THE COMPLEXITY OF RELIABLE DISTRIBUTED STORAGE

THÈSE N^{O} 3999 (2008)

PRÉSENTÉE LE 7 MARS 2008

À LA FACULTÉ INFORMATIQUE ET COMMUNICATIONS

Laboratoire de programmation distribuée

SECTION DES SYSTÈMES DE COMMUNICATION

ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

POUR L'OBTENTION DU GRADE DE DOCTEUR ÈS SCIENCES

PAR

Ron LEVY

ingénieur en systèmes de communication diplômé EPF
et de nationalité néerlandaise

acceptée sur proposition du jury:

Prof. R. Hersch, président du jury
Prof. R. Guerraoui, directeur de thèse
Prof. D. Kostic, rapporteur
Prof. F. Pedone, rapporteur
Prof. L. E. Rodrigues, rapporteur

ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Lausanne, EPFL
2008

Abstract

Distributed storage systems based on commodity hardware have gained in popularity because they are cheaper, can be made more reliable and offer better scalability than centralized storage systems. However, implementing and managing such systems is more complicated due to their distributed nature. This thesis presents efficient and reliable distributed storage that can be read and written by any number of clients. The focus is on atomic storage, which guarantees that from the clients' perspective, the distributed storage behaves exactly like a centralized one. Three key complexity metrics – time, number of logs and throughput – are considered. For each metric, precise performance bounds together with matching algorithms are provided. Experimental results are used to confirm the theoretical performance analysis wherever necessary.

(1) **Time-complexity** is an indication of the latency of read and write operations, i.e. the time between a client's invocation of an operation and the response of the storage. This thesis presents optimal *fast* atomic storage implementations, namely, implementations that complete both reads and writes in 1 round-trip between the client and the servers. Interestingly, the existence of fast implementations depends on the maximum number of clients that can read from the storage. More precisely, it is shown that a fast implementation is possible *if and only if* the number of readers is less than $n/f - 2$, with n servers out of which f can fail. Furthermore, it is shown that fast implementations are *impossible* for multiple writers if servers can fail.

(2) **Log-complexity** is an indication of the number of stable storage (hard disk) accesses needed in every *read* or *write* operation. Stable storage is used to log data in order to prevent data loss after a crash, in a context where servers can crash and recover. This thesis revises the notion of atomicity for this context, determines a lower bound on log-complexity and introduces an atomic storage matching this bound. The optimality of the storage is also established in terms of resilience, as well as time-complexity.

(3) **Throughput** measures the average number of client requests that can be completed per time unit. In order for a distributed storage to serve a high number of clients concurrently, high throughput is required. This thesis introduces an

atomic storage that provides optimal read throughput for homogeneous clusters of servers. The storage organizes servers around a ring and uses only point-to-point communication. It is resilient to the crash failure of any number of readers and writers as well as to the crash failure of all but one server. The storage was evaluated on a cluster of 24 nodes.

The same storage is modified to solve the more general uniform total order broadcast problem, which can be used to replicate any application reliably. Thus, the first uniform total order broadcast algorithm that is throughput optimal, regardless of message broadcast patterns, is introduced. The algorithm is based on a ring topology, only relies on point-to-point inter-server communication, and has a linear latency with respect to the number of processes. The implementation was benchmarked against two of the most widely used group communication packages and the results confirm that the algorithm is indeed throughput optimal.

Keywords: distributed storage, atomic storage, crash-recovery, total order, complexity, throughput.

Résumé

Les systèmes de stockage répartis utilisant du matériel grand public ont gagné en popularité parce qu'ils sont moins chers, peuvent être rendus plus fiables et offrent un meilleur passage à l'échelle que les systèmes centralisés utilisant du matériel spécialisé. En revanche, du fait de leur distribution, ces systèmes de stockages sont plus complexes à implémenter et à gérer que les systèmes centralisés. Dans cette thèse, nous proposons des algorithmes permettant de construire des systèmes de stockage efficaces et fiables. Ces systèmes peuvent être lus et écrits par un nombre illimité de clients. L'accent est mis sur les systèmes de stockage ayant une sémantique atomique. Ces derniers garantissent que, du point de vue des clients, le système de stockage se comporte exactement comme un système centralisé. Trois métriques de complexité clées sont étudiées: temps, nombre d'opérations de journalisation et débit. Pour chaque métrique, nous déterminons et prouvons des bornes précises sur les performances optimales qui peuvent être obtenues, et nous présentons des algorithmes conformes à ces bornes. Des résultats expérimentaux sont présentés pour confirmer les résultats théoriques lorsque c'est nécessaire.

(1) La **complexité en temps** (également appelée latence) mesure le temps moyen entre l'invocation d'une opération par un client (écriture ou lecture) et la réponse du système de stockage. Nous présentons des algorithmes de stockages optimaux en termes de latence. Plus précisément, ces algorithmes garantissent que les opérations de lecture et d'écriture ne requièrent qu'un aller-retour entre le client et les serveurs. Nous prouvons que la possibilité d'être optimal en latence dépend du nombre maximal de clients qui peuvent faire des opérations de lecture sur le système de stockage de faon concurrente. Plus précisement, il est montré qu'une implémentation optimale est possible *si et seulement si* le nombre de lecteurs est inférieur à $n/f - 2$, avec n représentant le nombre de serveurs et f le nombre de fautes tolérées. De plus, nous montrons qu'il est impossible de réaliser un algorithme optimal en latence tolérant à la fois de multiples écrivains et des pannes de serveurs.

(2) La **complexité en nombre d'opérations de journalisation** mesure le nombre moyen d'accès à un système de stockage persistant (disque dur) qu'il est

nécessaire d'effectuer pour chaque opération de lecture ou d'écriture. Le système de stockage persistant est utilisé pour journaliser des données afin d'éviter qu'elles ne soient pas perdues suite à une panne de serveur. L'utilisation d'un système de stockage persistant est nécessaire lorsque les serveurs tombant en panne peuvent redémarrer. Dans cette thèse, nous révisons la notion d'atomicité dans ce cadre, nous déterminons une borne inférieure sur la complexité en opérations de journalisation et nous proposons un algorithme permettant de réaliser un système de stockage conforme à cette borne inférieure. L'optimalité du système de stockage est aussi établi en termes de nombres de pannes et complexité en temps.

(3) Le **débit** mesure le nombre moyen de requêtes qui peuvent être effectuées par unité de temps. Un système de stockage doit garantir un débit élevé afin de permettre le traitement simultané d'un nombre élevé de requêtes. Nous proposons un algorithme permettant de construire un système de stockage atomique fournissant un débit optimal en lecture dans un environnement de type grappes de serveurs homogènes. L'algorithme proposé repose sur une organisation des serveurs en anneau, chaque serveur n'effectuant que des communications point-à-point vers son successeur dans l'anneau. Par ailleurs, l'algorithme proposé tolère la panne d'un nombre arbitraire de clients, et nécessite uniquement qu'un serveur fonctionne à tout moment. Nous avons réalisé une implémentation de cet algorithme et l'avons évalué sur une grappe de 24 noeuds.

Finalement, nous avons modifié l'algorithme précédent afin de résoudre le problème plus général de diffusion de messages avec ordre total uniforme. Un tel algorithme peut être utilisé pour répliquer n'importe quelle application de façon fiable. Nous introduisons le premier algorithme de diffusion avec ordre total uniforme qui soit optimal en débit, quels que soient le nombre de serveurs diffusant des messages. Nous avons réalisé une implémentation de cet algorithme et l'avons comparée aux deux implémentation de référence des protocoles de diffusion. Les résultats obtenus confirment que notre algorithme est en optimal en débit.

Mots-clés: système de stockage réparti, sémantique atomique, journalisation, ordre total, débit, complexité.

Acknowledgements

I am grateful to many people for help, both direct and indirect, in writing this thesis. However, most of my gratitude goes out towards my advisor Prof. Rachid Guerraoui, for without his continuous support, dedication and patience, none of this work would have been possible. I will fondly remember our many discussions, especially those held while surfing in Morocco and skiing in the Swiss Alps.

I would like to thank to Prof. Roger Hersch for presiding over my thesis exam and the members of the jury, Prof. Dejan Kostic, Prof. Fernando Pedone, and Prof. Luis Rodrigues, for the time they spent examining my thesis.

Of all the people I worked with, I spent the most time working with Vivien Quéma. It was truly a pleasure to discover algorithms together and to talk about soccer (forza Lyone), french politics and the upcoming scientific revolution (freecast). I would like to thank him for the hours he spent running experiments till late at night, his numerous visits to Lausanne and the times when he and Emilie housed me in Rome and in Grenoble.

Many thanks to all my friends and collegues at the EPFL who made my time as a PhD student truly memorable. In no particular order: Partha Dutta for always having answers to my questions and for inviting me to India. Maxime Monod for being the unofficial office event planner, always bringing ample food supplies and preparing me for combat. Bastian Pochon for discussing important algorithms while sailing and for showing me my office from the air. Sébastien Baehni for teaching me the fine art of "deskNis". Laura Gui for making me dance. Oana Jurca for inviting me to her wedding and for the ţuică I got as a souvenir. Jesper Honig Spring for sharing his passion for entrepreneurship, liquorice and women. Marko Vukolič for all things byzantine, tzatziki and beer! Fabien Salvi for patiently fixing my computer each time I managed to break it. David Leroux for his taste in cars and Basile Schaeli for calling to remind me to go for lunch.

A very special thanks goes out to Kristine Verhamme for making my life (and those of all other lab members) so much easier by making sure that everything is always taken care of. If there is any person at the EPFL who truly deserves a prize for her work it is Kristine. Thanks for caring.

Last but not least, I would like to thank my family – Mike, Jos and Danny – for standing by me during my PhD; but then again they always have.

Preface

This thesis concerns the PhD work I did under the supervision of Prof. Rachid Guerraoui at the School of Computer and Communication Sciences, EPFL, from 2002 to 2007. During this period, besides the work presented in this thesis, I also worked on eventually consistent gossip algorithms [BGLQT06] and byzantine storage implementations [DGLV05,GLV06].

This thesis focuses on the complexity of reliable distributed storage and is a composition of five published papers [GL04,CDGL04,GLPQ06,GKLQ07,GLPP07] and one that has been submitted to a peer reviewed journal [GLQ07].

[GL04] R. Guerraoui and R.R. Levy. Robust Emulations of Shared Memory in a Crash-Recovery Model. In *24th IEEE International Conference on Distributed Computing Systems (ICDCS'04)*, pages 400–407.

[CDGL04] A. Chakraborty, P. Dutta, R. Guerraoui and R. R. Levy. How fast can a distributed atomic read be? In *Proceedings of the twenty-third annual ACM symposium on Principles of distributed computing (PODC'04)*, pages 236–245. ACM Press.

[DGLV05] P. Dutta, R. Guerraoui, R. R. Levy, and M. Vukolič. How Fast can a Distributed Atomic Read be? Technical report.

[GLPQ06] R. Guerraoui, R. R. Levy, B. Pochon, and V. Quéma. High Throughput Total Order Broadcast for Cluster Environments. In *Proceedings of the IEEE International Conference on Dependable Systems and Networks (DSN'06)*.

[GLV06] R. Guerraoui, R. R. Levy, and M. Vukolic. Lucky Read/Write Access to Robust Atomic Storage. In *IEEE International Conference on Dependable Systems and Networks (DSN'06)*.

[BGLQT06] R. Baldoni, R. Guerraoui, R. R. Levy, V. Quéma, and S. Tucci Piergiovanni. Unconscious Eventual Consistency with Gossips. In *Eighth International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS'06)*.

[GKLQ07] R. Guerraoui, D. Kostic, R. R. Levy, and V. Quéma. A High Throughput Atomic Storage Algorithm. In *The 27th IEEE International Conference on Distributed Computing Systems (ICDCS'07)*.

[GLPP07] R. Guerraoui, R. R. Levy, B. Pochon, and J. Pugh. The Collective Memory of Amnesic Processes. *ACM Transactions on Algorithms (TALG)*.

[GLQ07] R. Guerraoui, R. R. Levy, and V. Quéma. Throughput optimal uniform total order broadcast for cluster environments. Technical report.

Contents

1 Introduction

> Interestingly, according to modern astronomers, space is finite. This is a very comforting thought – particularly for people who can never remember where they have left things.
>
> Woody Allen

A growing number of software services require fast and reliable storage of large quantities of data. Such services include stock exchange, data warehouse and e-commerce applications where data needs to be written and read concurrently by many clients. Traditionally, a centralized and highly specialized server is used for mission critical storage systems. The cost of such specialized centralized storage servers is very high, they do not offer protection against the loss of the entire server due to unforseen consequences and they are not very scalable. Distributed storage systems [Abd-El-Malek et al. 2005; Chun et al. 2006; Kenchammana-Hosekote et al. 2004; Saito et al. 2004] are gaining in popularity as appealing alternatives to their expensive centralized counterparts. A distributed storage system relies on a *cluster* of cheap distributed commodity machines. The goal of this distribution is to ensure resilience on the one hand and, on the other hand, to provide scalability by adjusting the number of servers to the number of clients to be served concurrently. Thus, not only are distributed storage systems cheaper, they are also more scalable. Figure 1.1 illustrates the difference between a centralized and a distributed storage.

At the heart of such a distributed storage system lies a *storage algorithm*. In short, a distributed storage algorithm provides an abstraction, usually called a *register* [Lamport 1978], that can be accessed by several clients concurrently. Clients can perform *read* and *write* operations on the storage with the knowledge that a read will return a recently written value. Distributed storage systems combine multiple read/write objects, each storing its share of data, as building blocks for a single large storage system. Not surprisingly, the performance of such a storage system depends on the performance of the underlying storage algorithm implementing the read/write objects.

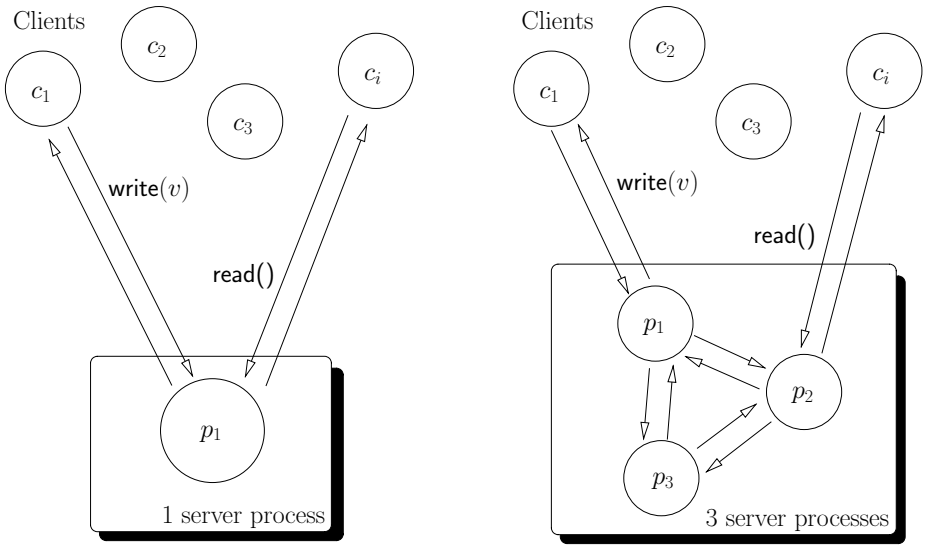

Figure 1.1: Difference between a centralized and a distributed storage.

The focus of this thesis is on *reliable* distributed storage. We consider *atomic* and *resilient* storage that can be read or written by an unbounded number of clients. An atomic storage is convenient because it provides the illusion of instantaneous execution despite concurrency: every *read* or *write* operation appears to take effect at some individual instant within the time interval between its invocation and reply events. To the clients it is as if they were accessing every variable sequentially one after the other [Lamport 1978; Herlihy and Wing 1990]. It was recently argued [Kenchammana-Hosekote et al. 2004; Saito et al. 2004] that atomicity is a desired property for distributed storage systems. In our context, *resilience* means that every non-faulty client eventually gets a reply to every (read or write) invocation, despite the failures of other clients or (a subset of the) servers. In short, from the client's perspective, an atomic and resilient distributed storage behaves like a highly available and scalable centralized storage.

This thesis improves the performance of storage algorithms along three dimensions which are introduced below:

1.1 Time-Complexity

Time-complexity is an indication of the latency of a single operation. The latency is defined as the time elapsed between the invocation and the response of the operation. A fast storage will have a low latency and conversely, a slow storage a high latency.

Time-complexity is typically measured in communication rounds, where in each round, every server process is supposed to:

1. compute a message at the beginning of the round,

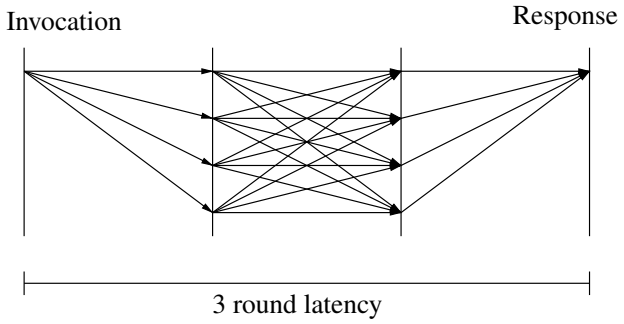

Figure 1.2: Time-complexity of an operation. In this example there are 3 rounds between the invocation and the response of the operation.

2. send (or broadcast) the message, and

3. receive all messages sent at the same round unless the sending process has crashed.

Latency is measured when there is a single client performing a single operation. Concurrent accesses to the storage are not taken into account.

We address the problem of designing an efficient implementation of an atomic read-write storage in an asynchronous system, i.e. where there are no bounds on message delays. Previous implementations tolerate the failure of any minority of servers (i.e., $f < n/2$ in a system with n servers where f servers may fail by crashing) and require 2 communication rounds (1 round-trip) for every write, and 4 communication rounds (2 round-trips) for every read. We are interested in *fast* implementations, namely, implementations that complete both reads and writes in 1 round-trip. Obviously, this represents the lowest possible time-complexity for a distributed storage. We show that the existence of a fast implementation depends on the maximum number of readers considered. More precisely, we demonstrate that a fast implementation is possible *if and only if* the number of readers is less that $n/f-2$. We also show that a fast implementation is *impossible* for multiple writers.

Our results draw sharp lines between the time-complexity of regular and atomic register implementations, as well as between single-writer and multi-writer implementations. The results also lead to revisit, in a message-passing context, the folklore theorem that "atomic reads must write".

1.2 Log-Complexity

Obviously, in most distributed systems, servers that crash are not thrown away, but are restarted after being fixed; sometimes they automatically recover without manual intervention. Forcing such recovered processes to remain out of the computation is not natural. Especially in long running applications, it can indeed be advantageous to take this recovery capacity into account, hopefully providing higher resilience for the overall system. However, a computer that crashes loses

the contents of its volatile memory. The only way it can remember its state before crashing is to continuously log information to its stable storage. However, logging to stable storage has a significant cost. In our local area network of Pentium IV workstations for instance, it takes around 0.1ms for a message to transit between two processes located at different workstations, whereas logging a single byte on a local disk might take twice as long; comparatively, it costs almost nothing for a process to execute a local operation that accesses only its volatile memory. But how do we take into account *log-complexity*? To illustrate this question, consider the implementation of a *write* operation using two algorithms A and B of Figure 1.3: [1]

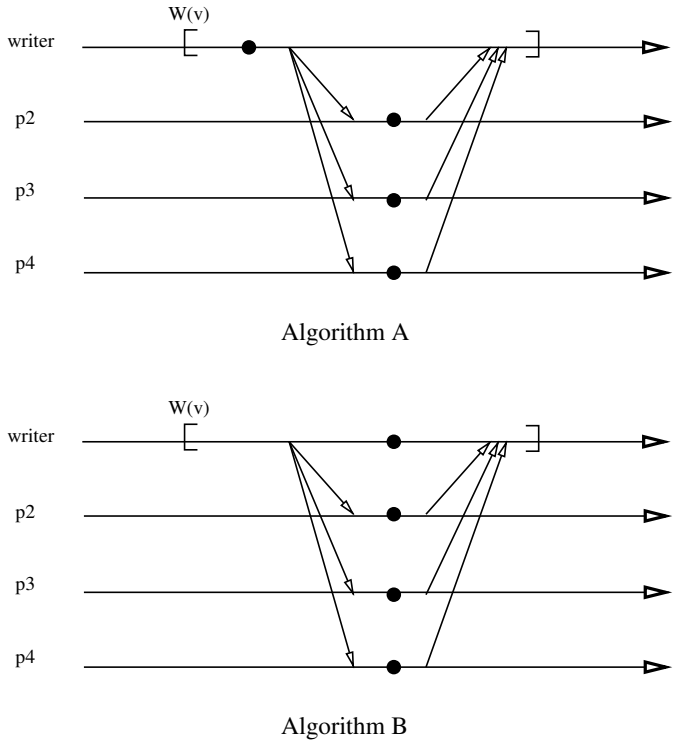

Figure 1.3: Differences in log-complexity between algorithms A and B. Logs are represented by dots on the time line. A *write* operation costs 2 causally related logs (or simply logs) in algorithm A and 1 log in algorithm B.

1. In algorithm A, the writer process first *logs* some information, then sends a message to all processes. Every server process that gets the message also *logs* some information, except the writer, before sending back an acknowledgment (ack). Once the writer gets back all acks, it returns from the *write*.

2. In algorithm B, the writer directly sends a message to all server processes. Every server process that gets the message logs some information before

[1] None of these algorithms robustly implement an atomic storage, but this is irrelevant for explaining the notion of log-complexity.

sending back an ack. Once the writer gets back all acks, it returns from the *write*.

In both algorithms, a *write* operation requires 2 communication steps, i.e. 1 round-trip between the writer and the rest of the processes. But how many logs are used in each algorithm? At first glance, it might appear that both algorithms use the same number of logs. Indeed, in both cases, all processes must log to terminate the *write*. However, a closer look at the algorithms reveals that logs are not used in the same manner. In A, the log of the writer *causally precedes* [Lamport 1978] the log of the other processes, whereas in B, there is no such causal precedence: all logs can be performed in parallel. We say that a *write* operation costs 2 causally related logs (or simply logs) in algorithm A and 1 log in algorithm B. In practice, even if distributed storage algorithms are devised in an asynchronous model, the most frequent case for which they need to be optimized is when the message transmission delay is within a reasonable time period (0.1 ms in our network). If we define the communication delay as δ and the log delay as λ, a *write* with A costs $2\delta + 2\lambda$, whereas a *write* with B only costs $2\delta + \lambda$.

We revisit the reasoning tools underlying atomicity in a crash-recovery model and give a generic algorithm that implements a multi-writer/multi-reader atomic storage in a crash-recovery model. Our algorithm is generic in the sense that it uses an abstract notion of *amnesia masking storage* which can be instantiated for several kinds of crash-recovery systems according to whether or not processes have access to stable storage and whether we can assume that a subset of processes do not crash in every execution.

Considering a system with n processes, including s processes with stable storage, a maximum of f faulty processes that can crash permanently or keep crashing and recovering forever, and u processes that do not crash, we establish the optimality of specific instances of our algorithm by proving the following bounds:

1. *Resilience*: $f < n/2$ and $u > f$ if $s \leq 2f$.

2. *Log-complexity*: If $s > 2f$ and $u \leq f$, 2 logs per *write* and 1 per *read* are necessary for a single writer/single reader and sufficient for a multi reader/multi writer storage algorithm.

3. *Time-complexity*: If $s = 0$, more than 1 round trip per *write* is necessary for a single writer and multi reader storage algorithm[2]. If $s \neq 0$ then 1 round-trip per *write* is sufficient for a single writer storage algorithm.

1.3 Throughput

In practice, when a high number of clients are served concurrently, low overall latency can only be provided with high *throughput*. In short, under high load, the latency perceived by clients is the sum of the time spent waiting for the operation

[2]The time-complexity of a *read* can be derived from existing results in crash-stop model [Attiya et al. 1995; Dutta et al. 2004].

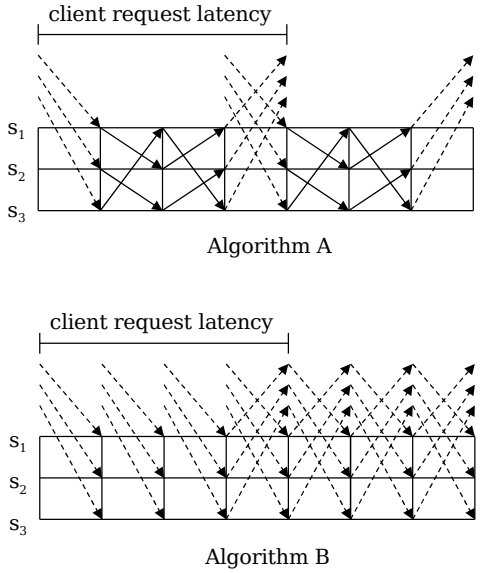

Figure 1.4: Throughput comparison between two algorithms: A and B. Both have the same latency but the throughput of B outperforms that of A. Client server communication messages are represented by the dotted lines.

to be served plus the actual service time. Clearly, when a lot of clients access the storage concurrently, the higher the throughput, the smaller the waiting time. Ideally, one would aim at *scalability*, meaning that increasing the number of machines should improve the throughput of the storage system.

To motivate the design of our algorithm and illustrate why studying isolated latency might be misleading, we compare two algorithms as presented in Figure 1.4. A *quorum-based* traditional one [Attiya et al. 1995; Lynch and Shvartsman 1997] and a less traditional one without inter-server communication. The example involves three servers and clients performing read operations on the storage. Clients always first contact a single server and communication between servers proceeds in a round-based manner. For simplicity, we assume that sending and receiving a message always takes the same time: one round. Therefore in each round, a server can receive a single message and send a single message. Algorithm A is a majority based algorithm: 2 out of 3 servers are needed to complete each operation. Upon receiving a request, server s_1 contacts s_2, and upon receiving a reply from s_2, replies to the client. Likewise, upon receiving client requests, s_2 contacts s_3 and s_3 contacts s_1. Thus, the servers need 3 rounds before they can receive a new client request. Under full load, the servers can complete 3 requests every 3 rounds, inducing a throughput of 1 read operation per round. In algorithm B the servers do not communicate in order to complete a read request. The client request latency is the same as that of algorithm A: 4 rounds. However, after an initial period of 4 rounds, the servers can complete 3 read operations each round, achieving a throughput of 3 read operations per round.

We exploit the availability of reliable failure detection in a homogeneous cluster

environment to provide high throughput distributed storage. We provide an atomic storage algorithm that is resilient to the crash failure of any number of readers and writers as well as to the crash failure of all but one server. In failure-free and synchronous periods, our algorithm has an optimal read throughput that grows linearly with the number of available servers. This is ensured even in the face of write contention. We organize the servers following a *ring* to ensure constant write throughput and avoid collisions during concurrent writes.

The implementation of our algorithm was evaluated on a cluster of 24 machines (dual Intel 900MHz Itanium-2 processors, 3GB of RAM) with dual fast ethernet network interfaces (100 Mbps). We achieve 81 Mbps of write throughput and 8×90 Mbps of read throughput (with up to 8 servers). Our algorithm is the first atomic storage algorithm to achieve a read throughput that grows linearly with the number of available servers.

Total Order. The idea of organizing servers in a ring combined with reliable failure detection allows us to apply the same design principles to build an even stronger abstraction, allowing us to ensure fault tolerance through state machine replication[3]. The key to making state machine replication work is a well designed software layer that hides all the difficulties behind replication from the application developer and renders it transparent to the clients.

Replication is typically used in replicated databases [Cecchet et al. 2004]. The idea is that all databases process the same write queries (INSERT and UPDATE) in the same order. Read queries (SELECT) do not change the state of the replicated database and do not have to be performed by all replicas. It is crucial that replicas do not execute any write query before making sure that all other replicas will also execute it. Indeed, consider the case where a replica executes a write request w_1, subsequently answers a client's read request r_1 and fails. If the other replicas do not execute w_1, the value of r_1 returned to the client will not be consistent with the replicated database.

Replication relies on an underlying ordering mechanism which ensures that all replicas perform the same operations on their copy in the same order, even if they subsequently fail. This mechanism is encapsulated by a communication abstraction called *uniform total order broadcast* (UTO-broadcast) [Hadzilacos and Toueg 1993]. Uniformity prevents faulty replicas from performing operations on their copy that will not be performed by the other (correct) replicas.

The *throughput* of a UTO-broadcast algorithm is crucial to the throughput of the associated replication mechanism. Throughput measures the number of requests that can be handled by the replicas under high load.

Numerous UTO-Broadcast algorithms have been published [Défago et al. 2004]. Algorithms relying on communication history [Peterson et al. 1989; Malhis et al. 1996; Ezhilchelvan et al. 1995; Ng 1991; Moser et al. 1993] and destination agreement [Chandra and Toueg 1996a; Birman and Joseph 1987b; Luan and Gligor 1990; Fritzke et al. 2001; Anceaume 1997] do not have good throughput

[3]State machine replication is more general than data replication. State machine replication allows any generic application to be replicated for fault tolerance while data replication can only be used to build a reliable storage application.

as they rely on a quadratic number of messages and an underlying consensus sub-algorithm. Low throughput is also the case for algorithms relying on a fixed sequencer [Kaashoek and Tanenbaum 1996; Armstrong et al. 1992; Carr 1985; Garcia-Molina and Spauster 1991; Birman and van Renesse 1993; Wilhelm and Schiper 1995]. While requiring fewer messages than the previously mentioned class of algorithms, they still exhibit bad throughput because the sequencer becomes a bottleneck. Algorithms using moving sequencers [Chang and Maxemchuk 1984; Whetten et al. 1994; Kim and Kim 1997; Cristian et al. 1997] have been proposed to overcome the limitation of fixed sequencer algorithms. While significantly improving the throughput, these algorithms do nevertheless not achieve optimal throughput due to the impossibility of piggy-backing acks in certain broadcast patterns (e.g. 1-to-n). Finally, a class of UTO-broadcast algorithms, called privilege-based algorithms [Friedman and Renesse 1997; Cristian 1991; Ekwall et al. 2004; Amir et al. 1995; Gopal and Toueg 1989], uses a ring topology of processes and a token passed among processes to grant the privilege of broadcasting. These algorithms are throughput optimal in the 1-to-n and n-to-n case, but not in the k-to-n case ($k \neq 1, n$). For instance, when two processes simultaneously want to broadcast messages, for fairness reasons, the token is constantly passed from one sender to the other, which reduces the throughput.

Our algorithm (called LCR) provides throughput optimality and fairness, regardless of the type of traffic. In our context, fairness conveys the equal opportunity of processes to have their broadcast messages delivered. We give a careful analysis of LCR's performance and fairness. We also provide performance results based on C and Java implementations of LCR that rely on TCP channels. The implementations are benchmarked against Spread and JGroups on a cluster of 9 machines and we show that LCR consistently delivers the highest throughput.

Outline. Chapter 2 looks at the time-complexity of storage algorithms by determining exactly how fast operations can be. Chapter 3 studies storage algorithms in the crash-recovery model through the notion of log-complexity. Chapter 4 looks at the throughput of storage and atomic broadcast algorithms and Chapter 5 concludes the thesis.

2

Time-complexity

A man with a watch knows what time it is.
A man with two watches is never sure.

Segal's Law

2.1 Introduction

Informally, atomicity requires that, even though each read or write operation may take an arbitrary period of time to complete, they appear to be instantaneous at some point in time, during their respective period of execution [Lamport 1985]. This requires ordering operations in such a way that they respect their physical order as well as the expected sequential specification of a read-write storage, namely, a read should return the last value written. The original implementation of [Attiya et al. 1995] maintains the required order among operations by associating timestamps with every written value. To write some value v, the writer increments its local timestamp, and sends v with the new timestamp ts to all processes (in [Attiya et al. 1995], readers and servers are the same set, the writer is one of the servers, and a minority of processes may crash). Every process, on receiving such a message, stores v and ts and then sends an acknowledgment (an ack) to the writer. On receiving acks from a majority, the writer terminates the write. In a read operation, the reader first gathers value and timestamp pairs from a majority of processes, and selects the value v with the largest timestamp ts. Then the reader sends v and ts to all processes, and returns v on receiving acks from a majority of processes.

Roughly speaking, the ordering is ensured by associating the value (that is written or read) to the largest timestamp in the system, and then storing it at a majority of processes. Since we are in a single-writer setting, and since only the writer introduces new timestamps in the system, the writer always knows the latest timestamp. Thus, on invoking a write operation, the writer just needs to increment its own timestamp to get a timestamp that is higher than any existing timestamp in the system. On the other hand, a reader does not know the latest

timestamp in the system, and hence, needs to spend one communication round-trip to know the latest value, and then another round-trip to send the value to a majority of processes.

The second round-trip is "required" in the above algorithm because the latest value learned in the first round-trip might be present at only a minority of processes. In a sense, every read includes, in its second communication round-trip, a "write phase", with the input parameter being the value selected in the first round-trip. This observation is captured by the folklore theorem that "atomic reads must write", which is actually borrowed from similar results in the shared-memory model [Lamport 1985; Attiya and Welch 1998]. In particular, a theorem from [Attiya and Welch 1998] states that, to simulate a multi-reader atomic storage from single-reader atomic storage, at least one of the readers must write into some single-reader storage. Along the same lines, when implementing atomic storage over weaker *regular* ones [Lamport 1985], a process that reads a value v also needs to write it, in order to make sure that no other process will subsequently read an *older* value v': with a regular storage, even if a value v' is written before a value v, v might be read before v', which is impossible with an atomic storage. Recently, [Fan and Lynch 2003] has shown that, in a message-passing system with n servers out of which f can fail, every atomic read must modify the state of at least f servers. However, in such a system, any message received by a server can potentially modify the server's state. Hence, a read can modify at least $n - f > f$ servers (assuming a majority of correct servers) in one round-trip. Thus [Fan and Lynch 2003] does not answer the question whether the second round-trip is necessary. Intuitively, processes (servers) are smarter than a basic (regular or single-reader) storage and might do more. Hence the motivation for this chapter: to determine the time-complexity of an atomic read.

In fact, we can reduce the time-complexity of the reads in [Attiya et al. 1995] by using a simple decentralization combined with a *max-min* technique. First, the reader sends messages to all servers. Every server, on receiving such a message, broadcasts its timestamp to all servers. On receiving timestamps from a majority of servers, every server selects the *maximum* timestamp, adopts the timestamp and its associated value, and sends the pair to the reader. On receiving such messages from a majority of servers, the reader returns the value with the *minimum* timestamp. To see why this ensures atomicity, observe that, when a write completes, its timestamp, say ts, is stored at a majority of servers. In any subsequent read, every server sees a timestamp that is higher than ts, before the server sends the message to the reader. Hence, the read returns a value that is not older than the written value. On the other hand, if a read returns a value with timestamp ts, then a majority of servers have a timestamp not lower than ts, and no subsequent read returns an older value. But can we do better? Is there a *fast* implementation where none of the operations (read or write) require more than one communication round-trip? This would clearly be optimal in terms of time-complexity. For example, consider the case of a fast implementation with two readers and $f < n/2$. Note that in the case of a single-reader and where a minority of servers may crash ($f < n/2$), it is trivial to modify the algorithm of [Attiya et al. 1995] such that the read takes only one round-trip, i.e., does not "write": the read can return the latest value learned from the servers in the

first round trip, provided it is not older than the value returned in the previous read. Otherwise, the reader returns the same value as in the previous read. Since there is only one reader, this clearly orders the reads in the desired fashion and ensures atomicity. To illustrate the example with two readers, suppose the writer writes v with timestamp 7, and the write message is received only by one server p_1. (The write is incomplete.) The first reader gets information from a majority of servers that includes p_1. The read must return v because the reader does not know whether the write of v is complete or not, and this reader has to return the value of the last preceding write. Now suppose the second reader invokes a read, queries a majority of servers, and misses p_1. Clearly, the second read returns a value with a timestamp lower than 7, violating atomicity: the second read returns an older value than the preceding read. At first glance, it seems impossible to have a fast implementation with two readers. But what if we tolerate fewer faulty servers?

2.1.1 Contributions

We show that, interestingly, the existence of a fast single-writer multi-reader (SWMR) atomic storage implementation depends on the maximum number R of readers. More precisely, the primary contribution is to show that there is a fast implementation of a SWMR storage if and only if $R < n/f - 2$.

1. Our fast implementation is based on the following observation: if a reader sees the latest timestamp ts at x servers, then any subsequent reader sees ts or a higher timestamp at $x - f$ servers; this is because, in a fast implementation, the first reader does not propagate ts, and the second reader might miss f servers seen by the first reader. A generalization of this observation helps determine when some reader can safely return the value associated with the latest timestamp. This is not entirely trivial because the safety of a value can not be simply deduced from the number of servers that has seen the value. To determine whether a value is safe, we have every server maintain, besides the latest value, the set of readers to which the server has sent that value.

2. Given n and f, we prove by contradiction that there is no fast implementation with $R \geq n/f - 2$. Given a fast implementation with $R \geq n/f - 2$, we consider a partial run which contains a write(1) that misses f servers, and we append it with a read that misses f other servers. Then we delete all the steps in the partial run that are not "visible" to the reader (basically, the steps of the f servers that the read missed). By atomicity, the read returns 1 in the resulting partial run. Now we iteratively append reads by distinct readers, and delete the steps in the partial run that are not visible to the last reader, until we exhaust all the readers. To ensure atomicity, the last read of each partial run returns 1. In the final partial run (obtained after exhausting all readers) the steps of write(1) are almost deleted. We modify this partial run to construct several additional partial runs, one of which violates atomicity.

Our result determines the exact conditions under which "atomic reads must write" in a message-passing system, and draws a line between the time-complexity of regular and atomic storage implementations. Indeed, whereas there is a fast implementation of an SWMR regular storage if and only if $f < n/2$, irrespective of the number of readers (as long as this number is finite), our result states that a fast implementation of a SWMR atomic storage exists if and only if $f < n/(R+2)$. The result also raises several interesting questions. One question has to do with the impact of multiple writers [Lynch and Shvartsman 1997]: is it possible to have a fast implementation of a MWMR atomic storage? We show that the answer is "no" if $f \geq 1$. A second question has to do with the length of the writes and whether semi-fast implementations are possible: namely, if we allow writes to be slow, can we have fast reads? This question raises the possibility of an interesting trade-off. In many practical settings, reads might be more frequent than writes, and we might like to have fast reads at the expense of slow writes. We partially determine this trade-off by showing that there is no (semi-fast) implementation of a SWMR storage with arbitrarily long writes and fast reads, if $R \geq n/f$.

2.1.2 Roadmap

The rest of the chapter is organized as follows. Section 2.2 sketches the system model and the necessary definitions. We present a fast implementation assuming $R < n/f - 2$ in Section 2.3. We prove $n/f - 2$ to be a tight bound for R in Section 2.5. Section 2.6 considers the multi-writer case and semi-fast implementations.

2.2 Model and Definitions

2.2.1 Basics

The distributed system we consider consists of three *disjoint* sets of processes: a set *servers* of size n containing processes $\{p_1, ..., p_n\}$, a set *writer* containing a single process $\{w\}$,[1] and a set *readers* of size R containing processes $\{r_1, ..., r_R\}$. Every pair of processes communicate by message-passing using a bi-directional reliable communication channel.

A distributed algorithm A is a collection of automata, where A_p is the automata assigned to process p. Computation proceeds in *steps* of A. A step of algorithm A is denoted by a pair of process id and a set of messages received in that step $< p, M >$ (M might be $\emptyset$). A *run* is an infinite sequence of steps of A. A *partial run* is a finite prefix of some run. A (partial) run r *extends* some partial run pr if pr is a prefix of r. At the end of a partial run, all messages that are sent but not yet received are said to be *in transit*. In any given run, any number of readers, the writer, and f out of n servers may crash.

[1] We discuss the multi-writer case in Section 2.6.

2.2.2 Atomic storage

A sequential storage is a data structure accessed by a single process. It provides two operations: write(v), which stores v in the storage, and read(), which returns the last value stored. An atomic storage is a distributed data structure that may be concurrently accessed by multiple processes and yet provides an "illusion" of a sequential (or centralized) storage to the accessing processes. We refer the readers to [Lamport 1985; Lynch 1996; Herlihy 1991; Herlihy and Wing 1990] for a formal definition of an atomic storage, and we simply recall below what is required to state and prove our results.

We assume that each process invokes at most one invocation at a time (i.e., does not invoke the next operation until it receives the response for the current operation). Only readers invoke read on the storage and only the writer invokes write on the storage. We further assume that the initial value of a storage is a special value $\perp$, which is not a valid input value for a write. In any run, we say that an operation $op1$ *precedes* operation $op2$ (or $op2$ *succeeds* $op1$) if the response step of $op1$ precedes the invocation step of $op2$ in that run. If neither $op1$ nor $op2$ precedes the other, the operations are said to be *concurrent*. We say that an operation is *complete* in a (partial) run if the run contains a response step for that operation.

An algorithm *implements* a storage if every run of the algorithm satisfies *termination* and *atomicity* properties. Termination states that if a process invokes an operation, then eventually, either the operation completes or the process crashes. Here we give a definition of atomicity for *single-writer* storage.

In the single-writer setting, the writes in a run have a natural ordering which corresponds to their physical order. Denote by wr_k the k^{th} write in a run ($k \geq 1$), and by val_k the value written by the k^{th} write. Let $val_0 = \perp$. We say that a partial run satisfies atomicity if the following properties hold: (1) if a read returns x then there is k such that $val_k = x$, (2) if a read rd is complete and it succeeds some write wr_k ($k \geq 1$), then rd returns val_l such that $l \geq k$, (3) if a read rd returns val_k ($k \geq 1$), then wr_k either precedes rd or is concurrent to rd, and (4) if some read $rd1$ returns val_k ($k \geq 0$) and a read $rd2$ that succeeds $rd1$ returns val_l, then $l \geq k$.

A *history* of a partial run is a sequence of invocation and response steps of read or write operations in the same order as they appear in the partial run. An *incomplete* invocation step in a history is an invocation step without a matching response step in that history. We say that a history $H1$ *completes* history $H2$ if $H1$ can be obtained through the following modification of $H2$: for each incomplete invocation step sp in $H2$, either sp is removed from $H2$, or any valid matching response for that invocation is appended to the end of $H2$.

A run satisfies atomicity, if for every history H' of any of its partial runs, there is a history H that completes H' and H satisfies the properties 1-3 below (Lemma 13.16 of [Lynch 1996]). Let Π be the set of all operations in H. There is an irreflexive partial ordering $\prec$ of all the operations in H such that: (1) if $op1$ precedes $op2$ in H then it is not the case that $op2 \prec op1$, (2) if $op1$ is a write operation in Π and $op2$ is any other operation in Π, then either $op1 \prec op2$ or $op2 \prec op1$ in Π, and (3) the value returned by each read operation is the value

written by the last preceding write operation according to $\prec$ (or $\perp$ if there is no such write operation).

2.2.3 Fast Implementations

Basically, we say that a read or a write operation is *fast* if it completes in one communication round-trip. In other words, in a fast read:

1. The reader sends messages to a subset of processes in the system (possibly all processes).
2. Processes on receiving such a message reply to the reader before receiving any other messages. Intuitively, this requirement forbids the processes to wait for some other message before replying to m.
3. The reader on receiving a sufficient number of such replies returns from the read.

Recall that implementations need to tolerate the crash of the writer, any reader, and up to f servers. Hence, in order to ensure termination, the reader cannot wait for replies from any other reader, or writer, or more than $n - f$ servers. We similarly say that a write operation is fast if it completes in one round-trip.

We say that an implementation has fast reads (or writes) if every complete read (resp. complete write) operation in every run is fast. A *fast implementation* is an implementation in which both reads and writes are fast. For an implementation that has fast reads, we can say without ambiguity that the messages sent by a reader, on invoking a read, are of type READ, and the messages sent by a process to the reader, on receiving a READ message, of type READACK. Similarly, we define WRITE and WRITEACK messages for fast writes.

2.3 A Fast Implementation

We describe in this section a fast implementation assuming $R < n/f - 2$ (Figure 2.1). For simplicity of presentation, we assume that the writer writes timestamps, and the readers read back timestamps. We ignore the value associated with the timestamp for now. Later we explain how to trivially modify our algorithm such that the writer and the readers associate some value with a timestamp.

The write procedure is similar to that of [Attiya et al. 1995]. On invoking a write, the writer increments its timestamp and sends a WRITE message with the timestamp to all servers. Servers on receiving the message store the timestamp, and send WRITEACK messages back to the writer. The writer returns OK once it has received WRITEACK messages from $n - f$ servers.

Implementing a fast read is more involved. Recall that, to maintain atomicity, a read needs to return a timestamp that is not lower than the timestamp of the last completed write, and has to guarantee that no subsequent read returns a lower timestamp. Our read procedure collects timestamps from $n - f$ servers (by sending READ messages and receiving READACK messages from the servers),

1: at the writer w
2: **procedure** initialization:
3: $ts \leftarrow 1$, $rCounter \leftarrow 0$
4: **end**
5: **procedure** write(v)
6: send(WRITE, ts, $rCounter$) to all servers
7: **wait until** receive(WRITEACK, ts, $*$, $rCounter$) from $n - f$ servers
8: $ts \leftarrow ts + 1$
9: return(OK)
10: **end**

11: at each reader r_i
12: **procedure** initialization:
13: $ts \leftarrow 0$; $rCounter \leftarrow 0$; $maxTS \leftarrow 0$
14: **end**
15: **procedure** read()
16: $rCounter \leftarrow rCounter + 1$; $ts \leftarrow maxTS$
17: send(READ, ts, $rCounter$) to all servers
18: **wait until** receive(READACK, $*$, $*$, $rCounter$) from $n - f$ servers
19: $rcvMsg \leftarrow \{m | r_i$ received (READACK, $*$, $*$, $rCounter)\}$
20: $maxTS \leftarrow$ **Maximum**$\{ts' |$ (READACK, ts', $*$, $rCounter) \in rcvMsg\}$
21: $maxTSmsg \leftarrow \{m | m.ts = maxTS$ **and** $m \in rcvMsg\}$
22: **if** there is $a \in [1, R+1]$ and there is $MS \subseteq maxTSmsg$ s.t., $(|MS| \geq n - af)$ **and** $(|\cap_{m \in MS} m.seen| \geq a)$ **then**
23: return($maxTS$)
24: **else**
25: return($maxTS - 1$)
26: **end if**
27: **end**

28: at each server p_i
29: **procedure** initialization:
30: $ts \leftarrow 0$; $seen \leftarrow \emptyset$; $counter[0...R] \leftarrow [0...0]$
31: **end**
32: **upon** receive($msgType$, ts', $rCounter'$) from $q \in \{w, r_1, ..., r_R\}$ **and** $rCounter' \geq counter[pid(q)]$ **do**
33: **if** $ts' > ts$ **then**
34: $ts \leftarrow ts'$; $seen \leftarrow \{q\}$
35: **else**
36: $seen \leftarrow seen \cup \{q\}$
37: **end if**
38: $counter[pid(q)] \leftarrow rCounter'$
39: **if** $msgType =$ READ **then**
40: send(READACK, ts, $seen$, $rCounter'$) to q
41: **else**
42: send(WRITEACK, ts, $seen$, $rCounter'$) to q
43: **end if**
44: **end upon**

Figure 2.1: Fast SWMR atomic storage implementation with $R < n/f - 2$.

and selects the highest timestamp, denoted by $maxTS$ in Figure 2.1. Then the reader checks if $maxTS$ has been seen by a "sufficient" number of servers and readers. If so, the read returns $maxTS$, else it returns $maxTS - 1$. The heart of the algorithm is the predicate for checking whether the latest value has been seen by a sufficient number of processes: (1) the predicate is true whenever the write with timestamp $maxTS$ precedes the current read, and (2) if there is no write with a timestamp higher than $maxTS$, then if the predicate is true for the current read, it is also true for all subsequent reads. In order to construct such a predicate however, the servers need to record more information than just the latest timestamp, as we explain below.

Consider the case of a write with timestamp ts that is followed by a read. In the first partial run pr_1, the write completes by writing ts at $n-f$ servers, say the set of servers be S_1. Subsequently, a reader reads from a set S_2 (of $n-f$ servers) that overlaps at $n-2f$ servers with S_1, i.e., misses f servers in S_1. By atomicity, the read returns ts. In the second partial run pr_2, the write is incomplete and the writer writes ts only to $n-2f$ servers in $S1 \cap S2$. A subsequent reader that reads from S_2 cannot distinguish pr_1 from pr_2, and returns ts. If we extend each partial run with another read by a distinct reader that misses f servers from $S1 \cap S2$, it is easy to see that the new read has to return ts, even if it sees ts at $n-3f$ servers that have already replied to both the write and the first read. Thus, we see that any reasonable predicate for fast reads must depend on the number of servers, as well as the number of readers, that have seen the most recent timestamp. Since any number of readers might crash, a reader cannot wait for the replies from other readers, but rather indirectly collect information about other readers from the servers.

Generalizing the above argument gives us the desired predicate. Along with the latest timestamp ts, every server maintains the list of readers and writer to which the server has replied after updating its timestamp to ts (including the reader or the writer which updated the timestamp of the server to ts). This set is denoted by *seen* in Figure 2.1. The predicate for the read procedure is as follows: if there is $a \geq 1$ such that the reader receives $maxTS$ in at least $n-af$ messages, and there are at least a processes that are in the list *seen* of each of these $n-af$ messages, then the predicate is true.

In addition, every reader r_i maintains a variable $rCounter$ that counts the number of reads of r_i. The servers keep an array, *counter*, such that $counter[i]$ contains the latest value of $rCounter$ that the server has received from r_i.[2] This helps distinguish READ and READACK messages from different reads of the same reader. At the writer, the variable $rCounter$ is always 0; the messages from different writes are distinguished by their respective timestamps.

This completes the brief description of the storage implementation. We now describe how to modify the algorithm so as to associate values with timestamps. In the modified algorithm, in each write, the writer attaches two tags with the timestamp, containing the current value to be written and the value of the immediately preceding write. If the reader returns $maxTS$ in the original algorithm, then it returns the current value attached to $maxTS$ in the modified algorithm.

[2]In the algorithm, $pid(q)$ is a function that maps the writer w to 0, and every reader r_i to i.

If the reader returns $maxTS - 1$ in the original algorithm, it returns the other tag attached to $maxTS$ in the modified algorithm.

2.4 Correctness

We prove here the correctness of the fast implementation in Figure 2.1. We do not assume that the lines in Figure 2.1 are atomic: processes may crash in the middle of a line or in between two lines. In particular, while sending messages to a set of processes, the sending process may crash after sending messages to an arbitrary subset. We assume that, if a process receives an incomplete message, the process can detect that the message is incomplete, and ignores such a message.

It is obvious that read and write procedures complete in one round-trip. To show atomicity, we recall that the write procedure directly writes the timestamp. Thus the conditions in Section 2.2.2 reduce to the following: (1) if a read returns, it returns a non-negative integer, (2) if a read rd is complete and it succeeds some write(k), then rd returns l such that $l \geq k$, (3) if a read rd returns k ($k \geq 1$), then write(k) either precedes rd or is concurrent to rd, and (4) if some read $rd1$ returns k ($k \geq 0$) and a read $rd2$ that succeeds $rd1$ returns l, then $l \geq k$. The proofs of the first and the third conditions are trivial. Below, we show the other two. In the proofs we refer to the global clock; however processes do not access this global clock.

Lemma 1 *If a server sets ts to x at time T, then the server never sets ts to a value that is lower than x after time T.*

PROOF: Obvious from line 27.

Lemma 2 *If a read() sends* READ *messages with $ts = x$, then the read does not return a value smaller than x.*

PROOF: Suppose read rd by r_i sends a READ message with $ts = x$. From line 27, every READACK message received by rd is with $ts \geq x$. Let z be the maximum timestamp received by rd (i.e., $maxTS$ computed in line 17). Notice that rd returns either z or $z - 1$. There are the following two cases to consider. (1) If $z > x$, then clearly, the return value is not smaller than x. (2) If $z = x$. then every READACK message received by rd has $ts = x$ and has $p_i \in seen$. Since rd receives $n - f$ READACK messages, the predicate in line 19 of rd holds with $a = 1$. Hence, rd returns x.

Lemma 3 *If a read rd is complete and it succeeds some write(k), then rd returns l such that $l \geq k$.*

PROOF: Suppose that write wr (by w) writes k and precedes read rd (by reader r_j). Let $S1$ be the set of $n-f$ servers from which wr received WRITEACK messages in line 6, and let $S2$ be the set of $n - f$ servers from which rd received READACK messages in line 15. Let $S12 = S1 \cap S2$. Obviously, $|S12| \geq n - 2f$. Let z be the maximum timestamp received by rd from servers in $S2$. Observe that rd returns either z or $z - 1$.

When a server in $S1$ replies to a WRITE message from wr, its ts is k.[3] From Lemma 1, servers in $S1$ (and hence, in $S12$) reply $ts \geq k$ to rd because write(k) precedes rd). Thus, the highest timestamp received by rd, $z \geq k$. There are the following two cases to consider:

1. $z > k$: As rd returns either z or $z-1$, it follows that rd does not return a timestamp lower than k.

2. $z = k$: We know that every server in $S12$ replies to rd with $ts \geq k$, and $z = k$ is the maximum timestamp received by rd from servers in $S2 \supseteq S12$. Thus every server in $S12$ replies $ts = k$ to rd. Let MS be the set of READACK messages sent by servers in $S12$ to rd. Since every server in $S12$ replies $ts = k$ to wr before sending $ts = k$ to rd, for every message m in MS, $w \in m.seen$. Furthermore, from line 30, $r_j \in m.seen$. Thus, $\{w, r_j\} \subseteq \cap_{m \in MS} m.seen$. As $|S12| \geq n - 2f$, in rd, the predicate in line 19 holds with $a = 2$. Consequently, rd returns $z = k$.

Lemma 4 *if some read $rd1$ returns x ($x \geq 0$) and a read $rd2$ that succeeds $rd1$ returns y, then $y \geq x$.*

PROOF: Suppose that read $rd1$ by process r_j returns x, read $rd2$ by process r_k returns z, and $rd1$ precedes $rd2$. Suppose $r_j = r_k$. Then, in the read immediately after $rd1$, r_j sends a READ message with $ts \geq x$, and hence, from Lemma 2, the read returns a value greater than or equal to x. Using Lemma 2 and a simple induction, we can derive that any read by r_j which follows $rd1$ (including $rd2$) returns $ts \geq x$. So in the rest of the proof we assume that $r_j \neq r_k$.
Let $S1$ and $S2$ be the set of servers (of size $n-f$) from which reads $rd1$ and $rd2$, respectively, receive $n-f$ READACK messages in line 15. Let $TS1$ be the highest timestamp received by $rd1$ from processes in $S1$ (i.e., the $maxTS$ evaluated in line 17 of $rd1$). Similarly, let $TS2$ be the highest timestamp received by $rd2$ from the processes in $S2$. There are the following two cases to consider:
$\langle 1 \rangle 1$. the predicate in line 17 does not hold in $rd1$.
It follows that $x = TS1 - 1$. Thus some servers have sent $ts = TS1 = x + 1$ to $rd1$, and hence, write($x+1$) has started before $rd1$ is completed. Thus write(x) has completed before $rd1$ is completed. Since $rd1$ precedes $rd2$, it follows that write(x) precedes $rd2$. From Lemma 3, $rd2$ returns $y \geq x$.
$\langle 1 \rangle 2$. the predicate in line 17 holds in $rd1$.
It follows that $x = TS1$, and there is some $a \in [1, R+1]$ such that there is a set MS consisting of at least $n - af$ messages received by $rd1$ with $ts = x$ and $|\cap_{m \in MS} m.seen| \geq a$. Let $S12 \subseteq S1$ be the set of servers which sent the messages that are in MS. Since $a \in [1, R+1]$ and $f < n/(R+2)$, $|S12| = |MS| = n - af > f$. There are the following two cases to consider:
$\langle 2 \rangle 1$. $y = TS2$
$y = TS2$. Since, $|S12| > f$ and $|S2| = n - f$, there is a server $p_i \in S2 \cap S12$. Since $rd1$ precedes $rd2$, p_i first replies $ts = x$ to $rd1$ then replies to $rd2$. From

[3] The server's ts is not higher than k because, unless the writer receives WRITEACK from all servers in $S1$, it does not complete write(k), and hence, no timestamp higher than k is present in the system until all servers in $S1$ reply to write(k).

Lemma 1, it follows that p_i replies to $rd2$ with $ts \geq x$. Thus the highest ts in $S2$ (i.e., $TS2 = y$) is greater than or equal to x.

$\langle 2 \rangle 2$. $y = TS2 - 1$

There are the following two subcases to consider:

$\langle 3 \rangle 1$. $y + 1 \neq x$

As in case $\langle 2 \rangle 1$, we can show that there is a server $p_i \in S2 \cap S12$, and p_i replies to $rd2$ with $ts \geq x$. Thus the highest ts in $S2$ (i.e., $TS = y + 1$) is greater than or equal to x. Since $y + 1 \neq x$, it follows that $y + 1 > x$, and hence, $y \geq x$.

$\langle 3 \rangle 2$. $y + 1 = x$

Consider the set of servers $S2 \cap S12$. As $|S12| = n - af$ and $|S2| = n - f$, so $|S2 \cap S12| \geq n - (a+1)f \geq 1$. Since $rd1$ precedes $rd2$ and processes in $S12$ replies $ts = x$ to $rd1$, processes in $S2 \cap S12$ reply to $rd2$ with $ts \geq x$. Since $y + 1$ is the maximum ts in $S2$, every process in $S2 \cap S12$ replies to $rd2$ with $ts = x = y + 1$. There are the following two cases to consider:

$\langle 4 \rangle 1$. $a \leq R$

Then $|S2 \cap S12| \geq n - (a+1)f > f$. Let $MS1$ be the set of READACK messages from processes in $S2 \cap S12$ to $rd1$. From the definition of $MS1$ and MS, $MS1 \subseteq MS$.[4] Thus, $\cap_{m \in MS1} m.seen \supseteq \cap_{m \in MS} m.seen$. Thus, $|\cap_{m \in MS1} m.seen| \geq a$. There are two cases to consider:

$\langle 5 \rangle 1$. $r_k \notin \cap_{m \in MS1} m.seen$

Let $MS2$ be the set of messages received by $rd2$ from processes in $S2 \cap S12$. For any server $p_i \in S2 \cap S12$, let $m1_i$ and $m2_i$ be the messages sent by p_i in $MS1$ and $MS2$ respectively. We know that $m1_i.ts = m2_i.ts = x$. Since $m1_i$ is sent before $m2_i$ and the ts is the same in both messages, $m1_i.seen \subseteq m2_i.seen$. Thus $\cap_{m \in MS1} m.seen \subseteq \cap_{m \in MS2} m.seen$. Since every process which replies to $rd2$, first adds r_k to its *seen* set, $r_k \in \cap_{m \in MS2} m.seen$. Since $r_k \notin \cap_{m \in MS1} m.seen$, it follows that $|\cap_{m \in MS2} m.seen| \geq |\cap_{m \in MS1} m.seen| + 1 \geq a + 1$. Since $|S2 \cap S12| \geq n - (a+1)f$, the number of message in $MS2$ is at least $n - (a+1)f$. As $a + 1 \leq R + 1$, the predicate in line 19 in $rd2$ holds with $a + 1$. Thus, the timestamp returned by $rd2$ is $x = y + 1$, a contradiction.

$\langle 5 \rangle 2$. $r_k \in \cap_{m \in MS1} m.seen$

Thus each server p_i in $S2 \cap S12$ has sent at least one READACK message with $ts = x$ to r_k, before p_i sent the $MS1$ message to r_j. Since the messages in $MS1$ are sent before the completion of $rd1$ (and hence, before the invocation of $rd2$), r_k has invoked at least one read before $rd2$. Let $rd2a$ be the last read of r_k which precedes $rd2$. Since $|S2 \cap S12| \geq n - (a+1)f > f$, there is at least one process p_i in $S2 \cap S12$ whose READACK message is received by $rd2a$, say message m. Now consider the last READACK message sent by p_i to r_k before $rd2$ is invoked, say message m'. Since we know that p_i sent a READACK message with $ts = x$ to r_k before sending a $MS1$ message (which was in turn sent before $rd2$ was invoked), from Lemma 1 it follows that m'

[4] See case $\langle 1 \rangle 2$ for the definition of MS.

was sent with $ts \geq x$. We now claim that $m = m'$. By definition of m', either $m = m'$ or m' is sent after m. Observe that p_i checks $counter[k]$ before replying to r_k. Thus, once m is sent by p_i, $counter[k]$ at p_i is set such that p_i can only reply to those message of r_k which are sent from $rd2a$ or a subsequent read of r_k. Thus, if m' is sent after m, then m' is sent in response to $rd2a$, or $rd2$, or a subsequent read of r_k. This contradicts the assumption that p_i replies only once to $rd2a$ (because channels do not duplicate messages) and m' is sent before $rd2$ is invoked. Thus $rd2a$ receives $m = m'$. We have already shown that m' is sent with $ts \geq x$. Hence the highest ts received by $rd2a$ is greater than or equal to x. It follows that $rd2$ sends READ messages with $ts \geq x$. From Lemma 2, $rd2$ returns a timestamp greater than or equal to x. As $x = y + 1$, $rd2$ does not return y, a contradiction.

$\langle 4 \rangle 2$. $a = R + 1$

Since $|\{w, r_1, ..., r_R\}| = R+1$ and $|\cap_{m \in MS} m.seen| \geq a = R+1$, we have $r_k \in \cap_{m \in MS} m.seen$. Observe that $|S12| \geq n - af > f$. (Recall that $S12$ is the set of processes which sent the messages that are in MS.) Substituting $MS1$ by MS, and $S2 \cap S12$ by $S12$, in the argument for the previous case (case $\langle 5 \rangle 2$), we can show that $rd2$ returns a value greater than or equal to x, a contradiction.

2.5 Optimality

The following proposition states that the resilience required by our fast implementation is indeed necessary.

Proposition 5 *Let $f \geq 1$ and $R \geq 2$. If $R \geq n/f - 2$, then there is no fast atomic storage implementation.*

Preliminaries. Recall first that w denotes the writer, r_i for $1 \leq i \leq R$ denote the readers, and p_i for $1 \leq i \leq S$ denote the servers. Suppose by contradiction that $R \geq n/f - 2$ and there is a fast implementation I of an atomic storage. Given that $f \geq n/(R+2)$, we can partition the set of servers into $R+2$ subsets (which we call *blocks*), denoted by B_i $(1 \leq i \leq R + 2)$, each of size less than or equal to f. For instance, one such partition is: for $1 \leq i \leq R+1$, $B_i = \{p_j \mid (\lfloor \frac{n}{R+2} \rfloor (i-1)+1) \leq j \leq (\lfloor \frac{n}{R+2} \rfloor i)\}$, and $B_{R+2} = \{p_j \mid (\lfloor \frac{n}{R+2} \rfloor (R+1)) \leq j \leq n\}$. However, if $R > n-2$ then the above partitioning is not possible. In that case we consider a system where, the number of readers is $n - 2$ and the set *readers* is $\{r_1, ..., r_{n-2}\}$, and show the impossibility. The impossibility still holds if we add more readers to this system (i.e., $R > n - 2$).

Since the writer, any number of readers, and up to f servers might crash in our model, the invoking process can only wait for reply messages from $n - f$ servers. Given that we assume a fast implementation, on receiving a READ (or a WRITE) message, the servers cannot wait for messages from other processes, before replying to the READ (or the WRITE) message. We can thus construct partial runs of a fast implementation such that only READ (or WRITE) messages from the invoking processes to the servers, and the replies from servers to the

invoking processes, are delivered in those partial runs. All other messages remain in transit. In particular, no server receives any message from other servers, and no invoking process receives any message from other invoking processes. In our proof, we only construct such partial runs.

We say that an *incomplete invocation inv skips* a set of blocks BS in a partial run, where $BS \subseteq \{B_1, ..., B_{R+2}\}$, if (1) no server in any block $B_i \in BS$ receives any READ or WRITE message from *inv* in that partial run, (2) all other servers receive the READ or the WRITE message from *inv* and reply to that message, and (3) *all these reply messages are in transit.* We say that a *complete invocation inv skips* a block B_i in a partial run, if (1) no server in B_i receives any READ or WRITE message from *inv* in that partial run, (2) all servers that are not in B_i receive the READ or WRITE message from *inv* and reply to that message, and (3) the invoking process *receives all these reply messages and returns from the invocation.*

PROOF: To show a contradiction, we construct a partial run of the fast implementation I that violates atomicity: a partial run in which some read returns 1 and a subsequent read returns an older value, namely, the initial value of the storage, $\perp$.

Partial writes. Consider a partial run wr in which w completes write(1) on the storage. The invocation skips B_{R+2}. We define a series of partial runs each of which can be extended to wr. Let wr_{R+2} be the partial run in which w has invoked the write and has sent the WRITE message to all processes, and all WRITE messages are in transit. For $1 \leq i \leq R+1$, we define wr_i as the partial run which contains an incomplete write(1) invocation that skips $\{B_{R+2}\} \cup \{B_j | 1 \leq j \leq i-1\}$. We make the following simple observations: (1) for $1 \leq i \leq R$, wr_i and wr_{i+1} differ only at servers in B_i, (2) wr is an extension of wr_1, such that, in wr, w receives the replies (that are in transit in wr_1) and returns from the write invocation, and hence, (3) wr and wr_1 differ only at w.

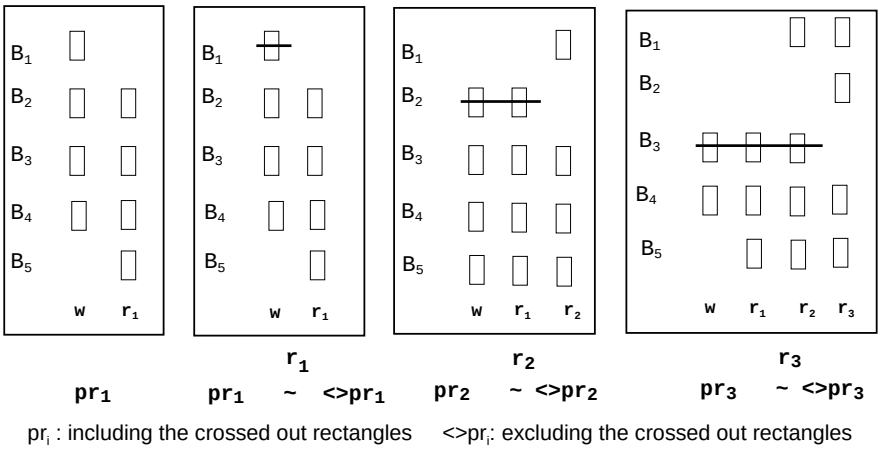

Figure 2.2: Partial runs: pr_i and $\triangle pr_i$.

Block diagrams. We illustrate a particular instance of the proof in Figure 2.2 and Figure 2.4, where $R = 3$ and the set of servers are partitioned into five blocks, B_1 to B_5. We depict an invocation inv through a set of rectangles, (generally) arranged in a single column. In the column corresponding to some invocation inv, we draw a rectangle in the i^{th} row, if all servers in block B_i have received the READ or WRITE message from inv and have sent reply messages, i.e., we draw a rectangle in the i^{th} row if inv does not skip B_i. Figure 2.3 presents a slightly more detailed diagram of the partial writes.

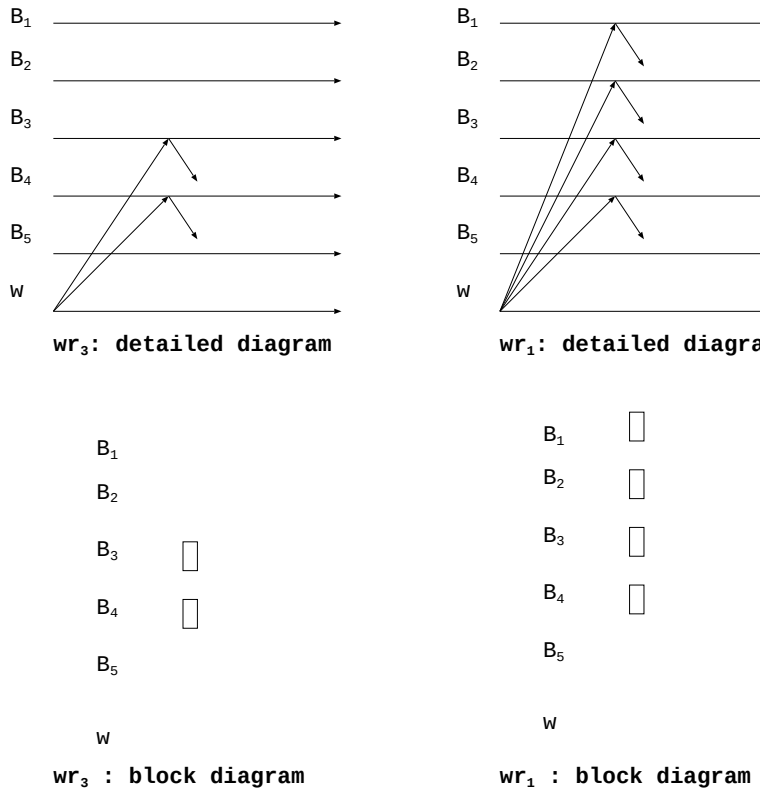

Figure 2.3: Partial writes: wr_i.

Appending reads. Partial run pr_1 extends wr by appending a complete read by r_1 that skips block B_1. By atomicity, the read returns 1. Observe that r_1 cannot distinguish pr_1 from some partial run $\triangle pr_1$, that extends wr_2 by appending a complete read by r_1 that skips B_1. To see why, notice that wr and wr_2 differ at w and at block B_1, and r_1 does not receive any message from these processes in both runs. Thus r_1's read returns 1 in $\triangle pr_1$.
Starting from $\triangle pr_1$, we iteratively define the following partial runs for $2 \leq i \leq R$. Partial run pr_i extends $\triangle pr_{i-1}$ by appending a complete read by r_i that skips B_i. Partial run $\triangle pr_i$ is constructed by deleting from pr_i, all steps of the servers in block B_i. Since the last read in pr_i by reader r_i skips block B_i, r_i cannot distinguish pr_i from $\triangle pr_i$. More precisely, partial run $\triangle pr_i$ extends wr_{i+1} by

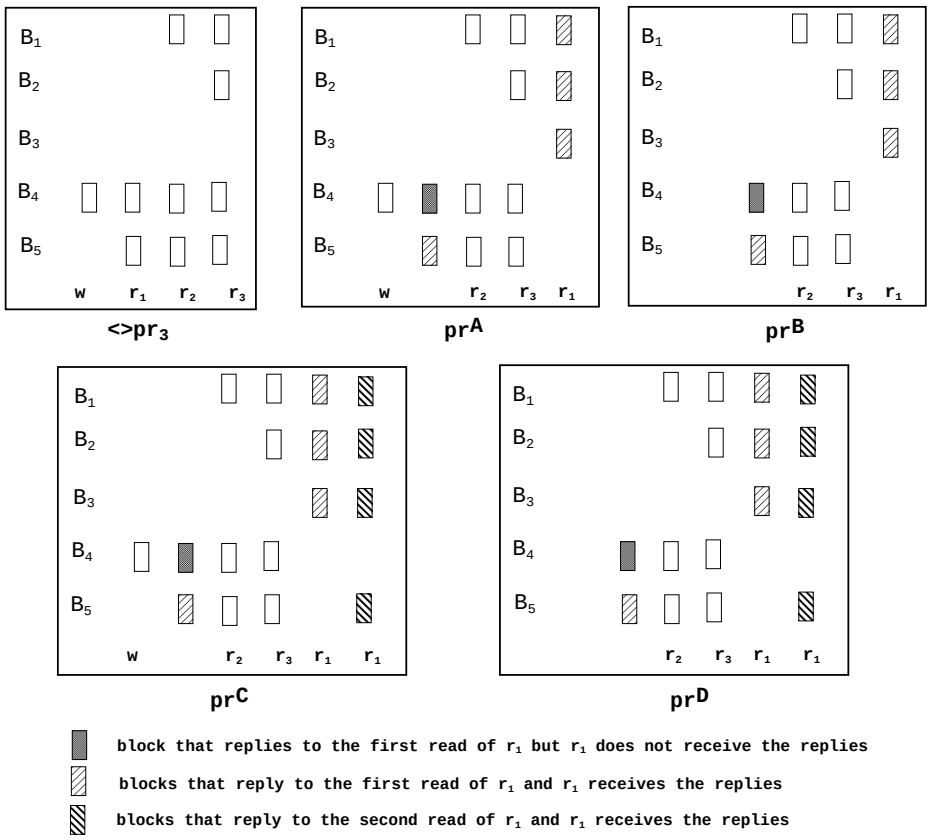

Figure 2.4: Partial runs: pr^A, pr^B, pr^C and pr^D.

appending the following i reads one after the other:[5] for $1 \leq h \leq i$, r_h does a read that skips $\{B_j | h \leq j \leq i\}$. Figure 2.2 depicts block diagrams of pr_i and $\triangle pr_i$ with $R = 3$. (The deletion of steps to obtain $\triangle pr_i$ from pr_i is shown by crossing out the rectangles corresponding to the deleted steps.)
Reader r_1's read in $\triangle pr_1$ returns 1. Since pr_2 extends $\triangle pr_1$, by atomicity, r_2's read in pr_2 returns 1. However, as r_2 cannot distinguish pr_2 from $\triangle pr_2$, r_2's read in $\triangle pr_2$ returns 1. In general, since pr_i extends $\triangle pr_{i-1}$, and r_i cannot distinguish pr_i from $\triangle pr_i$ (for all i such that $2 \leq i \leq R$), it follows from a trivial induction that r_i's read in $\triangle pr_i$ returns 1. In particular, r_R reads 1 in $\triangle pr_R$.

Partial run pr^A. Consider the partial run $\triangle pr_R$: wr_{R+1} extended by appending R reads by each reader r_h ($1 \leq h \leq R$) such that r_h's read skips $\{B_j | h \leq j \leq R\}$. The read by r_1 is incomplete in $\triangle pr_R$: only servers in B_{R+1} and B_{R+2} send replies to r_1, and those reply messages are in transit. Observe that, in $\triangle pr_R$, only the servers in B_{R+1} receive the WRITE message from the write(1) invocation. Consider the following partial run pr^A which extends $\triangle pr_R$ as follows. After $\triangle pr_R$, (1) r_1 receives the replies of its READ messages from B_{R+2} (that were in transit in $\triangle pr_R$), (2) the servers in B_1 to B_R receive the READ message from r_1 (that were in transit in $\triangle pr_R$) and reply to r_1, (3) reader r_1 receives these replies from

[5] The first $i - 1$ reads are incomplete whereas the last one is complete.

servers in B_1 to B_R, and then r_1 returns from the read invocation. (Notice that, r_1 received replies from $R+1$ blocks, and so, must return from the read.) However, r_1 does not receive the replies from servers in B_{R+1} (that were in transit in $\triangle pr_R$). Figure 2.4 depicts block diagrams for pr^A with $R = 3$.

Partial run pr^B. Consider another partial run pr^B with the same communication pattern as pr^A, except that write(1) is not invoked at all, and hence, servers in B_{R+1} do not receive any WRITE message (Figure 2.4). Clearly, only servers in B_{R+1}, the writer, and the readers r_2 to r_R can distinguish pr^A from pr^B. Reader r_1 cannot distinguish the two partial runs because it does not receive any message from the servers in B_{R+1}, the writer, or other readers. By atomicity, r_1's read returns (the initial value of the storage) $\perp$ in pr^B because there is no write($*$) invocation in pr^B, and hence, r_1's read returns $\perp$ in pr^A as well.

Partial runs pr^C and pr^D. Notice that, in pr^A, even though r_1's read returns $\perp$ after r_R's read returns 1, pr^A does not violate atomicity, because the two reads are concurrent. We construct two more partial runs: (1) pr^C is constructed by extending pr^A with another complete read by r_1, which skips B_{R+1}, and (2) pr^D is constructed by extending pr^B with another complete read by r_1, which skips B_{R+1} (Figure 2.4). Since r_1 cannot distinguish pr^A from pr^B, and r_1's second read skips B_{R+1} (i.e., the servers which can distinguish pr^A from pr^B), it follows that r_1 cannot distinguish pr^C from pr^D as well. Since there is no write($*$) invocation in pr^D, r_1's second read returns $\perp$ in pr^D, and hence, r_1's second read in pr^C returns $\perp$. Since pr^C is an extension of pr^A, r_R's read in pr^C returns 1. Thus, in pr^C, r_1's second read returns $\perp$ and succeeds r_R's read which returns 1. Clearly, partial run pr^C violates atomicity.

2.6 Further Results

Our lower bound result (Proposition 5) opens two directions for further investigation: multi-writer fast implementations, and semi-fast implementations, i.e., implementations where one of the operations, read or write, may not be fast. We present some results in these directions. Let W denote the number of writers in the system. The following proposition states the impossibility of fault-tolerant fast implementation, in the multi-writer case.

Proposition 6 *Let $f \geq 1$, $W \geq 2$ and $R \geq 2$. Any atomic storage implementation has a run in which some complete read or write is not fast.*

The atomicity definition extends to MWMR storage as well. In the proof of the impossibility below, we use two simple properties of MWMR atomic storage which can be easily deduced from atomicity. In any partial run (property **P1**) if a write wr that writes v, precedes some read rd, and all other writes precede wr, then if rd returns, it returns v, and (property **P2**) if there are two reads such that all writes precede both reads, then the reads do not return different values.

PROOF: It is sufficient to show the impossibility in a system where $W = R = 2$, and $t = 1$. Let the writers be w_1 and w_2, and the readers be r_1 and r_2. Let s_1 to

s_S be the servers. Suppose by contradiction that there is a fast implementation of an atomic storage in this system. To show the desired contradiction, we construct a series of runs, each consisting of two writes followed by a read.

Since the writer, any number of readers, and up to f servers might crash in our model, the invoking process can only wait for reply messages from $n - f$ servers. Given that we assume a fast implementation, on receiving a READ (or a WRITE) message, the servers cannot wait for messages from other processes, before replying to the READ (or the WRITE) message. We can thus construct partial runs of a fast implementation such that only READ (or WRITE) messages from the invoking processes to the servers, and the replies from servers to the invoking processes, are delivered in those partial runs. All other messages remain in transit. In particular, no server receives any message from other servers, and no invoking process receives any message from other invoking processes. In our proof, we only construct such partial runs.

We say that a complete invocation *inv skips* a server s_i in a partial run if every server distinct from s_i receives the READ or the WRITE message from *inv* and replies to that message, *inv* receives those replies and returns, and all other messages are in transit. In other words, only s_i does not receive READ or WRITE message from *inv*. Since $t = 1$, any complete invocation may skip at most one server. If a complete invocation does not skip any servers, we say that the invocation is *skip-free*.

Consider a partial run run_1 constructed with the following three non-overlapping invocations: (1) a skip-free write(2) by w_2, that precedes (2) a skip-free write(1) by w_1, that in turn precedes (3) a skip-free read() by r_1. From property P1, the read returns 1.

We now construct a similar partial run run_2 in which the order of the two writes are interchanged: (1) a skip-free write(1) by w_1, that precedes (2) a skip-free write(2) by w_2, that in turn precedes (3) a skip-free read() by r_1. From property P1, the read returns 2.

Consider a series of partial runs run^i, where i varies from 1 to $S + 1$. We define run^1 to be run_1. We iteratively define the remaining partial runs. We define run^{i+1} to be identical to run^i except in the following: s_i receives the WRITE message (and replies to that message) from w_1 before the message from w_2 (i.e., the replies of s_i are sent in the opposite order in r^{i+1} from that in run^1). Since servers do not receive any message from other servers in the partial runs we construct, the only server that can distinguishes run^i from run^{i+1} is s_i. Also w_1, w_2 and r_1 can distinguish the two partial runs. It is easy to see that no server can distinguish run^{S+1} from run_2, and hence, r_1 can not distinguish between the two runs as well. Thus r_1 returns 2 in run^{S+1}, and run^{S+1} and run_2 differ only at $w1$ and $w2$. Since $r1$ returns 1 in run^1, 2 in r^{S+1}, and either 1 or 2 in run^i $(2 \leq i \leq S)$, there are two partial runs, run^{i1} and run^{i1+1}, such that $1 \leq i1 \leq S$ and the read by $r1$ returns 1 in run^{i1} and returns 2 in run^{i1+1}.

Consider a partial run run' which extends run^{i1} with a read by r_2 that skips s_{i1}. From property P2, it follows that r_2 returns 1. Similarly we construct a partial run run'' which extends run^{i1+1} with a read by r_2 that skips s_{i1}. Recall that, only w_1, w_2, r_1 and s_{i1} can distinguish run^{i1} from run^{i1+1}. Since r_2 skips s_{i1} in both run' and run'', r_2 cannot distinguish the two partial runs. Thus r_2 returns

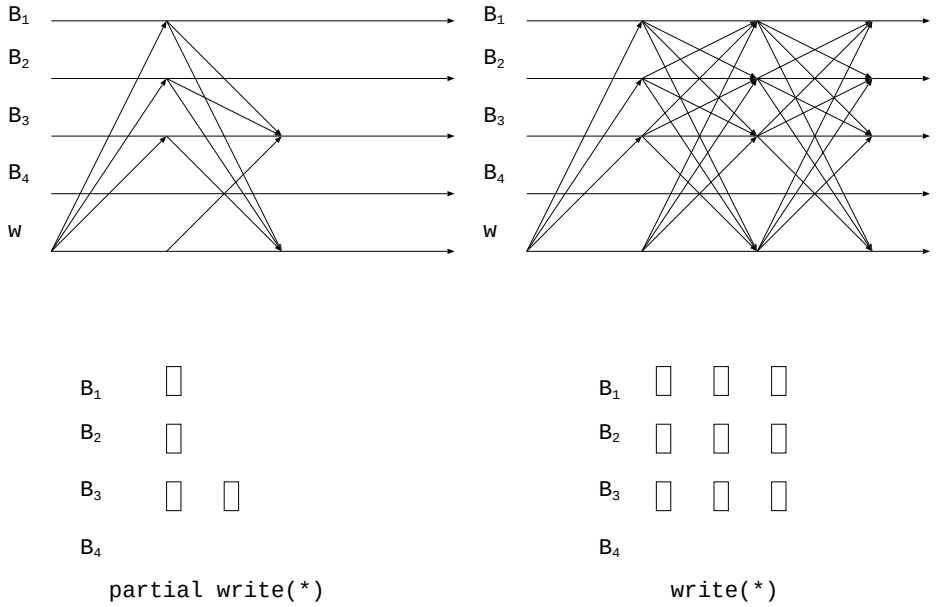

Figure 2.5: Partial writes ($K = 3, R = 4$).

1 in run''. However, r_1 returns 2 in run^{i1+1}, and hence, in returns 2 in run'' as well. Clearly, run^{i1+1} violates property P2.

To see why the above proof does not apply to the single writer case, observe that in most partial runs in the above proof, the two writes are concurrent. However, in our system model, a process can invoke at most one invocation at a time. Thus we cannot construct partial runs with concurrent writes in the single-writer case.

We say than an implementation is semi-fast if the implementation has fast reads *or* fast writes (not necessarily both). Semi-fast implementations might be interesting in systems where one of the operations (say read) is more frequent than the other. It is natural to ask, given n and f, whether we can increase the number of readers, if we allow the writes to be slow? The proposition below states that, at least in the single-writer case, we can hardly increase the number of readers if we allow slower writes.

Proposition 7 *Let $f \geq 1$, $W = 1$ and $R \geq 2$. If $R \geq n/f$ then any atomic storage implementation has a run in which some complete read is not fast.*

Proof: (*Brief sketch*) Suppose by contradiction that $R \geq n/f$ and there is an implementation I of atomic storage with fast reads. Given that $f \geq n/R$, we can partition the set of servers into R subsets (which we call *blocks*), denoted by B_i ($1 \leq i \leq R$), each of size less than or equal to f.
In all partial runs that we construct in this proof, a (partial) write is followed by a series of fast reads by distinct readers. The general style of the proof is similar to that for Proposition 5. However, in the present proof, the write may not be fast, and hence, we assume that servers exchange messages during a write. (The servers do not exchange message during reads in our partial runs, because the reads are fast.) Since we are interested in a lower bound, in the partial runs we

construct, we assume the most general communication pattern for the (non-fast) write that skips a block B_R. The writer sends messages to all servers, the writer, and the servers that are not in B_R, exchange messages among themselves K times (where K is a constant), and then, the writer returns from the invocation. Messages sent to servers in B_R, by the writer and other servers, remain in transit. We say that the message sent by the writer on invoking the write to be of *order 0*, and the messages in the k^{th} exchange of the write, to be of *order k-1*.

We draw a single column block diagram for each read, as in the proof of Proposition 5. For the write, we draw a multiple column block diagram: we draw a rectangle in row i of column k if processes in block B_i have received the messages of order $k-1$, and have sent the order k messages, in that partial run. Figure 2.5 presents block diagrams for partial writes where $R = 4$, $K = 3$ and the servers are divided into blocks B_1 to B_4.

We now apply the same pattern of reads as in the proof of Proposition 5. (Recall that the initial value of the storage is $\perp$.) Consider a partial run containing a complete write(1) that skips B_R. We append the write by a series of reads done one after the other, each read being invoked by a distinct reader, varying from r_1 to r_R. The read by r_i skips B_i. After appending each read, we delete all steps in the partial run that can be deleted such that the last reader cannot distinguish between the two partial runs (before and after deleting the steps). In other words, after appending a read by r_i that skips B_i, we delete the steps of servers in block B_i such that the deletion of steps is not "visible" to r_i. However there is an important difference from the proof of Proposition 5. We can only delete those steps of servers in B_i, that are either in the K^{th} column of write(1) or are subsequent to the K^{th} column of write(1). Since the servers exchange messages in write(1), if we delete steps in column $K-1$, the deletion will be "visible" in all steps in column K, and hence to the last reader. (See Figure 2.6). Additionally, observe that, on deleting the steps of B_{R-1} (after appending the read by r_{R-1}), we also delete the step of w in which w sent the order $K-1$ messages. To see why, notice that r_{R-1} cannot see that step of w because all steps of the servers in column K has been deleted. By atomicity, the last read in every constructed partial run returns 1.

After appending R reads, we notice that we have deleted all steps in column K of write(1) as well as, deleted all steps of the first read by r_1. (See Figure 2.6.) Thus we can again append a read by r_1 (we recycle r_1), and start deleting the steps in column $K-1$. On deleting all steps that are invisible to r_1, we notice that we have deleted all steps of the first read by r_2, and we can recycle r_2. The pattern must be obvious by now: each appended read frees up the oldest reader in the partial run, and we can use that reader for the next appended read. We repeat appending reads, until we delete all steps in write(1). (This would require repeating K times, the appending of reads by r_1 to r_R.) Thus the last partial run constructed would have no steps of write(1), but still the last read by r_R would return 1, a violation of atomicity.

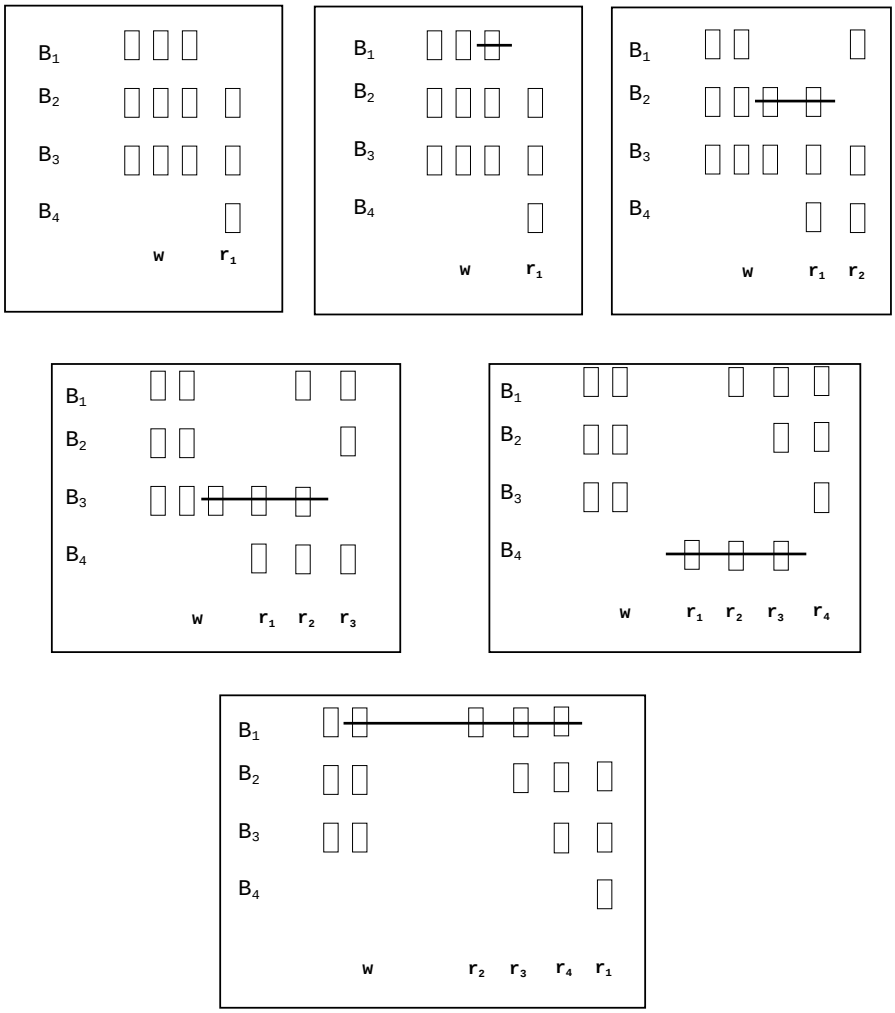

Figure 2.6: Appending reads ($K = 3, R = 4$).

3

Log-complexity

The report of my death was an exaggeration.

New York Journal, June 2, 1897
Mark Twain

3.1 Introduction

When processes recover, they lose all information present in their local volatile memory: they are in this sense *amnesic*. A process can indeed "cure" its amnesia by using stable storage, or by communicating with other processes upon recovery, but this adds a non negligible overhead that algorithms need to minimize. Maybe more fundamentally, the possibility of recovery requires to revisit some basic reasoning tools underlying the very notion of atomicity. It is thus appealing to study the actual meaning and the cost of implementing an atomic storage in a practical model where processes may crash and recover.

In an asynchronous message passing system where processes can fail by crashing and are supposed to never recover (*crash-stop* model), robust (fault-tolerant) atomic storage algorithms [Attiya et al. 1995; Lynch and Shvartsman 1997; Attiya 2000; Lynch and Shvartsman 2002] have typically assumed that, in every execution of the algorithm, a majority of the processes do not crash: *robustness* [Attiya et al. 1995] means here that any *read* or *write* operation, invoked by a process p which does not subsequently crash, eventually returns.

3.1.1 Atomicity and Histories

An atomic storage is convenient because it provides the illusion of instantaneous execution despite concurrency: every *read* or *write* operation appears to take effect at some individual instant within the time interval between the invocation and reply events of these operations. i.e. as if they were accessing every variable sequentially one after the other [Lamport 1978; Herlihy and Wing 1990].

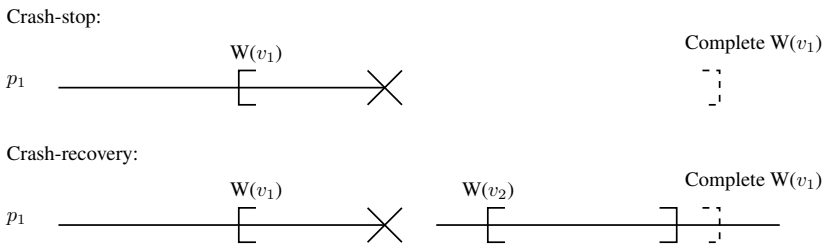

Figure 3.1: Completing *write* invocations.

A reliable atomic storage provides this illusion despite failures. Ideally, to clients, the fact that the underlying model is crash-stop or crash-recovery should not make any difference: atomicity semantics should stay intact.

Nevertheless, and as we elaborate in this thesis, the notion of *history*, key to defining atomicity [Lamport 1985], needs to be revisited in a crash-recovery model. In simple terms, a history is a sequence of invocation and reply events depicting an actual interaction between a process and a storage abstraction. Atomicity is traditionally defined with respect to *complete* histories where every invocation has a matching reply[1] [Lamport 1985]. As processes might crash, some replies can be missing: their matching invocations are in this sense incomplete. It is convenient to *complete* the histories by removing incomplete *read* invocations, and adding hypothetical missing replies to incomplete *write* invocations. In a crash-stop model, we simply append specific replies at the end of the history: in a sense, we assume that the process received the actual reply right before crashing and this is feasible because a crash event can only be the last event at a given process. In the crash-recovery model, appending replies at the end would make processes internally concurrent, as a process might have recovered in the meantime and invoked further operations: this is depicted in Figure 3.1. In this case atomicity becomes meaningless.

We propose in the thesis a specific way of completing histories which allows us to keep the traditional notion of atomicity intact from the client's perspective. Intuitively, this means that crashes and recoveries are transparent to the client. However, as we will see when establishing complexity bounds, this desirable transparency comes at a cost.

3.1.2 Quorums and Resilience

To get an idea of the ramifications underlying implementing an atomic storage over a crash-recovery message passing system, consider the storage algorithm over a crash-stop message passing system described in [Lynch and Shvartsman 1997]. (This algorithm is basically the same as the seminal single-writer algorithm of [Attiya et al. 1995] with the addition of id's for multiple writers.)

Monotonically increasing timestamps are used to order the values written in the storage: every process holds copies of the storage value, presumably the latest written value in the storage, with an associated timestamp. Implementing

[1] The goal is indeed to provide the illusion of a failure-free and sequential behavior where every invocation is immediately followed by a matching reply.

a *write* operation goes as follows. First, the writer process requests the highest timestamp from a majority of processes. The writer then increments this timestamp and broadcasts it, together with the value to be written, to all processes. Every process that receives this message updates its variable with the new value and timestamp,[2] then sends back an acknowledgment (ack) to the writer. Once the writer receives a majority of acks, it returns from the *write* operation (i.e. returns an "ok" indication). A *read* operation consists in selecting the value with the highest timestamp among a majority, and imposing it at a majority. The assumption of a correct majority ensures the robustness of the implementation. Writers and readers always access a correct process, and this ensures the persistence of the information and guarantees atomicity.

Intuitively, one would require in a crash-recovery model that such majorities always intersect at a process that is not *amnesic*, i.e. that could ensure the persistence of a written value. This can be achieved by equipping a number of processes with stable storage. Even without stable storage it is possible to ensure persistence throughout crashes. In practice, the probability that all processes fail at the same time during a given execution is usually quite small. This can be used to our advantage by making the hypothesis that a certain number of processes never crash during the duration of the execution. In this thesis, we precisely capture this notion by introducing a general notion of *amnesia masking storage*, of which we give different examples according to the underlying settings, i.e. whether processes have access to stable storage or not, and whether it is reasonable to assume that, in every execution, some of the processes do not crash.

We prove resilience lower bounds for each of those settings. More precisely, assuming that f (faulty) processes may crash permanently or keep crashing and recovering forever, we prove that in a system of n processes, implementing an atomic storage requires that there is a number $s > 2f$ of processes that have stable storage, or the number of processes u that never crash must be such that $u > f$ (in practice, it is enough that u processes are not crashed at the same time).

3.1.3 Complexity

In a crash-stop message passing model, the usual way to measure time-complexity is to count the number of inter-process communication steps needed for every *read* or *write* operation to complete (the local computation is neglected and the time to broadcast is assumed to be the same as the time to send a message to some process).

In the thesis, we introduce this notion of log-complexity and we prove a tight bound on the number of *logs* needed to implement *write* and a *read* operation of an atomic storage over a crash-recovery message passing system when stable storage is available. We also prove that the number of processes that may crash in every execution is equal to or higher than the number of faulty processes. We show that implementing an atomic storage in a crash-recovery model with stable

[2]Note that timestamps are sequence numbers (integers) associated with process ids, and these ids help order timestamps with the same sequence number.

storage requires at least 2 logs for a *write* and 1 log for a *read*. These lower bounds hold even for a single-writer/single-reader atomic storage, no matter how many messages or communication steps are used among processes.

To illustrate the issues underlying our tight bound on log complexity, consider the crash-stop storage from [Lynch and Shvartsman 1997]. In fact, by making some drastic adaptations, we could transform the storage to a crash-recovery model. We could for instance require from every process that it logs each of its updates in stable storage. The resulting algorithm would be very expensive in terms of logs (clearly not optimal). Let us discuss below the necessity of some of the underlying logs:

1. Before a *write* completes, a certain number of processes must have logged the new value and its associated timestamp so that a subsequent reader will be able to contact one of those processes. In other words, a *write* needs at least 1 log. Otherwise there might be no way for a written value to persist in the system and be eventually read (*forgotten-value*).

2. But do we need 2 logs? For instance, does the writer need to log the timestamp it associates with a value, before asking a majority of the processes to adopt the value with this timestamp? This seems desirable to prevent the case where the writer crashes and a single process adopts the new value and timestamp. Upon recovery, the writer might otherwise use the very same timestamp to write a different value, leading to two different values with the same timestamp (*confused-values*).

3. Furthermore, does the writer need to log the very fact that it is about to start writing some value v? Again, this seems desirable because, if the writer crashes during a *write* and recovers, it might start a new operation without finishing the previous *write* (*orphan-value*). We say that a *write* of value v is finished if no subsequent *read* can return a value written before v.

When not enough processes have access to stable storage, we need to assume that a sufficient number of processes do not crash at the same time. More precisely, the number of processes that do not crash should be such that $u > f$. Coming up with an algorithm in this setting that minimizes the number of communication steps is also not trivial: before a *write* completes, enough process must be aware of the new value and associated timestamp in order to ensure persistence. Processes that crash and recover must be informed of the latest value. How do we ensure that no two different values are written using the same timestamp? How does the writer "remember" that it started a *write* without logging?

3.1.4 Summary of Contributions

This thesis revisits the reasoning tools underlying atomicity in a crash-recovery model and gives a generic algorithm that implements a multi-writer/multi-reader atomic storage in a crash-recovery message passing model. Our algorithm is

generic in the sense that it uses an abstract notion of *amnesia masking storage* which can be instantiated for several kinds of crash-recovery systems according to whether or not processes have access to stable storage and whether we can assume that a subset of processes do not crash in every execution. Considering a system with n processes, including s processes with stable storage, a maximum of f faulty processes that can crash permanently or keep crashing and recovering forever, and u processes that do not crash, we establish the optimality of specific instances of our algorithm by proving the following bounds:

1. *Resilience*: $f < n/2$ and $u > f$ if $s \leq 2f$.
2. *Log-complexity*: If $s > 2f$ and $u \leq f$, 2 logs per *write* and 1 per *read* are necessary for a single writer/single reader and sufficient for a multi reader/multi writer storage algorithm.
3. *Time-complexity*: If $s = 0$, more than 1 round trip per *write* is necessary for a single writer and multi reader storage algorithm[3]. If $s \neq 0$ then 1 round-trip per *write* is sufficient for a single writer storage algorithm.

We also discuss the impact on these results of weakening the semantics of the distributed storage. In particular, we discuss safety and regularity as two alternatives to atomicity [Lamport 1985].

3.1.5 Road-Map

Section 3.2 describes the basic elements of a crash-recovery model. Section 3.3 revisits the essential tools needed to reason about atomicity in that model. Section 3.4 defines the notion of amnesia masking storage. Section 3.5 presents our generic storage algorithm based on that notion. Section 3.6 proves bounds on the resilience, log- and time-complexity of an atomic storage and derives the optimality of specific instances of our generic storage algorithm. Finally, Section 3.7 revisits our assumptions, discusses the impact of weakening the storage abstraction and looks at the performance of different instances of our storage.

3.2 Model

We consider an asynchronous message passing model, without any assumptions on communication delay or relative process speeds. To simplify the presentation of our algorithms, we assume the existence of a global clock. This clock however is a fictional device outside of the control of the processes.

The set of processes N, $|N| = n$, is static and every process executes a deterministic algorithm assigned to it, unless it *crashes*. The process does not behave maliciously. If it crashes, the process simply stops executing any computation, unless it possibly *recovers*, in which case the process executes a *recovery procedure* which is part of the algorithm assigned to it. Note that in this case we assume that the process is aware that it had crashed and recovered.

[3]The time-complexity of a *read* can be derived from existing results in crash-stop model [Attiya et al. 1995; Dutta et al. 2004].

Every process has a volatile storage and some processes may also have a stable storage. If a process crashes and recovers, the content of its volatile storage is lost but not the content of its stable storage. after recovering from a crash are its id and the value of its local clock. Each process has a local clock which provides an interface for retrieving a timestamp. The clock guarantees that each timestamp is unique and that the sequence of timestamps are monotonically increasing despite crashes and recoveries.[4]

By default, whenever a process updates one of its variables, it does so in its volatile storage. The process can decide to store information in its stable storage (if it has one) using a specific primitive store: we also say that the process *logs* the information. The process retrieves the information logged using the primitive retrieve. The processes with stable storage belong to a set denoted S, $S \subseteq N$. There are a total number of $0 \leq |S| = s \leq n$ processes with stable storage.

Whereas the sets N and S are static for all executions, the sets of processes that we will define now are not: they might be different for each execution (and unknown in advance). These sets are defined for an infinite execution, i.e. the sets can only be evaluated by an external observer of the system looking at infinite executions. The sets are defined in the same way as in [Aguilera et al. 1998]. Processes that crash at least once, always recover after a crash and eventually do not crash are *eventually-up* and belong to a set denoted C, $|C| = c$. These might crash (and recover) a large (but finite) number of times. A process is *faulty* (the process belongs to a set denoted F, $|F| = f$) if there is a time after which the process crashes and never recovers or it crashes infinitely many times. We also consider a set of processes that are *always-up* U, $|U| = u$ (in that given execution). Considering $n = c + f + u$, a number $c + f$ processes can crash and c eventually recover.

We assume underlying fair-lossy channels [Lynch 1996], which are defined as follows: if a process p_i sends a message m to a non-faulty process p_j an infinite number of times, then p_j receives m an infinite number of times. Furthermore, if a process p_j receives some message m, then some process p_i has sent m. On top of these channels we can easily implement more useful stubborn communication procedures which are used to send and receive messages reliably [Boichat and Guerraoui 2005]. More precisely, if a process p_i calls the procedure s-send to send a message m to a process p_j, then p_j will eventually s-receive m if p_i and p_j are non-faulty. We assume in this thesis that processes communicate using s-send and s-receive.

Any *read* or *write* invocation of the storage is translated into messages exchanged between processes. We say that an operation (*read* or *write*) invoked by a process p *contacts* a set of processes Q if, in the algorithm implementing the operation in our crash-recovery message passing model, p sends messages to at least $|Q|$ processes after the invocation of the operation and subsequently receives $|Q|$ causally dependent [Lamport 1978] responses, from $|Q|$ different processes, before returning from the operation.

[4] Almost all modern computers are equipped with a battery powered clock that keeps time even when the computer is switched off.

3.3 Atomicity

Roughly speaking, atomicity provides the illusion that the storage appears to be accessed in a failure-free and sequential way. We consider robust storage where a process that invokes a *read* or *write* operation, and does not crash after that invocation, eventually returns from the operation [Attiya et al. 1995; Herlihy 1991].

In the following section, we extend the traditional tools used to reason about the notion of atomic storage in the crash-stop model to encompass the crash-recovery model. Ideally, to the user of an atomic storage, it should make no difference if the model is crash-stop or crash-recovery. Formally however, and as we discuss below, we need to reason about histories to define atomicity and the concept of a history needs to be revisited in a crash-recovery model.

We first introduce basic elements underlying this concept. A *history* is a total order of events of four kinds: *invocations*, *replies*, *crashes* and *recoveries*. Every such event is modeled to take place at a given time of the global clock, and no two events are supposed to take place at the same time. Every invocation and every reply is associated with exactly one process and one object. A reply is said to *match* an invocation if both are associated with the same process and the same object: such a pair defines an operation execution (sometimes we simply say operation when there is no ambiguity). Invocations and replies are called *object events*. In our context, operations are either *read* or *write*. An invocation with no matching reply in a history is said to be *pending* in that history. An operation op is said to *precede* an operation op' in a history if the reply of op precedes the invocation of op' in that history. An operation op is said to *immediately precede* an operation op' in a history if the reply of op precedes the invocation of op' in that history and such that no operation op'' precedes op' where op precedes op''.

A *local history* is a sequence of events associated with the same process. A local history is said to be *well-formed* if: (a) its first event is either an invocation or a crash, (b) a crash is followed by a matching recovery event or is not followed by any event, and (c) an invocation is followed by a crash or a reply event. A history is said to be well-formed if all its local histories are well-formed. Two histories H and H' are said to be *equivalent* if, for every process p, the local history H restricted to p has the same object events as the local history H' restricted to p.

To define atomicity, we reason about histories that are *complete*. These are histories without any crash or recovery events where every invocation has a matching reply. In a crash-stop model, pending invocations are completed by appending a matching reply at the end of the history [Herlihy 1991]. In a crash-recovery model however, a pending invocation can be followed immediately by another invocation (i.e. if the process has recovered in the meantime), thus the need for changing the way histories are completed. In our crash-recovery model, given any well-formed history H_1, we say that H_2 *completes* H_1 if H_2 does not contain any crash or recovery events and is made of the very same object events in the same order as in H_1, with one exception: any pending invocation in H_1 is either absent in H_2, or has a matching reply that appears in H_2 before the subsequent invocation of the same process. A *completed operation* has a pending invocation in H_1 that has a matching reply that appears in H_2 before the subsequent invo-

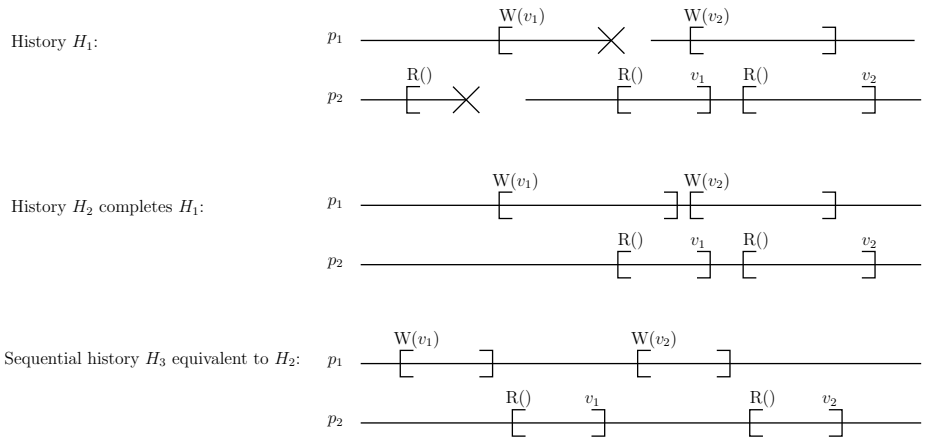

Figure 3.2: Completing and sequentializing a history.

cation of the same process. A history is said to be *sequential* if it is complete and every invocation is followed by a matching reply.

Every shared object has a sequential specification, defined by a set of sequential histories involving only events associated with that object. Roughly speaking, the sequential specification captures the acceptable behavior of the object in the absence of concurrency and failures. In our context, we are concerned with storage objects whose sequential specifications simply stipulate that a *read* returns the last written value. A sequential history is said to be *legal* if each of its restrictions to any object involved in the history belongs to the sequential specification of that object. A history H is said to be *atomic* if it can be *completed* to a history that is equivalent to some legal sequential history. An example is shown in Figure 3.2. We say that an algorithm implements an atomic storage if every history generated by the algorithm is atomic.

Our definition of atomicity ensures that all operations are linearized and that the linearization point is always in between the operation invocation and the response.

3.4 Amnesia Masking Storage

In this section we define the amnesia masking storage abstraction (AMS). Our generic atomic shared memory emulation algorithm presented in the subsequent section builds on this abstraction.

3.4.1 Properties

Our storage abstraction is shared by all processes and exports two procedures: WriteAMS(v, ts) that takes as input a value-timestamp pair and simply returns "ok" upon completion, and ReadAMS() that returns a set of value-timestamp pairs V. The notation X.WriteAMS(v, ts) and X.ReadAMS() means that the amnesia masking storage instance X is accessed. The WriteAMS() and ReadAMS() procedures satisfy the following properties:

- **Property P_1:**

  consider a set of value-timestamp pairs $[v_i, ts_i]$, each value v_i being associated with a timestamp ts_i. If ReadAMS() successfully completes and returns a set V of value-timestamp pairs, then V includes the value-timestamp pair $[v_h, ts_h]$, where ts_h is the highest timestamp among all WriteAMS(v_i, ts_i) invocations that successfully completed *before* the ReadAMS() invocation.

- **Property P_2:** when a process invokes a WriteAMS() or ReadAMS() procedure, a matching reply is eventually returned unless the invoking process crashes.

Intuitively, to satisfy property P_1, any implementation of the storage abstraction must ensure that the information v_i is stored at enough processes such that it will persist through crashes. If every WriteAMS() and ReadAMS() invocation contacts a set of processes that overlap at least one process, which is furthermore not amnesic, we can satisfy P_1. This means that every WriteAMS() invocation must contact either enough processes with stable storage or enough processes that do not crash. We denote the set of processes contacted by a WriteAMS() invocation by Q_W and the set of processes contacted by ReadAMS() by Q_R.

Figures 3.3 and 3.4 describe WriteAMS() and ReadAMS() implementations. We give two implementations, they both share the top level Initialize, WriteAMS() and ReadAMS() procedures. However, they have separate low level subroutines for processes with stable storage and without. During an execution, some process might have access to stable storage and others not, therefore each process executes the appropriate subroutine depending on whether it has access to stable storage or not.

The basic idea is to write a value by sending the value-timestamp to all processes and waiting for a number of replies that is equal to $|Q_W|$. Upon receiving such a WriteAMS() message, the other processes locally store the value-timestamp pair (in stable storage or volatile memory) if the received timestamp is higher than the one currently stored. During the recovery phase, the value-timestamp pairs are retrieved from stable storage. When a process without stable storage recovers, it sets its amnesic variable to true, which means that the process does not reply to any ReadAMS() request until after it has stored a new value-timestamp using WriteAMS().

We make the following assumptions on Q_W and Q_R:

1. $|Q_W| > n - u$ or if $s > 2f$ then $|Q_W| > n - s + f$
2. $|Q_W| \leq n - f$
3. $|Q_R| > 0$
4. $|Q_R| \leq n - f - c$ or if $s > 2f$ then $|Q_R| \leq s - f$
5. $|Q_W| + |Q_R| > n$

We will show in Section 3.6 that these bounds are tight.

```
1: procedure Initialize
2:    ts := 0, v :=⊥, lts := 0
3:    amnesic := false      {Variable is set to true if a process loses the contents of its volatile
      memory by crashing.}
4: end

5: function WriteAMS([v, ts]) at p_w
6:    s-send(W, [v, ts]) to all processes
7:    wait until received (W.ACK, [v, ts]) from |Q_W| processes
8: return  ok

9: function ReadAMS() at p_r
10:   lts := getTime()                                   {returns the local time}
11:   s-send(R,lts) to all processes
12:   wait until received (R.ACK, [v, ts], p_i, lts_i) from |Q_R| processes where p_i = p_r ∧ lts_i =
      lts
13:   V ← {m|p_r received (R.ACK, [v, ts])}
14: return  V                          {return the set of all received value timestamp pairs}
```

Figure 3.3: Amnesia masking storage procedures.

Lemma 1 *Q_W always contains at least one eventually-up process with stable storage or one process that is always-up if $|Q_W| > n - u$ or $|Q_W| > n - s + f$ when $s > 2f$.*

Proof: There are two cases to consider:

- $s \leq 2f$. In this case $|Q_W| > n-u$ and thus Q_W will always contain at least one process that is always-up.
- $s > 2f$. In this case $|Q_W| > n - s + f$. At worst we have $u = 0$ and therefore Q_W can at worst contain $n - s$ eventually-up processes without stable storage and f faulty processes with stable storage. The remaining processes in Q_W are therefore eventually-up with stable storage.

□

Lemma 2 *Q_W and Q_R always overlap in at least one eventually-up process with stable storage or one process that is always-up.*

Proof: Because of Lemma 1, Q_W always contains at least one eventually-up process with stable storage or one process that is always-up. In our implementation, processes without stable storage that crash do not reply to read requests (Figure 3.4, second box, line 9) and since by assumption Q_W and Q_R overlap in at least one process and $|Q_R| > 0$ the lemma is satisfied. □

Lemma 3 *The wait statement of line 7 eventually ends if $|Q_W| \leq n - f$.*

Proof: All processes in the implementation of Figures 3.3 and 3.4 reply when contacted and there are no wait statements (second and third box). Since $|Q_W| \leq$

Reception and recovery with stable storage:

```
 1: upon s-receive(W, [v', ts']) from p_w do
 2:    if ts' > ts then
 3:       [v, ts] := [v', ts']
 4:       store([v, ts])
 5:    end if
 6:    s-send(W.ACK, [v', ts']) to p_w
 7: end upon

 8: upon s-receive(R,lts) from p_i do
 9:    s-send(R.ACK, [v, ts], p_i, lts) to p_i
10: end upon

11: procedure Recovery
12:    [v, ts] := retrieve()
13: end
```

Reception and recovery without stable storage:

```
 1: upon s-receive(W, [v', ts']) from p_w do
 2:    if ts' > ts then
 3:       [v, ts] := [v', ts']
 4:       amnesic := false
 5:    end if
 6:    s-send(W.ACK, [v', ts']) to p_w
 7: end upon

 8: upon s-receive(R,lts) from p_i do
 9:    if ¬amnesic then
10:       s-send(R.ACK, [v, ts], p_i, lts) to p_i
11:    end if
12: end upon

13: procedure Recovery
14:    amnesic := true
15:    ts := 0
16:    v :=⊥
17: end
```

Figure 3.4: Amnesia masking storage reception and recovery procedures.

$n - f$ all eventually-up processes that crash eventually recover (by hypothesis, see Section 3.2) and all processes in Q_W eventually reply when contacted. □

Lemma 4 *The wait statements of line 12 eventually ends if* $|Q_R| \leq u$ *or* $|Q_R| \leq s - f$ *when* $s > 2f$.

PROOF: All processes that do not have their amnesic variable to true in the implementation of Figures 3.3 and 3.3 reply when contacted and there are no wait statements (second and third box). There are two cases to consider:

- If $|Q_R| \leq u$ at least u always-up processes in Q_R eventually reply when contacted.
- If $|Q_R| \leq s - f$ when $s > 2f$ then there are at least $s - f$ eventually-up processes with stable storage in Q_R that eventually reply when contacted.

□

We now show that the implementation in Figures 3.3 and 3.4 satisfy properties P_1 and P_2.

Lemma 5 *The amnesia masking storage implementation in Figures 3.3 and 3.4 satisfy property* P_1.

PROOF: We must show that in any execution, given a sequence of k complete WriteAMS(v_i, ts_i) invocations (with $1 \leq i \leq k$), any ReadAMS() that comes after the response of WriteAMS(v_k, ts_k) returns $\{[v_h, ts_h] \in V | ts_h = \max(ts) \forall ts \in V\}$. When WriteAMS($v_i, ts_i$) is invoked, $[v_i, ts_i]$ is sent to all processes (Figure 3.3, line 6) and acknowledged by $|Q_W|$ processes (first box, line 7). The processes that receive this value-timestamp pair store it only if the timestamp is higher than the currently stored timestamp. The processes with stable storage log the value. This mechanism ensures that after a sequence of k complete WriteAMS(v_i, ts_i) invocations, at least $|Q_W|$ processes have the timestamp-value pair with the highest timestamp stored locally. Because of Lemma 1, at least one process in Q_W will preserve this information indefinitely. The ReadAMS() that comes after the sequence of k complete WriteAMS(v_i, ts_i) invocations reads the value-timestamp pairs from $|Q_R|$ processes (first box, line 12). Because each ReadAMS() request is uniquely identified by a local timestamp and the id of the reader, no "old messages" can be returned by such a request. Furthermore, because of Lemma 2 we know that Q_W and Q_R always overlap in at least one eventually-up process with stable storage or one process that is always-up. It will therefore return a set of value-timestamp pairs V which includes $[v_h, ts_h]$, where ts_h is the highest timestamp among the k complete WriteAMS(v_i, ts_i) invocations. □

Lemma 6 *The amnesia masking storage implementation in Figures 3.3 and 3.4 satisfies property* P_2.

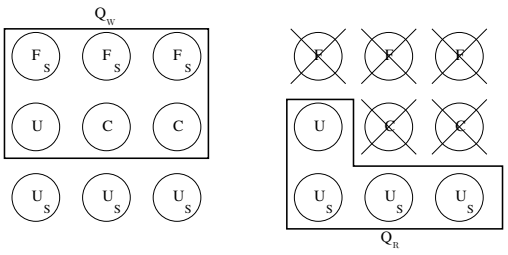

Figure 3.5: Example of amnesia masking storage configurations.

PROOF: The implementation for the WriteAMS() procedure (Figure 3.3) sends write messages to all processes and waits for $|Q_W|$ responses, therefore contacting all processes in set Q_W. The ReadAMS() procedure (Figure 3.3) sends read messages to all processes and waits for $|Q_R|$ responses, therefore contacting all processes in set Q_R. The implementation contains no other wait statements or loops and due to Lemmas 3 and 4 it satisfies property P_2. □

3.4.2 Instantiation Examples

The implementations of Figures 3.3 and 3.4 are very general and many different instantiations are possible, some of which we illustrate here. For all examples below, we consider a system with 9 processes ($n = 9$) and up to 3 faulty processes ($f = 3$).

- If we take $u = n - f = 6$ (processes that do not crash), $s = 0$ (no stable storage) and $|Q_W| = |Q_R| = 6$, we revert to a normal crash-stop model [Attiya et al. 1995].

- With $s = 0$ and $u = 4$, we have $|Q_W| = 6$ and $|Q_R| = 4$. The advantage of this setup is that despite possible crashes and recoveries, no stable storage accesses are needed at all.

- On the other hand, if we consider a system where all processes can potentially crash ($u = 0$), it is possible to implement the amnesia masking storage when all processes have stable storage: $s = 9$ and $|Q_W| = |Q_R| = 5$ for instance.

- Illustrated in Figure 3.5 is the case where not all processes have access to stable storage ($s = 6$) and four processes do not crash ($u = 4$). The size of Q_W must be bounded by: $n - u < |Q_W| \leq n - f \iff 9 - 4 < |Q_W| \leq 9 - 3 \iff 5 < |Q_W| \leq 6 \iff |Q_W| = 6$. Since Q_R must overlap with Q_W we can take $|Q_R| = 4$. By looking at the left drawing of Figure 3.5, it is easy to see why u must be bigger than 3: Q_W contains the top six processes, the top three can crash permanently and the middle three have no stable storage. In Section 3.6, we prove a lower bound on the minimum number of processes that must not crash.

3.5 The Generic Emulation Algorithm

3.5.1 Description

We now describe an algorithm that implements an atomic storage, i.e. that implements the Read() and Write() operations of our atomic storage. For clarity of presentation, we first focus on the single-writer/multi-reader version.

Our algorithm, given in Figure 3.7 is generic in the sense that it relies on the *amnesia masking storage* abstraction, defined in Section 3.4. The advantage of using this abstraction is that, on the surface, our atomic storage algorithm looks similar to the one used for the crash-stop model [Lynch and Shvartsman 1997]. The technical issues that are related to the crash-recovery model are encapsulated within specific PreRead(), PreWrite() and Recovery() procedures that we discuss later.

In Figure 3.7, the writer labels values with timestamps. In a first phase, value-timestamp pair is pre-written by the PreWrite() procedure and in a second phase they are stored using the WriteAMS() procedure. Two different amnesia masking storage instances are used in the implementation: one for the prewrites (and prereads) and one for the writes (and reads).

The Read() implementation is also divided in two phases: a first phase, which invokes ReadAMS() to obtain value-timestamp pairs, and a second phase, where the reader writes back the value with the highest timestamp collected in the previous phase by invoking WriteAMS().

We describe below the technicalities related to the crash-recovery model and highlight the differences between our algorithm and a crash-stop algorithm [Lynch and Shvartsman 1997]:

1. A PreWrite() procedure is invoked before the WriteAMS() call. The goal of the PreWrite() is to enable the writer to "*remember*" to finish the Write() upon recovery in case it crashed. We describe two implementations of PreWrite() depending on whether the writer has stable storage or not. If stable storage is used (Figure 3.8), the PreWrite() requires a single log; without stable storage (Figure 3.9), a WriteAMS() invocation is needed.

2. A PreRead() procedure is invoked during the Recovery() procedure. The PreRead() returns the latest *prewritten* value. If stable storage is used (Figure 3.8), the PreRead() reads from stable storage, otherwise (Figure 3.9) a ReadAMS() invocation is needed.

3. A timestamp mechanism provides monotonically increasing values despite writer crashes: before each new Write(), a local variable ts_w is incremented and used as the new timestamp. During the Recovery() procedure, the writer must ensure that ts_w is at least as great as it was before the crash. If stable storage is available to the writer, the latest timestamp stored by the last PreWrite() can be retrieved by calling a PreRead().

   Without stable storage at the writer, the mechanism is more complicated: the writer cannot be sure that the values returned by a PreRead() contain the latest value stored by a previous PreWrite() that was interrupted by a

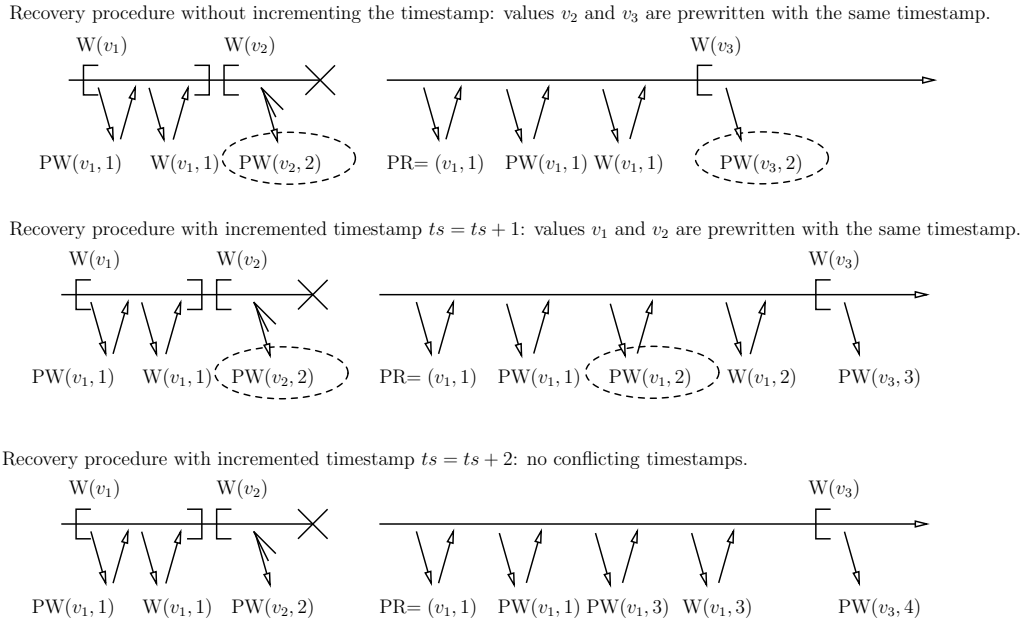

Figure 3.6: Illustration of why the timestamp needs to be incremented by 2 during the recovery phase. In all three runs W(v_2) is not complete and the prewrite messages are not received by all processes. Legend: W = write, PW = prewrite, PR = preread.

crash. If the writer would start a new Write() with the same timestamp, two different PreWrite() values would have the same timestamp. This is problematic, because when the writer crashes and recovers again, it cannot know which value to complete. The way we solve this problem is as follows. The value returned by the PreRead() is first stored by a PreWrite(), then incremented by 2 and then stored again by a PreWrite(). The first PreWrite() ensures that the next PreRead() will always return a value with a timestamp at least equal to the one returned by the current PreRead(). The reason for incrementing the timestamp by 2 is best illustrated by an example. Imagine that the timestamp at the writer is 0. The writer starts a Write(v) and increments the timestamp by one. Then the writer crashes while PreWrite($[v, 1]$) is being invoked. Even though few processes are aware of the timestamp 1, when the writer recovers the PreRead() only returns 0 as the highest timestamp. By incrementing 0 by 2 the algorithm ensures that the timestamp is higher than 1. The incremented timestamp is then again stored by a PreWrite() to ensure that any consecutive PreRead() will return at least 2 as the highest timestamp. Figure 3.6 illustrates with another example why the timestamp needs to be incremented by 2. It contains three runs: in the first run the timestamp is not incremented, in the second it is incremented by one and in the last it is incremented by 2.

```
 1: procedure Initialize
 2:    if writer then
 3:       ts_w := 0
 4:    end if
 5: end

 6: function Write(v) at p_w
 7:    ts_w := ts_w + 1
 8:    PreWrite([v, ts_w])
 9:    W.WriteAMS([v, ts_w])
10: return

11: function Read() at p_i
12:    V := W.ReadAMS()                    {V is the set of value timestamp pairs}
13:    [v_h, ts_h] := highest(V)           {"highest(V)" returns the value timestamp pair with the
       highest timestamp in V}
14:    W.WriteAMS([v_h, ts_h])
15: return   v_h
```

Figure 3.7: Generic single-writer/multi-reader atomic storage algorithm.

```
 1: function PreWrite([v, ts])
 2:    preW.store([v, ts])
 3: return

 4: function PreRead()
 5:    [v_r, ts_r] = preW.retrieve()
 6: return   [v_r, ts_r]

 7: procedure Recovery()
 8:    if writer then
 9:       [v_r, ts_r] := PreRead()
10:       W.WriteAMS([v_r, ts_r])
11:       ts_w := ts_r
12:    end if
13: end
```

Figure 3.8: Configuration with stable storage.

```
 1: procedure PreWrite([v, ts])
 2:    preW.WriteAMS([v, ts])
 3: end

 4: function PreRead()
 5:    V:=preW.ReadAMS()
 6:    [v_r, ts_r] = highest(V)
 7: return   [v_r, ts_r]

 8: procedure Recovery
 9:    if writer then
10:       [v_r, ts_r] := PreRead()
11:       PreWrite([v_r, ts_r])
12:       ts_r := ts_r + 2
13:       PreWrite([v_r, ts_r])
14:       W.WriteAMS([v_r, ts_r])
15:    end if
16: end
```

Figure 3.9: Configuration without stable storage.

3.5.2 Correctness

We now address the correctness of our atomic storage. In short, we use Lemma 13.16 of [Lynch 1996] to prove atomicity (remember that we consider the single-writer/multi-reader case: the multi-writer case is discussed in Section 3.5.3) as well as the properties (P_1 and P_2) of our amnesia masking storage abstraction. The lemma is as follows:

Lemma 13.16 *Let β be a (finite or infinite) sequence of actions of a read/write atomic object external interface. Suppose that β is well-formed for each i, and contains no incomplete operations. Let Π be the set of all operations in β.*

Suppose that $\prec$ is an irreflexive partial ordering of all the operations in Π, satisfying the following properties:

1. *For any operation $\pi \in \Pi$, there are only finitely many operations ϕ such that $\phi \prec \pi$.*
2. *If the response event for π precedes the invocation event for ϕ in β, then it cannot be the case that $\phi \prec \pi$.*
3. *If π is a WRITE operation in Π and ϕ is any operation in Π, then either $\pi \prec \phi$ or $\phi \prec \pi$.*
4. *The value returned by each READ operation is the value written by the last preceding WRITE operation according to $\prec$ (or v_0, if there is no such WRITE).*

Then β satisfies the atomicity property.

For a well-formed history H, the lemma lists four conditions involving a partial order on operations in H. It states that if there is a partial order relation on events

satisfying these four conditions then the atomicity property is satisfied. Although the lemma has been proven correct in the crash-stop model, it directly applies to the crash-recovery model if we only consider well-formed and complete histories (thus in fact abstracting crashes and recoveries away). For this proof, we assume that all histories generated by $\mathcal{A}$ are well-formed and complete and we later prove that this is actually the case (Lemma 9).

Assume that H is a well-formed and complete history. Let O be the set of operations in H, and τ be the timestamp associated with the value written or returned by each operation. We define the partial order $PO = \langle O, \prec \rangle$ on the operations by letting: $op_1 \prec op_2$ for $op_1, op_2 \in O$, if (a) $\tau(op_1) <_{lex} \tau(op_2)$, or if (b) op_1 is a Write(), op_2 is a Read(), and $\tau(op_1) =_{lex} \tau(op_2)$.

We give three lemmas that are sufficient to show that PO satisfies the required properties of Lemma 13.16.

Lemma 7 *If op_1 precedes op_2, then*
(i) if op_2 is a Read(), then $\tau(op_1) \leq_{lex} \tau(op_2)$, and
(ii) if op_2 is a Write(), then $\tau(op_1) <_{lex} \tau(op_2)$.

PROOF:
⟨1⟩1. if op_2 is a Read(), then $\tau(op_1) \leq_{lex} \tau(op_2)$
  ⟨2⟩1. True when op_1 is a Write()
    PROOF: op_2 is a Read(), therefore $\tau(op_2)$ is obtained by the reader by gathering timestamps from a ReadAMS() invocation and computing the maximum timestamp. The algorithm ensures that the value together with $\tau(op_1)$ has been stored using WriteAMS() before returning. Because of property P_1, $\tau(op_1) \leq_{lex} \tau(op_2)$.
  ⟨2⟩2. True when op_1 is a Read()
    PROOF: The algorithm ensures that the value that is returned by the Read() has been stored using the WriteAMS() procedure during the second round of the Read(), this implies $\tau(op_1) \leq_{lex} \tau(op_2)$.
  ⟨2⟩3. Q.E.D.
⟨1⟩2. op_2 is a Write(), then $\tau(op_1) <_{lex} \tau(op_2)$.
  ⟨2⟩1. True when op_1 is a Write()
    PROOF: $\tau(op_1)$ is stored using WriteAMS(). Since in a subsequent Write() the writer process obtains $\tau(op_2)$ by incrementing the previous timestamp by one, we have $\tau(op_1) <_{lex} \tau(op_2)$.
  ⟨2⟩2. True when op_1 is a Read()
    PROOF: No value smaller than $\tau(op_1)$ has been written by WriteAMS(). Because the writer increments the timestamp before sending it to all other processes, we have $\tau(op_1) <_{lex} \tau(op_2)$.
⟨1⟩3. Q.E.D.

Lemma 8 *For a Read() operation op, let the PO imposed on H give the set of Write() operations $\{op_1, op_2, ..., op_k\}$ such that $\forall i \in [1, k] : op_i \prec op$. Then op returns the value written by op_j such that $\tau(op_j) =_{lex} max_{i \in [1,k]}(\tau(op_i))$.*

PROOF: Every Write() op_j stores the value-timestamp pair using WriteAMS(). Any consecutive Read() op invokes ReadAMS and therefore receives at least one timestamp from a process written by WriteAMS(). Because of Lemma 7 we know that the timestamps impose a partial ordering on the writes such that the last Write() according to $\prec$ has the highest timestamp. Therefore the Read() op returns the value written by op_j such that $\tau(op_j) =_{lex} max_{i\in[1,k]}(\tau(op_i))$. □

Lemma 9 *The set of possible histories $\mathcal{H}$ generated by $\mathcal{A}$ are well-formed and complete.*

ASSUME: Amnesia masking storage property P_1 and P_2 are satisfied.
⟨1⟩1. $\mathcal{H}$ is *well-formed*.
⟨2⟩1. The first event is either an invocation or a crash.
PROOF: A process can only start by invoking a Read() or Write() event.
⟨2⟩2. A crash can only be followed by a matching recovery event.
PROOF: The system model states that a crashed process cannot perform any operations.
⟨2⟩3. An invocation can only be followed by a crash or a reply event.
PROOF: The algorithm only allows the execution of one operation at the same time.
⟨2⟩4. Q.E.D.
PROOF: By definition.
⟨1⟩2. Every history in $\mathcal{H}$ can be *completed*.
PROVE: Every incomplete $\mathsf{Write}_{IC}(v_n)$ is completed before the start of the next $\mathsf{Write}(v_{n+1})$ or $\mathsf{Write}_{IC}(v_n)$ is removed from $\mathcal{H}$.
⟨2⟩1. If $\mathsf{Write}_{IC}(v_n)$ is completed, it will be completed before the start of $\mathsf{Write}(v_{n+1})$ or $\mathsf{Write}_{IC}(v_n)$ will never be completed.
⟨3⟩1. A recovery procedure is executed after $\mathsf{Write}_{IC}(v_n)$, before the start of $\mathsf{Write}(v_{n+1})$.
PROOF: Upon recovery all operations are delayed until the end of the recovery procedure.
⟨3⟩2. Upon recovery a PreRead() is executed that returns the last prewritten value.
⟨4⟩1. The values written by PreWrite() are totally ordered by their associated timestamp ts.
⟨5⟩1. Each value written by PreWrite() has an associated timestamp.
PROOF: Line 2 of Figure 3.8 with stable storage and line 2 of Figure 3.9 without.
⟨5⟩2. The timestamps associated with the values written by PreWrite() increase monotonically.
⟨6⟩1. True when the writer uses stable storage.
PROOF: The timestamp is stored locally at line 2 of Figure 3.8 during the PreWrite() (before actually writing the value). Upon recovery this value is restored (line 5 of Figure 3.8) and incremented by one (Line 7 Figure 3.7). Hence, timestamps increase monotonically.

⟨6⟩2. True when the writer does not use stable storage.
PROVE: $ts_1 < \ldots < ts_n$ for all $n \geq 2$, where $ts_1, \ldots, ts_n$ are the timestamps associated with values $v_1, \ldots, v_n$, each written consecutively by a $\mathsf{PreWrite}_k([v_k, ts_k])$.
⟨7⟩1. True for $n = 1$.
PROVE: $ts_1 < ts_2$.
⟨8⟩1. If no crash occurred between $\mathsf{PreWrite}_1([v_1, ts_1])$ and $\mathsf{PreWrite}_2([v_2, ts_2])$, then $ts_1 < ts_2$.
PROOF:
The local variable ts is incremented before each PreWrite() (line 7 of Figure 3.7), since no crashes occurred, $ts_2 = ts_1 + 1$ and therefore $ts_1 < ts_2$.
⟨8⟩2. If one or more crashes occurred between $\mathsf{PreWrite}_1([v_1, ts_1])$ (which might be incomplete) and $\mathsf{PreWrite}_2([v_2, ts_2])$, then $ts_1 < ts_2$.
⟨9⟩1. Initially, the local timestamp ts is equal to 0.
PROOF: Line 3 of Figure 3.7.
⟨9⟩2. $ts_1 = 1$.
PROOF: Step ⟨9⟩1 and line 7 of Figure 3.7.
⟨9⟩3. $ts_2 \geq 2$.
⟨10⟩1. Before the invocation of $\mathsf{PreWrite}_2([v_2, ts_2])$, at least one recovery procedure is executed.
PROOF:
There is at least one crash in between $\mathsf{PreWrite}_1([v_1, ts_1])$ and $\mathsf{PreWrite}_2([v_2, ts_2])$, the system model ensures that a recovery procedure is executed upon recovery before the start of the next operation, and the PreWrite() is at the beginning of a Write() operation.
⟨10⟩2. During this recovery procedure, a PreRead() is executed which returns the highest timestamp from a ReadAMS() invocation.
PROOF: The PreRead() is executed at line 10 of Figure 3.9. The ReadAMS() procedure is invoked on line 5 and the highest value is selected on line 6.
⟨10⟩3. The lowest timestamp which can be read by the PreRead() is 0.
PROOF: Step ⟨9⟩1.
⟨10⟩4. During the recovery procedure, the highest timestamp read from the processes is incremented by 2.
PROOF: Line 12 of Figure 3.9.
⟨10⟩5. Q.E.D.
⟨9⟩4. Q.E.D.
⟨8⟩3. Q.E.D.
ASSUME: True for $n = l$.
⟨7⟩2. True for $n = l + 1$.
PROVE: By assumption $ts_1 < \ldots < ts_l$, we must therefore prove that $ts_l < ts_{l+1}$.

⟨8⟩1. If no crash occurred in between $\mathsf{PreWrite}_l(v_l)$ and $\mathsf{PreWrite}_{l+1}(v_{l+1})$, then $ts_l < ts_{l+1}$.

PROOF:

The local variable ts is incremented before each PreWrite() (line 7 of Figure 3.7), since no crashes occurred, $ts_{l+1} = ts_l + 1$ and therefore $ts_l < ts_{l+1}$.

⟨8⟩2. If one or more crashes occur in between $\mathsf{PreWrite}_l(v_l)$ (which might be incomplete) and $\mathsf{PreWrite}_{l+1}(v_{l+1})$, then $ts_l < ts_{l+1}$.

⟨9⟩1. Before the invocation of $\mathsf{PreWrite}_{l+1}(v_{l+1})$, at least one recovery procedure is executed.

PROOF: There is at least one crash in between $\mathsf{PreWrite}_l(v_l)$ and $\mathsf{PreWrite}_{l+1}(v_{l+1})$: by the network model of assumption, a recovery procedure is executed upon recovery before the start of the next operation, and the PreWrite() is at the beginning of a Write() operation.

⟨9⟩2. During this recovery procedure, a PreRead() is executed which returns the highest timestamp from a ReadAMS() invocation.

PROOF: The PreRead() is executed at line 10 of Figure 3.9. It selects the highest timestamp at line 6.

⟨9⟩3. The lowest timestamp which can be read by the PreRead() is $ts_l - 1$.

PROOF:

Before $\mathsf{PreWrite}_l([v_l, ts_l])$ starts, $ts_l - 1$ or a bigger timestamp is stored using WriteAMS(): before each PreWrite(), the local timestamp is incremented by one (Line 7 of Figure 3.7). This increment is preceded by another PreWrite() which stores $ts_l - 1$ using WriteAMS().

⟨9⟩4. During the recovery procedure, the highest timestamp read from a ReadAMS() invocation is incremented by 2.

PROOF: Line 12 of Figure 3.9.

⟨9⟩5. Q.E.D.

PROOF:the minimum possible timestamp ts_{l+1} is such that $ts_{l+1} = (ts_l - 1) + 2 + 1 = ts_l + 2$, thus $ts_{l+1} > ts_l$.

⟨8⟩3. Q.E.D.

⟨7⟩3. Q.E.D.

PROOF: By induction.

⟨6⟩3. Q.E.D.

⟨5⟩3. PreWrite($[v, ts]$) stores $[v, ts]$ using WriteAMS() without stable storage or store() with stable storage.

PROOF: Property P_1.

⟨5⟩4. PreRead() returns a value using ReadAMS() without stable storage or retrieve() with stable storage.

PROOF: Line 5 of Figures 6 and 7.

⟨5⟩5. Q.E.D.

⟨4⟩2. Q.E.D.

```
1: function Write(v) at p_i
2:    V := W.ReadAMS()
3:    ts_h := highest_ts(V)        {highest_ts(V) returns the highest timestamp in the set V}
4:    ts_w := ts_h + 1
5:    PreWrite([v, ts_w])
6:    W.WriteAMS([v, ts_w, i])
7: return
```

Figure 3.10: Modifications to the single-writer algorithm to support the multi-writer case.

⟨3⟩3. If, during the recovery phase, PreRead() returns v prewritten by incomplete $\mathsf{Write}_{IC}(v_n)$, $\mathsf{Write}_{IC}(v_n)$ will be completed before the start of the next $\mathsf{Write}(v_{n+1})$.

PROOF: Line 10 of Figure 3.8 and Line 14 of Figure 3.9 show that during the recovery procedure the value v returned by PreRead() is written using a WriteAMS() invocation, thus completing $\mathsf{Write}_{IC}(v_n)$.

⟨3⟩4. Q.E.D.

PROOF: Step ⟨3⟩2 shows that upon recovery, a PreRead() is executed before the start of the next $\mathsf{Write}_{IC}(v_n)$ that returns the last prewritten value. This value is either written, thus completing the $\mathsf{Write}_{IC}(v_n)$ (step ⟨3⟩3) or will never be completed in the future. This implies that if an incomplete $\mathsf{Write}_{IC}(v_n)$ is completed, it will be completed before the start of the next Write().

⟨2⟩2. Q.E.D.

⟨1⟩3. Q.E.D.

3.5.3 Multi-writer Case

Adapting the algorithm of Figure 3.7 to the multi-writer case requires only minor changes; the main difference being that the writer first needs to contact the processes in order to determine the latest timestamp. As in [Lynch and Shvartsman 1997] the timestamp is tagged with the writer's process *id* in order to distinguish between writers using the same timestamp. The PreWrite() procedures (Figures 3.8 and 3.9) do not need to be changed: the mechanism is local to each writer in the sense that a writer can only PreRead() its own PreWrite(). The specific changes to the algorithm are shown in Figure 3.10.

The proof of correctness is almost the same as for the single-writer algorithm, modulo the addition of the following Lemma:

Lemma 10 *If op_1 and op_2 are concurrent, then if op_1 is a Write(), either $op_1 \prec op_2$ or $op_2 \prec op_1$.*

PROOF: because the writer appends its process id to the sequence number, other processes can distinguish between two simultaneous writes when both writers use the same sequence numbers. These timestamps are compared lexicographically, thus ensuring that two concurrent writes do not have the same timestamp. □

3.6 Complexity

In this section we give lower bounds on the resilience, log- and time-complexity of robustly implementing atomic storage in a crash-recovery model. We also point out instances of our algorithms that match these bounds, showing that they are thus tight.

3.6.1 Resilience

The following bound determines the maximum number of faulty processes f that an atomic storage can tolerate. The first bound (which our algorithms match) is trivial and is a simple rephrasing in the crash-recovery case of the one in [Attiya et al. 1995] which states that a majority of correct processes are needed for any distributed storage:

Resilience Bound 1: An atomic read/write storage requires that $f < \frac{n}{2}$.

ASSUME: Possible with $f = \lceil \frac{n}{2} \rceil$.
PROVE: False.
PROOF: Imagine an execution with a *write* followed by a *read*. During the *write* operation only $\lfloor \frac{n}{2} \rfloor$ processes can be contacted because $f = \lceil \frac{n}{2} \rceil$ can be permanently crashed and robustness can be violated if a process contacts more than $n - f$ processes. If all the processes that were contacted during the *write* crash permanently (possible since $\lfloor \frac{n}{2} \rfloor \leq f$) then the subsequent *read* cannot return the latest written value: this contradicts the atomicity requirement. □

The next bound relates the number of processes that need stable storage to the number u of processes that do not crash. Our generic storage algorithm is correct with $u = f + 1$ and the bound is therefore tight.

Resilience Bound 2: An atomic read/write storage requires that $u > f$ if $s \leq 2f$.

ASSUME: Possible with $u = f$ and $s \leq 2f$.
PROVE: False.
⟨1⟩1. A *write* can contact a maximum of $n - f$ processes, we call this set Q_W.
PROOF: Robustness can be violated if a process waits for more than $n - f$ responses because f processes can be permanently crashed.
⟨1⟩2. It is possible that $|Q_W \cap S| \leq f$
PROOF: Because of $s \leq 2f$ and step ⟨1⟩1.
⟨1⟩3. It is possible that $F \subset Q_W$ and $(S \cap Q_W) \subseteq F$.
PROOF: Because of $f < |Q_W|$ and step ⟨1⟩2.
⟨1⟩4. It is possible that $Q_W \cap U = \emptyset$.
PROOF: Because of step ⟨1⟩1 and the assumption that $u = f$.
⟨1⟩5. It is possible that all processes in Q_W crash at the same time
PROOF: By step ⟨1⟩4 and the fact that all processes not in U can crash.
⟨1⟩6. Q.E.D.
PROOF: Consider an execution with a *write* followed by a *read*. The *write* contacts the processes in Q_W. It is possible that all processes in Q_W crash

and only processes without stable storage eventually recover (step ⟨1⟩3). A subsequent *read* will not return the latest written value: a contradiction.

3.6.2 Log-Complexity

In this section we give bounds on the log-complexity of implementing an atomic storage. We only consider the case where $u \leq f$, because otherwise stable storage is not necessary at all (resilience bound 2 above). Resilience bound 2 also states that $s > 2f$, because otherwise it is impossible to implement an atomic storage with $u \leq f$. We first show that with these system assumptions, it is impossible to *write* a value without logging.

Log-Complexity Bound 1: Any algorithm $\mathcal{A}$, robustly implementing a single-writer/single-reader atomic storage, where $s > 2f$ and $u \leq f$, has an execution in which a *write* needs at least 1 log.

ASSUME: possible to *write* without logging with $s > 2f$ and $u \leq f$.
PROVE: False.
⟨1⟩1. Consider an execution with a *write* followed by a *read*.
⟨1⟩2. During the *write* operation only $n - f$ processes can be contacted.
PROOF: Robustness requirement.
⟨1⟩3. The $n - f$ processes that are aware of the new *write* value cannot log.
PROOF: Due to assumption.
⟨1⟩4. If among these $n - f$ processes, f crash permanently, then only $n - 2f$ processes are aware of the latest value.
⟨1⟩5. Q.E.D.
Since $u \leq f$, all $n-2f$ process can crash and recover thus losing the content of their volatile memory. Since none of them logged, the subsequent *read* cannot return the latest written value: a contradiction.

When there are not enough processes that do not crash during the execution of the algorithm, stable storage must be used. The following bound uses the notion of *causal logs* to refer to stable storage accesses. We say that two logs are causal if there is a causal precedence between the two logs, i.e. not all logs can be performed in parallel.

In a configuration with stable storage, our storage algorithm uses 2 causal logs per *write*. The following bound states that in fact more than 1 log is indeed necessary, therefore the bound is tight and our algorithm is optimal in that configuration.

Log-Complexity Bound 2: Any algorithm $\mathcal{A}$, robustly implementing a single-writer/single-reader atomic storage where $s > 2f$ and $u \leq f$, has an execution in which a *write* needs more than 1 log.

PROOF SKETCH: We consider the case of n processes where $n \geq 3$. We construct an execution that violates atomicity and is inevitable if only 1 log per *write* is allowed. Figure 3.11 depicts this execution, denoted ρ_1. Process p_1 is the writer and p_2 is the reader. In ρ_1 the writer successfully writes the value v_1 but crashes while writing v_2. After the crash, the writer recovers and starts a new *write*

operation. There are two reads (R_1 and R_2) by p_2 that are concurrent with the third *write*. We will show that it is impossible to complete the second *write*, thus making it possible for R_1 to return v_1 and for R_2 to return v_2. This execution then violates atomicity.

ASSUME: • 1 causal log per *write* is enough for every execution.

• n processes where $n \geq 3$.

PROVE: False

⟨1⟩1. The history H_1 associated with execution ρ_1 is not complete.

PROOF: the invocation W(v_2) has no matching reply.

⟨1⟩2. H_1 can be completed to obtain H_1' by removing W(v_2) from the history or by completing the *write* by adding a matching reply event to H_1.

PROOF: By definition.

⟨1⟩3. If a reply event is added to H_1, it must be placed *before* the invocation event W(v_3) at process p_1.

PROOF: The resulting history must be completed. By definition this implies that W(v_2) must be completed before the start of the next *write* at the same process.

⟨1⟩4. The following property cannot be satisfied: if a *read* invoked after the invocation of W(v_3) returns v_1, then no subsequent *read* returns v_2.

⟨2⟩1. It is impossible to guarantee that no *read* returns v_1 after the start of W(v_3).

In the considered model, a recovering process can initiate a recovery phase that is not limited by the number of communication steps, messages or logs it is allowed to perform. There are two cases to consider:

⟨3⟩1. It is impossible to "Cancel" v_1: no subsequent *read* can return v_1.

PROOF: Consider a *read* R_1 that is invoked after the invocation of W(v_3). Since W(v_2) was not completed, R_1 may not return v_2. Because R_1 is concurrent with W(v_3), it may not return v_3. This implies that R_1 can return an old value, written before W(v_1). This violates atomicity because W(v_1) is a complete *write*: $\mathcal{A}$ cannot cancel v_1.

⟨3⟩2. It is impossible to complete W(v_2) such that a subsequent *read* will only return v_2 or v_3.

ASSUME: Possible

PROVE: False

PROOF: Consider execution ρ_2 which is the same as ρ_1, but where p_1 contacts only a single processes from Q_W before crashing. Since only a single log is allowed, p_1 could not have logged the fact that it started W(v_2): if it did, no other process could log and it would be easy to contradict atomicity. Now consider execution ρ_3 which is the same as ρ_1, except that there is no W(v_2) invocation. After the crash, executions ρ_2 and ρ_3 are indistinguishable if the single process that was contacted by p_1 in ρ_2 is never contacted in ρ_3. In ρ_3 R_1 can only return v_1 and therefore too in ρ_2. This is a contradiction.

⟨3⟩3. Q.E.D.

⟨2⟩2. It is impossible to guarantee that no *read* returns v_2 after the start of W(v_3)

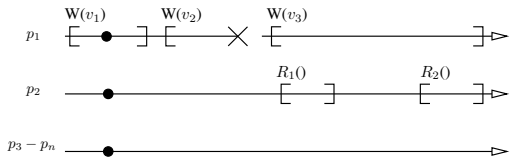

Figure 3.11: Execution ρ_1 (Proof of Log-Complexity Bound 2).

The only way to do this is to cancel v_2 so that all subsequent reads only return v_1 or v_3. But v_2 can only be canceled if v_2 has not yet been read by some other process. Upon recovery, the writer process (i.e. p_1) must initiate a recovery phase that first tests if v_2 has been read (say this phase is initiated at time T_1) and if not the recovery phase ensures that v_2 will never be read (from time T_2). If T_1 is not equal to T_2, then the reader could still *read* v_2 in between T_1 and T_2. Since a *read* initiated after T_2 can return v_1, atomicity can be violated. A completely asynchronous model is assumed and since the writer process must contact other processes to know if v_2 has been read, T_1 cannot be equal to T_2.

⟨2⟩3. Q.E.D.

⟨1⟩5. Q.E.D.

In a configuration with stable storage our generic storage algorithm uses 1 log per *read*. The following bound states when a *read* cannot do without logging:

Log-Complexity Bound 3: No algorithm $\mathcal{A}$, robustly implementing a single-writer/single-reader atomic storage where $s > 2f$ and $u \leq f$ has an execution in which a *read* does not log.

Proof sketch: We prove our result using indistinguishability arguments among three executions displayed in Figure 3.12. Let p_1 be the writer and p_2 be the reader with a total of $n \geq 3$ processes in the system.

Assume: There exists such an algorithm that never logs during a *read*.

Prove: False.

⟨1⟩1. Each execution ρ_i has an associated history $H_i, i \in 1, 2, 3, 4$.

Proof: By definition.

⟨1⟩2. Execution ρ_2 is atomic.

Proof: The writer p_1 writes value v_1 followed by v_2. The reader process crashes and reads v_1 after recovering. Execution ρ_2 satisfies atomicity because H_2 is equivalent to the legal sequential history made of the following ordered object events: W(v_1),R(v_1),W(v_2).

⟨1⟩3. Execution ρ_3 is atomic.

Proof: Process p_2 reads before crashing and returns v_2. Execution ρ_3 satisfies atomicity because H_3 is equivalent to the legal sequential history made of the following ordered object events: W(v_1),W(v_2),R(v_2).

⟨1⟩4. For process p_2, the execution ρ_4 is indistinguishable from execution ρ_3 up to time T.

Proof: In both executions up to time T, p_2 and the other processes perform

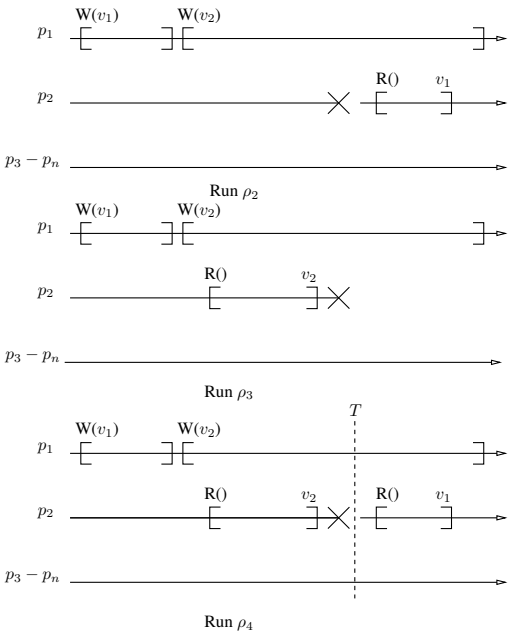

Figure 3.12: Executions ρ_2, ρ_3 and ρ_4 (Proof of Theorem 3.6.2).

exactly the same operations.

⟨1⟩5. After time T, the execution ρ_4 is indistinguishable from execution ρ_2 for p_2.

PROOF: In both executions after time T, p_2 and the other processes perform the exact same operations. Furthermore, because of the initial assumption that no process can log, process p_2 cannot "remember" anything about its previous state after it recovers from a crash.

⟨1⟩6. Q.E.D.

PROOF: Because of steps ⟨1⟩4 and ⟨1⟩5 execution ρ_4 is inevitable. This contradicts the assumption that the storage guarantees atomicity, since there is no legal sequential history which is equivalent to H_4 and that respects its operation precedence. Therefore it is impossible to implement an atomic storage that does not log during a *read*.

Intuitively, the previous bound makes sense considering that, in the crash-stop model, Theorem 10.4 of [Attiya and Welch 1998] states that every reader must "write" to implement a single-writer/multi-reader storage.

When $s > 2f$ and $u \leq f$, the storage algorithm presented in this thesis uses 2 causal logs per *write* and 1 causal log per *read* (whether the reader and writer have stable storage or not) and therefore the previous bounds are tight.

3.6.3 Time-Complexity

The way we measure time-complexity is the traditional counting of the number of round-trips [Lynch and Shvartsman 1997] needed for an operation to complete. If a process p sends messages to k different processes after the invocation of the operation and subsequently receives $r < k$ causally dependent [Lamport 1978]

responses from r different processes before returning from the operation, we say that the time-complexity of the operation is 1 round-trip.

When the writer uses stable storage, our algorithm shows that only 1 round-trip is needed per *write*. It is obvious that it cannot be done in less. However, when no stable storage is available to the writer, our algorithm uses 2 round-trips. The following bound shows that, in such a configuration, more than 1 round-trip is always necessary. Our algorithm is thus optimal and the bound is tight.

Time-Complexity Bound 1: Any algorithm $\mathcal{A}$, robustly implementing a single-writer/single-reader atomic storage where $s = 0$ has an execution in which a *write* needs more than 1 round-trip.

PROOF SKETCH: We consider the case of n processes where $n \geq 3$. We construct an execution that violates atomicity and is inevitable if only 1 round trip per *write* is allowed. Figure 3.11 displays this execution, denoted ρ_1. Process p_1 is the writer and p_2 is the reader. In ρ_1 the writer successfully writes the value v_1 but crashes while writing v_2. After the crash, the writer recovers and starts a new *write* operation. There are two reads (R_1 and R_2) by p_2 that are concurrent with the third *write*. We will show that it is impossible to complete the second *write*, thus making it possible for R_1 to return v_1 and for R_2 to return v_2. This execution then violates atomicity.

ASSUME:
- 1 round trip per *write* is enough for every execution.
- n processes where $n \geq 3$.

PROVE: False.

⟨1⟩1. The history H_1 associated with execution ρ_1 is not complete.
PROOF: The invocation $\mathrm{W}(v_2)$ has no matching reply.

⟨1⟩2. H_1 can be completed to obtain H_1' by removing $\mathrm{W}(v_2)$ from the history or by completing the *write* by adding a matching reply event to H_1.
PROOF: By definition.

⟨1⟩3. If a reply event is added to H_1, it must be placed *before* the invocation event $\mathrm{W}(v_3)$ at process p_1.
PROOF: The resulting history must be completed. By definition this implies that $\mathrm{W}(v_2)$ must be completed before the start of the next *write* at the same process.

⟨1⟩4. The following property cannot be satisfied: if a *read* invoked after the invocation of $\mathrm{W}(v_3)$ returns v_1, then no subsequent *read* returns v_2.

⟨2⟩1. It is impossible to guarantee that no *read* returns v_1 after the start of $\mathrm{W}(v_3)$.

In the crash-recovery model, a recovering process can initiate a recovery phase that is not limited by the number of communication steps or messages it is allowed to perform. There are two cases to consider:

⟨3⟩1. It is impossible to "Cancel" v_1: no subsequent *read* can return v_1.
PROOF: Consider a *read* R_1 that is invoked after the invocation of $\mathrm{W}(v_3)$. Since $\mathrm{W}(v_2)$ was not completed, R_1 may not return v_2. Because R_1 is concurrent with $\mathrm{W}(v_3)$, it may not return v_3. This implies that R_1 can return an old value, written before $\mathrm{W}(v_1)$. This violates atomicity because $\mathrm{W}(v_1)$ is a complete *write*: $\mathcal{A}$ cannot cancel v_1.

⟨3⟩2. It is impossible to complete W(v_2) such that a subsequent *read* will only return v_2 or v_3.

ASSUME: Possible.

PROVE: False.

PROOF: Consider execution ρ_2 which is the same as ρ_1, but where p_1 contacts only a single processes from Q_W before crashing. Since only a single round trip is allowed, p_1 could not have stored the fact that it started W(v_2) at other processes. Now consider execution ρ_3 which is the same as ρ_1, except that there is no W(v_2) invocation. After the crash, executions ρ_2 and ρ_3 are indistinguishable if the single process that was contacted by p_1 in ρ_2 is never contacted in ρ_3. In ρ_3 R_1 can only return v_1 and therefore too in ρ_2. This is a contradiction.

⟨3⟩3. Q.E.D.

⟨2⟩2. It is impossible to guarantee that no *read* returns v_2 after the start of W(v_3).

The only way to do this is to cancel v_2 so that all subsequent reads only return v_1 or v_3. But v_2 can only be canceled if v_2 has not yet been read by some other process. Upon recovery, the writer process (i.e. p_1) must initiate a recovery phase that first tests if v_2 has been read (say this phase is initiated at time T_1) and if not the recovery phase ensures that v_2 will never be read (from time T_2). If T_1 is not equal to T_2, then the reader could still *read* v_2 in between T_1 and T_2. Since a *read* initiated after T_2 can return v_1, atomicity can be violated. A completely asynchronous model is assumed and since the writer process must contact other processes to know if v_2 has been read, T_1 cannot be equal to T_2.

⟨2⟩3. Q.E.D.

⟨1⟩5. Q.E.D.

3.7 Discussion

3.7.1 Revisiting the assumptions

Throughout the thesis we assumed that besides an id, a process maintains a local clock that persists upon crashes and recoveries. This assumption is very realistic, for most machines we know off typically have battery powered clocks. One might wonder however whether the assumption of a local clock is actually needed; i.e. whether we cannot assume that the local clock is stored in volatile memory. The answer is no, and intuitively, this is because the clock is a key mechanism to uniquely identify requests. Assume by contradiction that a process does only remember its id upon recovery. We argue below that even a safe storage cannot be implemented if all but one process (the reader) is always up. Assume a system with n processes in which we implement a single reader, single writer safe storage SR. Out of the n processes, only the reader p_r is eventually up (i.e. can crash and recover), all other processes are always-up (i.e. they never crash). Assume that upon recovery, the reader has no information about its state before crashing and has no local clock. Consider the run α of SR as follows:

1. The reader p_r crashes and recovers.
2. Upon recovery p_r executes a recovery procedure followed by a read. The read returns v_0.
3. p_r crashes again.
4. The writer p_w invokes and completes the write of v_1.
5. p_r recovers and executes a recovery procedure followed by a read.

Remember that the channels we assume, fair-lossy channels, need to duplicate messages in order to ensure reliable delivery. Thus in run α after p_r's first crash and recovery (2), all messages sent in reply to p_r's requests can be duplicated. Because the algorithm is deterministic and p_r has no stable storage, the first message that is sent by p_r after the second recovery (3) is exactly the same as after the first recovery (2). Since p_r has no way of distinguishing the old duplicate messages from the new messages, it receives the same messages in (2) as in (3) due to asynchrony. Since the algorithm is deterministic, the value returned by the second read will be the same as in the first read: v_0, thus violating safety.

Maybe surprisingly, it was shown in [Aguilera et al. 1998] that consensus can be solved with processes that do not maintain any local clock, and yet tolerate crashes. The difference is that consensus is a one shot problem. Processes that crash and recover do not actively participate in the algorithm (they only wait for the decided value) and cannot initiate new requests.

3.7.2 Strong vs. Weak Completion

Because atomic storage is the strongest form of storage, it is also the most expensive to implement. In this section we discuss how by weakening the consistency requirements of the *read* and *write* operations, we allow faster implementations in terms of log and time-complexity.

We first introduce in the following a notion of *weak completion*. Remember that the atomicity criterion we consider in the crash-recovery model guarantees that crashes and recoveries are invisible to the client. To provide the illusion of transparency, we require that any *write* operation be completed before any new one from the same process is invoked. Weak completion differs from such a strong completion property in that the full illusion of hiding crashes and recoveries can be temporarily broken when a process recovers after a failure. More precisely, with weak completion, when a writer p_w crashes in the middle of executing a *write* operation, recovers and invokes a new *write* operation, other processes might have the impression that the two operations from the same process are invoked concurrently: the present *write*, as well as the *write* that p_w had invoked but not terminated prior to its last crash.

To illustrate the difference between the two forms of completion, we depict two executions in Figure 3.13: one of a storage that ensures atomicity and one that ensures weakly complete atomicity. The execution of the weakly complete atomic storage exhibits an "overlapping *write* behavior". What happens is that, during the third *write* ($\mathrm{W}(v_3)$) at p_1, the other processes do not know if the second

write (W(v_2)) was successful or not, and can still return the value written by the first *write*. The main difference is that the end of the second *write* can in fact be delayed until the end of a consecutive *write*. The writer itself would not be affected by the "overlapping" writes.

Strong completion relies on the completion of writes. Intuitively, this means that incomplete writes do not "overlap" with any consecutive writes at the same process, something that weak completion does not prevent. Although weak completion allows "strange" behavior after crashes, it is actually quite useful: in periods without crashes, the weakly complete storage will behave exactly the same as its stronger counter part. The advantage with weak completion is that it is cheaper, as we will discuss now.

Indeed, if we consider weakly complete atomicity, our second log-complexity bound (Section 3.6.2) and the first time-complexity bound (Section 3.6.3) do not hold: both lower bounds result from the need of completing writes. In fact it is therefore possible to implement a single-writer/single-reader weakly complete atomic storage where $s > 2f$ and $u \leq f$ with only 1 log per *write*, or without stable storage using only 1 round-trip per *write*.

3.7.3 Safety and Regularity Semantics

Several alternatives to atomicity have been defined in the literature. The weakest possible storage semantics are referred to as *safety*, in which inconsistent values can be returned in the case of concurrent access to the storage [Lamport 1985]: a *read* that is concurrent with a *write* can return any arbitrary value. A stronger form, called *regularity*, restricts reads that are concurrent with writes to return either the value being currently written, or the previously written value [Lamport 1985]. The original specification of regularity only considers single writer scenarios, but the specification has recently been extended to include multiple writers [Shao et al. 2003]. In order to implement a strongly complete regular storage, our second log-complexity lower bound (Section 3.6.2) and first time-complexity lower bound (Section 3.6.3) apply (also to the multi-writer case) and thus, if $s > 2f$ and $u \leq f$, 2 causal logs per *write* are necessary, or 2 round-trips without stable storage.

3.7.4 Performance Analysis

In order to analyze the performance of the algorithms described in the previous sections, a version of each storage algorithm was implemented and several experiments were run. The goal of these experiments was to precisely measure the

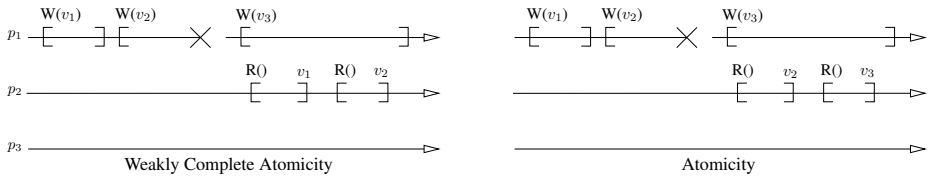

Figure 3.13: Atomic vs. weakly complete atomic storage.

cost of logging in an atomic storage. How much more expensive is it to support crash-recovery in the first place? How much more expensive is it to guarantee strong completion, rather than weak completion?

Implementation and Setup

Our algorithms are written in C, using low level network abstractions such as IP-multi-cast and UDP. The storage abstractions are implemented using files written to disk synchronously so that the operating system writes the data to disk immediately instead of buffering several writes together (which would violate even weak completion).

The experiments were run on a 100Mbps local area network using up to nine Pentium IV workstations running linux and equipped with standard IDE hard drives. Each workstation runs the same executable. The only parameter that needs to be set initially is the number of nodes in the storage. Every workstation runs one process participating in the storage and consists of two threads: one that listens for and executes read and write commands, and one that responds to broadcasted messages.

Experimental Results

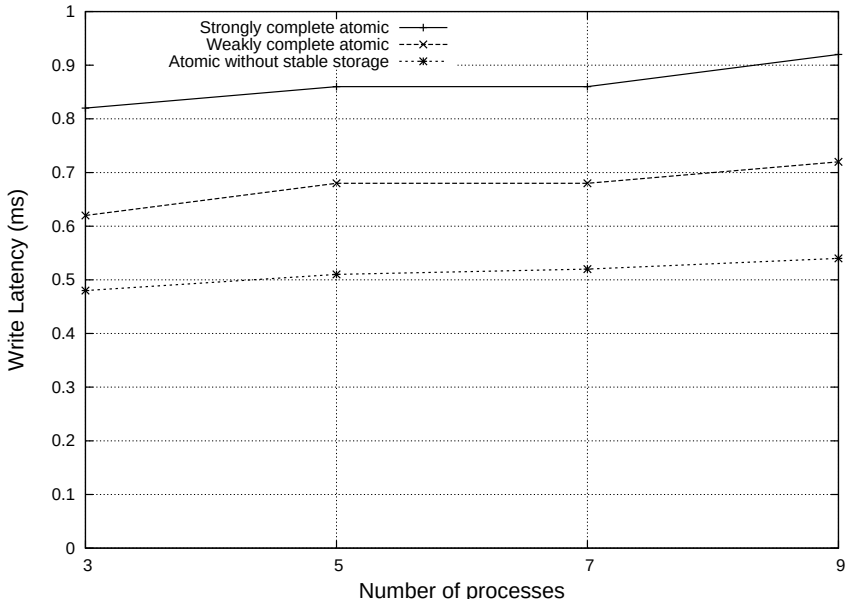

Figure 3.14: Latency of an atomic memory emulation.

The first experiment consisted of writing a 4 byte integer value and measuring the time that the operation took to complete, repeating the *write* fifty times and finally averaging the *write* times. These measurements were performed on a varying number of workstations for three different algorithms: atomic crash-stop, weakly complete atomic crash-recovery and strongly complete atomic crash-recovery. The results of the experiment are shown in the top graph of Figure 3.14.

The reason why the graph only shows the *write* latency is that in a run without any concurrency or crashes a *read* does not log, meaning that the execution times would be the same for each algorithm.

From the graph it is easy to distinguish between the three different algorithms: there is a clear performance impact due to logging. If we take the case of n=5 workstations, the average *write* latency without logging is 500μs, weak completion it is 700μs and for strong completion it is 900μs. Thus the performance impact due to logging is 200μs for weak completion and double that for the strong completion. This illustrates why counting the number of *causal* logs is so important: weak completion requires a single causal log and persistent atomicity two, reflecting the doubling of the latency due to logging.

The second experiment was designed to study the performance impact of increasing the size of the data stored in the storage. The size of the data that can be written by one *write* is limited by the fact that a UDP packet cannot contain more than 64KB of data; cutting up the data into chunks would change the algorithm by requiring more messages per write. Figure 3.15 plots the average write times with respect to the data size for five workstations. We can conclude from the graph that the time it takes to log and the time it takes to send a message over the network increases linearly. This is of course only true for systems where network congestion is not an issue.

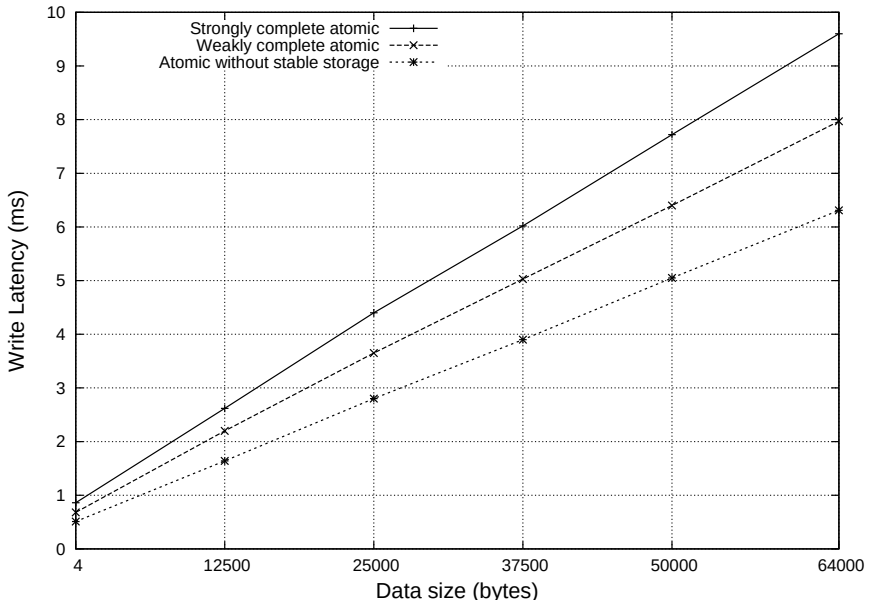

Figure 3.15: Latency with respect to data size.

4 Throughput

> We've heard that a million monkeys at a million keyboards could produce the complete works of Shakespeare; now, thanks to the Internet, we know that is not true.
>
> Robert Wilensky

4.1 Introduction

We focus on crash failures and we consider a homogeneous cluster of server machines. Such clusters usually have low inter-server communication latency and fine tuned TCP channels that make failure detection *reliable*. However, one might expect that such clusters also deliver low latency client operations, especially in failure-free and synchronous situations for these are considered the most frequent in practice.

Studying lower bounds on the latency of distributed storage algorithms has been a very active area of research in the last decade [Abraham et al. 2005; Dutta et al. 2004; Guerraoui and Vukolic 2006]. In general, such studies focus on the *isolated* latency of a read or a write operation, assuming in particular that every server is ready to perform this operation. In practice, when a high number of clients are served concurrently, low overall latency can only be provided with high *throughput*. In short, under high load, the latency perceived by clients is the sum of the time spent waiting for the operation to be served plus the actual service time. Clearly, when a lot of clients access the storage concurrently, the higher the throughput, the smaller the waiting time. Ideally, one would aim at *scalability*, meaning that increasing the number of machines should improve the throughput of the storage system.

Existing quorum-based algorithms do not scale very well since a majority of servers need to receive all messages, and thus adding more servers does not help. The problem is exacerbated by the fact that quorum-based algorithms typically

use one-to-many communication patterns (multicasts) to disseminate the information quickly. The rationale is mainly that the cost of receiving one message is equal to that of receiving multiple messages, especially when compared to the message propagation time. While this might be true in widely distributed environments (e.g., Internet), this assumption does not hold in a cluster environment subject to a heavy load, which we consider in this work. Clearly, techniques that aim at optimizing latency of isolated operations are not necessarily the best when high throughput is desired.[1]

In this work, we exploit the availability of reliable failure detection in a homogeneous cluster environment to alleviate the need for quorum-based strategies. In fact, it might appear trivial to devise a storage algorithm with high throughput if failure detection is reliable. This is not however the case as we discuss below. First, atomicity and resilience induce an inherent trade-off between the throughput of reads and that of writes. Basically, the more servers are updated by a write, the less servers need to be consulted by a read in order to fetch the last written value (and vice-versa). It is typical to favor reads at the expense of writes, following the argument that reads should be expedited for they occur more frequently than writes. Since in our case maximum resilience (tolerating the failure of all but one server) is required, the writer should update all servers. In this case, a simple *read-one write-all-available* algorithm might appear to do the job. To ensure atomicity however, one needs to solve the *read-inversion* problem and prevent any read from returning an old value after an earlier read returned a new value. One way to address this issue is to add a write phase to every read. However, this clearly decreases the throughput. Besides, if write messages are simply broadcast to all servers, the throughput would suffer even more drastically under high load. Modern full-duplex network interfaces can indeed receive and send messages at the same time. However, when receiving several messages at the same time, collisions occur at the network layer [Culler et al. 1993a]. A retransmission is thus necessary, in turn causing even more collisions, ultimately harming the throughput of write operations.

We present in this thesis an atomic storage algorithm that is resilient to the crash failure of any number of readers and writers as well as to the crash failure of all but one server. In failure-free and synchronous periods, our algorithm has a high write throughput and a read throughput that grows linearly with the number of available servers. This is ensured even in the face of contention. Our algorithm is based on two key ideas. First, we ensure atomicity and prevent the read-inversion problem by adding a *pre-write* phase to the write (instead of a *write-phase* to the read). This idea is, we believe, interesting in its own right because reads are local and immediate when there is no contention. Second, we organize the servers following a *ring* to ensure constant write throughput and avoid collisions during concurrent writes. This second idea was also used in implementing a total order broadcast primitive [Amir et al. 1995; Défago et al.

[1] Certain techniques can be used to improve the throughput of quorum-based algorithms. In [Saito et al. 2004] for instance, reads that are issued during contention free periods are handled more efficiently than reads that are concurrent with write operations. Yet reads still need to contact at least a majority of servers, which means that under high loads, there is no improvement of the throughput.

2004; Guerraoui et al. 2006], and one might actually wonder here whether it would not have been interesting to consider a modular approach in devising an atomic storage algorithm using such a primitive. Ensuring the atomicity of the storage would however have required to also totally order the reads, hampering its scalability. In [van Renesse and Schneider 2004], servers are organized in a chain to ensure high throughput for replica updates. This replication scheme can then be used to obtain a distributed atomic storage with high write throughput. However, the reads (also called queries) are always directed to the same single server and are therefore not scalable. We ensure liveness by imposing a *fairness* strategy on the servers.

We evaluated the implementation of our algorithm on a cluster of 24 machines (dual Intel 900MHz Itanium-2 processors, 3GB of RAM) with dual fast ethernet network interfaces (100 Mbps). We achieve 81 Mbps of write throughput and 8×90 Mbps of read throughput (with up to 8 servers). To our knowledge, our algorithm is the first atomic storage algorithm to achieve a read throughput that grows linearly with the number of available servers.

We also present LCR, a throughput optimal uniform total order broadcast algorithm that relies on point-to-point communication channels between processes. It is an algorithm using logical clocks and a ring topology (hence the name). Similarly to the train algorithm [Cristian 1991], each process only sends messages to the same single process. Unlike that algorithm however, messages in LCR are sequenced using logical vector clocks. These two characteristics ensure LCR throughput optimality and fairness, regardless of the type of traffic. In our context, fairness conveys the equal opportunity of processes to have their broadcast messages delivered.

We give a careful analysis of LCR's performance and fairness. We also provide performance results based on C and Java implementations of LCR that rely on TCP channels. The implementations are benchmarked against Spread and JGroups on a cluster of 9 machines and we show that LCR consistently delivers the highest throughput.

This chapter is organized as follows. Section 4.2 describes our system model. Section 4.3 presents the algorithm. The performance is discussed in Sections 4.4 and 4.5. Section 4.6 shows how to adapt the ring topology to provide uniform total ordering at the same cost. Section 4.7 describes the LCR algorithm in detail. Section 4.8 provides performance analysis and results and Section 4.9 compares the performance of LCR to that of Spread and JGroups.

4.2 Model

We consider a cluster environment where n homogeneous servers are connected via a local area network. We do not bound the number of client processes nor the concurrency among these. Any client can read or write in the storage. Every pair of processes communicate by message-passing using a bi-directional reliable communication channel (we do not assume FIFO channels here).[2]

[2]Even if we did assume FIFO channels, our fairness mechanism would make this assumption useless.

We focus on process crash failures: when a process crashes, it stops performing any further computation steps. A process that crashes is said to be *faulty*. A process that is not faulty is said to be *correct*. We make no assumption on the number of possible crashes except that at least one server should be correct in every computation.

We use a ring communication pattern, meaning that servers are organized in a ring and communicate only with their neighbors. Each server creates a TCP connection to its successor in the ring and maintains this connection during the entire execution of the algorithm (unless the successor fails). Because of the simple communication pattern, the homogeneous environment and low local area network latency, it is reasonable to assume that when a TCP connection fails, the server on the other side of the connection failed [Dunagan et al. 2004]. Using this mechanism we implement a *Perfect* failure detector (P) [Chandra and Toueg 1996b]. Although our algorithm tolerates asynchrony, its performance is optimized for synchronous periods during which message transmission delays are bounded by some known value. The throughput is measured during such periods.

Evaluating the performance of message-passing algorithms requires an adequate performance model. Some models only address point-to-point networks, where no native broadcast primitive is available [Bar-Noy and Kipnis 1994; Culler et al. 1993b]. Our algorithm does not use any broadcast primitive, but we do not wish to exclude it from our performance model for the sake of comparison with other algorithms. A recently proposed model [Urbán et al. 2000] is useful for reasoning about throughput, although it assumes that processes do not simultaneously send and receive messages. We would like to better capture the behavior of modern network cards which provide full duplex connectivity. The round-based model [Gafni 1998; Keidar and Shraer 2006; Lynch 1996] is in that sense more convenient as it assumes that a process can send a message to one or more processes at the start of each round, and can receive the messages sent by other processes at the end of the round. It is however not realistic in our cluster context to consider that several messages can be simultaneously received by the same process. Indeed, receiving two messages at the same time might result in a collision at the network level, requiring a retransmission. Whereas, this model is well-suited for proving lower bounds on the latency of algorithms, it is ill suited for predicting the throughput of these algorithms.

We propose to evaluate message-passing algorithms considering a synchronous round-based model but assuming the following: in each round k, every process p_i can execute the following steps: (1) p_i computes the message for round k, $m(i,k)$, (2) p_i broadcasts $m(i,k)$ to all or a subset of processes and (3) p_i receives at most one message sent at round k. The synchrony assumption implies that, at any time, all processes are in the same round. The broadcast primitive we assume corresponds to the multicast primitive provided at the ethernet level. There are no reliability guarantees, except in the absence of failures or collisions. The analytical analysis of the performance of our algorithm will be based on this model. Interestingly, the experimental evaluation confirms these numbers, conveying in some sense the validity of this model in our context of a homogeneous cluster.

4.3 The Storage Algorithm

Our storage algorithm was designed with the three following properties in mind: resilience, atomicity and high throughput. Our algorithm satisfies these properties using two key mechanisms: a *read-one pre-write/write-all-available* strategy and a *fairness* rule orchestrating the servers in a *ring*. In this section, we explain these key mechanisms and how they ensure the desired properties.

Figures 4.1 and 4.2 contain the pseudo-code of the storage algorithm.

```
At the client c:
 1: procedure initialization:
 2:    v ← ⊥, ts ← 0, id ← ⊥
 3:    pending_write_set ← ∅
 4:    forward_queue ← ∅
 5:    write_queue ← ∅
 6:    nb_msg[p_j] ← 0 ∀p_j ∈ S
 7: end

 8: procedure read ()
 9:    send ⟨read⟩ to any p_i ∈ S
10:    wait until receive ⟨read_ack,v⟩ from p_i
11:    return v
12: end

At the server process p_i:
13: upon receive ⟨read⟩ from c do
14:    if pending_write_set = ∅ then
15:       send ⟨read_ack, v⟩ to c
16:    else
17:       highest = max_lex(pending_write_set)
18:       wait until receive ⟨write, v', [ts', id']⟩ ∧ ([ts', id'] ≥_lex highest)
19:       send ⟨read_ack, v'⟩ to c
20:    end if
21: end upon

22: upon p_j crashed do
23:    if p_j = p_next then
24:       p_next = p_{j+1}
25:       send ⟨write, v, [ts, id]⟩ to p_next
26:       for each v', [ts', id'] ∈ pending_write_set do
27:          send ⟨pre_write, v', [ts', id']⟩ to p_next
28:       end for
29:    end if
30: end upon
```

Figure 4.1: The storage algorithm: initialization, read and recovery procedures.

Clients send Read and Write requests to any server in S. If the server contacted by the client crashes, the client re-issues the request to another server. Clients do not directly detect the failure of a server, but when their request times-out, they simply re-send it to another server.

As we pointed out, our algorithm is resilient in the sense that it tolerates the failure of $n-1$ out of n server processes and the failure of any number of clients. Atomicity [Herlihy and Wing 1990; Lamport 1998] dictates that every read or write operation appears to execute at an individual moment of time, between its invocation and responses. In particular, a read always returns the last written value, even if there is only one server that did not crash. We ensure this using a *write-all-available* scheme. Newly written values are sent to all processes before

```
At the client c:
 1: procedure write (v)
 2:    send ⟨write,v⟩ to any p_i ∈ S
 3:    wait until receive ⟨write_ack⟩ from p_i
 4:    return ok
 5: end

At the server process p_i:
 6: upon receive ⟨write, v'⟩ from c do
 7:    write_queue.last ← [v', c]
 8: end upon

 9: procedure write(v', c)
10:    highest = max_lex(pending_write_set)
11:    tag ← [max(highest.ts, ts) + 1, i]
12:    pending_write_set ← pending_write_set ∪ tag
13:    send ⟨pre_write, v', tag⟩ to p_next
14:    nb_msg[p_i] ← nb_msg[p_i] + 1
15:    write_queue ← write_queue − [v', c]
16: end

17: upon receive ⟨pre_write, v', [ts', id']⟩ do
18:    if id ≠ i then
19:       forward_queue.last ← ⟨pre_write, v', [ts', id']⟩
20:    else
21:       if [ts', id'] >_lex [ts, id] then
22:          [ts, id] ← [ts', id']
23:          v ← v'
24:       end if
25:       pending_write_set ← pending_write_set − [ts', id']
26:       send ⟨write, v, [ts', id']⟩ to p_next
27:    end if
28: end upon

29: upon receive ⟨write, v', [ts', id']⟩ do
30:    if id' ≠ i then
31:       if [ts', id'] >_lex [ts, id] then
32:          [ts, id] ← [ts', id']
33:          v ← v'
34:       end if
35:       pending_write_set ← pending_write_set − [ts', id']
36:       forward_queue.last ← ⟨write, v', [ts', id']⟩
37:    else
38:       send ⟨write_ack⟩ to c
39:    end if
40: end upon

41: task queue handler
42:    if forward_queue = ∅ then
43:       nb_msg[p_j] ← 0 ∀p_j ∈ S
44:       if write_queue ≠ ∅ then
45:          write(write_queue.first)
46:       end if
47:    else
48:       if write_queue ≠ ∅ then
49:          select p_j s.t. nb_msg[p_j] is minimal
50:       else
51:          select p_j ≠ p_i s.t. nb_msg[p_j] is minimal
52:       end if
53:       if p_j = p_i then
54:          write(write_queue.first)
55:       else
56:          msg ← select first in forward_queue sent by p_j
57:          send msg to p_next
58:          forward_queue ← forward_queue − msg
59:          pending_write_set ← pending_write_set ∪ msg.tag
60:          nb_msg[p_j] ← nb_msg[p_j] + 1
61:       end if
62:    end if
63: end
```

Figure 4.2: The storage algorithm: write procedures.

the write operation returns and each process keeps a local copy of the latest value.

Values are ordered using a timestamp which is stored together with the value. Processes only replace their locally stored values with values that have a higher timestamp. Since a write contacts all processes, a process wishing to perform a write does not need to contact any other process to get the highest timestamp. Before each write, the locally stored timestamp can simply be incremented, thus ensuring that timestamps increase monotonically. (Ties are broken using process ids).

Read operations do not involve any communication between server processes. Clients directly access the value stored locally at a server process. The difficulty here lies in ensuring atomicity using these *local* reads and in particular in preventing the *read inversion* problem. Consider an execution where a value is written and stored at all processes. Due to asynchrony, not all processes might learn about the new value at the same time. Before the write completes, a reader contacts a process that has the new value and thus returns the new value. Afterwards a second reader contacts a process which does not know of the new value (since the write is not yet completed) and thus returns the old value, violating atomicity.

Our algorithm handles this issue using a pre-write mechanism. The write involves two consecutive phases: a *pre_write* phase and a *write* phase. In the *pre_write* phase, all processes are informed of the new value that is going to be written. Only when all processes acknowledge the *pre_write*, does the *write* phase actually start.

During a read, if a process knows of a *pre_write* value that has not yet been written, it waits until the value has been written before returning it. This ensures that when the new value is returned, all processes know of the new value through the *pre_write* phase and any subsequent read will also return the new value. Consequently, when there are no unwritten *pre_write* values, processes can immediately return the latest written value. In the case of concurrent writes, processes might see several unwritten *pre_write* values, in which case they wait for the value with the highest timestamp to be written. As will be explained in Section 4.4, waiting increases the latency for a single request, but does not influence the throughput of a loaded system. An illustration of an algorithm execution is provided in Figure 4.3.

So far, no mention was made about the communication pattern that is used for contacting all processes during the actual writes. The choice of the communication pattern has no influence on the correctness of the algorithm, but it does influence the throughput. Our algorithm organizes all servers in a ring and messages are forwarded from each server to its neighbor. This simple communication pattern avoids any unpredictability causing message collisions, especially under high loads. Also, there is no need for explicit acknowledgment messages, since knowing that a message has been forwarded around the ring once implies that all processes have seen the message.

In the case of a crash of a server process p_j, the crash will eventually be detected by the crashed process' predecessor in the ring p_{j-1} using the perfect failure detector. The crashed process p_j will be removed from the ring. Any pending messages that were not forwarded due to the crash are forwarded to the

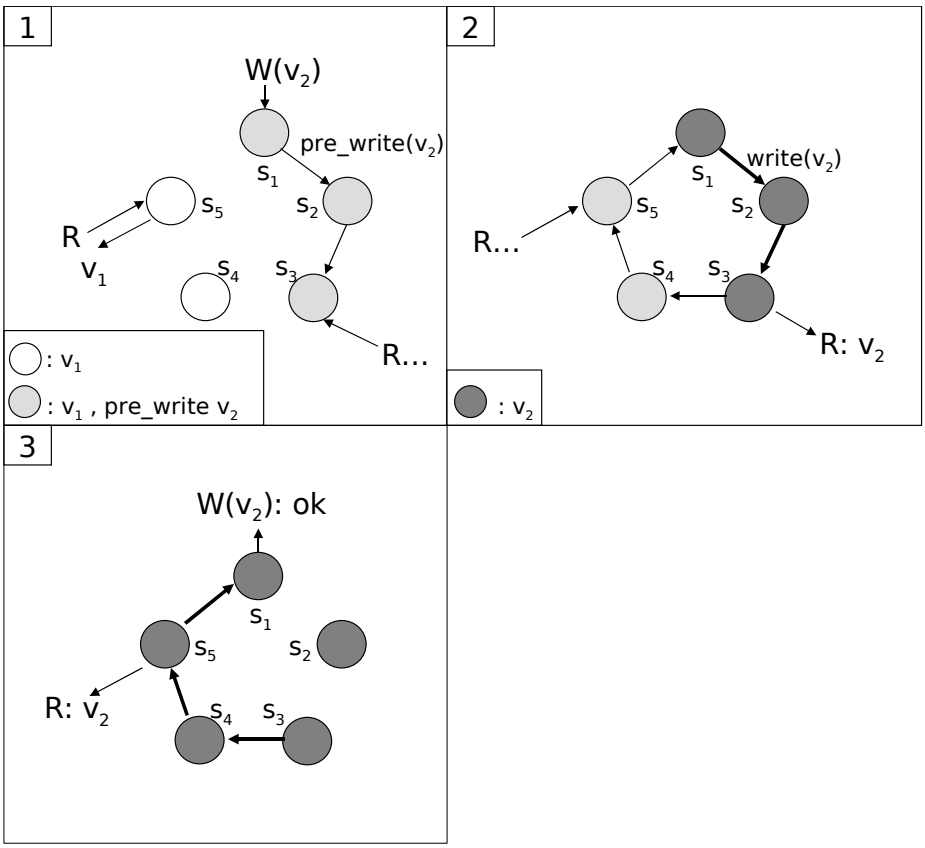

Figure 4.3: Illustration run of the storage algorithm. (1) After receiving the write request $W(v_2)$, s_1 sends a $pre\_write(v_2)$ message to its successor. A read request is received by s_3, which must wait before replying because of the pre-write, whereas s_5 can reply directly to the client's read request. (2) Upon receiving its own $pre\_write(v_2)$ message, s_1 sends a $write(v_2)$ message. Upon receiving this message s_3 can reply to its client's read request. Now s_5 must wait until it receives the $write(v_2)$ message before replying to a new read request. (3) Upon receiving its own $write(v_2)$ message, s_1 replies to the client and s_5 can also reply to its client.

new successor by p_j, ensuring that all *pre-write* and *write* messages are eventually forwarded around the ring. This however is not enough to ensure resilience. Under high loads, processes must decide to either forward messages received from their neighbor or initiate a new write upon receiving a request from a client. If each process would prioritize requests received from clients, no message would ever be forwarded on the ring.

Our algorithm handles this issue using a *fairness* mechanism which ensures that each process can complete its fair share of writes and that all write operations eventually complete. Each process keeps two queues: a *write_queue* which contains *write* requests received from clients and a $forward\_queue$ which contains messages received from the predecessor which are to be forwarded to the successor. A table nb_msg keeps track of how many messages have been forwarded for each process: there is an entry for each process p_j, counting the number of messages originating at p_j that were forwarded. Messages in the $forward\_queue$ are not forwarded in FIFO order, but the first message from the processes that has the smallest number of forwarded messages will be sent to the successor.

4.3.1 Correctness Proof

Theorem 8 *The multi-reader multi-writer storage algorithm guarantees atomicity.*

We prove atomicity using Lemma 13.16 of [Lynch 1996]. The lemma is as follows:
Lemma 13.16 *Let β be a (finite or infinite) sequence of actions of a read/write atomic object external interface. Suppose that β is well-formed for each i, and contains no incomplete operations. Let Π be the set of all operations in β.*

Suppose that $\prec$ is an irreflexive partial ordering of all the operations in Π, satisfying the following properties:

1. *For any operation $\pi \in \Pi$, there are only finitely many operations ϕ such that $\phi \prec \pi$.*
2. *If the response event for π precedes the invocation event for ϕ in β, then it cannot be the case that $\phi \prec \pi$.*
3. *If π is a WRITE operation in Π and ϕ is any operation in Π, then either $\pi \prec \phi$ or $\phi \prec \pi$.*
4. *The value returned by each READ operation is the value written by the last preceding WRITE operation according to $\prec$ (or v_0, if there is no such WRITE).*

Then β satisfies the atomicity property.

For a well-formed history H, the lemma lists four conditions involving a partial order on operations in H. It states that if there is a partial order relation on events satisfying these four conditions then the atomicity property is satisfied. Let O be the set of operations in H, and τ be the tag associated with the value written or returned by each operation. We define the partial order $PO = \langle O, \prec \rangle$ on the

operations by letting: $op_1 \prec op_2$ for $op_1, op_2 \in O$, if (a) $\tau(op_1) <_{lex} \tau(op_2)$, or if (b) op_1 is a Write, op_2 is a Read, and $\tau(op_1) =_{lex} \tau(op_2)$.

We start by proving the following preliminary lemmas before proceeding with the proof of Theorem 1.

Lemma 9 *When a message is added to the $forward\_queue$, it is eventually sent to p_{next}.*

Proof: Suppose that message m, originating at p_k was added to the end of the $forward\_queue$. Changes in the queue are handled in the algorithm at line 41 of Figure 4.2. The nb_msg table contains an entry for each process p_j, which represents the number of messages originating at p_j that were forwarded. Always, the first message from the processes that has been the least forwarded will be sent to p_{next}. In the worst case, suppose that $nb\_msg[p_k]$ is the highest in p_k. Three cases are possible: (1) several or all processes keep sending messages (2) no processes besides p_k send any further messages and (3), no processes send any further messages. In the first case, $nb\_msg[p_k]$ will eventually become the lowest value, all messages originating from p_k enqueued before m, and eventually m will be sent. In the second case, only messages sent by p_k will remain in the queue and m will be sent eventually. In the third case, the queue will empty and m will eventually be sent.

Lemma 10 *When a process receives a write request from a client, a prewrite message is created and forwarded around the ring, followed by a write message, before a reply is sent to the client.*

Proof: When process p_i receives a write request from a client, the request is added to the $write\_queue$ (line 7 of Figure 4.2). Then for the same reason as in Lemma 9, the write procedure is called (line 9 of Figure 4.2) which creates a prewrite message and sends it to p_{next}. First, consider the case without crashes. Because of Lemma 9, we know that the prewrite message will be forwarded around the ring until it arrives at p_i again. Then a write message is created and forwarded in the same way. When the write message is received by p_i a reply is sent to the client.
In the case of a crash of process p_j, the crash will eventually be detected by the crashed process' predecessor in the ring p_{j-1} due to the perfect failure detector P. The crashed process p_j will be removed from the ring. Any ongoing messages that were not forwarded due to the crash are forwarded to the new successor by p_{j-1}, thus ensuring that all prewrite and write messages are eventually forwarded around the ring.

Lemma 11 *The tag values stored at each process are monotone nondecreasing in H*

Proof: The tags are only changed in the algorithm at line 32 of Figure 4.2. A new tag is stored only if it is lexically greater than the previous tag.

Lemma 12 *If op_1 precedes op_2, then*
(i) if op_2 is a Read, then $\tau(op_1) \leq_{lex} \tau(op_2)$, and
(ii) if op_2 is a Write, then $\tau(op_1) <_{lex} \tau(op_2)$.

PROOF:
⟨1⟩1. op_2 is a Read.
⟨2⟩1. op_1 is a Read.
PROOF: Since op_1 completed, all correct processes received a *pending* message and at least one correct process has stored the tag $\tau(op_1)$. During the second Read op_2, if $pending\_write\_set = \emptyset$ at line 14 of Figure 4.1, there is no concurrent Write at the process p_r contacted by the client. This means that $\tau(op_1)$ has already been stored at p_r. Since this value is returned at line 15 of Figure 4.1, $\tau(op_1) \leq_{lex} \tau(op_2)$. In the case that $pending\_write\_set \neq \emptyset$ however, $\tau(op_1)$ may not yet have been stored at p_r. However, since $\tau(op_1)$ was returned during the first read, all processes know about $\tau(op_1)$ due to the *pre_write* phase (line 59 of Figure 4.2). Thus, p_r stored $\tau(op_1)$ in its $pending\_write\_set$, which ensures that the read will only be completed when it will have received a write message with tag greater than or equal to $\tau(op_1)$ (line 19 of Figure 4.1). Thus in all cases $\tau(op_1) \leq_{lex} \tau(op_2)$.
⟨2⟩2. op_1 is a Write.
PROOF: Since op_1 completed, all correct processes stored $\tau(op_1)$. Since the Read op_2 returns the value stored locally at a server process and comes after op_1, it is clear that $\tau(op_1) \leq_{lex} \tau(op_2)$.
⟨2⟩3. Q.E.D.
⟨1⟩2. op_2 is a Write.
⟨2⟩1. op_1 is a Read.
PROOF: Since op_1 completed, all correct processes received a *pending* message and at least one correct process has stored the tag $\tau(op_1)$. The process that will be contacted by Write op_2 will generate a higher timestamp than those stored in $pending\_write\_set$ (line 11 of Figure 4.2). Consequently, $\tau(op_1) <_{lex} \tau(op_2)$.
⟨2⟩2. op_1 is a Write.
PROOF: Since op_1 completed, all correct processes stored $\tau(op_1)$. During the Write op_2, before sending the process increments its locally stored timestamp by one (line 11 of Figure 4.2) before sending a new write message. Therefore $\tau(op_1) <_{lex} \tau(op_2)$.
⟨2⟩3. Q.E.D.
⟨1⟩3. Q.E.D.

We now prove Theorem 1 using Lemma 13.16:

PROOF:
⟨1⟩1. For any operation op_2 in H, there are only finitely many operations op_1 such that $op_1 \prec op_2$.
PROOF: Suppose by contradiction that the operation op_2 has infinitely many predecessors. We begin by showing that all $\tau(op)$ values for distinct Write operations in H are distinct. The tag τ combines a timestamp ts and the unique id of the server process which issued the write. Therefore write operations

issued by different server processes are unique. If we look at tags issued by a single server process we see that at line 11 of Figure 4.2, for each new write request, the local timestamp is incremented and stored. Therefore, no two distinct Write operations have the same tag $\tau(op)$ which implies that op_2 cannot have infinitely many Write predecessors. Without loss of generality, we can assume that op_2 is a Write.

Thus, there must be infinitely many Read operations with the same tag t, where $t < \tau(op_2)$. Since op_2 completes in H, this implies that eventually the tag $\tau(op_2)$ gets stored at all correct processes (the tag is propagated along a ring and the initiating process doesn't reply to the client until all correct processes have the new tag). Lemma 11 tells us that the tag values stored at each individual process are monotone nondecreasing in H and therefore any Read that is subsequently invoked is guaranteed to obtain a tag that is $\geq \tau(op_2) > t$. This contradicts the existence of infinitely many Read operations with tag t.

⟨1⟩2. If the response event for op_1 precedes the invocation event for op_2 in H, then it cannot be the case that $op_2 \prec op_1$.

There are four cases to consider:

⟨2⟩1. Read response precedes Write invocation.

PROOF: In this case, $\tau(op_1) <_{lex} \tau(op_2)$ by Lemma 12. Thus $op_2 \not\prec op_1$ by the PO construction.

⟨2⟩2. Read_1 response precedes Read_2 invocation.

PROOF: In this case, by the definition of PO, if $\tau(op_1) <_{lex} \tau(op_2)$, then $op_2 \not\prec op_1$, else if $\tau(op_1) =_{lex} \tau(op_2)$, then they are not ordered by the PO construction since they are both reads.

⟨2⟩3. Write response precedes Read invocation.

PROOF: In this case, $\tau(op_1) \leq_{lex} \tau(op_2)$ by Lemma 12 and $op_1 \prec op_2$ by the PO construction. Thus $op_2 \not\prec op_1$.

⟨2⟩4. Write_1 response precedes Write_2 invocation.

PROOF: In this case, $\tau(op_1) <_{lex} \tau(op_2)$ by Lemma 12. Thus $op_2 \not\prec op_1$ by the PO construction.

⟨2⟩5. Q.E.D.

⟨1⟩3. If op_2 is a Write operation in H and op_1 is any operation in H, then either $op_2 \prec op_1$ or $op_1 \prec op_2$.

PROOF: This follows directly from the definition of the PO. If op_1 is a Write operation then the tags will be unique since they are compared lexicographically. If op_1 is a Read operation then if its tag is smaller it implies $op_1 \prec op_2$, if its tag is larger it implies $op_2 \prec op_1$, or if it has the same tag $op_2 \prec op_1$.

⟨1⟩4. The value returned by each Read operation is the value written by the last preceding Write operation according to $\prec$ (or v_0, if there is no such Write).

PROOF: If we have a Write op_1 followed by a Read op_2, then according to Lemma 12, $\tau(op_1) = \tau op_2$ (If there is no preceding Write, then op_2 returns v_0).

⟨1⟩5. Q.E.D.

Theorem 13 *Every read and write request eventually completes.*

PROOF: Lemma 10 proves that all write requests eventually complete. Before a process can reply to a read request however, it has to wait if an uncompleted write

is detected (line 18 of Figure 4.2). Because of Lemma 10 all writes eventually complete and therefore all reads also eventually complete.

4.4 Analytical Evaluation

We consider two performance metrics: *Latency*, defined here as the number of rounds required for a client to complete a read or write invocation and *Throughput*, defined here as the number of invocations (read or write) that can be completed per round. Note that our throughput definition is similar to the one proposed in [Hendler and Kutten 2006].

4.4.1 Latency

The read latency of our algorithm is equal to 2 rounds, since a read operation only requires 1 round-trip from the client to the server. The latency of a write operation is equal to $2N + 2$ rounds. A write operation first requires the client to send a *write* message to the server (1 round). Then, the server sends a *pre_write* message along the ring (N rounds). Once it receives its own *pre_write* message, the server sends a *write* message along the ring (N rounds). Finally, upon the reception of its own *write* message, the server replies to the client with a *write_ack* message (1 round). The write latency is thus linear with respect to the number of servers.

4.4.2 Throughput

For simplicity of presentation, this analysis only considers messages exchanged between servers. Note that in our experimental setup, client messages do indeed transit on their own dedicated network. Our evaluation also shows however, that when clients and servers use the same network, they both share the available bandwidth evenly. Assuming that there exists at least 1 server that receives 1 new write request per round, our storage algorithm allows completing 1 write operation per round on average. This is due to the fact that (1) messages are disseminated along a ring once, (2) *write* messages are piggybacked on *pending_write* messages without the need for explicit acknowledgements, and (3) the fairness mechanism guarantees that write requests eventually complete. This ensures that each server can forward a new write message at the end of each round, and thus the algorithm allows 1 write request per round to complete on average.

Assuming that there are only read requests, the read throughput is equal to n. This is due to the fact that each of the n servers can reply to a different read invocation at each round. Thus, increasing the number of machines does not impact the write throughput, and favorably impacts the read throughput.

We now analyze the impact that concurrent writes have on reads. The fairness mechanism that is integrated into our algorithm guarantees that 1 write can be completed per round on average, and that the maximum latency of a write request is bounded (let l_{max} be this maximum latency). Consider a server s_i which receives an infinite number of read requests. Before replying to the client,

s_i must wait for the latest pre-write to complete, i.e. l_{max} rounds in the worst case. After an initial period of l_{max} rounds, s_i can fulfill 1 read request each round. The read throughput for s_i is therefore 1. Since reads do not involve additional communication between servers, each server can serve clients independently at a throughput of 1, bringing the total read throughput during R_{hl} to n.

4.5 Experimental Evaluation

Our algorithm was implemented in C (approximately 1500 lines). The implementation consists of separate code for a client (reader or writer) and a server. In order to stress the servers without needing an enormous number of client machines, the client application can emulate multiple clients, i.e. it can send multiple read and write requests in parallel. Thus, a single writing node can saturate the storage implementation (the servers and the network links) and so the maximum throughput, under high load, can be captured.

We performed the experiments using up to 24 homogeneous nodes (Linux 2.6, dual Intel 900MHz Itanium-2 processors, 3GB of RAM, 100Mbit/s switched ethernet). Similarly to the assumption made in Section 4.4.2, servers and clients are interconnected by two separate networks: server nodes are connected to each other on one network and communicate with clients on the other. The load is generated by two dedicated client machines for each server, either performing reads or writes depending on the experiment. Every measurement has been performed at least 3 times and the average over all measurements has been recorded.

An important parameter of the experiments is the size of each message that is being sent from a client to a server as a *write* request, or from a server to a client as a read response. In our implementation, this message size was constant among all clients/servers and during the whole experiment. Figure 4.4 depicts the relation between write throughput and message size which we measured in a setup of 8 servers. Clearly, for small messages the overhead of the standard message envelope and message processing time is too high and so the network cannot be saturated with payload data. For very high message size the throughput decreases slightly, which is probably due to limited buffer size on one of the many communication layers. Therefore, for all the further experiments we set the message size to 10 kB, which should give the maximum possible write throughput.

In the experiment of Figure 4.5, each server is connected to two client machines which generate read requests. There are no concurrent writes. The read throughput is measured at each client and the total is reported on the chart. It can easily be seen that the total read throughput increases linearly and is equal to 90 MBit/s per server. In the experiment of Figure 4.6, the clients generate only write requests. The write throughput when the number of servers is between 2 and 8 remains almost constant and is about 80 Mbit/s. It is also interesting to note that during the experiment, each client machine roughly observed the same write throughput, i.e. 80 Mbit/s divided by the number of servers.

The experiment of Figure 4.7 examines the total throughput of the system with write contention. The load on each server is generated by a dedicated reader and a dedicated writer. This represents a more realistic case in which read and write

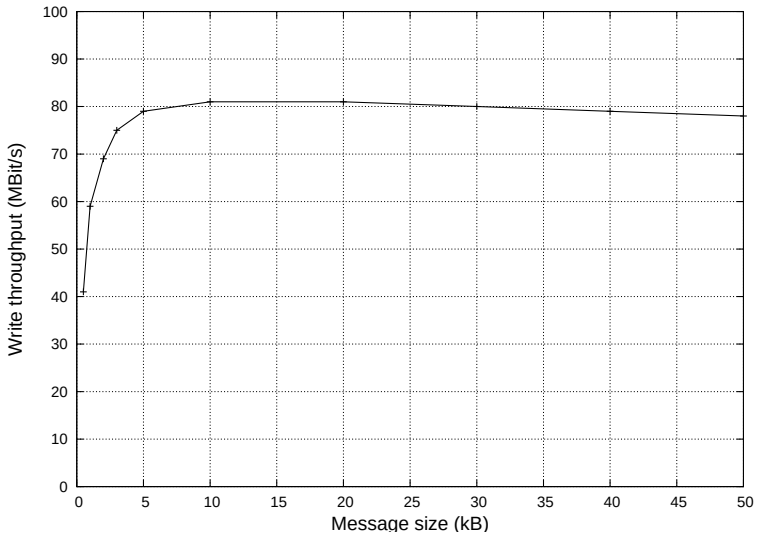

Figure 4.4: Influence of message size on the write throughput.

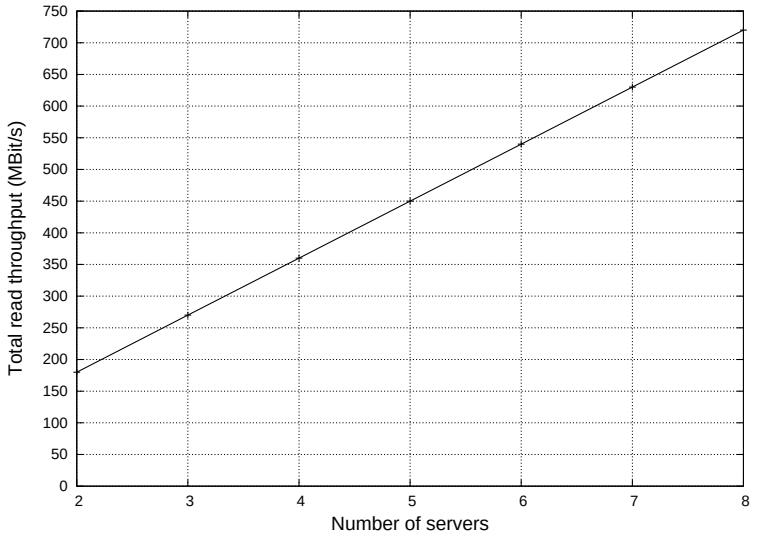

Figure 4.5: Read throughput without contention.

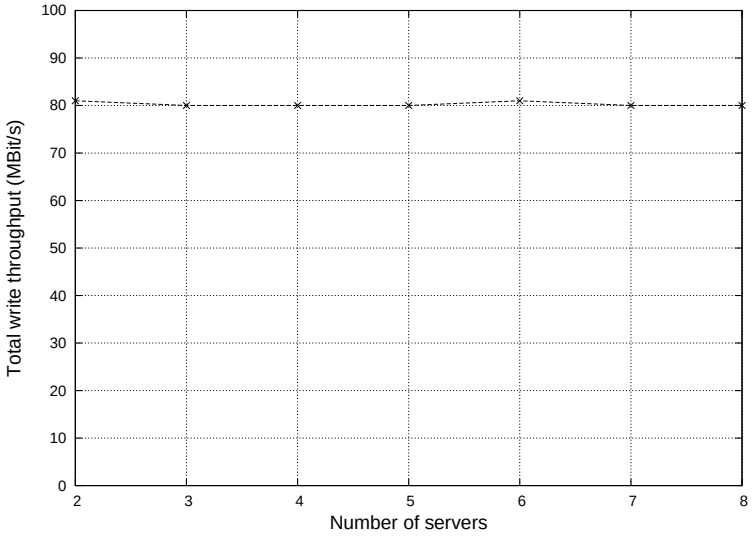

Figure 4.6: Write throughput without contention.

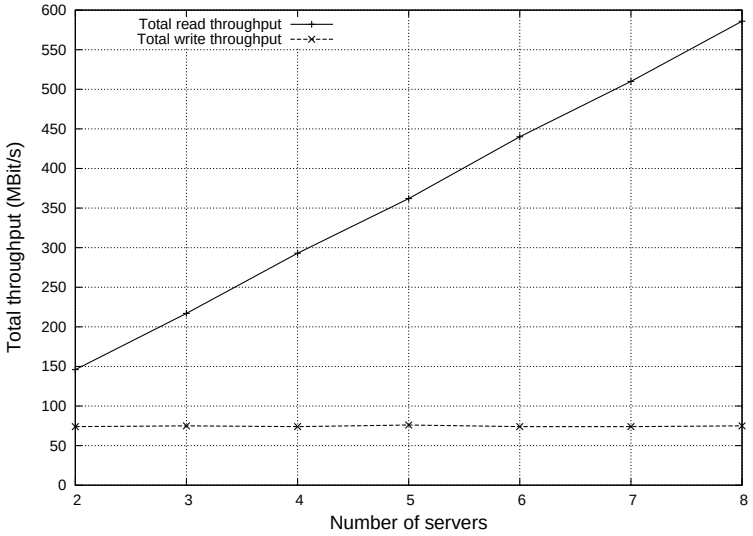

Figure 4.7: Read & write throughput contention on separate networks.

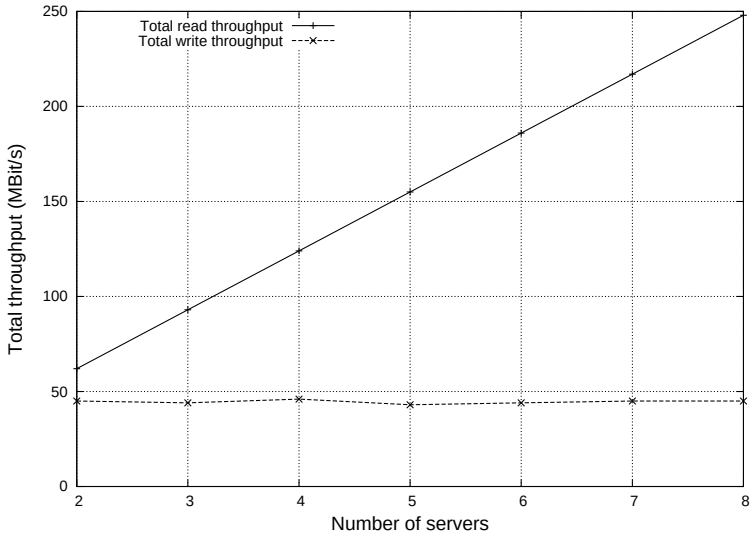

Figure 4.8: Read & write throughput contention on shared network.

requests are issued concurrently by many clients. The implementation behaves as predicted by our analytical analysis: the write throughput remains constant at around 80 Mbit/s and the read throughput scales linearly and is almost as high as in the contention free case (a performance penalty of about 15% is incurred). The decrease in performance is, we believe, due to the additional overhead of queuing client read requests while at the same time running the fairness mechanism for write requests.

The final throughput experiment of Figure 4.8 examines the total throughput of the system during contention when clients and servers share a single network connection. Obviously, read and write throughput suffer, but the write throughput remains constant at around 45 Mbit/s whereas the read throughput scales linearly at about 31 Mbit/s per additional server. This means that each server uses about 76 Mbit/s of its incoming and outgoing network bandwidth despite concurrency.

The latency measurements are presented in Figure 4.9. Because of the ring topology, the write latency grows linearly with the number of servers. The read latency stays constant since it involves only a single round-trip between the client and a server.

4.6 Total Order

The high throughput of the storage algorithm is due to the organization of servers in a ring combined with reliable failure detection available in clusters. Applying the same the same design principles allows us to build an even stronger abstraction: total order broadcast. This abstraction can be used for state machine replication. Roughly speaking, replication is about maintaining several copies of the same software object on different machines (also called replicas or processes), such that, if one or more replicas fail, enough replicas remain to guarantee acces-

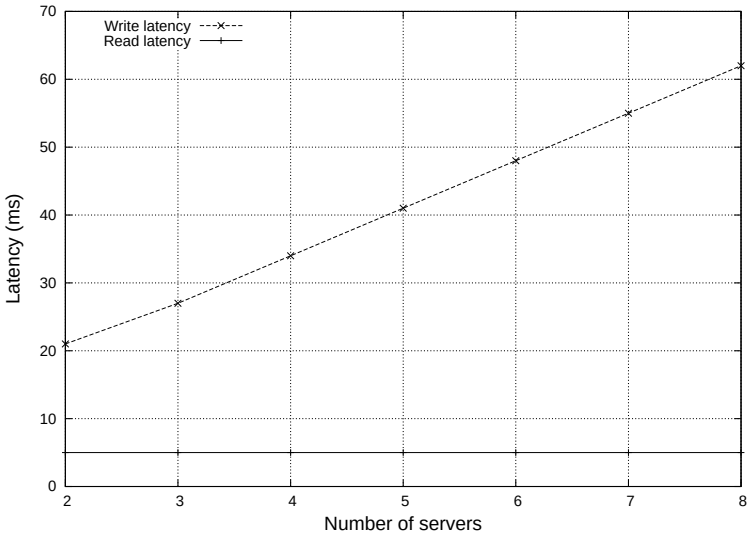

Figure 4.9: Read and write latency.

sibility to the object. The key to making state machine replication work is a well designed software layer that hides all the difficulties behind replication from the application developer and renders it transparent to the clients.

Replication relies on an underlying ordering mechanism which ensures that all replicas perform the same operations on their copy in the same order, even if they subsequently fail. This mechanism is encapsulated by a communication abstraction called *uniform total order broadcast* (UTO-broadcast) [Hadzilacos and Toueg 1993]. This abstraction ensures the following for all messages that are broadcast: (1) *Uniform agreement*: if a replica delivers a message m, then all correct processes eventually deliver m; (2) *Strong uniform total order*: if some replica delivers some message m before message m', then a replica delivers m' only after it has delivered m. Note that uniformity prevents faulty replicas from performing operations on their copy that will not be performed by the other (correct) replicas.

Throughput can be defined as the average number of completed UTO-broadcasts per round. A complete UTO-broadcast of message m meaning that all processes UTO-delivered m. In this model a UTO-broadcast algorithm is optimal if it achieves an average of one complete UTO-broadcast per round regardless of the number of broadcasters. Considering a cluster with n processes, we want the throughput to be optimal with k simultaneous broadcasters, k ranging from 1 to n.

4.6.1 Related Work

The five following classes of UTO broadcast algorithms were identified in [Défago et al. 2004]: fixed-sequencer, moving sequencer, privilege-based, communication history and destination agreement. In this section, we only survey time-free algorithms, i.e. algorithms that do not rely on physical time, since these are

the ones comparable to our LCR algorithm (which do not assume synchronized clocks).

Fixed Sequencer

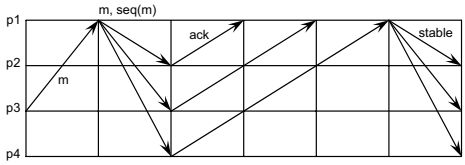

Figure 4.10: Fixed sequencer-based uto-broadcast.

In a fixed sequencer algorithm [Kaashoek and Tanenbaum 1996; Armstrong et al. 1992; Carr 1985; Garcia-Molina and Spauster 1991; Birman and van Renesse 1993; Wilhelm and Schiper 1995] (Figure 4.10), a single process is elected as the sequencer and is responsible for the ordering of messages. The sequencer is unique, and another process is detected as a new sequencer only in the case of sequencer failure. Three variants of the fixed sequencer algorithm exist [Baldoni et al. 2006] each using a different communication pattern. Fixed sequencer algorithms exhibit linear latency with respect to n [Défago et al. 2003], but poor throughput. The sequencer becomes a bottleneck because it must receive the acknowledgments (acks) from all processes[3] and also receives all messages to be broadcast. Note that this class of algorithms is popular for *non*-uniform total order broadcast algorithms since these do not require all processes to send acks back to the sequencer, thus providing much better latency and throughput.

Moving Sequencer

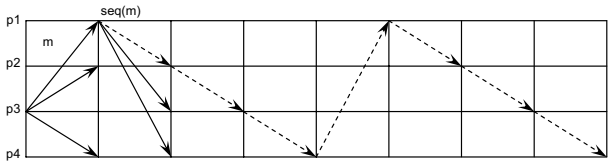

Figure 4.11: Moving sequencer-based uto-broadcast.

Moving sequencer algorithms [Chang and Maxemchuk 1984; Whetten et al. 1994; Kim and Kim 1997; Cristian et al. 1997] (Figure 4.11) are based on the same principle as fixed sequencer algorithms, but allow the role of the sequencer to be passed from one process to another (even in failure-free situations). This is achieved by a token which carries a sequence number and constantly circulates among the processes. The motivation is to distribute the load among sequencers, thus avoiding the bottleneck caused by a single sequencer. When a process p wants to broadcast a message m, it sends m to all other processes. Upon receiving

[3] Acknowledgments in the fixed sequencer can only be piggy-backed when all processes broadcast messages all the time [Défago et al. 2003].

m, the processes store m into a *receive* queue. When the current token holder q has a message in its *receive* queue, q assigns a sequence number to the first message in the queue and broadcasts that message together with the token. For a message m to be delivered, m has to be acknowledged by all processes. Acks are gathered by the token. Moving sequencer algorithms have a latency that is worse than that of fixed sequencer algorithms [Défago et al. 2004]. On the other hand, these algorithms achieve better throughput, although not optimal. Figure 4.11 depicts a 1-to-n broadcast of one message. It is clear from the figure that it is impossible for the moving sequencer algorithm to deliver one message per round. The reason is that the token must be received at the same time as the broadcast messages and the algorithm thus cannot achieve optimal throughput. Note that fixed sequencer algorithms are often preferred to moving sequencer algorithms because they are much simpler to implement [Défago et al. 2004].

Privilege-based Algorithms

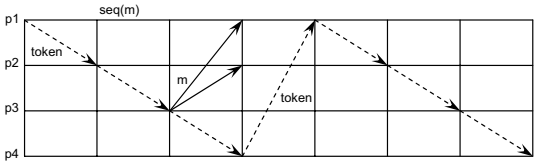

Figure 4.12: Privilege-based uto-broadcast.

These algorithms [Friedman and Renesse 1997; Cristian 1991; Ekwall et al. 2004; Amir et al. 1995; Gopal and Toueg 1989] (Figure 4.12) rely on the idea that senders can broadcast messages only when they are granted the privilege to do so. The privilege to broadcast (and order) messages is granted to only one process at a time, but this privilege circulates from process to process in the form of a token. Because at any given time there is only one process which can send messages, the throughput when all processes broadcast cannot be higher than when only one process broadcasts.

Communication History-based Algorithms

As in privilege-based algorithms, communication history-based algorithms [Peterson et al. 1989; Malhis et al. 1996; Ezhilchelvan et al. 1995; Ng 1991; Moser et al. 1993] use sender-based ordering of messages. Nevertheless, they differ by the fact that processes can send messages at any time. Messages carry logical clocks that allow processes to observe the messages received by the other processes in order to learn when delivering a message does not violate the total order. Communication history-based algorithms have poor throughput because they rely on a quadratic number of messages exchanged for each message that is broadcast.

Destination Agreement

In destination agreement algorithms, the delivery order results from an agreement between destination processes. Many such algorithms have been proposed [Chan-

dra and Toueg 1996a; Birman and Joseph 1987b; Luan and Gligor 1990; Fritzke et al. 2001; Anceaume 1997]. They mainly differ by the subject of the agreement: message sequence number, message set, or acceptance of a proposed message order. These algorithms have relatively bad performance because of the high number of messages that are generated for each broadcast.

Note that hybrid algorithms, combining two different ordering mechanisms have also been proposed [Ezhilchelvan et al. 1995; Rodrigues et al. 1996; Vicente and Rodrigues 2002]. Most of these algorithms are optimized for large scale networks instead of clusters, making use of multiple groups or optimistic strategies.

4.7 The LCR Algorithm

LCR combines a ring topology for dissemination with logical (vector) clocks for message ordering. When a broadcast is initiated, the message is forwarded around the ring (clockwise) until all processes receive the message. Each process maintains an ordered list, called pending to store received messages. Indeed, processes cannot deliver a message as soon as they receive it: if they crash after delivering a message without forwarding it, uniformity could be violated. Thus, before delivering a message, the processes must make sure that the message is stable, i.e. all other processes received it. To that end an acknowledgement is sent around the ring by the predecessor of the broadcast initiator. Upon receiving this ack, processes set the corresponding message stored in the pending list to stable. Whenever the first message in a pending list is stable, it is delivered.

4.7.1 Ordering

Each process $p_i \in \mathbb{P}$ has a logical vector clock $\mathcal{C}_{p_i} = (c_k)_{k=[0..n]}$. Let $\mathbb{M}$ be the set of messages that have been broadcast by all processes. We note $\mathcal{M}^{\mathcal{C}}_{p_i}$, a message sent by process p_i with vector clock $\mathcal{C} = (c_k)_{k=[0..n]}$. We note $\mathcal{C}_m[i]$ the i^{th} value of the vector clock carried by m. The total order on messages in $\mathbb{M}$ is defined as follows:

Definition 1 (Total order) *Let* $m = \mathcal{M}^{\mathcal{C}}_{p_i}$ *and* $m' = \mathcal{M}^{\mathcal{C}}_{p_j}$ *s. t.* $m, m' \in \mathbb{M}$

$$m \prec m' \Leftrightarrow \begin{cases} \mathcal{C}_m[i] \leq \mathcal{C}_{m'}[i] & \text{if} \quad i \leq j \\ \mathcal{C}_m[j] < \mathcal{C}_{m'}[j] & \text{if} \quad i > j \end{cases}$$

4.7.2 Operating Principle

The pseudo-code of the LCR protocol is depicted in Figure 4.13. The operating principle of the protocol is the following:

- upon reception of message m, a process updates its local logical clock by taking, for each index i, the maximum of its local value and the value of the clock carried by m.
- before sending a *new* message m, a process p_i increments the value stored at index i in its local vector clock; it then timestamps m with its logical

```
Procedures executed by any process p_i
 1: procedure initialize(initial_view)
 2:    pending_i ← ∅                                          {pending list}
 3:    C[p_1 ... p_n] ← {0, ..., 0}                           {local vector clock}
 4:    view ← initial_view
 5: end

 6: procedure utoBroadcast(m)
 7:    C[p_i] ← C[p_i] + 1
 8:    pending ← pending ∪ [m, p_i, C, ⊥]
 9:    Rsend ⟨m, p_i, C⟩ to successor(p_i, view)              {broadcast a message}
10: end

11: upon Rreceive ⟨m, p_j, C_m⟩ do
12:    if C_m[p_j] > C[p_j] then
13:       if p_i ≠ predecessor(p_j, view) then
14:          Rsend ⟨m, p_j, C_m⟩ to successor(p_i, view)     {forward the message}
15:          pending ← pending ∪ [m, p_j, C_m, ⊥]
16:       else
17:          pending ← pending ∪ [m, p_j, C_m, stable]       {m_j is stable}
18:          Rsend ⟨ACK, p_j, C_m⟩ to successor(p_i, view)   {send an ACK}
19:          tryDeliver()
20:       end if
21:       ∀k ∈ [1, n] : C[p_k] ←max(C[p_k], C_m[p_k])       {update local vector clock}
22:    end if
23: end upon

24: upon Rreceive ⟨ACK, p_j, C_m⟩ do
25:    if p_i ≠ predecessor(predecessor(p_j), view) then
26:       pending[C_m] ← [∗, ∗, ∗, stable]                   {m_j is stable}
27:       Rsend ⟨ACK, p_j, C_m⟩ to successor(p_i, view)      {forward the ACK}
28:       tryDeliver()
29:    end if
30: end upon

31: upon Rreceive ⟨RECOVER, m, p_j, C_m⟩ do
32:    pending ← pending ∪ [m, p_j, C_m, ⊥]
33:    Rsend ⟨ACK, p_j, C_m⟩ to successor(p_i, view)
34: end upon

35: procedure tryDeliver()
36:    while pending.first = [m, p_k, C_m, stable] do
37:       utoDeliver (m)                                     {deliver a message}
38:       pending ← pending − [m, p_k, C_m, stable]
39:    end while
40: end
```

Figure 4.13: Pseudo-code of the LCR protocol.

clock. Once set, the logical clock of a message is never changed. Note that no two messages have the same timestamp.

A process p can deliver a message m when it knows that m reached every process and when all messages that are before m in its pending list can be delivered. Intuitively the protocol works because the timestamps are unique and define a total order on the messages. If a process delivers a message, all non-crashed processes have received the message with the same timestamp. Therefore, even if a process delivers a message and crashes directly afterwards we are sure that all other correct processes will eventually deliver the message too.

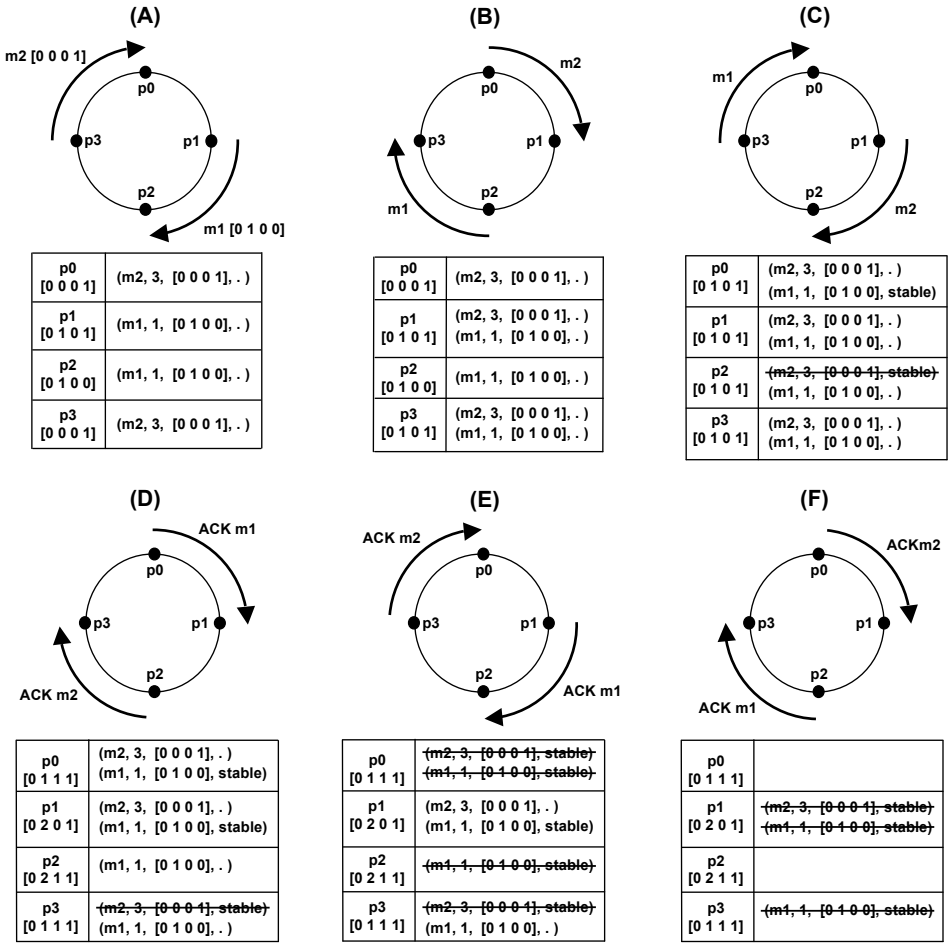

Figure 4.14: Illustration of a run of the LCR protocol with 4 processes.

Figure 4.14 illustrates a run of the LCR protocol. The state of the pending list of each process is represented in the second column of the table. For simplicity of presentation, consider that the computation proceeds in rounds. Round (A) consists in the broadcast of messages m_1 and m_2 by processes p_1 and p_3, respectively. Then at round (B), process p_1 receives message m_2 and adds it at the head of its pending list to respect the total order relation given by Definition 1. At the end of round (C), m_1 (resp. m_2) has done a full round along the ring. As a consequence

p_0 (resp. p_2) sets the message to stable. Moreover, p_2 delivers m_2 since it is first in its pending list, which ensures that it already delivered all messages preceding m_2. At the start of round (D), processes p_0 and p_2 send an ACK for messages m_1 and m_2, respectively. Upon reception of these ACK messages, processes p_3, p_0 and p_1 (resp. p_1, p_2 and p_3) set m_2 (resp. m_1) to stable. Message deliveries occur each time that the head of a pending list becomes a stable message.

4.7.3 Group Membership Changes

Our LCR protocol is built on top of a group communication system which provides virtually synchronous communications (VSC) [Birman and Joseph 1987a]. According to the virtual synchrony programming model, the processes are organized into groups. Processes can join and leave the group using the appropriate primitives. Faulty processes are excluded from the group after crashing. Upon a membership change, processes agree on a new view by using a view change protocol.

When a process joins or leaves the group, a *view_change* event is generated by the VSC layer and the current view v_r is replaced the new view v_{r+1}. This can happen when a process crashes or when a process actively wants to leave or join the group. As soon as a new view is installed it becomes the basis for the new ring topology.

The *view_change* procedure is detailed in Figure 4.15. Note that when a view change occurs, every process cancels the execution of all other procedures. The view change procedure works as follows: every process sends its pending to all other processes. Upon receiving this list every process adds to its pending list the messages it did not yet receive. They then send back an ACK_RECOVER message. Processes wait until they receive ACK_RECOVER messages from all processes before sending an END_RECOVERY message to all. When a process receives END_RECOVERY messages from all processes it can deliver all the messages in its pending list. Thus, at the end of the recovery procedure all pending lists have been emptied which guarantees that all messages from the old view have been handled.

4.7.4 Correctness

Let $\mathbb{M}_{p_i}$ be the set of messages that have been stored in the pending list of process p_i. In the correctness proof of the protocol, we use a local ordering relationship defined on messages in $\mathbb{M}_{p_i}$ as follows:

Definition 2 (Local order) *Let* $m = \mathcal{M}^{\mathcal{C}}_{p_j}$ *and* $m' = \mathcal{M}^{\mathcal{C}}_{p_k}$ *s. t.* $m, m' \in \mathbb{M}_{p_i}$

$$m <_{p_i} m' \Leftrightarrow m \text{ has been stored in pending before } m'$$

Moreover, we note $[i \frown j]$ the set of integers defined as follows:

$$[i \frown j] = \begin{cases} [i, j] & \text{if} \quad i < j \\ [i, n] \cup [0, j] & \text{otherwise} \end{cases}$$

The communication channels are assumed to be FIFO, which can be formally expressed as follows:

Procedures executed by any process p_i

1: **upon** view_change(new_view) **do**
2: Rsend $\langle$RECOVER$, p_i, pending\rangle$ to all $p_j \in \mathbb{P}$
3: Wait until received $\langle$ACK_RECOVER$\rangle$ from all $p_j \in \mathbb{P}$
4: Rsend $\langle$END_RECOVERY$\rangle$ to all $p_j \in \mathbb{P}$
5: Wait until received $\langle$END_RECOVERY$\rangle$ from all $p_j \in \mathbb{P}$
6: forceDeliver()
7: $view \leftarrow new\_view$
8: **end upon**

9: **upon** Rreceive $\langle$RECOVER$, p_j, pending_{p_j}\rangle$ **do**
10: **for each** $[m, p_l, \mathcal{C}_m, *] \in pending_{p_j}$ **do**
11: **if** $\mathcal{C}_m[p_l] > \mathcal{C}[p_l]$ **then**
12: $pending \leftarrow pending \cup [m, p_l, \mathcal{C}_m, \perp]$
13: **end if**
14: **end for**
15: Rsend $\langle$ACK_RECOVER$\rangle$ to p_j
16: **end upon**

17: **procedure** forceDeliver()
18: **for each** $[m, p_k, \mathcal{C}_m, *] \in pending$ **do**
19: utoDeliver (m) {*deliver a message*}
20: $pending \leftarrow pending - [m, p_k, \mathcal{C}_m, *]$
21: $\forall j \in [1, n] : \mathcal{C}[p_j] \leftarrow \max(\mathcal{C}[p_j], \mathcal{C}_m[p_j])$ {*update local vector clock*}
22: **end for**
23: **end**

Figure 4.15: Pseudo-code of the view_change procedure.

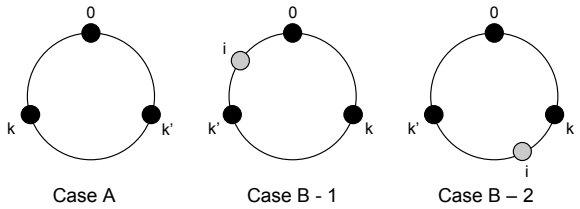

Figure 4.16: The different cases studied in the correctness proof.

Definition 3 (FIFO) *Let $m = \mathcal{M}_{p_i}$ and $m' = \mathcal{M}_{p_j}$ be two messages and let p_k be a process.*

$$k \in [i \frown j[\Rightarrow \begin{cases} m <_{p_k} m' \Rightarrow \forall\ l \in [i \frown j[\ m <_{p_l} m' & (3.1) \\ m' <_{p_k} m \Rightarrow \forall\ l \in [0 \frown n]\ m' <_{p_l} m & (3.2) \end{cases}$$

$$k \in [j \frown i[\Rightarrow \begin{cases} m <_{p_k} m' \Rightarrow \forall\ l \in [0 \frown n]\ m <_{p_l} m' & (3.3) \\ m' <_{p_k} m \Rightarrow \forall\ l \in [j \frown i[\ m' <_{p_l} m & (3.4) \end{cases}$$

Lemma 11 *Let $m = \mathcal{M}_{p_i}$ and $m' = \mathcal{M}_{p_j}$ be two messages.*

$$m \prec m' \Rightarrow \begin{cases} m <_{p_j} m' & \text{if} \quad i \leq j & (11.1) \\ m <_{p_i} m' & \text{otherwise} & (11.2) \end{cases}$$

Proof: If $i = j$, we have $\mathcal{C}_m[i] \leq \mathcal{C}_{m'}[i]$ thus m was sent before m' (Line 7 of Figure 4.13). If $i < j$, we have $\mathcal{C}_m[i] \leq \mathcal{C}_{m'}[i]$. Since the local clock at process p_j is only updated when a new message is received (Line 21 of Figure 4.13), we know that p_j received m before sending m'. Thus, we have $m <_{p_j} m'$.
If $i > j$, we have $\mathcal{C}_m[j] < \mathcal{C}_{m'}[j]$. Since the local clock at process p_i is updated whenever a new message is received (Line 21 of Figure 4.13), we know that p_i did not receive m' before sending m. Thus, we have $m <_{p_i} m'$.

Lemma 12 (Stable) *When a process p_i sets m to* stable*, the $\mathbb{M}_{p_i}$ set contains all the messages m' such that $m' \prec m$.*

Proof: Let $m = \mathcal{M}_{p_k}$ be a message and let p_i be a process setting m to stable by executing line 17 or 26 of Figure 4.13. Let us suppose that there exists a message $m' = \mathcal{M}_{p_{k'}}$ such that $m' \neq m$, $m' \prec m$ and $m' \notin \mathbb{M}_{p_i}$.

- **Case A:** $k' \leq k$

  We have $m' \prec m$ and $k' \leq k$. It follows that $m' <_{p_k} m$ (Lemma 11.1). Moreover, $\forall l \in [0 \frown n]\ m' <_{p_l} m$ (Lemma 3.3). In particular, $m' <_{p_i} m$. This contradicts the fact that $m' \notin \mathbb{M}_{p_i}$ when m is set to stable.

- **Case B:** $k' > k$

  - **Subcase 1:** $i \in [k' \frown k[$

    We have $m' \prec m$ and $k' > k$. It follows that $m' <_{p_{k'}} m$ (Lemma 11.2). Moreover, $\forall l \in [k' \frown k[\ m' <_{p_l} m$ (Lemma 3.1). In particular,

$m' <_{p_i} m$. This contradicts the fact that $m' \notin \mathbb{M}_{p_i}$ when m is set to stable.

– **Subcase 2:** $i \in [k \frown k'[$
We have $k \neq k'$, which implies that $p_i \neq predecessor(k)$. It follows that p_i must receive an ACK for m before setting it to stable. Moreover, $m' \prec m$ and $k' > k$. It follows that $m' <_{p_{k'}} m$ (Lemma 11.2). Thus, $m' <_{p_i} ACK(m)$. This contradicts the fact that $m' \notin \mathbb{M}_{p_i}$ when m is set to stable.

Lemma 13 *Every correct process that stores a message m in pending eventually* utoDelivers *it.*

Proof: Let $m = \mathcal{M}_{p_k}$ be a message and let p_i be a process that set m to stable. By Lemma 12, the $\mathbb{M}_{p_i}$ set contains *all* the messages m' such that $m' \prec m$. Thus, the pending list starts with the (possibly empty) set of *all* messages m' such that $m' \prec m$ and that have not yet been delivered by p_i. Let us call *undelivered* the ordered list containing these messages. Moreover, note that, the *undelivered* set cannot grow. Otherwise, this would contradict Lemma 12.
Let $m' = \mathcal{M}_{p_{k'}}$ be the first message in *undelivered*. The protocol ensures that m' will eventually be delivered by process p_i. Indeed, if there is a membership change, the recovery procedure guarantees that all messages (including m and m') will be received (Line 2 of Figure 4.15) and delivered (Line 6 of Figure 4.15) by all correct processes. If, on the other hand, there is no membership change, m' will be eventually set to stable (Line 26 of Figure 4.13) and subsequently delivered (Line 28 of Figure 4.13). The same reasoning applies to other messages in *undelivered*. Consequently, message m will eventually come first in the pending list and be delivered.

Lemma 14 *If a process p_i* utoDelivers *a message m before* utoDelivering *a message m', then $m \prec m'$.*

Proof: We prove the lemma by contradiction. Let $m = \mathcal{M}_{p_k}$ and $m' = \mathcal{M}_{p_{k'}}$ be two messages such that $m' \prec m$ and let p_i be a process that utoDelivers m before Delivering m'. If there is no membership change before p_i delivered m, then m was set to stable before being delivered. The fact that $m' \prec m$ implies that when p_i sets m to stable, $m' \in \mathbb{M}_{p_i}$ (Lemma 12), which contradicts the fact that p_i delivered m before m'. If, on the other hand, there is a membership change before p_i delivered m, then the protocol ensures that m and m' will have been delivered by all processes before any of them deliver a message (Lines 2 and 5 of Figure 4.15). Indeed, we are sure that m' was broadcast before the membership change. Otherwise, its timestamp would be greater than m's timestamp (Line 21 of Figure 4.15). Thus, p_i will deliver m' before m.

Theorem 4 (Validity) *If any correct process p_i* utoBroadcasts *a message m, then it eventually* utoDelivers *m.*

PROOF: Let p_i be a process and let $m = \mathcal{M}_{p_i}$ be a message broadcast by p_i. This message is added to p_i's pending list (Line 8 of Figure 4.13). By Lemma 13, we know that m will eventually be delivered by p_i.

Theorem 5 (Uniform Agreement) *If any process p_i* utoDelivers *any message m in the current view, then every correct process p_j in the current view eventually* utoDelivers *m.*

PROOF: Let $m = \mathcal{M}_{p_k}$ be a message and let p_i be a process that delivered m in the current view. The protocol ensures that m has been received by all correct processes in the current view. Indeed, if there is a membership change, the recovery procedure guarantees that all messages (including m) will be received (Line 2 of Figure 4.15) and delivered (Line 6 of Figure 4.15) by all correct processes in the current view. If, on the other hand, there is no membership change, m did a full round around the ring before being delivered by p_i. Thus, every correct process p_j in the current view either delivered m or stored it in its pending list. Consequently, we know (Lemma 13) that every correct process p_j in the current view will eventually deliver m.

Theorem 6 (Uniform Integrity) *For any message m, any process p_j that* utoDelivers *m,* utoDelivers *m at most once, and only if m was previously* utoBroadcast *by some process p_i.*

PROOF: By the absence of Byzantine failures, no spurious message is ever utoDelivered by a process. Thus, only messages that have been utoBroadcast are utoDelivered. Moreover, each process keeps a vector clock $\mathcal{C}$, which is updated in such a way that we are sure that every message is only delivered once. Indeed, if there is no membership change, Lines 12 and 21 of Figure 4.13 guarantee that no message can be stored in p_j's pending list twice. Similarly, when there is a membership change, Line 11 of Figure 4.15 guarantees that process p_j will not add to its pending list messages it delivered before the membership change. Moreover, Line 21 of Figure 4.15 guarantees that p_j's vector clock is updated after the membership change, thus preventing the future delivery of messages that have been delivered during the *view_change* procedure.

Theorem 7 (Uniform Total Order) *For any two messages m and m', if any process p_i* utoDelivers *m without having delivered m', then no process p_j* utoDelivers *m' before m.*

PROOF: We prove the lemma by contradiction. Let $m = \mathcal{M}_{p_k}$ and $m' = \mathcal{M}_{p_{k'}}$ be two messages and let p_i be a process that utoDelivers m without having delivered m'. Let us suppose that there exists a process p_j that utoDelivers m' before delivering m. This implies that $m' \prec m$ (Lemma 14). If there is no membership change before p_i delivered m, the protocol ensures that when p_i sets m to stable, it knows m' (Lemma 12). This contradicts the fact that p_i delivered m without delivering m'. If, on the other hand, there is a membership change before p_i delivered m, then m' is delivered by p_j during the view_change procedure. Otherwise,

p_i would have known m' before delivering m, contradicting the fact that p_i delivered m without delivering m'. Nevertheless, m and m' cannot have both been delivered during the view_change procedure. Otherwise, they they would both have been in the pending list of p_i (Lines 2 and 5 of Figure 4.15), contradicting the initial assumption.

Theorem 8 *LCR is a uniform total order broadcast protocol.*

PROOF: By lemmas 4, 6, 5, and 7, we know that the LCR protocol ensures validity, integrity, uniform agreement, and uniform total order. Thus, it is a uniform total order broadcast protocol.

4.8 LCR Performance

This section analyzes several key aspects of LCR's performance, both from a theoretical and a practical perspective. The performance of LCR is evaluated in failure free runs which we expect to be the most common case. The theoretical analysis proves that LCR is throughput optimal. However, in practice, several improvements need to be made in order to ensure a fast and fair implementation for all broadcasters.

4.8.1 Experimental Setup

The experiments were run on a cluster of machines with dual 900MHz Itanium-2 processors, 3GB RAM, Fast Ethernet adapter running Linux kernel 2.4.21-32 SMP. The raw latency and bandwidth over IP are measured with Netperf [Jones 2007] between two machines and displayed in Table 4.1. The LCR algorithm is implemented in C (1000 lines of code) and relies on the Spread toolkit to provide a group membership layer.

Protocol	Bandwidth
TCP	94 Mbit/s
UDP	93 Mbit/s

Table 4.1: Raw network performance measured using Netperf.

4.8.2 Throughput

Theoretical analysis

We propose to evaluate message-passing algorithms in a synchronous round-based model assuming that in each round k, every process p_i can execute the following steps:

1. p_i computes the message for round k, $m(i,k)$,
2. p_i sends $m(i,k)$ to all or a subset of processes and

3. p_i receives at most one message sent at round k.

The synchrony assumption implies that, at any time, all processes are in the same round. The *throughput* of a broadcast system refers to the number of completed broadcasts per round. We say that a broadcast is complete when all processes have delivered the message that was broadcast. We can now determine a bound on the throughput of broadcast algorithms:

Theorem 9 (Maximum throughput) *For a broadcast algorithm in a system with n processes in the round-based model introduced above, the maximum throughput μ_{max} in completed broadcasts per round is:*

$$\mu_{max} = \begin{cases} n/(n-1) & \textit{if there are } n \textit{ senders} \\ 1 & \textit{otherwise} \end{cases}$$

PROOF: We first prove the case with n senders. Each broadcast message must be received at least $n-1$ times in order to be delivered. The model states that at each round at most n messages can be received. Thus, for n processes to broadcast a message a minimum of $n-1$ rounds are necessary. Therefore, on average, at most $n/(n-1)$ broadcasts can be completed each round. In the case with less than n senders it is sufficient to look at a non sending process. Such a process can receive at most 1 message per round and since it doesn't broadcast any messages itself, it can deliver at most 1 message per round. Since the throughput is defined as the number of *completed* broadcasts per round, the maximum throughput with less than n senders is 1.

Determining the throughput of LCR is straight forward: processes receive one message per round and the acknowledgements are piggy-backed. Thus LCR allows each process to deliver one message per round if there is at least one sender. When there are n senders, each process can deliver one message per round broadcast by other processes in addition to its own messages. LCR thus matches the bound of Theorem 9 and is theoretically throughput optimal.

Evaluation

We now evaluate the performance of the LCR implementation. The benchmarks test k-to-n broadcasts, where n is the number of processes in the system and k the number of senders. All processes know a priori the number of messages they expect from other processes (each sender sends the same number of messages). A leader process is used to synchronize the experiment start-up. Upon receiving this message from the leader, each process starts sending a burst of messages. Both the message size s_m and burst size s_b are parameters of the experiment. When the last expected message from a sender is received, an acknowledgment is sent back to the leader. The leader then calculates the time t between the start-up message and the last received acknowledgment. The throughput is then calculated as the ratio of delivered bytes over the total experiment time ($(m_s * b_s * k)/t$). We ensure that the acknowledgement latency is negligible compared to the overall experience time. A new average is calculated after each experiment, and the

experiments are repeated a sufficient number of times, such that the variance of these averages is negligible.

Figure 4.17 shows the results of an experiment with $n = 5$ processes [4]. The number k of sending processes was adjusted for each run of the experiment. We can observe that the throughput obtained by our implementation is far from optimal: in practice, a theoretical throughput of 1 should be equal to the raw link speed between the processes, i.e. 94Mbit/s as shown in Table 4.1.

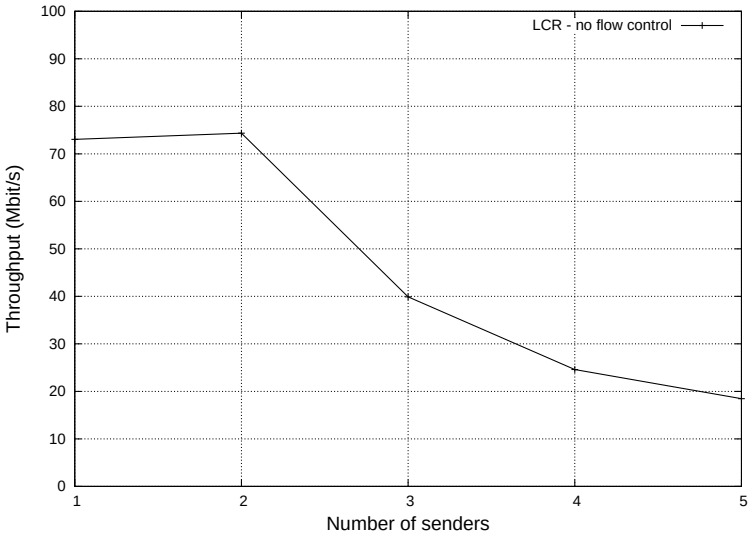

Figure 4.17: LCR throughput without flow control for a system with 5 processes. The lack of flow control quickly saturates the buffers and explains the low throughput.

The reason that the throughput is not optimal is that senders broadcast more messages than can be delivered, resulting in overflowing buffers and costly retransmissions. There is clearly a need to throttle senders to prevent them from injecting too many new messages into the ring. One simple flow control solution is to prevent processes from sending messages as long as they still have messages to forward. As can be seen in Figure 4.18, the throughput clearly improves and is now close to optimal. Also notice that the throughput with 5 senders is higher than with fewer senders, the reason being LCR's theoretical throughput of $n/(n-1)$ when all processes are senders. Note that a better flow control mechanism is presented in Section 4.8.3.

The two previous experiments used a message size of 5 kB. The experiment in Figure 4.19 measures the impact of varying the message size on the throughput of a system with 5 processes and one sender. We can observe that if the messages are too small the throughput suffers. This is due to the cost of ordering and providing uniformity which remains constant despite a decrease in payload size. Small messages however can easily be batched together into bigger messages when the load on the system is high. Identical results were obtained with different

[4] None of the graphs plotted in this section contain error bars due to the fact that the observed variance was negligible.

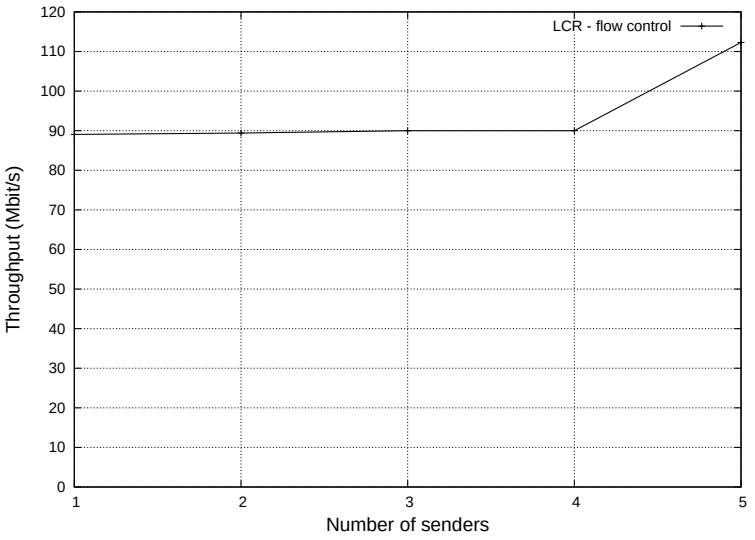

Figure 4.18: LCR throughput with flow control but without fairness in a system with 5 processes. Processes can only send a new message when they have no message to forward. LCR's throughput is close to the optimal of 94 Mbit/s with less than 5 senders and 5/(5-1)*94 = 117 Mbit/s with 5 senders.

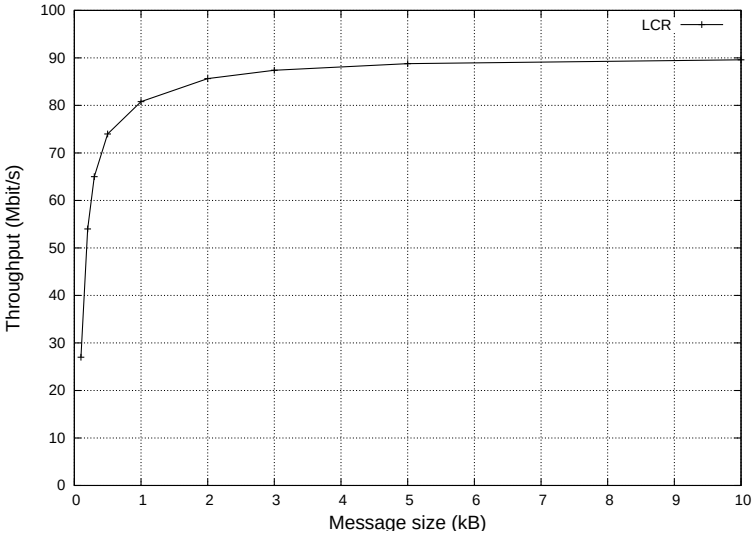

Figure 4.19: Throughput with respect to message size for a system of 5 processes with one sender.

number of processes and senders, thus for all further throughput experiments the message size is set to 5 kB.

4.8.3 Fairness

Even though the implementation of LCR with flow control already provides near optimal performance, there is still a problem. Consider two processes p_1 and p_2 that are neighbors on the ring. If p_1 is continuously broadcasting messages and p_2 systematically forwards p_1's messages, then p_2 cannot broadcast its own messages. Consequently, the algorithm is not *fair*.

Fairness captures the fact that each process has an equal opportunity of having its messages delivered by all processes. Intuitively, the notion of fairness means that in the long run no single process has priority over other processes when broadcasting messages. For instance when two processes want to broadcast a large number of messages, then each process should have approximately the same number of messages delivered by all processes.

Let b_i and d_i be such that:

- $b_i(r)$ is the number of messages process p_i wants to broadcast starting from round r.
- $d_i([r_1, r_2])$ is the number of messages process p_i has delivered during the interval $[r_1, r_2]$.

More formally, we define fairness as follows:

Definition 10 (Fairness) *A broadcast algorithm is* fair *during the interval* $[r_1, r_2]$ *if for the set of processes* $\mathcal{B}$*, such that* $\forall p_i \in \mathcal{B}$*,* $b_i(r_1) \geq (r_2 - r_1)$ *there is an* $l \in \mathbb{N}^+$ *(*$l \ll (r_2 - r_1)$*):*

$$d_i([r_1 + l, r_2]) \leq \left\lceil \frac{\sum_{j \in \mathcal{B}} d_j([r_1 + l, r_2])}{|\mathcal{B}|} \right\rceil$$

The mechanism for ensuring fairness in LCR acts locally at each process. If a process wishes to broadcast a new message, it must decide whether to forward a message received from its predecessor or to send its own. Figure 4.20 provides an illustration of the fairness mechanism as implemented in LCR. Processes put broadcast requests coming from the application level in their send queue. Messages received from predecessors that need to be forwarded are buffered in the forward queue. When there is more than one message in the send queue, the variable burst_nb containing the number of messages in the queue, is added to the first outgoing message. Each process keeps a data structure which stores burst_nb, received and sent for each process. The received variable keeps track of the number of messages that have been received from p_i since p_i's burst_nb has been updated. The sent variable keeps track of the number of messages that have been sent by the process itself since p_i's burst_nb has been updated. So, if the first message in a process' forward queue was originally broadcast by p_j, then if received is higher than sent for p_j, the process can send its own message.

The experiment of Figure 4.21 was performed on a system with 5 processes of which 3 are senders. Similarly to the previous experiments, a leader process is

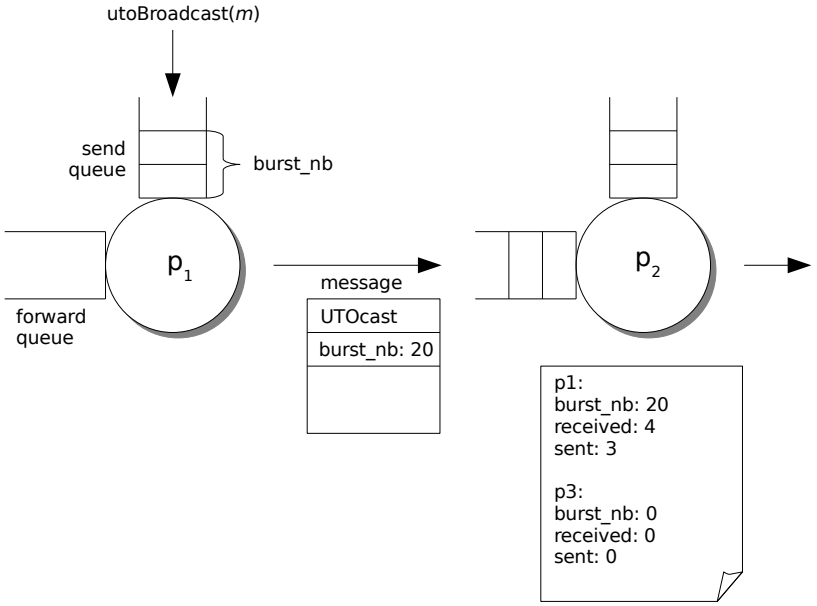

Figure 4.20: Illustration of the fairness mechanism as implemented in LCR. Each process has two queues (send and forward) and uses the burst_nb, received, sent variables to determine whether to forward messages or send its own.

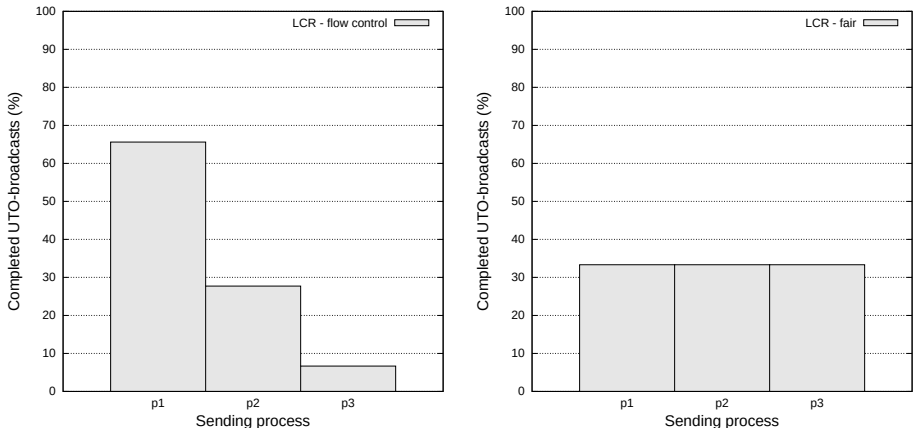

Figure 4.21: LCR without fairness (left) and with fairness (right). Experiments were performed with 5 processes and 3 senders. With the fairness mechanism all processes delivered an equal number of messages from each sender.

used to synchronize the experiment start-up. Upon receiving the start message from the leader, k processes start broadcasting continuously. The experiment is halted after a total of 10'000 messages have been delivered by all processes. For each sender we then calculate the percentage of these 10'000 delivered messages that were initiated by them.

As expected, LCR with only flow control is not fair: the first process in the ring initiated more than 65% of the delivered messages. Thanks to the fairness mechanism we can observe that all processes delivered 33% of messages from each sender. Note that despite the fairness mechanism, the throughput is identical to the LCR implementation with just flow control evaluated in Figure 4.18. All further experiments in the section are performed with the fair implementation of LCR.

4.8.4 Latency

The theoretical latency of broadcasting a single message is defined as the number of rounds that are necessary from the initial broadcast of message m until the last process delivers m. The latency of LCR is equal to $2n - 2$ rounds.

Figure 4.22 plots the latency without contention as a function of the number of processes. The experiment consists in 1-to-n casts of 1KB messages. Each delivered message is acknowledged by the last process in the ring delivering it. Moreover, there is a long period of inactivity in between each initiated broadcast. The represented latency is the average of a large number of experiments. The observed variance was below 3%.

The graph shows that the latency is linear with respect to the number of processes. However it is clear from the graph that the observed latency is closer to $n - 1$ than $2n - 2$. This is due to the fact that the latency of transmitting ack messages is small compared to the latency of the payload messages.

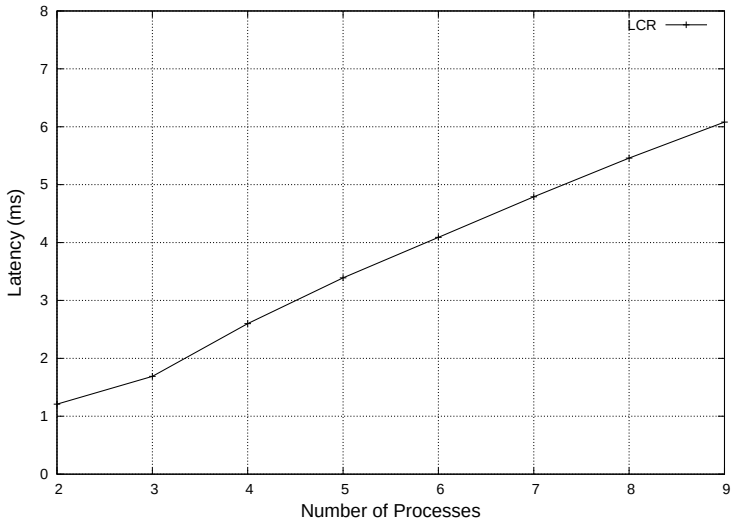

Figure 4.22: Latency as a function of the number of processes with a message size of 1KB and 1 sender.

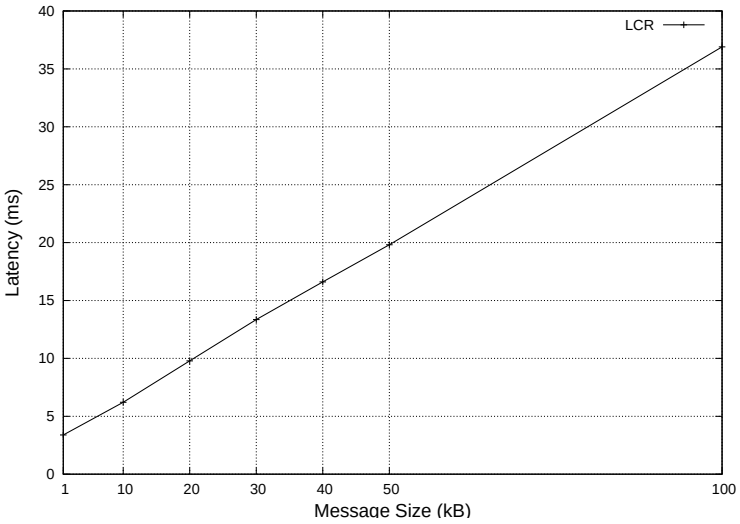

Figure 4.23: Latency as a function of the message size with 5 processes and 1 sender.

Figure 4.23 plots the latency without contention as a function of the message size. The experiment is the same as the previous one, except with 5 processes and a varying message size. The graph shows that the latency is linear with respect to the message size. Identical behavior was observed with different system sizes.

4.9 LCR Performance Comparison

In this section the implementations of LCR are benchmarked against Spread and JGroups, both industry standard group communication systems. The results show that LCR provides better throughput with all numbers of processes and senders. In fact, the benchmark results clearly show that our implementations are optimal. We also check that LCR's CPU consumption is reasonable.

4.9.1 Benchmarked Systems

We benchmarked four different systems:

- *Spread.* We used Spread version 4.0 [Amir et al. 2004] which we recompiled for the IA64 platform. The message type was set to `SAFE_MESS` which guarantees uniform total order. A Spread daemon was deployed on each machine. All daemons belong to the same Spread segment. Spread was tuned for bursty traffic according to Section 2.4.3 of the Spread user guide [Stanton 2002]. Our benchmark uses the native C API provided by Spread.

- *JGroups TCP.* We used JGroups version 2.5.1 [Ban 2007a] with BEA jrockit-R27.3.1 Java 5.0 JDK for the IA64 platform. The stack contains the following protocols: `TCP, MPING, FD_SOCK, VERIFY_SUSPECT, pbcast.NAKACK, pbcast.STABLE, pbcast.GMS, FC`. This stack provides only non uniform FIFO ordering and is the same as the one used by JGroups

developers in their most recent benchmarks [Ban 2007b]. We did not benchmark the existing total order implementations as they are known to be unstable. Moreover, we do not present results with a UDP based stack because of the bad obtained results, confirmed by other studies [Abdellatif et al. 2004].

- *LCR.* We benchmarked the fair implementation of LCR described in Section 4.8.3.
- *LCR JGroups.* We implemented a version of LCR for the JGroups framework. The resulting stack is the same as the JGroups TCP stack with the addition of the LCR algorithm but without the `FC` flow control algorithm since flow control is already implemented by LCR.

4.9.2 Throughput

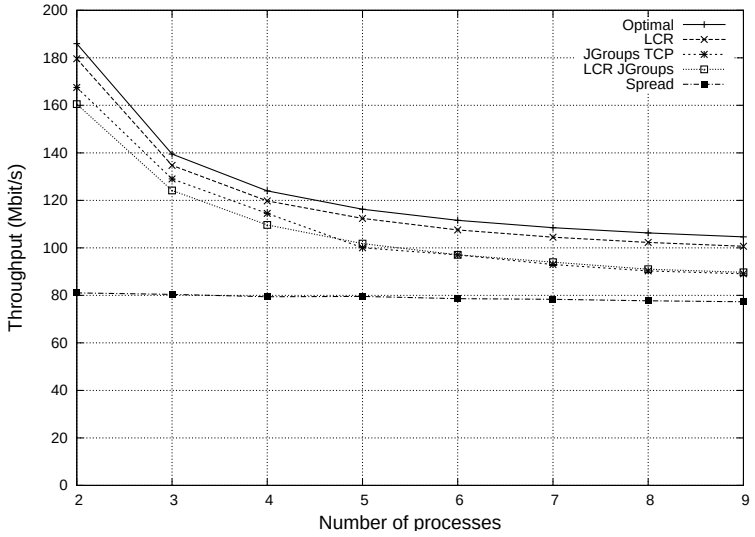

Figure 4.24: n-to-n throughput comparison. The optimal line is calculated as $n/(n-1) * 94$ Mbit/s. The LCR implementations closely follow the optimal line as does JGroups. JGroups however does not provide uniformity or ordering. Spread's throughput is limited by the underlying privilege-based broadcast scheme.

Figure 4.24 plots the throughput as a function of the number of processes. The experiment consists in n-to-n broadcasts of 5 KB messages. The graph shows the performance of Spread, LCR, LCR on JGroups and JGroups using TCP. The performance of JGroups using UDP is not represented because of the highly variable results in the n-to-n case [Ban 2007a]. The optimal line for best effort broad cast ($n/(n-1)$ times the maximum link speed of 94 Mbit/s) is plotted as a reference.

The following observations can be made:

- The throughput of LCR is very close to optimal. Since the optimality line is calculated for best effort broadcast and LCR provides stronger uniform

total order broadcast, the ordering, reliability and uniformity properties of LCR are effectively free.

- The JGroups TCP algorithm provides the same throughput in the n-to-n case as LCR but does not provide ordering or uniformity. This can be explained by the fact that with JGroups TCP each process sends $n-1$ unicast messages for each initiated broadcast. Consequently, as in LCR, the optimal $n/(n-1)$ throughput bound is reached.
- The implementation of LCR on JGroups provides the same throughput in the n-to-n case as JGroups TCP, which confirms the fact that the ordering and uniformity properties of LCR are free.
- The throughput of Spread is constant because of the underlying privilege based scheme (see Section 4.6.1).

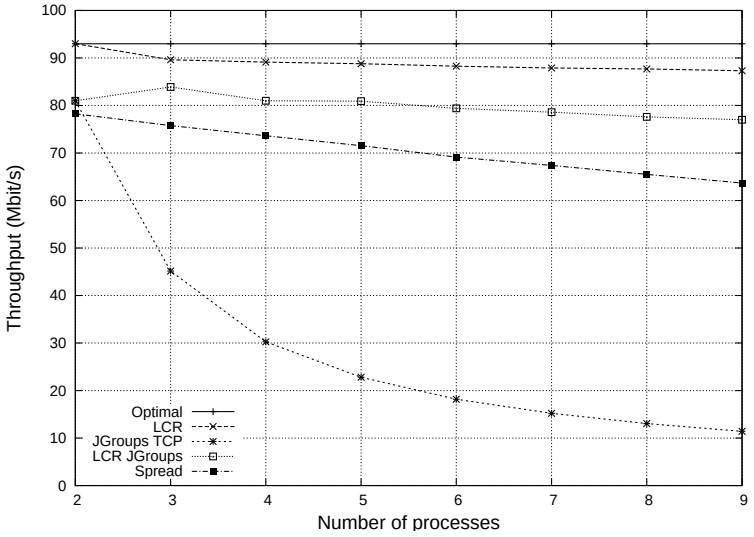

Figure 4.25: 1-to-n throughput comparison. The optimal line is constant at 94 Mbit/s. In JGroups TCP the sender sends messages to all processes sequentially, thus explaining the rapid drop off in throughput.

The next experiment consists in 1-to-n broadcasts of 5 KB messages. Again LCR is close to optimal and is almost constant despite the increasing number of processes. With $n = 2$ processes, LCR's throughput is exactly equal to the optimal of 94 Mbit/s as measured by NetPerf. This can be explained by the fact that apart from acks, the only traffic is from the sender to the other process. Spread's throughput suffers a bit more from increasing the number of processes. The bad performance of JGroups TCP can be explained by the fact that as explained before, in JGroups the sender sends unicast messages to all processes sequentially.

The final experiment consists in $\lfloor n/2 \rfloor$-to-n broadcasts of 5 KB messages. Again the LCR implementations are constant with respect to the number of processes. The irregular performance of JGroups TCP is explained by the fact

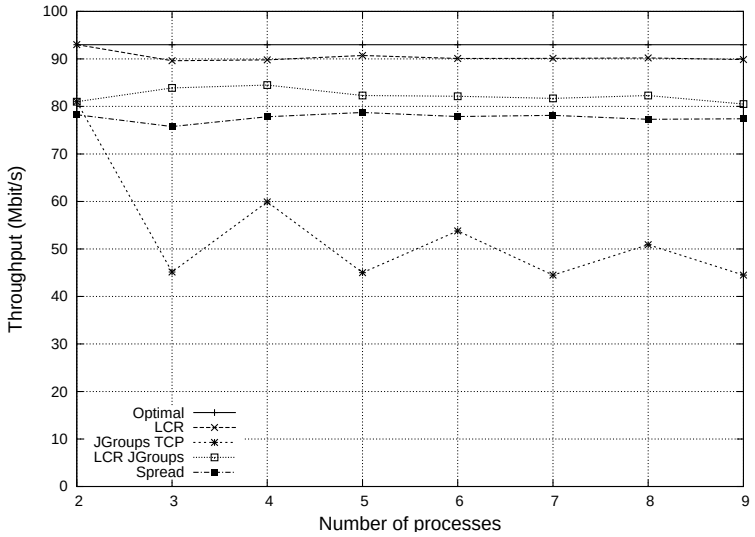

Figure 4.26: $\lfloor n/2 \rfloor$-to-n throughput comparison. The optimal line is constant at 94 Mbit/s. The throughput of JGroups TCP depends on the ratio between the number of processes and the number of senders. For odd and even numbers of processes this ratio is not explaining the observed variations.

that its throughput depends on the ratio between the number of processes and the number of senders. Indeed, since only senders participate in the broadcast, at any time, no more than $\lfloor n/2 \rfloor$ messages can be in transit in the network.

In the following sections, we only compare Spread and LCR since they provide the best throughput.

4.9.3 CPU Usage

The following experiments measure the CPU usage of Spread and LCR under high load. The experiment in Figure 4.27 plots the CPU usage measured during the experiment of Figure 4.24, and Figure 4.28 the CPU usage of Figure 4.25.

During the experiment, the CPU usage of all active LCR or Spread threads was periodically logged, added up together and averaged. The CPU usage was constant over the length of the experiment with very low variance. Varying the number of processes in the system had no impact on the CPU usage and thus experiments with 5 processes are plotted. Note that since the experiments were performed on dual processor machines, a CPU usage of 100% means that both processors are fully used. It is to be expected that LCR uses more CPU than Spread in the n-to-n case since LCR provides higher throughput. However, Figure 4.27 shows that this is only true for small messages.

In the 1-to-n case (Figure 4.28), the analysis is more complicated since not all processes perform the same tasks. In LCR the sender has less work to do than other processes since it does not receive messages to forward, but only receives acks. We see that for larger messages (when acks become insignificant compared to payload messages), the sender uses almost half as much CPU as the receivers.

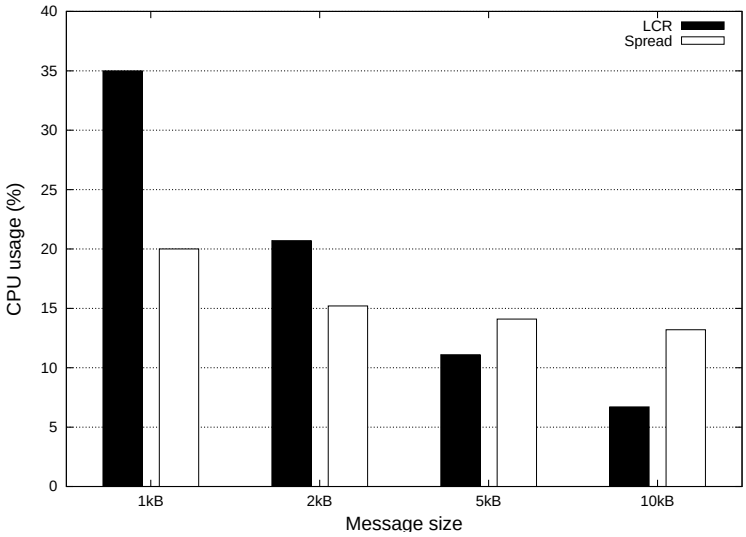

Figure 4.27: CPU usage during high load n-to-n broadcasts of the LCR and Spread implementations. Note that LCR provides higher throughput.

In Spread the sender and receivers use the same percentage of CPU time. It is also surprising that Spread uses more CPU than LCR for larger messages since in LCR processes need to forward messages while in Spread processes only receive them.

Finally, we can observe that LCR uses more CPU in the n-to-n case than in the 1-to-n case. This was expected since in the first case LCR provides higher throughput than in the latter.

4.9.4 Latency

The experiment of Figure 4.29 compares the latency of LCR to that of Spread without contention. As expected, the latency of LCR scales linearly due to ring topology, while Spread's latency is almost constant due the use of IP-multicast for message dissemination.

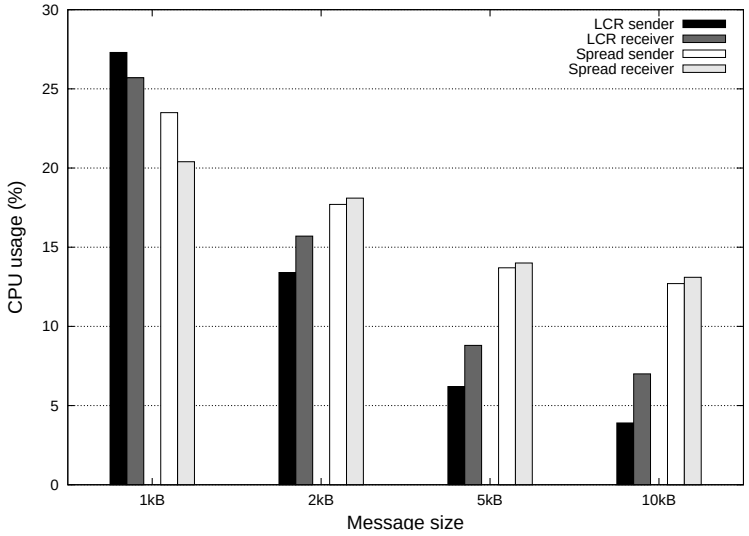

Figure 4.28: CPU usage during high load 1-to-n broadcasts of the LCR and Spread implementations. Surprisingly, for larger messages, receivers in Spread use more CPU than LCR despite the fact that in LCR processes need to forward messages while in Spread processes only receive.

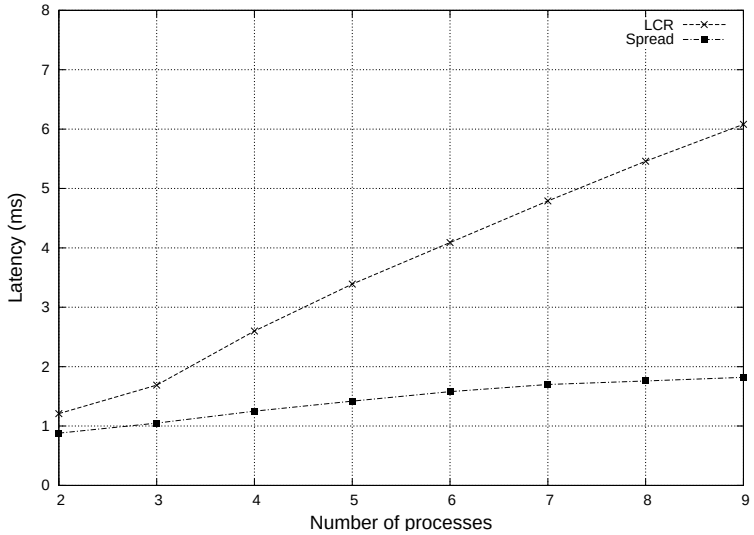

Figure 4.29: Latency comparison of the LCR and Spread implementations. The latency of LCR scales linearly due to ring topology. Spread's latency is fairly constant due the use of IP-multicast for message dissemination.

5

Concluding Remarks

> I now realise that if you don't understand the No, pretend it doesn't exist, was never said, then, slain by your incomprehension, it will transform itself abruptly into its opposite.
>
> *Maximum City : Bombay Lost and Found*
> Suketu Mehta

In this thesis, we presented efficient atomic storage that can be read and written by any number of clients. We considered three key complexity metrics: time, number of logs and throughput. For each metric we provided tight performance bounds and matching algorithms. We now briefly summarize our contributions and outline a few open issues and directions for future investigation.

Time-complexity. We presented optimal fast storage implementations that complete both reads and writes in 1 round-trip between the client and the servers. We showed that a fast implementation is possible if and only if the number of readers is less than $n/f - 2$, with n servers out of which f can fail. Furthermore, we showed that fast implementations are impossible for multiple writers if servers can fail.

We also determined a bound on the number of readers in semi-fast implementations, i.e., implementations where one of the operations, read or write, may not be fast. Determining whether this bound on the number of readers is tight is still however an open problem. Obviously, the fast implementation presented in this thesis is also a semi-fast implementation. Thus the tight bound R' on the number of readers in a single-writer semi-fast implementation satisfies $n/f - 2 \leq R' \leq n/f$. Apart from closing this gap, it would be interesting to investigate the possibility of semi-fast implementations in the multi-writer case.

Another direction of further investigation could be designing hybrid implementations, i.e., implementations that tolerate failures of any minority of servers, which (from our proposition) cannot be fast, but that would complete their operations in a fast mode whenever a sufficient number of servers are available. Hybrid

implementations might be useful in a dynamic setting such as that of [Lynch and Shvartsman 2002]: depending on the current configuration, such implementations would be able to switch between fast and slow modes.

Log-complexity. We revised the notion of atomicity for the crash-recovery model, determined a lower bound on log-complexity and introduced an atomic storage matching the bound. We also established optimality of the storage in terms of resilience, as well as time-complexity.

One possible way of extending our work is to look at the throughput of an atomic storage in the crash-recovery model. In this thesis we already showed how to improve the throughput of an atomic storage in the crash-stop model. This high throughput storage implementation can easily be adapted to a crash-recovery scenario thanks to the underlying group membership protocol: after a server crashes, it leaves the active group and simply rejoins once fixed. The key difference is that our high throughput storage tolerates the failure of all but one server, whereas an implementation *designed* for the crash-recovery model tolerates the simultaneous failure of *all* servers. To achieve this high level of fault tolerance, logging must be used. Since logging is expensive, it is a challenge to design a high throughput storage for the crash-recovery model.

Throughput. We introduced an atomic storage that provides optimal read throughput for homogeneous clusters of servers. The storage organizes servers around a ring and assumes point-to-point communication. We modified the storage algorithm to solve the more general uniform total order broadcast problem, which can be used to replicate any application reliably. The resulting algorithm is throughput optimal, regardless of message broadcast patterns. The performance evaluations confirmed the theoretical results.

Several techniques were used to successfully design high throughput algorithms. The first was to optimize performance for the synchronous and failure-free case which is considered to be the most frequent case in practice. The second was to consider a realistic round-based performance model where processes can only send and receive a single message per round. The third was to only use point-to-point communication links and avoiding all possible message collisions. The fourth and last technique was to piggy-back all acknowledgement messages, thus further reducing the possibility of collisions.

It would be interesting to apply the same techniques to design high throughput algorithms for the Byzantine failure model. Several recent studies have looked at designing efficient algorithms that tolerate malicious servers [Kotla et al. 2007; Castro and Liskov 2002; Cowling et al. 2006]. None of these studies however have taken a systematic approach to ensuring high throughput. Significant work is still required, especially in combining the above mentioned techniques with ways to minimize the number of digital signatures that also impact throughput.

Space-complexity is another metric that could be considered in future work. In a reliable distributed storage, the actual storage space accessible to clients is smaller than the total storage space used by the servers: there is a storage

overhead due to data replication. Recently however, it has been argued that erasure coding is a better alternative to data replication since it reduces the cost of ensuring fault tolerance.

For instance, it has been proven that a replicated storage system with 4 servers can tolerate at most 1 failure in an asynchronous environment ($n \geq 2f+1$). If each server has a storage capacity of 1 TB, the total capacity of the replicated storage system is still 1 TB. In this case the storage overhead (total capacity/useable capacity) is 4, i.e. only 1/4 of the total capacity is available. Erasure coding allows the reduction of this overhead to 2 in an asynchronous system, i.e. making 2 TB useable. In a synchronous system, it is even possible to further reduce this overhead and make 3 TB available to the user while still tolerating 1 failure. In this particular case tolerating 1 failure costs 1 TB instead of 3 TB with replication.

It would be interesting to determine tight bounds on space complexity in different scenarios and design matching algorithms. Especially since existing erasure coded storage algorithms are either not wait-free [Frolund et al. 2004; Aguilera et al. 2005], or do not offer optimal resilience in the crash-stop model [Cachin and Tessaro 2006].

Bibliography

Abd-El-Malek, M., Courtright, W. V., Cranor, C., Ganger, G. R., Hendricks, J., Klosterman, A. J., Mesnier, M., Prasad, M., Salmon, B., Sambasivan, R. R., Sinnamohideen, S., Strunk, J. D., Thereska, E., Wachs, M., and Wylie, J. J. 2005. Ursa minor: Versatile cluster-based storage. In *Proceedings of the FAST '05 Conference on File and Storage Technologies*. USENIX.

Abdellatif, T., Cecchet, E., and Lachaize, R. 2004. Evaluation of a group communication middleware for clustered j2ee application servers. In *Lecture Notes in Computer Science*. 1571–1589.

Abraham, I., Chockler, G., Keidar, I., and Malkhi, D. 2005. Byzantine disk paxos: optimal resilience with byzantine shared memory. *Distributed Computing 18,* 5, 387–408.

Aguilera, M., Chen, W., and Toueg, S. 1998. Failure detection and consensus in the crash-recovery model. In Proceedings of the 12th International Symposium on Distributed Computing (DISC). *ACM Transactions on Programming Languages and Systems*, 231–245.

Aguilera, M. K., Janakiraman, R., and Xu, L. 2005. Using erasure codes efficiently for storage in a distributed system. In *Proceedings of the 2005 International Conference on Dependable Systems and Networks (DSN'05)*. 336–345.

Amir, Y., Danilov, C., Miskin-Amir, M., Schultz, J., and Stanton, J. 2004. The spread toolkit: Architecture and performance. Tech. rep., CNDS-2004-1, Johns Hopkins University.

Amir, Y., Moser, L. E., Melliar-Smith, P. M., Agarwal, D. A., and Ciarfella, P. 1995. The Totem single-ring ordering and membership protocol. *ACM Transactions on Computer Systems 13,* 4, 311–342.

Anceaume, E. 1997. A lightweight solution to uniform atomic broadcast for asynchronous systems. In *Proceedings of the 27th International Symposium*

on Fault-Tolerant Computing (FTCS '97). IEEE Computer Society, Washington, DC, USA.

Armstrong, S., Freier, A., and Marzullo, K. 1992. Multicast transport protocol. RFC 1301, IETF.

Attiya, H. 2000. Efficient and robust sharing of memory in message-passing systems. *Journal of Algorithms 34,* 1, 109–127.

Attiya, H., Bar-Noy, A., and Dolev, D. 1995. Sharing memory robustly in a message passing system. *Journal of the ACM 42,* 1, 124–142.

Attiya, H. and Welch, J. 1998. *Distributed Computing, Fundamentals, Simulations and Advanced Topics*. McGraw-Hill International (UK).

Baldoni, R., Cimmino, S., and Marchetti, C. 2006. A Classification of Total Order Specifications and its Application to Fixed Sequencer-based Implementations. *to appear in Journal of Parallel and Distributed Computing*.

Ban, B. 2007a. *JGroups – A Toolkit for Reliable Multicast Communication*. http://www.jgroups.org.

Ban, B. 2007b. *Performance tests JGroups 2.5*. http://www.jgroups.org/javagroupsnew/perfnew/Report.html.

Bar-Noy, A. and Kipnis, S. 1994. Designing broadcasting algorithms in the postal model for message-passing systems. *Mathematical Systems Theory 27,* 5, 431–452.

Birman, K. and Joseph, T. 1987a. Exploiting virtual synchrony in distributed systems. In *Proceedings of the eleventh ACM Symposium on Operating systems principles (SOSP'87)*. 123–138.

Birman, K. and Joseph, T. 1987b. Reliable communication in the presence of failures. *ACM Trans. Comput. Syst. 5,* 1, 47–76.

Birman, K. and van Renesse, R. 1993. *Reliable Distributed Computing with the Isis Toolkit*. IEEE Computer Society Press.

Boichat, R. and Guerraoui, R. 2005. Reliable and total order broadcast in a crash-recovery model. *Journal of Parallel and Distributed Computing, to appear*.

Cachin, C. and Tessaro, S. 2006. Optimal resilience for erasure-coded byzantine distributed storage. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN'06)*. 115–124.

Carr, R. 1985. The tandem global update protocol. *Tandem Syst. Rev. 1*, 74–85.

Castro, M. and Liskov, B. 2002. Practical byzantine fault tolerance and proactive recovery. *ACM Transactions on Computer Systems (TOCS) 20,* 4 (Nov.), 398–461.

Cecchet, E., Marguerite, J., and Zwaenepoel, W. 2004. Cjdbc: Flexible database clustering middleware.

Chandra, T. and Toueg, S. 1996a. Unreliable failure detectors for reliable distributed systems. *J. ACM 43,* 2, 225–267.

Chandra, T. and Toueg, S. 1996b. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM 43,* 2, 225–267.

Chang, J.-M. and Maxemchuk, N. 1984. Reliable broadcast protocols. *ACM Trans. Comput. Syst. 2,* 3, 251–273.

Chun, B.-G., Dabek, F., Haeberlen, A., Sit, E., Weatherspoon, H., Kaashoek, F., Kubiatowicz, J., and Morris, R. 2006. Efficient replica maintenance for distributed storage systems. In *Proceedings of the 3rd USENIX Symposium on Networked Systems Design and Implementation (NSDI '06)*. San Jose, CA.

Cowling, J., Myers, D., Liskov, B., Rodrigues, R., and Shrira, L. 2006. Hq replication: A hybrid quorum protocol for byzantine fault tolerance. In *Proceedings of the Seventh Symposium on Operating Systems Design and Implementations (OSDI)*. Seattle, Washington.

Cristian, F. 1991. Asynchronous atomic broadcast. *IBM Technical Disclosure Bulletin 33,* 9, 115–116.

Cristian, F., Mishra, S., and Alvarez, G. 1997. High-performance asynchronous atomic broadcast. *Distrib. Syst. Eng. J. 4,* 2 (jun), 109–128.

Culler, D., Karp, R., Patterson, D., Sahay, A., Schauser, K., Santos, E., Subramonian, R., and von Eicken, T. 1993a. LogP: Towards a realistic model of parallel computation. In *Proceedings of the fourth ACM SIGPLAN symposium on Principles and practice of parallel programming.* 1–12.

Culler, D., Karp, R., Patterson, D., Sahay, A., Schauser, K., Santos, E., Subramonian, R., and von Eicken, T. 1993b. LogP: Towards a realistic model of parallel computation. In *Proceedings of the fourth ACM SIGPLAN symposium on Principles and practice of parallel programming.* 1–12.

Défago, X., Schiper, A., and Urbán, P. 2003. Comparative performance analysis of ordering strategies in atomic broadcast algorithms. *IEICE Trans. on Information and Systems E86-D,* 12, 2698–2709.

Défago, X., Schiper, A., and Urbán, P. 2004. Total order broadcast and multicast algorithms: Taxonomy and survey. *ACM Computer Survey 36,* 4, 372–421.

Dunagan, J., Harvey, N. J. A., Jones, M. B., Kostic, D., Theimer, M., and Wolman, A. 2004. Fuse: Lightweight guaranteed distributed failure notification. In *Proceedings of 6th Symposium on Operating Systems Design and Implementation (OSDI '04).*

Dutta, P., Guerraoui, R., Levy, R. R., and Chakraborty, A. 2004. How fast can a distributed atomic read be? In *Proceedings of the twenty-third annual ACM symposium on Principles of distributed computing (PODC).* ACM Press, 236–245.

Ekwall, R., Schiper, A., and Urban, P. 2004. Token-based atomic broadcast using unreliable failure detectors. In *Proceedings of the 23rd IEEE International Symposium on Reliable Distributed Systems (SRDS'04)*. IEEE Computer Society, Washington, DC, USA, 52–65.

Ezhilchelvan, P., Macedo, R., and Shrivastava, S. 1995. Newtop: a fault-tolerant group communication protocol. In *Proceedings of the 15th International Conference on Distributed Computing Systems (ICDCS'95)*. IEEE Computer Society, Washington, DC, USA.

Fan, R. and Lynch, N. 2003. Efficient replication of large data objects.

Friedman, T. and Renesse, R. V. 1997. Packing messages as a tool for boosting the performance of total ordering protocls. In *Proceedings of the 6th International Symposium on High Performance Distributed Computing (HPDC '97)*. IEEE Computer Society, Washington, DC, USA.

Fritzke, U., Ingels, P., Mostefaoui, A., and Raynal, M. 2001. Consensus-based fault-tolerant total order multicast. *IEEE Trans. Parallel Distrib. Syst. 12,* 2, 147–156.

Frolund, S., Merchant, A., Saito, Y., Spence, S., and Veitch, A. 2004. A decentralized algorithm for erasure-coded virtual disks. In *Proceedings of the 2004 International Conference on Dependable Systems and Networks (DSN'04)*. 125.

Gafni, E. 1998. Round-by-round fault detectors (extended abstract): unifying synchrony and asynchrony. In *Proceedings of the seventeenth annual ACM symposium on Principles of distributed computing (PODC'98)*. 143–152.

Garcia-Molina, H. and Spauster, A. 1991. Ordered and reliable multicast communication. *ACM Trans. Comput. Syst. 9,* 3, 242–271.

Gopal, A. and Toueg, S. 1989. Reliable broadcast in synchronous and asynchronous environments (preliminary version). In *Proceedings of the 3rd International Workshop on Distributed Algorithms*. Springer-Verlag, London, UK, 110–123.

Guerraoui, R., Levy, R. R., Pochon, B., and Quéma, V. 2006. High Throughput Total Order Broadcast for Cluster Environments. In *Proceedings of the IEEE International Conference on Dependable Systems and Networks (DSN 2006)*.

Guerraoui, R. and Vukolic, M. 2006. How Fast Can a Very Robust Read Be? In *Proceedings of the 25th ACM Symposium on Principles of Distributed Computing (PODC'06)*.

Hadzilacos, V. and Toueg, S. 1993. Fault-tolerant broadcasts and related problems. *Distributed systems (2nd Ed.)*, 97–145.

Hendler, D. and Kutten, S. 2006. Constructing shared objects that are both robust and high-throughput. In *Proceedings of 20th international symposium on distributed computing (DISC '06)*. 428–442.

Herlihy, M. 1991. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems 11,* 1, 124–149.

Herlihy, M. and Wing, J. 1990. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems 12,* 3, 463–492.

Jones, R. 2007. *Netperf.* http://www.netperf.org/.

Kaashoek, F. and Tanenbaum, A. 1996. An evaluation of the amoeba group communication system. In *Proceedings of the 16th International Conference on Distributed Computing Systems (ICDCS '96).* IEEE Computer Society, Washington, DC, USA.

Keidar, I. and Shraer, A. 2006. Timeliness, failure-detectors, and consensus performance. In *Proceedings of the twenty-fifth annual ACM symposium on Principles of distributed computing (PODC'06).* 169–178.

Kenchammana-Hosekote, D. R., Golding, R. A., Fleiner, C., and Zaki, O. A. 2004. The design and evaluation of network raid protocols. Research report RJ 10316, IBM Almaden Research Center.

Kim, J. and Kim, C. 1997. A total ordering protocol using a dynamic token-passing scheme. *Distrib. Syst. Eng. J. 4,* 2 (jun), 87–95.

Kotla, R., Alvisi, L., Dahlin, M., Clement, A., and Wong, E. 2007. Zyzzyva: speculative byzantine fault tolerance. In *SOSP '07: Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles.* ACM, New York, NY, USA, 45–58.

Lamport, L. 1978. Time, clocks and the ordering of events in a distributed system. *Communications of the ACM 21,* 7, 558–565.

Lamport, L. 1985. On interprocess communication - part i: Basic formalism, part ii: Algorithms. *DEC SRC Report 8.* Also in Distributed Computing, 1, 1986, 77-101.

Lamport, L. 1998. The part-time parliament. *DEC SRC 1989. (Also in ACM Transactions on Computer Systems.*

Luan, S. and Gligor, V. 1990. A fault-tolerant protocol for atomic broadcast. *IEEE Trans. Parallel Distrib. Syst. 1,* 3, 271–285.

Lynch, N. 1996. *Distributed Algorithms.* Morgan Kaufmann Publishers, San Mateo, CA.

Lynch, N. and Shvartsman, A. 1997. Robust emulation of shared memory using dynamic quorum-acknowledged broadcasts. *Proceedings of the 27th International Symposium on Fault-Tolerant Computing Systems (FTCS'97).*

Lynch, N. and Shvartsman, A. 2002. Rambo: A reconfigurable atomic memory service for dynamic networks. *Proceedings of the 16th International Symposium on Distributed Computing (DISC).*

Malhis, L., Sanders, W., and Schlichting, R. 1996. Numerical performability evaluation of a group multicast protocol. *Distrib. Syst. Eng. J. 3,* 1 (march), 39–52.

Moser, L., Melliar-Smith, P., and Agrawala, V. 1993. Asynchronous fault-tolerant total ordering algorithms. *SIAM J. Comput. 22,* 4, 727–750.

Ng, T. 1991. Ordered broadcasts for large applications. In *Proceedings of the 10th IEEE International Symposium on Reliable Distributed Systems (SRDS'91)*. IEEE Computer Society, Pisa, Italy, 188–197.

Peterson, L., Buchholz, N., and Schlichting, R. 1989. Preserving and using context information in interprocess communication. *ACM Trans. Comput. Syst. 7,* 3, 217–246.

Rodrigues, L., Fonseca, H., and Verissimo, P. 1996. Totally ordered multicast in large-scale systems. In *Proceedings of the 16th International Conference on Distributed Computing Systems (ICDCS '96)*. IEEE Computer Society, Washington, DC, USA.

Saito, Y., Frølund, S., Veitch, A., Merchant, A., and Spence, S. 2004. Fab: building distributed enterprise disk arrays from commodity components. *SIGOPS Operating Systems Review 38,* 5, 48–58.

Shao, C., Pierce, E., and Welch, J. 2003. Multi-writer consistency conditions for shared memory objects. *Proceedings of the 17th International Symposium on Distributed Computing (DISC)*.

Stanton, J. R. 2002. *A Users Guide to Spread.* http://www.spread.org/docs/guide/users_guide.pdf.

Urbán, P., Défago, X., and Schiper, A. 2000. Contention-aware metrics for distributed algorithms: Comparison of atomic broadcast algorithms. In *Proceedings of 9th IEEE International Conference on Computer Communications and Networks (IC3N 2000)*. 582–589.

van Renesse, R. and Schneider, F. B. 2004. Chain replication for supporting high throughput and availability. In *Proc. of the 6th Symposium on Operationg Systems Design and Implementation.*

Vicente, P. and Rodrigues, L. 2002. An indulgent uniform total order algorithm with optimistic delivery. In *Proceedings of the 21st IEEE Symposium on Reliable Distributed Systems (SRDS'02)*. IEEE Computer Society, Washington, DC, USA.

Whetten, B., Montgomery, T., and Kaplan, S. 1994. A high performance totally ordered multicast protocol. In *Selected Papers from the International Workshop on Theory and Practice in Distributed Systems.* Springer-Verlag, London, UK, 33–57.

Wilhelm, U. and Schiper, A. 1995. A hierarchy of totally ordered multicasts. In *Proceedings of the 14TH Symposium on Reliable Distributed Systems.* IEEE Computer Society, Washington, DC, USA.

List of Figures

Curriculum Vitæ

Ron R. Levy completed elementary school in Dordrecht, the Netherlands by 1990. From 1990 to 1997 he attended the Lycée International de Ferney-Voltaire in France, where he obtained a "Baccalauréat Scientifique à Option Internationale" and a "GCSE in English and English Literature".

In 1997, he started his studies at the Swiss Federal Institute of Technology in Lausanne, Switzerland (EPFL) in the Communication Systems section. He obtained his Masters degree (Msc) in 2002 after completing a six month internship at the T.J. Watson IBM research center in Hawthorne, New York, USA.

In 2002, he started a PhD thesis under the supervision of Prof. Rachid Guerraoui in the School of Computer and Communication Sciences of the EPFL. In 2007 he spent three months at Bell Labs Research in Bangalore, India as part of his PhD work.