomly, we implemented a membership protocol [GKM03]. In this protocol, each node has a set of neighbors called its view $\mathcal{N}_i$, it periodically updates its view and recomputes its set of neighbors. Algorithm 9 provides each node with a set of $m \geq k+1$ neighbors, so that phases of Algorithm 8 disseminate through a tree of width k.

Algorithm 9, presented here, is a variant of the Cyclon algorithm [VGvS05]. This algorithm shuffles the view at each cycle of its execution so that it provides randomness in the choice of neighbors. Moreover, it has been shown by simulation that the communication graph obtained with Cyclon is similar to a random graph where neighbors are picked uniformly among nodes [Iwa05]. Finally, for a different purpose we simulated exactly the same variant of Cyclon described on Algorithm 9. This simulation confirmed our thoughts since the results obtained was really similar to the one obtained with artificial uniformity. This simulation is described in Figure A.5(a) of the Appendix and in [FGJ+07].

Algorithm 9 Gossip-based Neighborhood Management using a variant of Cyclon.

Initial State of node i:
  $\mathcal{N}_i$, the view initially filled of some neighbor entries.
  $m \geq k+1$, the view size.

Active Thread at node i:
  for $j' \in \mathcal{N}_i$ **do**
    $t_{j'} \leftarrow t_{j'} + 1$
  $j \leftarrow j'' : t_{j''} = \max_{j' \in \mathcal{N}_i}(t_{j'})$
  send$(\mathsf{REQ}', \mathcal{N}_i \setminus \{e_j\} \cup \{\langle i, 0\rangle\})$ to j
  recv$(\mathsf{ACK}', \mathcal{N}_j)$ from j
  duplicated-entries $= \{e : e.id \in \mathcal{N}_j \cap \mathcal{N}_i\}$
  $\mathcal{N}_i^{init} \leftarrow \mathcal{N}_i$
  $\mathcal{N}_i \leftarrow \mathcal{N}_j \setminus$ *duplicated-entries* $\setminus \{e_i\}$
  for $e_k \in \mathcal{N}_i^{init}$ **do**
    if $|\mathcal{N}_i| < c$ **then**
      $\mathcal{N}_i \leftarrow \mathcal{N}_i \cup \{e_k\}$

Passive thread at node i activated upon message reception:
  recv$(\mathsf{REQ}', \mathcal{N}_j)$ from j
  send$(\mathsf{ACK}', \mathcal{N}_i)$ to j
  duplicated-entries $= \{e \in \mathcal{N}_j : e.id \in \mathcal{N}_j \cap \mathcal{N}_i\}$
  $\mathcal{N}_i^{init} \leftarrow \mathcal{N}_i$
  $\mathcal{N}_i \leftarrow \mathcal{N}_j \setminus$ *duplicated-entries*
  for $e_k \in \mathcal{N}_i^{init}$ **do**
    if $|\mathcal{N}_i| < m$ **then**
      $\mathcal{N}_i \leftarrow \mathcal{N}_i \cup \{e_k\}$

For the sake of uniformity, the membership procedure specified in Algorithm 9 is similar to the Cyclon algorithm: each node i maintains a view $\mathcal{N}_i$ containing one entry per neighbor. The entry of

a neighbor j corresponds to a tuple containing the neighbor identifier and its age. Node i copies its view, selects the oldest neighbor j of its view, removes the entry e_j of j from the copy of its view, and finally sends the resulting copy to j. When j receives the view, j sends its own view back to i discarding possible pointers to i, and i and j update their view with the one they receive by firstly keeping the entries they received. This variant of Cyclon exchanges all entries of the view at each step and uses two additional parameters.

4.3.3 Correctness and Analysis of a Scalable and Dynamic DSM

This Subsection shows that Algorithm 8 implements a timed quorum system and that it emulates the probabilistic atomic object abstraction defined in Definition 4.1.2. The key points of this proof is to show that quorums are sufficiently re-activated by new operations to face dynamism and that subsequent quorums intersect with very high probability to achieve probabilistic atomicity.

Assumptions. First, we only consider executions starting with at least q nodes that own the default value of the object. In these executions, at least one propagation phase from a successful operation starts every Δ time units and let the time of any phase be bounded by δ time units. We assume that during a propagation that propagates a value v to q nodes and that executes between time t and $t+\delta$, there is at least one instant t' where the q nodes own value v simultaneously. This instant, t', can occur arbitrarily between time t and $t+\delta$. Even if this assumption may not seem realistic since propagation occurs in parallel of churn (i.e., at the time the propagation contacts the q^{th} node the first contacted node may have left the system), our motivations for this assumption comes from the sake of clarity of the proof and we claim that the absence of this assumption leads to the same results.

Second, we assume that Algorithm 9 used as our underlying communication protocol provides each node with a view that represents a set of neighbors uniformly drawn at random among the set of all active nodes. This assumption is reasonable since, as already mentioned, the underlying algorithm is based on Cyclon that shuffles node views and provides communication graph similar to a random graph [Iwa05].

Next, we show that Algorithm 8 implements a probabilistic object. Observe that the liveness part of this proof relies simply on the activity of neighbors, and the fact that messages are eventually received. More precisely, by examination of the code of Algorithm 8 and Algorithm 9, messages are gossiped among neighbors while neighbors are uniformly chosen. It is clear that operation termination depends on eventual message delivery. As a result, only the safety part of the proof follows.

Consistency proof. First Lemma computes the ratio of nodes that leave the system as time elapses, given a churn of c. The result is a bound on the number of nodes that leave and join, and helps computing the probability that up-to-date values remain reachable despite dynamism.

Lemma 4.3.1 *The number of initial nodes that have been replaced after τ time units is at most $C = 1-(1-c)^{\tau}$.*

Proof. We claim that the number of initial nodes that are still in the system after τ time units is at least $n(1-c)^{\tau}$. The proof is by induction on the time instants. Let us remind that c is an upper bound on the percentage of nodes that are replaced in one time unit.

- Base case. At time 1, at least $n - nc = n(1-c)$ nodes have not been replaced.
- Induction case. Let us assume that at time $\tau - 1$, the number of initial nodes that have not been replaced is at least $n(1-c)^{\tau-1}$. Let us consider the time instant τ. The number of initial nodes that are not replaced after τ time units is at least $n(1-c)^{\tau-1} - n(1-c)^{\tau-1}c$, i.e., $n(1-c)^{\tau}$, which proves the claim.

It follows from the previous claim that the number of initial nodes that are replaced during τ time units is at most $n - n(1-c)^{\tau}$. Therefore, $C = (n - n(1-c)^{\tau})/n = 1-(1-c)^{\tau}$. □

The following Lemma gives a lower bound on the number of nodes that own the up-to-date value at any time in the system. (Recall that an up-to-date value is either the value with the largest tag and whose propagation is complete, or any value with a larger tag, but whose propagation is ongoing.)

Lemma 4.3.2 *At any time t in the system, the number of nodes that own an up-to-date value is at least $q(1-c)^{\Delta}$, where Δ is the maximum time between two subsequent propagation starts, q is the quorum size, and c is the churn of the system.*

Proof. With no loss of generality, let $\rho_1, ..., \rho_k$ be all the ongoing propagations at time t and let ρ be the latest successful propagation that is already finished at time t. By definition, all $v(\rho_i)$ for any $i \geq 0$ are the up-to-date values in the system. Propagations $\rho_1, ..., \rho_k$ must all have started after time $t - \delta$. By the periodicity assumption of propagate phase, propagation ρ_0 cannot start earlier than time $t - \Delta + \delta$. Due to propagation ρ_0, there must be q nodes with value $v(\rho)$ between times $t - \Delta + \delta$ and $t - \Delta + 2\delta$.

Since the number of replaced nodes increases as time elapses, assume a worst case scenario in which q nodes own value $v(\rho)$ at time $t_1 = t - \Delta + \delta$, we show that at least $q(1-c)^{\Delta}$ nodes with value $v(\rho)$ remains in the system at time $t_2 = t + \delta$. By Lemma 4.3.1, we know that during period $t_2 - t_1 = \Delta$ exactly $\lfloor q(1-(1-c)^{\Delta}) \rfloor$ nodes with value $v(\rho)$ are replaced. Since propagations $\rho_1, ..., \rho_k$ are ongoing, there may be some successful propagations among those ones that overwrite some node values. Observe that if this overwriting happens only to nodes that already own value $v(\rho_i)$, then the number of nodes with value $v(\rho_i)$ remains at least $q(1-c)^{\Delta}$ at time $t + \delta$; if this overwriting happens to nodes that do not own value $v(\rho_i)$ then this number increases. That is, $q(1-c)^{\Delta}$ is a lower bound on the number of nodes with value $v(\rho_i)$ at time $t + \delta$. □

The following Fact gives a well-known bound on the exponential function, provable using the Euler's method.

Fact 4.3.3 $(1+\frac{x}{n})^n \leq e^x$, *for* $n > |x|$.

Next Lemma lower bounds the probability that any consultation consults an up-to-date value v. Recall that sometime it might happen that a value v' is unsuccessfully propagated. This may happen when a write operation fails in consulting the largest tag just before propagating value v'. Observe that in any case, a successful consultation returns only successfully propagated values.

Lemma 4.3.4 *If the number of nodes that own an up-to-date value is at least* $q(1-c)^\Delta$ *during the whole period of execution of consultation* ϕ*, then consultation* ϕ *succeeds with high probability.* ($\mathcal{P} \geq 1 - e^{-\beta^2}$.)

Proof. The consultation of Algorithm 8 draws uniformly at random q nodes, without replacement. To lower bound the probability $\mathcal{P}$ that any consultation consults an up-to-date value v, we compute the probability that this value is obtained after q drawings with replacement. It is clear that the probability of obtaining a specific node after q drawings is larger without replacement than with replacement. The probability for a node x uniformly chosen at random not to own the value v is $\Pr[x \notin Q] = 1 - \frac{q(1-c)^\Delta}{n}$ that is, the probability not to consult value v after q drawings, with replacement, is $\Pr[x_1 \notin Q, ..., x_q \notin Q] = \left(1 - \frac{q(1-c)^\Delta}{n}\right)^q$. By Fact 4.3.3, $\Pr[x_1 \notin Q, ..., x_q \notin Q] \leq e^{-\frac{q^2}{n}(1-c)^\Delta}$. By replacing the q by the quorum size given at Line 2 of Algorithm 8 in the contrapositive $\mathcal{P} \geq 1 - e^{-\frac{q^2}{n}(1-c)^\Delta}$ we obtain the result $\mathcal{P} \geq 1 - e^{-\beta^2}$. □

This corollary simply concludes the two previous Lemma stating that any consultation executed in the system succeeds by returning an up-to-date value.

Corollary 4.3.5 *Any consultation* ϕ *succeeds with high probability.* ($\mathcal{P} \geq 1 - e^{-\beta^2}$.)

Proof. The result is straightforward from Lemma 4.3.2 and Lemma 4.3.4. □

Last but not least, the two theorems conclude the proof by showing that Algorithm 8 implements a Δ-TQS and verifies probabilistic atomicity.

Theorem 4.3.6 *Algorithm 8 implements a* Δ*-Timed Quorum System, where* Δ *is the maximum time between two subsequent propagation starts.*

Proof. First observe that the set of quorums is the set of subsets of q active nodes over the system at time t. The timed access strategy at time t over the set of all quorums is the uniform access strategy over all quorums since each node is chosen with a uniform access strategy among the active nodes at time t. By Corollary 4.3.5, it is clear that the intersection between two quorums is ensured with high probability as long as one quorum starts being contacted Δ time before the other ends being contacted. □

Theorem 4.3.7 *Algorithm 8 implements a probabilistic atomic object.*

Proof. The proof shows that it exists an ordering $\prec$ defined by the tags such that $\pi_i \prec \pi_j$ is equivalent to either $tag(\pi_i) = tag(\pi_j)$ and π_i is a write and π_j is a read, or $tag(\pi_i) < tag(\pi_j)$. We prove separately for each property of Definition 4.1.2 that the ordering $\prec$ satisfies it.

1. The proof is done in two parts. First, we show that Property 1 holds if consultation phase of operation π_2 obtains an up-to-date value. Second, we show that this consultation phase obtains an up-to-date value with high probability.

   On the one hand, we denote by ϕ_i and by ρ_i the respective consultation phase and propagation phase of any operation π_i. We show by contradiction that Property 1 holds if ϕ_2 consults an up-to-date value. By absurd, assume that it is false. That is, assume that ϕ_2 consults an up-to-date value, the response of π_1 precedes the invocation of π_2, and $\pi_2 \prec \pi_1$. Since ϕ_2 consults an up-to-date value, we have $tag(\phi_2) \geq tag(\pi_1)$. Now there are two cases to consider: either π_2 is a read or a write. First, if π_2 is a write then $tag(\pi_2) > tag(\phi_2) \geq tag(\pi_1)$ by examination of the code of Algorithm 8 (cf. Lines 21). By definition of $\prec$, if $tag(\pi_2) > tag(\pi_1)$ and π_2 is a write, then it cannot happen that $\pi_2 \prec \pi_1$. Second, if π_2 is a read then $tag(\pi_2) = tag(\phi_2) \geq tag(\pi_1)$ by examination of the code of Algorithm 8 (cf. Lines 15). By definition of $\prec$, if $tag(\pi_2) \geq tag(\pi_1)$ and π_2 is a read, then it cannot happen that $\pi_2 \prec \pi_1$. As a result, this contradicts the assumption, showing that Property 1 holds if ϕ_2 obtains an up-to-date value.

   On the other hand, Corollary 4.3.5 shows that any consultation obtains the most up-to-date value with high probability. Since Property 1 holds if a consultation of π_2 consults an up-to-date value, and since any consultation consults an up-to-date value with high probability, the result follows.

2. Property 2 follows simply from the way tags are chosen. Let π_1 and π_2 be any two operations. On the one hand, if π_1 and π_2 are initiated at node i, then they have distinct tag counters. On the other hand, if π_1 and π_2 are initiated at two distinct nodes, then they have distinct tag identifiers i and j. As a result, two operations have different tags and either $tag(\rho_1) > tag(\rho_2)$ or $tag(\rho_1) < tag(\rho_2)$ holds.

3. Property 3 fails only if the read operation is unsuccessful. The probability P_π for an operation π to be unsuccessful is lower than the probability P_ϕ that its consultation ϕ is unsuccessful. Since we know by Corollary 4.3.5 that this later probability P_ϕ is very low ($P_\phi = e^{-\beta^2}$), the probability P_π that an operation is unsuccessful is very low too ($P_\pi \leq e^{-\beta^2}$). It follows that Property 3 holds with high probability ($\geq 1 - e^{-\beta^2}$).

□

Performance analysis. The following Theorems show the performance of our solution by measuring the time complexity and the communication complexity of any operation. More precisely, the first Theorem gives the minimal number of messages required to make an operation while the second Theorem gives the expected time complexity of our solution.

Observe that operations complete provided that sent messages are reliably delivered. Assuming this, an operation completes after contacting $O(\sqrt{nD})$ nodes. The following Theorem shows this result.

Theorem 4.3.8 *An operation completes after having contacted $O(\sqrt{nD})$ nodes.*

Proof. This is straightforward from the fact that termination of the dissemination process is conditioned to the number of distinct nodes contacted: $q = O(\sqrt{nD})$, with $D = (1-c)^{-\Delta}$ (cf. Line 2). Since there are two disseminating phases in each operation, an operation is executed after contacting $O(\sqrt{nD})$ nodes. □

The following Lemma indicates that contacting a quorum of $q = O(\sqrt{n})$ nodes consists in contacting approximately q nodes uniformly at random.

Lemma 4.3.9 *Let n be the total number of nodes and assume that in a trial a node is drawn uniformly at random with replacement. The expectation of the number of trials q' to obtain $q = O(\sqrt{nD})$ distinct nodes is $q' \approx q = O(\sqrt{nD})$.*

Proof. Let $H(n)$ denote the n^{th} Harmonic number in this proof. The goal is to compute the number of trials to get q distinct nodes, that is, there is an analogy between our problem and the coupon collector problem where coupons are successively bought uniformly at random and the goal is to complete the collection. From the coupon collector problem, we know that the waiting time between coupons $i-1$ and i is a random variable with expectation $n/(n-i+1)$. Let q' be the number of trials to obtain q distinct nodes. Its expectation is thus:

$$\begin{aligned} E[q'] &= \sum_{i=1}^{q} \frac{n}{n-i+1}, \\ &= n\left(\sum_{i=1}^{n} \frac{1}{i} - \sum_{j=1}^{n-q} \frac{1}{j}\right), \\ &= n\left(H(n) - H(n-q)\right). \end{aligned} \tag{4.1}$$

The upper and lower bounds of the n^{th} Harmonic number is given by Theorem 2 of [QCCG05] as:

$$\frac{1}{2n+\frac{1}{1-\gamma}-2} \leq H(n) - \ln n - \gamma \leq \frac{1}{2n+\frac{1}{3}},$$

where $\gamma = 0.57721566\ldots$ is the Euler-Mascheroni constant.

Consequently, we have $H(n) - H(n-q) \approx \ln n - \ln(n-q) = -\ln\left(1-\frac{q}{n}\right)$. Since $\ln(1+x) \approx x$ where $|x| < 1$, we have $H(n) - H(n-q) \approx \frac{q}{n}$. Using this approximation in Equation 4.1 leads to:

$$E[q'] \approx n(-\ln\frac{n}{n-q}) \approx q.$$

□

Next Theorem indicates that an operation terminates in $O(\log\sqrt{nD})$ message delays, in expectation.

Theorem 4.3.10 *If messages are not lost, the expected time of an operation is $O(\log\sqrt{nD})$ message delays.*

Proof. The proof relies on the fact that q' nodes are contacted uniformly at random with replacement. By Lemma 4.3.9, in expectation the number q' that must be contacted to obtain q distinct nodes is $q' = O(\sqrt{nD})$. Since all q' nodes are contacted in parallel along a tree of depth ℓ' and width k, the time required to contact all the nodes on the tree is $\ell' = O(\log_k \sqrt{nD})$ message delays. □

4.3.4 Exact Probability for Practical Measurements

From a practical standpoint, high probability must be translated into exact values representing the quality of service. It is crucial for the system designer of a company to know what is the exact expectation provided by a service. For instance, if a server is guaranteed to be up during 99,99% of the time, then the company can estimate the waste because of server crash. Previously, we showed that Algorithm 8 implements probabilistic atomicity, that is, we bound the intersection probability of the quorum system. Differently, we now compute the exact intersection probability between any two quorums that are probed in the same period Δ. The exact probability measurement is of practical interest to predict the success of an operation in a distributed shared memory service.

Refining the model of dynamics. Since the purpose is no longer to achieve high probability but rather an exact probability value, we refine our model of dynamics, using a more complex but more realistic model. We present several measurements of this probability depending on some parameters: churn C, period Δ, and quorum size q. In the previous sections, the number of nodes replaced during period δ is fixed and represents a portion of the system and any of its subsystems. That is, given any subset S of system nodes, $C|S|$ nodes are replaced during period δ in S.

This model can be refined to be more realistic. Even though Cn nodes have been replaced in the system, realistically it is unsured that $C|S|$ nodes have been replaced in S. Indeed, if $|S| \leq nC$, it is possible that all the nodes of subset $|S|$ have been replaced. For the purpose of obtaining a more realistic measure of intersection guarantee we now recompute the number of nodes that have been

replaced as a random variable. We then redraw our result based on the new churn model obtained. More technically, the number of nodes that are replaced during period τ in the system remains $C = 1-(1-c)^{\tau}$ (cf. Lemma 4.3.1), however, the number of nodes that are replaced in any subset S of the system is a random variable that depends on the size $|S|$ of the subset.

Evaluating the exact probability of intersection. Let two quorums $Q(t)$ and $Q(t')$ be two sets of q nodes that are probed at time t and $t+\Delta$, respectively. Next, we focus on the exact probability that $Q(t)$ and $Q(t')$ intersect. Let an *initial* node be a node that belongs to $Q(t)$. Moreover, without loss of generality, let $t = 0$ (hence, $t' = \Delta$). Before evaluating the probability that $Q(t)$ and $Q(t')$ intersect, we determine the exact number α of nodes that leave the system during period Δ. First, recall that C is an upper bound on the percentage of nodes that leave and join the system during Δ time units.

Unlike the previous proof, this theorem assumes the refined model of dynamics and measures precisely the probability that, at time $t' = t+\Delta$, an arbitrary node cannot obtain the object value when it queries q nodes arbitrarily chosen. For this purpose, using result of Lemma 4.3.1 we take the number of elements that have left the system during the period Δ as $\alpha = \lfloor Cn \rfloor = \lfloor (1-(1-c)^{\Delta})n \rfloor$. This number allows us to evaluate the aforementioned probability.

Theorem 4.3.11 *Let $x_1, \ldots, x_q$ be any node in the system at time $t' = t+\Delta$. The probability that none of these nodes belong to the initial quorum is*

$$\frac{\sum_{k=a}^{b}\left[\binom{n+k-q}{q}\binom{q}{k}\binom{n-q}{\alpha-k}\right]}{\binom{n}{q}\binom{n}{\alpha}},$$

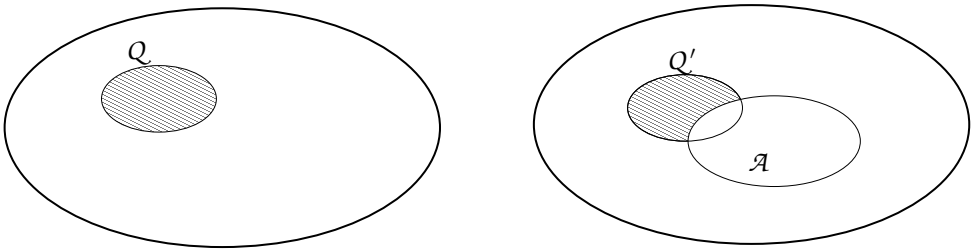
where $\alpha = \lfloor (1-(1-c)^{\Delta})n \rfloor$, $a = \max(0, \alpha - n + q)$, and $b = \min(\alpha, q)$.

Proof. The problem we have to solve can be represented in the following way:

The system is an urn containing n balls (nodes), such that, initially, q balls are green (they represent the initial quorum $Q(t)$ and are represented by the set $\mathcal{Q}$ in Figure 4.4), while the $n-q$ remaining balls are black.

We randomly draw $\alpha = \lfloor Cn \rfloor$ balls from the urn (according to a uniform distribution), and paint them red. These α balls represent the initial nodes that are replaced by new nodes after Δ units of time (each of these balls was initially green or black). After it has been colored red, each of these balls is put back in the urn (so, the urn contains again n balls).

We then obtain the system as described in the right part of Figure 4.4 (which represents the system state at time $t' = t+\Delta$). The set $\mathcal{A}$ is the set of balls that have been painted red. $\mathcal{Q}'$ is the quorum set $\mathcal{Q}$ after some of its balls have been painted red (these balls represent the nodes of the quorum that have left the system). This means the set $\mathcal{Q}' \setminus \mathcal{A}$, that we denote by $\mathcal{E}$, contains all the green balls and only them.



The system at time t The system at time t'

Figure 4.4: System at times t and $t' = t + \Delta$

We denote by β the number of balls in the set $Q' \cap \mathcal{A}$. It is well-known that β has a hypergeometric distribution, i.e., for $a \leq k \leq b$ where $a = \max(0, \alpha - n + q)$ and $b = \min(\alpha, q)$, we have

$$\Pr[\beta = k] \quad = \quad \frac{\binom{q}{k}\binom{n-q}{\alpha-k}}{\binom{n}{\alpha}}. \tag{4.2}$$

We finally draw randomly and successively q balls $x_1, ..., x_q$ from the urn (system at time t') without replacing them. The problem consists in computing the probability of the event {none of the selected balls $x_1, ..., x_q$ are green}, which can be written as $\Pr[x_1 \notin \mathcal{E}, ..., x_q \notin \mathcal{E}]$.

As $\{x \in \mathcal{E}\} \Leftrightarrow \{x \in Q'\} \cap \{x \notin Q' \cap \mathcal{A}\}$, we have (taking the contrapositive) $\{x \notin \mathcal{E}\} \Leftrightarrow \{x \notin Q'\} \cup \{x \in Q' \cap \mathcal{A}\}$, from which we can conclude

$\Pr[x \notin \mathcal{E}] = \Pr[\{x \notin Q'\} \cup \{x \in Q' \cap \mathcal{A}\}]$. As the events $\{x \notin Q'\}$ and $\{x \in Q' \cap \mathcal{A}\}$ are disjoints, we obtain

$\Pr[x \notin \mathcal{E}] = \Pr[x \notin Q'] + \Pr[x \in Q' \cap \mathcal{A}]$. The system contains n balls. The number of balls in Q', $\mathcal{A}$ and $Q' \cap \mathcal{A}$ is equal to q, α and β, respectively.

Since there is no replacement, we get,

$$\Pr[x_1 \notin \mathcal{E}, ..., x_q \notin \mathcal{E} \;/\; \beta = k] \quad = \sum_{k=a}^{b} \prod_{i=1}^{q} \left(1 - \tfrac{q-k}{n-i+1}\right) = \sum_{k=a}^{b} \frac{\binom{n-q+k}{q}}{\binom{n}{q}}. \tag{4.3}$$

To uncondition the aforementioned result (4.3), we simply multiply it by (4.2), leading to

$$\Pr[x_1 \notin \mathcal{E}, \ldots, x_q \notin \mathcal{E}] \quad = \quad \frac{\sum_{k=a}^{b} \left[\binom{n+k-q}{q} \binom{q}{k} \binom{n-q}{\alpha-k} \right]}{\binom{n}{q}\binom{n}{\alpha}}.$$

□

Analyzing quality measures. Now, given a value C set by an application developer, two parameters may influence the overhead of maintaining a quorum in the system and the probabilistic guarantee of having such a quorum. The overhead may be measured in a straightforward manner in this context as the number of nodes that need to be probed, namely q. Intuitively, for a given C, as q increases, the probability of probing a node of the initial quorum increases.

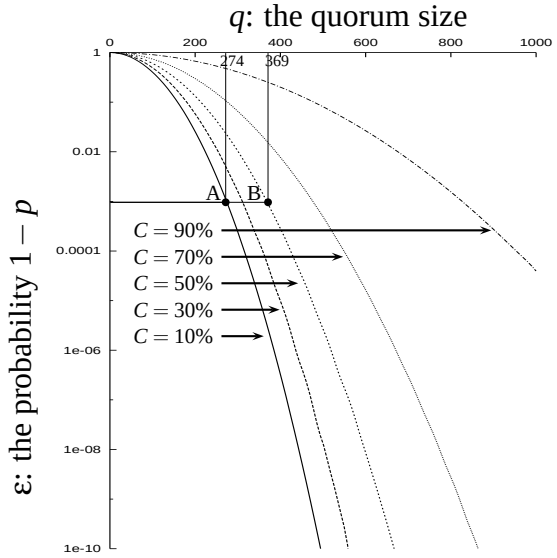


Figure 4.5: Quorum size for the intersection probability $p = 1 - \varepsilon$

Let us consider the value ε determined by Theorem 4.3.11. That value can be interpreted the following way: $p = 1 - \varepsilon$ is the probability that, at time $t' = t + \Delta$, one of the q queries issued (randomly) by a node hits a node of the quorum. An important question is then the following: How are ε and q related? Or equivalently, how increasing the size of q allows decreasing ε? This relation is depicted in Figure 4.5 where several curves are represented for $n = 10,000$ nodes.

Each curve corresponds to a percentage of the initial nodes that have been replaced. (As an example, the curve 30% corresponds to the case where $C = 30\%$ of the initial nodes have left the

system.) Let us consider $\varepsilon = 10^{-3}$. The curves show that $q = 274$ is a sufficient quorum size for not bypassing that value of ε when up to 10% of the nodes are replaced (point A, Figure 4.5). Differently, $q = 274$ is not sufficient when up to 50% of the nodes are replaced; in that case, the size $q = 369$ is required (point B, Figure 4.5).

The curve of Figure 4.5 provides the system designer with realistic hints to set the value of Δ (deadline before which a data transfer protocol establishing a new quorum has to be executed). It shows that, when $10^{-3} \leq \varepsilon \leq 10^{-2}$, the probability $p = 1 - \varepsilon$ increases very rapidly towards 1, though the size of the quorum increases only in a very slight way. As an example, a quorum of $q = 224$ nodes ensures an intersection probability $= 1 - \varepsilon = 0.99$, and a quorum of $q = 274$ nodes ensures an intersection probability $= 1 - \varepsilon = 0.999$.

Interestingly, this phenomenon is similar to the *birthday paradox*[3] [Isa95] that can be roughly summarized as follows. How many persons must be present in a room for two of them to have the same birthday with probability $p = 1 - \varepsilon$? Actually, for that probability to be greater than $1/2$, it is sufficient that the number of persons in the room be equal (only) to 23! When, there are 50 persons in the room, the probability becomes 97%, and increases to 99.9996% for 100 persons. In our case, we observe a similar phenomenon: the probability $p = 1 - \varepsilon$ increases very rapidly despite the fact that the frequency of the quorum size q increases slightly.

In our case, this means that the system designer can choose to slightly increase the size of the probing set q (and therefore only slightly increase the associated overhead) while significantly increasing the probability to access a node of the quorum.

Comparative quorum sizes of static and dynamic systems. In the following we investigate the way the size and lifetime of the quorum are related when the required intersection probability is 99% or 99.9%. We chose these values to better illustrate our purpose, as we believe they reflect what could be expected by an application designer. For both probabilities we present two different figures summarizing the required values of q.

Figure 4.6 focuses on the quorum size that is required in a static system and in a dynamic system (according to various values of the ratio C). The static system implies that no nodes leave or join the system while the dynamic system contains nodes that join and leave the system depending on several churn values. For the sake of clarity we omit values of Δ and simply present C taking several values from 10% to 80%. The analysis of the results depicted in the figure leads to two interesting observations.

First, when Δ is big enough for 10% of the system nodes to be replaced, then the quorum size required is amazingly close to the static case (873 versus 828 when $n = 10^5$ and the probability is 0.999). Moreover, q has to be equal to 990 only when C increases up to 30%. Second, even when Δ is sufficiently large to let 80% of the system nodes be replaced, the minimal number of nodes to probe remains low with respect to the system size. For instance, if Δ is sufficiently large to let $6,000$ nodes be replaced in a system with $10,000$ nodes, then only 413 nodes must be randomly

[3]The paradox is with respect to intuition, not with respect to logics.

Intersection probability	Churn $C = 1-(1-c)^{\Delta}$	Quorum size $n = 10^3$	$n = 10^4$	$n = 10^5$
99%	static	66	213	677 *
	10%	70	224	714
	30%	79	255	809
	60%	105	337	1071
	80%	143	478	1516
99.9%	static	80 *	260	828 *
	10%	85	274	873 *
	30%	96	311	990 *
	60%	128	413 *	1311
	80%	182	584	1855

Figure 4.6: The quorum size depending on the system sizes and the churn rate.

probed to obtain an intersection with probability $p = 0.999$.

4.4 Discussion and Conclusion

4.4.1 Approximating the System Size

An approximation n' is needed in Algorithm 8 to determine the best size for quorum system. Although the exact system size n is impossible to obtain in a large-scale and dynamic systems, many solutions have been proposed in the literature to approximate this value. Since the result obtained are very close, those algorithm can be used to obtain a sufficient approximation.

Several approaches to dynamically, and in a fully decentralized way, estimate the system size exist in the literature. Three main approaches to distributed counting approaches can be distinguished. The first one rely on probabilistic polling approaches. The basic idea of such approaches is to probe the network in a probabilistic way and to infer the size of the systems based on the replies [FT99, KPG+05]. The second approach relies on epidemic algorithms [JM04] and provides very accurate information. The last class of approaches rely on random walks such as the *Sample and collide* algorithm [MLKG06]. A comparison of the efficiency and accuracy of candidates approaches from these three classes can be found in [LKM06].

Most of these solution provides a very good approximation n' of the system size, in a number of steps logarithmic in the system size. Some of these solutions can be coupled to Algorithm 9 very naturally, since they execute using prototypical gossip [CGJ+07].

4.4.2 Modeling the Churn in Dynamic Systems

From a practical point of view, monitoring the dynamism intensity, or churn, is a difficult task for many reasons. First, the dynamic system has to be monitored through a central point that experience no failure. Second, participants may leave the system and modify their state before rejoining. This prevents further identification. In [SGG02, BSV03, SR06], the authors strive to give overviews of the churn in peer-to-peer systems. Their result differ slightly, for instance the last one [SR06] shows, for the first time, that the distribution of peer uptime follows a Weibull law.

From a theoretical point of view, modeling the churn is also a difficult task. As mentioned above, although we have a better understanding of the churn, it suffers from inherent difficulties. In this chapter, we modeled the churn in two manners. First, we modeled the churn as a local rate, such that each subset of the system experiences the same portion of arrivals and the same portion of departures. Second, we modeled the churn as a global rate, such that each subset of the system may experience a quantity of churn that is related to some probability: even if the global churn is low, a large portion of a small subset of the system might be affected by churn but this is unlikely.

In statistics, a mathematical model helps estimating population evolution over time. This model called birth and death process uses continuous-time Markov chain to model the population size: a birth increments the population size while a death decrements it. This model is well-known in queuing theory where an arrival in the queue can be seen as a birth and a leaving as a death. Recently, Markov processes appeared as promising tools for modeling churn in peer-to-peer systems. In [CKKM06], the birth and death process has been suggested for the modeling of replica birth and replica death in a peer-to-peer system. Differently, in [DA06] the authors use discrete-time Markov processes to model two concurrent mechanisms: replication and arrival/departure of nodes. This model uses two distinct Markov processes so that the output state of one is the input state of the other; by averaging one state of each process they approximate a potentially reachable state of the system. An interesting research direction is to model two concurrent mechanisms using continuous-time markov process. As an example, it would be interesting to model arrival/departure mechanism and an individual replication mechanism altogether to understand the impact of the former mechanism on the global performance of the latter mechanism.

4.4.3 Conclusion

This chapter addressed the problem of emulating a distributed shared memory that tolerates scalability and dynamism while being efficient, in terms of operation latency. Motivated by the need of a more realistic model of dynamic system in which deterministic guarantees are almost impossible, this chapter investigated Timed Quorum Systems (TQS) where intersection is timed and holds with high probability.

TQS ensures probabilistic intersection of quorums in a timely fashion. Because the intersection is temporal, such quorum systems are well-suited for dynamic context. Because of probabilistic intersection, such quorums systems match more realistic models of dynamism, where nodes act

independently. Interestingly, we showed that some TQS implementation verifies a consistency criterion weaker but similar to atomicity: probabilistic atomicity. This roughly states that any operation provided by some TQS satisfies the ordering required for atomicity with high probability. The given implementation of TQS verifies probabilistic atomicity, provides lightweight and fast operations, and does not require reconfiguration mechanism since periodic replication is piggybacked into operations. More precisely, the communication complexity of any operation is $O(\sqrt{nD})$, where n is the system size and D is the dynamic parameter; and its time complexity is $O(\log \sqrt{nD})$ message delays. Consequently, if operations are frequent enough for the dynamic parameter, D, to be constant, then this complexity reaches the $O(\sqrt{n})$ complexity shown in [MRWW01] for static systems. Thus, our TQS implementation is optimal in this sense.

The TQS implementation presented in this chapter differs mainly from previous works by the fact that it is structureless. We believe that structureless quorum systems present an interesting direction for further research. Indeed, a structureless memory does not require a client to access a specific predefined set of nodes, but simply require a predefined number of nodes to see or write the value before the client returns. This non-determinism in the choice of the quorum members may translate into powerful operations. For example, an interesting work would be to propose TQS that would use a multiple-source dissemination mechanism instead of the single-source dissemination proposed in this chapter. Such multi-source dissemination would allow a read operation to execute very rapidly by simply *(i)* collecting the number of nodes that have participated in a recent dissemination, and *(ii)* returning the most up-to-date value as soon as the this number of nodes becomes larger than $O(\sqrt{nD})$.

This chapter has enlightened a very fundamental problem: modeling dynamism. As far as we know, it does not exist a realistic model of dynamism intensity (i.e., churn). The misunderstanding of churn is an important issue that limits the application of theoretical solutions to practical dynamic environments like ad-hoc networks or peer-to-peer networks. An important characteristic is the independent behavior of participants that prevents any deterministic solution from being realistically achievable: all solutions that verifies atomicity and whose operations complete, are subject to failures with high probability in any infinite execution. This is due to the fact that at some point too many nodes will leave the memory in a small period of time.

To conclude this chapter, we claim that probabilistic consistency is a prevalent goal for distributed shared memory in dynamic and large-scale systems. Unlike deterministic guarantees, probabilistic ones can provide high quality of service during an arbitrarily long period of time. Modeling dynamism is another promising research direction and would lead to important aids for various kind of problems related to dynamism.

Conclusion and Open Issues

This thesis pointed out that large-scale dynamic systems are very complex environments: First, in such context it is impossible to emulate an efficient and deterministic distributed shared memory (DSM) that satisfies both scalability and dynamism since strong assumptions on the dynamism intensity have to be made. Second, probabilistic DSMs are well-suited for this kind of environment.

Bad News?

This thesis has identified an important tradeoff that prevents from emulating an efficient deterministic DSM that tolerates dynamism and that is scalable. Either tolerating dynamism is very communication costly, or achieving scalability leads to high operation latency. As an example of the two extremes, two distinct solutions have been presented:

1. In Chapter 2, RDS tolerates dynamism by allowing fast decision upon the replacement of failed quorum systems. RDS achieves very efficient operations that may last only two message delays. However, this solution relies on an important degree in the communication graph in order to achieve fast operation and fast reconfiguration. In a large-scale system, the configuration must scale in order to support the load induced by participation. That is, each dynamic event implies to update the state of a tremendous number of nodes, by exchanging a number of messages that scales with the system size while bandwidth do not.

2. In Chapter 3, Square provides scalable memory that tolerates unpredictable requests of a large number of clients. The memory self-adapts its amount of resources in face of load variation being able to treat any request even if the load is quite high. The reconfiguration is made cost-efficient by restraining the degree of the communication graph. As a result, probing a quorum requires many message delays and executing an operation may be very long. This phenomenon is strengthened when load increases since the memory expands, leading to larger quorums and longer operation latency.

On Classifying Quorum Systems

In an effort to find the best quorum system for DSM in large-scale and dynamic systems, this thesis investigated in details existing quorum systems. As a result of this investigation a new classification of quorum systems showed up, as depicted in Figure 4.7.

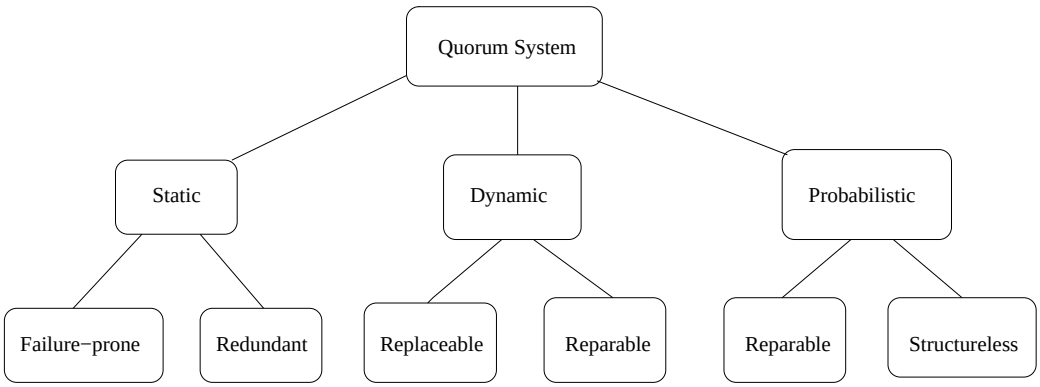


Figure 4.7: Quorum System Classification

Interestingly, this classification gives a natural ranking of quorum systems representing their usability in large-scale dynamic systems. First, at the very left-hand side are the failure-prone quorum systems, which can not tolerate any failure (e.g. the star quorum system). At the right-hand side of the failure-prone ones, the redundant quorum systems can tolerate a bounded number of failures but can not tolerate dynamism (e.g. grid quorum system [Mae85]). Then, the reconfigurable ones may tolerate an arbitrary number of failures while not being scalable (e.g. RDS [CGG+05]). The reparable quorum systems that cope with failures in a scalable way does not provide optimal performance (e.g. square [GAV07]) while probabilistic and reparable ones provide better performance (e.g. dynamic probabilistic quorum systems [AM05]). Finally, structureless quorum system (e.g. timed quorum system [GKM+06]) appears to be the most promising solution for DSM that tolerates dynamism and scalability.

Actually, the classification is the following. First, failure-prone and redundant quorum systems are static. Second, reconfigurable and replaceable quorum systems are both dynamic. Third, some replaceable quorum systems and the structureless ones are both probabilistic quorum systems. The main difference is between the deterministic (static and dynamic quorum systems) and the probabilistic quorum systems, since they offer very different guarantees. An interesting research direction would be to define structureless quorum systems that provide deterministic guarantees. That is, it would tolerate high dynamism while guaranteeing deterministic properties.

To conclude, we are convinced that structureless quorum systems can be adapted for different needs. Guided by mutual exclusion [Ray86], consensus [Lam06a], data retrieval [MTK06], or

other quorum-based applications, researchers could take benefit of the scalability and dynamism of these quorum systems.

Good News?

This thesis claims that probabilistic consistency presents ways to define acceptable consistency conditions and time complexity while achieving ideal performance in terms of communication complexity. This results essentially from the previous tradeoff that translates into the lack of performance when willing to emulate a deterministic DSM: either congestion provokes request losses, or enlarging memory delays operations. There are two major points in favor of probabilistic consistency:

1. First point in favor is that large-scale dynamic systems can not be modeled with participants whose actions are always dependent. Indeed, it is unreasonable to think at a dynamic system in which many participants act altogether so that a small number of nodes leave the system during a small period of time. If the system is large, participants are more likely to act independently and to either join or leave the memory at arbitrary instants. Because of this unpredictability due to the independence of behaviors, many nodes may leave at the same time, even though it is very unlikely. It is far more realistic to admit that nodes act independently and that there exist a small probability that some nodes leave at the same time. In this case, it is only possible to ensure properties with specific probabilities. Chapter 4 has presented probabilistic atomicity as a promising consistency criteria that allows all operations to satisfy atomicity partial ordering with high probability.

2. Second point in favor is that there exist implementations of Timed Quorum System (TQS), with probabilistic requirements, that supersede performance of deterministic solutions. The implementation of TQS presented in Chapter 4 achieves faster operations than Square can do. More generally, any solution that improves on Square by adapting it with a larger degree than its communication graph requires a more costly reconfiguration mechanism, that is completely absent from the TQS implementation. Furthermore, this TQS implementation supersedes RDS by using a constant degree in the underlying communication graph, and by masking reconfiguration power behind existing operations. That is, each node communicates only with a constant number of neighbors and no additional and costly reconfiguration is required.

Finally, as a step towards an effective application of DSM in large-scale dynamic systems, Chapter 4 also outlined that high probability can be translated into some practical quality of service exploitable by system designers.

Future Work and Open Questions

This thesis provides consistency guarantee in large-scale systems. Nowadays, an increasing amount of applications, including web-services, must face variations of participation over time, leading either to a resource overload or to a waste of resource. Providing consistency guarantees despite the lack of control and the dynamism of the environment allows to make many existing applications benefit from the resource multiplication, collaboration, fault-tolerance and low cost of large-scale distributed systems. Nevertheless, several important questions remain open and may complicates such an adaptation.

First, one of the major open issue remains the evaluation of large-scale systems. The observations we presented in this thesis rely either on a dozen of machines experiment, as in Section 2.3.5, or a PeerSim [JMB04] simulation on tens of thousands of nodes, as in Section 3.3.6 and Appendix A. While the former settings do not scale with the number of clients, the latter settings do not simulate the bandwidth constraints. In an attempt to make realistic but smaller scale experiment we also tried a worldwide testbed of hundreds of computers, called PlanetLab [BBC+04]. Additionally, we are currently experimenting slicing algorithms (cf. Appendix A) on an emulation testbed, called EmuLab [WLS+02], that includes about one hundred machines from the University of Utah. Finally, and to evaluate the real cost required by gossip-based protocols, we are developing *GossiPeer* [GKL07], an undergoing project that provides gossip-based protocols as implementation building blocks. Hence, an interesting future work is to develop an implementation of Timed Quorum System on top of GossiPeer, on an as realistic testbed as possible to see if theoretical expectation are practically confirmed.

Second, an important issue of numerous large-scale dynamic systems is security. Indeed, the lack of control and the openness of such systems may suffer from the misbehavior of participants. The model of failure we assumed in this thesis was the possibly unbounded crash-recovery model. An additional type of failures, called *Byzantine* [LF82] or malicious, models the misbehavior of nodes: in such a model some nodes may not respect their specification and thus may act arbitrarily. This Byzantine faults may impact dramatically on the system performance or even lead to impossibility results [FLP85]. Some solutions [MR04], rapidly evoked in this thesis, aims at enlarging the intersection of traditional quorums such that at least a majority of nodes contacted during a quorum access are not Byzantine and can testify of the value propagated to the other quorum. However, as far as we know, all existing solutions require the number of failure to be bounded. A recent improvement appeared with *probabilistic opaque quorum systems* [MR07] and an interesting work would be to extend this work into Timed Opaque Quorum Systems using a model where each node has a fixed probability of being Byzantine during some time but where the number of Byzantine faults is potentially unbounded.

Third, although this thesis is motivated by the fact that resources scale with the size of the system, several social issues may interfere with this idea in open systems like in peer-to-peer networks or more generally in Internet-based applications. Indeed, a non-negligible amount of participants aim at satisfying their own interest despite the common interest. In response to these

issues, many efforts have been devoted to develop incentive mechanisms and tit-for-tat policy-based mechanisms have already been proved efficient to prevent *free-riding* in peer-to-peer file sharing applications [Coh03]. However, these behaviors are more generally inherent to social networking and problems like *lurking* also diminishes the expected power of collaborations in large-scale systems [NP00]. Those questions appear to be very interesting research challenges from an economic or a sociological point of view.

Appendix A

Distributed Slicing in Dynamic Networks

A.1 Introduction

A.1.1 Context and Motivations

The peer to peer (P2P) communication paradigm has now become the prevalent model to build large-scale distributed applications, able to cope with both scalability and system dynamics. This is now a mature technology: P2P systems are slowly moving from application-specific architectures to a generic-service oriented design philosophy. More specifically, P2P protocols integrate into platforms on top of which several applications, with various requirements, may cohabit. This leads to the interesting issue of resource assignment or how to allocate a set of nodes for a given application. Examples of targeted platforms for such a service are telecommunication platforms, where some set of peers may be automatically assigned to a specific task depending on their capabilities, testbed platform such as Planetlab [BBC+04], or desktop-grid-like applications [And04].

Even in a single application, a P2P system should be able to balance the load taking into account that capabilities are heterogeneous at the peers. This ability would be of great interest since many recent works have unveiled the heavy-tailed distribution of storage space, bandwidth, and uptime of peers [SGG02, BSV03, SR06]. Currently, this heterogeneity has two drawbacks. First, the service guarantees offered by the P2P system are unpredictable and can consequently provide the clients with a poor quality of service. Second, when low capable peers are overloaded, the general performance of the system can be affected. For instance, the completely decentralized P2P application, Gnutella [gnua], suffered from congestion when applied to large-scale systems because nodes with a low bandwidth capability were queried. Since then, the Gnutella protocol [gnub] has evolved and tends to request *ultrapeers* (which are peers with larger lifetime and larger bandwidth capabilities), more often than regular peers. Moreover, Kazaa-like applications [kaz] try to benefit similarly from the power of supernodes/superpeers.

Large scale dynamic distributed systems consist of many participants that can join and leave at will. Identifying peers in such systems that have a similar level of power or capability (for instance,

in terms of bandwidth, processing power, storage space, or uptime) in a completely decentralized manner is a difficult task. It is even harder to maintain this information in the presence of churn. Due to the intrinsic dynamics of contemporary P2P systems it is impossible to obtain accurate information about the capabilities (or even the identity) of the system participants. Consequently, no node is able to maintain accurate information about all the nodes. This disqualifies centralized approaches.

The slicing service [JK06] enables peers in a large-scale unstructured network to self-organize into a partitioning, where partitions (slices) are connected overlay networks that represent a given percentage of some resource. Such slices can be either allocated to specific applications later on, or associated with specific roles (e.g., normal peers and superpeers). The slicing is ordered in the sense that peers get ranked according to their capabilities expressed by an attribute value. Building upon the work on ordered slicing proposed in [JK06], here we focus on the issue of *accurate* slicing. That is, we focus on improving quality by slicing the network accurately, and improving stability of slices by minimizing the impact of the churn. Taking this into account, we can summarize the distributed slicing problem we tackle here: we need to rank nodes depending on their relative capability, slice the network depending on these capabilities and, most importantly, readapt the slices continuously to cope with system dynamism.

A.1.2 Contributions

We present two gossip-based solutions to slice the nodes according to their capability (reflected by an attribute value) in a distributed manner with high probability. The first algorithm of the appendix builds upon the ordered slicing algorithm proposed in [JK06] that we call the JK algorithm in the sequel of this appendix. The second algorithm is a different approach based on rank approximation through statistical sampling.

In JK, each node i maintains a random number r_i, picked up uniformly at random (between 0 and 1), and an attribute value a_i, expressing its capability according to a given metric. Each peer periodically gossips with another peer j, randomly chosen among the peers it knows about. If the order between r_j and r_i is different from the order between a_j and a_i, random values are swapped between nodes. The algorithm ensures that eventually the order on the random values matches the order of the attribute ones. The quality of the ranking can then be measured by using a global disorder measure expressing the difference between the exact rank and the actual rank of each peer along the attribute value.

The first contribution is to locally compute a disorder measure so that a peer chooses the neighbor to communicate with in order to maximize the chance of decreasing the global disorder measure. The purpose of this approach is to speed up the convergence. We provide the analysis and experimental results of this improvement.

Then, we identify two issues that prevent accurate slicing and motivate us to find an alternative approach to this algorithm and JK.

On the one hand, once peers are ordered along the attribute values, the slicing in JK takes place

as follows. Random values are used to calculate which slice a node belongs to. For example, a slice containing 20% of the best nodes according to a given attribute, will be composed of the nodes that end up holding random values greater than 0.8. The accuracy of the slicing (independent from the accuracy of the ranking) fully depends on the uniformity of the random value spread between 0 and 1 and the fact that the proportion of random values between 0.8 and 1 is approximately (but usually not exactly) 20% of the nodes. This observation means that the problem of ordering nodes based on uniform random values is not fully sufficient for determining slices.

On the other hand, another motivation for an alternative approach is related to churn and dynamism. It may well happen that the churn is actually correlated to the attribute value. For example, if the peers are sorted according to their connectivity potential, a portion of the attribute space (and therefore the random value space) might be suddenly affected. New nodes will then pick up new random values and eventually the distribution of random values will be skewed towards high values. If this happens we say that the churn is *attribute-correlated*.

The second contribution is an alternative algorithm solving these issues by approximating the rank of the nodes in the ordering locally, without the application of random values. The basic idea is that each node periodically estimates its rank along the attribute axis depending of the attributes it has seen so far. This algorithm is robust and lightweight due to its gossip-based communication pattern: each node communicates periodically with a restricted dynamic neighborhood that guarantees connectivity and provides a continuous stream of new samples. Based on continuously aggregated information, the node can determine the slice it belongs to with a decreasing error margin. We show that this algorithm provides accurate estimation and recovery ability in presence of attributes-correlated churn at the price of a slower convergence.

A.1.3 Related Work

Most of the solutions proposed so far for ordering nodes come from the context of databases, where parallelizing query executions is used to improve efficiency. A large majority of the solutions in this area rely on centralized gathering or all-to-all exchange, which makes them unsuitable for large-scale networks. For instance, the *external sorting problem* [DNS91] consists in providing a distributed sorting algorithm where the memory space of each processor does not necessarily depend on the input. This algorithm must output a sorted sequence of values distributed among processors. The solution proposed in [DNS91] needs a global merge of the whole information, and thus it implies a centralization of information. Similarly, the *percentile finding* problem [IRV89], which aims at dividing a set of values into equally sized sets, requires a logarithmic number of all-to-all message exchanges.

Other related problems are the selection problem and the ϕ-quantile search. The selection problem [FR75, BFP+72] aims at determining the i^{th} smallest element with as few comparisons as possible. The ϕ-*quantile* search (with $\phi \in (0,1]$) is the problem to find among n elements the $(\phi n)^{th}$ element. Even though these problems look similar to our problem, they aim at finding a specific node among all, while the distributed slicing problem aims at solving a global problem where each

node maintains a piece of information. Additionally, solutions to the quantile search problem like the one presented in [KDG03] use an approximation of the system size. The same holds for the algorithm in [SDCM06], which uses similar ideas to determine the distribution of a utility in order to isolate peers with high capability—i.e., super-peers.

As far as we know, the distributed slicing problem was studied in a P2P system for the first time in [JK06]. In this paper, a node with the k^{th} smallest attribute value, among those in a system of size n, tries to estimate its normalized index k/n. The *JK algorithm* proposed in [JK06] works as follows. Initially, each node draws independently and uniformly a random value in the interval $(0,1]$ which serves as its first estimate of its normalized index. Then, the nodes use a variant of Newscast [JMB05] to gossip among each other to exchange random values when they find that the relative order of their random values and that of their attribute values do not match. This algorithm is robust in face of frequent dynamics and guarantees a fast convergence to the same sequence of peers with respect to the random and the attribute values. At every point in time the current random value of a node serves to estimate the slice to which it belongs (its slice).

A.1.4 Outline

The rest of Appendix A is organized as follows: The system model is presented in Section A.2. The first contribution of an improved ordered slicing algorithm based on random values is presented in Section A.3 and the second algorithm based on dynamic ranking in Section A.4. Section A.5 concludes Appendix A.

A.2 Model and Problem Statement

A.2.1 System Model

We consider a system Σ containing a set of n uniquely identified nodes.[1] The set of identifiers is denoted by $I \subset \mathbb{N}$. Each node can leave and new nodes can join the system at any time, thus the number of nodes is a function of time. Nodes may also crash. In Appendix A, we do not differentiate between a crash and a voluntary node departure.

Each node i maintains a fixed attribute value $a_i \in \mathbb{N}$, reflecting the node capability according to a specific metric. These attribute values over the network might have an arbitrary skewed distribution. Initially, a node has no global information neither about the structure or size of the system nor about the attribute values of the other nodes.

We can define a total ordering over the nodes based on their attribute value, with the node identifier used to break ties. Formally, we let i precede j if and only if $a_i < a_j$, or $a_i = a_j$ and $i < j$. We refer to this totally ordered sequence as the *attribute-based sequence*, denoted by $A.sequence$. The attribute-based rank of a node i, denoted by $\alpha_i \in \{1,...,n\}$, is defined as the index of a_i in

[1] The value n is observed instantaneously but may vary over time.

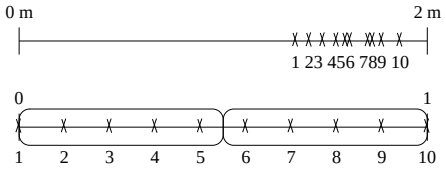


Figure A.1: Slicing of a population based on a height attribute.

A.sequence. For instance, let us consider three nodes: 1, 2, and 3, with three different attribute values $a_1 = 50$, $a_2 = 120$, and $a_3 = 25$. In this case, the attribute-based rank of node 1 would be $\alpha_1 = 2$. In the rest of Appendix A, we assume that nodes are sorted according to a single attribute and that each node belongs to a unique slice. The sorting along several attributes is out of our scope.

A.2.2 Distributed Slicing Problem

Let $\mathcal{S}_{l,u}$ denote the *slice* containing every node i whose normalized rank, namely $\frac{\alpha_i}{n}$, satisfies $l < \frac{\alpha_i}{n} \leq u$ where $l \in [0,1)$ is the slice lower boundary and $u \in (0,1]$ is the slice upper boundary so that all slices represent adjacent intervals $(l_1,u_1],(l_2,u_2]$... Let us assume that we partition the interval $(0,1]$ using a set of slices, and this partitioning is known by all nodes. The distributed slicing problem requires each node to determine the slice it currently belongs to. Note that the problem stated this way is similar to the ordering problem, where each node has to determine its own index in *A.sequence*. However, the reference to slices introduces special requirements related to stability and fault tolerance, besides, it allows for future generalizations when one considers different types of categorizations.

Figure A.1 illustrates an example of a population of 10 persons, to be sorted against their height. A partition of this population could be defined by two slices of the same size: the group of short persons, and the group of tall persons. This is clearly an example where the distribution of attribute values is skewed towards 2 meters. The rank of each person in the population and the two slices are represented on the bottom axis. Each person is represented as a small cross on these axes.[2] Each slice is represented as an oval. The slice $S_1 = \mathcal{S}_{0,\frac{1}{2}}$ contains the five shortest persons and the slice $S_2 = \mathcal{S}_{\frac{1}{2},1}$ contains the five tallest persons.

Observe that another way of partitioning the population could be to define the group of short persons as that containing all the persons shorter than a predefined measure (e.g., $1.65m$) and the group of tall persons as that containing the persons taller than this measure. However, this way of partitioning would most certainly lead to have empty groups that contains no nodes (while a slice is almost surely non-empty). Since the distribution of attribute values is unknown and hard to predict, defining relevant groups is a difficult task. For example, if the distribution of the human heights

[2]Note that the shortest (resp. largest) rank is represented by a cross at the extreme left (resp. right) of the bottom axis.

were unknown, then the persons taller than $1m$ could be considered as tall and the persons shorter than $1m$ could be considered as short. In this case, the first of the two groups would be empty, while the second of the two groups would be as big as the whole system. Conversely, slices partition the population into subsets representing a predefined portion of this population. Therefore, in the rest of Appendix A, we consider slices as defined as a proportion of the network.

A.2.3 Facing Churn

Node churn, that is, the continuous arrival and departure of nodes is an intrinsic characteristic of P2P systems and may significantly impact the outcome, and more specifically the accuracy of the slicing algorithm. The easier case is when the distribution of the attribute values of the departing and arriving nodes are identical. In this case, in principle, the arriving nodes must find their slices, but the nodes that stay in the system are mostly able to keep their slice assignment. Even in this case however, nodes that are close to the border of a slice may expect frequent changes in their slice due to the variance of the attribute values, which is non-zero for any non-constant distribution. If the arriving and departing nodes have different attribute distributions, so that the distribution in the actual network of live nodes keeps changing, then this effect is amplified. However, we believe that this is a realistic assumption to consider that the churn may be correlated to some specific values (for example if the considered attribute is uptime mean or connectivity).

A.3 Dynamic Ordering by Exchange of Random Values

This section proposes an algorithm for the distributed slicing problem improving upon the original JK algorithm [JK06], by considering a local measure of the global disorder function. In this section we present the algorithm along with the corresponding analysis and simulation results.

A.3.1 On Using Random Numbers to Sort Nodes

This Section presents the algorithm built upon JK. We refer to this algorithm as *mod-JK* (standing for modified JK). In JK, each node i generates a real number $r_i \in (0,1]$ independently and uniformly at random. The key idea is to sort these random numbers with respect to the attribute values by swapping (i.e., exchanging) these random numbers between nodes, so that if $a_i < a_j$ then $r_i < r_j$. Eventually, the attribute values (that are fixed) and the random values (that are exchanged) should be sorted in the same order. That is, each node would like to obtain the x^{th} largest random number if it owns the x^{th} largest attribute value. Let *R.sequence* denote the *random sequence* obtained by ordering all nodes according to their random number. Let $\rho_i(t)$ denote the index of node i in *R.sequence* at time t. When not required, the time parameter is omitted.

To illustrate the above ideas, consider that nodes 1, 2, and 3 from the previous example have three distinct random values: $r_1 = 0.85$, $r_2 = 0.1$, and $r_3 = 0.35$. In this case, the index ρ_1 of node

1 would be 3. Since the attribute values are $a_1 = 50$, $a_2 = 120$, and $a_3 = 25$, the algorithm must achieve the following final assignment of random numbers: $r_1 = 0.35$, $r_2 = 0.85$, and $r_3 = 0.1$.

Once sorted, the random values are used to determine the portion of the network a peer belongs to.

A.3.2 Definitions

View. Every node i keeps track of some neighbors and their age. The *age* of neighbor j is a timestamp, t_j, set to 0 when j becomes a neighbor of i. Thus, node i maintains an array containing the id, the age, the attribute value, and the random value of its neighbors. This array, denoted $\mathcal{N}_i$, is called the *view* of node i. The views of all nodes have the same size, denoted by c.

Misplacement. A node participates in the algorithm by exchanging its rank with a misplaced neighbor in its view. Neighbor j is misplaced if and only if

- $a_i > a_j$ and $r_i < r_j$, or
- $a_i < a_j$ and $r_i > r_j$.[3]

We can characterize these two cases by the predicate $(a_j - a_i)(r_j - r_i) < 0$.

Global Disorder Measure. In [JK06], a measure of the relative disorder of sequence *R.sequence* with respect to sequence *A.sequence* was introduced, called the *global disorder measure (GDM)* and defined, for any time t, as

$$GDM(t) = \frac{1}{n}\sum_i (\alpha_i - \rho(t)_i)^2.$$

The minimal value of GDM is 0, which is obtained when $\rho(t)_i = \alpha_i$ for all nodes i. In this case the attribute-based index of a node is equal to its random value index, indicating that random values are ordered.

A.3.3 Improved Ordering Algorithm

In this algorithm, each node i searches its own view $\mathcal{N}_i$ for misplaced neighbors. Then, one of them is chosen to swap random value with. This process is repeated until there is no global disorder. In this version of the algorithm, we provide each node with the capability of measuring disorder locally. This leads to a new heuristic for each node to determine the neighbor to exchange with which decreases most the disorder.

[3] Note that j is not misplaced in case $a_i = a_j$, regardless of values r_i and r_j.

The proposed technique attempts to decrease the global disorder in each exchange as much as possible via selecting the neighbor from the view that minimizes the local disorder (or, equivalently, maximizes the order *gain*) as defined below. Referring to this disorder measure as a criterion, the decrease of the global criterion is related to the decrease of local criteria, similarly to [ADGR05].

For a node i to evaluate the gain of exchanging with a node j of its current view $\mathcal{N}_i$, we define its *local disorder measure* (abbreviated LDM_i). Let $LA.sequence_i$ and $LR.sequence_i$ be the local attribute sequence and the local random sequence of node i, respectively. These sequences are computed locally by i using the information $\mathcal{N}_i \cup \{i\}$. Similarly to *A.sequence* and *R.sequence*, these are the sequences of neighbors where each node is ordered according to its attribute value and random number, respectively. Let, for any $j \in \mathcal{N}_i \cup \{i\}$, $\ell\rho_j(t)$ and $\ell\alpha_j(t)$ be the indices of r_j and a_j in sequences $LR.sequence_i$ and $LA.sequence_i$, respectively, at time (t). At any time t, the local disorder measure of node i is defined as:

$$LDM_i(t) = \frac{1}{c+1} \sum_{j \in \mathcal{N}_i(t) \cup \{i\}} (\ell\alpha_j(t) - \ell\rho_j(t))^2.$$

We denote by $G_{i,j}(t+1)$ the reduction on this measure that i obtains after exchanging its random value with node j between time t and $t+1$. We define it as:

$$\begin{aligned} G_{i,j}(t+1) &= LDM_i(t) - LDM_i(t+1), \\ G_{i,j}(t+1) &= [(\ell\alpha_i(t) - \ell\rho_i(t))^2 + (\ell\alpha_j(t) - \ell\rho_j(t))^2 - \\ &\quad (\ell\alpha_i(t) - \ell\rho_j(t))^2 - (\ell\alpha_j(t) - \\ &\quad \ell\rho_i(t))^2] \frac{1}{c+1}. \end{aligned} \tag{A.1}$$

The heuristic used chooses for node i the misplaced neighbor j that maximizes $G_{i,j}(t+1)$.

Sampling uniformly at random. The algorithm relies on the fact that potential misplaced nodes are found so that they can swap their random numbers thereby increasing order. If the global disorder is high, it is very likely that any given node has misplaced neighbors in its view to exchange with. Nevertheless, as the system gets ordered, it becomes more unlikely for a node i to have misplaced neighbors. In this stage the way the view is composed plays a crucial role: if fresh samples from the network are not available, convergence can be slower than optimal.

Several protocols may be used to provide a random and dynamic sampling in a P2P system such as Newscast [JMB05], Cyclon [VGvS05] or Lpbcast [JGKvS04]. They differ mainly by their *closeness* to the uniform random sampling of the neighbors and the way they handle churn. We chose to use a variant of the Cyclon protocol, to construct and update the views, as it is reportedly the best approach to achieve a uniform random neighbor set for all nodes [Iwa05].

Initial state of node i

(1) $period_i$, initially set to a constant;
r_i, a random value chosen in $(0,1]$; a_i, the attribute value;
$slice_i \leftarrow \perp$, the slice i belongs to; $\mathcal{N}_i$, the view;
$gain_{j'}$, a real value indicating the gain achieved by
exchanging with j';
gain-max $= 0$, a real.

Active thread at node i

(2) wait($period_i$)
(3) recompute-view()$_i$
(4) **for** $j' \in \mathcal{N}_i$
(5) **if** $gain_{j'} \geq$ *gain-max* **then**
(6) *gain-max* $\leftarrow gain_{j'}$
(7) $j \leftarrow j'$
(8) **end for**
(9) send(REQ, r_i, a_i) to j
(10) recv(ACK, r'_j) from j
(11) $r_j \leftarrow r'_j$
(12) **if** $(a_j - a_i)(r_j - r_i) < 0$ **then**
(13) $r_i \leftarrow r_j$
(14) $slice_i \leftarrow \mathcal{S}_{l,u}$ such that $l < r_i \leq u$

Passive thread at node i **activated upon reception**

(15) recv(REQ, r_j, a_j) from j
(16) send(ACK, r_i) to j
(17) **if** $(a_j - a_i)(r_j - r_i) < 0$ **then**
(18) $r_i \leftarrow r_j$
(19) $slice_i \leftarrow \mathcal{S}_{l,u}$ such that $l < r_i \leq u$

Figure A.2: Dynamic ordering by exchange of random values.

Description of the algorithm. The algorithm is presented in Figure A.2. The active thread at node i runs the membership (gossiping) procedure (recompute-view()$_i$) and the exchange of random values periodically as mentioned in Chapter 4 using the algorithm presented in Figure 9. As mentioned in this chapter, each node i maintains a view $\mathcal{N}_i$ containing one entry per neighbor. Node i copies its view, selects the oldest neighbor j of its view, removes the entry e_j of j from the copy of its view, and finally sends the resulting copy to j. When j receives the view, j sends its own view back to i discarding possible pointers to i, and i and j update their view with the one they receive by firstly keeping the entries they received.

The algorithm for exchanging random values from node i starts by measuring the ordering that can be gained by swapping with each neighbor (Lines 4–8). Then, i chooses the neighbor $j \in \mathcal{N}_i$ that maximizes gain $G_{i,k}$ for any of its neighbor k. Formally, i finds $j \in \mathcal{N}_i$ such that for any $k \in \mathcal{N}_i$,

we have

$$G_{i,j}(t+1) \quad \geq \quad G_{i,k}(t+1). \tag{A.2}$$

Using the definition of $G_{i,j}$ in Equation (A.1), Equation (A.2) is equivalent to

$$\begin{aligned} &\ell\alpha_i(t)\ell\rho_j(t) + \ell\alpha_j(t)\ell\rho_i(t) - \ell\alpha_j(t)\ell\rho_j(t) \quad \geq \\ &\ell\alpha_i(t)\ell\rho_k(t) + \ell\alpha_k(t)\ell\rho_i(t) - \ell\alpha_k(t)\ell\rho_k(t). \end{aligned} \tag{A.3}$$

In Figure A.2 of node i, we refer to $gain_j$ as the value of $\ell\alpha_i(t)\ell\rho_j(t) + \ell\alpha_j(t)\ell\rho_i(t) - \ell\alpha_j(t)\ell\rho_j(t)$.

From this point on, i exchanges its random value r_i with the random value r_j of node j (Line 11). The passive threads are executed upon reception of a message. In Figure A.2, when j receives the random value r_i of node i, it sends back its own random value r_j for the exchange to occur (Lines 15–16). Observe that the attribute value of i is also sent to j, so that j can check if it is correct to exchange before updating its own random number (Lines 17–18). Node i does not need to receive attribute value a_j of j, since i already has this information in its view and the attribute value of a node never changes over time.

A.3.4 Analysis of Slice Misplacement

In mod-JK, as in JK, the current random number r_i of a node i determines the slice s_i of the node. The objective of both algorithms is to reduce the global disorder as quickly as possible. Algorithm mod-JK consists in choosing one neighbor among the possible neighbors that would have been chosen in JK, plus the GDM of JK has been shown to fit an exponential decrease. Consequently mod-JK experiences also an exponential decrease of the global disorder. Eventually, JK and mod-JK ensure that the disorder has fully disappeared. For further information, please refer to [JK06].

However, the accuracy of the slices heavily depends on the uniformity of the random value spread between 0 and 1. It may happen, that the distribution of the random values is such that some peers decide upon a wrong slice. Even more problematic is the fact that this situation is unrecoverable unless a new random value is drawn for all nodes. This may be considered as an inherent limitation of the approach. For example, consider a system of size 2, where nodes 1 and 2 have the random values $r_1 = 0.1$, $r_2 = 0.4$. If we are interested in creating two slices S_1 and S_2 of equal size ($S_1 = \mathcal{S}_{0,\frac{1}{2}}$ and $S_2 = \mathcal{S}_{\frac{1}{2},1}$), both nodes will wrongly believe to belong to the same slice S_1, since r_1 and r_2 belong to $(0, \frac{1}{2}]$. This wrong estimate holds even after perfect ordering of the random values.

Therefore, an important step is to characterize the inaccuracy of slice assignment and how likely it may happen. To this end, we lower bound the deviation of random values distribution from the mean, and the probability that this happen with only two slices. First of all, consider a slice S_p of length p. In a network of n nodes, the number of nodes that will fall into this slice is a random variable X with a binomial distribution with parameters n and p. The standard deviation

of X is therefore $\sqrt{np(1-p)}$. This means that the relative proportional expected difference from the mean (i.e., np) can be approximated as $\sqrt{(1-p)/(np)}$, which is very large if p is small, in fact, goes to infinity as p tends to zero, although a very large n compensates for this effect. For a "normal" value of p, and a reasonably large network, the variance is very low however.

To stay with this random variable, the following result bounds, with high probability, its deviation from its mean.

Lemma A.3.1 *For any $\beta \in (0,1]$, a slice S_p of length $p \in (0,1]$ has a number of peers $X \in [(1-\beta)np, (1+\beta)np]$ with probability at least $1-\varepsilon$ as long as $p \geq \frac{3}{\beta^2 n}\ln(2/\varepsilon)$.*

Proof. The way nodes choose their random number is like drawing n times, with replacement and independently uniformly at random, a value in the interval $(0,1]$. Let $X_1,...,X_n$ be the n corresponding independent identically distributed random variables such that:

$$\begin{cases} X_i & = 1 \text{ if the value drawn by node } i \text{ belongs to } S_p \text{ and} \\ X_i & = 0 \text{ otherwise.} \end{cases}$$

We denote $X = \sum_{i=1}^{n} X_i$ the number of elements of interval S_p drawn among the n drawings. The expectation of X is np. From now on we compute the probability that a bounded portion of the expected elements are misplaced. Two Chernoff bounds [MR95] give:

$$\left.\begin{array}{ll} \Pr[X \geq (1+\beta)np] & \leq e^{-\frac{\beta^2 np}{3}} \\ \Pr[X \leq (1-\beta)np] & \leq e^{-\frac{\beta^2 np}{2}} \end{array}\right\}$$

$\Rightarrow \Pr[|X-np| \geq \beta np] \leq 2e^{-\frac{\beta^2 np}{3}}$,
with $0 < \beta \leq 1$. That is, the probability that more than (β time the number expected) elements are misplaced regarding to interval S_p is bounded by $2e^{-\frac{\beta^2 np}{3}}$. We want this to be at most ε. This yields the result. □

To measure the effect discussed above during the simulation experiments, we introduce the slice disorder measure (SDM) as the sum over all nodes i of the distance between the slice i actually belongs to and the slice i believes it belongs to. For example (in the case where all slices have the same size), if node i belongs to the 1^{st} slice (according to its attribute value) while it thinks it belongs to the 3^{rd} slice (according to its rank estimate) then the distance for node i is $|1-3| = 2$. Formally, for any node i, let S_{u_i,l_i} be the actual correct slice of node i and let $S_{\hat{u}_i,\hat{l}_i}(t)$ be the slice i estimates as its slice at time t. The slice disorder measure is defined as:

$$SDM(t) = \sum_i \frac{1}{u_i - l_i} \left| \frac{u_i + l_i}{2} - \frac{\hat{u}_i + \hat{l}_i}{2} \right|.$$

$SDM(t)$ is minimal (equals 0) if for all nodes i, we have $S_{\hat{u}_i,\hat{l}_i}(t) = S_{u_i,l_i}$.

In fact, it is simple to show that, in general, the probability of dividing n peers into two slices of the same size is less than $\sqrt{2/n\pi}$. This value is very small even for moderate values of n. Hence, it is highly possible that the random number distribution does not lead to a perfect division into slices.

A.3.5 Simulation Results

We present simulation results using PeerSim [JMB04], using a simplified cycle-based simulation model, where all messages exchanges are atomic, so messages never overlap. First, we compare the performance of the two algorithms: JK and mod-JK. Second, we study the impact of concurrency that is ignored by the cycle-based simulations.

Performance comparison. We compare the time taken by these algorithms to sort the random values according to the attribute values (i.e., the node with the j^{th} largest attribute value of the system value obtains the j^{th} random value). In order to evaluate the convergence speed of each algorithm, we use the slice disorder measure as defined in Section A.3.4.

We simulated 10^4 participants in 100 equally sized slices (when unspecified), each with a view size $c = 20$. Figure A.3(a) illustrates the difference between the global disorder measure and the slice disorder measure while Figure A.3(b) presents the evolution of the slice disorder measure over time for JK, and mod-JK.

Figure A.3(a) shows the different speed at which the global disorder measure and the slice disorder measure converge. When values are sufficiently large, the GDM and SDM seem tightly related: if GDM increases then SDM increases too. Conversely, there is a significant difference between the GDM and SDM when the values are relatively low: the GDM reaches 0 while the SDM is lower bounded by a positive value. This is because the algorithm does lead to a totally ordered set of nodes, while it still does not associate each node with its correct slice. Consequently the GDM is not sufficient to rightly estimate the performance of our algorithms.

Figure A.3(b) shows the slice disorder measure to compare the convergence speed of our algorithm to that of JK with 10 equally sized slices. Our algorithm converges significantly faster than JK. Note that none of the algorithm reaches zero SDM, since they are both based on the same idea of sorting randomly generated values. Besides, since they both used an identical set of randomly generated values, both converge to the same SDM.

Remark. For the sake of fairness JK and mod-JK are compared using the same underlying view management protocol in our simulation: the variant of Cyclon. Nevertheless, we simulated JK on top of Newscast as it appeared in [JK06] (running a single cycle of Newscast in each cycle of JK, as for Cyclon and its variant in mod-JK). As expected, the convergence speed of JK was even slower due to the difference between the clustering coefficient of the communication graph obtained by

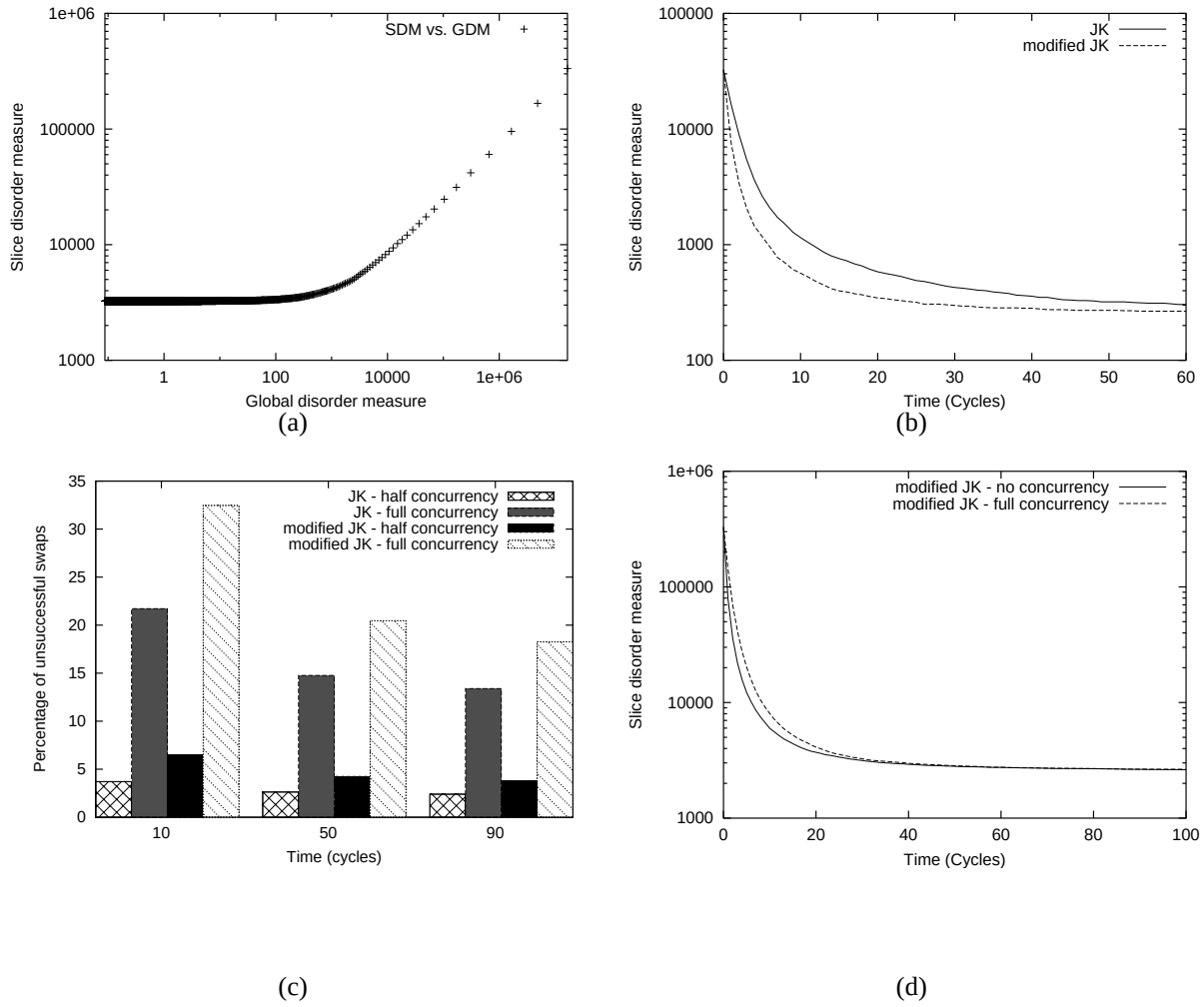


Figure A.3: (a) Contrast between slice disorder measure and global disorder measure, observed on the same experiment. (b) Slice disorder measure over time. (c) Percentage of unsuccessful swaps in the ordering algorithms. (d) Convergence speed under high concurrency.

Newscast and Cyclon, respectively [Iwa05]. The comparison of the underlying view management protocols both in terms of randomness and fault-tolerance is out of the scope of Appendix A.

A.3.6 Concurrency

The simulations are cycle-based and at each cycle an algorithm step is done atomically so that no other execution is concurrent. More precisely, the algorithms are simulated such that in each cycle, each node updates its view before sending its random value or its attribute value. Given this implementation, the cycle-based simulator does not allow us to realistically simulate concurrency, and a drawback is that view is up-to-date when a message is sent. In the following we artificially introduce concurrency (so that view might be out-of-date) into the simulator and show that it has only a slight impact on the convergence speed.

Adding concurrency raises some realistic problems due to the use of non-atomic push-pull [JGKvS04] in each message exchange. That is, concurrency might lead to other problems because of the potential staleness of views: unsuccessful swaps due to useless messages. Technically, the view of node i might indicate that j has a random value r while this value is no longer up-to-date. This happens if i has lastly updated its view before j swapped its random value with another j'. Moreover, due to asynchrony, it could happen that by the time a message is received this message has become useless. Assume that node i sends its random value r_i to j in order to obtain r_j at time t and j receives it by time $t+\delta$. With no loss of generality assume $r_i > r_j$. Then if j swaps its random value with j' such that $r'_j > r_i$ between time t and $t+\delta$, then the message of i becomes *useless* and the expected swap does not occur (we call this an *unsuccessful swap*).

Figure A.3(d) indicates the impact of concurrent message exchange on the convergence speed while Figure A.3(c) shows the amount of useless messages that are sent. Now, we explain how the concurrency is simulated. Let the *overlapping messages* be a set of messages that mutually overlap: it exists, for any couple of overlapping messages, at least one instant at which they are both in-transit. For each algorithm we simulated *(i)* full concurrency: in a given cycle, all messages are overlapping messages; and *(ii)* half concurrency: in a given cycle, each message is an overlapping message with probability $\frac{1}{2}$. Generally, we see that increasing the concurrency increases the number of useless messages. Moreover, in the modified version of JK, more messages are ignored than in the original JK algorithm. This is due to the fact that some nodes (the most misplaced ones) are more likely targeted which increases the number of concurrent messages arriving at the same nodes. Since a node i ignored more likely a message when it receives more messages during the same cycle, it comes out that concentrating message sending at some targets increases the number of useless messages.

Figure A.3(d) compares the convergence speed under full concurrency and no concurrency. We omit the curve of half-concurrency since it would have been similar to the two other curves. Full-concurrency impacts on the convergence speed very slightly.

A.4 Dynamic Ranking by Sampling of Attribute Values

In this section we propose an alternative algorithm for the distributed slicing problem. This algorithm circumvents some of the problems identified in the previous approach by continuously ranking nodes based on observing attribute value information. Random values no longer play a role, so non-perfect uniformity in the random value distribution is no longer a problem. Besides, this algorithm is not sensitive to churn even if it is correlated with attribute values.

In the remaining part of Appendix A we refer to this new algorithm as the ranking algorithm while referring to JK and mod-JK as the ordering algorithms. Here, we elaborate on the drawbacks arising from the ordering algorithms relying on the use of random values that are solved by the ranking approach.

Impact of attribute correlated with dynamics. As already mentioned, the ordering algorithms rely on the fact that random values are uniformly distributed. However, if the attribute values are not constant but correlated with the dynamic behavior of the system, the distribution of random values may change from uniform to skewed quickly. For instance, assume that each node maintains an attribute value that represents its own lifetime. Although the algorithm is able to quickly sort random values, so nodes with small lifetime will obtain the small random values, it is more likely that these nodes leave the system sooner than other nodes. This results in a higher concentration of high random values and a large population of the nodes wrongly estimate themselves as being part of the higher slices.

Inaccurate slice assignments. As discussed in previous sections in detail, slice assignments will typically be imperfect even when the random values are perfectly ordered. Since the ranking approach does not rely on ordering random nodes, this problem is not raised: the algorithm guarantees eventually perfect assignment in a static environment.

Concurrency side-effect. In the previous ordering algorithms, a non negligible amount of messages are sent unnecessarily. The concurrency of messages has a drastic effect on the number of useless messages as shown previously, slowing down convergence. In the ranking algorithm concurrency has no impact on convergence speed because all received messages are taken in account. This is because the information encapsulated in a message (the attribute value of a node) is guaranteed to be up to date, as long as the attribute values are constant, or at least change slowly.

A.4.1 Ranking Algorithm Specification

The pseudocode of the ranking algorithm is presented in Figure A.4. As opposed to the ordering algorithm of the previous section, the ranking algorithm does not assign random initial unalterable

values as candidate ranks. Instead, the ranking algorithm improves its rank estimate each time a new message is received.

The ranking algorithm works as follows. Periodically each node i updates its view $\mathcal{N}_i$ following an underlying protocol that provides a uniform random sample (Line 3); later, we simulate the algorithm using the variant of Cyclon protocol presented in Section A.3.3. Node i computes its rank estimate (and hence its slice) by comparing the attribute value of its neighbors to its own attribute value. This estimate is set to the ratio of the number of nodes with a lower attribute value that i has seen over the total number of nodes i has seen (Line 15). Node i looks at the normalized rank estimate of all its neighbors. Then, i selects the node j_1 closest to a slice boundary (according to the rank estimates of its neighbors). Node i selects also a random neighbor j_2 among its view (Line 12). When those two nodes are selected, i sends an update message, denoted by a flag UPD, to j_1 and j_2 containing its attribute value (Line 13–14).

The reason why a node close to the slice boundary is selected as one of the contacts is that such nodes need more samples to accurately determine which slice they belong to (subsection A.4.2 shows this point). This technique introduces a bias towards them, so they receive more messages.

Upon reception of a message from node i, the passive threads of j_1 and j_2 are activated so that j_1 and j_2 compute their new rank estimate r_{j_1} and r_{j_2}. The estimate of the slice a node belongs to, follows the computation of the rank estimate. Messages are not replied, communication is one-way, resulting in identical message complexity to JK and mod-JK.

A.4.2 Theoretical Analysis

The following Theorem shows a lower bound on the probability for a node i to accurately estimate the slice it belongs to. This probability depends not only on the number of attribute exchanges but also on the rank estimate of i.

Theorem A.4.1 *Let p be the normalized rank of i and let $\hat{p}$ be its estimate. For node i to exactly estimate its slice with confidence coefficient of $100(1-\alpha)\%$, the number of messages i must receive is:*

$$\left(Z_{\frac{\alpha}{2}} \frac{\sqrt{\hat{p}(1-\hat{p})}}{d}\right)^2,$$

where d is the distance between the rank estimate of i and the closest slice boundary, and $Z_{\frac{\alpha}{2}}$ represents the endpoints of the confidence interval.

Proof. Each time a node receives a message, it checks whether or not the attribute value is larger or lower than its own. Let $X_1,...,X_k$ be k $(k>0)$ independent identically distributed random variables described as follows. $X_j = 1$ with probability $\frac{i}{n} = p$ (indicating that the attribute value is lower) and $j \in \{1,...,k\}$, otherwise $X_j = 0$ (indicating the attribute value is larger). By the central limit

Initial state of node i

(1) $period_i$, initially set to a constant; r_i, a value in $(0,1]$; a_i, the attribute value; b, the closest slice boundary to node i; g_i, the counter of encountered attribute values; l_i, the counter of lower attribute values; $slice_i \leftarrow \perp$; $\mathcal{N}_i$, the view.

Active thread at node i

(2) wait($period_i$)
(3) recompute-view()$_i$
(4) *dist-min* $\leftarrow \infty$
(5) **for** $j' \in \mathcal{N}_i$
(6) $\quad g_i \leftarrow g_i + 1$
(7) $\quad$ **if** $a_{j'} \leq a_i$ **then** $\ell_i \leftarrow \ell_i + 1$
(8) $\quad$ **if** dist$(a_{j'}, b) <$ *dist-min* **then**
(9) $\quad\quad$ *dist-min* $\leftarrow$ dist$(a_{j'}, b)$
(10) $\quad\quad$ $j_1 \leftarrow j'$
(11) **end for**
(12) Let j_2 be a random node of $\mathcal{N}_i$
(13) send(UPD, a_i) to j_1
(14) send(UPD, a_i) to j_2
(15) $r_i \leftarrow \ell_i / g_i$
(16) *slice* $\leftarrow \mathcal{S}_{l,u}$ such that $l < r_i \leq u$

Passive thread at node i **activated upon reception**

(17) recv(UPD, a_j) from j
(18) **if** $a_j \leq a_i$ **then** $\ell_i \leftarrow \ell_i + 1$
(19) $g_i \leftarrow g_i + 1$
(20) $r_i \leftarrow \ell_i / g_i$
(21) *slice* $\leftarrow \mathcal{S}_{l,u}$ such that $l < r_i \leq u$

Figure A.4: Dynamic ranking by exchange of attribute values.

theorem, we assume $k > 30$ and we approximate the distribution of $X = \sum_{j=1}^{k} X_j$ as the normal distribution. We estimate X by $\hat{X} = \sum_{j=1}^{k} \hat{X}_j$ and p by $\hat{p} = \frac{\hat{X}}{k}$.

We want a confidence coefficient with value $1-\alpha$. Let Φ be the standard normal distribution function, and let $Z_{\frac{\alpha}{2}}$ be $\Phi^{-1}(1-\frac{\alpha}{2})$. Now, by the Wald large-sample normal test in the binomial case, where the standard deviation of $\hat{p}$ is $\sigma(\hat{p}) = \frac{\sqrt{\hat{p}(1-\hat{p})}}{\sqrt{k}}$, we have:

$$\begin{aligned} \left|\frac{\hat{p}-p}{\sigma(\hat{p})}\right| &\leq Z_{\frac{\alpha}{2}} \\ \hat{p} - Z_{\frac{\alpha}{2}}\sigma(\hat{p}) \leq \quad p \quad &\leq \quad \hat{p} + Z_{\frac{\alpha}{2}}\sigma(\hat{p}). \end{aligned}$$

Next, assume that $\hat{p}$ falls into the slice $S_{l,u}$, with l and u its lower and upper boundaries, respectively. Then, as long as $\hat{p} - Z_{\frac{\alpha}{2}}\sqrt{\frac{\hat{p}(1-\hat{p})}{k}} > l$ and $\hat{p} + Z_{\frac{\alpha}{2}}\sqrt{\frac{\hat{p}(1-\hat{p})}{k}} \leq u$, the slice estimate is exact with a confidence coefficient of $100(1-\alpha)\%$. Let $d = \min(\hat{p}-l, u-\hat{p})$, then we need

$$\begin{aligned} d &\geq Z_{\frac{\alpha}{2}}\sqrt{\frac{\hat{p}(1-\hat{p})}{k}}, \\ k &\geq \left(Z_{\frac{\alpha}{2}}\frac{\sqrt{\hat{p}(1-\hat{p})}}{d}\right)^2. \end{aligned}$$

□

To conclude, under reasonable assumptions all node estimate its slice with confidence coefficient $100(1-\alpha)\%$, after a finite number of message receipts. Moreover a node closer to the slice boundary needs more messages than a node far from the boundary.

A.4.3 Simulation Results

This section evaluates the ranking algorithm by focusing on three different aspects. First, the performance of the ranking algorithm is compared to the performance of the ordering algorithm[4] in a large-scale system where the distribution of attribute values does not vary over time. Second, we investigate if sufficient uniformity is achievable in reality using a dedicated protocol. Third, the ranking algorithm and ordering algorithm are compared in a dynamic system where the distribution of attribute values may change. Finally, a sliding window technique is given to prevent the SDM from increasing.

For this purpose, we ran two simulations, one for each algorithms. The system contains (initially) 10^4 nodes and each view contains 10 uniformly drawn random nodes and is updated in each cycle. The number of slices is 100, and we present the evolution of the slice disorder measure over time.

[4]We omit comparison with JK since the performance obtained with mod-JK are either similar or better.

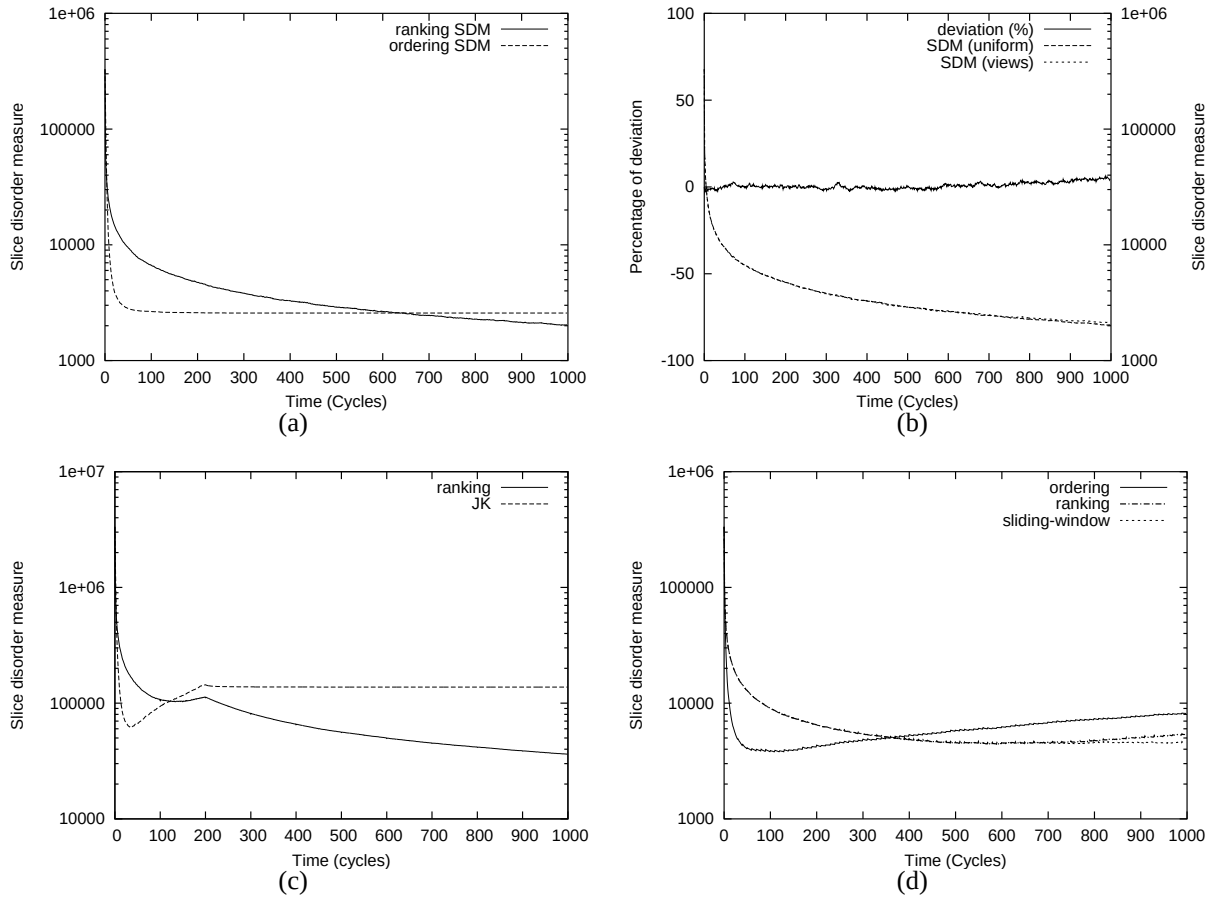


Figure A.5: (a) Comparing performance of the ordering algorithm and the ranking algorithm (static case). (b) Comparing the ranking algorithm on top of a uniform drawing or a Cyclon-like protocol. (c) Effect of burst of attribute-correlated churn on the convergence of the ordering algorithm and the ranking algorithm. (d) Effect of a low and regular attribute-correlated churn on the convergence of the ordering algorithm and the ranking algorithm.

Performance comparison in the static case. Figure A.5(a) compares the ranking algorithm to the ordering algorithm while the distribution of attribute values do not change over time (varying distribution is simulated below).

The difference between the ordering algorithm and the ranking algorithm indicates that the ranking algorithm gives a more precise result (in terms of node to slice assignments) than the ordering algorithm. More importantly, the slice disorder measure obtained by the ordering algorithm is lower bounded while the one of the ranking algorithm is not. Consequently, this simulation shows that the ordering algorithm might fail in slicing the system while the ranking algorithm keeps improving its accuracy over time.

Feasibility of the ranking algorithm. Figure A.5(b) shows that the ranking algorithm does not need artificial uniform drawing of neighbors. Indeed, an underlying view management protocol might lead to similar performance results. In the presented simulation we used an artificial protocol, drawing neighbors randomly at uniform in each cycle of the algorithm execution, and the variant of the Cyclon view management protocol presented above. Those underlying protocols are distinguished on the figure using terms ”uniform” (for the former one) and ”views” (for the latter one). As said previously, the Cyclon protocol [VGvS05] consists of exchanging views between neighbors such that the communication graph produced shares similarities with a random graph. This figure shows that both cases give very similar results. The SDM legend is on the right-handed vertical axis while the left-handed vertical axis indicates what percentage the SDM difference represents over the total SDM value. At any time during the simulation (and for both type of algorithms) its value remains within plus or minus 7%. The two SDM curves of the ranking algorithm almost overlap. Consequently, the ranking algorithm and the variant of Cyclon presented in subection A.3.3 achieve very similar result.

To conclude, the variant of Cyclon algorithm presented in the previous section can be easily used with the ranking algorithm to provide the shuffling of views. More generally, an underlying distributed protocol that shuffles the view among nodes may provide nearly-optimal results.

Performance comparison in the dynamic case. In Figure A.5(c) each of the two curves represents the slice disorder measure obtained over time using the ordering algorithm and the ranking algorithm respectively. We simulate the churn such that 0.1% of nodes leave and 0.1% of the nodes join in each cycle during the 200 first cycles. We observe how the SDM converges. The churn is reasonably and pessimistically tuned compared to recent experimental evaluations [SR06] of the session duration in three well-known P2P systems.[5]

The distribution of the churn is correlated to the attribute value of the nodes. The leaving nodes are the nodes with the lowest attribute values while the entering nodes have higher attribute values than all nodes already in the system. The parameter choices are motivated by the need

[5] In [SR06], roughly all nodes have left the system after 1 day while there are still 50% of nodes after 25 minutes. In our case, assuming that in average a cycle lasts one second would lead to more than 54% of leave in 9 minutes.

of simulating a system in which the attribute value corresponds to the (fixed) session duration of nodes, for example.

The churn introduces a significant disorder in the system which counters the fast decrease. When, the churn stops, the ranking algorithm readapts well the slice assignments: the SDM starts decreasing again. However, in the ordering algorithm, the convergence of SDM gets stuck. This leads to a poor slice assignment accuracy.

In Figure A.5(d), each of the two curves represent the slice disorder measure obtained over time using the ordering algorithm, the ranking algorithm, and a modified version of the ranking algorithm using attribute values recorded in a sliding-window, respectively. (The simulation obtained using sliding windows is described in the next subsection.) The churn is diminished and made more regular than in the previous simulation such that 0.1% of nodes leave and 0.1% of nodes join every 10 cycles.

The curves fits a fast decrease (superlinear in the number of cycles) at the beginning of the simulation. At first cycles, the ordering gain is significant making the impact of churn negligible. This phenomenon is due to the fact that SDM decreases rapidly when the system is fully disordered. Later on, however, the decrease slope diminishes and the churn effect reduces the amount of nodes with a low attribute value while increasing the amount of nodes with a large attribute value. This unbalance leads to a messy slice assignment, that is, each node must quickly find its new slice to prevent the SDM from increasing. In the ordering algorithm the SDM starts increasing from cycle 120. Conversely, with the ranking algorithm the SDM starts increasing not earlier than at cycle 730. Moreover the increase slope is much larger in the former algorithm than in the latter one.

Even though the performance of the ranking algorithm is much better, its adaptiveness to churn is not surprising. Unlike the ordering algorithm, the ranking one keeps re-estimating the rank of each node depending on the attribute values present in the system. Since the churn increases the attribute values present in the system, nodes tend to receive more messages with higher attribute values and less messages with lower attribute values, which turns out to keep the SDM low, despite churn. Further on, we propose a solution based on sliding-window technique to limit the increase of the SDM in the ranking algorithm.

To conclude, the results show that when the churn is related to the attribute (e.g., attribute represents the session duration, uptime of a node), then the ranking algorithm is better suited than the ordering algorithm.

Sliding-window for limiting the SDM increase. In Figure A.5(d), the ”sliding-window” curve presents a slightly modified version of the ranking algorithm that encompasses SDM increase due to churn correlated to attribute values. Here, we present this enrichment.

In Section A.4, the ranking algorithm specifies that each node takes into account all received messages. More precisely, upon reception of a new message each node *i* re-computes immediately its rank estimate and the slice it thinks it belongs to without remembering the attribute values it has seen. Consequently the messages received long-time ago have as much importance as the fresh

messages in the estimate of *i*. The drawback, as it appeared in Figure A.5(d) of Section A.3.5, is that if the attribute values are correlated to churn, then the precision of the algorithm might diminish.

To cope with this issue, the previous algorithm can be easily enriched in the following way. Upon reception of a message, each node *i* records an information about the attribute value received in a fixed-size ordered set of values. Say this set is a first-in first-out buffer such that only the most recent values remain. Right after having recorded this information, node *i* can re-compute its rank estimate and its slice estimate based on the most relevant piece of information (having discarded the irrelevant piece). Consequently, the estimate would rely only on fresh attribute values encountered so that the algorithm would be more tolerant to changes (e.g., dynamics or non-uniform evolution of attribute values). Of course, since the analysis (cf. Section A.4.2) shows that nodes close to the slice boundary require a large number of attribute values for estimating precisely their estimates, it would be unaffordable to record all these last attribute values encountered due to space limitation.

Actually, the only necessary relevant information of a message is simply whether it contains a lower attribute value than the attribute value of *i*, or not. Consequently, a single bit per message would be sufficient to record the necessary information (e.g., adding a 1 meaning that the attribute value is lower, and 0 otherwise). Thus, even though a node *i* would require 10^4 messages to rightly estimate its slice (with high probability), node *i* simply needs to allocate an array of size $10^4/(8*1000) = 1,25$ kB.

As expected, Figure A.5(d) shows that the sliding-window method applied to the ranking algorithm prevents its SDM from increasing. Consequently, at some point in time, the resulting slice assignment may become even more accurate.

A.5 Conclusion

A.5.1 Summary

Peer to peer systems may now be turned into general frameworks on top of which several applications might cohabit. To this end, allocating resources to applications, while resources are heterogeneously spread over the system, require specific algorithms to partition the network in a relevant way. The sorting algorithm proposed in [JK06] provided a first attempt to "slice" the network, taking into account the potential heterogeneity of nodes. This algorithm relies on each node drawing a random value uniformly and swapping continuously those random values, with candidate nodes, so that the order between attributes values (reflecting the capabilities of nodes) and random ones match.

Here, we first proposed an improvement over the initial sorting algorithm resulting in the ordering algorithm. This improvement comes from a judicious choice of candidate nodes to swap values. Each node makes this choice depending on the potential decrease of the disorder measure it can compute locally.

Our second contribution is the definition of the slice disorder measure. The slice disorder measure evaluates how nodes wrongly estimate the slice they belong to. We showed that the existing global disorder measure can not indicate whether nodes found their slice. That is, the slice disorder measure is necessary to show that an algorithm solves the distributed slicing problem.

Using the slice disorder measure, we identified two issues related to the use of static random values. The first one refers to the fact that slice assignment heavily depends on the degree of uniformity of the initial random value. The second is related to the fact that once sorted along one attribute axis, the churn (or failures) might be correlated to the attribute, therefore leading to a unrecoverable skewed distribution of the random values. This phenomenon results in a wrong slice assignment despite the system seems to be rightly ordered.

Last but not least, we provided a ranking algorithm that accurately maintains slices of the system even in the presence of churn. This algorithm minimizes the effect of correlated churn on slice disorder and recovers efficiently after a period of correlated churn. For this purpose, nodes continuously re-estimate their rank relatively to other nodes based on their sampling of the network. The convergence speed up of the first algorithm and the accuracy of the second algorithm are proved through theoretical analysis and simulations.

A.5.2 Perspective

Our solution uses a variant of the Cyclon protocol to obtain quasi-uniform distribution of neighbors. There are various protocols that might be used for different purpose. For instance, Newscast can be used for its resilience to very high dynamics as in [JK06]. Some other protocols exist in the literature. Deciding exactly how to parameterize the underlying peer sampling service might be an interesting future direction.

Appendix B

IOA Specification of a Dynamic DSM

Algorithm 10 Signature of the Reconfigurable Distributed Storage

Signature:
  Input:
    $\mathsf{join}(W)_i$, $i \in I$, W a set of nodes
    read_i, $i \in I$
    $\mathsf{write}(v)_i$, $i \in I$, v a value
    $\mathsf{recon}(c,c')_i$, $i \in I$, c and c' two configurations
    $\mathsf{recv}(m)_i$, $i \in I$, m a message
    fail_i, $i \in I$
    $\mathsf{leader}(b)_i$, $i \in I$, b a ballot

  Output:
    $\mathsf{join\text{-}ack}_i$, $i \in I$
    $\mathsf{read\text{-}ack}(v)_i$, $i \in I$, v a value
    $\mathsf{write\text{-}ack}_i$, $i \in I$
    $\mathsf{recon\text{-}ack}(r)_i$, $i \in I$
    $\mathsf{send}(m)_i$, $i \in I$, m a message

  Internal:
    $\mathsf{query\text{-}fix}_i$, $i \in I$
    $\mathsf{prop\text{-}fix}_i$, $i \in I$
    $\mathsf{prepare}(b)_i$, $i \in I$, b a ballot
    $\mathsf{prepare\text{-}done}(b)_i$, $i \in I$, b a ballot
    $\mathsf{init\text{-}propose}(k)_i$, $i \in I$, k an integer
    $\mathsf{propose}(k)_i$, $i \in I$, k an integer
    $\mathsf{propose\text{-}done}(k)_i$, $i \in I$, k an integer

State:
  status $\in \{\mathsf{idle}, \mathsf{joining}, \mathsf{active}, \mathsf{failed}\}$,
  world, a finite subset of I
  value $\in V$,
  tag $\in T$,
  cmap $\in CMap$,
  pnum1 $\in \mathbb{N}$,
  pnum2, a mapping from I to $\mathbb{N}$
  isLeader $\in B$
  confirmed, a set of tags
  failed $\in B$,
  op, a record with fields:
    type $\in \{\mathsf{read}, \mathsf{write}\}$
    phase $\in \{\mathsf{idle}, \mathsf{query}, \mathsf{prop}, \mathsf{done}\}$
    pnum $\in \mathbb{N}$
    cmap $\in CMap$
    acc, a finite subset of I
    tag $\in T$
    value $\in V$
  pxs, a record with fields:
    pnum $\in \mathbb{N}$
    phase $\in \{\mathsf{idle}, \mathsf{prepare}, \mathsf{propose}, \mathsf{propagate}\}$
    conf-index $\in \mathbb{N}$
    conf $\in C$
    acc, a finite subset of I
  ballot, a ballot with fields:
    id $\in \mathbb{N} \times I$
    conf-index $\in \mathbb{N}$
    conf $\in C$
  voted-ballots, a set of ballots.

Algorithm 11 Reconfigurable Distributed Storage – Operation transitions

Input read_i
  Effect:
    if $\neg failed \wedge status = \mathsf{active}$ **then**
      $pnum \leftarrow pnum + 1$
      $op.pnum \leftarrow pnum$
      $op.type \leftarrow \mathsf{read}$
      $op.phase \leftarrow \mathsf{query}$
      $op.cmap \leftarrow cmap$
      $op.acc \leftarrow \emptyset$

Input $\mathsf{write}(v)_i$
  Effect:
    if $\neg failed \wedge status = \mathsf{active}$ **then**
      $pnum1 \leftarrow pnum1 + 1$
      $op.pnum \leftarrow pnum1$
      $op.type \leftarrow \mathsf{write}$
      $op.phase \leftarrow \mathsf{query}$
      $op.cmap \leftarrow cmap$
      $op.acc \leftarrow \emptyset$
      $op.value \leftarrow v$

Internal $\mathsf{query\text{-}fix}()_i$
  Precondition:
    $\neg failed \wedge status = \mathsf{active}$
    $op.type \in \{\mathsf{read}, \mathsf{write}\}$
    $op.phase = \mathsf{query}$
    for $k \in \mathbb{N}, c \in C$ **do**
      $op.cmap(k) = c \Rightarrow \exists Cons \in consultation\text{-}quorums(c) : Cons \subseteq op.acc$
  Effect:
    if $op.type = \mathsf{read}$ **then**
      $op.value \leftarrow value$
      $op.tag \leftarrow tag$
    else
      $value \leftarrow op.value$
      $tag \leftarrow [[tag.seq + 1, i]]$
      $op.tag \leftarrow tag$
    $pnum1 \leftarrow pnum1 + 1$
    $op.pnum \leftarrow pnum1$
    $op.phase \leftarrow \mathsf{prop}$
    $op.cmap \leftarrow cmap$
    $op.acc \leftarrow \emptyset$

Internal $\mathsf{prop\text{-}fix}()_i$
  Precondition:
    $\neg failed \wedge status = \mathsf{active}$
    $op.type \in \{\mathsf{read}, \mathsf{write}\}$
    $op.phase = \mathsf{query}$
    for $k \in \mathbb{N}, c \in C$ **do**
      $op.cmap(k) = c \Rightarrow \exists P \in propagation\text{-}quorums(c) : P \subseteq op.acc$
  Effect:
    $op.phase \leftarrow \mathsf{done}$
    $confirmed \leftarrow confirmed \cup \{op.tag\}$

Output $\mathsf{read\text{-}ack}(v)_i$
  Precondition:
    $\neg failed \wedge status = \mathsf{active}$
    $op.type = \mathsf{read}$
    $op.phase = \mathsf{done}$
    $v = \mathsf{op.value}$
  Effect:
    $op.phase = \mathsf{idle}$

Output $\mathsf{read\text{-}ack}(v)_i$
  Precondition:
    $\neg failed \wedge status = \mathsf{active}$
    $op.type = \mathsf{read}$
    $op.phase = \mathsf{prop}$
    $op.tag = \mathsf{confirmed}$
    $v = \mathsf{op.value}$
  Effect:
    $op.phase = \mathsf{idle}$

Output $\mathsf{write\text{-}ack}_i$
  Precondition:
    $\neg failed \wedge status = \mathsf{active}$
    $op.type = \mathsf{write}$
    $op.phase = \mathsf{done}$
    $v = \mathsf{op.value}$
  Effect:
    $op.phase = \mathsf{idle}$

```
75: Output send(W, v, t, cnf, cm, pns, pnr, b, p, vb)_{i,j}
76:     Precondition:
77:         ¬failed ∧ status ≠ idle
78:         j ∈ world
79:         W = world
80:         v = value
81:         t = tag
82:         cnf = confirmed
83:         cm = cmap
84:         pnr = pnum1
85:         pns = pnum2(j)
86:         b = ballot
87:         p = pxs
88:         vb = voted-ballots
89:
90:     Effect:
91:         None

92: Input leader(b)_i
93:     Effect:
94:         isLeader ← b

95: Input join(w)_i
96:     Effect:
97:         status ← joining
98:         world ← world ∪ w

99: Output join(w)_i
100:    Precondition:
101:        status = active
102:        joined = false
103:    Effect:
104:        joined ← true

105: Input fail()_i
106:    Effect:
107:        failed ← true

108: Input recv(W, v, t, cnf, cm, pns, pnr, b, p, vb)_{j,i}
109:    Effect:
110:        if ¬failed ∧ status ≠ idle then
111:            status ← active
112:            world ← world ∪ W
113:            confirmed ← confirmed ∪ cnf
114:            if t > tag then
115:                [[value, tag]] ← [[v, t]]
116:            if b.id > ballot.id then
117:                ballot ← b
118:                pxs.phase ← idle
119:                pxs.acc ← ∅
120:            if p.conf-index > pxs.conf-index then
121:                if recon-in-progress = false then
122:                    pxs.conf-index ← p.conf-index
123:                    pxs.conf ← p.conf
124:                    pxs.old-conf ← p.old-conf
125:                    pxs.phase ← idle
126:                    pxs.acc ← ∅
127:            voted-ballots ← voted-ballots ∪ vb
128:            cmap ← update(cmap, cm)
129:            pnum2(j) ← max(pnum2(j), pns)
130:            if op.phase ∈ {query, prop} ∧ pnr ≥ op.pnum then
131:                op.cmap ← extend(op.cmap, cm)
132:                if op.cmap ∈ Truncated then
133:                    op.acc ← op.acc ∪ {j}
134:                else
135:                    op.acc ← ∅
136:                    op.cmap ← cmap
137:            if pxs.phase = prepare then
138:                if pnr ≥ pxs.pnum then
139:                    pxs.acc ← pxs.acc ∪ {j}
140:            else if pxs.phase = propose then
141:                if ballot ∈ vb ∧ ballot = b then
142:                    pxs.acc ← pxs.acc ∪ {j}
143:            else if pxs.phase = propagate then
144:                if cm(ballot.conf-index) = ballot.conf then
145:                    pxs.acc ← pxs.acc ∪ {j}
```

Algorithm 12 Reconfigurable Distributed Storage – Reconfiguration transitions

Input recon$(c, c')_i$
  Effect:
    if $\neg failed \wedge status = \text{active}$ **then**
      let $k = \max(\ell : cmap(\ell) \in C)$
      $pxs.conf\text{-}index \leftarrow k+1$
      $pxs.old\text{-}conf \leftarrow c$
      $pxs.cconf \leftarrow c'$
      $pxs.phase \leftarrow \text{idle}$
      $pxs.acc \leftarrow \emptyset$
      $recon\text{-}in\text{-}progress \leftarrow \text{true}$

Internal init$(c)_i$
  Precondition:
    $\neg failed \wedge status = \text{active}$
    $c = pxs.conf \neq \perp$
    $k = pxs.conf\text{-}index \neq \perp$
    $cmap(k) = \perp$
    $cmap(k-1) = pxs.old\text{-}conf \neq \perp$
    if $k > 1$ **then**
      $cmap(k-2) = \pm$
    $isLeader$
  Effect:
    $pxs.phase \leftarrow \text{idle}$
    $pxs.acc \leftarrow \emptyset$
    $ballot.conf \leftarrow c$
    $ballot.conf\text{-}index \leftarrow k$

Output recon-ack$(r)_i$
  Precondition:
    $\neg failed \wedge status = \text{active}$
    $recon\text{-}in\text{-}progress = \text{true}$
    $cmap(k) \in C \vee cmap(k-2) \neq \pm \vee cmap(k-1) \neq pxs.old\text{-}conf$
    let $k = pxs.conf\text{-}index$
    if $cmap(k) = pxs.conf$ **then**
      $r = \text{ok}$
    else
      $r = \text{failed}$
  Effect:
    $pxs.conf \leftarrow \perp$
    $pxs.conf\text{-}index \leftarrow \perp$
    $recon\text{-}in\text{-}progress \leftarrow \text{false}$

Internal prepare$(b)_i$
  Precondition:
    $\neg failed \wedge status = \text{active}$
    $isLeader$
    $b = ballot$
  Effect:
    $pnum1 \leftarrow pnum1 + 1$
    $pxs.pnum \leftarrow pnum1$
    $pxs.acc \leftarrow \emptyset$
    $ballot.id \leftarrow [[ballot.id.seqno + 1, i]]$
    $pxs.phase \leftarrow \text{prepare}$

Internal prepare-done$(b)_i$
  Precondition:
    $\neg failed \wedge status = \text{active}$
    $isLeader$
    $b = ballot$
    $pxs.phase = \text{prepare}$
    let $k = ballot.conf\text{-}index$
    let $c = cmap(k-1)$
    $c \in C$
    $\exists Cons \in consultation - quorums(c) : Cons \subseteq pxs.acc$
  Effect:
    $pxs.prepared\text{-}id \leftarrow ballot.id$
    $pxs.acc \leftarrow \emptyset$
    $pxs.phase \leftarrow \text{idle}$

65: **Internal** init-propose$(k)_i$
66: **Precondition:**
67: $\neg failed \wedge status = \text{active}$
68: $isLeader$
69: $ballot.conf\text{-}index = k \neq \bot$
70: $ballot.id = pxs.prepared\text{-}id$
71: $pxs.conf\text{-}index = ballot.conf\text{-}index$
72: $pxs.conf = ballot.conf$
73: **Effect:**
74: $pxs.phase \leftarrow \text{idle}$
75: let $S = \{b \in vote\text{-}ballots : b.conf\text{-}index = k\}$
76: **if** $S \neq \emptyset$ **then**
77: let $b' = b'' : b''.id = \max_{b \in S}(b.id)$
78: $ballot.conf \leftarrow b'.conf$
79: $voted\text{-}ballots \leftarrow voted\text{-}ballots \cup \{ballot\}$

80: **Internal** propose$(k)_i$
81: **Precondition:**
82: $\neg failed \wedge status = \text{active}$
83: $ballot.conf\text{-}index = k \neq \bot$
84: $ballot \in voted\text{-}ballot$
85: **Effect:**
86: $pxs.phase \leftarrow \text{propose}$
87: $pnum1 \leftarrow pnum1 + 1$
88: $pxs.pnum \leftarrow pnum1$
89: $pxs.acc \leftarrow \emptyset$

90: **Internal** propose-done$(k)_i$
91: **Precondition:**
92: $\neg failed \wedge status = \text{active}$
93: $pxs\text{-}phase = \text{propose}$
94: let $k = ballot.conf\text{-}index$
95: let $c = cmap(k-1)$
96: $c \in C$
97: $\exists Cons_1 \in consultation - quorums(c) : Cons_1 \subseteq pxs.acc$
98: $\exists Prop_1 \in propagation - quorums(c) : Prop_1 \subseteq pxs.acc$
99: **Effect:**
100: $pxs.phase \leftarrow \text{idle}$
101: $cmap(k) \leftarrow ballot.conf$
102: $pxs.acc \leftarrow \emptyset$

103: **Internal** propagate$(k)_i$
104: **Precondition:**
105: $\neg failed \wedge status = \text{active}$
106: let $k = ballot.conf\text{-}index$
107: $cmap(k) \in C$
108: **Effect:**
109: $pxs.phase \leftarrow \text{propagate}$
110: $pnum1 \leftarrow pnum1 + 1$
111: $pxs.pnum \leftarrow pnum1$
112: $pxs.acc \leftarrow \emptyset$

113: **Internal** propagate-done$(k)_i$
114: **Precondition:**
115: $\neg failed \wedge status = \text{active}$
116: $pxs\text{-}phase = \text{propagate}$
117: let $k = ballot.conf\text{-}index$
118: let $c = cmap(k-1)$
119: let $c' = cmap(k)$
120: $c \in C$
121: $c' \in C$
122: $\exists Cons_2 \in propagation - quorums(c) : Cons_2 \subseteq pxs.acc$
123: $\exists Prop_2 \in consultation - quorums(c) : Prop_2 \subseteq pxs.acc$
124: **Effect:**
125: $cmap(k-1) \leftarrow \pm$
126: $pxs.phase \leftarrow \text{idle}$
127: $pxs.acc \leftarrow \emptyset$

Appendix C

IOA Specification of a Scalable DSM

Algorithm 13 *LoadBalancer*$_i$ – Signature and state

Signature:
  Input:
    read-write-ack$(v,id)_i$, $i,id \in I, type \in \{$read, write$\}, v \in V$
    rcv$(rqst)_{j,i}$, $i,j \in I$, *rqst* a request
    fail$_i$, $i \in I$
    share-load-rcv$(b)_{j,i}$, $i \in I$, b an array of requests
  Internal:
    load-balance$(rqst)_i$, $i \in I$, *rqst* a request
  Output:
    read-write$(type,v,id)_i$, $i,id \in I$, $type \in \{$read, write$\}, v \in V$
    snd$(rqst)_{i,j}$, $i,j \in I$, *rqst* a request
    shrink$_i$, $i \in I$
    expand$(j)_i$, $i,j \in I$
    share-load-snd$(b)_{i,j}$, $i \in I$, b an array of requests

State:
  rqst a record with fields
    sender $\in I$, the id of the requester
    type $\in \{$read, write$\}$
    target $\in \mathbb{R}^2$, the next requested coordinate
    next $\in \mathbb{R}^2$, the point of the next replica (on the path to the target).
    str-pt $\in \mathbb{R}^2$, the first requested coordinate
    val $\in V$, the value returned by the request
  failed a boolean
  expanding a boolean
  replica a boolean indicating whether it is a replica or not
  batch the set of requests received
  to-treat the set of requests that must be treated
  treating the set of requests being treated
  to-fwd the set of requests that must be forwarded
  to-rspd the set of requests to which respond

Derived Variables:
  overloaded $= (c \leq |\{r \in$ *to-treat* $\cup$ *treating* $\cup$ *batch*$\}|)$, where $c \in \mathbb{N}^{>0}$ is the capacity.

Initial States:
  rqst.sender initialized by the requester as its own identifier
  rqst.type initialized by the requester to read or write
  rqst.target $= \perp$
  rqst.next $= \perp$
  rqst.str-pt $= \perp$
  val $= v_0$ initialized as the value to write or to 0 (if the request refers to a read operation)
  failed $=$ false
  expanding $=$ false
  replica, true if the node maintains a value of the object, false otherwise
  batch $= \emptyset$
  to-treat $= \emptyset$
  treating $= \emptyset$
  to-fwd $= \emptyset$

Algorithm 14 $LoadBalancer_i$ – Transitions

Transitions:

Input $\mathsf{rcv}(rqst)_{j,i}$
- **Effect:**
  - **if** $\neg failed \wedge rqst.next \in zone$ **then**
    - **if** $rqst.str\text{-}pt \in zone$ **then**
      - $expanding \leftarrow \mathsf{true}$
      - $batch \leftarrow batch \cup \{rqst\}$
      - $last\text{-}request\text{-}time \leftarrow \infty$
    - **else if** $rqst.target = \perp$ **then**
      - $rqst.str\text{-}pt \leftarrow pt | pt \in zone$
      - $batch \leftarrow batch \cup \{rqst\}$
      - $last\text{-}request\text{-}time \leftarrow \infty$
    - **else if** $rqst.target \in zone$ **then**
      - $batch \leftarrow batch \cup \{rqst\}$
      - $last\text{-}request\text{-}time \leftarrow \infty$
    - **else**
      - $rqst.next = \mathsf{closest\text{-}pt}(rqst.target)$
      - $to\text{-}fwd \leftarrow to\text{-}fwd \cup \{rqst\}$

Output $\mathsf{read\text{-}write}(type, v, id)_i$
- **Precondition:**
  - $\neg failed$
  - $\neg expanding$
  - $rqst \in to\text{-}treat$
  - $type = rqst.type$
  - $v = rqst.val$
  - $id \leftarrow rqst.sender$
- **Effect:**
  - $treating \leftarrow treating \cup \{rqst\}$
  - $to\text{-}treat \leftarrow to\text{-}treat \setminus \{rqst\}$

Output $\mathsf{snd}(rqst)_{i,j}$
- **Precondition:**
  - $\neg failed$
  - $\neg expanding$
  - $(rqst \in fwd$
    - $\wedge rqst.next = \mathsf{closest\text{-}pt}(rqst.target))$
    - $\wedge j = \mathsf{nbr}(rqst.next) \vee (rqst \in to\text{-}rspd$
    - $\wedge j = rqst.sender)$
- **Effect:** none

Input fail_i
- **Effect:**
  - $failed \leftarrow \mathsf{true}$

$\mathsf{time\text{-}passage}(t)$
- **Precondition:**
  - **if** $\neg failed$ **then**
    - $now + t \leq last\text{-}request\text{-}time$
- **Effect:**
  - $now \leftarrow now + t$

Internal $\mathsf{load\text{-}balance}(rqst)_i$
- **Precondition:**
  - $\neg failed$
  - $\neg expanding$
  - $rqst \in batch$
- **Effect:**
  - **if** $overloaded$ **then**
    - $rqst.target \leftarrow \mathsf{next\text{-}pt\text{-}on\text{-}diag}(rqst.str\text{-}pt)$
    - $to\text{-}fwd \leftarrow to\text{-}fwd \cup \{rqst\}$
  - **else**
    - $to\text{-}treat \leftarrow to\text{-}treat \cup \{rqst\}$
  - $batch \leftarrow batch \setminus \{rqst\}$
  - **if** $batch = \emptyset$ **then**
    - $last\text{-}request\text{-}time \leftarrow now$
      $+ unloaded\text{-}period$

Input $\mathsf{read\text{-}write\text{-}ack}(v, id)_i$
- **Effect:**
  - **if** $\neg failed$ **then**
    - **if** $rqst \in treating \wedge rqst.id = id$ **then**
      - $rqst.val \leftarrow v$
      - $treating \leftarrow treating \setminus \{rqst\}$
      - $to\text{-}rspd \leftarrow to\text{-}rspd \cup \{rqst\}$

Output $\mathsf{expand}(j)_i$
- **Precondition:**
  - $\neg failed \wedge expanding$
  - $j \leftarrow any\text{-}active\text{-}node$
- **Effect:**
  - $replicating \leftarrow replicating \cup \{j\}$

Output $\mathsf{share\text{-}load\text{-}snd}(b)_{i,j}$
- **Precondition:**
  - $\neg failed \wedge expanding$
  - $j \in replicating$
  - $b \leftarrow \mathsf{second\text{-}half}(batch)$
- **Effect:**
  - $expanding \leftarrow \mathsf{false}$
  - $batch \leftarrow \mathsf{first\text{-}half}(batch)$
  - $replicating \leftarrow replicating \setminus \{j\}$

Input $\mathsf{share\text{-}load\text{-}rcv}(b)_{j,i}$
- **Effect:**
  - **if** $\neg failed \wedge status = \mathsf{idle}$ **then**
    - $batch \leftarrow b$
    - $last\text{-}request\text{-}time \leftarrow \infty$

Output shrink_i
- **Precondition:**
  - $last\text{-}request\text{-}time \leq now$
  - $\neg failed$
- **Effect:** none

Algorithm 15 *Traversal*$_i$ – Signature and state

Signature:
  Input:
    read-write$(type, v, id)_i$, $i, id \in I$, $type \in \{$read, write$\}, v \in V$
    rcv$(msg)_{j,i}$, $i, j \in I$, $msg \in M$
    fail$_i$, $i \in I$
    expand$(j)_i$, $i, j \in I$
    shrink$_i$, $i \in I$
    failure-detect$(j)_i$, $i \in I$
    notify-rcv$(t, v, z, n, gn)_{j,i}$, $i, j \in I$, $t \in T$, $v \in V$, $n \in I^*$, $gn \in \mathbb{N}$
    takeover-rsp$(j, k)_i$, $i, j, k \in I$
    replicate-rcv$_{i,i}$, $i, j \in I$
  Output:
    snd$(msg)_{i,j}$, $i, j \in I$, $msg \in M$
    read-write-ack$(v, id)_i$, $i, id \in I$, $v \in V$
    is-failed$(j)_i$, $i, j \in I$
    notify-snd$(t, v, z, n, gn)_{i,j}$, $i, j \in I$, $t \in T$, $v \in V$, $n \in I^*$, $gn \in \mathbb{N}$
    takeover-qry$(j)_i$, $i, j \in I$
    replicate-snd$_{i,j}$, $i, j \in I$
  Internal:
    cons-upd-init$(op)_i$, $i \in I$, $op \in \Pi$
    prop-init$(op)_i$, $i \in I$, $op \in \Pi$
    cons-upd-end$(op)_i$, $i \in I$, $op \in \Pi$
    prop-end$(op)_i$, $i \in I$, $op \in \Pi$

State:
  op an record with fields
    $id \in \mathbb{N} \times I$, the operation id
    $intr \in I$, the initiator replica of *op*
    $type \in \{$read, write$\}$
    $phase \in \{$idle, cons, update, prop, end$\}$
    tag, a record with fields
      $ct \in \mathbb{N}$, a counter
      $id \in I \cup \{\perp\}$
    $val \in V$
  msg, a record with fields
    $op \in \Pi$, the operation *msg* is part of
    $sense \in \{$north, south, east$\}$, the message sense
    $intvl \in \{$east, south, north$\} \mapsto \mathbb{R} \times \mathbb{R}$, given a sense, the interval of abscissas or ordinates the message covers
  tag, a record with fields
    $ct \in \mathbb{N}$
    $id \in I$
  $val \in V$, initially v_0
  failed, a boolean
  propagated, a boolean
  // The state for the adjustment follows
  $leaving \subset I$
  $changed \subset I$
  $rcvd\text{-}from \subset I$
  $nbrs \subset I$
  $detect\text{-}time \in \mathbb{R}^{>0}$
  $detect\text{-}period \in \mathbb{R}^{>0}$, a constant
  $notif\text{-}time \in \mathbb{R}^{>0}$
  $notif\text{-}period \in \mathbb{R}^{>0}$, a constant
  $zone \in \mathbb{R}^4$, a zone
  nbrs, a set of replica ids
  $gnum \in \mathbb{N}$
  replica a record with fields
    id, the replica id
    zone, the replica zone

Initial state:
  $op.\langle id, intr, type, phase\rangle = \langle \perp, \perp, \perp, \perp\rangle$
  $op.tag.ct = 0$ and $op.tag.id = \perp$
  $op.val = v_0$, the default value of the object.
  $failed =$ false
  $propagated =$ true
  $leaving, changed, rcvd\text{-}from, nbrs, zones = \emptyset$
  clock, the clock value at the beginning
  $notif\text{-}time, notif\text{-}period = 0$
  $gnum = 0$

Algorithm 16 $Traversal_i$ – Operation transitions

```
73: Operation Transitions:
74:   Input read-write(type, v, id)_i
75:     Effect:
76:       if ¬failed ∧ status ≠ idle then
77:         op.type ← type
78:         if type = read then
79:           op.phase ← cons
80:           op.⟨tag, val⟩ ← ⟨tag, val⟩
81:         else if type = write then
82:           op.phase ← upd
83:           op.⟨tag, val⟩ ← ⟨⊥, v⟩
84:         op.intr ← i
85:         ops ← ops ∪ {op}

86:   Internal cons-upd-init(op)_i
87:     Precondition:
88:       ¬failed ∧ status ≠ idle
89:       op ∈ ops
90:       op.phase ∈ {cons, upd}
91:     Effect:
92:       msg.op ← op
93:       msg.sense ← east
94:       msg.trajectory ← (ymax − ymin)/2
95:       to-send ← to-send ∪ {msg}

96:   Internal prop-init(op)_i
97:     Precondition:
98:       ¬failed ∧ status ≠ idle
99:       op ∈ ops
100:      op.phase = prop
101:    Effect:
102:      msg1.op ← msg2.op ← op
103:      msg1.sense ← south
104:      msg2.sense ← north
105:      msg.trajectory ← (xmax − xmin)/2
106:      mrcv[op.id] ← 0
107:      to-send ← to-send ∪ {msg1, msg2}

108:  Output read-write-ack(v, id)_i
109:    Precondition:
110:      ¬failed ∧ status ≠ idle
111:      op ∈ ops
112:      op.phase = end
113:      v = op.val
114:    Effect:
115:      op.phase ← idle

116: Internal cons-upd-end(op)_i
117:   Precondition:
118:     ¬failed ∧ status ≠ idle
119:     msg ∈ rcvd
120:     op = msg.op
121:     op.phase ∈ {cons, upd}
122:   Effect:
123:     op.⟨tag, val⟩ ← update(op.⟨tag, val⟩, ⟨tag, val⟩)
124:     rcvd ← rcvd \ {msg}
125:     if zone ⊂ msg.intvl[east] then
126:       // i has already participated
127:       if propagated ∧ op.type = read then
128:         op.phase = end
129:       else
130:         op.phase = prop
131:         if op.type = write then
132:           increments(op.tag)
133:     else
134:       msg.op ← op
135:       msg.sense ← east
136:       to-send ← to-send ∪ {msg}
137:     msg.intvl[east] ← msg.intvl[east] ∪ zone

138: Internal prop-end(op)_i
139:   Precondition:
140:     ¬failed ∧ status ≠ idle
141:     msg ∈ rcvd
142:     op = msg.op
143:     op.phase = prop
144:   Effect:
145:     ⟨tag, val⟩ ← op.⟨tag, val⟩
146:     if (zone ⊂ msg.intvl[north]
147:     ∧zone ⊂ msg.intvl[south]) then
148:       // i has already participated twice
149:       op.phase ← end
150:     else
151:       msg.intvl[msg.sense] ← msg.intvl[msg.sense]
152:         ∪zone
153:       if (zone ⊂ msg.intvl[north]
154:      ∧zone ⊂ msg.intvl[south]) then
155:         // i participates for the second time
156:         propagated ← true
157:       msg.op ← op
158:       to-send ← to-send ∪ {msg}
159:     rcvd ← rcvd \ {msg}
```

Algorithm 17 *Traversal*$_i$ – Communication transitions

160: **Output** $\mathsf{snd}(msg)_{i,j}$
161: **Precondition:**
162: $\neg failed \wedge status = \mathsf{participating}$
163: $msg \in to\text{-}send$
164: $msg.next \leftarrow \mathsf{next\text{-}pt\text{-}on\text{-}line}(i, msg.sense,$
165: $msg.trajectory)$
166: $j = \mathsf{nbr}(msg.next)$
167: **Effect:** none

168: **Input** $\mathsf{rcv}(msg)_{j,i}$
169: **Effect:**
170: **if** $\neg failed \wedge status \neq \mathsf{idle} \wedge msg.next \in zone$ **then**
171: $rcvd \leftarrow rcvd \cup \{msg\}$
172: $ops \leftarrow ops \cup \{msg.op\}$

173: **Input** fail_i
174: **Effect:**
175: $failed \leftarrow \mathsf{true}$

Algorithm 18 *Traversal*$_i$ – Adjustment transitions

Adjustment Transitions:

Input expand$(j)_i$
  Effect:
    if $\neg failed \wedge status \neq$ idle **then**
      $z = zone$ // we choose a zone to split
      $j.zone \leftarrow$ second-half(z)
      update$(j.nbrs, \langle j, j.zone, nbrs, 0\rangle)$
      $zone \leftarrow$ first-half(z)
      update$(nbrs, \langle i, zone, nbrs, 0\rangle)$
      $changed \leftarrow changed \cup \{j\}$
      $rcvd\text{-}from \leftarrow \emptyset$
      $gnum \leftarrow gnum + 1$
      $status \leftarrow$ expanding

Input shrink$_i$
  Effect:
    if $\neg failed \wedge status \neq$ idle **then**
      $leaving \leftarrow leaving \cup \{i\}$

Input failure-detect$(j)_i$
  Effect:
    if $\neg failed \wedge status \neq$ idle **then**
      $leaving \leftarrow leaving \cup \{j\}$

Input notify-rcv$(t, v, z, n, gn)_{j,i}$
  Effect:
    if $\neg failed$ **then**
      update$(\langle tag, val\rangle, \langle t, v\rangle)$
      update$(nrbs, \langle j, z, n, gn\rangle)$
      if abut$(\bigcup_{\forall k \in rcvd\text{-}from} k.zones, zones)$ **then**
        // i heard from all its north and
        // south neighbors
        $status \leftarrow$ participating
      if $gn \geq gnum$ **then**
        $gnum \leftarrow gn$
        $rcvd\text{-}from \leftarrow rcvd\text{-}from \cup \{j\}$
        $notif\text{-}time \leftarrow now + notif\text{-}period$

Output notify-snd$(t, v, z, n, gn)_{i,j}$
  Precondition:
    $\neg failed \wedge status \neq$ idle
    $notif\text{-}time \leq now$
    $t = tag$
    $v = val$
    $gn = gnum$
    $j \in nbrs$
    $z = j.zone$
    $z = j.nbrs$
  Effect: none

time-passage(t)
  Precondition:
    if $\neg failed$ **then**
      $now + t \leq notif\text{-}time$
      $now + t \leq detect\text{-}time$
  Effect:
    $now \leftarrow now + t$

Output takeover-qry$(j)_i$
  Precondition:
    $\neg failed \wedge status \neq$ idle
    // either i is in charge of looking for
    // the takeover or it is shrinking
    $j \in leaving \wedge i = \min\{k \in \text{nbrs}(j)\}) \vee i = j$
  Effect: none

Input takeover-rsp$(j, k)_i$
  Effect:
    if $\neg failed \wedge status \neq$ idle **then**
      $k.zone \leftarrow j.zone$
      $k.nbrs \leftarrow j.nbrs$
      if $j = i$ **then**
        // the leaving replica is the current one
        $status \leftarrow$ idle
      $changed \leftarrow changed \cup \{k\}$
      $leaving \leftarrow leaving \setminus \{j\}$

Output replicate-snd$(t, v, z, n)_{i,j}$
  Precondition:
    $\neg failed \wedge status \neq$ idle
    $j \in changed$
    $z = j.zone$
    $n = j.nbrs$
    $\langle t, v\rangle = \langle tag, val\rangle$
  Effect: none

Input replicate-rcv$(t, v, z, n)_{j,i}$
  Effect:
    if $\neg failed$ **then**
      $zone \leftarrow z$
      $tag \leftarrow t$
      $val \leftarrow v$
      update$(nrbs, \langle j, z, n, 0\rangle)$
      $rcvd\text{-}from \leftarrow \emptyset$
      $gnum \leftarrow gnum + 1$

Output is-failed$(j)_i$
  Precondition:
    $\neg failed \wedge status \neq$ idle
    $detect\text{-}time \leq now$
    $j \in nbrs$
  Effect:
    $detect\text{-}time \leftarrow now + detect\text{-}period$

Bibliography

[1996] Special issue on group communication services. *Communications of the ACM*, 39(4), 1996.

[ABND95] Hagit Attiya, Amotz Bar-Noy, and Danny Dolev. Sharing memory robustly in message-passing systems. *J. ACM*, 42(1):124–142, 1995.

[ACLS94] Divyakant Agrawal, Manhoi Choy, Hong Va Leong, and Ambuj Singh. Mixed consistency: a model for parallel programming. In *Proceedings 13th ACM Symposium on Principles of Dist. Computing*, pages 101–110, 1994.

[ADGR05] Emmanuelle Anceaume, Xavier Defago, Maria Gradinariu, and Matthieu Roy. Towards a theory of self-organization. In *Proceedings of 9th Int'l Conference on Principles of Distributed Systems*, 2005.

[AE89] Divyakant Agrawal and Amr El Abbadi. Efficient solution to the distributed mutual exclusion problem. In *PODC '89: Proceedings of the eighth annual ACM Symposium on Principles of distributed computing*, pages 193–200, New York, NY, USA, 1989. ACM Press.

[AE90] Divyakant Agrawal and Amr El Abbadi. The tree quorum protocol: an efficient approach for managing replicated data. In *Proceedings of the 16th international conference on Very large databases*, pages 243–254, San Francisco, CA, USA, 1990. Morgan Kaufmann Publishers Inc.

[AF92] Hagit Attiya and Roy Friedman. A correctness condition for high-performance multiprocessors (extended abstract). In *STOC '92: Proceedings of the twenty-fourth annual ACM symposium on Theory of computing*, pages 679–690, New York, NY, USA, 1992. ACM Press.

[AF98] Hagit Attiya and Roy Friedman. A correctness condition for high-performance multiprocessors. *SIAM Journal on Computing*, 27(6):1637–1670, 1998.

[AGGV05] Emmanuelle Anceaume, Maria Gradinariu, Vincent Gramoli, and Antonino Virgillito. P2P architecture for self* atomic memory. In *Proceedings of the 8th International Symposium on Parallel Architectures, Algorithms and Networks*, pages 214–219. ISCA, Dec. 2005.

[AGMT95] Yehuda Afek, David S. Greenberg, Michael Merritt, and Gadi Taubenfeld. Computing with faulty shared objects. *J. ACM*, 42(6):1231–1274, 1995.

[AM05] Ittai Abraham and Dahlia Malkhi. Probabilistic quorums for dynamic systems. *Distributed Computing*, 18(2):113–124, 2005.

[And04] David P. Anderson. Boinc: a system for public-resource computing and storage. In *Proceedings of the 5th IEEE/ACM International Workshop on Grid Computing*, pages 4–10, 2004.

[AV86] Baruch Awerbuch and Paul Vitanyi. Atomic shared register access by asynchronous hardware. In *Proceedings of 27th IEEE Symposium on Foundations of Computer Science*, pages 233–243, 1986.

[AW98] Hagit Attiya and Jennifer Welch. *Distributed Computing. Fundamentals, Simulations, and Advanced Topics*. McGraw-Hill, 1998.

[BBC+04] Andy Bavier, Mic Bowman, Brent Chun, David Culler, Scott Karlin, Steve Muir, Larry Peterson, Timothy Roscoe, Tammo Spalink, and Mike Wawrzoniak. Operating system support for planetary-scale network services. In *Symposium on Networked Systems Design and Implementation*, pages 253–266, 2004.

[Bel99] Fatima Belkouch. *Quorums auto-stabilisants: Applications dans les systèmes répartis*. PhD thesis, Université Technologique de Compiègne, Sept. 1999.

[BFP+72] Manuel Blum, Robert Floyd, Vaughan Pratt, Ronald Rivest, and Robert Tarjan. Time bounds for selection. *J. Computer and System Sciences*, 7:448–461, 1972.

[BG83] Philip A. Bernstein and Nathan Goodman. The failure and recovery problem for replicated databases. In *Proceedings of the second annual ACM symposium on Principles of distributed computing (PODC'83)*, pages 114–122, New York, NY, USA, 1983. ACM Press.

[BGMR01] Fransisco Brasilero, Fabíola Greve, Achour Mostefaoui, and Michel Raynal. Consensus in one communication step. In *Proceedings of the 6th International Conference on Parallel and Computing Technologies*, pages 42–50, 2001.

[BHK+91] Mary G. Baker, John H. Hartman, Michael D. Kupfer, Ken W. Shirriff, and John K. Ousterhout. Measurements of a distributed file system. In *SOSP '91: Proceedings of the 13th ACM symposium on Operating systems principles*, pages 198–212, New York, NY, USA, 1991. ACM Press.

[BJ87] Ken Birman and Thomas Joseph. Exploiting virtual synchrony in distributed systems. In *Proceedings of the 11th ACM Symposium on Operating systems principles*, pages 123–138. ACM Press, 1987.

[BPS05] Jean-Michel Busca, Fabio Picconi, and Pierre Sens. Pastis: a highly-scalable multi-user peer-to-peer file system. In *EuroPar*, LNCS, September 2005.

[BSV03] Ranjita Bhagwan, Stefan Savage, and Geoffrey Voelker. Understanding availability. In *Proceedings of the 2nd International Workshop on Peer-to-Peer Systems*, pages 256–267, February 2003.

[CDHP+05] Paz Carmi, Shlomi Dolev, Sariel Har-Peled, Matthew Katz, and Michael Segal. Geographic quorum system approximations. *Algorithmica*, 41(4):233–244, 2005.

[CGG+05] Gregory Chockler, Seth Gilbert, Vincent Gramoli, Peter M. Musial, and Alex A. Shvartsman. Reconfigurable distributed storage for dynamic networks. In *Proceedings of 9th International Conference on Principles of Distributed Systems*, pages 214–219, December 2005.

[CGJ+07] Paulo Costa, Vincent Gramoli, Márk Jelasity, Gian Paulo Jesi, Erwan Le Merrer, Alberto Montresor, and Leonardo Querzoni. Exploring the interdisciplinary connections of gossip-based systems. *ACM SIGOPS Operating System Review*, 41(4), 2007. to appear.

[CHT96] Tushar Deepak Chandra, Vassos Hadzilacos, and Sam Toueg. The weakest failure detector for solving consensus. *J. ACM*, 43(4):685–722, 1996.

[CKKM06] Byung-Gon Chun, M. Frans Kaashoek, John Kubiatowicz, and Robert Morris. Efficient replica maintenance for distributed storage systems. In *Proceedings of the 3rd Symposium on Networked Systems Design*, 2006.

[CLMT05] Gregory Chockler, Nancy A. Lynch, Sayan Mitra, and Joshua Tauber. Proving atomicity: An assertional approach. In *Proceedings of the 19th International Symposium on DIStributed Computing (DISC)*, volume 3724, pages 152–168. Springer-Verlag GmbH in Lecture Notes in Computer Science, Sept. 2005.

[Coh03] Bram Cohen. Incentives build robustness in bittorrent. In *Proceedings of Workshop on Economics of Peer-to-Peer Systems*, June 2003.

[CS98] Oleg M. Cheiner and Alex A. Shvartsman. Implementing and evaluating an eventually-serializable data service. In *PODC '98: Proceedings of the 17th annual ACM symposium on Principles of distributed computing*, page 317, New York, NY, USA, 1998. ACM Press.

[DA06] Anwitaman Datta and Karl Aberer. Internet-scale storage systems under churn – a study of the steady-state using markov models. In *P2P'06: Proceedings of the 6th IEEE International Conference on Peer-to-Peer Computing*, pages 133–144, 2006.

[dbl] Digital bibliography & library project.

[DGFGK05] Carole Delporte-Gallet, Hugues Fauconnier, Rachid Guerraoui, and Petr Kouznetsov. Mutual exclusion in asynchronous systems with failure detectors. *J. Parallel Distrib. Comput.*, 65(4):492–505, 2005.

[DGH+87] Alan Demers, Dan Greene, Carl Hauser, Wes Irish, John Larson, Scott Shenker, Howard Sturgis, Dan Swinehart, and Doug Terry. Epidemic algorithms for replicated database maintenance. In *Proceedings of the sixth annual ACM Symposium on Principles of distributed computing*, pages 1–12, New York, NY, USA, 1987. ACM Press.

[DGL+05] Shlomi Dolev, Seth Gilbert, Nancy A. Lynch, Alex A. Shvartsman, and Jennifer Welch. Geoquorums: Implementing atomic memory in ad hoc networks. *Distributed Computing*, 18(2):125–155, 2005.

[DGLC04] Partha Dutta, Rachid Guerraoui, Ron R. Levy, and Arindam Chakraborty. How fast can a distributed atomic read be? In *PODC '04: Proceedings of the twenty-third annual ACM symposium on Principles of distributed computing*, pages 236–245, New York, NY, USA, 2004. ACM Press.

[DNS91] David J. DeWitt, Jeffrey F. Naughton, and Donovan A. Schneider. Parallel sorting on a shared-nothing architecture using probabilistic splitting. In *Proceedings of the First International Conference on Parallel and Distributed Information Systems*, pages 280–291, 1991.

[DS90] Michel Dubois and Christoph Scheurich. Memory access dependencies in shared-memory multiprocessors. *IEEE Trans. Softw. Eng.*, 16(6):660–673, 1990.

[DSB86] Michel Dubois, Christoph Scheurich, and Fayé Briggs. Memory access buffering in multiprocessors. In *ISCA '86: Proceedings of the 13th annual international symposium on Computer architecture*, pages 434–442, Los Alamitos, CA, USA, 1986. IEEE Computer Society Press.

[DSB88] Michel Dubois, Christoph Scheurich, and Fayé A. Briggs. Synchronization, coherence, and event ordering in multiprocessors. *Computer*, 21(2):9–21, 1988.

[ES00] Burkhard Englert and Alex A. Shvartsman. Graceful quorum reconfiguration in a robust emulation of shared memory. In *Proceedings of International Conference on Distributed Computer Systems*, pages 454–463, 2000.

[FGJ+07] Antonio Fernández, Vincent Gramoli, Ernesto Jiménez, Anne-Marie Kermarrec, and Michel Raynal. Distributed slicing in dynamic systems. In *Proceedings of the 27th International Conference on Distributed Computing Systems (ICDCS'07)*. IEEE Computer Society Press, Jun 2007.

[FLP85] Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. Impossibility of distributed consensus with one faulty process. *J. ACM*, 32(2):374–382, 1985.

[FR75] Robert W. Floyd and Ronald L. Rivest. Expected time bounds for selection. *Commun. ACM*, 18(3):165–172, 1975.

[Fri95] Roy Friedman. Using virtual synchrony to develop efficient fault tolerant distributed shared memories. Technical Report TR95-1506, Cornell University, 1995.

[FRT05] Roy Friedman, Michel Raynal, and Corentin Travers. Two abstractions for implementing atomic objects in dynamic systems. In *Proceedings of 9th International Conference on Principles of Distributed Systems*, pages 73–87, 2005.

[FT99] Ted Friedman and Don Towsley. Multicast session membership size estimation. In *Proceedings of the 12th Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM'93)*, 1999.

[GAV07] Vincent Gramoli, Emmanuelle Anceaume, and Antonino Virgillito. Square: Scalable quorum-based atomic memory with local reconfiguration. In *Proceedings of the 22nd ACM Symposium on Applied Computing (SAC'07)*, pages 574–579. ACM Press, mar 2007.

[Gif79] David K. Gifford. Weighted voting for replicated data. In *Proceedings of the seventh ACM symposium on Operating systems principles*, pages 150–162. ACM Press, 1979.

[GKL07] Vincent Gramoli, Anne-Marie Kermarrec, and Erwan Le Merrer. GossiPeer: A development framework for gossip-based protocols, 2007. http://gossipeer.gforge.inria.fr/.

[GKM03] Ayalvadi J. Ganesh, Anne-Marie Kermarrec, and Laurent Massoulié. Peer-to-peer membership management for gossip-based protocols. *IEEE Trans. Comput.*, 52(2):139–149, 2003.

[GKM+06] Vincent Gramoli, Anne-Marie Kermarrec, Achour Mostefaoui, Michel Raynal, and Bruno Sericola. Core persistence in peer-to-peer systems: Relating size to lifetime. In *Proceedings of the On-The-Move International Workshop on Reliability in Decentralized Distributed Systems*, volume 4278 of *LNCS*, pages 1470–1479. Springer, Oct. 2006.

[GLS03] Seth Gilbert, Nancy A. Lynch, and Alex A. Shvartsman. RAMBO II: Rapidly reconfigurable atomic memory for dynamic networks. In *Proceedings of International Conference on Dependable Systems and Networks*, pages 259–268, 2003.

[GMB85] Hector Garcia-Molina and Daniel Barbara. How to assign votes in a distributed system. *J. ACM*, 32(4):841–860, 1985.

[GMS04] Chryssis Georgiou, Peter M. Musial, and Alex A. Shvartsman. Long-lived RAMBO: Trading knowledge for communication. In *Proceedings of 11th Colloquium on Structural Information and Communication Complexity*, pages 185–196. Springer, 2004.

[GMS05] Vincent Gramoli, Peter M. Musial, and Alex A. Shvartsman. Operation liveness and gossip management in a dynamic distributed atomic data service. In *Proceedings of the ISCA 18th International Conference on Parallel and Distributed Computing Systems (PDCS'05)*, pages 206–211, September 2005.

[GNS06] Chryssis Georgiou, Nicolas C. Nicolaou, and Alex A. Shvartsman. Fault-tolerant semifast implementations of atomic read/write registers. In *18th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, 2006.

[gnua] Gnutella homepage. http://www.gnutella.com.

[gnub] The gnutella protocol development homepage. http://www.the-gdf.org.

[Goo89] James R. Goodman. Cache consistency and sequential consistency. Technical Report 61, SCI Committee, March 1989.

[HA90] Phillip W. Hutto and Mustaque Ahamad. Slow memory: Weakening consistency to enhance concurrency in distributed shared memories. In *Proceedings of the 10th Int'l Conf. on Distributed Computing Systems (ICDCS)*, pages 302–311, May 1990.

[Her86] Maurice Herlihy. A quorum-consensus replication method for abstract data types. *ACM Trans. Comput. Syst.*, 4(1):32–53, 1986.

[Her87] Maurice Herlihy. Dynamic quorum adjustment for partitioned data. *ACM Trans. Database Syst.*, 12(2):170–194, 1987.

[HMP95] Ron Holzman, Yosi Marcus, and David Peleg. Load balancing in quorum systems (extended abstract). In *WADS '95: Proceedings of the 4th International Workshop on Algorithms and Data Structures*, pages 38–49, London, UK, 1995. Springer-Verlag.

[HW90] Maurice Herlihy and Jeannette M. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Trans. on Programming Languages and Systems (TOPLAS)*, 12(3):463–492, 1990.

[IRV89] Balakrishna Iyer, Gary Ricard, and Peter Varman. Percentile finding algorithm for multiple sorted runs. In *Proceedings of the 15th International Conference on Very Large Data Bases*, pages 135–144, August 1989.

[Isa95] Richard Isaac. *The pleasure of probabilities*. Springer Verlag, Reading in Mathematics, 1995.

[Iwa05] Konrad Iwanicki. Gossip-based dissemination of time. Master's thesis, Warsaw University - Vrije Universiteit Amsterdam, 2005.

[JGKvS04] Márk Jelasity, Rachid Guerraoui, Anne-Marie Kermarrec, and Maarten van Steen. The peer sampling service: experimental evaluation of unstructured gossip-based implementations. In *Proceedings of the 5th ACM/IFIP/USENIX Int'l Conference on Middleware*, pages 79–98, 2004.

[JK06] Márk Jelasity and Anne-Marie Kermarrec. Ordered slicing of very large-scale overlay networks. In *Proceedings of the 6th IEEE International Conference on Peer-to-Peer Computing*, pages 117–124, 2006.

[JM04] Márk Jelasity and Alberto Montresor. Epidemic-style proactive aggregation in large overlay networks. In *Proceedings of the 24th International Conference on Distributed Computing Systems (ICDCS'04)*, pages 102–109, 2004.

[JMB04] Márk Jelasity, Alberto Montresor, and Özalp Babaoglu. A modular paradigm for building self-organizing peer-to-peer applications. In *Engineering Self-Organising Systems: Nature-Inspired Approaches to Software Engineering*, number 2977 in Lecture Notes in Artificial Intelligence, pages 265–282. Springer-Verlag, April 2004.

[JMB05] Márk Jelasity, Alberto Montresor, and Özalp Babaoglu. Gossip-based aggregation in large dynamic networks. *ACM Transactions on Computer Systems*, 23(3):219–252, 2005.

[kaz] Kazaa homepage. http://www.kazaa.com.

[KDG03] David Kempe, Alin Dobra, and Johannes Gehrke. Gossip-based computation of aggregrate information. In *Proceedings of 44th Annual IEEE Symposium of Foundations of Computer Science*, pages 482–491, 2003.

[KH97] Yu-Chen Kuo and Shing-Tsaan Huang. A geometric approach for constructing coteries and *k*-coteries. *IEEE Transactions on Parallel and Distributed Systems*, 8:402–411, 1997.

[KPG+05] Dionysios Kostoulas, Dimitrios Psaltoulis, Indranil Gupta, Ken Birman, and Alan Demers. Decentralized schemes for size estimation in large and dynamic groups. In *Proceedings of 4th IEEE International Symposium on Network Computing and Applications (NCA'05)*, pages 41–48, July 2005.

[KR02] Idit Keidar and Sergio Rajsbaum. On the cost of fault-tolerant consensus when there are no faults, 2002. PODC 2002 Tutorial Slides. http://www.ee.technion.ac.il/~idish/ftp/podc02-tutorial-v2.ppt.

[Lam78] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communication of the ACM*, 21(7):558–565, July 1978.

[Lam86] Leslie Lamport. On interprocess communication. *Distributed Computing*, 1(2):77–101, 1986.

[Lam89] Leslie Lamport. The part-time parliament. Technical Report 49, Digital SRC, September 1989.

[Lam98] Leslie Lamport. The part-time parliament. *ACM Transactions on Computer Systems*, 16(2):133–169, 1998.

[Lam05] Leslie Lamport. Fast paxos. Technical Report MSR-TR-2005-112, Microsoft Research, July 2005.

[Lam06a] Leslie Lamport. Fast paxos. *Distributed Computing*, 19(2):79–103, Oct. 2006.

[Lam06b] Leslie Lamport. Lower bounds for asynchronous consensus. *Distributed Computing*, 19(2), Oct. 2006.

[LF82] Leslie Lamport and Michael Fischer. Byzantine generals and transaction commit protocols. unpublished, 1982.

[LKM06] Erwan Le Merrer, Anne-Marie Kermarrec, and Laurent Massoulié. Peer to peer size estimation in large and dynamic networks: A comparative study. In *15th International Symposium on High performance Distributed Computing (HPDC)*, Paris, France, 2006.

[LMR92] Neilsen Mitchell L., Masaaki Mizuno, and Michel Raynal. A general method to define quorums. In *Proceedings of the 12th International Conference on Distributed Computing Systems*, pages 657–664, June 1992.

[LS88] Richard J. Lipton and Jonathan S. Sandberg. PRAM: A scalable shared memory. Technical Report CS-TR-180-88, Dept. of Computer Science, Princeton University, September 1988.

[LS97] Nancy A. Lynch and Alex A. Shvartsman. Robust emulation of shared memory using dynamic quorum-acknowledged broadcasts. In *Proceedings of 27th Int-l Symp. on Fault-Tolerant Comp.*, pages 272–281, 1997.

[LS02] Nancy A. Lynch and Alex A. Shvartsman. RAMBO: A reconfigurable atomic memory service for dynamic networks. In *Proceedings of 16th International Symposium on Distributed Computing*, pages 173–190, 2002.

[LW05] Hyunyoung Lee and Jennifer L. Welch. Randomized registers and iterative algorithms. *Distributed Computing*, 17(3):209–221, 2005.

[LWV03] Hyunyoung Lee, Jennifer L. Welch, and Nitin H. Vaidya. Location tracking using quorum in mobile ad hoc networks. *Ad Hoc Networks*, 1(4):371–381, 2003.

[Lyn96] Nancy A. Lynch. *Distributed Algorithms*. Morgan Kaufmann Publishers, 1996.

[MA04] Jean-Philippe Martin and Lorenzo Alvisi. A framework for dynamic byzantine storage. In *Proceedings of the 2004 International Conference on Dependable Systems and Networks (DSN)*, page 325. IEEE Computer Society, 2004.

[Mae85] Mamoru Maekawa. A $\sqrt{N}$ algorithm for mutual exclusion in decentralized systems. *ACM Trans. Comput. Syst.*, 3(2):145–159, 1985.

[MKB05] Roie Melamed, Idit Keidar, and Yoav Barel. Octopus: A fault-tolerant and efficient ad-hoc routing protocol. In *Proceedings of the 24th IEEE Symposium on Reliable Distributed Systems*, 2005.

[MKKB01] Robert Morris, David Karger, Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *ACM SIGCOMM 2001*, San Diego, CA, September 2001.

[MLKG06] Laurent Massoulié, Erwan Le Merrer, Anne-Marie Kermarrec, and Ayalvadi Ganesh. Peer counting and sampling in overlay networks: random walk methods. In *Twenty-Fifth Annual ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing (PODC 2006)*, Denver (CO), 2006. to appear.

[MNR02] Dahlia Malkhi, Moni Naor, and David Ratajczak. Viceroy: a scalable and dynamic emulation of the butterfly. In *PODC*, pages 183–192, 2002.

[MR95] Rajeev Motwani and Prabhakar Raghavan. *Randomized Algorithms*. Cambridge University Press, Cambridge, England, June 1995.

[MR98] Dahlia Malkhi and Michael Reiter. Secure and scalable replication in phalanx. In *Proceedings of the 17th IEEE Symposium on Reliable Distributed Systems*, page 51, Washington, DC, USA, 1998. IEEE Computer Society.

[MR04] Dahlia Malkhi and Michael Reiter. Byzantine quorum systems. *Distributed Computing*, 11(4):203–213, 2004.

[MR07] Michael G. Merideth and Michael Reiter. Probabilistic opaque quorum systems. In *Proceedings of the 21st International Symposium on Distributed Computing (DISC'07)*, volume 4731 of *Lecture Notes in Computer Science*, pages 403–419, September 2007.

[MRW97] Dahlia Malkhi, Michael Reiter, and Rebecca Wright. Probabilistic quorum systems. In *PODC '97: Proceedings of the 16th annual ACM symposium on Principles of distributed computing*, pages 267–273. ACM Press, 1997.

[MRWW01] Dahlia Malkhi, Michael Reiter, Avishai Wool, and Rebecca Wright. Probabilistic quorum systems. *The Information and Computation Journal*, 170(2):184–206, November 2001.

[MS04] Peter M. Musial and Alex A. Shvartsman. Implementing a reconfigurable atomic memory service for dynamic networks. In *Proceedings of 18th International Parallel and Distributed Symposium — FTPDS WS*, page 208b, 2004.

[MT00] Michael Merritt and Gadi Taubenfeld. Computing with infinitely many processes. In *Proceedings of the 14th International Conference on Distributed Computing (DISC)*, pages 164–178, London, UK, 2000. Springer-Verlag.

[MTK06] Ken Miura, Taro Tagawa, and Hirotsugu Kakugawa. A quorum-based protocol for searching objects in peer-to-peer networks. *IEEE Transactions on Parallel and distributed Systems*, 17(1), January 2006.

[NN05] Uri Nadav and Moni Naor. The dynamic and-or quorum system. In Pierre Fraigniaud, editor, *Distributed algorithms*, volume 3724 of *Lecture Notes In Computer Science*, pages 472–486, September 2005.

[NP00] Blair Nonnecke and Jenny Preece. Lurker demographics: counting the silent. In *CHI '00: Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 73–80, New York, NY, USA, 2000. ACM Press.

[NW98] Moni Naor and Avishai Wool. The load, capacity, and availability of quorum systems. *SIAM Journal on Computing*, 27(2):423–447, 1998.

[NW03] Moni Naor and Udi Wieder. Scalable and dynamic quorum systems. In *Proceedings of the 22th annual symposium on Principles of distributed computing (PODC'03)*, pages 114–122. ACM Press, 2003.

[NW05] Moni Naor and Udi Wieder. Scalable and dynamic quorum systems. *Distributed Computing*, 17(4):311–322, 2005.

[PW02] David Peleg and Avishai Wool. How to be an efficient snoop, or the probe complexity of quorum systems. *SIAM Journal*, 15(3):416–433, 2002.

[QCCG05] Feng Qi, Run-Qing Cui, Chao-Ping Chen, and Bai-Ni Guo. Some completely monotonic functions involving polygamma functions and an application. *Journal of Mathematical Analysis and Applications*, 310(1):303–308, 2005.

[Ray86] Michel Raynal. *Algorithms for mutual exclusion*. MIT Press, Cambridge, MA, USA, 1986.

[RD01] Antony Rowstron and Peter Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In Rachid Guerraoui, editor, *Middleware 2001*, volume 2218 of *LNCS*, pages 329–350. Springer-Verlag, 2001.

[RFH+01] Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard M. Karp, and Scott Shenker. A scalable content-addressable network. In *Proceedings of the ACM SIGCOMM*, pages 161–172, 2001.

[SDCM06] Jan Sacha, Jim Dowling, Raymond Cunningham, and Rene Meier. Using aggregation for adaptive super-peer discovery on the gradient topology. In *IEEE International Workshop on Self-Managed Networks, Systems and Services*, pages 77–90, 2006.

[SGG02] Stefan Saroiu, P. Krishna Gummadi, and Steven Gribble. A measurement study of peer-to-peer file sharing systems. In *SPIE Multimedia Computing and Networking (MMCN2002)*, 2002.

[SR06] Daniel Stutzbach and Reza Rejaie. Understanding churn in peer-to-peer networks. In *IMC '06: Proceedings of the 6th ACM SIGCOMM on Internet measurement*, pages 189–202, New York, NY, USA, 2006. ACM Press.

[SZ98] Nir Shavit and Asaph Zemach. Combining funnels: a new twist on an old tale... In *PODC '98: Proceedings of the 17th annual ACM symposium on Principles of distributed computing*, pages 61–70, New York, NY, USA, 1998. ACM Press.

[Tho79] Robert H. Thomas. A majority consensus approach to concurrency control for multiple copy databases. *ACM Trans. Database Syst.*, 4(2):180–209, 1979.

[UD90] Aydin Üresin and Michel Dubois. Parallel asynchronous algorithms for discrete data. *J. ACM*, 37(3):588–606, 1990.

[Val84] Leslie G. Valiant. Short monotone formulae for the majority function. *J. Algorithms*, 5(3):363–366, 1984.

[VGvS05] Spyros Voulgaris, Daniela Gavidia, and Maarten van Steen. Cyclon: Inexpensive membership management for unstructured p2p overlays. *Journal of Network and Systems Management*, 13(2):197–217, 2005.

[Vid96] Krishnamurthy Vidyasankar. Weak atomicity: A helpful notion in the construction of atomic shared variables. *SADHANA: Journal of Engineering Sciences of the Indian Academy of Sciences 21*, pages 245–259, 1996.

[WLS+02] Brian White, Jay Lepreau, Leigh Stoller, Robert Ricci, Shashi Guruprasad, Mac Newbold, Mike Hibler, Chad Barb, and Abhijeet Joglekar. An integrated experimental environment for distributed systems and networks. In *OSDI02*, pages 255–270, Boston, MA, December 2002. USENIXASSOC.