Abstractions for Asynchronous Distributed Computing with Malicious Players

THÈSE N^{O} 4241 (2008)

PRÉSENTÉE LE 5 DÉCEMBRE 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

Marko VUKOLIĆ

acceptée sur proposition du jury:

Prof. A. Schiper, président du jury
Prof. R. Guerraoui , directeur de thèse
Dr C. Cachin, rapporteur
Prof. I. Keidar, rapporteur
Prof. A. Lenstra, rapporteur

ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Lausanne, EPFL
2008

Résumé

L'un des caractéristiques des systèmes répartis modernes est que les pannes y sont une norme plutôt qu'une exception. Dans un grand nombre de cas, ces pannes ne sont pas bénignes. Internet peut par exemple provoquer de nombreux comportements malicieux (également appelés Byzantins ou arbitraires) et asynchrones. En conséquence, la recherche dans le domaine des systèmes repartis asynchrones et tolérants aux pannes Byzantines (TPB) a fleurit.

La tolérance au comportement arbitraire et à l'asynchronisme demande des algorithmes très sophistiqués. Cela est en particulier le cas pour les solutions TPB qui visent à assurer les propriétés comme: (a) la résilience optimale, i.e., la tolérance à un aussi grand nombre de pannes Byzantines que possible et (b) la performance optimale par rapport à une métrique de complexité adéquate.

La plupart des algorithmes TPB sont développés è partir de zéro ou par modification de solutions existantes de manière non modulaire, ce qui rend ces algorithmes difficiles à comprendre et gène leur adoption. Nous attribuons cette complexité au manque d'abstractions adéquates pour la programmation asynchrone TPB. La motivation de cette thèse est de proposer les abstractions réutilisables pour mettre des algorithmes répartis asynchrones TPB qui ont une résilience et/ou une complexité optimales, avec un accent fort sur l'une des importantes métriques de complexité - la complexité en temps (également appelée la latence).

Les abstractions proposées dans cette thèse sont construites avec comme objectif trois applications réparties fondamentales: (a) le stockage écriture/lecture (également appelé le registre), (b) le consensus et (c) la réplication de machine d'état ("state machine replication" - RME). Dans cette thèse, nous expliquons comment utiliser nos abstractions dans ces applications pour inventer des algorithmes asynchrones TPB, caractérisés par la meilleure complexité parmi tous les algorithmes que nous connaissons, outre la résilience optimale.

Nous présentons tout d'abord la notion de système de quorums raffinés (SQR) d'un ensemble S comme l'ensemble de trois classes de sous-ensembles de S: les quorums de la première classe sont également des quorums de la deuxième classe, eux-mêmes étant également des quorums de la troisième classe. Les quorums de la première classe ont de grandes intersections avec tous les autres quorums, les

quorums de la deuxième classe ont typiquement des intersections plus petites avec ceux de la troisième classe, ces derniers correspondent simplement à des quorums traditionnels. L'abstraction de systèmes de quorums raffinés aide la conception d'algorithmes tolérant la concurrence entre les processus, l'asynchronisme d'une longueur indéfinie et un grand nombre des pannes, néanmoins qui se réagissent vite si peu de pannes ce produisent, si le système est synchrone et s'il n'y a pas de concurrence, i.e., sous les conditions qui sont considérés fréquentes en pratique. En d'autres mots, les SQR aident combiner la résilience optimale avec la latence du meilleur cas optimale. Intuitivement, sous les conditions synchrones et sans concurrence, l'implémentation d'un objet reparti faciliterait l'opération si le quorum de la première classe est accédé, puis se dégraderait gracieusement selon qu'un quorum de la deuxième ou de la troisième classe est accédé. Notre notion de RQS a été inventée en supposant un adversaire général, et ceci essentiellement peut permettre aux algorithmes qui dépendent de system RQS de relâcher l'hypothèse de pannes indépendants des processus. Nous illustrons le pouvoir des RQS en présentant de nouvelles implémentations optimales d'un stockage atomique et un algorithme de consensus.

Notre deuxième abstraction est un nouveau mécanisme d'estampillage nommé les estampilles à haute définition (eHD), qui pourraient être vues comme une variante des horloges matricielles. Grosso modo, une estampille à haute définition contient une matrice d'estampilles locales de (un sous-ensemble de) processus tel que vu par (un sous-ensemble) des autres processus. Complémentaires aux SQR, les eHD simplifient la conception d'algorithmes répartis TPB qui combinent la résilience optimale et la latence au pire cas optimale. Nous appliquons les estampilles à haute définition afin de concevoir des algorithmes de stockage écriture/lecture dans lesquels les eHDs sont utilisées pour la détection et le filtrage des processus Byzantins, ce qui ouvre la voie à des algorithmes de stockage TPB qui combinent la résilience optimale avec la latence au pire cas optimale.

Finalement, nous introduisons *Abstract*, une abstraction générique qui simplifie la tache notoirement difficile du développement des protocoles de réplication de machine d'état TPB. *Abstract* ressemble à une réplication de machine d'état et pourrait être utilisée afin de rendre n'importe quel service partagé tolérant aux pannes Byzantines, avec une exception: elle pourra parfois abandonner la requête de client. La condition de non trivialité sous laquelle *Abstract* ne pourra pas abandonner est un paramètre générique. Nous voyons un protocole RME-TPB comme une composition d'instances d'*Abstract* développées et analysées indépendamment. Afin de illustrer notre approche, nous décrivons deux nouveaux algorithmes TPB caractérisés par une résilience optimale. Le première, qui utilise nos quorums raffinés, a la plus petite latence parmi tous les protocoles RME-TPB que nous connaissons dans les périodes synchrones qui sont libres de concurrence et pannes. Le deuxième algorithme à le débit le plus haut en périodes synchrones sans pannes; cet algorithme soutient l'applicabilité générale d'*Abstract* en développant des services partagés TPB caractérisés par une complexité optimale, au delà de la complexité en temps.

Mots-clés: asynchronisme, complexité, consensus, système de stockage réparti, résilience optimale, réplication de machine d'état.

Abstract

In modern distributed systems, failures are the norm rather than the exception. In many cases, these failures are not benign. Settings such as the Internet might incur malicious (also called Byzantine or arbitrary) behavior and asynchrony. As a result, and perhaps not surprisingly, research on asynchronous Byzantine fault-tolerant (BFT) distributed systems is flourishing.

Tolerating arbitrary behavior and asynchrony calls for very sophisticated algorithms. This is in particular the case with BFT solutions that aim to provide properties such as: (a) *optimal resilience*, i.e., tolerating as many Byzantine failures as possible and (b) optimal performance with respect to some relevant complexity metric.

Most BFT algorithms are built from scratch or by modifying existing solutions in a non-modular manner, which often renders these algorithms difficult to understand and, consequently, impedes their wider adoption. We attribute this complexity to the lack of sufficient number of adequate abstractions for asynchronous BFT distributed computing.

The motivation of this thesis is to propose *reusable* abstractions for devising asynchronous BFT distributed algorithms that are optimally resilient and/or have optimal complexity, with strong focus on one of the most important complexity metrics — *time complexity* (or *latency*). The abstractions proposed in this thesis are devised with three fundamental distributed applications in mind: (a) *read/write storage* (also called *register*), (b) *consensus* and (c) *state machine replication (SMR)*. We demonstrate how to use our abstractions in these applications to devise asynchronous BFT algorithms that feature the best complexity among all algorithms we know of, in addition to optimal resilience.

First, we introduce the notion of a *refined* quorum system (RQS) of some set S as a set of three classes of subsets (quorums) of S: first class quorums are also second class quorums, themselves being also third class quorums. First class quorums have large intersections with all other quorums, second class quorums typically have smaller intersections with those of the third class, the latter simply correspond to traditional quorums. The refined quorum system abstraction helps design algorithms that tolerate contention (process concurrency), arbitrarily long

periods of asynchrony and the largest possible number of failures, but perform fast if few failures occur, the system is synchronous and there is no contention, i.e., under conditions that are assumed to be frequent in practice. In other words, RQS helps combine optimal resilience and *optimal best-case time complexity*. Intuitively, under uncontended and synchronous conditions, a distributed object implementation would expedite an operation if a quorum of the first class is accessed, then degrade gracefully depending on whether a quorum of the second or the third class is accessed. Our notion of RQS is devised assuming a general adversary structure, and this basically allows algorithms relying on RQS to relax the assumption of independent process failures. We illustrate the power of refined quorums by introducing two new optimal BFT atomic object implementations: an atomic storage and consensus algorithm.

Our second abstraction is a novel timestamping mechanism called *high resolution timestamps (HRts)*, which can be seen as a variation of a matrix clocks. Roughly speaking, a high resolution timestamp contains a matrix of local timestamps of (a subset of) processes as seen by (a subset of) other processes. Complementary to RQS, HRts simplify the design of BFT distributed algorithms that combine optimal resilience and *worst-case time complexity*. We apply high-resolution timestamps to design read/write storage algorithms in which HRts are used to detect and filter out Byzantine processes, which paves the path to the first BFT storage algorithms that combine optimal resilience with optimal worst-case time complexity.

Finally, we introduce *ABsTRACT* (Abortable Byzantine faulT-toleRant stAte maChine replicaTion), a generic abstraction that simplifies the notoriously difficult task of developing BFT state machine replication algorithms. ABsTRACT resembles BFT-SMR and it can be used to make any shared service Byzantine fault-tolerant, with one exception: it may sometimes *abort* a client request. The non-triviality condition under which ABsTRACT cannot abort is a generic parameter. We view a BFT-SMR algorithm as a composition of instances of ABsTRACT, each instance developed and analyzed independently. To illustrate our approach, we describe two new optimally resilient BFT algorithms. The first, that makes use of our refined quorums, has the lowest time complexity among all BFT-SMR algorithms we know of, in synchronous periods that are free from contention and failures. The second algorithm has the highest peak throughput in failure-free and synchronous periods; this algorithm argues for general applicability of ABsTRACT in developing BFT shared services that feature optimal complexity, beyond the time complexity metric.

Keywords: asynchrony, complexity, consensus, distributed storage, optimal resilience, state-machine replication.

Acknowledgements

To begin with, I would like to thank my advisor Rachid Guerraoui, for his constant support and guidance, and, in particular, for always being available for answering my questions and exchanging thoughts. But, above all, for being a true friend, more than just an advisor.

I would further like to thank to André Schiper for presiding over my thesis committee, as well as the members of the committee, Christian Cachin, Idit Keidar, and Arjen Lenstra, for the time they spent examining my thesis and the valuable comments they provided. Special thanks goes to Christian Cachin, for providing me with an opportunity to do an internship in IBM Zurich during the course of my doctorate, as well as for many interesting discussions on various topics on distributed computing and beyond.

Radmila Todosijević proofread those parts of the thesis that feature obviously better English. Patrick Droz and Robert Haas were very understanding of my work in parallel on writing the thesis and in IBM.

I am also grateful to my colleagues and co-workers. Specifically, to Partha Dutta for being always willing to share his knowledge with me while I was making my first steps in the scientific world. To Ron Levy for many interesting discussions, both on and off shore. To Vivien Quéma, for all his help in the work presented here, notably simulations and experiments. To all other LPD members, past and present, who made my time in the lab a great experience. Special thanks goes to Kristine Verhamme, for always caring.

I am deeply grateful to my parents for teaching me the true values in life, as well as for their love and colossal support they provide me with.

Finally, and above all, I would like to thank my family; my loving wife Ana for her patience and support, my son Vuk for the immense joy he brings to my life and the baby that we are eagerly expecting.

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 2003 to 2008. During this period, besides the work presented in this thesis, I also worked on fair exchange algorithms resilient to malicious (Byzantine) failures [AGGV05] and tools for automated atomicity verification [GV08]. In addition, I coauthored a magazine article (accepted for publication) which covers some of the work presented in this thesis [CGKV].

This thesis focuses on the abstractions for devising asynchronous Byzantine fault-tolerant distributed algorithms in the context of distributed storage, consensus and state machine replication and is a composition of three published papers [GLV06, GV06, GV07], as well as two papers that have been submitted for publication to peer reviewed conferences/journals [DGLV05, GQV08]

[AGGV05] G. Avoine, F. C. Gärtner, R. Guerraoui and M. Vukolić. Gracefully degrading fair exchange with security modules. In *Proceedings of the 5th European Dependable Computing Conference (EDCC)*, pages 55–71, 2005.

[DGLV05] P. Dutta, R. Guerraoui, R. R. Levy and M. Vukolić. How fast can a distributed atomic read be? Technical Report LPD-REPORT-2005-001, EPFL, School of Computer and Communication Sciences, Lausanne, Switzerland. 2005.

[GLV06] R. Guerraoui, R. R. Levy and M. Vukolić. Lucky read/write access to robust atomic storage. In *Proceedings of the IEEE International Conference on Dependable Systems and Networks (DSN)*, pages 125–136, 2006.

[GV06] R. Guerraoui and M. Vukolić. How fast can a very robust read be? In *Proceedings of the 25th ACM Symposium on Principles of Distributed Computing (PODC)*, pages 248–257, 2006.

[GV07] R. Guerraoui and M. Vukolić. Refined quorum systems. In *Proceedings of the 26th annual ACM Symposium on Principles of Distributed Computing (PODC)*, pages 119–128, 2007. The paper is invited to the special issue of *Distributed Computing*.

[GV08] R. Guerraoui and M. Vukolić. A scalable and oblivious atomicity assertion. In *Proceedings of the 19th International Conference on Concurrency Theory (CONCUR)*, pages 52–66, 2008.

[GQV08] R. Guerraoui, V. Quéma, and M. Vukolić. The next 700 BFT protocols. Technical Report LPD-REPORT-2008-008, EPFL, School of Computer and Communication Sciences, Lausanne, Switzerland. 2008.

[CGKV] G. Chockler, R. Guerraoui, I. Keidar and M. Vukolić. Reliable distributed storage. *IEEE Computer. To appear.*

Contents

1 Introduction

1.1 Context

A distributed system consists of several entities (typically called *processes*) able to compute and/or store information locally and communicate with each other in order to achieve some common goal. Distributed computing is about computing on such systems.

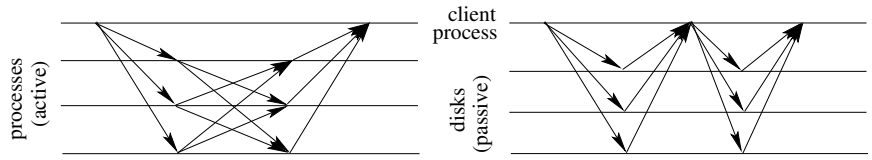
When compared to traditional computing on a singe centralized server that represents a single point of failure and may be a performance bottleneck, distributed computing solutions can offer fault-tolerance and increase availability, as well as provide better scalability. Furthermore, fault-tolerant distributed solutions based on commodity hardware may be considerably cheaper than the expensive dedicated centralized servers. Distributed computing solutions can also facilitate deployment of new services, since such solutions are less subject to expensive licenses and regulations; some examples include mobile ad hoc networks, sensor networks and, of course, the Internet.

Two challenges underlie distributed computing: *failures* and *asynchrony*. By increasing the number of processes performing the computation, failures become the norm, rather than the exception. In addition, in many settings, it is very risky to assume that these failures are of a benign type. For example, today, a significant number of home and corporate networks are connected to the Internet, which makes these networks a potential target for a malicious hacker, worm or virus, to name just few threats. Simply put, the more players into the game, the higher the risk of some of them being malicious.

Asynchrony means that a distributed solution cannot (always) rely on the network to be delivering messages in a timely manner. Synchronous solutions, often suitable for closed networks, expose very clearly their vulnerability: an attacker may simply target the timely delivery of messages in order to compromise the service. On the other hand, asynchrony, while tolerating unpredictable message delays, poses a considerable challenge to distributed algorithm design since it makes it impossible to distinguish slow processes from faulty ones.

1.2 Motivation

Above, we briefly discussed an obvious need for an extensive research in the area of asynchronous distributed malicious-fault-tolerant computing. It is, therefore, not surprising that ever since 1980, when Pease, Shostak and Lamport introduced malicious failures in their seminal paper [PSL80] (they named malicious failures *Byzantine* in [LSP82]) and opened a new chapter in the history of distributed computing, great number of papers on Byzantine fault-tolerant (BFT) systems have been published. Two of the most important challenges addressed by these BFT algorithms are to provide: (a) *optimal resilience*, i.e., tolerating as many Byzantine processes as possible (see e.g., [LSP82, CL99, MAD02a, CML+06, ACKM06, KAD+07]) and (b) optimal performance with respect to some relevant complexity metric. One of the obvious performance metrics of distributed algorithms is their *time complexity* (or *latency*) that, roughly speaking, measures how quickly a given algorithm can terminate. Time complexity is typically measured by the number of *message delays* (or *rounds* of communication) before an algorithm terminates [Awe85, Sch97] (see Figure 1.1(a) for illustration). Frequently, when client processes are accessing passive processes, e.g., commodity disks, a *communication round-trip* (equivalent to two rounds of communication) is used as a time complexity metric (see Figure 1.1(b)). BFT algorithms that optimize time complexity are numerous (e.g., [CL99, GWGR04, CML+06, AGG+05, RC05, MA06, Zie06, ACKM06, KAD+07, ACKM07]). The other important metrics are *throughput*, i.e., the number of requests that can be treated per time unit (e.g., [GGL03, vRS04, GLPQ06, GKLQ07]) and *message-complexity*, i.e., the total number of messages exchanged (e.g., [CKS00, GGK07]).



(a) 3 message delays (rounds) (b) 2 round-trips (4 rounds)

Figure 1.1: Time complexity (latency)

The motivation of this thesis is to propose reusable *abstractions* for devising asynchronous BFT distributed algorithms that are optimally resilient and/or have optimal complexity. The main focal complexity metric in this thesis is time complexity (although we discuss some throughput optimal implementations as well). In general, a point common to many BFT distributed algorithms, especially those that aim at combining optimal resilience and optimal time complexity, is that they are typically built from scratch (e.g., [CL99, KAD+07, AGG+05, MA06]), or by modifying existing solutions in a non-modular manner (e.g., [CML+06]). This can often render such BFT algorithms difficult to understand and, consequently, impede their wider adoption. We attribute this complexity to the lack of a sufficient number of adequate abstractions for asynchronous BFT distributed computing, especially those that are envisioned for devising optimally resilient and/or latency optimal BFT algorithms. The goal of this thesis is to propose

such abstractions.

The abstractions presented in the thesis are designed with three fundamental problems in mind:

1. *Read/write storage* (also called read/write *register*) [Lam86, ABD95, MR98, JCT98]. Distributed storage algorithms constitute an active area of research and are appealing alternatives to classical centralized storage systems based on specialized hardware [SFV+04, ACC+05, CDH+06, KHGFZ04]. In distributed storage, client processes access the *base objects* over which storage is implemented, such that the end user is provided with an illusion of accessing centralized storage. At the heart of such distributed storage lies a read/write storage abstraction (register). This thesis focuses on a fundamental class of read/write storage algorithms that support a single writer and multiple readers (SWMR) [ABD95, ACKM06, ACKM07]. The challenge, when devising distributed storage algorithms, is to ensure that reads and writes have low latency while (a) tolerating asynchrony and the (possibly Byzantine) failures of any number of clients that access the storage (wait-freedom [Her91]), as well as the largest possible number of Byzantine base object failures (optimal resilience) and (b) ensuring strong consistency.

   The most desirable consistency criterion for distributed storage is *atomicity* [Lam86] (also called *linearizability* [HW90]). Atomicity provides an illusion to the clients that storage is accessed sequentially. Other (weaker) consistency criterions include *safety* and in particular *regularity* [Lam86]. Safe storage guarantees that a read which is not concurrent with any write returns the last value written. The applicability of safe storage is limited since a read concurrent with a write may return an arbitrary value. On the other hand, regular storage strengthens safety by ensuring that read always returns a value actually written, and is not older than the value written by the last preceding write.

2. *Consensus* [LSP82, FLP85, DLS88, CT96, Lam98]. Consensus is probably the most fundamental problem in distributed computing. In (BFT) consensus, processes propose a value and are required to agree on a common value, such that: (a) no two *correct* (non-faulty) processes decide differently (agreement), (b) every correct process eventually decides (termination), and (c) if all processes that propose a value are correct, then the decision value is one of the proposed ones.

   Unfortunately, asynchronous fault-tolerant consensus is impossible [FLP85], even with a single non-Byzantine process failure.[1] Therefore, in this thesis, under the notion of *asynchronous* consensus algorithms, we consider *indulgent* [Gue00] algorithms, i.e., those that tolerate arbitrarily long periods of asynchrony and that provide agreement and validity even in the purely

[1] The impossibility result of [FLP85] applies to *deterministic* consensus. On the other hand, this is not the case if a probabilistic form of consensus is assumed [CKS00, BO83, Rab83], in which consensus properties may sometimes be violated (albeit with negligible probability). However, the focus of this thesis is on deterministic object implementations (although many of our abstractions and results are relevant in the probabilistic case as well).

asynchronous system. However, to circumvent the impossibility of [FLP85] and make termination possible, we study the consensus problem in an *eventually synchronous* model [DLS88,Gaf98] in which (roughly speaking) there is a time, not known to the processes, after which the system becomes synchronous and messages are delivered in a timely manner.

3. *State-machine replication* (SMR) [Lam78, Sch90, Lam98]. State machine replication is a software technique for tolerating failures using commodity hardware. The critical service (be it storage, consensus, or any other service) to be made fault-tolerant is modeled by a state machine. Several, possibly different, copies of the state machine are then placed on different nodes. Clients of the service access the replicas through a replication algorithm which ensures that, despite concurrency and failures, replicas perform client requests in the same order. At the heart of a SMR algorithm lies a repeated form of consensus.

1.3 Contributions

This thesis proposes the following abstractions for asynchronous BFT distributed computing:

1.3.1 Refined Quorum Systems

The first proposed abstraction is *refined quorum systems (RQS)*. Quorum systems are powerful mathematical tools to reason about distributed implementations of shared objects, in particular read/write storage (see e.g., [ABD95,MR98,JCT98]) and consensus abstractions. More specifically, quorum systems have been used to reason about algorithms that tolerate arbitrarily long periods of asynchrony and process failures (i.e., indulgent algorithms). Originally, a quorum system was defined as a set of subsets that intersect [Gif79] and this notion was key to reasoning about crash-resilient (non-Byzantine) asynchronous algorithms. More sophisticated forms of quorum systems have been introduced to cope with Byzantine failures: these require larger intersections among subsets [MR98].

However, while being very useful to reason about the resilience dimension, traditional quorums (be they simple or Byzantine) are not adequate to capture the complexity dimension. This is particularly important given the appealing nature of *optimistic* [Ped01] distributed object implementations, e.g., [Sch97,CL99, BGMR01,GWGR04,RC05,AGG+05,Lam06b,CBS06,MA06,CML+06,Zie06,KAD+07]. In addition to being indulgent, these implementations are also geared to reduce *best-case complexity*, i.e., latency under situations of synchrony and no-contention, which are typically argued to be "normal", i.e., the most frequent in practice. More specifically, these implementations are tuned to expedite operations in uncontended and synchronous situations, provided *"enough"* servers/base objects are accessed. This very notion of *"enough servers to expedite an operation"* is crucial, but is not captured by traditional quorum systems. It is natural to search for a mathematical abstraction that captures it in precise yet general terms. This is the motivation behind *refined quorum systems*.

In short, an RQS of some set of elements S is a set of three classes of subsets (quorums) of S: first class quorums are also second class quorums, which are also third class quorums. Quorums of the first class have large intersections with quorums of other classes, those of the second class typically have smaller intersections with those of the third class, the latter simply correspond to traditional quorums. In the context of a distributed object implementation, set S would typically contain a set of failure-prone base object processes over which some object abstraction (e.g., storage or consensus) is implemented. Intuitively, under uncontended and synchronous conditions, a distributed object implementation would expedite an operation if a quorum of the first class is available, then degrade gracefully depending on whether a quorum of the second or the third class is available. Moreover, (as detailed in Chapter 3) the RQS property of third class quorums is anyway necessary to provide resilience, since it is necessary to prevent network partitioning [MAD02a]. As a result, our RQS allow for combining optimal resilience and optimal best-case complexity of asynchronous BFT algorithms.

Furthermore, our refined quorum systems are designed to handle a general adversary structure in which various subsets of processes can collude to defeat the algorithm [HM97, MR98, JM03]. With such a general structure, we relax the assumption of independent and identically distributed failures (assumed in e.g., [CL99,GWGR04,RC05,AGG$^+$05,CML$^+$06,Lam06b,MA06,Zie06,KAD$^+$07]).

We illustrate the power of our notion of refined quorum systems by introducing two new atomic object implementations: (1) single-writer multi-reader atomic storage and (2) consensus. As detailed in Chapter 3 of this thesis, each of the two algorithms is interesting in its own right and is, in a precise sense, the first fully optimal algorithm of its kind in terms of best case time complexity.

1.3.2 High-resolution timestamps

As described previously, refined quorum systems are a useful abstraction for combining optimal best-case time complexity and optimal resilience of asynchronous BFT algorithms. The question that naturally arises if one desires a complete picture is: what about the optimal worst-case time complexity? In this thesis, this question is tackled from the perspective of read write-storage. Strictly speaking, fault-tolerant asynchronous consensus is impossible even with one benign failure [FLP85] and the notion of worst-case complexity is not clear.

We introduce a new timestamping [Lam78] mechanism called *high-resolution timestamps* to achieve the first asynchronous BFT (safe and regular) storage algorithm that combines optimal worst-case time complexity and optimal resilience. In effect, high resolution timestamps can be seen as a variation of matrix clocks [WB84,RS96]. In matrix clocks, roughly speaking: (a) all processes in the system maintain their own local timestamp, (b) all processes in the system maintain the *view* on other timestamps in the system (i.e., copies of other timestamps), and (c) all processes maintain a copy of a view of every other process. In contrast, in high-resolution timestamps: (a) not all processes are required to maintain their own local timestamp (only clients are) and (b) clients do not store copies of timestamps of other processes in the system; this information is stored

only at base objects.

Basically, a high resolution timestamp consists of the traditional writer's monotonically increasing timestamp accompanied by the base objects' view (i.e., their latest copies) of the readers' local timestamps. Evidently, high-resolution timestamps require readers to write (meta-data) into storage, in order to optimize the (worst-case time) complexity. The fact that readers need to write in order to achieve the optimal complexity follows from [ACKM06] and the lower bounds and algorithms presented in Chapter 4. This observation is of independent interest and complements the previous findings that readers need to write in order to implement (regular) wait-free BFT storage with limited storage capacities [CGK07] and fault-tolerant atomic storage [Lam86, ABD95].

High resolution timestamps allow our optimally resilient safe and regular storage algorithms to have both reads and writes that complete in two round trips. This complexity was proven optimal for the case of a write operation by [ACKM06], whereas this thesis proves it optimal for the case of a read operation. Moreover, by using our results and a simple transformation from regular to atomic storage [GR06] the worst-case optimal latency of atomic BFT storage can be bounded to between 2 and 4 round-trips (the exact latency remains an open problem).

We further complement these results by answering the following questions: (1) is there a *fast* BFT implementation where none of the operations (read or write) requires more than one communication round-trip? and (2) if yes, how much resilience do we need to sacrifice in order to have such an implementation? Clearly, fast implementations would be optimal in terms of time complexity. Since the answer to the two question is known in the case of safe and regular storage [MR98], our focus is on atomic storage. Our results, that in effect extend the results of [DGLC04] from the crash-only fault model to BFT, answer the first question affirmatively and precisely quantify the resilience tradeoff in the hybrid failure model [TP88] (which distinguishes Byzantine and benign failures). We express this tradeoff as a function of the number of readers R, the threshold on the total number of faulty (i.e, Byzantine or crash-faulty) base objects t and the threshold on the number of Byzantine base objects b ($b \leq t$). Having these parameters in mind, we establish a tight lower bound on the number S of base objects required by a fast BFT atomic storage implementation — $S \geq (R+2)t + (R+1)b + 1$.

1.3.3 ABsTRACT (Abortable BFT-SMR)

Finally, we present *A*BsTRACT, Abortable Byzantine faulT-toleRant stAte maChine replicaTion (simply written *Abstract*): a new abstraction that significantly reduces the development and maintenance cost of BFT-SMR algorithms and makes it significantly easier to develop efficient ones. *Abstract* resembles state machine replication and it can be used to make any shared service Byzantine fault-tolerant, with one exception: it may sometimes *abort* a client request. The (*non-triviality*) condition under which *Abstract* cannot abort is a generic parameter.

At one extreme, one can for example specify a (useless) *Abstract* instance that could abort in every execution. At the other extreme, one can prevent *Abstract*

from ever aborting: this is exactly BFT-SMR. Interesting instances are those in between, e.g., (a) an *Abstract* instance that cannot abort if there is no concurrency, asynchrony or failures, or (b) one that can abort only if there is asynchrony or failures.

When a particular instance of *Abstract* aborts a client request, *Abstract* returns an *unforgeable* (digitally signed) request history that can be used by the client to "recover" using another instance of *Abstract*. This paves the path to the *composability* of *Abstract*. Any composition of *Abstract* instances is possible; we expect many of these to lead to interesting flexible BFT-SMR algorithms. In fact, and to illustrate *Abstract* composability, this thesis presents *Modular BFT-SMR*: a BFT-SMR algorithm built using two *Abstract* instances: (i) the first, which is denoted as *any Abstract*, would typically be an *Abstract* with a weak non-triviality condition that can be implemented very efficiently in a *speculative* [KAD+07] and optimistic manner, whereas (ii) the second is a stronger *Abstract* (called *Backup*) with a non-triviality property that guarantees not to abort a certain number of requests k; this can easily be implemented on top of *any* BFT-SMR algorithm (e.g., [AGG+05, CL99, CML+06, KAD+07]).

Such a modular approach allows for "black-box" code reuse and can significantly reduce the development cost of new BFT-SMR algorithms. Namely, all BFT-SMR algorithms we looked at (e.g., [AGG+05, CL99, CML+06, KAD+07]) consist of more than 20.000 lines of C++ code. Moreover, each of these algorithms assumes certain "normal" conditions; as soon as these conditions are not met, the algorithm fails to deliver its optimal performance.

On the other hand, developing new and mimicking existing BFT-SMR algorithms using our *Modular BFT-SMR* scheme requires significantly less code (sometimes less than 25%, as detailed in Chapter 6). Furthermore, the *Abstract* composability allows for modular code extension in case of fluctuating "normal" conditions: it suffices to switch to another, small-footprint, optimistic *Abstract* box, which can be often implemented in a purely asynchronous system, without worrying, e.g., about the complex BFT view-change and leader election [CL99] mechanisms (captured within the *Backup Abstract* in our *Modular BFT-SMR* scheme).

We also illustrate how new *Abstract*-based BFT-SMR algorithms can be developed by giving two optimistic *Abstract* implementations and plug these into our *Modular BFT-SMR* scheme. The first *Abstract* implementation (called *Decentralized Abstract* or simply *DEC*) leverages our experience with refined quorum systems (Section 1.3.1) to build a very optimistic, yet optimally resilient BFT-SMR algorithm that outperforms all algorithms we know of in terms of time complexity (latency) when there is no concurrency, asynchrony or failures. Its C++ implementation improves the latency of [KAD+07] and [AGG+05] in such executions by more than 33%. Our second optimistic *Abstract* implementation, called *Chain* is the first chain-based [vRS04] BFT-SMR algorithm; it also outperforms all algorithms we know of in terms of peak throughput in synchronous and failure-free executions. Its C++ implementation improves the throughput performance of [KAD+07] and [CL99] by up to 375%.

1.4 Roadmap

Chapter 2 of our thesis gives our system model and important definitions used in the remainder of the thesis. Chapter 3 presents our notion of refined quorum systems (RQS) and gives our best-case optimal algorithms that use RQS. Chapter 4 presents our safe and regular storage algorithms built using high-resolution timestamps. These combine worst-case optimal time complexity and optimal resilience. We extend these results in Chapter 5 by establishing a tight bound on fast BFT atomic storage algorithms. Chapter 6 presents our *ABsTRACT* abstraction, as well as our new throughput and latency optimal BFT-SMR algorithms built around it. The thesis concludes in Chapter 7.

2

Preliminaries

2.1 Processes and Algorithms

We model processes as deterministic I/O automata [LR89]. Processes are interconnected with point-to-point communication channels and communicate via message-passing. The state of the communication channel $ch_{p,q}$ between processes p and q is modeled as a set $mset_{p,q} = mset_{q,p}$ containing messages that are sent by p to q (or vice versa) but not yet received processes (p and q are called *ends* of the communication channel $ch_{p,q}$). We assume that every message m has two tags which identify the sender and the receiver of the message.

We model a distributed algorithm as a collection of automata A_p, each assigned to a process p. A computation of a process p proceeds in *steps* of A_p. A step of A is denoted by a pair of process id and message set $< p, M >$ (M might be $\emptyset$). In step $sp =< p, M >$, a benign process p atomically does the following (we say that p *takes* step sp): (1) (*receive substep*) p removes the messages of M from $mset_{p,*}$ (we also say: p *receives* the messages of M), (2) (*computation substep*) p applies M and its current state st_p to A_p, which outputs a new state st'_p and a set of messages M' to be sent, and p adopts st'_p as its new state and (3) (*send substep*) puts the output messages (M') in $mset_{p,*}$ (we also say: p *sends* the messages of M'). We say that channels are *reliable* if, for every message m and every process p and q, if there is a step by q that puts a message m in $mset_{q,p}$ such that p is the receiver of m and both q and p take an infinite number of steps of A_q and A_p (respectively), then there is a subsequent step $< p, M >$ such that $m \in M$ (i.e., eventually, p receives m).

A *Byzantine* process [LSP82] p_B (also called arbitrary faulty [JCT98]) does not need to follow an automaton A_{p_B} assigned to it. In this thesis, we distinguish two models that define the possible behavior of Byzantine processes:

1. In the first model, called *unauthenticated*, a Byzantine process p_B can perform arbitrary *actions*: (a) p_B can remove/put an arbitrary message m from/into $mset_{p_B,*}$ at an arbitrary time t, and (b) p_B can change its state in an arbitrary manner. Still we assume that channels are *secure*, i.e., that a Byzantine process p_B cannot put (resp., remove) any message into (resp., from) the channel p_B is not an end of (in practice, this is often achieved

using message authentication codes (MACs) [MVO96]). Throughout the thesis, unless explicitly stated otherwise, we assume an unauthenticated Byzantine failure model.

2. The second model, called *authenticated* is strictly stronger than the unauthenticated model. Namely, in the authenticated model, we assume that every non-Byzantine process (such processes are also called *benign*) can produce cryptographic digital signatures. The functionality of the digital signature scheme provides two operations: σ for signing and V for signature verification. The invocation of σ takes a process ID, say p and a bit string m as parameters and returns a bit string sig, called *signature*. The verification operation V takes a process ID p, and two bit strings m and sig as parameters and returns a boolean. The verification function has the property that $V(p, m, sig)$ invoked by a benign process evaluates to $true$ if and only if process p executed $\sigma(p, m)$ in some previous step. Furthermore, no process (including Byzantine ones) other than p may invoke $\sigma(p, m)$ (we say signatures are *unforgeable*); hence, alternatively, we also write $\sigma(p, m)$ as $\sigma_p(m)$ In the following, we refer to a pair $(m, \sigma_p(m))$ as m_{σ_p} (we say that m is an *authenticated* or *(digitally) signed* message). Moreover, if a benign process can evaluate $V(p, m, sig)$ to $true$ (i.e., if $sig = \sigma_p(m)$), we say that a pair (m, sig) contains a *valid* signature (or, alternatively, that (m, sig) is *valid*).

Given any algorithm A, an *execution* of A is an infinite sequence of steps of A taken by benign processes, and actions of Byzantine processes, such that the following properties hold for every benign process p: (1) initially, for each benign process q, $mset_{p,q} = \emptyset$, (2) the current state in the first step of p is a special state *Init*, and (3) for each step $< p, M >$ of A, and for every message $m \in M$, p is the receiver of m and $\exists q, mset_{p,q}$ that contains m immediately before the step $< p, M >$ is taken. A process p is *correct* in execution ex if p takes in ex an an infinite number of steps of A_p. Moreover, we say that a benign process p fails by *crashing* in ex if p takes a finite number of steps in ex. We say that a process is *faulty* (in execution ex) if p is Byzantine or if it fails by crashing (in ex).

A *partial execution* is a finite prefix of some execution. A (partial) execution ex *extends* some partial execution ex' if ex' is a prefix of ex. At the end of a partial execution, all messages that are sent but not yet received are said to be *in transit*.

2.2 Time Complexity

We assume that the system is asynchronous: there is no bound on message propagation delays (i.e., we consider *asynchronous algorithms*). However, for ease of presentation, we sometimes refer to a global clock that is not accessible to processes.

In this thesis we rely on the definition of *time complexity* of an asynchronous algorithm of [Awe85]. The (worst-case) *time complexity* (or *latency*) of an asynchronous algorithm is the worst-case number of time units over all possible ex-

ecutions from the start to the completion of the algorithm, assuming that the *propagation delay* of every message sent from a correct process p to another correct process q is at most one time unit (of course, an asynchronous algorithm must be correct with arbitrary propagation delays). In the above definition, the *propagation delay* of message m is defined as the difference between the time receiver q receives m and the time sender p sends m. We sometimes refer to latency of k time units as k *message delays*.

We also speak about *best case latency*; this is the worst case number of time units measured not over all possible execution, but rather over a subset of executions that satisfy certain constraints (e.g., when the system is *synchronous* and there is no concurrency or failures). In particular, we say that an asynchronous system is *synchronous* (during time interval $[t, t']$) if, for every two correct processes p_1 and p_2, the propagation delay of every message sent by p_1 to p_2 (during time interval $[t, t']$) is at most one time unit Δ, where Δ is known to all correct processes. Similarly, we say that a set of processes P is synchronous if the above holds for every two correct processes $p_1, p_2 \in P$.

Moreover, we assume that, when the system is synchronous, it takes negligible time for a correct process p to take a step (by doing this, our latency metric becomes equivalent to the notion of the *latency degree* of [Sch97]). This applies to both unauthenticated and authenticated model; however, it is worth noting that, in practice, all known techniques that allow for the assumption on unforgeable signatures are computationally expensive. Therefore, verifying and, in particular, sending signed messages in the authenticated model is considerably more computationally expensive than receiving/sending plain messages in the unauthenticated model. Hence, the main focus of this thesis is to provide latency efficient techniques that can be applied to the more general of two models, i.e., the unauthenticated model.

2.3 Distributed Storage

A distributed storage (or, simply, storage, also called a *register* [Lam86]) can be viewed as a read/write abstraction implemented by a finite set S of processes called *base objects* (we use also sometimes the notion of *servers*), and a distinct, potentially unbounded, set of processes called *clients*. We assume that any particular client may be faulty, in contrast to only a fraction of base objects. Precise assumptions on the number of allowed base object failures are problem specific and are detailed in the following Chapters; however, we say that a storage algorithm is *optimally resilient* if it tolerates the maximum possible number of base object failures.

We further assume single-writer multi-reader (SWMR) storage, i.e., storage in which the set clients has two distinct subsets, a singleton *writer* and a set *readers* (with cardinality R). We assume clients and servers are related with reliable point-to-point channels (defined as in Section 2.1). In this thesis we assume that base objects do not intercommunicate (i.e., we assume no channels among base objects). By doing this, we follow the frequently assumed model in distributed and cluster-based storage research, motivated by storage area networks (SANs)

and network attached storage (NAS) (see e.g., [GWGR04, ACC+05, CM05]), in which base objects model commodity disks with read-modify-write (see [AW98]) capabilities [RFGN01]. While we give all our storage related results in the above model, we also discuss how our respective results extend to the stronger model where base objects can communicate among themselves as the first class processes (we call such model the *server-centric* storage model).

A read/write storage is a shared object consisting of the following:

1. set of values D, and a special value $val_0 = \perp \notin D$ (called the initial value),
2. set of operations $write(v)$, $v \in D$ and $read()$
3. set of responses $D \cup \{ack\}$,
4. sequential specification of storage is any sequence of read/write operations such that the responses of operations comply with the following:
   a) $write(v) \triangleq x := v$; return ack (where x is initialized to v_0)
   b) $read() \triangleq$ return x

To access the storage, a client *invokes* an *operation execution* by taking a step called an *invocation step* (when there is no risk of ambiguity we say *operation* when we should be saying *operation execution*). Clients access the storage through two operations: (1) write(v) (invoked by the writer), to write a value v in storage, and (2) read() (invoked by readers), to read the value from storage. We say that an operation *op* invoked by client c is *complete* if c takes a special *response step* for *op*. In this thesis, we focus solely on *wait-free* [Her91] storage algorithms in which every read/write operation invoked by a correct client eventually completes.

An algorithm execution ex is said to be *well-formed*, if (a) no benign client c invokes a new operation in ex before all operations previously invoked by c have completed in ex, and (b) no operation completes at a benign client before it is invoked. In this thesis we consider only well-formed executions. An operation *op* is said to be *pending* in an execution ex, if ex contains the invocation step of *op*, but not its response step.

A *history* of a (partial) execution is a sequence of invocation and response steps of read or write operations in the same order as they appear in the (partial) execution. 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 $H2$.

Moreover, we say that a complete operation *op precedes* an operation op' (or, alternatively, that op' *follows op*) in execution ex (these definitions also extend to execution histories) if the response step for *op* precedes the invocation step of op' in ex; we denote this by $op \rightsquigarrow_{ex} op'$, where the index ex is omitted in places where the execution (or the history) is evident from the context). Let *op* and op' be two invoked operations in ex; if neither $op \rightsquigarrow_{ex} op'$), nor $op' \rightsquigarrow_{ex} op$), we say that *op* and op' are *concurrent* (in ex). In addition, we say that an operation *op* is *uncontended* if *op* is not concurrent with any write operation. We also say that *op* is *synchronous* if the system is synchronous during the interval between the invocation and completion of *op*.

2.3.1 Fast Storage Implementations

In our storage model, in which there is no base object intercommunication, and in which any particular client may fail, the message exchange pattern is that of *communication round-trips* (we also say simply *rounds*), in which: (i) a client c sends a message to (a subset of) base objects, (ii) a base object on reception of a message from c, processes the received message (in a computation substep) and sends a reply to c, and (iii) c collects certain number of such replies before possibly proceeding to a subsequent round.

Therefore, we say that a storage implementation (algorithm) A completes in n communication round-trips (rounds) if its time complexity is $2n$. This naming convention extends to individual read/write operations as well. Moreover, we say that a storage implementation is *fast* if it completes in a single round. We also extend this notion to individual read/write operations.

Furthermore, we generalize the notion of *fast* storage algorithms in two ways. First, we define $(m, \boldsymbol{P})$–*fast* storage algorithms in the following way (where $\boldsymbol{P}$ is a set of sets of base objects):

Definition 1. $(m, \boldsymbol{P})$–**fast *storage algorithm***. *Consider any synchronous and uncontended operation op invoked by a correct client. We say that a storage algorithm A is* $(m, \boldsymbol{P})$–fast *if in every execution of A in which some set $P \in \boldsymbol{P}$ contains only correct base objects, op completes in at most m rounds.*

Second, we say that a storage implementation is a *fast read* (resp., *fast write*) implementation if all read (resp., write) operations complete in a single round. In other words, a fast storage implementation is at the same time a fast read and a fast write implementation. However, a fast read implementation needs not necessarily be a fast write one (and vice versa).

2.3.2 Safe, Regular and Atomic Storage

The notions of *safe*, *regular* and *atomic* storage were introduced by Lamport in [Lam86]. A storage algorithm A is safe (resp., regular, atomic), if every execution of A satisfies *safety* (resp., *regularity*, *atomicity*). In the following, we give definitions of *safety*, *regularity* and *atomicity* restricted to our single-writer setting, in which write operations in (well-formed) executions have a natural ordering which corresponds to their physical order. Denote by wr_k the k^{th} write an execution ($k \geq 1$), and by val_k the value written by wr_k. Recall that $val_0 = \perp$.

A (partial) execution satisfies *safety* if every uncontended read operation returns (1) a value val_k, if there is a write wr_k such that $wr_k \rightsquigarrow rd$ (i.e., wr_k precedes rd) and there is no $l > k$ such that $wr_l \rightsquigarrow rd$ (we say that wr_k is the *last preceding write* for rd), or (2) val_0 in case there is no such a write. A read concurrent with a write is allowed to return any value.

Similarly, we say that a (partial) execution ex satisfies *regularity* if it satisfies *safety* and, if every contended read returns a value written by one of the concurrent writes or the value written by the last preceding write.

Finally, we define the *atomicity* property. Roughly speaking, an execution ex satisfies atomicity if ex satisfies regularity and *read inversion* does not occur in

ex, i.e., if a read rd' follows some other read rd, then rd' does not return an older value than rd.

More formally, a (partial) execution satisfies atomicity, if for every history H' of any of its partial executions, there is a history H that completes H' and H satisfies the properties A1-A3 below (Lemma 13.16 of [Lyn96]). 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: (A1) if $op1$ precedes $op2$ in H then it is not the case that $op2 \prec op1$, (A2) 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 (A3) the value returned by each read operation is the value written by the last preceding write operation according to $\prec$ (or $val_0 = \perp$ if there is no such write operation).

In our single writer setting, the above sufficient condition for atomicity can be considerably simplified using the natural ordering among writes already discussed above. Indeed, consider a relation $\prec$ such that $op_1 \prec op_2$ if and only if op_1 returns val_i and op_2 returns val_j such *smaller* that $i < j$ (here 'returned' stands for 'written' in case an operation is a write). Then, it is straightforward to show that the $\prec$ is a partial ordering that satisfies atomicity properties A1-A3, if the following properties are satisfied:

- (SWA1) If a read returns, it returns a value written by a writer (i.e., some val_i, for $i > 0$) or val_0.
- (SWA2) If a read rd is complete and $wr_k \rightsquigarrow rd$ (i.e., rd follows write wr_k), then rd returns val_l such that $l \geq k$.
- (SWA3) If a read rd returns val_k ($k \geq 1$), then $wr_k \rightsquigarrow rd$ (i.e., wr_k precedes rd) or wr_k is concurrent with rd.
- (SWA4) If some read $rd1$ returns val_k ($k \geq 0$) and a read $rd2$ returns val_l and $rd_1 \rightsquigarrow rd_2$ then $l \geq k$.

Indeed, it is straightforward to see that property (A1) of $\prec$ is implied by the above properties (SWA2) and (SWA4), whereas property (A3) is implied by properties (SWA1), (SWA2) and (SWA3). Finally, Property (A2) follows immediately from our definition of $\prec$, the ordering of write operations and (SWA1).

2.4 Consensus

Our consensus framework is composed of three sets of processes: *proposers*, *acceptors* and *learners* [Lam98]. Roughly, proposers propose values that are to be agreed upon by learners, where the role of acceptors is to help learners agree. Consensus exports one operation: propose(v), that can be invoked only by proposers (we say that a proposer p *proposes* v), whereas it returns a value at every learner (we say that a learner l *learns* v). We assume that every proposer p is initialized with a single proposal value and all processes are interconnected with unreliable point-to-point communication channels. In this thesis, as in for example [YMV+03], we assume that the set *acceptors* does not intersect with the set $proposers \cup learners$, i.e., no proposer or learner can be an acceptor (note that

we allow a proposer to be also a learner). We assume that any proposer/learner can be Byzantine, whereas only a fraction of acceptors can be crash-faulty or Byzantine (as detailed later).

An algorithm A solves consensus if every (partial) execution of A satisfies the following properties.

- *(Validity)* If a benign learner learns a value v and all proposers are benign, then some proposer proposed v;[1]
- *(Agreement)* No two benign learners learn different values;
- *(*Termination) If a correct proposer proposes a value, then eventually, every correct learner learns a value.

To circumvent the impossibility of an asynchronous consensus [FLP85] with faulty processes, we assume that the system is eventually synchronous [DLS88, Gaf98]. Eventual synchrony means that there is a point in time GST (Global Stabilization Time), not known to processes, such that, after GST, the system is synchronous (i.e., the system can be asynchronous for arbitrarily long, yet finite period of time). In addition, we assume that no message sent between two correct processes before GST, is received after GST.

[1] Some similar definitions of *Validity* property, e.g., the one in [Lam03]: "Only a value proposed by a proposer can be learned", are impossible to ensure in the presence of Byzantine proposers. Intuitively, a Byzantine proposer p_B cannot be restricted of invoking *propose* with a value v and than immediately replacing v with a different value v', as if p_B were correct and proposed v'.

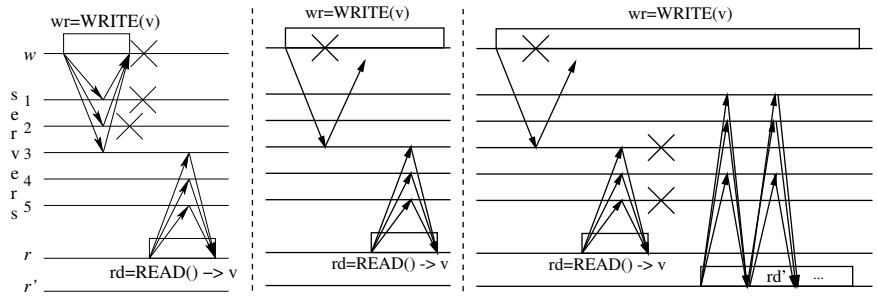
3

Refined Quorum Systems

3.1 Introduction

To illustrate the motivation behind refined quorum systems, consider the simple context of a crash-resilient asynchronous implementation of an atomic storage. For instance, the classical, optimally crash-resilient single-writer multi-reader (SWMR) solution [ABD95] (that assumes a majority of correct base objects) requires two rounds for a read, in contrast to only a single round for writes.

As we discussed in Chapter 1, it is practically appealing to look into best-case complexity and ask if it is possible to expedite *both* reads and writes within a single round in a synchronous and contention-free period. Clearly, if the reader (resp. the writer) access all base objects in the first round, then it can immediately return a valid response. But do we need to access all base objects for that? How many base objects actually *need* to be accessed to achieve such fast read/write operations under best-case conditions?



(a) ex2 (b) ex3 (c) ex4

Figure 3.1: Violation of atomicity in case the single-round operations access only 3 base objects.

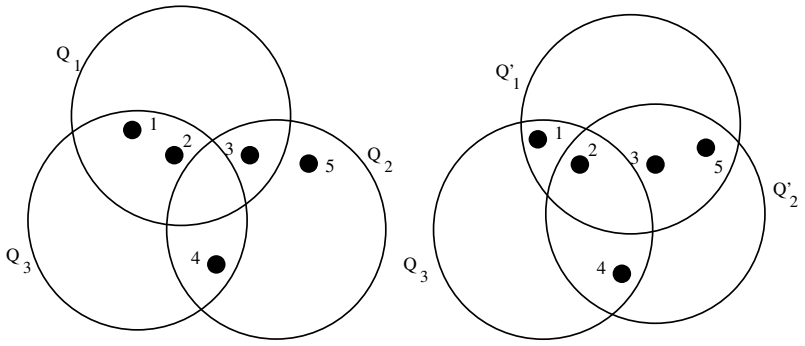
Consider 5 base objects implementing a crash-tolerant atomic storage assuming $t = 2$ base object failures (optimal resilience). We argue below that any algorithm that greedily expedites synchronous and uncontended read/write operations in one round whenever $S - t = 3$ base objects are accessed, violates atomicity. This is depicted through several executions of such an algorithm (Figure 3.1):

1. In the first execution ($ex1$), the writer w invokes $wr = \mathsf{write}(v)$ and base objects 4 and 5 are faulty. Then, wr writes value v into the subset of base objects $Q_1 = \{1, 2, 3\}$ and completes in a single round.

2. The second execution ($ex2$, Fig. 3.1(a)) is slightly different because base objects 4 and 5 are actually correct. Yet wr also completes in a single round, after writing in Q_1, since the writer cannot distinguish ex_1 and ex_2. Then, base objects 1 and 2 crash, as well as the writer, and a read rd (by reader r) is invoked. Assuming synchrony, rd accesses base object set $Q_2 = \{3, 4, 5\}$ and (since there is no contention) completes in a single round. Notice that, since rd follows wr, rd must return v.

3. The third execution ($ex3$, Fig. 3.1(b)) is similar to $ex2$ except that (1) the write is incomplete (the writer crashes while executing wr) and writes only to base object 3, (2) base objects 1 and 2 (i.e., base objects from set $Q_1 \setminus Q_2$) are correct, but the communication between the reader and the base objects from $Q_1 \setminus Q_2$ is delayed. Read rd does not distinguish $ex3$ from $ex2$ and completes in a single round, returning v.

4. Finally, the fourth execution ($ex4$, Fig. 3.1(c)) extends $ex3$ by: (1) the crash of base objects 3 and 5 and (2) the invocation of read rd' by a different reader r'. This reader cannot return v using $Q_3 = \{1, 2, 4\}$ regardless of how many rounds are used. Atomicity is violated.



(a) $|Q_1| = |Q_2| = 3$ (b) $|Q'_1| = |Q'_2| = 4$

Figure 3.2: Quorum intersections

Essentially, atomicity is violated because $Q_1 \cap Q_2 \cap Q_3 = \emptyset$ (Figure 3.2(a)). On the other hand, we can devise a storage algorithm in which reads and writesare fast whenever 4 base objects are accessed. For instance:

- A write wr completes in a single round only if it writes v to 4 base objects, say $Q'_1 = \{1, 2, 3, 5\}$. A subsequent single-round read rd will also have to access at least 4 base objects, say $Q'_2 = \{2, 3, 4, 5\}$ (including at least 3 base objects from Q'_1). A subsequent read rd' that accesses some subset Q_3 of 3 base objects will surely learn about v since there is a set $X = Q'_1 \cap Q'_2$ of (at least) 3 base objects that witnessed both wr and rd, and X intersects with

any set of 3 base objects. This base object in the intersection will inform rd' about the value written by wr.

The key to ensuring atomicity is to have $Q'_1 \cap Q'_2 \cap Q_3 \neq \emptyset$. Namely, (Figure 3.2(b)) in a system of 5 elements, any two subsets of 4 elements intersect with any subset of 3 elements. Basically, boosting complexity requires to access subsets of base objects that have larger intersections than traditional quorums. The above example is (relatively) simple because we considered: a) crash failures only, b) a threshold adversary (at most t faulty processes) and c) no *graceful degradation*, i.e., achieving the next best possible latencies, when the best possible one (e.g., a single round) cannot be achieved.

Our motivation is precisely to characterize the required intersection properties in a precise and general manner. We aim at a characterization that is necessary and sufficient for optimizing the best-case complexity of various distributed object implementations (with emphasis on atomic storage and consensus), in various failure models, under various adversary structures, and also considering graceful degradation. We believe that this characterization is provided by our notion of *refined quorum systems (RQS)*.

3.1.1 Contributions

As we already mentioned in the introduction in Chapter 1, a refined quorum system of some set of elements S is a set of three classes of subsets (quorums) of S: first class quorums are also second class quorums, which are also third class quorums. In the context of a distributed object implementation, set S would typically contain the set of fault-prone base object processes over which some object abstraction (e.g., storage or consensus) is implemented.

Under uncontended and synchronous conditions, a distributed object implementation would expedite an operation if a quorum of the first class is available, achieving the *best possible* latency, and then degrade gracefully depending on whether a quorum of the second or the third class is available. We argue that our quorum notion is, in a sense, complete: there is no need for further refinement of quorums with the goal of optimizing best-case efficiency. Indeed, as we explain in the following, the properties provided by our third class quorums are anyway necessary for hindering the partitioning of the asynchronous system, which is key to any resilient distributed object implementation. Moreover, and as we show in the following, optimally resilient and best-case efficient implementations of the seminal register and consensus abstractions have exactly *three* possible latencies under uncontended and synchronous conditions. This observation is of independent interest.

We illustrate the power of our notion of refined quorum systems by introducing two new atomic object implementations. The first algorithm illustrates the case of a quorum system accessed by benign (non-Byzantine) clients, where the second alleviates the need for this assumption.

- Our first object implementation is a new BFT asynchronous distributed atomic storage algorithm that uses a refined quorum system. Our algorithm

combines optimal resilience with the lowest possible *read/write* latency in best-case conditions (no-contention and synchrony). Under such conditions, our algorithm expedites storage operations (*reads* and *writes*) in a single communication round-trip (round) if a first class quorum is accessed, in two rounds if a second class quorum is accessed and in three rounds otherwise. The latter case is when a third class quorum is available which is a necessary condition for resilience anyway. Our algorithm assumes an unauthenticated model (i.e., does not rely on using messages authenticated using digital signatures), and matches the resilience lower bound of [MAD02a] (even when this bound is extended to a general adversary structure), together with two new time/complexity lower bound we establish here. Our new bounds capture the best-case complexity of fast and gracefully degrading atomic storage implementations.

- Our second algorithm implements a BFT consensus abstraction in the general consensus framework of [Lam98], distinguishing different process roles: *proposers* that propose values to be learned by *learners* with the mediation of *acceptors*, as described in Chapter 2. Our algorithm is the first to tolerate (1) *any number* of Byzantine failures of proposers and learners, (2) the largest possible number of acceptor failures (i.e., our algorithm is optimally resilient), and (3) arbitrarily long periods of asynchrony. On the other hand, under best-case conditions, our algorithm allows a value to be learned in only two message-delays in case a first class quorum is accessed, and in three (resp., four) message delays in case a second (resp., third) class quorum is accessed. Note here that (a) learning in a single message delay is obviously impossible with multiple or potentially Byzantine proposers, and (b) the availability of a third class quorum is anyway necessary for resilience. Our algorithm matches the resilience and complexity lower bounds of [Lam03] (including when these bounds are extended to a general adversary structure), together with a new complementary bound we establish here on consensus algorithms that degrade gracefully in best-case executions. These bounds state minimal conditions under which consensus (and consequently, the state-machine replication approach) can be made optimally resilient and best-case efficient. Until now, it was not clear whether the conditions of [Lam03] were also sufficient. We show they are and we complement them.

We believe that it would have been very hard to devise such algorithms, especially in the context of a general adversary structure, without the notion of a refined quorum system, though we might be subjective here.

3.1.2 Roadmap

The rest of the Chapter is organized as follows. Section 3.2 first presents our quorum notion and illustrates how it generalizes previous ones through examples from the literature. Sections 3.3 and 3.4 introduce our two new distributed object implementations that exploit the full features of refined quorums. Proofs of

correctness of our two algorithms are postponed to the end of the chapter for better readability.

3.2 Refined Quorum Systems

In the following, sets of sets of elements are written in boldface for better readability; e.g., $\boldsymbol{S}= \{S_1, S_2\}$, where S_i denotes a set of (base) elements.

3.2.1 Definitions

A refined quorum system is expressed in the abstract context of a non-empty set S of elements, and an *adversary structure* (or, simply, *adversary*) $\boldsymbol{B}$ defined as follows [HM97]:

Definition 2. *Let $\boldsymbol{B}$ be any set of subsets of S. $\boldsymbol{B}$ is an* adversary *(for S) if:* $\forall B \in \boldsymbol{B}$: $B' \subseteq B \Rightarrow B' \in \boldsymbol{B}$.

Let $\boldsymbol{RQS}$ be any set of subsets of S.

Definition 3. ***Refined Quorum System.*** *We say that $\boldsymbol{RQS}$ is a* refined quorum system *for a set S and adversary $\boldsymbol{B}$, if $\boldsymbol{RQS}$ has two subsets $\boldsymbol{QC_1} \subseteq \boldsymbol{QC_2} \subseteq \boldsymbol{RQS}$ such that the following properties hold: (every $\boldsymbol{QC_i}$ is called a* quorum class*, and elements of $\boldsymbol{QC_i}$ are called* class i elements*)*

Property 1. *The intersection of any two elements of $\boldsymbol{RQS}$ does not belong to $\boldsymbol{B}$, i.e.,*

- $\forall Q, Q' \in \boldsymbol{RQS}$: $Q \cap Q' \notin \boldsymbol{B}$.

Property 2. *The intersection of any two class 1 elements and any element of $\boldsymbol{RQS}$ is not a subset of the union of any two elements of $\boldsymbol{B}$, i.e.,*

- $\forall Q_1, Q_1' \in \boldsymbol{QC_1}$, $\forall Q \in \boldsymbol{RQS}$, $\forall B_1, B_2 \in \boldsymbol{B}$: $Q_1 \cap Q_1' \cap Q \nsubseteq B_1 \cup B_2$.

Property 3. *The intersection of any class 2 element Q_2 and any element Q of $\boldsymbol{RQS}$ is:*
(a) not a subset of the union of any two elements of $\boldsymbol{B}$ (we say $P_{3a}(Q_2, Q)$ holds),
or
(b) its intersection with every class 1 element[1] is not an element of $\boldsymbol{B}$ (we say $P_{3b}(Q_2, Q)$ holds), i.e.,

- $\forall Q_2 \in \boldsymbol{QC_2}$, $\forall Q \in \boldsymbol{RQS}$, $\forall B_1, B_2 \in \boldsymbol{B}$:
  $(Q_2 \cap Q \nsubseteq B_1 \cup B_2) \vee (\boldsymbol{QC_1} \neq \emptyset \wedge \forall Q_1 \in \boldsymbol{QC_1}$: $Q_1 \cap Q_2 \cap Q \notin \boldsymbol{B})$.

We simply call elements of a refined quorum system — *quorums*, and we sometimes refer to any quorum that is not a class 2 quorums as a *class 3* quorum (we write $\boldsymbol{QC_3} = \boldsymbol{RQS}$). Note that class 1 quorums are also class 2 quorums, which are also class 3 quorums. Notice also that, when $\boldsymbol{QC_1} = \boldsymbol{QC_2}$, Property

[1] Assuming there is at least one class 1 element, i.e., $\boldsymbol{QC_1} \neq \emptyset$.

2 implies Property 3. Furthermore, when $\boldsymbol{B} = \emptyset$, Property 1 implies Property 3. Therefore, Property 3 is interesting on its own only if $\boldsymbol{B} \neq \emptyset$ and $\boldsymbol{QC_1} \neq \boldsymbol{QC_2}$.

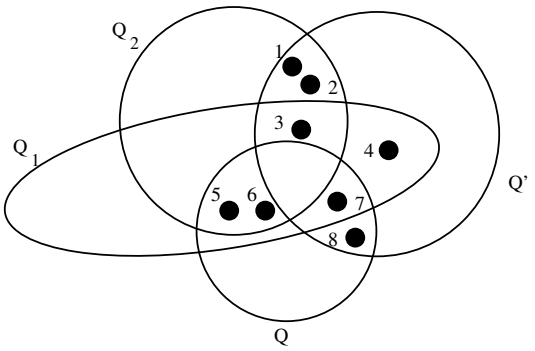
To get an intuition of these properties, we instantiate them here in the context of a *k-bounded threshold adversary*, denoted $\boldsymbol{B_k}$. This is a special case of an adversary that contains all subsets of S with cardinality at most k (i.e., $\boldsymbol{B_k} = \{B | B \subseteq S \wedge |B| \leq k\}$). In this context, the RQS properties can be expressed as follows:

Property 1. Any two quorums intersect in at least $k + 1$ elements.

Property 2. The intersection of any two class 1 quorums intersects with any quorum in at least $2k + 1$ elements.

Property 3. Any class 2 quorum intersects with any quorum in at least $2k + 1$ elements or this intersection itself intersects with any class 1 quorum in at least $k + 1$ elements.

3.2.2 Examples



Q={5,6,7,8}

Q'={1,2,3,4,7,8}

Q_2={1,2,3,5,6}

Q_1={3,4,5,6,7}

WQS= {Q,Q',Q_2,Q_1}
QC2= {Q_2,Q_1}
QC1= {Q_1}

∩	Q	Q'	Q_2	Q_1
Q	Q	{7,8}	{5,6}	{5,6,7}
Q'	{7,8}	Q'	{1,2,3}	{3,4,7}
Q_2	{5,6}	{1,2,3}	Q_2	{3,5,6}
Q_1	{5,6,7}	{3,4,7}	{3,5,6}	Q_1

Figure 3.3: Example of a RQS for the threshold adversary $\boldsymbol{B_k}$ ($k = 1$).

Example 1. Figure 3.3 depicts a simple illustration of a refined quorum system for the *1-bounded threshold adversary* $\boldsymbol{B_1}$: 4 quorums are involved. Every pair of depicted sets intersects in at least $k + 1$ elements (satisfying Property 1). Q_1 intersects with every other set in at least $2k + 1$ elements (satisfying Property 2, for an intersection with itself). Moreover, $P_{3a}(Q_2, Q')$ and $P_{3a}(Q_2, Q_1)$ hold (since

$|Q_2 \cap Q'| = 2k+1 = |Q_2 \cap Q_1|$) as well as $P_{3b}(Q_2, Q)$ (since $|Q_2 \cap Q \cap Q_1| = k+1$). Hence, $\boldsymbol{RQS} = \{Q, Q', Q_2, Q_1\}$ is a refined quorum system, where Q_1 (resp., Q_2) is a class 1 (resp., class 2) quorum.

As depicted by the example, the cardinality of a quorum is not always a good indication of its class: it is the intersection with others that matters. Quorum Q_1 contains 5 elements and is a class 1 quorum, while quorum Q' contains 6 elements yet is only a class 3 quorum.

In the following, we give more illustrations of our quorum notions by explaining how it extends traditional ones. Later in the Chapter, we will introduce new optimal algorithms that make full use of our quorum notion. In the following, we consider that an adversary $\boldsymbol{B}$ for a set of processes S contains all subsets of S that can simultaneously be Byzantine. In our description, a process that simply fails by crashing is not called Byzantine. We also denote by $\boldsymbol{Q_i}$ the set of subsets of S that contains all subsets of S that contain all but at most i elements of S, i.e., $\boldsymbol{Q_i} = \{P | P \subseteq S \wedge |P| \geq |S| - i\}$.

Example 2. Consider the case where: (a) $\boldsymbol{B} = \{\emptyset\}$, (b) $\boldsymbol{QC_1} = \boldsymbol{QC_2} = \emptyset$ and (c) $\boldsymbol{RQS} = \boldsymbol{Q}_{\lfloor(|S|-1)/2\rfloor}$. In other words, every majority subset of S is a quorum. Property 1 is trivially satisfied. So are Properties 2 and 3, since $\boldsymbol{QC_1} = \boldsymbol{QC_2} = \emptyset$. This quorum system is typically used when devising algorithms that tolerate (a minority of) crash-failures, e.g., [Tho79, Gif79, NW94, ABD95, CT96, Lam98].

Example 3. Consider the case of an adversary $\boldsymbol{B}_{\lfloor(|S|-1)/3\rfloor}$, where (a) $\boldsymbol{QC_1} = = \boldsymbol{QC_2} = \emptyset$ and (b) $\boldsymbol{RQS} = \boldsymbol{Q}_{\lfloor(|S|-1)/3\rfloor}$. In this case, each quorum contains more than two thirds of processes and satisfies Property 1. Properties 2 and 3 are also satisfied (since $\boldsymbol{QC_1} = \boldsymbol{QC_2} = \emptyset$). Such a quorum system is typically used to tolerate (up to one third of) Byzantine failures, e.g., [PSL80, CL99, MAD02a, CML$^+$06].

Example 4. A refined quorum system for which $\boldsymbol{QC_1} = \boldsymbol{QC_2} = \emptyset$ is a *disseminating quorum system* in the sense of [MR98]. In [MR98], disseminating quorum systems are used to build resilient distributed services that store authenticated (also called self-verifying) data. On the other hand, a refined quorum system in which $\boldsymbol{QC_1} = \emptyset$ and $\boldsymbol{QC_2} = \boldsymbol{RQS}$ is a *masking quorum system* in the sense of [MR98]. These systems have been used to build resilient distributed services that store unauthenticated data.

So far, in examples 2-4, we considered refined quorum systems in which $\boldsymbol{QC_1} = \emptyset$. In the rest of the Chapter, we study the more general case where $\boldsymbol{QC_1} \neq \emptyset$. This is the case where our refined quorum systems capture both the resilience and the best-case complexity dimensions of distributed algorithms.

Example 5. Consider the case of a refined quorum system where $\emptyset \neq \boldsymbol{QC_1} = \boldsymbol{QC_2}$. Such a RQS corresponds to the quorum system used in [Lam06b] for the specific

case $\boldsymbol{B} = \{\emptyset\}$, to devise a consensus algorithm that tolerates asynchronous periods and a threshold t of process (crash) failures, yet expedites decisions in best-case scenarios. In fact, although not used in the algorithm, the idea of a *fast* quorum (class 1 quorum in our terminology) was used to explain the algorithm. In the special case of an adversary $\boldsymbol{B_k}$, where (a) $\boldsymbol{RQS} = \boldsymbol{Q_t}$, and (b) $\boldsymbol{QC_1} = \boldsymbol{QC_2} = \boldsymbol{Q_q}$ ($q \leq t$), Property 2 is satisfied if $|S| > 2q+t+2k$ and Property 1 is satisfied if $|S| > 2t + k$. These inequalities correspond to Lamport's lower bounds for "asynchronous" consensus [Lam03]. The special case of this RQS where $k = q = t$ (i.e., where $\boldsymbol{QC_1} = \boldsymbol{RQS}$) corresponds to the quorum system used in [AGG$^+$05, MA06], also with the goal of boosting consensus latency.

Example 6. Even more interesting is the case where $\emptyset \neq \boldsymbol{QC_1} \neq \boldsymbol{QC_2} \subseteq \boldsymbol{RQS}$ (e.g., Fig. 3.3), especially when $\boldsymbol{RQS}$, $\boldsymbol{QC_1}$ and the adversary are defined as in Example 5, $\boldsymbol{QC_2} = \boldsymbol{Q_r}$, and $0 \leq q < r \leq t$. In other words, each quorum contains all but at most t processes, while class 1 (resp., class 2) quorums contain all but at most q (resp., r) processes. $\boldsymbol{RQS}$ satisfies (i) Property 1 if $|S| > 2t + k$, (ii) Property 2 if $|S| > t + 2k + 2q$, and (iii) Property 3 if $|S| > t + r + k + min(k, q)$, i.e., $\boldsymbol{RQS}$ is a refined quorum system if $|S| > t+k+max(t, k+2q, r+min(k, q))$. This RQS corresponds to the quorum system we used in [DGV05, GLV06], and which was later used in [Zie06].

A very important instantiation of this refined quorum system is the one with $|S| = 3t + 1$ processes out of which t may be Byzantine ($k = t$), and where all quorums are class 2 quorums ($r = t$), whereas the set that contains all processes is a class 1 quorum ($q = 0$). It is important to notice that, in this particular case, there are no genuine class 3 quorums (i.e., class 3 quorums that do not belong at the same time to class 2).

As we just discussed, our RQS notion was implicitly used, in partial forms, in various distributed objects implementations. In the following, we present two new algorithms that make full and explicit use of our notion of RQS.

3.3 Atomic storage

We show in this section how to use a refined quorum system to wait-free implement the abstraction of a SWMR atomic storage with optimal resilience and optimal best-case time complexity (latency). As defined in Chapter 2, optimal resilience means here tolerating the maximal number of base object failures while still ensuring wait-freedom in the face of contention and asynchrony (worst-case conditions).

We present our storage algorithm assuming the unauthenticated model, as specified in Chapter 2, complemented as follows. We assume that communication channels are *reliable*. Denoting the set of base objects by S, and the adversary by $\boldsymbol{B}$, we construct a refined quorum system $\boldsymbol{RQS}$ (obeying the properties defined in Section 3.2) known to all clients. In the above, we assume that, for any execution ex, $B_{ex} \in \boldsymbol{B}$, where B_{ex} contains all base objects Byzantine in ex. Moreover, any number of clients and base objects may fail by crashing, as long as there is at least one quorum in $\boldsymbol{RQS}$ that contains only correct base objects.

In the following, we overview our storage algorithm and then state its optimality. This includes establishing two new tight bounds on latency efficient atomic storage implementations. The full correctness proof of our atomic storage implementation is given in Section 3.5.

3.3.1 Atomic storage algorithm

Our storage algorithm is $(m, \boldsymbol{QC_m})$–*fast* (see Section 2.3.1 for the definition) for all $m \in \{1, 2, 3\}$. Note that this implies that, in our algorithm, all synchronous and uncontended operations complete in at most 3 communication round-trips (we say simply *rounds*).

```
Initialization:
0: ts := ts_0; timeout := 2Δ; QC'_2 := ∅

write(v) is {
1:   inc(ts)
2:   round(1)
3:   if acks received from some class 1 quorum then return(OK)
4:   forall Q_2 ∈ QC_2
5:     if acks received from Q_2 then QC'_2 := QC'_2 ∪ {Q_2}
6:   round(2)
7:   if acks received from some quorum from QC'_2 then return(OK)
8:   QC'_2 := ∅; round(3)
9: return(OK) }

round(ρ) is {
10:  send wr⟨ts, v, QC'_2, ρ⟩ to all base objects
11:  if ρ < 3 then trigger(timeout)
12:  wait for (reception of wr_ack⟨ts, ρ⟩ from some quorum) and (expiration of timeout or ρ = 3) }
```

Figure 3.4: Atomic storage algorithm: writer code

The pseudocode of the algorithm is given in Figures 3.4, 3.5 and 3.6. The write code (Figure 3.4) is simple, thanks to the underlying RQS. The writer maintains a monotonically increasing local timestamp ts that is assigned to the written value v and sent to base objects in every round. A write consists of at most three rounds. In every round ρ, the writer sends a $\mathsf{wr}\langle ts, v, \boldsymbol{QC'_2}, \rho\rangle$ message containing value v to be written, along with timestamp ts and a set of quorums (i.e., quorum ids) $\boldsymbol{QC'_2}$ to all base objects (this set is empty in rounds 1 and 3 and only used in round 2, as explained below). In every round, the writer awaits acks from some quorum, and in the first two rounds, awaits for the timer set to 2Δ.

A write terminates at the end of the first round, if the writer receives acks from some class 1 quorum by the expiration of the timer. If this is not the case, the writer proceeds to round 2. If, in round 1, the writer received acks from some class 2 quorums, the ids of these quorums are added to $\boldsymbol{QC'_2}$ (lines 4-5, Fig. 3.4). If the writer receives again acks from some quorum from $\boldsymbol{QC'_2}$ in round 2 (line 7, Fig. 3.4), the write terminates at the end of round 2. Finally, if this is not the case, the writer proceeds to round 3 and completes at the end of this round, upon reception of round 3 acks from any quorum.

The pseudocode of the base objects is given in Figure 3.5. Upon receipt of the message $\mathsf{wr}\langle ts, v, \boldsymbol{QC'_2}, \rho\rangle$, a base object s_i stores the received data in its $history_i$

matrix, by storing $history_i[ts, j].tsval = \langle ts, v\rangle$ for all $j, 1 \leq j \leq \rho$ and by adding $\boldsymbol{QC'_2}$ to $history_i[ts, \rho].sets$. Then, a base objects sends an ack to the client.

```
Initialization:
1: history_i[*,*].tsval := ⟨ts_0, ⊥⟩; history_i[*,*].sets := ∅

2: upon reception of a wr⟨ts, v, QC'_2, ρ⟩ message from some client c do
3:    for rnd = 1 to ρ do
4:       if history_i[ts, rnd] = ⟨⟨ts_0, ⊥⟩, ∅⟩ or history_i[ts, rnd] = ⟨⟨ts, v⟩, *⟩ then
5:          history_i[ts, rnd].tsval := ⟨ts, v⟩
6:          if rnd = ρ then history_i[ts, rnd].set := history_i[ts, rnd].sets ∪ QC'_2
7:    send wr_ack⟨ts, ρ⟩ to c

8: upon reception of a rd⟨tsr, ρ⟩ message from reader r_j
9:    send rd_ack⟨tsr, ρ, history_i⟩ to reader r_j
```

Figure 3.5: Atomic storage algorithm: base object s_i code

The read is more involved, yet our pseudocode in Figure 3.6 (in the remainder of this section we refer to this Figure) is easy to follow since we assume an RQS. As many other atomic storage algorithms (e.g., [ABD95]) our algorithm consists of two parts: (1) the part that implements regular storage (lines 20-35, along with the predicates defined in lines 3-12) and (2) the part that prevents read inversion and enforces atomicity, in which readers write the value back to a "sufficient" number of base objects (lines 40-49, with predicates in lines 1-2).

In the heart of our algorithm, in particular of the second part that ensures atomicity, lies a *Best-Case Detector* abstraction (BCD, defined by the predicates in lines 1-2). Roughly, BCD detects if a read operation is synchronous and uncontended. The second, critical part of the algorithm is orchestrated around the outcome of BCD, as well as the accessed RQS quorums.

In the following, we explain the intuition behind the best-case latency of read. Consider indeed an uncontended and synchronous read rd. Moreover, let wr be the last write that precedes rd and assume that wr wrote value v with timestamp ts in W rounds. Note that, by atomicity, rd must return v.

In the first part of the algorithm (that implements regular semantics) a reader r_i first invokes series of rounds (lines 22-34) in which r_i sends a rd message containing the unique identifier of the r_i's read operation, denoted by $read\_no$, as well as the current round number of read $read\_no$, rd_rnd. In every round, the reader waits for responses (i.e., rd_ack messages) from some quorum. Moreover, in the first round, the reader waits for acks until the expiration of the timer (lines 27-28) and stores ids of all class 2 quorums it received replies from in the set $\boldsymbol{QC'_2}$ (lines 30-31).

This series of read rounds is invoked until the following conditions become satisfied, for some timestamp/value pair $c = \langle ts', v'\rangle$: (a) $safe(c)$ holds, i.e., c is confirmed in the history of some subset of base objects that is not an element of an adversary (lines 11 and 33-34), and (b) $highCand(c)$ holds, i.e. all values v' with a $ts' > ts$ are *invalid* (lines 9,12 and 33-34). A sufficient condition for a pair $c' = \langle ts', v'\rangle$ to become *invalid* is that the response of every base object from some quorum Q contains only values with timestamps smaller than ts' (or a pair $\langle ts', v'' \neq v\rangle$). In this case none of the predicates $valid_1(c', Q)$ (line 6),

Definitions:
1: $BCD(c,1,W) ::= \exists Q_1 \in \boldsymbol{QC_1}, \exists Q_W \in \boldsymbol{QC_W}, \exists \boldsymbol{QC} \subseteq \boldsymbol{QC_2} \cup \emptyset$:
$(Q_1 \cap Q_W \subseteq \{i : history[i, c.ts, W] = \langle c, \boldsymbol{QC}\rangle\}) \wedge ((W \neq 2) \vee (Q_W \in \boldsymbol{QC}))$
2: $\boldsymbol{BCD}(c,2,W) ::=$
$\{Q_2 \mid (Q_2 \in \boldsymbol{QC'_2}) \wedge (\exists Q_W \in \boldsymbol{QC_W}: Q_W \cap Q_2 \subseteq \{i : history[i, c.ts, W] = \langle c, *\rangle\})\}$

3: $c \preceq c' ::= (c.ts < c'.ts) \vee (c = c')$
4: $last[i,\rho] ::= c : (\forall ts' > c.ts : history[i, ts', \rho].tsval = \langle ts_0, \perp\rangle) \wedge (history[i, c.ts, \rho] = \langle c, *\rangle)$
5: $last_{set}[i, Q] ::= c : (\forall ts' > c.ts : Q \notin history[i, ts', 1].sets) \wedge$
$\wedge(\exists \boldsymbol{QC} \subseteq \boldsymbol{QC_2}: (history[i, c.ts, 1] = \langle c, \boldsymbol{QC}\rangle) \wedge (Q \in \boldsymbol{QC}))$
6: $valid_1(c, Q) ::= \exists T \subseteq Q, \forall s_i \in T : (T \notin \boldsymbol{B}) \wedge (\exists c' : last[i,1].tsval = c' \wedge c \preceq c')$
7: $valid_{2a}(c, Q) ::= \exists s_i \in Q, \exists c' : (last[i,2].tsval = c') \wedge (c \preceq c')$
8: $valid_{2b}(c, Q) ::= \exists s_i \in Q, \exists c', \exists Q_2 \in \boldsymbol{QC_2}: (last_{set}[i, Q_2] = c') \wedge P_{3b}(Q_2, Q) \wedge (c \preceq c')$
9: $invalid(c) ::= \exists Q \in \boldsymbol{RQS}: \neg(valid_1(c, Q) \vee valid_{2a}(c, Q) \vee valid_{2b}(c, Q)) \vee (c'.ts > highest\_ts)$

10: $read(c, i) ::= \exists \rho \in \{1, 2\} : history[i, c.ts, \rho] = \langle c, *\rangle$
11: $safe(c) ::= \{i : read(c, i)\} \notin \boldsymbol{B}$
12: $highCand(c) ::= \forall c', \forall i : read(c', i) \wedge (c'.ts > c.ts) \Rightarrow invalid(c')$

Initialization:
$timeout := 2\Delta$; $history[*, *, *] := \langle\langle ts_0, \perp\rangle, \emptyset\rangle$; $highest\_ts := ts_0$; $read\_no := 0$; $first = 1$; $second = 2$

read() is {
20: $rd\_rnd := 0$
21: **inc**$(read\_no)$
22: **repeat**
23: **inc**(rd_rnd)
24: **if** $rd\_rnd = 1$ **then trigger**$(timeout)$
25: send rd$\langle read_no, rd_rnd\rangle$ to all servers
26: **wait for** receive rd_ack$\langle read_no, rd_rnd, *\rangle$ from some quorum
27: **if** $rd\_rnd = 1$ **then**
28: **wait for** expiration of $timeout$
29: $highest\_ts := last[i, \rho].tsval.ts : (\forall j, \rho' : last[j, \rho'].tsval.ts \leq last[i, \rho].tsval.ts)$
30: **forall** $Q_2 \in \boldsymbol{QC_2}$
31: **if** acks received from Q_2 **then** $\boldsymbol{QC'_2} := \boldsymbol{QC'_2} \cup \{Q_2\}$
32: **endif**
33: $C := \{c : (safe(c) \wedge highCand(c))\}$
34: **until** $C \neq \emptyset$
35: $c_{sel} := c : (c \in C) \wedge (\forall c' \in C : c' \preceq c)$

40: **if** $(\exists i \in \{1, 2, 3\} : BCD(c_{sel}, 1, i))$ **and** $(rd\_rnd = 1)$ **then return**$(c_{sel}.val)$

41: **if** $(\exists i \in \{1, 2, 3\} : \boldsymbol{BCD}(c_{sel}, 2, i) \neq \emptyset)$ **and** $(rd\_rnd = 1))$ **then**
42: **if** $(\exists i \in \{2, 3\} : \boldsymbol{BCD}(c_{sel}, 2, i) \neq \emptyset)$ **then writeback**$(second, c_{sel}, \emptyset)$; **return**$(c_{sel}.val)$
43: **trigger**$(timeout)$
44: **writeback**$(first, c_{sel}, \boldsymbol{BCD}(c_{sel}, 2, 1))$
45: **wait for** expiration of $timeout$
46: **if** acks received from some quorum from $\boldsymbol{BCD}(c_{sel}, 2, 1)$ **then return**$(c_{sel}.val)$
47: **writeback**$(second, c_{sel}, \emptyset)$; **return**$(c_{sel}.val)$
48: **endif**

49: **writeback**$(first, c_{sel}, \emptyset)$; **writeback**$(second, c_{sel}, \emptyset)$; **return**$(c_{sel}.val)$

upon reception of rd_ack$\langle read_no, rd_rnd, history_i\rangle$ from server s_i **do**
50: $history[i] := history_i$
}
writeback$(round, c, Set)$ is {
60: send wr$\langle c.ts, c.val, Set, round\rangle$ message to all servers
61: **wait for** reception of wr_ack$\langle c.ts, round\rangle$ message from some quorum
}

Figure 3.6: Atomic storage algorithm: reader code

$valid_{2a}(c', Q)$ (line 7) or $valid_{2b}(c', Q)$ (line 8) will hold. This sufficient condition is met whenever the read is synchronous and uncontended, since we assume the availability of some quorum that contains only correct base objects. Hence, in this case, the first part of the algorithm (i.e., lines 20-35) is executed only once and, therefore, the first part of rd takes only a single communication round. In this case, the reader selects timestamp-value pair $c_{sel} = \langle ts, v \rangle$ in line 35 (i.e., the timestamp/value pair written by wr, the last write that precedes rd).

Then, the reader proceeds to the second part of the read algorithm (that guarantees atomicity, lines 40-49). Basically, this part of the algorithm is a sophisticated *writeback* procedure, based on the outcome of a BCD: it proceeds as follows. First, the reader queries BCD with $c_{sel} = \langle ts, v \rangle$ as a parameter (line 40). The outcome of BCD governs the remainder of the writeback procedure, namely:

1. If the reader received acks from a class 1 quorum (containing only correct processes) in round 1, then $BCD(c_{sel}, 1, W)$ holds (line 1) and rd completes at the end of round 1, without any additional writeback round (line 40). Notice that, by line 1, $BCD(c_{sel}, 1, W)$ holds only if there is a class 1 quorum Q_1 and a class W quorum Q_W such that *all* base objects from $Q_1 \cap Q_W$ had received a round W wr message containing ts and v (either from the writer or some reader writing-back the pair c_{sel}) and responded to the reader. Since read rd is uncontended, $BCD(c_{sel}, 1, W)$ is guaranteed to hold in case rd accesses a class 1 quorum of correct processes in the first round.

2. Else, if the reader received acks from some class 2 quorum(s) Q_2 (containing only correct processes) in round 1, then set $\boldsymbol{X} = \boldsymbol{BCD}(c_{sel}, 2, W)$ (line 2) is non-empty (since $Q_2 \in \boldsymbol{X}$). Indeed, notice that $\boldsymbol{X} = \boldsymbol{BCD}(c_{sel}, 2, W)$ contains a set of all class 2 quorums Q_2 such that: (a) the reader received replies from Q_2 in round 1 (i.e., $Q_2 \in \boldsymbol{QC'_2}$), and (b) there is a class W ($W \in 1 \ldots 3$) quorum Q_W such that *all* base objects from $Q_2 \cap Q_W$ received the round W wr message containing ts and v. Since the read rd is uncontended, $\boldsymbol{X}$ is guaranteed to contain all class 2 quorums that replied to the reader in round 1 (more precisely, all such class 2 quorums that contain only correct processes).

   Since $\boldsymbol{X}$ is non-empty, rd proceeds to round 2 (or, in other words, the first round of the writeback procedure, line 41). If $W = 2$ or $W = 3$, then the reader sends wr$\langle ts, v, \emptyset, 2 \rangle$ to all base objects, waits for acks from some quorum and returns (lines 42 and 60-61). Else, if $W = 1$, the first round of the writeback procedure (round 2 of rd) is more sophisticated. Namely, in this case, the reader: (a) triggers a timer (line 43), (b) sends wr$\langle ts, v, \boldsymbol{X}, 1 \rangle$ to all base objects (line 44 and 60), and (c) waits for acks until some quorum responds and the timer expires (lines 44-45 and 61). The read completes at the end of round 2 (the first writeback round) only if the reader receives acks from some quorum from $\boldsymbol{X} = \boldsymbol{BCD}(c_{sel}, 2, 1)$ (line 46).

3. Otherwise, if no quorum from $\boldsymbol{X}$ replies, the second round of the writeback procedure (i.e., the third round of rd) is invoked (line 47). Note that the

second part of the read algorithm (i.e., in lines 40-49) takes at most 2 rounds. Hence, when the read is synchronous and uncontended, it completes in at most 3 rounds.

The correctness proof of our atomic storage algorithm is given in Section 3.5. The critical part of the proof consists of several short lemmas. The main theorems (that make use of the above mentioned lemmas), despite being long, are easy to follow, since these are mainly case-by-case analysis, where the intersection properties of RQS allow critical lemmas to be easily applied.

3.3.2 Optimality

Let $\boldsymbol{Q}$ be any set of subsets of the set of base objects S. We say that an algorithm A is $(\boldsymbol{Q}, \boldsymbol{B})$*–atomic*, if A wait-free implements an atomic SWMR storage despite the adversary $\boldsymbol{B}$ provided that in every execution of A, there is a set $Q \in \boldsymbol{Q}$ that contains only correct base objects. Moreover, denote by $P_3(\boldsymbol{Q^{(1)}},\boldsymbol{Q^{(2)}},\boldsymbol{Q^{(3)}})$ (resp., $P_1(\boldsymbol{Q^{(3)}})$; $P_2(\boldsymbol{Q^{(1)}},\boldsymbol{Q^{(3)}})$) the property obtained from Property 3 (resp., Property 1; Property 2) of Definition 3 by replacing $\boldsymbol{QC_i}$ with $\boldsymbol{Q^{(i)}}$, for $i = 1 \ldots 3$. The minimality of our RQS is captured via the following three theorems.

Theorem 1. *If an algorithm A is $(\boldsymbol{Q^{(3)}}, \boldsymbol{B})$–*atomic*, then $P_1(\boldsymbol{Q^{(3)}})$ holds.*

Theorem 2. *If a $(\boldsymbol{Q^{(3)}}, \boldsymbol{B})$–*atomic *algorithm A is $(1, \boldsymbol{Q^{(1)}})$–*fast*, then $P_2(\boldsymbol{Q^{(1)}},\boldsymbol{Q^{(3)}})$ holds.*

Theorem 3. *If a $(\boldsymbol{Q^{(3)}}, \boldsymbol{B})$–*atomic *algorithm A is both $(1, \boldsymbol{Q^{(1)}})$–*fast *(for some $\boldsymbol{Q^{(1)}} \neq \emptyset$) and $(2, \boldsymbol{Q^{(2)}})$–*fast*, then $P_3(\boldsymbol{Q^{(1)}},\boldsymbol{Q^{(2)}},\boldsymbol{Q^{(3)}})$ holds.*

As a corollary of Theorems 1–3, our atomic storage implementation of Section 3.3.1 is optimally resilient and has the optimal (best-case) complexity.

Theorem 1 has been established for the special case of threshold-based quorums and with an implicit notion of quorums in [MAD02a]. It is not difficult to extend this bounds to the general adversary structure and the RQS setting.

Theorems 2 and 3 are novel; Theorem 3 is particularly interesting, due to the unusual *or* condition that appears in Property 3 of RQS. In the following we prove Theorem 2 and Theorem 3.

Proof. **(Theorem 2)**. Theorem 2 states that there is no $(\boldsymbol{Q^{(3)}}, \boldsymbol{B})$*–atomic* storage algorithm that is also $(1, \boldsymbol{Q^{(1)}})$*–fast*, if Property 2 of RQS is violated. Assume by contradiction that such a storage algorithm A exists even if Property 2 of RQS is violated. Consider a simple SWMR storage algorithm with a single writer w and two distinct readers r_1 and r_2 ($w \neq r_1 \neq r_2 \neq w$). In the following, we denote by $\overline{X}$ the set $S \setminus X$, where X is any subset of S (recall that S denotes the set of all base objects). Negating $P_2(\boldsymbol{Q^{(1)}},\boldsymbol{Q^{(3)}})$ (Property 2 of RQS) yields:

$$\exists Q_1, Q_1' \in \boldsymbol{Q^{(1)}},\ \exists Q \in \boldsymbol{Q^{(3)}},\ \exists B_1', B_2' \in \boldsymbol{B}\colon\ (Q_1 \cap Q_1' \cap Q) \subseteq (B_1' \cup B_2').$$

Since $(Q_1 \cap Q_1' \cap Q) \subseteq (B_1' \cup B_2')$ and $\boldsymbol{B}$ is an adversary for S, we can fix, without loss of generality B_1 and B_2 such that $B_1 \subseteq B_1'$ and $B_2 \subseteq B_2'$, such that $Q_1 \cap Q_1' \cap Q = B_1 \cup B_2$.

To exhibit a contradiction, we construct several partial executions (sketched in Figure 3.7) of the algorithm A including one in which atomicity is violated. More specifically, in this particular partial execution, a read operation returns a value that was never written.

- Let ex_1 be the execution in which all base objects from Q_1 are correct, while all others (i.e., those from $\overline{Q_1}$) fail by crashing at the beginning of the execution. Furthermore, let wr_1 be the write operation invoked at time t_1 by the correct writer w in ex_1 to write value $v_1 \neq \perp$ in the storage. Moreover, assume that the system is synchronous in ex_1. Hence, wr_1 is synchronous and uncontended. Since A is $(1, \boldsymbol{Q^{(1)}})$–*fast* and $Q_1 \in \boldsymbol{Q^{(1)}}$, wr_1 completes in ex_1, say at time t'_1, in a single communication round, after the writer receives the replies from base objects belonging to Q_1 in round 1.

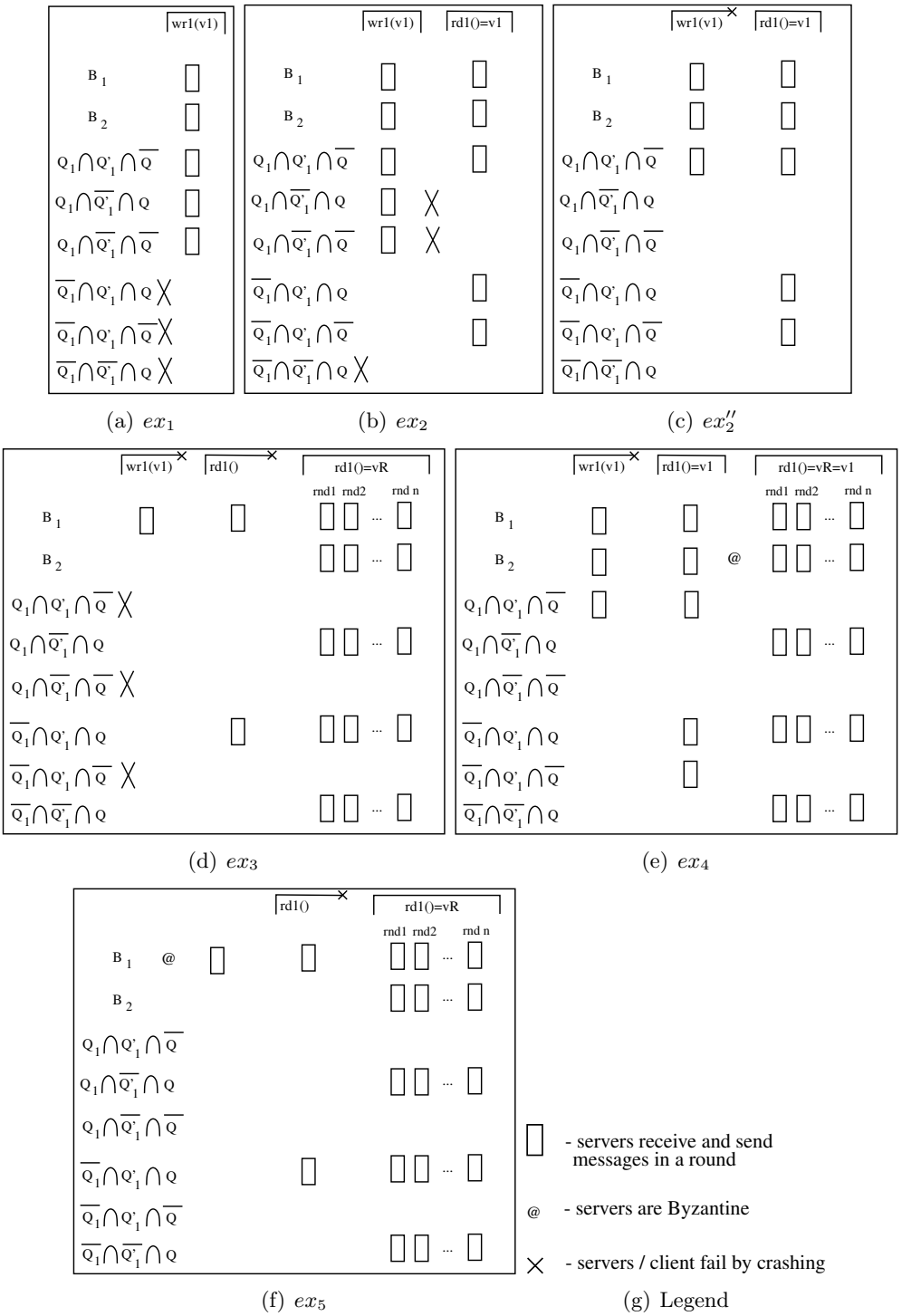
- Let ex'_1 be the partial execution that ends at t'_1, such that ex'_1 is identical to ex_1 up to time t'_1, except that, in ex'_1, base objects from $\overline{Q_1}$ do not crash, but, due to asynchrony, all messages sent by the writer to $\overline{Q_1}$ during wr_1 remain in transit. Since the writer cannot distinguish ex'_1 from ex_1, wr_1 completes in ex'_1, in a single communication rounds, at time t'_1.

- Let the partial execution ex_2 extend ex'_1 such that: (1) base objects from $\overline{Q'_1}$ crash at t'_1, (2) rd_1 is a synchronous read operation invoked by the correct reader r_1 after t'_1, and (3) no other operation is invoked in ex_2 (hence, rd_1 is uncontended). Since A is $(1, \boldsymbol{Q^{(1)}})$–*fast*, rd_1 completes in a single round (since a set Q'_1 of base objects is correct and $Q'_1 \in \boldsymbol{Q^{(1)}}$) at time t_2 and returns (by atomicity) v_1. Moreover, let ex_2 end at t_2. All messages that were in transit in ex'_1 remain in transit in ex_2.

- Let ex'_2 be the partial execution identical to ex_2 except that in ex'_2 base objects from $\overline{Q'_1}$ do not crash, but, due to asynchrony, the message sent from r_1 to base objects in $\overline{Q'_1}$ during rd_1 remains in transit in ex'_2. Since r_1 and all base objects, except those from $\overline{Q_1}$, cannot distinguish ex'_2 from ex_2, rd_1 completes in ex'_2 in a single round, at time t_2, and returns v_1.

- Let ex''_2 be the partial execution identical to ex'_2 except that, in ex''_2: (1) the writer crashes during wr_1 and its round 1 messages are not received by any base object from $\overline{Q'_1}$. In other words, in ex''_2, no object that belongs to $\overline{Q'_1} \cup \overline{Q_1}$ receives the round 1 message from the writer; only base objects from $Q'_1 \cap Q_1 = B_1 \cup B_2 \cup (Q'_1 \cap Q_1 \cap \overline{Q})$ receive the round 1 message from the writer. Since r_1 and all base objects, except those from $\overline{Q'_1} \cap Q_1$, cannot distinguish ex''_2 from ex'_2, rd_1 completes in ex''_2 at time t_2 and returns v_1.

- Consider now a partial execution ex_3 slightly different from ex''_2 in which the writer (resp., reader r_1) crashes during the round 1 of wr_1 (resp., rd_1) such that the round 1 messages sent by the writer (resp., r_1) in wr_1 (resp., rd_1) are received only by the base objects from set B_1 (resp., $B_1 \cup (\overline{Q_1} \cap Q'_1 \cap Q)$). We refer to the state of the base objects that belong to set B_1 (if any) *after* sending the reply to wr_1 as to σ_1. In ex_3, all base objects are correct except those from set $\overline{Q}$ that fail by crashing at the beginning of partial execution



(a) ex_1 (b) ex_2 (c) ex_2''

(d) ex_3 (e) ex_4

(f) ex_5 (g) Legend

Figure 3.7: Illustration of the partial executions used in the proof of Theorem 2. Only base objects that belong to set $Q_1 \cup Q_1' \cup Q$ are depicted.

ex_3. Assume that the writer crashes at time t_{fail_w} and that r_1 crashes at time $t_{fail_r} > t_{fail_w}$. Let rd_2 be a read operation invoked by the correct reader $r_2 \neq r_1$ at time $t'_3 > max(t_{fail_r}, t_2)$. Since all base objects from set Q are correct in ex_3 and A is a $(\boldsymbol{Q^{(3)}, B})$–*atomic* storage algorithm, rd_2 eventually completes, at some point in time t_3, after n communication rounds and returns value v_R.

- Let ex_4 be a partial execution identical to ex''_2 except that in ex_4: (1) a read operation rd_2 is invoked by the correct reader r_2 at t'_3 (as in ex_3), (2) due to asynchrony all messages sent by the base objects from $\overline{Q}$ to r_2 are delayed until after t_3 (i.e., until after n^{th} round of rd_2) and (3) in ex_4, all base objects from B_2 (if any) are Byzantine: these base objects forge their state at time t_2 to σ_0 (the initial state of base objects); otherwise, base objects from B_2 respect the algorithm (including with respect to the writer and reader r_1). Note that r_2 and the base objects from $Q \setminus B_2$ cannot distinguish ex_4 from ex_3 and, hence, rd_2 completes in ex_4 at time t_3 (as in ex_3) and returns v_R. On the other hand, r_1 cannot distinguish ex_4 from ex''_2. Hence, rd_1 completes in a single round and returns v_1. By atomicity, since rd_1 precedes rd_2, v_R must equal v_1.

- Consider now the partial execution ex_5, identical to ex_3, except that in ex_5: (1) wr_1 is never invoked, (2) base objects from B_1 (if any) are Byzantine in ex_5 and forge their state to σ_1 (see ex_3) at the beginning of the execution; otherwise, base objects from B_1 send the same messages as in ex_3, and (3) base objects from $\overline{Q}$ do not crash in ex_5, but, due to asynchrony, all messages sent from base objects from $\overline{Q}$ to r_2 are delayed until after t_3 (i.e., n^{th} round of rd_2). Reader r_2 and the base objects from $Q \setminus B_1$ cannot distinguish ex_5 from ex_3, so rd_2 completes at time t_3 and returns v_R, i.e., v_1 (see ex_4). However, by atomicity, in ex_5, rd_2 must return $\perp$, the initial value of the atomic storage. Since $v_1 \neq \perp$, ex_5 violates atomicity.

□

Now we prove Theorem 3. The techniques used in this proof resemble those used in the proof of Theorem 2.

Proof. **(Theorem 3)**. Theorem 3 states that there is no $(\boldsymbol{Q^{(3)}, B})$–*atomic* storage algorithm that is both $(1, \boldsymbol{Q^{(1)}})$–*fast* (for some $\boldsymbol{Q^{(1)}} \neq \emptyset$) and $(2, \boldsymbol{Q^{(2)}})$–*fast*, if Property 3 of RQS is violated. Assume by contradiction that such a storage algorithm A exists even if Property 3 of RQS is violated. As in the proof of Theorem 2, consider a simple SWMR storage algorithm with a single writer w and two distinct readers r_1 and r_2 ($w \neq r_1 \neq r_2 \neq w$). Again, we denote by $\overline{X}$ the set $S \setminus X$, where X is any subset of the set of base objects S. Negating $P_3(\boldsymbol{Q^{(1)}},\boldsymbol{Q^{(2)}},\boldsymbol{Q^{(3)}})$ (Property 3 of RQS) yields (having in mind $\boldsymbol{Q^{(1)}} \neq \emptyset$):

$\exists Q_2 \in \boldsymbol{Q^{(2)}}, \exists Q \in \boldsymbol{Q^{(3)}}, \exists Q_1 \in \boldsymbol{Q^{(1)}}, \exists B'_1, B'_2 \in \boldsymbol{B}$: $((Q_2 \cap Q) \subseteq (B'_1 \cup B'_2) \wedge (Q_2 \cap Q \cap Q_1) \in \boldsymbol{B}$.

In the following, we denote the set $Q_2 \cap Q \cap Q_1$ by B_0 (note that $B_0 \in \boldsymbol{B}$). Since $B_0 \subseteq Q_2 \cap Q$ and $\boldsymbol{B}$ is an adversary for S, we may fix, without loss of generality, B_1 and B_2, such that:

- $B_1, B_2 \in \boldsymbol{B}$,
- $B_1 \subseteq B'_1$,
- $B_2 \subseteq B'_2$,
- $Q_2 \cap Q = B_1 \cup B_2$, and
- $B_0 \subseteq B_1$.

Hence, $Q_2 \cap Q \cap \overline{Q_1} = B_2 \cup (B_1 \setminus B_0)$.

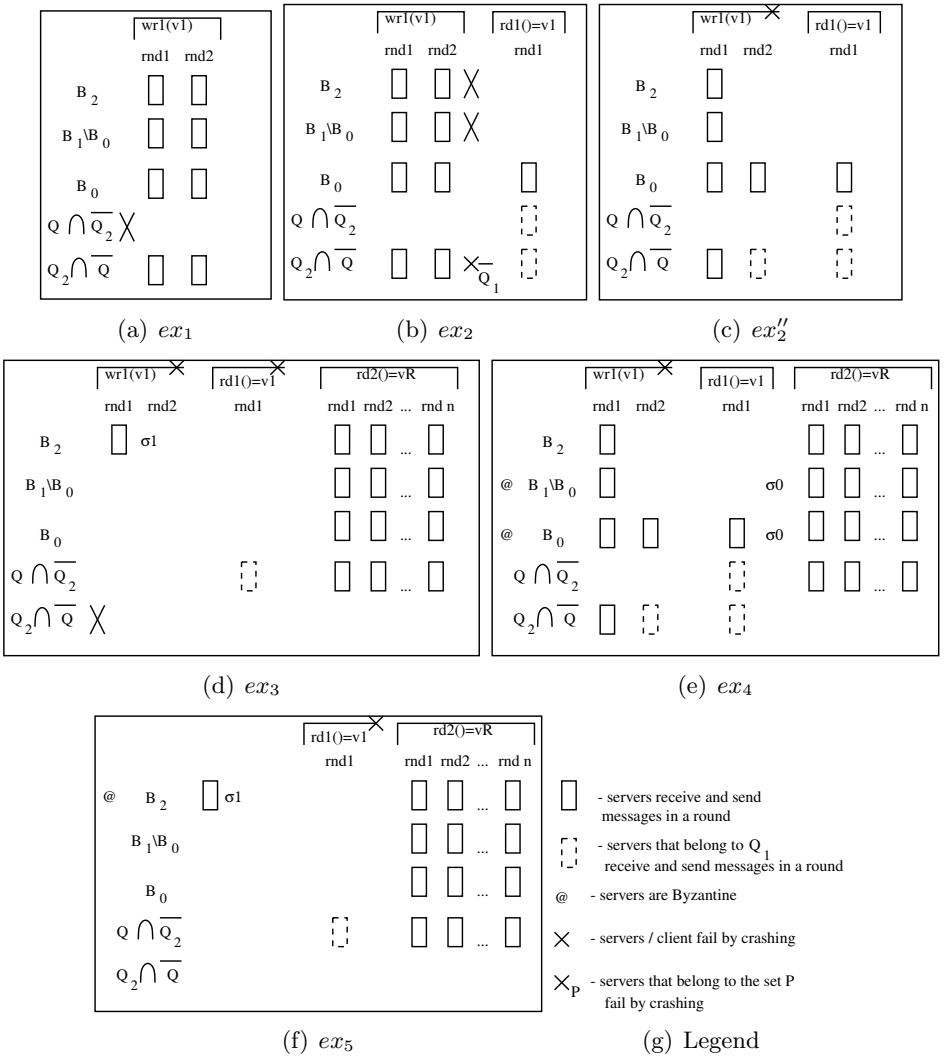
To exhibit a contradiction, we construct several partial executions (sketched in Figure 3.8) of the algorithm A including one in which atomicity is violated. More specifically, in this particular partial execution, a read operation returns a value that was never written.

- Let ex_1 be the execution in which all base objects from Q_2 are correct, while all others (i.e., those from $\overline{Q_2}$) fail by crashing at the beginning of the execution. Furthermore, let wr_1 be the write operation invoked at time t_1 by the correct writer w in ex_1 to write value $v_1 \neq \perp$ in the storage. Moreover, assume that the system is synchronous in ex_1. Hence, wr_1 is synchronous and uncontended. Since A is $(2, \boldsymbol{Q^{(2)}})$–*fast*, wr_1 completes in ex_1, say at time t'_1, in at most two communication rounds, after the writer receives the replies in round 2 from base objects from Q_2.

- Let ex'_1 be the partial execution that ends at t'_1, such that ex'_1 is identical to ex_1 up to time t'_1, except that, in ex'_1, base objects from $\overline{Q_2}$ do not crash, but, due to asynchrony, all messages sent by the writer to $\overline{Q_2}$ during wr_1 remain in transit. Since the writer cannot distinguish ex'_1 from ex_1, wr_1 completes in ex'_1, in two communication rounds, at time t'_1.

- Let the partial execution ex_2 extend ex'_1 such that: (1) base objects from $\overline{Q_1}$ crash at t'_1, (2) rd_1 is a synchronous read operation invoked by the correct reader r_1 after t'_1, and (3) no other operation is invoked (hence, rd_1 is uncontended). Since A is $(1, \boldsymbol{Q^{(1)}})$–*fast*, rd_1 completes in a single round (since set Q_1 contains correct base objects) at time t_2 and returns v_1. Moreover, let ex_2 end at t_2. All messages that were in transit in ex'_1 remain in transit in ex_2.

- Let ex'_2 be the partial execution identical to ex_2 except that in ex'_2 base objects from $\overline{Q_1}$ do not crash, but, due to asynchrony, the message sent from r_1 to base objects in $\overline{Q_1}$ during rd_1 remains in transit in ex'_2. Since r_1 and all base objects, except those from $\overline{Q_1}$, cannot distinguish ex'_2 from ex_2, rd_1 completes in ex'_2 in a single round, at time t_2, and returns v_1.



(a) ex_1 (b) ex_2 (c) ex_2''

(d) ex_3 (e) ex_4

(f) ex_5 (g) Legend

Figure 3.8: Illustration of the partial executions used in the proof of Theorem 3. Only base objects that belong to set $Q_2 \cup Q$ are depicted.

- Let ex_2'' be the partial execution identical to ex_2' except that, in ex_2'': (1) the writer crashes during wr_1 and its round 2 messages are not received by any base objects from $\overline{Q_1} \cup \overline{Q_2}$ (i.e., only base objects from $Q_1 \cap Q_2$ receive the round 2 message from the writer). Note that all base objects from set $B_2 \cup (B_1 \setminus B_0)$ belong to $\overline{Q_1}$ and, hence, do not receive a round 2 message from the writer. Since r_1 and all base objects, except those from $\overline{Q_1} \cap Q_2$, cannot distinguish ex_2'' from ex_2', rd_1 completes in ex_2'' at time t_2 and returns v_1.

- Consider now a partial execution ex_3 slightly different from ex_2'' in which the writer (resp., reader r_1) crashes during the round 1 of wr_1 (resp., rd_1) such that the round 1 messages sent by the writer (resp., r_1) in wr_1 (resp., rd_1) are received only by the base objects from set B_2 (resp., $Q \cap \overline{Q_2} \cap Q_1$). We refer to the state of the base objects that belong to set B_2 *after* sending the reply to the round 1 message of wr_1 as to σ_1. In ex_3, all base objects are correct except those from set $\overline{Q}$ that fail by crashing at the beginning of partial execution ex_3. Assume that the writer crashes at time t_{fail_w} and that r_1 crashes at time $t_{fail_r} > t_{fail_w}$. Let rd_2 be a read operation invoked by the correct reader $r_2 \neq r_1$ at time $t_3' > max(t_{fail_r}, t_2)$. Since all base objects from set Q are correct in ex_3 and A is a $(\boldsymbol{Q^{(3)}, B})$–*atomic* storage algorithm, rd_2 eventually completes, at some point in time t_3, after n communication rounds and returns value v_R.

- Let ex_4 be a partial execution identical to ex_2'' except that in ex_4: (1) a read operation rd_2 is invoked by the correct reader r_2 at t_3' (as in ex_3), (2) due to asynchrony all messages sent by the base objects from $\overline{Q}$ to r_2 are delayed until after t_3 (i.e., until after n^{th} round of rd_2) and (3) in ex_4, all base objects from B_1 (and B_0, since $B_0 \subseteq B_1$) are Byzantine: these base objects forge their state at time t_2 to σ_0 (the initial state of base objects); otherwise, base objects from B_1 respect the algorithm (including with respect to the writer and reader r_1). Note that r_2 and the base objects from $Q \setminus B_1 = B_2 \cup (Q \cap \overline{Q_2})$ cannot distinguish ex_4 from ex_3 and, hence, rd_2 completes in ex_4 at time t_3 (as in ex_3) and returns v_R. On the other hand, r_1 cannot distinguish ex_4 from ex_2''. Hence, rd_1 completes in a single round and returns v_1. By atomicity, since rd_1 precedes rd_2, v_R must equal v_1.

- Consider now the partial execution ex_5, identical to ex_3, except that in ex_5: (1) wr_1 is never invoked, (2) base objects from B_2 are Byzantine in ex_5 and forge their state to σ_1 (see ex_3); otherwise, base objects from B_2 send the same messages as in ex_3, and (3) base objects from $\overline{Q}$ do not crash in ex_5, but, due to asynchrony, all messages sent from base objects from $\overline{Q}$ to r_2 are delayed until after t_3 (i.e., until after the n^{th} round of rd_2). Reader r_2 and the base objects from $Q \setminus B_2 = B_1 \cup (Q \cap \overline{Q_2})$ cannot distinguish ex_5 from ex_3, so rd_2 completes at time t_3 and returns v_R, i.e., v_1 (see ex_4). However, by atomicity, in ex_5, rd_2 must return $\perp$, the initial value of the atomic storage. Since $v_1 \neq \perp$, ex_5 violates atomicity.

□

3.4 Consensus

In our storage algorithm, we assumed that (1) processes that access a RQS (the clients) might crash but cannot be Byzantine, and (2) processes that form a RQS (base objects) do not communicate directly with each other. In this section, we illustrate the case where (1) processes accessing a RQS might be Byzantine and (2) processes that form a RQS may directly communicate. By considering such a different model we: (a) argue for general applicability of RQS and (b) give the intuition about the best-case latency that can be achieved by applying RQS to the server-centric storage model.

The example we consider here is that of implementing a *consensus* abstraction. The consensus algorithm we present tolerates Byzantine failures of processes and unbounded periods of asynchrony. In fact, it is the first consensus algorithm that tolerates an unbounded number of Byzantine proposers and learners. The algorithm is optimal in terms of resilience as well as complexity, matching the lower bounds of [Lam03] and closing, we believe, a very important gap. The notion of complexity considered here is again best-case complexity for this is considered practically appealing on one hand and, on the other hand, the worst-case time complexity of a consensus algorithm that tolerates arbitrarily long periods of asynchrony is anyway unbounded. Our algorithm expedites the consensus decision under best-case conditions (synchrony and no contention) without using authenticated (digitally signed) messages; however, when best-case conditions are not met, authenticated messages are indeed used.

We assume the authenticated model specified in Chapter 2 complemented as follows.

To illustrate application of RQS on different models, channels are not assumed to be reliable (i.e., messages can be lost). Moreover, we construct a refined quorum system $\boldsymbol{RQS}$ around the set *acceptors* for an adversary $\boldsymbol{B}$, such that $\boldsymbol{RQS}$ is known to all processes. Recall that, in addition to these Byzantine acceptors, any number of proposers and learners can be Byzantine. Consensus *safety* (i.e., Validity and Agreement) is guaranteed as long as the set of Byzantine acceptors in any execution belongs to $\boldsymbol{B}$, while consensus *liveness* (i.e., Termination) is ensured if there is a correct quorum of acceptors $Q_c \in \boldsymbol{RQS}$.

In the remainder of this section, we describe our consensus algorithm and then state its optimality. The correctness proof of our consensus algorithm is given in Section 3.6.

3.4.1 Consensus Algorithm

Our consensus algorithm is designed to have optimal time complexity in best case executions. More precisely, we say that an execution ex of a consensus algorithm is a *best-case* execution if, in ex: (1) there is no contention, i.e., (a) all proposers are benign and (b) exactly one proposer p proposes, say some value v at time t (and p is correct) and (2) the system is synchronous (during $[t, t + 4\Delta]$ — see Chapter 2 for the definition of synchronous system and Δ). Let $\boldsymbol{P}$ be any set of subsets of $acceptors$.

Definition 4. $(m, \boldsymbol{P})$–**fast *consensus algorithm***. *We say that a consensus algorithm is* $(m, \boldsymbol{P})$–fast *if its time complexity over all best-case execution ex in which some set* $P \in \boldsymbol{P}$ *contains only correct acceptors, is* $m + 1$ *message delays, and no authenticated messages are used in such executions.*

Our consensus algorithm is $(m, \boldsymbol{QC_m})$–*fast* for all $m \in \{1, 2, 3\}$.

The algorithm consists of two modules: (1) a *Locking* module that ensures safety, and (2) an *Election* module used to help ensure liveness. The algorithm proceeds in a sequence of *views*. In every view, a single proposer is the *leader*, except in the initial view ($initView$) where all proposers can be seen as leaders. The *Locking* module consists of a *consult* (Fig. 3.11) and an *update* phase (Fig. 3.10). In $initView$, the proposer, on proposing a value, skips the consult phase and executes immediately the update phase (see Fig. 3.9). In the following, we first explain the update phase.

```
Initialization:
view, initView := 0

propose(v) is {
  if (view ≠ initView) then
    consult phase
  endif
  update phase }

upon p_j is elected
  propose(v)
```

Figure 3.9: The *Locking* module: High level pseudocode of a proposer p_j

Update phase.

This phase proceeds in the following communication steps (we refer to Fig. 3.10):

1. On proposing value v, proposer p sends a prepare message to all acceptors containing (line 9): (a) its proposal value v, (b) view number $view$ (initially $view = initVew = 0$), and (c) array of authenticated messages $vProof$ that originates from a quorum Q of acceptors (the ID of quorum Q is also included in the prepare message). Roughly, $vProof$ serves as a certificate for the proposed value v. We detail how $vProof$ is constructed later when explaining the consult phase. It is important to notice that, in the $initView$, $vProof$ equals nil (i.e., contains no messages).

2. A benign acceptor a_j, on reception of a message $m = \mathsf{prepare}\langle v, view, vProof, Q\rangle$ from p, checks if (line 31): (a) a_j is in $view$, (b) a_j did not already receive a prepare message in $view$ (or in a higher view), and (c) (unless $view = initView$) whether p is the leader of $view$ and whether $vProof$ matches value v (this is done using the $choose()$ function that is explained later in details). If these checks succeed, a_j stores v into a local variable $Prep$ and the view number in the set variable $Prep_{view}$ (we simply

at every proposer p_j:
Initialization:
$view, initView := 0$; $vProof := nil$; $Q := \emptyset$
9: send $\mathsf{prepare}\langle v, view, vProof, Q\rangle$ to $acceptors$

at every acceptor a_j:
Initialization:
$view_{a_j} := initView$;
$Prep_{view}, old, Update_{proof}[*,*], Update_{view}[*], Update_{QRM}[*,*] := \emptyset$;
$Prep, Update[*] := nil$

upon received $m = \mathsf{prepare}\langle v, view_{a_j}, vProof, Q\rangle$ from p_i
31: **if** $(\forall w \in Prep_{view} : w < view_{a_j})$ **and**
$((view_{a_j} = initView)$ **or** $((p_i$ is the leader of $view_{a_j})$ **and** (v matches **choose**$(v, vProof, Q))))$ **then**
32: **if** $Prep = v$ **then** $Prep_{view} := Prep_{view} \cup \{view_{a_j}\}$ **else** $Prep := v$; $Prep_{view} := \{view_{a_j}\}$
33: send $m_1 = \mathsf{update}_1\langle v, view_{a_j}, \emptyset\rangle$ to $acceptors \cup learners$; $old := old \cup m_1$

upon received $m = \mathsf{update}_{step}\langle v, view_{a_j}, *\rangle$ from some quorum Q (for $step \in \{1, 2\}$) **and**
$v = Prep$ **and**
$view_{a_j} \in Prep_{view}$
34: **if** $Update[step] = v$ **then** $Update_{view}[step] := Update_{view}[step] \cup \{view_{a_j}\}$
35: **else** $Update[step] := v$; $Update_{view}[step] := \{view_{a_j}\}$;
36: $Update_{QRM}[step, *] := \emptyset$; $Update_{proof}[step, *] := \emptyset$
37: **if** $(Q \notin Update_{QRM}[step, view_{a_j}]$ **and** $step = 1)$ **or**
$(Update_{QRM}[step, view_{a_j}] = \emptyset$ **and** $step = 2)$ **then**
38: $Update_{QRM}[step, view_{a_j}] := Update_{QRM}[step, view_{a_j}] \cup Q$
39: send $m_{step+1} = \mathsf{update}_{step+1}\langle v, view_{a_j}, Q\rangle$ to $acceptors \cup learners$; $old := old \cup m_{step+1}$

at every acceptor and learner x:
upon received the same $\mathsf{update}_1\langle v, view, *\rangle$ from $Q_1 \in \boldsymbol{QC_1}$
51: **if** x has not yet decided **then decide**(v)

upon received the same $\mathsf{update}_2\langle v, view, Q_2\rangle$ from $Q_2 \in \boldsymbol{QC_2}$
52: **if** x has not yet decided **then decide**(v)

upon received the same $\mathsf{update}_3\langle v, view, *\rangle$ from $Q_3 \in \boldsymbol{RQS}$
53: **if** x has not yet decided **then decide**(v)

at every learner l_j:
upon l_j decides v
60: **learn**(v)

Figure 3.10: The *Locking* module: *update* phase

say, a_j *prepares* v in *view*). If $Prep$ was already equal to v, then a_j simply adds *view* to set $Prep_{view}$ (line 32). Then a_j echoes v by sending an $\mathsf{update}_1\langle v, view, \emptyset\rangle$ message to all acceptors and learners (line 33).

3. A benign acceptor a_j, upon receiving update_1 messages from some quorum Q with the same value v and view number *view*, checks if its local view equals *view* and if it already prepared a message with value v in *view*. If this check succeeds, a_j performs the following local computations (we say a_j *1-updates* v in *view* with the quorum Q). In case the local variable $Update[1]$ does not equal v (i.e., if a new value is 1-updated — line 35), a_j stores v into $Update[1]$, *view* into $Update_{view}[1]$ and empties sets $Update_{QRM}[1, *]$ and $Update_{proof}[1, *]$. In case value v was already 1-updated (in some previous view — line 34), a_j simply adds *view* into $Update_{view}[1]$. Then, a_j adds the identifier of the quorum Q into set $Update_{QRM}[1, view]$ and sends an $\mathsf{update}_2\langle v, view, Q\rangle$ message to all acceptors and learners (here, an update_2 message is sent once per every different quorum Q — line 38).

4. A benign acceptor a_j, upon receiving update_2 messages from some quorum, performs the similar steps as when receiving a quorum of update_1 messages (we say a_j *2-updates* v in *view*), including sending an update_3 message containing v and *view* to all acceptors and learners (lines 34-39). The difference with respect to the step (3) is that an acceptor sends only one $\mathsf{update}_3\langle v, view, *\rangle$ message to other acceptors and learners.

Moreover, all acceptors and learners decide on v upon receiving update_1 messages with the same value v and view number *view* from a class 1 quorum (line 51). Similarly, acceptors and learners decide on v upon receiving the same $\mathsf{update}_2\langle v, view, Q_2\rangle$ messages from some class 2 quorum Q_2 (note here that, besides the value and the view number, the quorum identifier within update_2 messages must be the same — line 52). Finally, acceptors and learners decide on v upon receiving update_3 messages with the v and *view* from any (class 3) quorum of acceptors (line 53). Besides, a benign learner l_j learns v as soon as l_j decides v (line 60).

The above scheme guarantees that, in the best case execution, in which only a single proposer proposes in the $initView$ and the system is synchronous, all correct learners learn v in two (resp., three; four) message delays in case a class 1 (resp., class 2; class 3) quorum of correct acceptors is available.[2] Note that, in the above sequence, all messages are unauthenticated.

Consult phase.

In the best-case execution, the *Election* module, responsible for view changes, does not change the view before all correct acceptors (and learners) decide v. However, if more than one proposer proposes in $initView$, or some proposer is Byzantine, or if the system is asynchronous, the *Election* module might designate

[2]Since an availability of a class 3 quorum of acceptors is anyway assumed, our algorithm guarantees that a value will be learned by all correct learners in at most four message delays in any best-case execution.

at every proposer p_j**:**
Initialization:
$view, initView := 0$; $viewProof, vProof := nil$; $\boldsymbol{faulty} := \emptyset$
2: send new_view$\langle view, viewProof\rangle$ to $acceptors$
3: **repeat**
4: **wait for** valid acks from some quorum $Q \in \boldsymbol{RQS} \setminus \boldsymbol{faulty}$
5: $vProof :=$ array of received acks from Q
6: $(v, abort) :=$ **choose**$(v, vProof, Q)$
7: **if** $abort$ **then** $\boldsymbol{faulty} := \boldsymbol{faulty} \cup \{Q\}$
8: **until** $\neg(abort)$

at every acceptor a_j**:**
Initialization:
$view_{a_j} := initView$;
$Prep_{view}, old, Update_{proof}[*,*], Update_{view}[*], Update_{QRM}[*,*] := \emptyset$;
$Prep, Update[*] := nil$

upon received new_view$\langle view, viewProof\rangle$ from p_i
21: **if** $(view > view_{a_j})$ **and** (p_i is the leader of $view$) **and** ($viewProof$ matches $view$) **then**
22: $view_{a_j} := view$
23: $\forall step \in \{1,2\}, \forall w : w \in Update_{view}[step] \wedge Update_{proof}[step, w] = \emptyset$ **do**
24: send sign_req$\langle Update[step], w, step\rangle$ to some quorum in $Update_{QRM}[step, w]$
25: **for** every sent sign_req$\langle Update[step], w, step]\rangle$ message
26: **wait for** acks with a valid signature from some subset of $acceptors$ $T_{step,w}$, $T_{step,w} \notin \boldsymbol{B}$
27: $Update_{proof}[step, w] :=$ received acks from $T_{step,w}$
28: send new_view_ack$\langle view_{a_j}, Prep, Prep_{view}, Update[1..2], Update_{view}[1..2],$
$Update_{proof}[1..2,*], Update_{QRM}[1..2,*]\rangle_{\sigma_{a_j}}$ to p_i

upon received sign_req$\langle v, w, step\rangle$ from a_i
29: **if** $m =$ update$_{step}\langle v, w, *\rangle \in old$ **then** send sign_ack$\langle m\rangle_{\sigma_{a_j}}$ to a_i

Figure 3.11: The *Locking* module: *consult* phase

a different proposer p_w to be the leader for the new view w (see Fig. 3.9). Upon being elected, proposer p_w starts the consult phase of w by sending the new_view message to acceptors (line 2, Fig. 3.11). The new_view message contains a view number and a set of messages, $viewProof$, which are provided to the proposer by the *Election* module. Set $viewProof$ consists of signed (authenticated) messages from a quorum of acceptors — this vouches for the authenticity of the new_view message. After sending the new_view message, p_w waits for a quorum Q of *valid* signed acks (line 4) containing the last prepared, 1-updated and 2-updated values, along with the corresponding view numbers. An acceptor acks a new_view message only if (line 21, Fig. 3.11): (a) view number w is higher than the acceptor's local view number, (b) p_w is the leader of view w (i.e., if $p_w = p_{w \bmod |proposers|}$), and (c) set $viewProof$ matches w, i.e., if $viewProof$ proof contains a quorum of authenticated messages vouching that p_w may issue a new_view message for view w.

An ack (i.e., a new_view_ack message — line 28, Fig. 3.11) for view w is considered *valid* in line 4, Fig. 3.11 if every value v_{step} in $Update[step]$ and every view number w' in $Update_{view}[step]$ ($step \in \{1, 2\}$) is accompanied by a set of signatures, $Update_{proof}[step, w']$. Here, every $Update_{proof}[step, w']$ must be a set of signed $\mathsf{update}_{step}\langle v_{step}, w', * \rangle$ messages sent from all acceptors from some subset of acceptors that is not an element of an adversary (to guarantee that a message is signed by at least one benign acceptor). An acceptor a_j must obtain all the necessary sets of signatures before replying to the new_view message — this is done in lines 23-27, Fig. 3.11, unless a_j already possesses the required proofs.

Then, the leader p_w evaluates acks from Q using the *choose*() function (line 6, Fig. 3.11 and Fig. 3.12). This function ensures the following crucial property: *if any value v is decided in some view w, then a benign acceptor accept only v in any view higher than w*. We sketch the arguments (based on RQS properties) behind this property, for view $w + 1$ (which gives the base step of the induction-based proof). In the following, we refer to Figure 3.12.

Let v be the value decided by some benign process (acceptor or learner) in view w upon receiving update_1 (resp., update_2; update_3) messages from some $Q_1 \in \boldsymbol{QC_1}$ (resp., $Q_2 \in \boldsymbol{QC_2}$; $Q_3 \in \boldsymbol{QC_3}$). Then, for any Q, substituting for Q_1 (resp., Q_2; Q_3), $Cand_2(v, w)$ holds in line 2 (resp., $Cand_3(v, w, char)$ holds for some $char \in \{`a`, `b`\}$ in line 3; $Cand_4(v, w)$ holds in line 4). In this case, we say that v is a *candidate* with view w. It is not difficult to see that there can be no candidate v with view $w' > w$ (since no benign acceptor prepares or updates any value in a view higher than w), i.e., $w = view_{max}$ in line 12. Hence, *choose*() may return only the candidate with view w. In case such candidate is not unique:

— If $Cand_3(v, w, `a`)$ holds (line 13), $P_{3a}(Q_2, Q)$ holds. From Property 3a of RQS and line 3, acceptors from some subset that is not an element of an adversary 1-updated v in w, including at least one benign acceptor. Similarly, if $Cand_4(v, w)$ holds, at least one benign acceptor 2-updated v in w (there is a subset of signatures from a subset of acceptors that is not an element of an adversary in $Update_{proof}[2]$ received from some $a_j \in Q$). Since benign acceptors 1-update or 2-update v in w only if a quorum of acceptors prepared v in w (more precisely, only if all benign acceptors from some quorum prepared v in w), all

Definitions:
2: $Cand_2(v, w) ::= \exists Q_1 \in \boldsymbol{QC_1}, \exists B \in \boldsymbol{B}, \forall a_j \in (Q_1 \cap Q) \setminus B :$
$(w \in vProof[a_j].Prep_{view}) \wedge (vProof[a_j].Prep = v)$
3: $Cand_3(v, w, char) ::= \exists Q_2 \in \boldsymbol{QC_2}: \exists B \in \boldsymbol{B}, \forall a_j \in (Q_2 \cap Q) \setminus B :$
$\bigwedge (char = `a` \Rightarrow P_{3a}(Q_2, Q) \wedge char = `b` \Rightarrow P_{3b}(Q_2, Q))$
$\bigwedge (vProof[a_j].Update[1] = v)$
$\bigwedge (w \in vProof[a_j].Update_{view}[1])$
$\bigwedge (Q_2 \in vProof[a_j].Update_{QRM}[1, w])$
4: $Cand_4(v, w) ::= \exists a_j \in Q : (vProof[a_j].Update[2] = v) \wedge (w \in vProof[a_j].Update_{view}[2])$

choose$(v', vProof, Q)$ **returns**$(v_{ret}, abort)$ **is** {
10: $v_{ret} := v'$; $abort := false$
11: **if** $\exists v : Cand_2(v, *) \vee Cand_3(v, *, *) \vee Cand_4(v, *)$ **then**
12: $view_{max} := max\{w | \exists v : (Cand_2(v, w)) \vee Cand_3(v, w, *) \vee Cand_4(v, w)\}$
13: **if** $\exists v : Cand_3(v, view_{max}, `a`) \vee Cand_4(v, view_{max})$ **then** $v_{ret} := v$; **return**
14: **if** $\exists v, v' : (v \neq v') \wedge Cand_2(v, view_{max}) \wedge Cand_3(v', view_{max}, `b`)$ **then**
15: $abort := true$; **return**
16: **if** $\exists v, v' : (v \neq v') \wedge Cand_3(v, view_{max}, `b`) \wedge Cand_3(v', view_{max}, `b`)$ **then**
17: $abort := true$; **return**
18: $v_{ret} := v : (Cand2(v, view_{max})) \vee (Cand_3(v, view_{max}, `b`))$; **return**
19: **return**

Figure 3.12: *choose()* function

benign acceptors from some quorum indeed prepared v in w — by Property 1 of RQS, *choose*() can return v.

— If Q contains only benign acceptors, then the condition in line 14 (resp., 16) cannot hold by Property 3b (resp., Property 1) of RQS. Namely, if these conditions hold, then at least one Byzantine acceptor is in Q — *choose*() aborts, waits for other ack(s) and reiterates (lines 3-8, Fig. 3.11).

The Election module. The Election module given in Figure 3.13 is very simple and guarantees progress in case the system is eventually synchronous. It is based on an exponential increase of the timeouts (maintained by acceptors) between views. This scheme can be seen as inefficient, and impact the worst-case performance of our algorithm. Different optimizations of this simple scheme are possible, but these are out of the scope of this thesis.

The pseudocode of the *Locking* module, combined from pseudocodes of Figures 3.9, 3.10 and 3.11 is given in Figure 3.14. The additional part of the *Locking* module consists of lines 40 and 101-103, Fig. 3.14, that serve solely to facilitate the property of the *Election* module that its view-change mechanism can be permanently stopped (line 9, Fig. 3.13) as well as to halt proposers (line 203, Fig. 3.14). If we allow view changes to occur permanently, lines 40 and 101-103 of Fig. 3.14 can be omitted.

The correctness proof of our consensus algorithm is given in Section 3.6. In the following we state the optimality of our algorithm.

3.4.2 Optimality

We say that an algorithm A implements ($\boldsymbol{Q}$,$\boldsymbol{B}$)*–consensus* if A ensures consensus Validity and Agreement, as long as, for any execution ex of A, the set of accep-

at every acceptor a_j:
$suspectTimeout, initTimeout := 5\Delta$; $nextView_{a_j} := initView$ % Initialization

upon reception of prepare$\langle *, initView, *, * \rangle$ or sync message for the first time
0: **trigger**($suspectTimeout$)

upon expiration of ($suspectTimeout$)
1: $suspectTimeout := suspectTimeout * 2$
2: **inc**($nextView_{a_j}$)
3: $nextLeader := nextView_{a_j}$ mod $|proposers|$
4: send view_change$\langle nextView_{a_j} \rangle_{\sigma_{a_j}}$ to $p_{nextLeader}$
5: **trigger**($suspectTimeout$)

upon decide(v)
8: send decision$\langle v \rangle$ to $acceptors$

upon reception of a valid decision$\langle v \rangle$ from some quorum $Q \in \boldsymbol{RQS}$
9: **stop**($suspectTimeout$)

at every proposer p_j:

upon reception of view_change$\langle nextView \rangle_{\sigma_{a_i}}$ with the same $nextView$ from all a_i from some $Q \in \boldsymbol{RQS}$
10: **if** $nextView > view$ **then**
11: $viewProof :=$ ∪ received signed view_change$\langle nextView \rangle$ messages
12: $view := nextView$
13: **elect**($self$)

upon p_j proposed a value for the first time
201: **wait** some preset time
202: send ⟨decision_pull⟩ to $acceptors$

upon received decision$\langle v \rangle$ (with the same v) from some quorum $Q \in \boldsymbol{RQS}$
203: **halt**

Figure 3.13: The *Election* module

at every proposer p_j:
Initialization:
$view, initView := 0$; $viewProof, vProof := nil$; Q, $\boldsymbol{faulty} := \emptyset$
propose(v) **is** {
1: **if** $(view \neq initView)$ **then** % *consult* phase
2: send new_view$\langle view, viewProof\rangle$ to $acceptors$
3: **repeat**
4: **wait for** valid acks from some quorum $Q \in \boldsymbol{RQS} \setminus \boldsymbol{faulty}$
5: $vProof :=$ array of received acks from Q
6: $(v, abort) :=$ **choose**$(v, vProof, Q)$
7: **if** $abort$ **then** $\boldsymbol{faulty} := \boldsymbol{faulty} \cup \{Q\}$
8: **until** $\neg(abort)$
9: send prepare$\langle v, view, vProof, Q\rangle$ to $acceptors$ % *update* phase

upon p_j is elected
10: **propose**(v)

---

at every acceptor a_j:
Initialization:
$view_{a_j} := initView$; $Prep, Update[*] := nil$;
$Prep_{view}, old, Update_{proof}[*,*], Update_{view}[*], Update_{QRM}[*,*] := \emptyset$;

upon received new_view$\langle view, viewProof\rangle$ from p_i % *consult* phase (lines 21-29)
21: **if** $(view > view_{a_j})$ **and** (p_i is the leader of $view$) **and** ($viewProof$ matches $view$) **then**
22: $view_{a_j} := view$
23: $\forall step \in \{1,2\}, \forall w : w \in Update_{view}[step] \wedge Update_{proof}[step, w] = \emptyset$ **do**
24: send sign_req$\langle Update[step], w, step\rangle$ to some quorum in $Update_{QRM}[step, w]$
25: **for** every sent sign_req$\langle Update[step], w, step]\rangle$ message
26: **wait for** acks with a valid signature from some subset of $acceptors$ $T_{step,w}$, $T_{step,w} \notin \boldsymbol{B}$
27: $Update_{proof}[step, w] :=$ received acks from $T_{step,w}$
28: send new_view_ack$\langle view_{a_j}, Prep, Prep_{view}, Update[1..2], Update_{view}[1..2],$
$Update_{proof}[1..2,*], Update_{QRM}[1..2,*]\rangle_{\sigma_{a_j}}$ to p_i
upon received sign_req$\langle v, w, step\rangle$ from a_i
29: **if** $m =$ update$_{step}\langle v, w, *\rangle \in old$ **then** send sign_ack$\langle m\rangle_{\sigma_{a_j}}$ to a_i

upon received $m =$ prepare$\langle v, view_{a_j}, vProof, Q\rangle$ from p_i % *update* phase (lines 31-39 and 51-60)
31: **if** $(\forall w \in Prep_{view} : w < view_{a_j})$ **and**
$((view_{a_j} = initView)$ **or** $((p_i$ is the leader of $view_{a_j})$ **and** (v matches **choose**$(v, vProof, Q))))$ **then**
32: **if** $Prep = v$ **then** $Prep_{view} := Prep_{view} \cup \{view_{a_j}\}$ **else** $Prep := v$; $Prep_{view} := \{view_{a_j}\}$
33: send $m_1 =$ update$_1\langle v, view_{a_j}, \emptyset\rangle$ to $acceptors \cup learners$; $old := old \cup m_1$

upon received $m =$ update$_{step}\langle v, view_{a_j}, *\rangle$ from some quorum Q (for $step \in \{1,2\}$) **and**
$v = Prep$ **and** $view_{a_j} \in Prep_{view}$
34: **if** $Update[step] = v$ **then** $Update_{view}[step] := Update_{view}[step] \cup \{view_{a_j}\}$
35: **else** $Update[step] := v$; $Update_{view}[step] := \{view_{a_j}\}$;
36: $Update_{QRM}[step, *] := \emptyset$; $Update_{proof}[step, *] := \emptyset$
37: **if** $(Q \notin Update_{QRM}[step, view_{a_j}]$ **and** $step = 1)$ **or**
$(Update_{QRM}[step, view_{a_j}] = \emptyset$ **and** $step = 2)$ **then**
38: $Update_{QRM}[step, view_{a_j}] := Update_{QRM}[step, view_{a_j}] \cup Q$
39: send $m_{step+1} =$ update$_{step+1}\langle v, view_{a_j}, Q\rangle$ to $acceptors \cup learners$; $old := old \cup m_{step+1}$

upon reception of $\langle$decision_pull$\rangle$ from a process q
40: **if** decided v **then** send decision$\langle v\rangle$ to $acceptors \cup \{q\}$

---

at every acceptor and learner x:
upon received the same update$_1\langle v, view, *\rangle$ from $Q_1 \in \boldsymbol{QC_1}$
51: **if** x has not yet decided **then decide**(v)

upon received the same update$_2\langle v, view, Q_2\rangle$ from $Q_2 \in \boldsymbol{QC_2}$
52: **if** x has not yet decided **then decide**(v)

upon received the same update$_3\langle v, view, *\rangle$ from $Q_3 \in \boldsymbol{RQS}$
53: **if** x has not yet decided **then decide**(v)

at every learner l_j:
upon l_j decides v
60: **learn**(v)

---

at every learner l_j:
upon l_j received decision$\langle v\rangle$ from some subset of $acceptors$ T, $T \notin \boldsymbol{B}$
101: **if** l_j has not yet learned a value **then learn**(v)

upon value not learned
102: **wait** some preset time
103: **if** value not learned **then** send $\langle$decision_pull$\rangle$ to $acceptors$

Figure 3.14: The *Locking* module

tors Byzantine in ex belongs to $\boldsymbol{B}$, as well as Termination in case the system is eventually synchronous and there is a set $Q \in \boldsymbol{Q}$ that contains only correct acceptors. In addition, analogously to Section 3.3.2, we define the properties $P_1(\boldsymbol{Q^{(3)}})$, $P_2(\boldsymbol{Q^{(1)}},\boldsymbol{Q^{(3)}})$ and $P_3(\boldsymbol{Q^{(1)}},\boldsymbol{Q^{(2)}},\boldsymbol{Q^{(3)}})$, where $\boldsymbol{Q^{(i)}}$ is some set of subsets of *acceptors*. The following theorems capture the minimality of our RQS, assuming $|proposers| \geq 2$ and $|learners| \geq 3$.

Theorem 4. *If an algorithm A implements* $(\boldsymbol{Q^{(3)}}, \boldsymbol{B})$–consensus*, then* $P_1(\boldsymbol{Q^{(3)}})$ *holds.*

Theorem 5. *If a* $(\boldsymbol{Q^{(3)}}, \boldsymbol{B})$–consensus *algorithm A is* $(1, \boldsymbol{Q^{(1)}})$–fast*, then* $P_2(\boldsymbol{Q^{(1)}},\boldsymbol{Q^{(3)}})$ *holds.*

Theorem 6. *If a* $(\boldsymbol{Q^{(3)}}, \boldsymbol{B})$–consensus *algorithm A is both* $(1, \boldsymbol{Q^{(1)}})$–fast *(for some* $\boldsymbol{Q^{(1)}} \neq \emptyset$*) and* $(2, \boldsymbol{Q^{(2)}})$–fast*, then* $P_3(\boldsymbol{Q^{(1)}},\boldsymbol{Q^{(2)}},\boldsymbol{Q^{(3)}})$ *holds.*

Theorem 4 is not new; it follows directly from [JM03]. Moreover, in the special threshold case, where (a) $\boldsymbol{B}$=$\boldsymbol{B_k}$, (b) all elements of $\boldsymbol{Q^{(1)}}$ (resp., $\boldsymbol{Q^{(3)}}$) contain all but at most q (resp., t) acceptors, and (c) $q = t-2k$, Theorems 4–5 correspond to the lower bounds identified in [Lam03].

In the following, we prove Theorem 6. To strengthen the optimality result established by Theorem 6, we assume that proposers and learners may not be Byzantine, yet that any number of proposers and learners may fail by crashing.

Proof. To prove Theorem 6, we assume full information protocols in the *round-by-round* eventually synchronous model [Gaf98, KS06]. The assumption of a full information protocol is indeed without loss of generality, since if, in some particular algorithm A, a process p does not send a message to process q in round rnd, we model this by having process p send to q a default (unauthenticated) message msg_{nil} in rnd and q does not change its state upon reception of a message msg_{nil}. Moreover, denote by $m^j_{rnd}[i]$ the message sent by process j to process i in round rnd of some execution. For presentation simplicity we assume that, in each round, every process combines all the messages $m^j_{rnd}[i]$ it is about to send in rnd and sends the same message m^j_{rnd} to all processes, such that every process i (including Byzantine ones) ignores all the portions of the message except $m^j_{rnd}[i]$ (it is not difficult to see that this is indeed without loss of generality).

Assume, by contradiction, that there is a $(\boldsymbol{Q^{(3)}}, \boldsymbol{B})$–*consensus* that is both $(1, \boldsymbol{Q^{(1)}})$–*fast* (for some $\boldsymbol{Q^{(1)}} \neq \emptyset$) and $(2, \boldsymbol{Q^{(2)}})$–*fast* such that $P_3(\boldsymbol{Q^{(1)}},\boldsymbol{Q^{(2)}},\boldsymbol{Q^{(3)}})$ is violated, i.e.:

$$\exists Q_2 \in \boldsymbol{Q^{(2)}},\ \exists Q \in \boldsymbol{Q^{(3)}},\ \exists Q_1 \in \boldsymbol{Q^{(1)}},\ \exists B_1, B_2 \in \boldsymbol{B}\colon\ ((Q_2 \cap Q) \subseteq (B_1 \cup B_2)) \wedge (Q_2 \cap Q \cap Q_1) \in \boldsymbol{B}.$$

In the following, we denote the set $Q_2 \cap Q \cap Q_1$ by B_0 (note that $B_0 \in \boldsymbol{B}$). Since $B_0 \subseteq Q_2 \cap Q$ and $\boldsymbol{B}$ is an adversary for *acceptors*, we may assume, without loss of generality, that $B_0 \subseteq B_1$. Moreover, since $\boldsymbol{B}$ is an adversary for *acceptors*, without loss of generality we may fix B_1 and B_2 such that $Q_2 \cap Q$ equals $B_1 \cup B_2$. Note that this implies $(B_1 \cup B_2) \subseteq Q_2$ and $(B_1 \cup B_2) \subseteq Q$. Furthermore, denote by

$\overline{X}$ the set $acceptors \setminus X$, where X is any subset of $acceptors$. Hence, $Q_2 \cap Q \cap \overline{Q_1} = B_2 \cup (B_1 \setminus B_0)$.

Denote by p_0 and p_1 two distinct proposers ($p_0 \neq p_1$) (such proposers exist since $|proposers| \geq 2$). Since there are at least three learners, there exists a learner $l_1 \in learners \setminus \{p_0, p_1\}$. Denote by l_2 some learner different from l_1. Without loss of generality, we assume that if $l_2 \in \{p_0, p_1\}$ then $p_0 = l_2$. Recall that $(proposers \cup learners) \cap acceptors = \emptyset$.

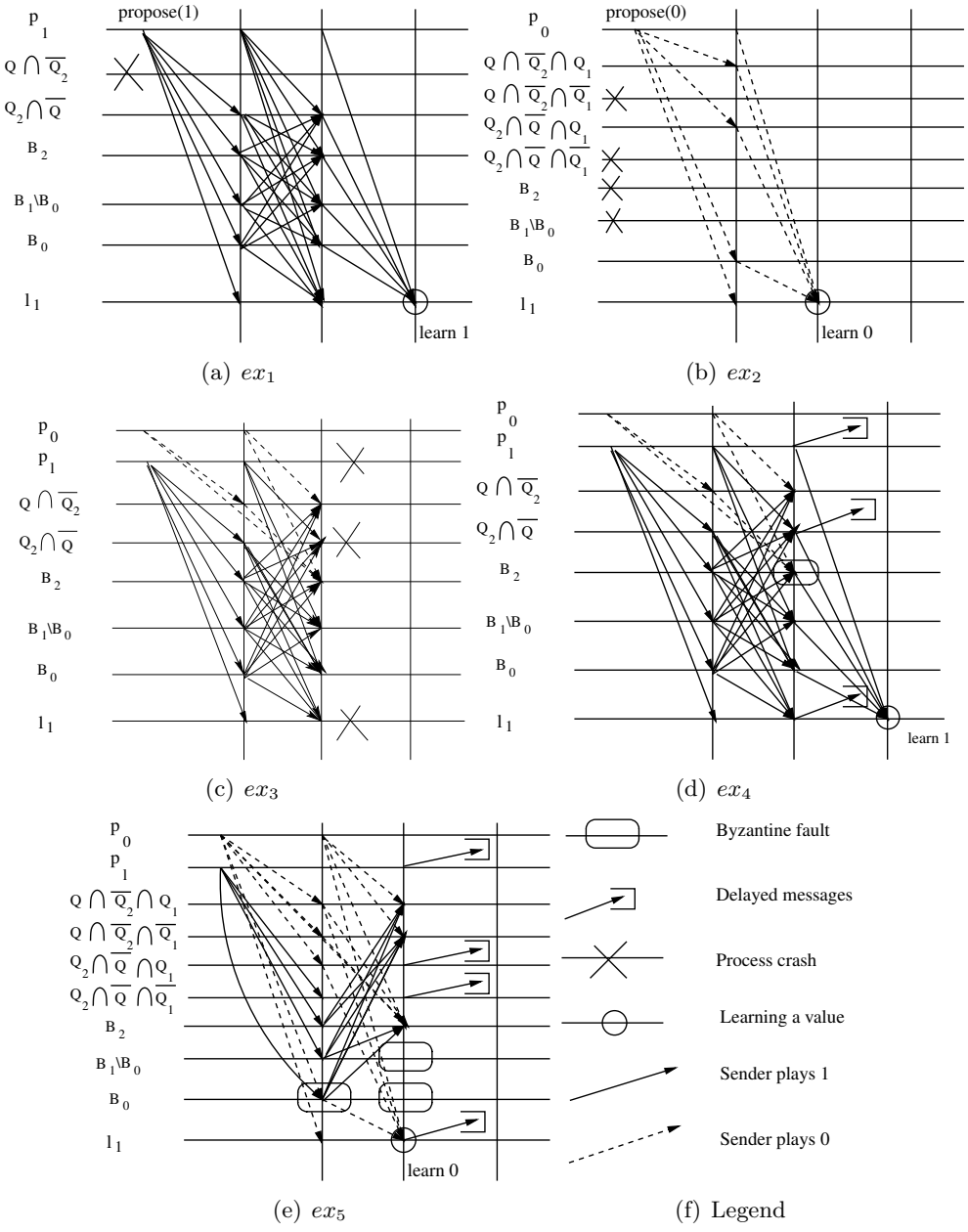
We only consider the cases where p_0 proposes 0 and p_1 proposes 1 (as this is sufficient to prove the theorem). Let $m0 = m_0^{p_0}$ (resp., $m1 = m_1^{p_1}$) be the message sent by p_0 (resp., p_1) in round 1 of some best case execution ex,when p_0 (resp., p_1) is correct and proposes 0 (resp., 1). We say that a process a_i *plays* 0 (resp. 1) to some process a_j in round 2 of ex if a_j cannot distinguish, at round 2, execution ex from some execution ex' in which (1) a_i has received $m0$ (resp. $m1$) from p_0 (resp., p_1) in the first round, and (2) a_i is correct.

To exhibit a contradiction, we construct several (partial) executions (sketched in Figure 3.15) of the algorithm A, including the one in which *agreement* is violated. In these executions, we consider only processes from set $acceptors \cup \{p_0, p_1, l_1, l_2\}$. Other processes can be assumed without loss of generality to fail by crashing at the beginning of each of the following executions.

- Let ex_1 be the best-case execution in which all processes fail by crashing at the beginning of ex_1, except p_1, l_1 and acceptors from Q_2 that are correct in ex_1 (such an execution is possible since p_1 and l_1 are not acceptors, and, hence, p_1 and l_1 are not in $\overline{Q_2}$). Moreover, a correct proposer p_1 proposes 1 in round 1 at the beginning of ex_1 (we denote this time by t_0). Since the system is synchronous in the first three rounds of a best-case execution ex_1 (i.e., during $[t_0, t_0 + 3\Delta]$), all round 1-3 messages exchanged among all correct processes are delivered in ex_1. By our assumption that A is $(2, \boldsymbol{Q_2})$–*fast*, l_1 learns 1 by the end of round 3 (i.e., in three message-delays).

- Let ex_2 be the best case execution in which all processes fail by crashing at the beginning of ex_2, except p_0, l_1 and acceptors from Q_1 that are correct in ex_2 (such an execution is possible since p_l and l_1 are not acceptors, and, hence, p_l and l_1 are not in $\overline{Q_1}$). Moreover, a correct proposer p_l proposes 0 in round 1 at the beginning of ex_2 (i.e., at time t_0). Since the system is synchronous in the first two rounds of a best-case execution ex_1 (during $[t_0, t_0 + 2\Delta]$), all round 1 and 2 messages exchanged among all correct processes are delivered in ex_2. By our assumption that A is $(1, \boldsymbol{Q_1})$–*fast*, l_1 learns 0 by the end of round 2 (i.e., in two message-delays).

- Let ex_3 be a partial execution in which all processes are correct *except* (1) acceptors from $\overline{Q}$, (2) learner l_1 and (3) proposer p_1, which all crash in ex_3 at the beginning of round 3. At the beginning of ex_3, at time t_0, both proposers p_0 and p_1 propose values 0 and 1, respectively. The messages sent in the first two rounds of ex_3 are delivered as follows:
  - (Round 1 messages.) By the end of round 1: processes from $Q_2 \cup \{l_1\}$ receive $m1$, while processes from $\overline{Q_2} \cup \{l_2\}$ receive $m0$. Moreover,



(a) ex_1 (b) ex_2

(c) ex_3 (d) ex_4

(e) ex_5 (f) Legend

Figure 3.15: Illustration of the partial executions used in the proof of Theorem 6. Only acceptors that belong to set $Q_2 \cup Q$. For clarity, learner l_2, as well as messages received by proposers and those sent by learner l_1 are not depicted.

acceptors from B_2 receive the message from p_0 (i.e., $m0$) in round 2, while those from B_1 receive $m0$ in round 3. No other process receives $m1$ (since p_1 crashes in ex_3).

– (Round 2 messages) The following round 2 messages are delivered by the end of round 2: (a) from $\{p_0, l_2\} \cup Q$ to p_0, l_2, $Q \cap \overline{Q_2}$ and B_2, (b) from $\{p_1, l_1\} \cup Q_2$ to p_1, l_1, $Q_2 \cap \overline{Q}$ and B_1. Note that, at the end of round 1, processes from $Q_2 \cup \{p_1, l_1\}$ (resp., $\overline{Q_2} \cap Q_1$) cannot distinguish ex_3 from ex_1 (resp., ex_2) and therefore play 1 (resp., 0) to all processes in round 2. Moreover, acceptors from B_1 receive the round 2 messages sent by processes from $(Q \cap \overline{Q_2}) \cup \{p_0, l_2\}$ in round 3. Finally, no other round 2 message is delivered in ex_3, (this is possible, since the only remaining round 2 messages are (a subset of) those sent by/to processes from $\overline{Q} \cup \{p_1, l_1\}$ which fail by crashing in ex_3). In particular, note that processes from $\{p_0, l_2\} \cup B_2 \cup (Q \cap \overline{Q_2})$ never receive any round 2 message sent by processes from $\{p_1, l_1\} \cup (Q_2 \cap \overline{Q})$.

Note that, in ex_3, by the end of round 2, all processes from $\{p_0, p_1, l_1, l_2\} \cup Q_2 \cup Q$ receive all the messages sent in the first two rounds by (a) at least one proposer, (b) some quorum of acceptors, and (c) at least one learner. Processes from $\{p_1, l_1\} \cup (Q_2 \cap \overline{Q}) \cup B_1$ cannot distinguish, at the end of round 2, ex_3 from ex_1, while processes from $\{p_0, l_2\} \cup (Q \cap \overline{Q_2}) \cup B_2)$ cannot wait for any additional round 1 or round 2 message since these processes received all the messages sent by correct processes in first two rounds. Therefore, in ex_3 no process from $\{p_0, p_1, l_1, l_2\} \cup Q_2 \cup Q$ waits for any additional message before moving to round 3. At the beginning of round 3 processes from $\overline{Q} \cup \{p_1, l_1\}$ fail by crashing such that no process receives any message sent by some of these processes in round 3. Furthermore, assume that in every round $i \geq 3$, all round i messages exchanged among correct processes are delivered by the end of round i. Since (a) p_0 is correct in ex_3 and proposes a value (note that p_0 can be correct in ex'_3 since $p_0 \notin \{p_1, l_1\} \cup \overline{Q}$), (b) there is a quorum of correct acceptors $Q \in \boldsymbol{Q^{(3)}}$, (c) the system is eventually synchronous, and (d) A implements $(\boldsymbol{Q^{(3)}}, \boldsymbol{B})$–*consensus*, eventually a correct learner l_2 (l_2 can be correct since, $l_2 \notin \{p_1, l_1\} \cup \overline{Q}$) learns some value $v \in \{0, 1\}$, say at round K.

- Let ex_4 be an execution identical to ex_3, except that, in ex_4:
  – Processes from $\overline{Q} \cup \{p_1, l_1\}$ do not crash at the beginning of round 3, but due to asynchrony, no message sent in rounds 3 to K by some process from $\overline{Q} \cup \{p_1, l_1\}$ is delivered before round $K+1$, with the exception of round 3 messages sent by processes from $\{p_1, l_1\} \cup (Q_2 \cap \overline{Q})$ to learner l_1 (we assume here that, in ex_4, GST is not reached by the end of round K).
  – Acceptors from B_2 are Byzantine and in round 3 send the same message as in the round 3 of ex_1 to l_1 (i.e., as if they received round 2 messages (only) from $Q_2 \cup \{p_1, l_1\}$). Otherwise, acceptors from B_2 in rounds 3 to K respect the algorithm A and send the same messages as in ex_3.

– In round 3 of ex_4, the messages sent by $\{l_1, p_l\} \cup Q_2$ (including those from B_2) are delivered to l_1; other messages sent to l_1 are delayed until after round K. Since benign processes $\{l_1, p_l\} \cup Q_2$ do not distinguish round 2 of ex_3 (and, hence, ex_4) from round 2 of ex_1, they send the same messages in round 3 of ex_4 as in round 3 of ex_1.

Hence, at the end of round 3, l_1 cannot distinguish ex_4 from ex_1 and learns 1. Other round 3 and later messages in ex_4 are delivered as in ex_3. Since proposer p_0 is correct in ex_4, (note that $p_0 \notin \{p_1, l_1\} \cup B_2 \cup \overline{Q}$), a correct learner l_2 ($l_2 \notin \{p_1, l_1\} \cup B_2 \cup \overline{Q}$) cannot distinguish ex_4 and ex_3. Hence, l_2 learns value v by the end of round K. Since both l_1 and l_2 are correct in ex_4, by the *agreement* property, v must equal 1.

- Let ex_5 be an execution in which all processes are correct except acceptors from B_1 that are Byzantine. At the beginning of ex_5, p_0 proposes 0, while p_1 proposes 1. In round 1 of ex_5 the message are delivered exactly as in round 1 of ex_3, except that (1) benign acceptors from Q_1 (including those from $Q_2 \cap \overline{Q} \cap Q_1$) receive $m0$, but not $m1$ and (2) acceptors from B_0 receive both $m1$ and $m0$. In round 2 and later rounds, B_0 plays 1 to processes other than l_1. Moreover all acceptors from Q_1 (including those from B_0) play 0 to l_1 in round 2 (here, benign acceptors from Q_1 obey the algorithm — they cannot distinguish round 1 in ex_5 from that of ex_2) and all such messages are delivered in round 2. Moreover, all other round 2 messages sent to l_1 are delivered in round 3. Clearly, at the end of round 2, l_1 cannot distinguish ex_5 from ex_2 (since l_1 receives the same set of messages from acceptors in round 2, and the same message from p_0 in rounds 1 and 2 in both executions), and hence, learns 0. All other round 2 messages are delivered as in ex_3. Hence, just like in ex_3, in ex_5 no process from $\{p_0, l_2\} \cup (Q \cap \overline{Q_2}) \cup B_2$ receives any round 2 message from any process from $\{p_1, l_1\} \cup (Q_2 \cap \overline{Q})$. Starting from round 3 acceptors from B_1 forge their state as if they received the round 2 messages as in ex_3 and ex_1, i.e., as if all processes from $Q_2 \cup \{p_1, l_1\}$ played 1 to acceptors from B_1 in round 2 of ex_5 (which is actually the case, except for processes from $(Q_2 \cap \overline{Q} \cap Q_1) \cup \{l_1\}$), and acceptors from B_1 received these messages (this is possible since the messages sent in the round 2 of the best-case execution ex_1 are not authenticated). Moreover, no message sent in rounds 3 to K by some process from $\overline{Q} \cup \{p_1, l_1\}$ is delivered before round $K + 1$. All other messages in round 3 and later are delivered as in ex_3. Since a correct learner l_2 cannot distinguish ex_5 and ex_3 (note that $p_0, l_2 \notin \{p_1, l_1\} \cup B_1 \cup \overline{Q}$), l_2 learns value v by the end of round K. Since v equals 1 (see ex_4), and both l_1 and l_2 are correct, in ex_5 *agreement* is violated.

□

3.5 Correctness of the atomic storage algorithm

In this Section, we prove the correctness of our atomic storage algorithm of Section 3.3.1. For simplicity of presentation, we introduce the following notation

and definitions.

Definition 5. *Consider a set of elements S and an adversary for S, $\boldsymbol{B}$. We say that $Q \subseteq S$ is a* basic *(resp.,* large*) subset (of S), if Q is not a subset of any element (resp., a union of any two elements) of an adversary structure, i.e., $Q \notin \boldsymbol{B}$ (resp., $\forall B_1, B_2 \in \boldsymbol{B}$: $Q \nsubseteq (B_1 \cup B_2)$).*

We say that a base object *responds to*, or *acks* a $\mathsf{wr}\langle ts, *, *, rnd\rangle$ (resp., $\mathsf{rd}\langle tsr, rnd\rangle$) message from client, if base object sends a $\mathsf{wr\_ack}\langle ts, rnd\rangle$ (resp., $\mathsf{rd\_ack}\langle tsr, rnd, *\rangle$) to the client. We say that a benign base object s_i *stores* value $c.val$ and timestamp $c.ts$ (in round rnd), if, at some point in time, $history_i[c.ts, rnd].tsval = c = \langle c.ts, c.val\rangle \neq \langle ts_0, \perp\rangle$.

We first prove atomicity and then we proceed to wait-freedom and complexity. To prove atomicity, we first prove few lemmas.

Lemma 1. ***Size of basic sets.*** *In every execution of our storage algorithm, any basic subset of base objects contains at least one benign base object.*

Proof. The lemma follows directly from the definition of a basic subset (Definition 5). □

Lemma 2. ***Size of large sets.*** *In every execution of our storage algorithm, for any large subset T_2 of base objects, there is a basic subset $T_1 \subseteq T_2$ that contains only benign base objects.*

Proof. Let B_{ex} an element of an adversary structure that contains all Byzantine base objects in execution ex. By Definition 5, for any large subset T_2, $T_2 \setminus B_{ex}$ is a basic subset. By our assumption on B_{ex}, $T_2 \setminus B_{ex}$ contains only benign base objects in ex. Hence the lemma. □

Lemma 3. ***Values written by readers.*** *If some reader r sends a $\mathsf{wr}\langle ts, v, *, *\rangle$ to base objects, then r executes line 35, Fig. 3.6, and assigns $c_{sel} = \langle ts, v\rangle$ (we simply say r* selects *timestamp/value pair c_{sel}).*

Proof. By our assumption that readers are benign and by trivial inspection of the read pseudocode in Fig. 3.6. □

Lemma 4. ***No-creation.*** *If a read rd selects timestamp/value pair $c_{sel} = \langle c_{sel}.ts, c_{sel}.val\rangle$ in line 35, Fig. 3.6, then either $c_{sel}.val$ was written by some write, or $c_{sel}.val$ is the initial value $\perp$.*

Proof. Suppose by contradiction that some read selects c_{sel} with timestamp $c_{sel}.ts$ that is neither $\perp$, nor written by some write. In that case, let rd be the first read (according to the global clock) to select such timestamp/value pair c_{sel} in line 35, Fig. 3.6, at time t. Therefore, up to time t, by Lemma 3, readers send only wr messages containing written values or $\perp$. Hence, no benign base object stores a value not written by some write that is different from $\perp$ by the time t.

Note that $safe(c_{sel})$ holds (line 33, Fig. 3.6), i.e., base objects from a *basic* subset T have sent a read_ACK message containing c_{sel} in their $history[c_{sel}.ts, 1]$ or $history[c_{sel}.ts, 2]$ variables. By Lemma 1 at least one benign base object $s_i \in T$ stored $c_{sel}.val \neq \perp$ before t, such that $c_{sel}.val$ was not written by some write. A contradiction. □

Lemma 5. ***Stored values.*** *Benign objects store only timestamp/value pairs written by some* write *(or initial pair $\langle ts_0, \perp \rangle$).*

Proof. By Lemmas 3 and 4. □

Lemma 6. ***No ambiguity.*** *No two benign base objects ever store different values with the same timestamp (i.e., for every two benign base objects s_i and s_j, any ts, and any $\rho_i, \rho_j \in \{1, 2, 3\}$, $(history_i[ts, \rho_i].val \neq \perp) \wedge (history_j[ts, \rho_j].val \neq \perp) \Rightarrow history_i[ts, \rho_i] = history_j[ts, \rho_j]$).*

Proof. By Lemma 5, the assumption that the writer is benign and the fact that the writer never assigns different values to the same timestamp. □

Lemma 7. ***Non-decreasing timestamps.*** *For any $\rho \in \{1, 2, 3\}$, the highest timestamp stored in round ρ by a benign base object s_i never decreases, (i.e., the highest value ts for which $history_i[ts, \rho] \neq \langle ts_0, \perp \rangle$ never decreases).*

Proof. From base object code inspection (line 3, Fig. 3.5), once a timestamp/value pair is stored in $history_i[ts, \rho]$, it is never overwritten by a different timestamp/value pair. Hence the lemma. □

The following lemma is crucial for proving *atomicity*. We make an extensive use of this lemma in the rest of the proof.

Lemma 8. ***Locking the value.*** *Assume that, for every quorum Q, by time t', at least one of the following three properties holds:*

- *(a) there is a basic subset T_Q that contains* only *benign servers, such that $T_Q \subseteq Q$, such that, every element $s_i \in T_Q$ stores $history_i[ts_i, 1].tsval = c_Q = \langle ts_i, v_i \rangle \neq \langle ts_0, \perp \rangle$, where $ts_i \geq ts'$,*
- *(b) a benign server $s_i \in Q$ stores $history_i[ts_i, 2].tsval = c_Q = \langle ts_i, v_i \rangle \neq \langle ts_0, \perp \rangle$, where $ts_i \geq ts'$,*
- *(c) a benign server $s_i \in Q$ stores $history_i[ts_i, 1].tsval = c_Q = \langle ts_i, v_i \rangle \neq \langle ts_0, \perp \rangle$ and exists a class 2 quorum Q_2 such that $Q_2 \in history_i[ts_i, 1].sets$ and $P_{3b}(Q_2, Q)$ holds, where $ts_i \geq ts'$.*

Then, a complete read *rd invoked after t' cannot return value $c.val$ such that $c.ts < ts'$.*

Proof. We prove that $highCand(c)$ cannot hold in rd. Denote by $\boldsymbol{Q}(t)$ the set of quorums that responded to the reader during rd by time t, where $t \geq t_{inv}(rd) > t'$, where $t_{inv}(rd)$ denotes the time rd is invoked. More precisely, we say that quorum Q responds to the reader (during rd) if the reader received at least one rd_ack message (during rd) from every server from Q. Notice that, by definition of $\boldsymbol{Q}(t)$, $t_1 < t_2 \Rightarrow \boldsymbol{Q}(t_1) \subseteq \boldsymbol{Q}(t_2)$. Moreover, denote by $c(t)$ timestamp/value pair c such that there is $Q \in \boldsymbol{Q}(t)$, such that $c = c_Q$, and for all $Q' \in \boldsymbol{Q}(t)$, $c_Q.ts \leq c_{Q'}.ts$. In other words, $c(t)$ is timestamp/value pair c_Q with the *minimum timestamp* over all quorums Q that responded to the reader by time t. Notice here that, since $t_1 < t_2 \Rightarrow \boldsymbol{Q}(t_1) \subseteq \boldsymbol{Q}(t_2)$, $c(t).ts$ is monotonically decreasing. We show that, for

any $t \geq t_{inv}(rd)$ (in the following we assume only such times t), there is a server s_i, such that $read(c(t), i)$ and $invalid(c(t))$ does not hold. Since, by definitions of $c(t)$ and properties (a)–(c), $c(t).ts > c.ts$ for any t, $highCand(c)$ cannot hold.

Assume by contradiction that there is time t such that, at time t: (i) there is no server s_i such that $read(c(t), i)$, or (ii) $invalid(c(t))$. It is trivial to see that (i) contradicts definitions of $c(t)$ and $\boldsymbol{Q}(t)$; establishing a contradiction in case (ii) is less obvious.

Let t_1 be time such that $invalid(c(t_1))$ holds. There are two possibilities (see line 9, Fig. 3.6): (1) $c(t_1).ts > highest\_ts$, or (2) there is quorum $Q' \in \boldsymbol{RQS}$ such that none of the predicates $valid_1(c(t_1), Q')$, $valid_{2a}(c(t_1), Q')$ and $valid_{2b}(c(t_1), Q')$ holds.

Consider case (1). Denote by t_{first} the time the first round of rd ends. By definitions of $c(t)$ and $highest\_ts$ (line 29, Fig. 3.6), $c(t_{first}).ts \leq highest\_ts$. Hence, $c(t_1).ts > highest\_ts$ contradicts the fact that $c(t)$ is a monotonically decreasing function.

Consider now case (2). We distinguish three cases depending on which of the properties (a)–(c) holds for Q'.

- First, assume property (a) holds for Q'. In this case, since $valid_1(c(t_1), Q')$ does not hold, all servers from all basic subsets of Q' respond in rd by t_1, such that, for every such a basic subset $T_{Q'}$ there is a benign server $s_i \in T_{Q'}$ such that $last[i, 1].ts < c(t_1).ts$. Notice here that, by Lemma 6, it is not possible that $last[i, 1].ts = c(t_1).ts$ and $last[i, 1].val \neq c(t_1).val$. As a corollary of Lemma 7, timestamp $last[i, 1].ts$ is monotonically increasing (for any benign server, including s_i). Hence, by property (a) of Q', we have $c_{Q'}.ts \leq last[i, 1].ts$. Finally, we have $c_{Q'}.ts < c(t_1).ts$, which contradicts the definition of $c(t)$.

- Assume now property (b) holds for Q'. Since $valid_{2a}(c(t_1), Q')$ does not hold, every benign server $s_i \in Q'$ responds to the reader in rd by t_1 such that $last[i, 2].ts < c(t_1).ts$. Notice again that, by Lemma 6, it is not possible that $last[i, 2].ts = c(t_1).ts$ and $last[i, 2].val \neq c(t_1).val$. As a corollary of Lemma 7, timestamp $last[i, 2].ts$ is monotonically increasing (for any benign server s_i). Hence, by property (b) of Q', $c_{Q'}.ts \leq last[i, 2].ts$. Finally, we have $c_{Q'}.ts < c(t_1).ts$, which contradicts the definition of $c(t)$.

- Assume now property (c) holds for Q'. Since $valid_{2b}(c(t_1), Q')$ does not hold, every benign server $s_i \in Q'$ responds to the reader in rd by t_1 such that, for every $Q_2 \in \boldsymbol{QC_2}$, $last_{set}[i, Q_2].ts < c(t_1).ts$. By Lemma 6 and the definition of predicate $last_{set}[i, Q]$, line 5, Fig. 3.6, it is not possible that (for some $Q_2 \in \boldsymbol{QC_2}$) $last_{set}[i, Q_2].ts = c(t_1).ts$ and $last_{set}[i, Q_2].val \neq c(t_1).val$. Since, by Lemma 7, the highest value ts for which $history_i[ts, 1] \neq \langle ts_0, \perp \rangle$ never decreases, and since, by server code (Fig. 3.5), no set is ever removed from $history_i[ts, 1].sets$, timestamp $last_{set}[i, Q_2].ts$ is monotonically increasing (for any benign server s_i and any class 2 quorum Q_2). Hence, by property (c) of Q', there is a class 2 quorum Q_2 such that $c_{Q'}.ts \leq last_{set}[i, Q_2].ts$. Finally, we have $c_{Q'}.ts < c(t_1).ts$, which contradicts the definition of $c(t)$.

□

Now we prove atomicity of read operations.

Lemma 9. ***read/write atomicity.*** *If a read rd is complete and it follows some complete $wr = \mathsf{write}(v)$, then rd does not return a value older than v.*

Proof. First, recall that timestamps monotonically increase at the writer. Moreover, by Lemma 3 and 4, all values that a reader may write to a base object (and, all values reader may return) are written by the writer (or the value is $\perp$). Therefore, timestamps associated by the writer to the written values, totally order values that readers return. Now we show that rd does not return $c'.val$, such that $c'.ts < ts$, where ts is associated to v in $\mathsf{write}(v) = wr$.

First, suppose that wr completes in a single round trip. Then, a pair $\langle ts, v\rangle$ is written to $history_*[ts, 1].tsval$ variables of all benign base objects of some class 1 quorum Q_1. By Property 2 of RQS, and Definition 5, for every quorum Q, $Q_1 \cap Q$ is a large subset, and every large subset is a superset of a basic subset that contains only benign base objects (by Lemma 2). Therefore, for every quorum Q there is a basic subset T_Q that contains only benign base objects such that $T_Q \subseteq Q$ and $T_Q \subseteq Q_1$. Hence, we can apply Lemma 8 (for all quorums property (a) holds), and conclude that rd does not return a value with a timestamp smaller than ts, i.e., a value older than v.

Now, suppose that wr completes in two or three round-trips. Then, a pair $\langle ts, v\rangle$ is written to $history_*[ts, 2].tsval$ variables of all benign base objects of some quorum Q'. Since any quorum intersects with any other quorum Q in a basic subset (by Property 1 of RQS, and Definition 5), $Q' \cap Q$ is a basic subset that contains at least one benign base object (by Lemma 1). Therefore, for every quorum Q there is a benign base object s_{i_Q} such that $s_{i_Q} \in Q$ and $s_{i_Q} \in Q'$. Hence, we can apply Lemma 8 (for all quorums property (b) holds), and conclude that rd does not return a value with a timestamp smaller than ts, i.e., older than v.

□

Now we prove atomicity among read operations.

Lemma 10. ***read atomicity.*** *If a read rd is complete and it follows some complete read rd' that returns v', then rd does not return a value older than v'.*

Proof. We show that rd does not return $c.val$, such that $c.ts < ts'$, where ts' is associated to v' by the write that wrote v', (for brevity, we use $c' = \langle ts', v'\rangle$). We consider three exhaustive cases, where : (1) rd' completes in a single round, (2) rd' completes in exactly two rounds, and (c) rd' completes in more than two rounds.

1. If rd' completes in a single round, then, at the end of the first round of rd', $BCD(c', 1, *)$ holds. We consider the following three exhaustive cases where: (a) $BCD(c', 1, 1)$ holds, (b) $BCD(c', 1, 2)$ holds, and (c) $BCD(c', 1, 3)$ holds.

a) In this case there exist class 1 quorums Q_1 and Q_1' (possibly $Q_1 = Q_1'$), such that, at the end of round 1 of rd', for all benign base objects $s_i \in Q_1 \cap Q_1' = X$, $history_i[ts', 1].tsval = c'$. Since, by Property 2 of RQS and Definition 5 an intersection of a pair of class 1 quorums with any quorum Q is a large subset. Hence, $X \cap Q$ is a large subset, and every large subset is a superset of a basic subset that contains only benign base objects (by Lemma 2). Therefore, for every quorum Q there is a basic subset T_Q that contains only benign base objects such that $T_Q \subseteq Q$ and $T_Q \subseteq X$. Hence, we can apply Lemma 8 (for all quorums property (a) holds), and conclude that rd does not return a value with a timestamp smaller than ts', i.e., older than v'.

b) In this case there exist a class 1 quorum Q_1 and a class 2 quorum Q_2, such that, at the end of round 1 of rd', for all benign base objects $s_i \in Q_1 \cap Q_2 = X$, $history_i[ts', 2] = \langle c', \boldsymbol{QC_2''} \rangle$ and $Q_2 \in \boldsymbol{QC_2''}$. Note that there is at least one benign base object in X by Property 1 of RQS (X is a basic subset as an intersection of two quorums) and Lemma 1. Note also that, by writer and base object code, a benign object s_i stores $history_i[ts', 2] = \langle c', \boldsymbol{QC_2''} \rangle$, where $Q_2 \in \boldsymbol{QC_2''}$ only if all benign base objects from Q_2 have stored $history_*[ts', 1].tsval = c'$. Moreover, since Q_2 is a class 2 quorum, for any quorum Q: (i) $Q_2 \cap Q$ is a large subset (i.e., $P_{3a}(Q_2, Q)$ holds), or (ii) $X \cap Q$ is a basic subset (i.e., $P_{3b}(Q_2, Q)$ holds).

   i. In this case, $Q_2 \cap Q$ is a large subset, and every large subset is a superset of a basic subset that contains only benign base objects (by Lemma 2). Therefore, there is a basic subset T_Q that contains only benign base objects such that $T_Q \subseteq Q$ and $T_Q \subseteq Q_2$. I.e., all (benign) base objects in T_Q store $history_*[ts', 1].tsval = c'$, before rd' completes. Hence, for all quorums Q, for which $P_{3a}(Q_2, Q)$ holds, property (a) of Lemma 8 is also satisfied.

   ii. $X \cap Q$ is a basic subset, and every basic subset contains at least one benign base object (by Lemma 1). Therefore, there is a benign base object s_{i_Q} such that $s_{i_Q} \in Q$ and $s_{i_Q} \in X$. Hence, s_{i_Q} stores $history_{i_Q}[ts', 2].tsval = c'$ before rd' completes. Hence, for all quorums Q, for which $P_{3b}(Q_2, Q)$ holds, property (b) of Lemma 8 is also satisfied.

   Hence, we can apply Lemma 8, and conclude that rd does not return a value with a timestamp smaller than ts', i.e., older than v'.

c) In this case there exist a class 1 quorum Q_1 and a quorum Q, such that, at the end of round 1 of rd', for all benign base objects $s_i \in Q_1 \cap Q = X$, $history_i[ts', 3].tsval = c'$. Since, by Property 1 of RQS, any quorum intersection is a basic subset, by Lemma 1, X contains at least one benign base object. Note also that, by base object code, a benign base object stores $history_i[ts', 3].tsval = c'$, only upon all benign base objects from some quorum Q' store $history_i[ts', 2].tsval = c'$. By Property 1 of RQS and Definition 5, $Q' \cap Q''$ is a basic subset for any

quorum Q''. Moreover, every basic subset contains at least one benign base object (by Lemma 1). Therefore, for every quorum Q'' there is a benign base object $s_{i_{Q''}}$ such that $s_{i_{Q''}} \in Q''$ and $s_{i_{Q''}} \in Q'$. Hence, we can apply Lemma 8 (for all quorums property (b) holds), and conclude that rd does not return a value with a timestamp smaller than ts', i.e., older than v'.

2. If rd' completes in exactly two rounds, then $\exists i \in \{1, 2, 3\} : BCD(c_{sel}, 2, i) \neq \emptyset$ (line 41, Fig. 3.6). We consider the following two exhaustive cases, where, in rd': (a) $BCD(c', 2, i) \neq \emptyset$ holds for $i = 2$ or $i = 3$, and (b) $BCD(c', 2, i) \neq \emptyset$ holds only for $i = 1$.

   a) In this case, a client that invoked rd' received acks for its $\mathsf{wr}\langle c'.ts, c'.val, \emptyset, 2\rangle$ message from at least a quorum Q' of base objects (when executing the *writeback* procedure in line 42). Hence, by the time rd' completes, all benign base objects from Q' store $history_i[ts', 2].tsval = c'$. By Property 1 of RQS and Definition 5 for any quorum Q, $Q' \cap Q$ is a basic subset. Moreover, every basic subset contains at least one benign base object (by Lemma 1). Therefore, for every quorum Q there is a benign base object s_Q such that $s_Q \in Q$ and $s_Q \in Q'$. Hence, we can apply Lemma 8 (for all quorums property (b) holds), and conclude that rd does not return a value with a timestamp smaller than ts', i.e., older than v'.

   b) In this case, since $BCD(c', 2, 1) \neq \emptyset$, a Q_2 is an element of $BCD(c', 2, 1)$ if: (I) Q_2 is a class 2 quorum that responded in the first round of rd', and there is a class 1 quorum Q_1, such that for all base objects from $X = Q_1 \cap Q_2$ stored $history_*[ts', 1].tsval = c'$. In the round 2 of rd' the reader sends the $\mathsf{wr}\langle c'.ts = ts', c'.val, BCD(c', 2, 1), 1\rangle$ to all base objects. Since rd' completes in exactly two rounds, some (class 2) quorum from $BCD(c_{sel}, 2, 1)$ responds in the round 2 of rd' as well. Denote this quorum by Q'_2. Since Q'_2 is a class 2 quorums, for any quorum Q, at least one of the following two properties hold: (i) $P_{3a}(Q'_2, Q)$, i.e., $Q'_2 \cap Q$ is a large subset, or (ii) $P_{3b}(Q'_2, Q)$ holds, i.e., $X \cap Q$ is a basic subset.

      i. In this case, $Q_2 \cap Q$ is a large subset, and every large subset is a superset of a basic subset that contains only benign base objects (by Lemma 2). Therefore, there is a basic subset T_Q that contains only benign base objects such that $T_Q \subseteq Q$ and $T_Q \subseteq Q_2$. Recall that all benign base objects from Q_2 have stored $history_*[ts', 1].tsval = c'$, before rd' completes. Hence, for all quorums Q, for which $P_{3a}(Q_2, Q)$ holds, property (a) of Lemma 8 is also satisfied.

      ii. $X \cap Q$ is a basic subset, and every basic subset contains at least one benign base object (by Lemma 1). Therefore, there is a benign base object s_{i_Q} such that $s_{i_Q} \in Q$ and $s_{i_Q} \in X$. Hence, s_{i_Q} stores $history_{i_Q}[ts', 1].tsval = c'$ and $Q'_2 \in history_{i_Q}[ts', 1].sets$ before

rd' completes. Hence, for all quorums Q, for which $P_{3b}(Q_2, Q)$ holds, property (c) of Lemma 8 is also satisfied.

Hence, we can apply Lemma 8, and conclude that rd does not return a value with a timestamp smaller than ts', i.e., older than v'.

3. If rd' completes in more than two rounds, then a client that invoked rd' received acks for its $\mathsf{wr}\langle c'.ts, c'.val, 2\rangle$ message from at least a quorum Q' of base objects (when executing the *writeback* procedure in line 47, or line 49). Hence, by the time rd' completes, all benign base objects from Q' store $history_i[ts', 2] = c'$. By Property 1 of RQS and Definition 5, for any quorum Q, $Q' \cap Q$ is a basic subset. Moreover, every basic subset contains at least one benign base object (by Lemma 1). Therefore, for every quorum Q there is a benign base object s_Q such that $s_{s_Q} \in Q$ and $s_Q \in Q'$. Hence, we can apply Lemma 8 (for all quorums property (b) holds), and conclude that rd does not return a value with a timestamp smaller than ts', i.e., older than v'.

□

Theorem 7. ***Atomicity.*** *The algorithm in Figures 3.4, 3.5 and 3.6 is atomic.*

Proof. By Lemmas 4, 9 and 10. More specifically:

- atomicity property (SWA1) (see Section 2.3.2) follows from Lemma 4,
- (SWA2) is equivalent to Lemma 9,
- (SWA3) follows from Lemmas 4 and 9, and
- (SWA4) is equivalent to Lemma 10.

□

We proceed to prove the wait-freedom property. First we prove the following important lemma.

Lemma 11. ***Safety of 2-round read.*** *Let Q_c be a quorum that contains only correct base objects. No base object $s_i \in Q_c$ stores $history_i[c.ts, 2].tsval = c$, or stores $Q_2 \in history_i[c.ts, 1].sets$, for some class 2 quorum Q_2, such that $P_{3b}(Q_2, Q_c)$, before there is a basic subset $T \subseteq Q_c$ such that all $s_j \in T$ stored $history_j[c.ts, 1].tsval = c$.*

Proof. First we prove the first part of the lemma (no $s_i \in Q_c$ stores $history_i[c.ts, 2].tsval = c$).

By the algorithm's pseudocode, before any client c sends a $\mathsf{wr}\langle c.ts, c.val, *, 2\rangle$ message to base objects, c has already received acks for its $\mathsf{wr}\langle c.ts, c.val, *, 1\rangle$ message from some quorum Q, except in case of a writeback in line 42. In all other cases, $Q \cap Q_c$ is a basic subset (by Property 1 of RQS and Definition 5) — hence the lemma.

In the case of a writeback in line 42, assume, by contradiction, that there is a read rd that issues a message $\mathsf{wr}\langle c.ts, c.val, *, 2\rangle$, such that no basic subset

$T \subseteq Q_c$ previously stored $history_j[c.ts, 1] = c$. Moreover, let rd be the first such read according to the global clock that executes the *writeback* procedure in line 42, at time t.

In this case, $BCD(c, 2, 2)$ or $BCD(c, 2, 3)$ hold in line 42 of rd. If $BCD(c, 2, 2)$ (resp., $BCD(c, 2, 3)$) holds, then there exist two class 2 quorum Q_2 and Q_2' (resp., a class 2 quorum Q_2 and a quorum Q) such that, for all benign base objects from $Q_2 \cap Q_2' = X$ (resp., $Q_2 \cap Q = X$), $history_*[c.ts, 2] = c$ (resp., $history_*[c.ts, 3] = c$) holds. By Property 1 of RQS, Definition 5 and Lemma 1, there is at least one benign base object in X, i.e., by the end of round 1 of rd at least one benign base object received $\mathsf{wr}\langle c.ts, c.val, *, 2\rangle$ (resp., $\mathsf{wr}\langle c.ts, c.val, *, 3\rangle$) from some client. By algorithm pseudocode and by our assumption on rd, before time t, a client c sends a message $\mathsf{wr}\langle c.ts, c.val, *, 2\rangle$ (resp., $\mathsf{wr}\langle c.ts, c.val, true, 3\rangle$) only upon c received acks for its $\mathsf{wr}\langle c.ts, c.val, *, 1\rangle$ message from some quorum Q' of base objects. Note that $Q' \cap Q_c = T_c$ is a desired basic subset. A contradiction.

Now we prove the first part of the lemma (no $s_i \in Q_c$ stores $Q_2 \in history_i[c.ts, 1].sets$).

The only case to analyze is the writeback in line 44. Assume, by contradiction, that there is a read rd that issues a message $\mathsf{wr}\langle c.ts, c.val, SET, 1\rangle$, where some class 2 quorum $Q_2 \in SET$, such that $P_{3b}(Q_2, Q_c)$, such that no basic subset $T \subseteq Q_c$ previously stored $history_j[c.ts, 1] = c$. Moreover, let rd be the first such read according to the global clock that executes the *writeback* procedure in line 44, at time t.

In this case, $BCD(c, 2, 1)$ holds in line 41 of rd, and condition in line 42 is not satisfied. If $BCD(c, 2, 1)$ holds, then there exist a class 1 quorum Q_1 such that for all benign base objects from $Q_1 \cap Q_2 = X$, $history_*[c.ts, 1].tsval = c$ holds. Since $P_{3b}(Q_2, Q)$ holds, $X \cap Q_c = T_c$ is a desired basic subset, a contradiction. □

Theorem 8. ***(Wait-freedom.)*** *The algorithm in Figures 3.4, 3.5 and 3.6 is wait-free.*

Proof. The argument for the wait-freedom of a write operation is based on the assumption that there is at least one quorum Q_c containing only correct servers. In every round of a write, the writer waits for acks from at least one quorum, so the writer is guaranteed to receive the awaited acks eventually. The timer that the writer awaits eventually expires and write eventually completes.

The argument for the wait-freedom of a read operation is more involved. We show that any read operation invoked by a correct client does not block in line 34, Fig. 3.6; the remainder of the proof is straightforward. We distinguish two cases: (1) the case where there is an infinite (unbounded) number of write operations in the execution, and (2) the case where the writer issues a finite number of write operations in the execution. We denote by Q_c the quorum that contains only correct servers.

1. In this case, there is an infinite number of writes. Suppose, by contradiction, that rd never completes. Let $highest\_ts = ts$ at the end of round 1 of rd. Since the writer issues an unbounded number of writes, the writer will also issue a write with timestamp ts, writing some value v. Since all benign servers from some quorum Q store $history_*[ts', 1] = \langle ts', v'\rangle$ at time t and

since by Property 1 of RQS $Q \cap Q_c$ is a basic subset, after rd receives at least one ack from every server from Q_c sent after t, and, hence $safe\langle ts, v\rangle$ will hold. Moreover, for all other timestamp/value pairs c with $c.ts > ts$, $c.ts > highest\_ts$ will also hold and, hence, $highCand\langle ts, v\rangle$ also eventually holds. Hence $\langle ts, v\rangle$ is eventually in C and rd terminates. A contradiction.

2. In this case, there is a write operation with the highest timestamp. Let wr denote the last complete write operation that writes v with timestamp ts (or $v = v_0$, $ts = \perp$ if there is none). We denote with wr' a possible later (incomplete) write that writes v' with ts'.

   Assume, by contradiction, that read rd never returns a value. Note that $highest\_ts$ denotes the highest timestamp that the reader received in the first round of rd (line 29, Fig. 3.6). First consider the case, where $ts < highest\_ts$.

   Then, rd invokes rounds on all correct servers, sending rd messages infinitely many times. We distinguish two cases: (a) no basic subset T such that $T \subseteq Q_c$ ever stores $history_*[ts', 1] = \langle ts', v'\rangle$, and (b) there is a time t at which some basic subset $T \subseteq Q_c$ stores $history_*[ts', 1] = \langle ts', v'\rangle$. In case (a), let t be the time at which the last correct server stores $history_*[ts', 1] = \langle ts', v'\rangle$. Moreover, let $t' > t$ be the time at which rd receives at least one response from every server from Q_c sent after t (in both cases (a) and (b)).

   a) Since wr completed, there is a quorum Q such that all benign servers from Q have stored $history_*[ts, 1] = \langle ts, v\rangle$. By Property 1 of RQS, $Q \cap Q_c = T_v$ is a basic subset. Hence, from time t' onward, rd received at least one ack from all servers from T_v sent after wr completed and, hence, $\langle ts, v\rangle$ is *safe* (by definition of the predicate $safe$, line 11, Fig. 3.6).

      Moreover, by Lemma 11 and assumption (a), no server from Q_c ever stores $history_*[ts'', 2] = \langle ts'', v'\rangle$, for any timestamp $ts'' > ts$, nor stores some $Q_2 \in history_*[ts'', 1].sets$ such that $P_{3b}(Q_2, Q_c)$. Therefore, for every timestamp-value pair c, such that $\langle ts, v\rangle \prec c$, $valid_{2a}(c, Q_c)$ and $valid_{2b}(c, Q_c)$ do not hold. Finally, by our assumption (a) no $T \subseteq Q_c$ stores $history_*[ts'', 1] = \langle ts'', v'\rangle$ for any $ts'' > ts$. Therefore, after time t', for any value c, such that $\langle ts, v\rangle \prec c$, $valid_1(c, Q_c)$ does not hold. Hence, at the next iteration, $highCand(\langle ts, v\rangle)$ and $safe(\langle ts, v\rangle)$ hold, $\langle ts, v\rangle \in C$ and rd returns, a contradiction.

   b) In this case, after t', $\langle ts', v'\rangle$ there is a basic subset T for which $last[*, 1] = \langle ts', v'\rangle$. Hence, $safe(\langle ts', v'\rangle)$ holds after t'. It is not difficult to see, since no subsequent valid value is present in the system (since wr' is the last write invoked), that for every timestamp/value pair c'' such that $c''.ts > c'.ts \vee (c''.ts = c'.ts \wedge c''.val \neq c'.val)$ none of the predicates $valid_1(c'', Q_c)$, $valid_{2a}(c'', Q_c)$ or $valid_{2b}(c'', Q_c)$ holds, i.e., $invalid(c'')$ holds. Hence, $highCand(\langle ts', v'\rangle)$ also holds. Thus, in the next iteration, $\langle ts', v'\rangle \in C$ and read returns: a contradiction.

   Consider now the case, where $ts \geq highest\_ts$. Since a write wr, with

timestamp ts completed, then, a write with timestamp $highest\_ts$ also completed. It is not difficult to see (along the lines of the proof of case (1)) that rd returns the value written with timestamp $highest\_ts$.

□

Theorem 9. *(**Best-Case Latency.**) The storage algorithm in Figures 3.4, 3.5 and 3.6 is* $(m, \boldsymbol{QC_m})$–fast *for all* $m \in \{1, 2, 3\}$.

Proof. For write operation, the proof is straightforward. For read, it is important to show that whenever the read is synchronous and uncontended, lines 20-35 are executed only once. This proof is given in the following. The rest of the proof is straightforward, by using the output of BCD (lines 1-2, Fig. 3.6).

Since there is no contention, let wr writing timestamp value pair $c = \langle ts, v \rangle$ be the last (complete) write that precedes read rd. Regardless of whether wr completed in 1, 2, or 3 rounds, wr wrote $c = \langle ts, v \rangle$ into some quorum of base objects Q. Moreover, no benign base object stores any value with a higher timestamp than ts by Lemma 5. Since rd is synchronous, a quorum Q_c that contains only correct base object will respond in the first round of rd. By Property 1 of RQS and Definition 5 $Q_c \cap Q = T_c$ is a basic subset that contains only correct base objects, and, hence, $safe(c)$ holds at the end of round 1 of rd. It is not difficult to see that for any value $c'.val$ with $c'.ts > ts$, none of the predicates $valid_1(c', Q_c)$, $valid_{2a}(c', Q_c)$ and $valid_{2b}(c', Q_c)$ will hold. Hence, for any such timestamp/value pair $invalid(c')$ holds. Hence, at the end of round 1 of rd, $highCand(c)$ also holds and hence $c \in C$ in line 33, Fig. 3.6. □

3.6 Correctness of the consensus algorithm

In this Section we prove the correctness of our consensus algorithm of Section 3.4.1. First, we give few definitions.

Definition 6 (Value decided in a view)**.** *We say that value v is* Decided-2*,* Decided-3 *or* Decided-4 *in view w, if there is benign process (acceptor or learner) p that eventually decides a value by receiving (respectively):*

- *(Decided-2)* $\mathsf{update}_1\langle v, w, *\rangle$ *messages from a class 1 quorum (line 51, Fig. 3.14).*
- *(Decided-3)* $\mathsf{update}_2\langle v, w, Q_2\rangle$ *messages from a class 2 quorum Q_2 (line 52, Fig. 3.14).*
- *(Decided-4)* $\mathsf{update}_3\langle v, w, *\rangle$ *messages from some quorum (line 53, Fig. 3.14).*

We also say that value v is decided *in view w, if some benign process p* Decided-m *v in view w (where $m \in \{2, 3, 4\}$).*

Definition 7 (Prepares)**.** *We say that an acceptor a_i* prepares *value v in view w, if it eventually receives* $\mathsf{prepare}\langle v, w, *, *\rangle$ *and executes lines 31-33, Fig. 3.14.*

Definition 8 (Updates)**.** *We say that a benign acceptor a_i* updates *value v in view w, if it eventually receives* $\mathsf{update}_{step}\langle v, w, *\rangle$ *for some step $\in \{1, 2\}$ and executes lines 34-39, Fig. 3.14. More precisely, we say a_i* 1-updates *(resp.,* 2-updates*) v in w if step $= 1$ (resp., step $= 2$).*

Definition 9 (Accepts)**.** *We say that a benign acceptor a_i* accepts *value v in view w, if it prepares or updates v in view w.*

We also make use of the Definition 5 of Section 3.5 (definition of *basic* and *large* subsets).

Lemma 12. ***Size of basic sets.*** *In every execution of our consensus algorithm, any basic subset contains at least one benign acceptor.*

Proof. The lemma follows directly from the definition of a basic subset (Definition 5). □

Lemma 13. ***Size of large sets.*** *In every execution of our consensus algorithm. for any large subset T_2, there is a basic subset $T_1 \subseteq T_2$ that contains only benign acceptors.*

Proof. Let B_{ex} be an element of an adversary structure that contains all Byzantine acceptors in execution ex. By Definition 5, for any large subset T_2, $T_2 \setminus B_{ex}$ is a basic subset. By our assumption on B_{ex}, $T_2 \setminus B_{ex}$ contains only benign acceptors. Hence the lemma.

□

We first prove *Validity.*

Lemma 14. ***Validity of the choose function.*** *If $choose(v, vProof, Q)$ (where $vProof$ consists of valid* new_view_ack *messages) returns v such that v is a candidate with view w, then at least one benign acceptor a_i prepared v in w.*

Proof. Assume $Cand_2(v, w)$ holds (line 2, Fig. 3.12). In this case, every acceptor a_j from the set $X = (Q_1 \cap Q) \setminus B$ (where B is an element of adversary $\boldsymbol{B}$ and Q_1 is a class 1 quorum) reported that it prepared v in w. Note that, by Property 2 of RQS, $Q_1 \cap Q$ is a large subset. Hence, X is a basic subset. By Lemma 12, X contains at least one benign acceptor.

Assume now $Cand_3(v, w, *)$. From line 3, Fig. 3.12, it follows that all acceptors from the set $X = (Q_2 \cap Q) \setminus B$, (where B is an element of adversary $\boldsymbol{B}$ and Q_2 is a class 2 quorum) reported that they updated v in w (i.e., $\forall a_j \in X : vProof[a_j].Update[1] = v$ and $w \in vProof[a_j].Update_{view}[1]$). Note that, by Property 1 of RQS, $Q_2 \cap Q$ is a basic subset. Hence, by Definition 5, X is a non-empty set. In this case, $vProof[a_j].Update_{proof}[1, w]$ contains at least a basic subset of signed $\mathsf{update}_1\langle v, w, *\rangle$ messages. By Lemma 12 at least one of these signed messages comes from a benign acceptor a_i that indeed prepared v in view w.

The argument for the case where $Cand_4(v, w)$ holds (line 4, Fig. 3.12) is very similar to the case where $Cand_3(v, w, *)$ holds.

□

Theorem 10. ***(Validity)*** *If a benign learner learns value v and all proposers are benign, then some proposer proposed v.*

Proof. A benign learner learns value v by receiving (1) update_* messages (lines 51-53 and 60, Fig. 3.14), or (2) by receiving a basic subset of decision messages (line 101, Fig. 3.14). In case (b), by Lemma 12 and line 40, Fig. 3.14 at least one benign acceptor decided v before the learner learned v.

Hence, in both cases, if a learner learns v, then v was accepted in some view w (prepared or updated) by benign acceptors from some quorum of acceptors. Since any quorum is a basic subset, there is at least one such benign acceptor a_j (by Property 1 of RQS, and Lemma 12). Note that a_j updates v in w only upon a_j prepares v in w. We prove the following statement using induction on view numbers: *if a benign acceptor prepares v in view w, then some proposer proposed v.*

Base Step: $(w = initView)$

Benign acceptors prepare value v in $initView$ only if they receive a $\mathsf{prepare}\langle v, initView, *, *\rangle$ message from some proposer. Since all proposers are benign, no proposer sends a prepare message containing v unless it proposes v. Hence, if some benign acceptor accepts v, v was indeed proposed by some proposer.

Inductive Hypothesis (IH): For every view $w, w' > w \geq initView$, if a benign acceptor accepts v in w, then some proposer proposed v.

Inductive Step: We prove the statement is true for view w'. In view w', an acceptor accepts only values returned by $choose(*, vProof, Q)$, where $vProof$ contains valid new_view_ack messages. If $choose(*, vProof, Q)$ returns a candidate value v, by Lemma 14, some benign acceptor prepared v in view $w, w < w'$, and by IH, v was proposed by some proposer.

Finally, if $choose()$ returns value v in line 19, Figure 3.12, then either (a) v is the initial proposal value of the leader of w', proposer $p_{w'}$, or (b) some previous invocation of $choose()$ by $p_{w'}$, in some view $w < w'$, returned v as a candidate value. In case (a), v was obviously proposed was by $p_{w'}$. In case (b), by Lemma 14, some benign acceptor prepared v in view w, and by IH, v was proposed by some proposer. □

Now we prove *Agreement*.

Lemma 15. *After sending a new_view_ack message for view w, a benign acceptor cannot accept value v with view number $w' < w$.*

Proof. By the **upon** condition on lines 31-33, Fig. 3.14, a benign acceptor cannot prepare a value with $w' < w$. Moreover, a benign acceptor a_i updates value v in some view w'' only after a_j prepares v in w''. Hence the lemma.

□

Lemma 16. *If two values v and v' are decided in view w, then $v = v'$.*

Proof. Suppose $v \neq v'$. From Def. 6, all acceptors from some quorum Q (resp., Q') sent $\mathsf{update}_m\langle v, w\rangle$ (resp., $\mathsf{update}_{m'}\langle v', w\rangle$) message, for some $m, m' \in \{1, 2, 3\}$. Hence, all benign acceptors from Q (resp., Q') prepare v (resp., v') in w. By Property 1 of RQS and Definition 5, $Q \cap Q'$ is a basic subset, which contains at least one benign acceptor a_i (by Lemma 12). That is, there exists a benign acceptor that prepared different values in the same view. A contradiction. □

Lemma 17. ***Unique Cand2(v,w).*** *For any $vProof$ that consists of a quorum of valid new_view_ack messages, there are no two different values v and v' such that, in $choose(*, vProof, Q)$, both $Cand_2(v', w)$ and $Cand_2(v, w)$ hold, for the same w.*

Proof. Assume by contradiction that such values v and v' exist. By definition of the predicate $Cand_2()$ (line 2, Fig. 3.12), there exist sets $X = (Q_1 \cap Q) \setminus B$ and $X' = (Q'_1 \cap Q) \setminus B'$, such that (1) $B, B' \in \boldsymbol{B}$, (2) Q_1 and Q'_1 are class 1 quorums, and (3) all acceptors from set X (resp., X') prepared v (resp., v') in w. By Property 2 of RQS, $Q_1 \cap Q'_1 \cap Q$ is a large subset. Applying Definition 5, we conclude that $X \cap X'$ is a non-empty set. Hence, there is an acceptor $a_j \in Q$ such that $vProof[a_j].Prep = v$ and $vProof[a_j].Prep = v'$. Hence, $v = v'$, or $vProof$ is not a set of valid (signed) new_view_ack messages. A contradiction.

□

Lemma 18. ***Cand3(v,w,'a')/Cand4(v,w).*** *Let $vProof$ be a set of valid new_view_ack messages from a quorum Q. Then, if for some value v $Cand_3(v, w, \text{`}a\text{`})$ or $Cand_4(v, w)$ hold in $choose(*, vProof, Q)$, then all benign acceptors from some quorum Q' prepared v in w.*

Proof. Assume first $Cand_3(v, w, `a`)$ holds. Then there is a set $X = (Q_2 \cap Q) \setminus B'$ such that (i) $B' \in \boldsymbol{B}$, (ii) Q_2 is a class 2 quorum and (iii) $P_{3a}(Q_2, Q)$ holds. By Property 3a of RQS, $Q_2 \cap Q$ is a large subset, and, by Definition 5, X is a basic subset. By Lemma 12 there is at least one benign acceptor in X that updated v in w. Therefore, by lines 34-39 in Fig. 3.14, all benign acceptors from some quorum Q' prepared v' in w.

Assume now $Cand_4(v, w)$ holds. Then there exists an acceptor $a_j \in Q$ such that:

- $vProof[a_j].Update[2] = v'$,
- $w \in vProof[a_j].Update_{view}[2]$, and
- $vProof[a_j].Update_{proof}[2, w]$ contains a basic subset of signatures of $\mathsf{update}_2\langle v, w, * \rangle$ messages including at least one from a benign acceptor a_b.

Hence a benign acceptor a_b updated v' in w. Therefore, all benign acceptors from some quorum Q' prepared v' in w. □

Lemma 19. *If w is the lowest view number in which some value v is Decided-2, then no benign acceptor a_i prepares any value v', $v' \neq v$ in any view higher than w.*

Proof. We prove this lemma by induction on view numbers.
Base Step: First, we prove that no benign acceptor a_i can prepare any value different from v in view $w + 1$. A benign acceptor a_i prepares value v' in $w + 1$ only if the *choose*() function on the valid $vProof$ in view $w + 1$ returns v', without setting the *abort* flag. Therefore, it is sufficient to prove that for any valid $vProof$, $choose(*, vProof, Q)$ returns v, or *abort* flag is set.

By Definitions 6 and 7, all benign acceptors from a class 1 quorum Q_1 prepared v in w. Since no basic subset of acceptors is Byzantine, there is a set $B \in \boldsymbol{B}$, such that B contains all Byzantine acceptors. Since the valid $vProof$ for view $w + 1$ consists of new_view_ack messages from a quorum Q, there is a set $X = (Q_1 \cap Q) \setminus B$ that contains only benign acceptors. By Lemma 15, every acceptor $a_i \in X$ prepared v in w, before replying with the new_view_ack message to the leader of view $w + 1$. In the meantime, no acceptor $a_j \in X$ prepared any other value, as this would mean that a_j would be in the higher view then $w + 1$ when replying with new_view_ack for view $w+1$, which is impossible. Therefore, $Cand_2(v, w)$ (line 2, Fig. 3.12) holds in $choose(*, vProof, Q)$, for any Q.

By Lemma 14, it is not difficult to see that there is no value v' such that $Cand_2(v', w')$, $Cand_3(v', w', *)$ or $Cand_4(v', w')$ holds for some $w' > w$.

By Lemma 17, there is no other value $v' \neq v$ such that $Cand_2(v', w)$ holds.

We show now that there is no value v' such that $Cand_3(v', w', `a`)$ or $Cand_4(v', w')$ hold and $v' \neq v$ and $w' = w$. Assume, by contradiction, that such value v' exists. Then, by Lemma 18, all benign acceptors from some quorum Q' prepared v' in w. By Property 2 of RQS, $Y = Q_1 \cap Q'$ is a large subset that contains at least a basic subset of benign acceptors. Therefore some benign acceptor in Y prepared both v and v' in w. A contradiction.

By inspection of *choose*() pseudocode, *choose*() returns v or *abort* flag is set (if there is a $v' \neq v$ such that $Cand_3(v', w, \text{`}b\text{`})$ holds).

Inductive Hypothesis (IH): Assume that no benign acceptor a_i prepares any value different from v in any view from $w+1$ to $w+k$. We prove that no benign acceptor a_i can prepare any value different from v in view $w+k+1$.

Inductive Step: It is sufficient to prove that for any valid $vProof$, $choose(*, vProof, Q)$ returns v, or *abort* flag is set.

By Definitions 6 and 7, all benign acceptors from a class 1 quorum Q_1 prepared v in w. By IH, all benign acceptors from Q_1 can prepare only v in views $w+1$ to $w+k$. Moreover, there is a set $B \in \boldsymbol{B}$, such that B contains all Byzantine acceptors. Since the valid $vProof$ of view $w+k+1$ consists of new_view_ack messages from a quorum Q, there is a set $X = (Q_1 \cap Q) \setminus B$ that contains only benign acceptors. By Lemma 15, no acceptor $a_i \in X$ prepares a value in a higher view than $w+k$ before sending a new_view_ack message to the leader of view $w+k+1$. Hence, by definition of predicate $Cand_2()$, $Cand_2(v, w)$ holds in $choose(*, vProof, Q)$, for any Q.

By Lemma 17, there is no other value $v' \neq v$ such that $Cand_2(v', w)$ holds.

By Lemma 14 and IH, it is not difficult to see that there is no value $v' \neq v$ such that $Cand_2(v', w')$, $Cand_3(v', w', *)$ or $Cand_4(v', w')$ holds for some $w' > w$.

We show now that there is no value v' such that $Cand_3(v', w', \text{`}a\text{`})$ or $Cand_4(v', w')$ hold and $v' \neq v$ and $w' = w$. Assume, by contradiction, that such value v' exists. Then, by Lemma 18, all benign acceptors from some quorum Q' prepared v' in w. By Property 2 of RQS, $Y = Q_1 \cap Q'$ is a large subset that contains at least a basic subset of benign acceptors. Therefore some benign acceptor in Y prepared both v and v' in w. A contradiction.

By inspection of *choose*() pseudocode, *choose*() returns v or *abort* flag is set (in case there is value v', $v' \neq v$, such that $Cand_3(v', w, \text{`}b\text{`})$ holds). □

Similarly to Lemma 19, we prove the following two lemmas using the properties of RQS and induction on view numbers.

Lemma 20. *If w is the lowest view number in which some value v is Decided-3, then no benign acceptor a_i prepares any value v', $v' \neq v$ in any view higher than w.*

Proof. We prove this lemma by induction on view numbers.
Base Step: First, we prove that no benign acceptor a_i can prepare any value different from v in view $w+1$. It is sufficient to prove that for any valid $vProof$, $choose(*, vProof, Q)$ returns v, or *abort* flag is set.

By Definitions 6 and 8, all benign acceptors from a class 2 quorum Q_2 updated-1 (and prepared) v in w. Moreover, there is a set $B \in \boldsymbol{B}$, such that B contains all Byzantine acceptors. Since the valid $vProof$ of view $w+1$ consists of new_view_ack messages from a quorum Q, there is a set $X = (Q_2 \cap Q) \setminus B$ that contains only benign acceptors. By Lemma 15, every acceptor $a_i \in X$ prepared v in w, before replying with the new_view_ack message to the leader of view $w+1$. In the meantime, no acceptor $a_j \in X$ prepared any other value, as this

would mean that a_j would be in the higher view then $w+1$ when replying with new_view_ack for view $w+1$, which is impossible. Therefore, $Cand_3(v, w, *)$ (line 3, Fig. 3.12) holds in $choose(*, vProof, Q)$, for any Q.

By Lemma 14, it is not difficult to see that there is no value v' such that $Cand_2(v', w')$, $Cand_3(v', w', *)$ or $Cand_4(v', w')$ holds for some $w' > w$.

We show now that there is no value v' such that $Cand_3(v', w', `a`)$ or $Cand_4(v', w')$ hold and $v' \neq v$ and $w' = w$. Assume, by contradiction, that such value v' exists. Then, by Lemma 18, all benign acceptors from some quorum Q' prepared v' in w. By Property 1 of RQS, $Y = Q_2 \cap Q'$ is a basic subset that contains at least one benign acceptor. Therefore some benign acceptor in Y prepared both v and v' in w. A contradiction.

We distinguish two cases: (a) $Cand_3(v', w', `a`)$, and (b) $Cand_3(v', w', `b`)$ holds. In case (a) by inspection of *choose*() pseudocode, *choose*() returns v. In case (b) either *choose*() returns v or *abort* flag is set (if there is a $v' \neq v$ such that $Cand_3(v', w, `b`)$ or $Cand_2(v', w)$ hold).

Inductive Hypothesis (IH): Assume that no benign acceptor a_i prepares any value different from v in any view from $w+1$ to $w+k$. We prove that no benign acceptor a_i can prepare any value different from v in view $w+k+1$.

Inductive Step: It is sufficient to prove that for any valid $vProof$, $choose(*, v, Q)$ returns v, or *abort* flag is set.

By Definitions 6 and 8, all benign acceptors from a class 2 quorum Q_2 update-2 v in w. By IH, all benign acceptors from Q_2 can update-2 only v in views $w+1$ to $w+k$. Moreover, there is a set $B \in \boldsymbol{B}$, such that B contains all Byzantine acceptors. Since the valid $vProof$ of view $w+k+1$ consists of new_view_ack messages from a quorum Q, there is a set $X = (Q_2 \cap Q) \setminus B$ that contains only benign acceptors. By Lemma 15, no acceptor $a_i \in X$ prepares (nor 1-updates) a value in a higher view than $w+k$ before sending a new_view_ack message to the leader of view $w+k+1$. Hence, for every $a_i \in X$ $vProof[a_i].Update[1] = v$ and $w \in vProof[a_i].Update_{view}[1]$. Hence, by definition of predicate $Cand_3()$, $Cand_3(v, w)$ holds in $choose(*, vProof, Q)$, for any Q.

By Lemma 17, there is no other value $v' \neq v$ such that $Cand_2(v', w)$ holds.

By Lemma 14 and IH, it is not difficult to see that there is no value $v' \neq v$ such that $Cand_2(v', w')$, $Cand_3(v', w', *)$ or $Cand_4(v', w')$ holds for some $w' > w$.

We show now that there is no value v' such that $Cand_3(v', w', `a`)$ or $Cand_4(v', w')$ hold and $v' \neq v$ and $w' = w$. Assume, by contradiction, that such value v' exists. Then, by Lemma 18, all benign acceptors from some quorum Q' prepared v' in w. By Property 1 of RQS, $Y = Q_2 \cap Q'$ is a basic subset that contains at least one benign acceptor. Therefore some benign acceptor in Y prepared both v and v' in w. A contradiction.

We distinguish two cases: (a) $Cand_3(v', w', `a`)$, and (b) $Cand_3(v', w', `b`)$ holds. In case (a) by inspection of *choose*() pseudocode, *choose*() returns v. In case (b) either *choose*() returns v or *abort* flag is set (if there is a $v' \neq v$ such that $Cand_3(v', w, `b`)$ or $Cand_2(v', w)$ hold). □

Lemma 21. *If w is the lowest view number in which some value v is Decided-4, then no benign acceptor a_i prepares any value v', $v' \neq v$ in any view higher than w.*

Proof. We prove this lemma by induction on view numbers.
Base Step: First, we prove that no benign acceptor a_i can prepare any value different from v in view $w+1$. It is sufficient to prove that for any valid $vProof$, $choose(*, vProof, Q)$ returns v.

By Definitions 6 and 8, all benign acceptors from some quorum Q_3 updated-2 v in w. Moreover, there is a set $B \in \boldsymbol{B}$, such that B contains all Byzantine acceptors. Since the valid $vProof$ of view $w+1$ consists of new_view_ack messages from a quorum Q, by Property 1 of RQS $X = Q_3 \cap Q$ is a basic subset that contains at least one benign acceptor a_i (by Lemma 12). By Lemma 15, a_i updated-2 v in w, before replying with the new_view_ack message to the leader of view $w+1$. In the meantime, a_j did not prepare (nor update-2) any other value, as this would mean that a_j would be in the higher view then $w+1$ when replying with new_view_ack for view $w+1$, which is impossible. Therefore, $Cand_4(v, w)$ (line 4, Fig. 3.12) holds in $choose(*, vProof, Q)$, for any Q.

By Lemma 14, it is not difficult to see that there is no value v' such that $Cand_2(v', w')$, $Cand_3(v', w', *)$ or $Cand_4(v', w')$ holds for some $w' > w$.

We show now that there is no value v' such that $Cand_3(v', w', `a`)$ or $Cand_4(v', w')$ hold and $v' \neq v$ and $w' = w$. Assume, by contradiction, that such value v' exists. Then, by Lemma 18, all benign acceptors from some quorum Q' prepared v' in w. By Property 1 of RQS, $Y = Q_3 \cap Q'$ is a basic subset that contains at least one benign acceptor. Therefore some benign acceptor in Y prepared both v and v' in w. A contradiction.

By inspection of *choose*() pseudocode, *choose*() returns v.

Inductive Hypothesis (IH): Assume that no benign acceptor a_i prepares any value different from v in any view from $w+1$ to $w+k$. We prove that no benign acceptor a_i can prepare any value different from v in view $w+k+1$.

Inductive Step: It is sufficient to prove that for any valid $vProof$, $choose(*, vProof, Q)$ returns v, or *abort* flag is set.

By Definitions 6 and 8, all benign acceptors from some quorum Q_3 updated-2 v in w. By IH, all benign acceptors from Q_2 can prepare (and, hence, update-2) only v in views $w+1$ to $w+k$. Since the valid $vProof$ of view $w+k+1$ consists of new_view_ack messages from a quorum Q, there is a set $X = Q_3 \cap Q$ that contains at least one benign acceptor a_j (by Property 1 of RQS and Lemma 12). Hence, by definition of predicate $Cand_4()$, $Cand_4(v, w')$ holds in $choose(*, vProof, Q)$, for any Q, for some $w', w+k \geq w' \geq w$.

By Lemma 14 and IH, it is not difficult to see that there is no value $v' \neq v$ such that $Cand_2(v', w'')$, $Cand_3(v', w'', *)$ or $Cand_4(v', w'')$ holds for some $w'' > w$.

We show now that there is no value v' such that $Cand_3(v', w, `a`)$ or $Cand_4(v', w)$ hold and $v' \neq v$. Assume, by contradiction, that such value v' exists. Then, by Lemma 18, all benign acceptors from some quorum Q' prepared v' in w. By Property 1 of RQS, $Y = Q_3 \cap Q'$ is a basic subset that contains at least one

benign acceptor. Therefore some benign acceptor in Y prepared both v and v' in w. A contradiction.

By inspection of *choose*() pseudocode, *choose*() returns v. □

Theorem 11. ***(Agreement)*** *No two benign learners learn different values.*

Proof. If a benign learner learns a value then a value was decided in some view (by some benign process). Indeed, a benign learner learns value v by receiving (1) update_* messages (lines 51-53 and 60, Fig. 3.14), or (2) by receiving a basic subset of decision messages (line 101, Fig. 3.14). In case (b), by Lemma 12 at least one benign acceptor decided v before the learner learned v.

It is not difficult to see that, if some value v' is decided in view w, then some benign acceptor prepared v' in w. The theorem follows from Lemmas 16, 19, 20 and 21. □

It is straightforward to show that our algorithm is $(m, \boldsymbol{QC_m})$*–fast*, for $m \in \{1, 2, 3\}$.

The following two lemmas are critical for ensuring *Termination* property.

The first lemma proves that our algorithm does not block in lines 3-8, Fig. 3.14, in case some quorum contains only correct acceptors.

Lemma 22. *If a valid $vProof$ consists only of* new_view_ack *messages sent by a quorum Q that contains only benign acceptors, the abort flag is never set in $choose(*, vProof, Q)$.*

Proof. It is sufficient to prove that if $choose(*, vProof, Q)$ sets *abort* flag, then Q contains at least one Byzantine acceptor. We consider two exhaustive cases.

Case (a): *choose*() aborts in line 15, as there are two values v and $v' \neq v$ such that both $Cand_2(v, w)$ and $Cand_3(v', w, \text{`}b\text{`})$ hold (for $w = view_{max}$). In this case, by definition of predicate $Cand_3()$ (line 3, Fig. 3.12) there is an acceptor $a_j \in Q$ and a class 2 quorum Q_2 such that: (1) $P_{3b}(Q_2, Q)$, and (2) a_j claims that all (benign) acceptors from Q_2 prepared v' in w. Moreover, by definition of predicate $Cand_2()$ (line 2, Fig. 3.12), all acceptors from some set $X = (Q_1 \cap Q) \setminus B)$ (where Q_1 is a class 1 quorum and $B \in \boldsymbol{B}$) claim that they prepared v in w. By Property 3b of RQS, $Q_1 \cap Q \cap Q_2$ is a basic subset. Hence, $Q_2 \cap X$ is non-empty. Let a_i be in $Q_2 \cap X$. Then a_j claims that a_i prepared v' in w, while a_i claims that it prepared $v \neq v'$ in w. Hence, at least one acceptor from the set $\{a_j, a_i\} \subset Q$ is faulty.

Case (b): *choose*() aborts in line 17, as there are two values v and $v' \neq v$ such that both $Cand_3(v, w, \text{`}b\text{`})$ and $Cand_3(v', w, \text{`}b\text{`})$ hold (for $w = view_{max}$). In this case, by definition of predicate $Cand_3()$ (line 3, Fig. 3.12) there are acceptors $a_i, a_j \in Q$ and class 2 quorums Q_2 and Q_2' such that: (1) $P_{3b}(Q_2, Q)$, (2) a_i claims that all (benign) acceptors from Q_2 prepared v in w, (3) $P_{3b}(Q_2', Q)$, (2) a_j claims that all (benign) acceptors from Q_2' prepared v' in w. By Property 1 of RQS $Q_2 \cap Q_2'$ is a basic subset, that by Lemma 12 contains at least one benign acceptor a_x. Hence, a_i claims that a benign acceptor a_x prepared v in w, while

a_j claims that a_x prepared $v' \neq v$ in w. Hence, at least one acceptor from the set $\{a_j, a_i\} \subset Q$ is faulty. □

The second lemma proves that our algorithm does not block in lines 23-27, Fig. 3.14, in case some quorum contains only correct acceptors. In the following proof, we explicitly make use of the assumption of an eventually synchronous system, i.e., of an existence of a global stabilization time GST after which the system is synchronous.

Lemma 23. ***(Availability of signatures.)*** *If a correct acceptor a_j issues a $\mathsf{sign\_req}\langle Update[step], w, step\rangle$ message (line 24, Fig. 3.14) after GST, then a_j eventually receives signed $\mathsf{update}_{step}\langle Update[step], w, *\rangle$ messages from some basic subset of acceptors.*

Proof. A correct acceptor a_j issues a $\mathsf{sign\_req}\langle Update[step], w, step\rangle$, only if (at a_j) $w \in Update_{view}[step]$, i.e., only if a_j updated value $v = Update[step]$ in w. In other words, before issuing a sign_req message, a_j received $\mathsf{update}_{step}\langle v, w, *\rangle$ messages from some quorum Q and executed lines 34-38, Fig. 3.14. In particular a_j adds the identifier of the quorum Q to the $Update_{QRM}[step, w]$ set (line 37, Fig. 3.14). Without loss of generality, we can assume that a_j sent a $\mathsf{sign\_req}\langle v, w, step\rangle$ message to acceptors from quorum Q.

Let Q_c be the quorum that contains only correct acceptor. By Property 1 of RQS, $T = Q \cap Q_c$ is a basic subset. Since $T \subseteq Q_c$, T contains only correct acceptors. Since after GST the system is synchronous, a_j eventually receives the desired set of signatures. □

We also need the following two simple lemmas. In the remainder of the Chapter, we denote by Q_c the quorum that contains only correct acceptors.

Lemma 24. *If some process receives decision messages with the same value v from some quorum of acceptors Q, then every correct learner learns a value.*

Proof. Suppose, by contradiction, that some correct learner l_k never learns a value.

By Property 1 of RQS and assumption on Q_c, $T = Q_c \cup Q$ is a basic subset of correct acceptors that decided value v. Denote by t the time after which all acceptors from T have decided v. By lines 102-103, Fig. 3.14, and our assumption that l_k never learns the value, l_k sends an infinite number of decision_pull messages to all acceptors. Those messages sent after $max(t, GST)$ are received by all acceptors from T who send decision messages to l_k. These messages are received by l_k and, by line 101 Fig. 3.14, l_k learns v — a contradiction. □

Lemma 25. *Every message m sent by a correct process p_1 at time $t \geq GST$ to another correct process p_2 is received by p_2 at latest by $t + \Delta$. Moreover, if p_2 sends some messages in the same step upon receiving m, it will do so also by $t + \Delta$.*

Proof. Since we assume that the system is synchronous after GST and that, when the system is synchronous every message sent between two correct processes is received in at most one message delay Δ, p_2 receives m by $t' = t + \Delta$. Moreover,

since we assume that taking a step takes negligible time when the system is synchronous, if p_2 sends messages upon reception of m it will do so at latest by $t + \Delta$. □

Towards proving *Termination*, we first show that our algorithm (or, more precisely, its *Locking* module) satisfies a weaker property we call *Eventual Obstruction-Free Termination* (EOFT), defined as follows.

Definition 10. ***(Eventual Obstruction-Free Termination.)*** *Assume a correct proposer p_k proposes a value at time t_p, after GST ($t_p > GST$) with view number $view_{high}$ such that: (a) p_k is the leader of $view_{high}$, (b) p_k has a valid $viewProof$ for $view_{high}$, (c) no value with a view number higher than $view_{high}$ is proposed up to time t, and (d) no proposer proposes a value (with valid $viewProof$) for the view higher than $view_{high}$ by $t_p + D_{OF}$, where $D_{OF} = 7\Delta$. Then, every correct learner eventually learns a value.*

Lemma 26. ***(EOFT.)*** *The* Locking *module of our consensus algorithm satisfies Eventual Obstruction-Free Termination property.*

Proof. By our assumption of an eventually synchronous system, all acceptors from Q_c receive the new_view message for $view_{high}$ sent by p_k. Moreover, by assumptions (a)-(d) of Definition 10 we conclude that the condition in line 21 (Fig. 3.14) is satisfied for every acceptor from Q_c, which then proceeds to execute lines 22-28. By Lemma 23, this part of the code is non-blocking. In case some acceptor from Q_c sends some sign_req message (line 24), it will do so by $t_p + \Delta$ (by Lemma 25, and similarly send sign_ack messages (line 29) by $t_p + 2\Delta$. Hence, all acceptors from Q_c execute line 28 of Fig. 3.14 and send the new_view_ack messages to p_k by $t_p + 3\Delta$ (Lemma 25). Denote the set of these new_view_ack messages sent by quorum Q_c (and received by proposer p_k) by $vProof$. By Lemma 22, the $choose(*, vProof, Q_c)$ does not abort, but rather returns some value v. Therefore, at latest by $t_p + 4\Delta$ (Lemma 25), p_k sends the $\mathsf{prepare}\langle v, view_{high}, vProof, Q_c\rangle$ message to all acceptors — hence, all acceptors from Q_c receive this message. By assumptions (a) and (d) of Definition 10 and Lemma 25, we conclude that the condition in line 31, Fig. 3.14 is satisfied and that all acceptors from Q_c prepare v in $view_{high}$ and send the $\mathsf{update}_1\langle v, view_{high}, \emptyset\rangle$ message to all acceptors (and learners) by $t_p + 5\Delta$. By assumption (d) of Definition 10 and Lemma 25, given that all acceptors from Q_c prepare v in $view_{high}$, we conclude that all acceptors from Q_c send an $\mathsf{update}_2\langle v, view_{high}, Q_c\rangle$ (by $t_p + 6\Delta$), and, later, an $\mathsf{update}_3\langle v, view_{high}, Q_c\rangle$ (by $t_p + 7\Delta = t_p + D_{OF}$) to every acceptor and learner. As soon as a correct learner receives a $\mathsf{update}_3\langle v, view_{high}, Q_c\rangle$ message from every acceptor of Q_c it learns a value (lines 53 and 101, fig. 3.14), unless it already learned a value. □

We are now ready to prove *Termination*. Basically, what is left to show is that, in every execution in which a correct proposer proposes a value, eventually, the *Election* module ensures that the assumptions of Definition 10 eventually hold.

Theorem 12. *(Termination) If a correct proposer proposes a value, then eventually, every correct learner learns a value.*

Proof. Suppose, by contradiction, that some correct learner l_k never learns a value even if some correct proposer, say p_k, proposes a value.

Note that, if p_k proposes a value, by (1) lines 0 and 101-103, Fig. 3.13 and (2) the assumption of an eventually synchronous system, either (a) all correct acceptors eventually trigger their *suspectTimeout* (line 0, Fig. 3.13), or (b) p_k receives a decision message from some quorum Q of acceptors and halts (line 104, Fig. 3.13). In the latter case (i.e., case (b)), by Lemma 24, every correct learner eventually learns a value — a contradiction.

We now focus on the case (a), where all correct acceptors eventually trigger *suspectTimeout* (line 0, Fig. 3.13) – we denote this time by $t_{trigger}$. Let $GST' = max(GST, t_{trigger})$.

We distinguish two sub-cases: (i) when no correct acceptor stops its timer *suspectTimeout* permanently (i.e., no correct acceptor executes the line 7, Fig. 3.13), and (ii) when some correct acceptor stops its *suspectTimeout* permanently.

We first consider case (i). We define functions (t representing time):

- $view_{min}(t) = min\{nextView_{a_i} | a_i \in Q_c\}$, and
- $view_{max}(t) = max\{nextView_{a_i} | a_i \in Q_c\}$.

It is not difficult to see (lines 1-5, Fig. 3.13), that, in case (i), at every correct acceptor a_j, variable $nextView_{a_j}$ is: (1) monotonically increasing, (2) unbounded, and (3) non-skipping (it always increments by one). Hence, every correct acceptor sends an infinite number of view_change messages, for every view number from 1 (i.e., $initView + 1$) to ∞. Moreover, functions $view_{min}(t)$ and $view_{max}(t)$ are also monotonically increasing and unbounded.

Let $viewGST' = view_{max}(GST')$. Hence, every correct proposer receives all view_change messages sent by acceptors from Q_c for view numbers $viewGST' + 1$ and higher. Therefore, every correct proposer p_k proposes a value in every view $w_k \geq viewGST' + 1$, such that $k = (w_k \bmod |proposers|)$.

Note that a correct acceptor a_i, on sending a view_change message with view number w, at some time t_w, triggers a timer equal to $initTimeout * 2^w$ (lines 1-5, Fig. 3.13). After expiration of this timeout, a_i sends the subsequent view_change message. Hence, the time between a_i sends the view_change messages for view numbers w and $w + 1$ is at most $initTimeout * 2^w$.

Let t be any point time in time after GST'. Let $view(t)$ be the first view in which p_k proposes a value, such that $view(t) > view_{max}(t) + 1$. Note that no acceptor from Q_c sends a view_change message for a view higher than $view(t)$ before $T_{OF}(t) = t + initTimeout * 2^{view(t)}$. By Property 1 of RQS and the proposer code of an *Election* module, we conclude that no proposer can propose a value with valid $viewProof$ and the view number higher than $view(t)$ before $T_{OF}(t)$.

On the other hand, all acceptors from the quorum Q_c will send the view_change message for the $view(t)$ at latest by $T_{vc}(t) = t + InitTimeout * (2^{view_{min}(t)} + 2^{view_{min}(t)+1} + \ldots + 2^{view(t)-1})$. These will be received by p_k, which will propose a value with view number $view(t)$ at latest by $T_{prop}(t) = T_{vc}(t) + \Delta$ (Lemma 25).

Therefore, p_k proposes a value with $view(t)$ at $T_{prop}(t) > GST$, and (a) p_k is the leader of $view(t)$, (b) p_k has a valid $viewProof$ for $view(t)$ (view_change messages from Q_c), (c) no value with a view number higher than $view(t)$ is

proposed up to time $T_{prop}(t)$, and (d) no proposer proposes a value (with a valid $viewProof$) for the view higher than $view(t)$ by $T_{OF}(t)$. Hence, in order to apply Lemma 26 and reach contradiction, we need to show that there exist t', such that $T_{OF}(t') - T_{prop}(t') > D_{OF}$ (where $D_{OF} = 7\Delta$).

Since $T_{OF}(t') - T_{prop}(t') = initTimeout * (2^{view(t')} - (2^{view_{min}(t')} + 2^{view_{min}(t')+1} + \ldots + 2^{view(t')-1})) - (\Delta)$, $T_{OF}(t') - T_{prop}(t') > D_{OF} \Leftrightarrow 2^{view_{min}(t')-1} > (D_{OF} + \Delta)/initTimeout = c$, where c is a constant. Since $view_{min}(t)$ is monotonically increasing and unbounded, such t' exists.

In case (ii) the contradiction follows directly from Lemma 24. □

4

Optimizing Worst Case Latency Using High-Resolution Timestamps

4.1 Introduction

In Chapter 3 we discussed the best case optimal time complexity of asynchronous Byzantine fault-tolerant (BFT) storage and consensus algorithms. To complete the picture, it is natural to ask what is the worst-case time complexity of such BFT algorithms. As we already mentioned, the famous result of Fischer, Lynch and Patterson [FLP85] established the impossibility of fault-tolerant asynchronous consensus even if failures are benign. Therefore, we focus on worst-case time complexity of (wait-free) BFT distributed storage.

In fact, the complexity[1] of *writing* into distributed storage has actually been carefully studied. Consider distributed storage implemented over a set of S base objects out of which t of might fail, and b ($b > 0$) of these failures may be Byzantine (i.e., we consider here a threshold adversary as defined in Chapter 3). Then, the tight lower bound on the worst-case complexity of the write operation was shown to be 2 rounds when at most $2t + 2b$ of these objects are used; if more than $2t + 2b$ base objects are available, then a single round suffices [ACKM06][2]. This lower bound is general, since it was established for any safe storage. In fact, it was also shown in [ACKM06] that this bound is tight, even for stronger, *regular* storage.

On the other hand, no such general picture for a *read* operation exists. Actually, the complexity of reading was studied, but only in some specific cases. For instance, it was shown that, for any safe storage, when readers do not modify the state of the base objects, the optimal read complexity with less than $2t + 2b$ base objects is $b + 1$ rounds [ACKM06].

But what is the general complexity of a read operation? In this and the following chapter we take on this issue and address this fundamental question. In this chapter, we focus on optimally resilient storage implementations (i.e., that toler-

[1] In this chapter, unless explicitly stated otherwise, under the notion of complexity we refer to the worst-case time complexity.

[2] [ACKM06] does not distinguish between crash and Byzantine failures, i.e., assumes $b = t$. Still, it is not difficult to extend its results to the general $b \neq t$ case.

ate as many base object failures as possible). In the following chapter, we allow for sub-optimally resilient implementations. More specifically, in this chapter:

- We prove a 2-round lower bound for reading from *safe* storage that uses at most $2t + 2b$ base objects, independently of the number or rounds needed by the writer.
- We then prove the lower bound tight even for *regular* storage that is optimally resilient and uses $2t + b + 1$ base objects (this bound can trivially be obtained from [MAD02a], which proves the bound for $b = t$). Our storage algorithm combines these desirable goals by relying on a novel timestamping mechanism we call *high resolution timestamps*. Moreover, our regular storage algorithm (as well as our lower bound) assumes the unauthenticated Byzantine failure model. It is worth noting that, in the authenticated model, (SWMR) regular storage can be implemented fairly simply, while achieving both optimal resilience and fast (i.e., single round-trip) reads/writes [MR98].
- Using the transformation from regular to atomic storage [GR06] and our results for regular storage, we narrow down the possible range for a lower bound on reading from optimally resilient *atomic* storage in the unauthenticated model to between 2 and 4 rounds (inclusive of two boundary values). The exact minimum complexity of atomic storage remains an open problem.

In this chapter, we proceed through three major steps.

1. We first prove in Section 4.2 that $S = 2t + 2b$ base objects are insufficient for a safe SWSR wait-free storage implementation in the unauthenticated Byzantine failure model (see Chapter 2 for details) in which every read is fast (i.e., completes in a single round-trip). Our proof applies to the unauthenticated model in which any client may fail by crashing. Roughly, our proof derives a contradiction from three executions that are indistinguishable to the reader. In the first execution, a read is concurrent with the write and all base objects are correct; in the second one, the write precedes the read but Byzantine base objects forge their state to simulate the concurrency of the first execution; finally, in the third execution, Byzantine base objects forge their state to simulate the above mentioned concurrency, although the write is never invoked. As the read must return the same value in all three executions, without invoking additional communication round-trips, safety is violated in either the second, or the third execution. In our proof, we do not make any assumption on the time complexity of the write operation.
2. We then describe in Section 4.3 an optimally resilient SWMR safe storage algorithm that features optimal (worst-case) time complexity for both read and write operations: 2 rounds. This algorithm is interesting in its own right as it contradicts the conjecture of [ACKM06] suggesting that $b + 1$ rounds are needed in order to read from safe storage. The algorithm uses

novel techniques to combine optimal resilience with optimal time complexity. Roughly, unlike in traditional safe storages we know of, in both of their communication rounds, readers both change the state of base objects and read their current state. The writer does the same in its first round, along with simply writing in the second round. Basically, by allowing readers to change the state of the base objects, twice in a row, we allow the readers to carefully filter the responses from Byzantine base objects that may be trying to mislead the reader.

In fact, in our safe storage algorithm every reader writes its own local timestamps to base objects. These timestamps are then combined by the writer into what we call a *high resolution timestamp* $HRts$, a matrix of readers's local timestamps (in which rows correspond to base objects, columns correspond to readers and where $HRts[i][j]$ reflects the latest copy of the reader r_j's local timestamp that the base object s_i reported to the writer). High resolution timestamps are ultimately used by readers in detecting and filtering out the responses from Byzantine base objects.

High resolution timestamps can be seen as a variation of matrix clocks [WB84, RS96]; the differences are in that base objects do not maintain their own local timestamps/clocks and in that a reader needs not to keep track of any other timestamp except its own — as a result, high-resolution timestamp is a rectangular, rather than a square matrix. While vector [Fid91, Mat89] and matrix clocks have been previously used to cope with Byzantine failures (see e.g., [MS02, CSS07]), to our knowledge, no variation of matrix clocks has been used in achieving optimally resilient or latency optimal Byzantine fault tolerant algorithms prior to this work.

Our safe storage implementation (and our regular storage implementation as well) assumes an unauthenticated model and tolerates writer's crash failure and any number of Byzantine failures of readers (this is to be contrasted with our lower bound of Section 4.2, which assumes that readers are crash-prone, not Byzantine).

3. Finally, we show in Section 4.4 how to modify our safe implementation and obtain a regular one without sacrificing neither optimal resilience nor optimal time complexity. Our regular implementation relies however on the fact that base objects keep all the values they receive from the writer (which is not the case with our safe implementation). Although some very practical storage systems rely on the same assumption [GWGR04], this might raise issues of storage exhaustion and needs careful garbage collection. A recent comparable regular wait-free storage implementation that does not rely on this assumption [ACKM07] is not optimally resilient.

To conclude the chapter, we make use of the above three steps and discuss the possible range of the worst-case time complexity of optimally resilient atomic wait-free storage (Section 4.5), as well as the applicability of our results to the server-centric model in which base objects may communicate among themselves (Section 4.6).

4.2 Lower Bound

We prove in this section that there is no safe storage implementation with at most $2t + 2b$ objects in which every read is *fast*. In our proof, we assume that a set of readers is a singleton, thus broadening the scope of our lower bound.

Proposition 1. There is no fast read implementation I of a single reader (SWSR) safe storage that makes use of less than $2t + 2b + 1$ objects.

Preliminaries. Recall first that w denotes the writer, r_1 the reader, and s_i for $1 \leq i \leq S$ denote the objects. Suppose, by contradiction, that there is a safe storage implementation I that uses at most $2t + 2b$ objects, such that, in every (partial) execution of I every read operation completes in a single round (i.e., every read is *fast*).

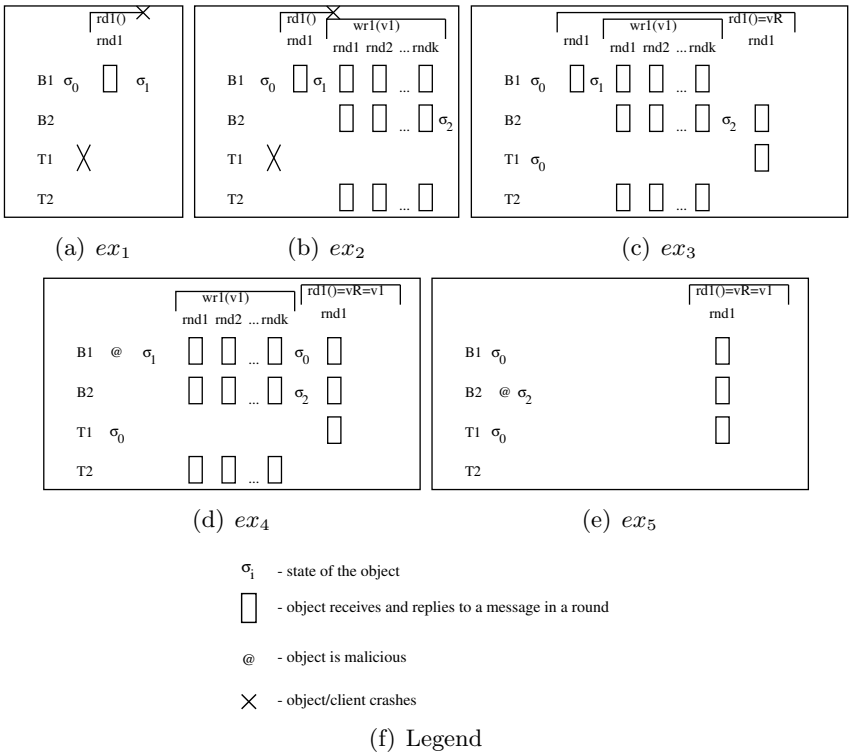
Notice that, in our model (Chapter 2), for a fast read implementation, we can say without ambiguity that the messages sent by a reader, on invoking a read, are of type READ, and the messages sent by a base object to the reader, on receiving a READ message, of type READACK.

We partition the set of objects into four distinct subsets (which we call *blocks*), denoted by T_1 and T_2, each of size exactly t, and B_1 and B_2 of size at least 1 and at most b. Note that we assume $S \geq 2t + 2$, without loss of generality since the number of objects for any implementation I must conform with the optimal resilience lower bound of $S \geq 2t+b+1$ [MAD02a] (recall that we assume $b > 0$). Therefore, without loss of generality, we can assume that each of the blocks T_1, T_2, B_1 and B_2 contains at least one object. We refer to the initial state of every correct object as σ_0.

We say that a message m of a round *rnd* of an incomplete operation *op skips* a set of blocks BS in a partial execution (where $BS \subseteq \{T_1, T_2, B_1, B_2\}$), if (1) no object in any block $BL \in BS$ receives m in round *rnd* of *op* in that partial execution, (2) all other objects receive m in round *rnd* of *op* and reply to that message, and (3) *all these reply messages are in transit*. We say that a *complete operation op skips* a set of blocks BS in a partial execution, if (1) no object in any block $BL \in BS$ receives any message in any round of *op* in that partial execution, (2) all objects that are not in any block $BL \in BS$ receive the message from the invoking client in every round of *op* and reply to such message, and (3) the invoking client *receives all these reply messages* and, finally, returns from the invocation.

We illustrate the idea behind the proof in Figure 4.1. We depict a round *rnd* of an operation *op* through a set of rectangles, arranged in a single column. In the column corresponding to some round *rnd* of an operation *op*, we draw a rectangle in the particular row, if all objects in the corresponding block BL have received the message from the client in round *rnd* of *op* and have sent reply messages, i.e., we draw a rectangle in the row corresponding to BL if round *rnd* of *op* does not skip BL.

Proof. To exhibit a contradiction, we construct a partial execution of the safe



(a) ex_1 (b) ex_2 (c) ex_3

(d) ex_4 (e) ex_5

(f) Legend

Figure 4.1: Illustration of the executions used in the proof of Proposition 1

implementation I that violates safety. More specifically, we exhibit a partial execution in which some read returns a value that was never written.

- Let ex_1 be the partial execution in which all objects are correct except T_1 that crashes at the beginning of ex_1. Furthermore, let rd_1 be the read operation by the reader (r_1) and no other operation is invoked in ex_1. In ex_1, r_1 crashes and rd_1 skips B_2, T_1 and T_2. After B_1 sends READACK to r_1, ex_1 ends. We refer to the state of object B_1, at the end of ex_1 as to σ_1.

- Let ex_2 extend ex_1 by appending write wr_1 invoked by the correct writer to write value $v_1 \neq \perp$ in the storage. Since I is wait-free, wr_1 completes in ex_2, say at time t_1 after invoking a finite number (k) of rounds. Therefore, wr_1 skips T_1, and completes (at latest) after the writer receives the replies in round k from correct objects (B_1, B_2, and T_2). We refer to the state of the correct object B_2 at time t_1 as to σ_2.

- Let ex_2' be the partial execution that ends at t_1, such that ex_2' is identical to ex_2 up to time t_1, except that in ex_2' object T_1 does not crash, but, due to asynchrony, all messages sent by the writer to T_1 during wr_1 remain in transit. Since the writer cannot distinguish ex_2 from ex_2', wr_1 skips T_1 and completes in ex_2' at t_1.

- Let ex_2'' be the partial execution identical to ex_2' up to time t_1, except that, in ex_2'', (1) the reader does not crash in ex_2'', but, due to asynchrony, all

messages that were in transit in ex'_2 are delayed in ex''_2 until after t_1, and (2) object T_2 crashes at t_1. By our assumption on the wait-freedom of I, rd_1 completes in ex''_2 at t_2 after receiving READACK messages from correct objects (B_1, B_2 and T_1) and returns some value v_R, skipping T_2.

- Let ex_3 be the partial execution identical to ex''_2, except that, in ex_3, T_2 does not crash, but, due to asynchrony, all messages exchanged between r_1 and T_2 during rd_1 are delayed until after t_2. Since r_1 cannot distinguish ex_3 from ex''_2, rd_1 completes in ex_3 at t_2 and returns v_R. Note that in ex_3 all objects are correct.

- Let ex_4 be the partial execution similar to ex_3, except that, in ex_4: (1) rd_1 is invoked only after wr_1 completes (after t_1) (2) B_1 is Byzantine and forges it state to σ_1 at the beginning of the execution (as if it received a round 1 message of rd_1 from the reader, as in ex_3), before wr_1 is invoked, (3) after t_1, a read rd_1 is invoked and (4) at t_1, B_1, before replying to rd_1, forges its state to σ_0, the initial state of correct objects. Other messages are delivered as in ex_3, in particular, messages exchanged between r_1 and T_1 are transit in ex_4. Note that wr_1 cannot distinguish ex_4 from ex_3 and hence, wr_1 completes in ex_4 at t_1. Note also that, rd_1 is invoked after wr_1 completes, so safety implies that rd_1 must return v_1. However, note that in ex_3 and ex_4 the reader receives in rd_1 the identical messages and, since the processes do not have access to global clock, r_1 (as well as the correct objects B_2, T_1 and T_2) cannot distinguish ex_4 from ex_3. Therefore, in ex_3 and ex_4 rd_1 returns the same value, i.e., v_R, that, by safety, must equal v_1.

- Finally, consider the partial execution ex_5 in which wr_1 is never invoked, but B_2 is Byzantine and forges its state to σ_2 at the beginning of the execution. Read rd_1 is invoked in ex_5 as in ex_4. Since, upon receiving READACK messages from B_1, B_2 and T_1, the reader receives identical information as in ex_4, the reader cannot distinguish ex_4 from ex_5 (neither can correct objects B_1, T_1 and T_2), and rd_1 completes in ex_5 and returns a $v_R = v_1$. However, by safety, in ex_5, rd_1 must return $\perp$. Since $v_1 \neq \perp$, safety is violated in ex_5. □

4.3 Safe Implementation

Our algorithm uses $S = 2t + b + 1$ objects (optimal resilience) to implement a SWMR safe storage. Besides its optimal resilience, our implementation features optimal (worst-case) time complexity for both read and write operations, i.e., two communication round-trips. In fact, the existence of our algorithm proves the following proposition:

Proposition 2. There is an optimally resilient implementation I of a SWMR safe storage such that, in every partial execution of I, every (read/write) operation completes in at most two communication round-trips.

Finally, our safe storage implementation tolerates Byzantine failures of any number of readers and crash-failures of the writer.

In the following, we first give a detailed description of our algorithm, and then proceed by proving its correctness.

4.3.1 Overview

In our algorithm, both read and write operations take at most two rounds. In each round, the client (reader or writer) sends a message to all objects. A round terminates at the latest when the client receives the responses from $S - t$ correct objects. In the first round, the writer, in addition to writing data, reads control data from the objects. Readers write control data and read data written by the writer in both rounds.

Base objects maintain the following variables (we call *fields*) pw, w and the array $tsr[1, ..., R]$ (where R is the number of readers). Fields pw and w are written by the writer, whereas each field $tsr[j]$ is written by reader r_j.

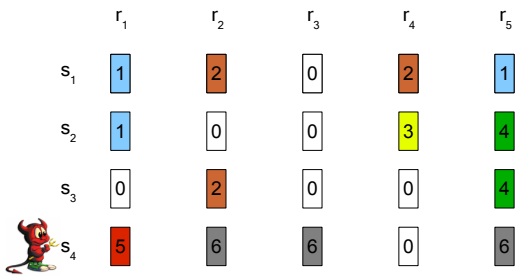
In each of the two rounds of the read, the reader r_j: (1) increases its local timestamp tsr'_j and stores it in the objects' $tsr[j]$ field[3] and (2) reads the objects' fields pw and w.

In the first round of the write (denoted by PW, for "pre-write"), the writer, writing the value v: (1) increases its local timestamp ts, (2) assigns the timestamp-value pair $\langle ts, v\rangle$ to its variable pw', (3) writes pw' to the objects' pw fields and the last copy of its variable w' (which is modified only in the second round of write) to the objects' w fields, (4) reads the values of objects' fields $tsr[*]$ that are written by readers and (5) combines the read arrays $tsr[*]$ into, what we call, a *high-resolution timestamp* ($HRts$). In effect, a high resolution timestamp is an array of arrays of readers' local timestamps, provided to the writer by base objects, that plays the crucial role in achieving the optimal worst-case time complexity of read operation (high-resolution timestamps are depicted in Figure 4.2). Finally, upon receiving $S - t$ responses from different objects in round PW, the writer proceeds to the second round.

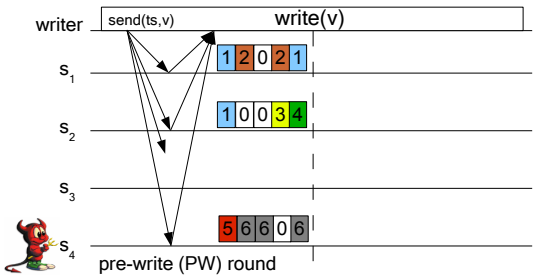
In the second round of the write (denoted by W), the writer: (1) assigns $w' := \langle pw', HRts\rangle$ and (2) writes pw' to objects' pw fields and w' to objects' w fields. Effectively, the writer combines its traditional local timestamp ts with a high resolution timestamp $HRts$ and writes them along with the corresponding value into base objects' w field. Upon receiving $S - t$ responses from different objects in round W, the write completes. As in many previous storage implementations based on timestamps, a base object changes the values of $tsr[*]$, pw, and w only if it receives a newer copy than the one already stored (Figure 4.4).

The write implementation is given in Figure 4.3. In the following, we detail the read implementation, which is somewhat more involved.

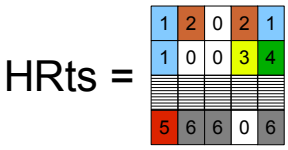
[3]Notice that, in array tsr, base objects simply store the latest timestamps received from readers, without maintaining their own timestamp, which is to be contrasted with vector clocks [Fid91, Mat89].



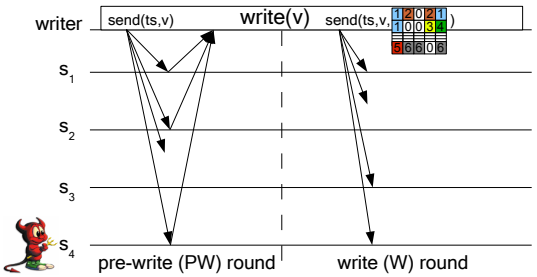
(a) In each access to a server, a reader stores to server a copy of its local timestamp. Byzantine base objects (e.g., s_4) may modify the readers' timestamps since these are not signed.



(b) In the first round of a write the writer stores into base objects value v along with the writer's own local timestamp ts. At the same time, the writer reads the copies of readers' timestamps from base objects.



(c) The writer combines the copies of readers' timestamps into a high resolution timestamp ($HRts$). Some rows in $HRts$ may be empty (e.g., the third row in the above figure) since the writer can wait only for $S - t$ base objects to respond.



(d) In the second round, the writer writes ts, v and $HRts$ to a (possibly different) set of $S-t$ base objects. High resolution timestamps are ultimately used by readers to detect Byzantine base objects.

Figure 4.2: High Resolution Timestamps: an example with 4 base objects ($t = b = 1$) and 5 readers

Initialization:
1: $initHRts[i][j] := nil$, $1 \leq i \leq S$, $1 \leq j \leq R$
2: $pw := \langle 0, \perp \rangle$; $ts := 0$; $w := \langle pw, initHRts \rangle$

write(v) **is** {
3: **inc**(ts); $HRts := initHRts$
4: $pw := \langle ts, v \rangle$
5: send PW$\langle ts, pw, w \rangle$ to all objects
6: **wait for** PW_ACK$\langle ts, tsr \rangle$ from $S - t$ different objects
7: $w := \langle pw, HRts \rangle$
8: send W$\langle ts, pw, w \rangle$ message to all objects
9: **wait for** W_ACK$\langle ts \rangle$ from $S - t$ different objects
10: **return**(OK)

upon reception of PW_ACK$\langle ts, tsr \rangle$ from s_i
11: $HRts[i] := tsr$
}

Figure 4.3: SWMR safe storage: write implementation - code of the writer

Initialization:
1: $ts := 0$; $initHRts[i][j] := nil$, $1 \leq i \leq S$, $1 \leq j \leq R$
2: $pw := \langle 0, \perp \rangle$; $w := \langle pw, initHRts \rangle$; $tsr[j] := 0$, $1 \leq j \leq R$

3: **upon** reception of PW$\langle ts', pw', w' \rangle$ message from the writer **do**
4: **if** $ts' > ts$ **then**
5: $ts := ts'$; $pw := pw'$; $w := w'$
6: send PW_ACK$\langle ts, tsr \rangle$ to the writer
7: **endif**

8: **upon** reception of W$\langle ts', pw', w' \rangle$ message from the writer **do**
9: **if** $ts' \geq ts$ **then**
10: $ts := ts'$; $pw := pw'$; $w := w'$
11: send W_ACK$\langle ts \rangle$ to the writer
12: **endif**

13: **upon** reception of READ$_k\langle tsr' \rangle$ mess. from r_j ($k \in \{1, 2\}$) **do**
14: **if** $tsr' > tsr[j]$ **then**
15: $tsr[j] := tsr'$
16: send READ$_k$_ACK$\langle tsr[j], pw, w \rangle$ to the reader r_j
17: **endif**

Figure 4.4: SWMR safe storage: code of object s_i

4.3.2 Read implementation

The full read implementation is given in Figure 4.5. In the following, unless explicitly stated otherwise, we refer to Figure 4.5.

As we previously mentioned, in both rounds of the read, the reader: (1) increases its local timestamp tsr'_j (lines 9 and 12), and stores it in the objects' $tsr[j]$ fields using READ1 (in the first round), or READ2 (in the second round) messages (lines 10 and 13) and (2) reads the objects' fields pw and w by receiving READ1_ACK, or READ2_ACK messages (lines 11, 14 and 21-26).

When the reader receives a timestamp-value pair pw' from the pw field of object s_i (we say s_i *reports* pw'), the reader adds i, the index of object s_i, to the set $RPW(pw')$ that is initially empty. Similarly, if s_i reports tuple w' in its w field, the reader adds i to the set $RW(w')$. If this occurs in the first round of the read, the reader also adds i to $FirstRW(w')$. (lines 22, 23 and 26)

Every tuple c reported by some object in its w field in the first round of the read, is added by the reader to the set of *candidate values*, the set C (line 24). A candidate value c is automatically removed from C if at least $t + b + 1$ objects respond (in any round of the read) without c in their w field (lines 2 and 27-28).

In the first round, reader r_j awaits responses from a set that contains at least $S - t = t + b + 1$ objects such that there is no *conflict* between any 2 objects s_i and s_k that belong to this set (set $Resp1OK$, line 11). A conflict between two objects is detected using high-resolution timestamps. A conflict arises when one object, say s_k, reports in its w field a candidate value c, such that $c.HRts[i][j] > tsrFR$ (line 4), where $tsrFR$ is the timestamp of reader r_j in the first round of read (line 9). In other words, object s_k claims that the object s_i reported to the writer a timestamp of reader r_j higher than any timestamp that r_j has issued so far. Intuitively, in this case, at least one of the objects s_k or s_i is Byzantine. Hence, in a set that contains only correct objects, there is no conflict between any two objects. As there are at least $S - t$ correct objects, hence the intuition on why the first round of read eventually completes (i.e., why the condition in line 11 eventually holds).

At the beginning of the second round of the read, reader r_j increments its local timestamp tsr'_j once more (line 12) and sends READ2$\langle tsr'_j \rangle$ to all objects (line 13). Then the reader waits for the responses from objects until there is a candidate value c with the highest (traditional) timestamp in C (i.e., $highCand(c)$ holds, line 4), such that $safe(c)$ holds or until C is empty (this can occur only if the read is concurrent with some write). Predicate $safe(c)$ holds if at least $b + 1$ different objects have responded either in their w (or pw) fields with c (or $c.tsval$ for pw), or with a value with a higher timestamp (line 3).

Our implementation guarantees that the condition in line 14 is eventually satisfied in every read. In the following, we give a rough intuition behind this guarantee. This is followed by the detailed proof of algorithm correctness (Section 4.3.3).

Assume, by contradiction, that there is read rd by some reader r_j (in execution ex) such that rd never completes, i.e., there is candidate value c in rd, such that c is never eliminated from C and c is never *safe*. Consider the following three cases.

Definitions:
1: $conflict(i,k) ::= \exists c \in C : ((k \in FirstRW(c)) \wedge (c.HRts[i][j] > tsrFR))$
2: $RespondedWO(c) := \{i : \exists c' \neq c, i \in RW(c')\}$
3: $safe(c) ::= |RW(c) \cup RPW(c.tsval) \bigcup_{c'.tsval.ts > c.tsval.ts} (RW(c') \cup RPW(c'.tsval))| \geq b+1$
4: $highCand(c) ::= (c \in C) \wedge (\neg \exists c' \in C : c'.tsval.ts > c.tsval.ts)$
5: $Resp1 ::= \{i : RespFirst[i] = true\}$

Initialization:
6: $tsr'_j := 0$

read() is {
7: $C := FirstRW := RW := RPW := \emptyset$
8: $RespFirst[i] := false, 1 \leq i \leq S$
9: **inc**(tsr'_j); $tsrFR := tsr'_j$
10: send READ1$\langle tsr'_j \rangle$ to all objects
11: **wait for** READ1_ACK messages **until**
$\exists Resp1OK \subseteq Resp1 : (|Resp1OK| \geq S - t) \wedge (\forall i,k \in Resp1OK : \neg conflict(i,k))$
12: **inc**(tsr'_j)
13: send READ2$\langle tsr'_j \rangle$ to all objects
14: **wait for** READ2_ACK messages **until**
$\exists c_{ret} \in C : ((safe(c_{ret}) \wedge highCand(c_{ret})) \vee (C = \emptyset))$
15: **if** $C = \emptyset$ **then**
16: **return**(v_0)
17: **else**
18: $c_{ret} := c : ((c \in C) \wedge (safe(c)) \wedge (highCand(c)))$
19: **return**$(c_{ret}.tsval.v)$
20: **endif**

21: **upon** reception of READ1_ACK$\langle tsr'_j, pw', w' \rangle$ from s_i **do**
22: $FirstRW(w') := FirstRW(w') \cup \{i\}$
23: $RW(w') := RW(w') \cup \{i\}$; $RPW(pw') := RPW(pw') \cup \{i\}$
24: $C := C \cup \{w'\}$; $RespFirst[i] := true$

25: **upon** reception of READ2_ACK$\langle tsr'_j, pw', w' \rangle$ from s_i **do**
26: $RW(w') := RW(w') \cup \{i\}$; $RPW(pw') := RPW(pw') \cup \{i\}$;

27: **upon** $(c \in C)$ **and** $(|RespondedWO(c)| \geq t+b+1)$
28: $C := C \backslash \{c\}$
}

Figure 4.5: SWMR safe storage: read implementation - code of reader r_j

- Candidate value c was reported by at least one correct base object in the first round of the read rd. In this case, at least $b+1$ correct objects have already set their pw fields to $c.tsval$ before the second round of rd is invoked and these objects reply in the second round with $c.tsval$ or a later value in their pw fields. Hence, $safe(c)$ eventually holds.

- Consider now the second case, in which no correct object ever reports c in its w field to r_j. Eventually all correct objects, at least $S - t = t + b + 1$ of them respond with some value different from c in their w fields and c is excluded from C (lines 27-28).

- Finally, consider the third case in which: (1) no correct object reports c in its w field in the first round of read rd and (2) at least one correct object reports c in its w field in the second round of rd. In this case, some Byzantine objects have forged c, but c was indeed later written concurrently with read rd. Note that the content of the high resolution timestamp, $c.HRts$, is crucial in this case. It contains values of $tsr[j]$ fields of at least $S - t - t = b + 1$ correct objects that those objects reported to the writer during write wr (concurrent with rd) that actually wrote c. Denote by $tsrFR$ the timestamp of reader r_j in the first round of rd. Note that a correct object s_i sets $tsr[j]$ to a value higher than $tsrFR$ (i.e., to $tsrFR+1$, since by our assumption, rd never completes and, therefore, r_j never sets its timestamp to a value higher than $tsrFR + 1$) only upon s_i receives a second round message of rd.

  For every such correct object s_i, if $c.HRts[i][j] \leq tsrFR$, s_i responds to the second round of rd with $c.tsval$ in its pw field or with a later value (otherwise $c.HRts[i][j] > tsrFR$ in the PW round of wr). On the other hand, if $c.HRts[i][j] > tsrFR$ at the end of the first round of rd, every (Byzantine) object that reported c in its w field in the first round of the read will be in conflict with s_i. Therefore, (1) at the end of the first round of read, s_i is not in $Resp1OK$ and (2) s_i responds *without* c (and $c.tsval$) in the second round of the read. Roughly, in our algorithm, below a certain threshold of correct objects s_i for which $c.HRts[i][j] > tsrFR$, $safe(c)$ will eventually hold. If the number of correct objects s_i such that $c.HRts[i][j] > tsrFR$ crosses this threshold, then, the number of objects that responded without c in their w fields eventually becomes larger than $t + b$, i.e., c is removed from C.

In other words, in any execution ex of our algorithm, for any $c \in C$, $safe(c)$ eventually holds in ex, or c is eventually removed from C (in ex).

4.3.3 Correctness

We first prove *safety*.

Theorem 13. *(Safety) The algorithm in figures 4.3, 4.4 and 4.5 is safe.*

Proof. We consider the case in which read rd by reader r_j is not concurrent with any write. Let $c_k = \langle\langle k, val_k\rangle, HRts_k\rangle = \langle tsval_k, HRts_k\rangle$ be the tuple written in

the W round of the latest write wr_k (that writes value val_k) that precedes rd (or $c_0 = \langle\langle 0, \perp\rangle, initHRts\rangle$ if there is no such a write). We show that rd does not return a value other than val_k (where $val_0 = \perp$).

By write implementation, timestamp/value pair $c_k.tsval = tsval_k$ (resp., tuple c) has been written in the pw (resp., w) fields of at least $S - t = t + b + 1$ objects before write completes, including at least $t + 1$ non-Byzantine objects (or, to all of the $2t + 1$ non-Byzantine objects, in case $c_k = c_0$). Therefore, throughout the duration of rd: (1) at least $t + 1$ non-Byzantine objects have $tsval_k$ in their pw, and c_k in their w fields and (2) at most $t + b$ objects have in their w field a tuple different than c_k. By the read code, responses from at least $t + b + 1$ objects are awaited in the first round of rd (line 11, Fig. 4.5). Hence, at least one of non-Byzantine objects will respond with c_k in its w field in the first round of rd. Hence, by the end of the first round of rd, $c_k \in C$. Moreover, since at most $t + b$ objects have in their w fields a tuple different than c_k throughout rd, c_k is never excluded from C in lines 27-28, Fig. 4.5. Hence, C does not return a default value v_0 (lines 15 and 16, Fig. 4.5). Moreover, note that no tuple c with $c.tsval.ts > k$ can be returned, as no such a tuple (candidate value) c can be $safe(c)$. Indeed, note that throughout rd no non-Byzantine object, out of at least $S - b = 2t + 1$ of them, will reply in its pw or w field with a value with $ts' > k$, or $ts' = k \wedge v' \neq val_k$, i.e., at most b objects may respond with such a value. Hence, no value other than val_k is returned in line 19, Fig. 4.5.

□

We now proceed to proving *wait-freedom*. First we prove a couple of important lemmas. In the remainder of this section, we prove properties for read operations invoked by correct readers.

Lemma 27. *(No conflict between correct objects) At any point in time during the first round of any read operation, for every pair of correct objects s_i, s_k, $conflict(i, k) = false$.*

Proof. Assume, by contradiction, that there is read operation rd by r_j in which $conflict(i, k) = true$ during the first round of rd (i.e., before r_j executes the code in line 12 in Figure 4.5) and objects s_i and s_k are correct. Let the timestamp of r_j in the first round of rd be $tsrFR = tsr'_j$. Since $conflict(i, k) = true$, a correct object s_k reported, in the first round of rd, in its w field, a candidate value $c = \langle\langle ts, v\rangle, HRts\rangle$, such that $HRts[i][j] > tsrFR$. Since, by our assumption, s_k is correct, it only changes its w field upon s_k receives a PW or W message from the writer. Since the writer is not Byzantine, a timestamp value pair $\langle ts, v\rangle$ was indeed written, say by write wr_{ts}, and PW round of wr_{ts} has completed before s_k changed its w field to c (this occurs upon s_k receives a W message in write wr_{ts} or a PW message in write wr_{ts+1}, the write that immediately follows wr_{ts}). Hence, the writer received $tsr[j] > tsrFR$ from s_i and set $HRts[i][j] = tsr[j]$, before sending a W message in wr_{ts} (or a PW message in wr_{ts+1}), i.e., before s_k replied to the reader in the first round of rd and before the reader received this reply during the first round of rd. Hence, object s_i has set its $tsr[j]$ field to $tsr[j] > tsrFR$ before the reader has changed its timestamp to a value higher than $tsrFR$. According to the object code, no correct object can have the reader

r_j's timestamp ($tsr[j]$) higher than r_j itself (tsr'_j) at any point in time. Therefore, s_i is not correct, a contradiction. □

Lemma 28. *(First round of* read *terminates) The* read *operation implementation never remains indefinitely blocked at line 11, Fig. 4.5.*

Proof. In our model, there are at least $t + b + 1$ correct objects that will all eventually respond to the first round of every read (if the condition in line 11 is not satisfied earlier). Denote this set as X, $X \subseteq Resp1$. By Lemma 27, for no two $i, k \in X$ $conflict(i, k) = true$. Finally, as $|X| \geq t + b + 1$, the *until* condition in line 11 is satisfied in every read. □

Lemma 29. *(Second round of* read *terminates) The* read *operation implementation never remains indefinitely blocked at line 14, Fig. 4.5.*

Proof. Suppose, by contradiction, that there is read rd by r_j that remains indefinitely blocked at line 14. It is not difficult to see that, in this case, there exists a candidate value/tuple $c = \langle tsval, HRts \rangle$ such that $c \in C$ (i.e., $C \neq \emptyset$) forever, but $safe(c)$ never holds. We consider two cases: (1) c has been reported in the w field of some correct object s_i in the first round of rd and (2) no correct object s_i reported c in its w field in the first round of rd.

1. Consider first the case in which some correct object s_i has reported in its w field a tuple $c = \langle tsval, HRts \rangle$ (where $tsval = \langle ts, val \rangle$) in the first round of rd. Since correct objects set their w fields upon reception of the W message from the writer in wr_{ts}, or upon reception of the PW message from the writer in wr_{ts+1} and since the writer sends those messages only when at least $b + 1$ correct objects respond to its PW message in wr_{ts}, we conclude that, by the time s_i sends its response in the first round of rd, at least $b + 1$ correct objects have set their pw fields to $tsval$ and before the second round of rd is invoked. These correct objects eventually respond in the second round of rd with $tsval$ or with a higher timestamp in their pw fields. Hence, eventually $safe(c)$ holds. A contradiction.

2. Consider now the case in which no correct object s_i has reported in its w field a tuple $c = \langle tsval, HRts \rangle$ in the first round of rd (i.e., in this case, $FirstRW(c)$ contains only Byzantine objects). We distinguish two subcases:

   a) no correct object reports c in its w field in the second round of rd. In this case, c is excluded from C as soon as all correct objects respond to the second round of rd (lines 27-28, Fig. 4.5). A contradiction.

   b) there is a correct object s_k that reports c in its w field in the second round of rd. In this case, let $tsrFR$ be the timestamp of r_j during the first round of rd. Since $c = \langle tsval, HRts \rangle$ is reported by a correct object s_k in its w field in the second round of rd, c is indeed written by the writer at some point, concurrently with rd. Therefore, exactly $t + b + 1$ coordinates of $HRts[*][j]$ have non-nil values, out of which at least $b + 1$ correspond to correct objects. Denote this set of correct

objects by $X_{correct}$ (actually, the set of object indices). Denote by X_{fake} the set $X_{correct} \cap \{i : HRts[i][j] > tsrFR\}$.

Denote by $Resp1OK_c$ the set which satisfies the condition in line 11, at the end of the first round of rd. Note that such a set exists, and it contains (an index of) at least 1 Byzantine object s_m that reported c in its w field in the first round of rd (i.e., $m \in FirstRW(c)$); indeed if all objects in $Resp1OK_c$ were correct (or none of them reported c in its w field), c would be removed from set C, since no correct object responds in the first round of rd with c in its w field. Note also that $X_{fake} \cap Resp1OK_c = \emptyset$, since for every $i \in X_{fake}$ and every $m \in FirstRW(c)$, $conflict(i, m) = true$.

Furthermore, let $|FirstRW(c)| = f \geq 1$ (recall that $FirstRW(c)$ contains only Byzantine objects) and $|X_{fake}| = f' \geq 0$. At the end of the first round of rd, $|Resp1OK_c \backslash FirstRW(c)| \geq t + b + 1 - f$ (counting all those objects from $Resp1OK_c$ that did not respond with c in their w fields), i.e., by the end of the first round of rd at least $t+b+1-f$ objects responded without c in their w field, and this does not include any of the objects from X_{fake}.

Since c is indeed written (say by write wr) concurrently with rd, correct objects from X_{fake} must have responded to PW message of wr with the timestamp of reader r_j $tsr[j] = tsrFR + 1$, *after* they respond to the second round of rd, when they set their $tsr[j]$ fields to $tsrFR + 1$. By our assumption on rd, the second round of rd does not complete and r_j never sets its timestamp tsr'_j to a value higher than $tsrFR + 1$. Hence, for any $s_i \in X_{fake}$, $tsr[j]$ is not higher than $tsrFR + 1$. Therefore, by the time rd receives the second round responses from all correct objects, all objects from X_{fake} respond without c in their w fields and the number of objects that have responded during rd without c in their w fields, $|RespondedWO(c)|$, is at least $t + b + 1 - f + f'$.

On the other hand, all of at least $b + 1 - f'$ correct objects from $X_{correct} \backslash X_{fake}$ respond to the PW round of wr *before* they reply to the second round of rd. Therefore, these at least $b + 1 - f'$ objects reply to the second round of rd with $c.tsval$ or the value with a higher timestamp in their pw field. Hence, by the time rd receives the second round responses from all correct objects, the number of objects that have responded with c in their w field, or $c.tsval$ in their pw fields, or with a later value, is at least $f + b + 1 - f'$.

By our assumption: (i) $safe(c)$ never holds during rd and (ii) c is never excluded from C during rd. These conditions can be written as:

(i) $f + b + 1 - f' < b + 1$
(ii) $t + b + 1 - f + f' < t + b + 1$

However, it is not difficult to see that, for any values of f and f', at least one of these inequalities is false. Indeed, rewriting (i) and (ii)

yields:

(i) $f < f'$
(ii) $f' < f$

apparently, at least one of the last two inequalities must be false. Therefore, we conclude that, eventually (at latest upon rd receives second round responses from all correct objects), either $safe(c)$ holds, or c is eliminated from C. A contradiction.

□

Theorem 14. *(Wait-Freedom) The algorithm in figures 4.3, 4.4 and 4.5 is wait-free.*

Proof. The proof of wait-freedom of the write implementation is straightforward. The wait-freedom of the read implementation follows from Lemmas 28 and 29. □

4.4 Regular Implementation

Our tight lower bound on the time complexity of read operations extends to stronger storage semantics: optimally resilient *regular* storage. In this section, we show how to transform our safe implementation (Section 4.3) to provide regular semantics while retaining optimal resilience and optimal time complexity of read and write operations (i.e., rounds). The proof of correctness of our regular implementation is given in Section 4.4.2.)

The main difference between our regular implementation and our safe implementation, is that objects keep track of all values they receive from the writer throughout the entire execution (for simplicity we say that objects *store the entire history*). For presentation simplicity, we will assume in the following that in every read round, objects send all the values received from the writer (i.e., the entire history) to the reader. However, later, in Section 4.4.1, we show how to simply optimize our implementation in order to drastically decrease the size of messages exchanged between objects and readers in our algorithm (as well as memory requirements and computational complexity at readers).

The communication pattern of our regular implementation is the same as that of our safe implementation of Section 4.3. Moreover, the principle of choosing the value to return in the reader code is essentially the same, only the set of candidate values to choose from becomes bigger than in our safe implementation.

The write implementation remains unchanged, i.e., we can reuse the implementation given in Figure 4.3, Section 4.3.

However, object s_i, on reception of $\mathrm{PW}\langle ts', pw', w'\rangle$ from the writer, with $ts' > ts$, where ts is the timestamp of the latest PW or W message received by s_i from the writer, updates ts and assigns $history_i[ts'] := \langle pw', nil\rangle$ and $history_i[ts' - 1] := \langle w'.tsval, w'\rangle$ (lines 5-7, Figure 4.6). Similarly, on reception of $\mathrm{W}\langle ts', pw', w'\rangle$ from the writer, with $ts' \geq ts$, s_i updates ts and assigns $history_i[ts'] := \langle pw', w'\rangle$ (lines 11-12, Figure 4.6).

Initialization:
1: $ts := 0$; $pw_0 := \langle 0, \perp \rangle$; $history_i[0] := \langle pw_0, \langle pw_0, initHRts \rangle \rangle$
2: $initHRts[i][j] := nil$, $1 \leq i \leq S$, $1 \leq j \leq R$
3: $tsr[j] := 0$, $1 \leq j \leq R$

4: **upon** reception of PW$\langle ts', pw', w' \rangle$ message from the writer **do**
5: **if** $ts' > ts$ **then**
6: $history_i[ts] := \langle pw', nil \rangle$; $history_i[ts-1] := \langle w'.tsval, w' \rangle$
7: $ts := ts'$
8: send PW_ACK$\langle ts, tsr \rangle$ to the writer
9: **endif**

10: **upon** reception of W$\langle ts', pw', w' \rangle$ message from the writer **do**
11: **if** $ts' \geq ts$ **then**
12: $ts := ts'$; $history_i[ts] := \langle pw', w' \rangle$
13: send W_ACK$\langle ts \rangle$ to the writer
14: **endif**

15: **upon** reception of READK$\langle tsr' \rangle$ mess. from r_j ($k \in \{1, 2\}$) **do**
16: **if** $tsr' > tsr[j]$ **then**
17: $tsr[j] := tsr'$
18: send READK_ACK$\langle tsr[j], history_i \rangle$ to reader r_j
19: **endif**

Figure 4.6: SWMR regular storage: code of object s_i

Moreover, on reception of the READK message from the reader with a timestamp tsr', the object s_i replies with the message READK_ACK$\langle tsr', history_i \rangle$, where K denotes the round (K$\in \{1, 2\}$). (We later show, in Section 4.4.1, how the size of READK_ACK messages can be drastically decreased). The entire modified object code is given in Figure 4.6.

We give the modified reader code in Figure 4.7. Reader r_j, on receiving READK_ACK$\langle tsr', history_i \rangle$ message from object s_i in round k of read rd, assigns $history[k][i] := history_i$ (line 19 and 24, Fig. 4.7). If, for some ts' the entry $history_i[ts']$ does not exist, r_j considers $history[k][i][ts'] = history_i[ts'] = \langle nil, nil \rangle$. The reader adds (non-nil) values of tuples $history[1][i][*].w$, i.e., the values objects report in their $history_i[*].w$ fields, into the set of candidate values C throughout the first round of rd (line 20, Fig. 4.7)

Similarly to our safe implementation, in the first round of rd, reader r_j awaits responses from a set that contains at least $S - t$ objects such that there is no *conflict* between any 2 objects s_i and s_k that belong to this set (line 11, Fig. 4.7). We again detect conflicts using high-resolution timestamps; a conflict between two objects arises when one object, say s_k reports (in the first round of read) in one of its $history_k[*].w$ fields a candidate value c, such that $c.HRts[i][j] > tsrFR$, where $tsrFR$ is the timestamp of reader r_j in the first round of the read. As in our safe implementations, there can be no conflict between two correct objects s_i and s_k.

We define two key predicates for candidate values $c \in C$, $safe(c)$ and $invalid(c)$ as follows:

- $safe(c)$. A candidate value c is *safe* if at least $b+1$ objects s_i have responded with either $c.tsval$ or c in pw or w field (respectively) of $history_i[c.tsval.ts]$ in either the first, or the second round of the read. (line 3, Fig.4.7). In other

Definitions:

1: $conflict(i,k) ::= \exists c \in C, \exists ts' : (history[1][k][ts'].w = c) \wedge (c.HRts[i][j] > tsrFR))$

2: $invalid(c) ::= |\{i : \exists rnd \in \{1,2\} :$
$(history[rnd][i][c.tsval.ts].w = nil) \vee$
$\vee (history[rnd][i][c.tsval.ts].pw \neq c.tsval) \vee$
$\vee (history[rnd][i][c.tsval.ts].w \neq c)\}| \geq t + b + 1$

3: $safe(c) ::= |\{i : \exists rnd \in \{1,2\} :$
$(history[rnd][i][c.tsval.ts].pw = c.tsval) \vee$
$\vee (history[rnd][i][c.tsval.ts].w = c)\}| \geq b + 1$

4: $highCand(c) ::= (c \in C) \wedge (\neg \exists c' \in C : c'.tsval.ts > c.tsval.ts)$

5: $Resp1 ::= \{i : RespFirst[i] = true\}$

Initialization:

6: $tsr'_j := 0$

read() is {

7: $history[1..2][1..S] := init$

8: $tsr[i] := 0$; $RespFirst[i] := false, 1 \leq i \leq S$

9: **inc**(tsr'_j); $tsrFR := tsr'_j$

10: send READ1$\langle tsr'_j \rangle$ to all objects

11: **wait for** READ1_ACK messages **until**
$\exists Resp1OK \subseteq Resp1 : (|Resp1OK| \geq S - t) \wedge (\forall i, k \in Resp1OK : \neg conflict(i,k)))$

12: **inc**(tsr'_j)

13: send READ2$\langle tsr'_j \rangle$ to all objects

14: **wait for** READ2_ACK messages **until**
$\exists c_{ret} \in C : ((safe(c_{ret}) \wedge (highCand(c_{ret}))))$

15: $c_{ret} := c : (c \in C) \wedge safe(c) \wedge (highCand(c))$

16: **return**$(c_{ret}.tsval.v)$

17: **upon** reception of READ1_ACK$\langle tsr'_j, history_i \rangle$ from s_i **do**

18: **if** $(tsr'_j > tsr[i])$ **then**

19: $tsr[i] := tsr'_j$; $history[1][i] := history_i$

20: $C := C \cup \{history_i[*].w'\}$; $RespFirst[i] := true$

21: **endif**

22: **upon** reception of READ2_ACK$\langle tsr'_j, pw', w' \rangle$ from s_i **do**

23: **if** $(tsr'_j > tsr[i])$ **then**

24: $tsr[i] := tsr'_j$; $history[2][i] := history_i$

25: **endif**

26: **upon** $(c \in C)$ **and** $(invalid(c))$

27: $C := C \backslash \{c\}$

}

Figure 4.7: SWMR regular storage: read implementation - code of reader r_j

words, c if safe if at least $b + 1$ objects confirm that the timestamp-value pair $c.tsval$ has been written by the writer in a write with a timestamp $c.tsval.ts$.

- $invalid(c)$. A candidate value c is deemed *invalid* if at least $t+b+1$ objects s_i are missing the entry $history_i[c.tsval.ts].w$ (i.e., if $history_i[c.tsval.ts].w = nil$), or reply with a value different than $c.tsval$ (resp., c) in the pw (resp., w) field of their $history_i[c.tsval.ts]$, in either the first, or the second round of read. (line 2, Fig.4.7). In other words, c if invalid if at least $t + b + 1$ objects did not receive c with a timestamp $c.tsval.ts$ from the writer.

As soon as the predicate $invalid(c)$ holds, c is removed from the set C (lines 26 and 27, Fig. 4.7).

The reader receives READ2_ACK messages (in the second round of read) until there is a candidate value c such that $safe(c)$ holds and there is no other candidate value with a higher timestamp. This is guaranteed to occur at latest after the reader receives the responses from all correct objects in the second round of read. Roughly, the principle behind this fact, is the same as in our safe implementation.

4.4.1 Performance optimization

It is relatively easy to see how we can simply modify our regular implementation such that objects do not send their entire histories to readers within the READK_ACK messages. Consider read rd by r_j. It is sufficient that reader r_j stores (caches) the value $cache_j.val$ it returned in its last read that preceded rd along with the timestamp associated with $cache_j.val$, $cache_j.ts$. Then, in the first round of rd, r_j includes $cache_j.ts$ in its READ1 message, and the object s_i send in READK_ACK messages in rd only the portion of the $history_i$ from $history_i[cache_j.ts]$ onwards. It may occur in this case that, after two rounds of read, the set C is empty. In this case, r_j simply returns $cache_j.val$. The rest of the algorithm can be reused as such.

4.4.2 Correctness

First we prove *regularity.*

Theorem 15. *(Regularity) The algorithm in figures 4.3, 4.6 and 4.7 is regular.*

Proof. Consider read rd by reader r_j, such that the last value written by some complete write (wr_k) that precedes rd is val_k (with a timestamp k), or $val_0 = \perp$ if there is no such write.

We show that no value older than val_k is returned by rd. Moreover, we show that if val_l is returned by rd then there is a wr_l that writes val_l.

Let $c_k = \langle\langle k, val_k\rangle, HRts_k\rangle = \langle tsval_k, HRts_k\rangle$ be the tuple written in round W of the latest complete write wr_k that precedes rd (or $c_k = c_0 = \langle\langle 0, \perp\rangle, initHRts\rangle$ if there is no such a write). We show that rd does not return a value older than val_k.

By write implementation, a timestamp value pair val_k (resp, a tuple c_k) has been written in $history_*[k].pw$ (resp., $history_*[k].w$) fields of at least $S - t - b = t + 1$ non-Byzantine objects before write completes (or, to all non-Byzantine objects, in case $tsval_0 = \langle 0, \perp \rangle$). Therefore, throughout the duration of rd the following conditions hold:

- Condition (1). At least $t + 1$ non-Byzantine objects have $tsval_k$ in their $history_*[k].pw$, and c_k in their $history_*[k].w$ fields.
- Condition (2). At most $t+b$ objects have in their $history_*[k].w$ (or $history_*[k].pw$) fields a tuple different than c_k (resp., $tsval_k$), or they do not have an entry for $history_*[k]$.

By the read code, responses from at least $S - t = t + b + 1$ objects are awaited in the first round of read (line 11, Fig. 4.7). Therefore, by condition (1), $w_k \in C$. Moreover, by condition (2), w_k is never excluded from C in lines 26-27, Fig. 4.7). Therefore, rd never returns a value older than $c_k.tsval.val = val_k$.

Moreover, note that no tuple c such that $c.tsval.val$ has never been written by the writer can be returned, since no such a tuple (candidate value) c can be $safe(c)$. Indeed, since $c.tsval.val$ has never been written by writer, no non-Byzantine object, out of at least $S - b = 2t + 1$ of them, will ever store $history_*[c.tsval.ts].pw = c.tsval$, or $history_*[c.tsval.ts].w = c$, i.e., at most b objects may respond with such values in their $history_*[c.tsval.ts]$ fields. □

Performance optimization. Now we prove that our performance optimization described in Section 4.4.1 preserves regularity.

Again, consider read rd by reader r_j, such that the last value written by some complete write (wr_k) that precedes rd is val_k (with a timestamp k), or val_0 if there is no such write. Denote by $cache_j.val$ the value returned by the last read invoked by r_j that immediately precedes rd (or $\perp$ if there is no such a value) and by $cache_j.ts$ the timestamp associated by the writer to that value in wr_{ts}, (or $cache_j.ts = 0$ if there was no such a write). We distinguish two cases:

- $(ts < k)$. In this case, entries $history_*[k]$ will be sent by all (non-Byzantine) objects in both rounds of rd, so the argument we used above for the non-optimized version can be reused.
- $(ts \geq k)$. In this case, rd returns a val_ts or a newer value. Regularity is preserved.

□

We now proceed to proving wait-freedom. We revisit the lemmas used in Section 4.3.3 in the proof of correctness of our safe storage implementation.

Lemma 30. *(No conflict between correct objects) At any point in time during the first round of any read operation, for every pair of correct objects s_i, s_k, $conflict(i, k) = false$.*

Proof. Suppose, by contradiction, that there is read operation rd by r_j in which $conflict(i,k) = true$ during the first round of rd (i.e., before r_j executes the code in line 12) and objects s_i and s_k are correct. Suppose that the timestamp of r_j in the first round of rd is $tsrFR = tsr'_j$. Since $conflict(i,k) = true$, correct object s_k reported candidate value $c = \langle\langle ts, v\rangle, HRts\rangle$ in the first round of rd, in its $history_k[ts].w$ field (for some ts), such that $HRts[i][j] > tsrFR$. By the assumptio that, s_k is correct, s_k only changes its w field to c upon it receives a W message in wr_{ts} from the writer. Since, the writer is not Byzantine, the writer has received $tsr[j] > tsrFR$ from s_i and set $HRts[i][j] = tsr[j]$ in the first round of wr_{ts}, before sending a W message in wr, i.e., before s_k replied to the reader in the first round of rd and before the reader received this reply during the first round of rd. Hence, object s_i has sent to the writer a timestamp of a reader r_j $tsr[j] > tsrFR$ before reader r_j has changed its timestamp to a value higher than $tsrFR$. According to the object code, no correct object can have a reader r_j's timestamp higher than r_j itself at any point of time. Therefore, s_i is not correct, a contradiction. □

Lemma 31. *(First round of read terminates) The read operation implementation never remains indefinitely blocked at line 11, Fig. 4.7.*

Proof. The proof is an analogue of that of Lemma 28, Section 4.3.3. □

Lemma 32. *(Second round of read terminates) The read operation implementation never remains indefinitely blocked at line 14, Fig. 4.7.*

Proof. Suppose, by contradiction, that there is read rd by r_j that remains indefinitely blocked at line 14.

It is not difficult to see that, in case of our original non-optimized implementation, the set C is never empty, since the initial tuple $c_0 = \langle pw_0 = \langle 0, \perp\rangle, initHRts\rangle$ appears in C and is never excluded since all $2t+1$ non-Byzantine objects have $history_*[0] = \langle pw_0, c_0\rangle$. On the other hand, in our optimized version, if C is empty then rd returns a $cache_j.val$ value and, hence, the second round of rd terminates and rd completes.

Therefore, there exists a candidate value/tuple $c = \langle tsval, HRts\rangle \neq c_0$ such that $c \in C$ (i.e., $C \neq \emptyset$) forever, but $safe(c)$ never holds. We consider two cases: (1) c has been reported in the $history_i[tsval.ts].w$ field in the first round of rd by some correct object s_i, and (2) no correct object s_i reported c in its $history_i[tsval.ts].w$ field in the first round of rd.

1. Consider first case in which some correct object s_i reports tuple $c = \langle tsval, HRts\rangle$ (where $tsval = \langle ts, val\rangle$) in its $history_i[tsval.ts].w$ field, in the first round of rd. Since correct objects can set their $history_i[tsval.ts].w$ fields to c only upon reception of a W message from the writer in wr_{ts} and since the writer sends this message only after at least $b+1$ correct objects respond to its PW message in wr_{ts}, we conclude that, by the time s_i sends its response in the first round of rd, at least $b+1$ correct objects have set their $history_i[c.tsval.ts].pw$ fields to $tsval$ before the second round of rd is invoked. These correct objects eventually respond in the second round of rd

with $history_i[c.tsval.ts].pw = tsval$, and, hence, eventually $safe(c)$ holds. A contradiction.

2. Consider now case in which no correct object s_i has reported in its field $history_i[c.tsval.ts].w$ a tuple $c = \langle tsval, HRts \rangle$ in the first round of rd. We distinguish two cases: (

   a) no correct object reports c in its $history_i[c.tsval.ts].w$ fields in the second round of rd. It is not difficult to see that c is deemed *invalid* and excluded from C, as soon as all correct objects (at least $t + b + 1$ of them) respond to the second round of rd. A contradiction.

   b) there is correct object s_k that reports c in its $history_i[c.tsval.ts].w$ fields in the second round of rd. Let $tsrFR$ be the timestamp of r_j during the first round of rd. Since c is reported by correct object s_k in its $history_i[c.tsval.ts].w$ field in the second round of rd, c is indeed written by the writer at some point, during rd. Therefore, exactly $t + b + 1$ coordinates of $HRts[*][j]$ have non-*nil* values, out of which at least $b + 1$ correspond to correct objects. Denote this set of correct objects as $X_{correct}$ (actually the set of object indices). Denote by X_{fake} the set $X_{correct} \cap \{i : HRts[i][j] > tsrFR\}$.

   Denote by $Resp1OK_c$ the set which satisfies the condition in line 11, at the end of the first round of rd. Note that such a set exists, and it contains (an index of) at least 1 Byzantine object s_m that reported c in its $history_i[c.tsval.ts].w$ field in the first round of rd; indeed if all objects in $Resp1OK_c$ were correct (or none of them reported c in its $history_i[c.tsval.ts].w$ field), c would be removed from set C (i.e., $invalid(c)$ would hold), since no correct object responds in the first round of rd with c in its $history_i[c.tsval.ts].w$ fields. Note also that $X_{fake} \cap Resp1OK_c = \emptyset$, since for every $i \in X_{fake}$ and every $m \in FirstRW(c)$, $conflict(i, m) = true$.

   Furthermore, denote by f the cardinality of set M of (Byzantine) objects that have reported c in the first round of rd ($f \geq 1$) in their $history_i[c.tsval.ts].w$ fields and $|X_{fake}| = f' \geq 0$. At the end of the first round of rd, $|Resp1OK_c \backslash M| \geq t + b + 1 - f$ (counting all those objects from $Resp1OK_c$ that did not respond with c in their $history_i[c.tsval.ts].w$ fields), i.e., by the end of the first round of rd at least $t + b + 1 - f$ objects responded without c in their $history_i[c.tsval.ts].w$ fields, and this does not include any of the objects from X_{fake}.

   Since c is indeed written (by wr_{ts}) concurrently with rd, correct objects from X_{fake} must have responded to PW message of wr with the timestamp of reader r_j $tsr[j] = tsrFR + 1$, *after* they respond to the second round of rd. Therefore, all objects from X_{fake} eventually respond to the second round of rd with no entry for $history_i[c.tsval.ts]$. Hence, by the time rd receives the second round responses from all correct objects, the object count for $invalid(c)$ (line 2, Fig. 4.7) predicate is at least $t + b + 1 - f + f'$.

On the other hand, all of at least $b + 1 - f'$ correct objects from $X_{correct} \backslash X_{fake}$ respond to the PW round of wr_{ts} *before* they reply to the second round of rd. Therefore, these at least $b + 1 - f'$ objects reply to the second round of rd with $history_i[c.tsval.ts].pw = c.tsval$. Hence, by the time rd receives the second round responses from all correct objects, the number of objects that have responded with c in their $history_i[c.tsval.ts].w$ field, or $c.tsval$ in their $history_i[c.tsval.ts].pw$ fields, is at least $f + b + 1 - f'$.

By our assumption: (i) $safe(c)$ never holds during rd and (ii) $invalid(c)$ never holds during rd. These conditions can be written as:

(i) $f + b + 1 - f' < b + 1 \iff f < f'$
(ii) $t + b + 1 - f + f' < t + b + 1 \iff f' < f$

apparently, at least one of the last two inequalities must be false. Therefore, we conclude that, eventually (at latest upon rd receives second round responses from all correct objects), either $safe(c)$ holds, or c is eliminated from C. A contradiction.

□

Finally, Lemma 28 and 29 prove the following theorem, since the wait-freedom of write is straightforward.

Theorem 16. *(Wait-Freedom) The algorithm in figures 4.3, 4.6 and 4.7 is wait-free.*

4.5 Atomic storage

Using the simple transformation from SWMR regular register to SWMR atomic register [GR06], and our regular algorithm shown in Section 4.4, we can narrow down the possible range for the optimal latency of read operation in optimally resilient Byzantine fault tolerant atomic storage. Namely [GR06], by using $R+1$ (where R is the number of readers) SWMR regular registers implemented over the same set of base objects, such that each of the R registers is written by some (atomic) reader and read by all other readers, as well as a write-back technique based on timestamps (in the vein of [ABD95]), we can very simply implement a SWMR atomic register in which the complexity of read is 4 rounds, Notice that such an algorithm is optimally resilient with respect to base object Byzantine failures but assumes non-Byzantine clients.

Hence, we can conclude that the optimal complexity for optimally resilient BFT atomic storage is between 2 and 4 rounds. The exact complexity remains an open problem.

4.6 Server-Centric Model

Here, we consider the extension of our storage model of Chapter 2 to a *server-centric model*, by assuming point-to-point channels among base objects (servers) and removing the restriction that base objects can send messages only in response to clients. In other words, in the server-centric model, base objects are first class active processes (servers) that can exchange messages with other servers and even send unsolicited messages to clients (i.e., *push* messages). As a consequence, the range of communication patterns is very broad and not bound by the pattern of a communication round-trip.

For example, clients in a server-centric model may send only one message to (a subset of) servers and wait for the reception of pushed messages, until they receive sufficient amount of information for returning a value. It is not difficult to see that, in an asynchronous system, clients need only to send this first message m to a subset of servers and to receive one message that causally depends [BJ95] on m, in order to return a meaningful value. Intuitively, the fastest possible operation in this model is similar to that of our model of Chapter 2; i.e., a 2 message delay operation *op* in which: (a) the client c sends messages to (a subset of) servers, (b) servers, on receiving such a message, reply to c, *without waiting for the reception of any other message from any other server or client* and (c) upon c receiving a sufficient number of these replies *op* completes.

Hence, the complexity of 2 message delays remains the lower bound on the (worst case) time complexity of storage implementations even in the server-centric model, i.e., our lower bound (Proposition 1 of Section 4.2) holds in this model as well. In other words, even in the server-centric model, if at most $2t + 2b$ servers are used, then it is impossible to construct a SWSR safe storage in which the complexity of read is at most 2 message delays. Devising a tight bound algorithm for a server-centric model is, however, out of the scope of this thesis. Intuitively, one can expect to obtain a 3 message delay regular storage algorithm by exploiting the all-to-all communication among servers and by using the idea behind our high-resolution timestamps.

5

Fast BFT Atomic Storage

In the previous Chapter, we have established a new tight lower bound on worst-case time complexity of reading from optimally resilient BFT storage. Namely, we showed that, in the unauthenticated model, reading from any (safe) storage implemented over at most $2t + 2b$ base objects (where t and b are, respectively, thresholds on the total number of failures and the number of Byzantine failures) requires at least 2 round-trips, i.e., no *fast* **read** implementation is possible in this case. The analog result for *fast* **write** implementations was established in [ACKM06].

Given these results, it is somehow natural to ask whether it is possible to achieve fast implementations by trading in the number of base objects, i.e., by giving away optimal resilience. The question is particularly interesting in the case of the atomic storage. The answer to this question in the crash-only model, given in [DGLC04], may give a good intuition to what can we expect when allowing for Byzantine failures. Namely, it was shown in [DGLC04] that a tight lower bound on the number of base objects S required for a fast atomic storage implementation (i.e., an atomic storage implementation in which all reads and writes complete in at most one round-trip) is $S \geq (R + 2)t + 1$, where R is the total number of readers in the system. In other words, fast atomic implementations impose an inherent limitation with respect to the number of readers they can support. Naturally, a BFT fast atomic storage must require at least as many base objects. Still, is this number sufficient?

In this Chapter, we show a new tight lower bound on the number of base objects required for any BFT fast atomic storage implementation. Namely, we show that such an implementation requires $S \geq (R + 2)t + (R + 1)b + 1$ base objects, i.e., that such an implementation can support up to $R < \frac{S+b}{t+b} - 2$ readers. Interestingly, we show that this lower bound applies even in the authenticated model, where the space of allowable Byzantine behavior is considerably smaller than in the unauthenticated model (see Chapter 2 for details). In this chapter, as in Chapter 3, we assume that clients are not Byzantine (only base objects can be Byzantine; however, any number of clients may fail by crashing).

It is worth noting that this limitation on the number of readers that a fast BFT storage implementation can support comes directly from the requirement

that we require storage to be *atomic*. Namely, in the case of weaker, regular storage, fast BFT implementations in the authenticated model are almost trivial to achieve [MR98].

In the remainder of the Chapter, we first show a fast BFT atomic storage implementation (in the authenticated model) that makes use of $S \geq (R+2)t + (R+1)b + 1$ base objects. Then, we prove our implementation optimal with respect to the number of required base objects. Both algorithm and lower bound proof are, we believe, interesting in their own rights.

Our fast BFT atomic storage implementation, exploits the idea introduced by its predecessor in the crash failure model [DGLC04], the idea of *traces* left by readers in the base objects they access. These traces are then used to determine which value to return while preserving atomicity. In a sense, our implementation exploits base objects' read-modify-write capabilities, just like our algorithms of Chapter 4 (it would not be possible to directly port our algorithms to read/write shared memory model).

On the other hand, to get an intuition of our lower bound, consider S base objects, t among which can be faulty, out of which b can be Byzantine. Our lower bound proves by contradiction that there is no fast implementation with $R \geq \frac{S+b}{t+b} - 2$. Notice that our lower bound imposes the "price" of $t+b$ base objects on average for each additional reader. To see why, consider a partial execution which contains a write(1) that misses t base objects. Since a writer may fail, one cannot expect the remaining base objects to receive the written value and all the subsequent reads must return 1. Now append to the partial execution a read by reader r_1 that misses t other base objects, whereas, at the same time b Byzantine base objects (we denote this set of base objects by B_1) that witnessed both operations (a read and write(1)) fail by "losing memory" (this attack is perfectly possible even in the authenticated model). By atomicity, even in this case read must return 1. Then, before we append another read by reader r_2, we obtain a new partial execution by deleting all the steps in the partial execution that are not "visible" to the reader r_1 (basically, the steps of the t base objects that the read by r_1 missed) as well as all steps performed on the Byzantine base objects from the set B_1 that are correct in the new execution. By indistinguishability, the read returns 1 in the resulting partial execution. However, we deleted all the steps performed on $t+b$ base objects. Hence the intuition behind the average cost per reader in our lower bound. Notice that, since we require a fast implementation, readers "do not have time" to writeback the information about the value they read in the first round, which prohibits classical writeback techniques used in atomic storage implementations [ABD95].

Finally, we note that the results that we present in this Chapter directly extend to server-centric storage. Namely, and as we argued in Section 4.6, the notion of *fast* storage, as storage with the lowest possible read/write latency, extends to the server-centric storage model as well.

5.1 A Fast BFT Atomic Storage Implementation

5.1.1 Preliminaries

For simplicity of presentation, we first present our algorithm assuming that the writer writes timestamps, and the readers read back timestamps. More precisely, we assume that in every (partial) execution of our algorithm, the writer writes a timestamp k in the k^{th} invocation of write(k), where $k \geq 1$. In this section, we refer to such a write as wr_k. Moreover, reading an initial timestamp 0 corresponds to reading the initial storage value $\perp$. In other words, at first we ignore the values associated with the timestamps greater than 0. Later we explain how to trivially generalize our algorithm such that the writer and the readers associate some value with a timestamp.

With this simplification our definition of atomicity of a (partial) execution in our single-writer setting (properties (SWA1)-(SWA4) of Section 2.3.2, Chapter 2) is also simplified. Namely, to show that an execution is atomic we need to show the following:

1. If a read returns, it returns a non-negative integer.
2. If a read rd is complete and it follows some write wr_k, then rd returns l such that $l \geq k$.
3. If a read rd returns k ($k \geq 1$), then wr_k either precedes rd or is concurrent to rd.
4. If some read $rd1$ returns k ($k \geq 0$) and a read $rd2$ that follows $rd1$ returns l, then $l \geq k$.

5.1.2 Algorithm

The pseudo code of our fast implementation is given in Figure 5.1. Since our implementation has fast reads, we denote without ambiguity the messages sent by a reader, on invoking a read, by READ, and the messages sent by a base object to a reader, on receiving a READ message, by READACK. Similarly, we denote messages sent in our fast write operation by WRITE and WRITEACK messages. For better readability, pseudo code, the writer process w is denoted by 0, whereas a reader r_i ($i \in 1 \ldots R$) is denoted by i.

The write procedure is similar to that of [ABD95]. On invoking a write, the writer increments its timestamp (initialized to 0) and sends a WRITE message with the signed timestamp to all base objects in line 4 (in the remainder of this Section, we refer to Figure 5.1). Upon receiving the message, base objects store the timestamp (lines 25-26), along with the copy of the timestamp that contains the digital signature of the writer, and send WRITEACK messages back to the writer (lines 30-33). The writer returns OK once it has received WRITEACK messages from $S - t$ base objects (lines 5-7).

Implementing a fast read is more involved. Our read procedure collects timestamps (signed by the writer) from $S - t$ base objects (by sending READ messages and receiving READACK messages from the base objects), and selects the highest

0: **at the writer** $w = 0$**:**
1: **procedure** initialization:
2: $ts := 1$, $rCounter := 0$
3: **procedure** write(v)
4: send (WRITE, ts, $\sigma_w(ts)$, $rCounter$) to all base objects
5: **wait until** receive (WRITEACK, ts, $\sigma_w(ts)$, $*$, $rCounter$) from $S - t$ base objects
6: $ts := ts + 1$
7: return(OK)

---

at each reader $r_i = i$ **(**$i \in 1 \dots R$**):**

8: $admissible(TS, Msg, a) \equiv \exists \mu \subseteq Msg, \forall m \in \mu :$
$(m.ts = TS) \wedge (|\mu| \geq S - at - (a-1)b) \wedge (|\bigcap_{m' \in \mu} m'.updated| \geq a)$
9: **procedure** initialization:
10: $ts := 0$; $rCounter := 0$; $maxTS := 0$; $sig := \perp$
11: **procedure** read()
12: $rCounter := rCounter + 1$
13: send(READ, $maxTS$, sig, $rCounter$) to all base objects
14: **wait until** receive (READACK, ts', $\sigma_w(ts')$, $updated'$, $rCounter$) from $S-t$ base objects, such that: $\sigma_w(ts')$ is valid, $ts' \geq ts$ and $r_i \in updated'$
15: $rcvMsg := \{m | r_i$ received (READACK, $*$, $*$, $*$, $rCounter$) in line 14 $\}$
16: $maxTS := \mathbf{Max}\{ts' |$ (READACK, ts', $\sigma_w(ts')$, $*$, $rCounter$) $\in rcvMsg\}$
17: $sig := \sigma_w(maxTS)$
18: **if** there is $a \in [1, R+1]$: $admissible(maxTS, rcvMsg, a)$ **then**
19: return($maxTS$)
20: **else**
21: return($maxTS - 1$)

---

at each base object s_i**:**

22: **procedure** initialization:
23: $ts := 0$; $updated := \emptyset$; $sig := \perp$; $counter[0 \dots R] := [0 \dots 0]$
24: **upon** receive ($msgType$, ts', $\sigma_w(ts')$, $rCounter'$) from $q \in \{w, r_1, ..., r_R\}$ **and** ($rCounter' >$ $counter[q]$ **or** $q = w$) **do**
25: **if** $ts' > ts$ **then**
26: $ts := ts'$; $sig := \sigma_w(ts')$; $updated := \{q\}$
27: **else**
28: $updated := updated \cup \{q\}$
29: $counter[q] := rCounter'$
30: **if** $msgType =$ READ **then**
31: send(READACK, ts, sig, $updated$, $rCounter'$) to q
32: **else**
33: send(WRITEACK, ts, sig, $updated$, $rCounter'$) to q

Figure 5.1: Fast atomic storage implementation with $S \geq (R+2)t + (R+1)b + 1$

timestamp $maxTS$ (line 16). Moreover, the reader stores the writer's signature $\sigma_w(maxTS)$ into reader's variable sig (line 17). The pair $(maxTS, sig)$ selected by the reader will be written back by the reader in its next read invocation (lines 14 and 24-26). However, notice that in our fast implementation, base object s_i, besides storing the highest received timestamp ts and its copy containing the writer's signature sig, maintains also the set $updated$ that contains all clients to which s_i has sent ts. Basically, the set $updated$ contains all clients to which s_i has sent an update (in the form of a READACK or a WRITEACK message) about its highest timestamp. This information is read in our read procedure along with the signed timestamps. Hence, a reader collects timestamps and sets $updated$ from $S - t$ base objects using READ and READACK messages (lines 13-14).

As we described above, in every read invocation the reader sends the timestamp that it returned in its last preceding read, along with the respective writer's signature (using field ts of the READ message, line 13). The only exception to this is, obviously, the first read invocation by the reader, in which the reader issues a read message with the default timestamp 0 (see lines 10 and 23), which is also the initial timestamp at base objects. We assume that this initial value is known by all readers (and hence, does not need to be digitally signed by the writer) and treated by readers as valid.

Upon receiving $S-t$ READACK messages (collected in set $rcvMsg$, line 15), the read is precluded from waiting for more messages (the remaining t base objects may be faulty). Moreover, in order to have a fast implementation, the read may only perform some local computations and then it must return the value. In our fast implementation, the essence of this local computation is captured by predicate $admissible$ (line 8) which relies on sets $updated$ (received by the reader from base objects) to evaluate whether the highest received timestamp $maxTS$ may be returned. Namely, if there is integer a $(1 \leq a \leq R + 1)$, such that $admissible(maxTS, rcvMsg, a)$ holds in line 18 (we simply say that $maxTS$ is $admissible$ (with degree a)), a read returns $maxTS$ (line 19). Otherwise, if $maxTS$ is not admissible, a read returns $maxTS-1$ (line 21). In any case, $maxTS$ is cached locally and is written back by the reader in its following invocation of read using the ts field of a READ message as described previously (line 14).

In the following, we give an intuition behind the predicate $admissible()$ which is the heart of our fast implementation. The predicate is designed to guarantee that:

(a) $maxTS = k$ is admissible in read rd whenever wr_k precedes rd — this is vital for ensuring Property (2) of Section 5.1.1, and

(b) if $maxTS = k$ is admissible in read rd, then no rd' that follows rd returns a timestamp smaller than k — this is vital for ensuring Property (4) of Section 5.1.1.

First, we explain how our predicate guarantees (a). Consider the following partial execution, pr_1. In pr_1, write wr_k $(k \geq 1)$ completes by writing k to all base objects from some set Σ_1 containing $S - t$ base objects. There are no writes in pr_1 that follow wr_k. Moreover, read rd (by some reader r_i), that follows wr_k, reads from set Σ_2 (of $S - t$ base objects) that overlaps at $S - 2t$ base objects with Σ_1, i.e., misses t base objects in Σ_1. Moreover, in pr_1, b base objects from

$\Sigma_1 \cap \Sigma_2$ are Byzantine and respond to the reader with the timestamp $k-1$. By atomicity, rd must return k in pr_1, and it must do so without waiting for messages from base objects from $\overline{\Sigma_2}$ or the writer, since these may be faulty; the reader cannot actually tell Byzantine objects since these might as well be correct and not yet received WRITE message in write wr_k. In this case: (i) in line 16 of rd, $maxTS = k$, and (ii) for any message m received by r_i in rd from base objects from $\Sigma_1 \cap \Sigma_2 \cap \Sigma^{NB}$ (where Σ^{NB} denotes the set of benign, or non-Byzantine, base objects), we have $m.ts = maxTS$ and $\{w, r_i\} \subseteq m.updated$. Since $|\Sigma_1 \cap \Sigma_2 \cap \Sigma^{NB}| \geq S - 2t - b$, $maxTS$ is admissible in rd with degree $a = 2$.

On the other hand, the key to guaranteeing (b) is the following invariant (hereafter, $maxTS_{op}$ denotes $maxTS$ computed in line 16 of the read operation op):

Lemma 33. *Let rd' be a complete read (by reader r_j) that follows a complete read rd (by r_i). If $maxTS_{rd}$ is admissible with degree $a_{rd} \leq R+1$, then:*

- *$maxTS_{rd'} > maxTS_{rd}$, or*
- *$maxTS_{rd'} = maxTS_{rd}$ and $maxTS_{rd'}$ is admissible with degree 1 or degree $a_{rd}+1$ in rd'.*

Here, we sketch the proof of Lemma 33 (full correctness proof of our implementation can be found in Section 5.1.3).

Proof. By the definition of predicate *admissible* (line 8), there is a set of READACK messages μ_{rd}, sent by base objects from the set Σ_{rd}, such that, for every message m in μ_{rd}, $m.ts = maxTS_{rd}$, $|\Sigma_{rd}| = |\mu_{rd}| \geq S - a_{rd}t - (a_{rd}-1)b$ and $|\Pi_{rd}| \geq a_{rd}$, where $\Pi_{rd} = \bigcap_{m \in \mu_{rd}} m.updated$. Since (1) rd' follows rd, (2) $|\Sigma_{rd}| \geq t + b + 1$ and (3) rd' reads from $S - t$ base objects, rd' receives a READACK from at least 1 benign base object from Σ_{rd}. Hence, $maxTS_{rd'} \geq maxTS_{rd}$.

If $maxTS_{rd'} = maxTS_{rd} = k$, we distinguish two cases:
Case (i), $r_j \notin \Pi_{rd}$ (note that this case is possible only if $a_{rd} \leq R$). It is not difficult to see that k is admissible in rd' with degree $a_{rd}+1$. Indeed, rd' will miss at most $t+b$ benign base objects from Σ_{rd}, receiving at least $S-(a_{rd}+1)t-a_{rd}b$ READACK messages containing timestamp k, and the *updated* fields of these messages will be a superset of $\Pi_{rd} \cup \{r_j\}$, hence each containing at least $a_{rd}+1$ clients.
Case (ii), $r_j \in \Pi_{rd}$ (which indicates that rd' is not the first read by r_j). In this case, all benign base objects from Σ_{rd}, at least $t+1$ of them, have sent a READACK message to r_j containing the timestamp k before rd' is invoked. At least one of those must have been received by r_j in read rd'' that immediately precedes rd'. Hence, $maxTS_{rd''} = k$, and $ts = k$ in line 12 of rd'. Finally, eventually $S-t$ base objects send READACK to rd' with the timestamp equal to k and set *updated* that contains $\{r_j\}$ (see lines 25-28), i.e., k is admissible with degree 1 in rd'. □

Finally, to help distinguish READ and READACK messages from different reads of the same reader, we use the variable $rCounter$ that counts the number of reads of each particular reader. At the writer, the variable $rCounter$ is always 0; the messages from different writes are distinguished by their respective timestamps ($rCounter$ is kept here to simplify the base object code).

This completes the description of our fast 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. If the reader returns $maxTS - 1$ in the original algorithm, it returns the other tag attached to $maxTS$ in the modified algorithm.

5.1.3 Correctness of the Fast Implementation

Having in mind that we assume at least $S-t$ correct base objects, it is straightforward to show that read and write procedures complete in one round-trip. To show atomicity, we rely on Properties (1)-(4) of Section 5.1.1. The proof of Property (1) is trivial and it is not difficult to see that Property (3) holds, having in mind the unforgeability property of the writer's digital signatures and our assumption on benign writer. Below, we prove properties (2) and (4). In the proof, we use the following notation:

- $rcvMsg_{op}$ denotes the set of received READACK messages the reader collects in read operation op in lines 14-15;
- Σ_{op} denotes the set of base objects from which the reader received readack messages in $rcvMsg_{op}$ (in case op is a read), or the set of base objects from which the writer received WRITEACK messages in line 5 of op (in case op is a write). Notice that, for every complete operation op, $|\Sigma_{op}| = S - t$;
- $maxTS_{op}$ denotes $maxTS$ computed by the reader in line 16, in read op (i.e., the highest timestamp in messages in $rcvMsg_{op}$);
- $\mu_{op,a}$ denotes, in case $maxTS_{op}$ is admissible with degree a in op, the subset of $rcvMsg_{op}$, such that: (a) $|\mu_{op,a}| \geq S - at - (a-1)b$, (b) for all $m \in \mu_{op,a}$, $m.ts = maxTS_{op}$ and $m.sig = \sigma_w(maxTS_{op})$ and (c) $|\bigcap_{m \in \mu_{op,a}} m.updated| \geq a$ (see line 8, definition of predicate $admissible$);
- $\Sigma_{\mu_{op,a}}$ denotes the set of base objects that sent messages in $\mu_{op,a}$; and
- Finally, Σ^{NB} denotes the set of all benign (non-Byzantine) base objects. Notice that $|\Sigma^{NB}| \geq S - b$.

Lemma 34. *If benign object s sets ts to x at time T, then s never sets ts to a value that is lower than x after time T.*

Proof. By trivial base object code inspection. □

Lemma 35. *A read operation rd may only return either $maxTS_{rd}$ or $maxTS_{rd} - 1$.*

Proof. By lines 18-21 and the definition of $maxTS_{rd}$. □

Lemma 36. *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 $ts = x$. By lines 25-26, every READACK message received by r_i in rd from some benign base object s_j is with $ts_j \geq x$. Since $S > t + b$, $\Sigma^{NB} \cap \Sigma_{rd} \neq \emptyset$. Hence, $maxTS_{rd} \geq x$. There are the following two cases to consider. (1) If $maxTS_{rd} > x$, by Lemma 35, the return value is not smaller than x. (2) If $maxTS_{rd} = x$, then every READACK message m in $rcvMsg_{rd}$ has $m.ts = x$ and has $r_i \in m.updated$ (possibly different READACK messages (sent by Byzantine base objects) are discarded in line 14). Hence, predicate *admissible* holds for $maxTS_{rd}$ with degree $a = 1$ and rd returns $maxTS_{rd} = x$. □

The following Lemma proves Property 2 of Section 5.1.1.

Lemma 37. ***(Property 2)*** *If* read rd *is complete and follows the* write wr_k*, then rd returns l such that $l \geq k$.*

Proof. Suppose that wr_k precedes read rd (by reader r_i). Let $\Sigma' = \Sigma^{NB} \cap \Sigma_{wr_k} \cap \Sigma_{rd}$. Notice that Σ' contains only benign base objects. Since $|\Sigma^{NB}| = S - b$, $|\Sigma_{wr_k}| = S - t$ and $|\Sigma_{rd}| = S - t$, we have $|\Sigma'| \geq S - 2t - b$.

When a benign object s_i in Σ_{wr_k} (and, hence, in Σ') replies to a WRITE message from wr_k, its ts_i is at least k (the timestamp is not smaller than k due to the condition in line 25). Since wr_k precedes rd, by Lemma 34, benign base objects in Σ' reply with $ts_* \geq k$ to rd. Hence, $maxTS_{rd} \geq k$. There are the following two cases to consider:

1. $maxTS_{rd} > k$
   By Lemma 35, rd does not return a timestamp lower than k.

2. $maxTS_{rd} = k$
   Let μ' be the set of READACK messages sent by base objects in Σ' to rd. By definition of Σ_{rd} and since $\Sigma' \subseteq \Sigma_{rd}$, we have $\mu' \subseteq rcvMsg_{rd}$. Since (a) every (benign) base object $s_j \in \Sigma'$ replies to rd with $ts_j \geq k$, (b) $\mu' \subseteq rcvMsg_{rd}$ and (c) $maxTS_{rd} = k$, we have that every (benign) base object $s_j \in \Sigma'$ replies with $ts_j = k$ to rd (and r_j receives these replies).

   Moreover, since every (benign) object $s_j \in \Sigma'$ replies $ts_* = k$ to wr_k (since $\Sigma' \subseteq \Sigma_{wr_k}$) before sending $ts = k$ to rd (since wr_k precedes rd), for every message m in μ', $w \in m.updated$. Furthermore, since s_j replies with $ts_j = k$ to rd, by line 28, $r_i \in m.updated$. Thus, $\{w, r_i\} \subseteq \cap_{m \in \mu'} m.updated$. Since, $|\Sigma'| \geq S - 2t - b$, $maxTS$ is admissible in rd with degree $a = 2$. Hence, rd returns $maxTS_{rd} = k$.

□

The following auxiliary lemmas help prove Property 4 of Section 5.1.1.

Lemma 38. *Assume that $maxTS_{rd}$ is admissible with degree $a \in [1, R + 1]$ in some* read rd *and that a complete* read rd' *follows rd. Then, $\Sigma_{\mu_{rd,a}} \cap \Sigma_{rd'}$ contains at least $S - (a + 1)t - ab \geq 1$ benign base objects.*

Proof. Since $|\Sigma_{\mu_{rd,a}}| = |\mu_{rd,a}| \geq S - at - (a-1)b$, $|\Sigma^{NB}| = S - b$ and $|\Sigma_{rd'}| = S - t$, it follows that $|\Sigma^{NB} \cap \Sigma_{rd'} \cap \Sigma_{\mu_{rd,a}}| \geq S - (a+1)t - ab$. Moreover, since $a \in [1, R+1]$ and $S > (R + 2)t + (R + 1)b$, we have $S - (a + 1)t - ab \geq 1$. □

Lemma 39. *Assume that:*

1. *$maxTS_{rd}$ is admissible with degree $a \in [1, R+1]$ in some* read *rd,*
2. *a complete* read *rd' by reader r_j follows rd,*
3. *there is a set $X \subseteq \Sigma_{\mu_{rd,a}}$ of at least $t+1$ benign base objects, such that for all $s_i \in X$, s_i sends message $m_i \in \mu_{rd,a}$ with $r_j \in m_i.updated$.*

Then, rd' does not return a value smaller than $maxTS_{rd}$.

Proof. Since the messages in $\mu_{rd,a}$ are sent before the completion of rd (and hence, before the invocation of rd') and since there is a benign base object s_i that sends $m_i \in \mu_{rd,a}$ with $r_j \in m_i.updated$, r_i has invoked at least one read before rd'. Let rd'' be the last read by reader r_j which precedes rd'. Since $|X| \geq t+1$ and $|\Sigma_{rd''}| = S - t$, there is at least one benign base object s_k in $X \cap \Sigma_{rd''}$, such that the READACK message m sent by s_k is received by r_j in rd''. In the following paragraph, we show that $m.ts \geq maxTS_{rd}$.

By contradiction, assume $m.ts < maxTS_{rd}$. Since s_k is benign, it checks $counter[j]$ before replying to r_j: once m is sent by s_i (in reply to read rd''), $counter[j]$ at s_k is set such that s_k can only reply to those READ messages of r_j which are sent during reads by r_j that follow rd'' (lines 24 and 29). Hence there is a read rd_α by r_j, such that rd_α follows rd'' and s_k sends a READACK message m_α to rd_α, before s_k sends $m_k \in \mu_{rd,a}$, i.e., before rd' is invoked. Hence, rd'' is not the last read by reader r_j which precedes rd'. A contradiction.

Since $m.ts \geq maxTS_{rd}$ and $m \in rcvMsg_{rd''}$, we have $maxTS_{rd''} \geq maxTS_{rd}$. Since rd' follows rd'', it follows that r_j in rd' sends READ messages with $ts \geq maxTS_{rd}$. By Lemma 36, rd' returns a timestamp greater than or equal to $maxTS_{rd}$. □

The following Lemma proves Property 4 of Section 5.1.1.

Lemma 40. ***(Property 4)*** *If some* read *$rd1$ returns ret_{rd1} ($ret_{rd1} \geq 0$) and a* read *$rd2$ that follows $rd1$ returns ret_{rd2}, then $ret_{rd2} \geq ret_{rd1}$.*

Proof. Suppose that read $rd1$ by reader r_1 returns ret_{rd1}, read $rd2$ by reader r_2 returns ret_{rd2}, and $rd1$ precedes $rd2$. Suppose first $r_1 = r_2$. Then, in the read immediately after $rd1$, r_1 sends a READ message with $ts \geq ret_{rd1}$, and hence, by Lemma 36, the read returns a value greater than or equal to ret_{rd1}. Using Lemma 36 and simple induction, we can conclude that any read by r_1 which follows $rd1$ (including $rd2$) returns $ts \geq ret_{rd1}$. Hence, in the rest of the proof we assume that $r_1 \neq r_2$. We distinguish the following two cases:

⟨1⟩.1 $maxTS_{rd1}$ is not admissible in $rd1$.

It follows that $ret_{rd1} = maxTS_{rd1} - 1$. Since rd_1 does not return $maxTS_{rd1}$, and by the unforgeability of signatures, write $wr_{ret_{rd1}+1}$ started before $rd1$ completed, i.e., write $wr_{ret_{rd1}}$ completed before $rd1$ completed. Since $rd1$ precedes $rd2$, it follows that $wr_{ret_{rd1}}$ precedes $rd2$. By Lemma 37, $rd2$ returns $ret_{rd2} \geq ret_{rd1}$.

⟨1⟩.2 $maxTS_{rd1}$ is admissible in $rd1$.

It follows that $ret_{rd1} = maxTS_{rd1}$ and there is some $a \in [1, R+1]$ such that ret_{rd1} is admissible in $rd1$ with degree a. By Lemma 38, there is a benign base object $s_i \in \Sigma_{\mu_{rd1,a}} \cap \Sigma_{rd2}$. Furthermore, since $rd1$ precedes $rd2$, s_i first replies with $ts_i = ret_{rd1}$ to $rd1$ before s_i replies to $rd2$. Finally, by Lemma 34, it follows that s_i replies to $rd2$ with $ts_i \geq ret_{rd1}$, i.e., $maxTS_{rd2} \geq ret_{rd1}$.

We distinguish the following three exhaustive cases:

⟨2⟩.1 $maxTS_{rd2} > ret_{rd1}$

By Lemma 35, we have $ret_{rd2} \geq ret_{rd1}$.

⟨2⟩.2 $maxTS_{rd2} = ret_{rd1}$ and $maxTS_{rd2}$ is admissible in $rd2$

By lines 18-19, $ret_{rd2} = maxTS_{rd2} = ret_{rd1}$.

⟨2⟩.3 $maxTS_{rd2} = ret_{rd1}$ and $maxTS_{rd2}$ is not admissible in $rd2$

In this case, by lines 18-21, $ret_{rd_2} = maxTS_{rd2} - 1 = ret_{rd1} - 1 = maxTS_{rd1} - 1$. By Lemma 38, there is at least one benign base object in $\Sigma_{\mu_{rd1,a}} \cap \Sigma_{rd2}$. Since $rd1$ precedes $rd2$ and benign base objects in $\Sigma_{\mu_{rd1,a}}$ reply with $ts_* = ret_{rd1}$ to $rd1$, by Lemma 34, benign base objects in $\Sigma_{\mu_{rd1,a}} \cap \Sigma_{rd2}$ reply to $rd2$ with $ts_* \geq ret_{rd1}$. Since $ret_{rd2} + 1 = ret_{rd1} = maxTS_{rd2}$, every benign base object in $\Sigma_{rd2} \cap \Sigma_{\mu_{rd1,a}}$ replies to $rd2$ with $ts = ret_{rd1} = ret_{rd2} + 1$. There are the following two cases to consider:

⟨3⟩.1 $a \leq R$

In this case, by Lemma 38, $|\Sigma_{\mu_{rd1,a}} \cap \Sigma_{rd2} \cap \Sigma^{NB}| \geq S - (a+1)t - ab > t + b$. Let μ_1 be the set of READACK messages sent by objects in $\Sigma_{\mu_{rd1,a}} \cap \Sigma_{rd2} \cap \Sigma^{NB}$ (i.e., by benign objects in $\Sigma_{\mu_{rd1,a}} \cap \Sigma_{rd2}$) to $rd1$. There are two cases to consider:

⟨4⟩.1 $r_2 \notin \bigcap_{m \in \mu_1} m.updated$

Notice that, by definitions of μ_1 and $\mu_{rd1,a}$, $\mu_1 \subseteq \mu_{rd1,a}$. Hence, we have $\bigcap_{m \in \mu_1} m.updated \supseteq \bigcap_{m \in \mu_{rd1,a}} m.updated$. Thus, $|\bigcap_{m \in \mu_1} m.updated| \geq a$.

Let μ_2 be the set of messages received by $rd2$ from base objects in $\Sigma_{\mu_{rd1,a}} \cap \Sigma_{rd2} \cap \Sigma^{NB}$ (i.e., from benign objects in $\Sigma_{\mu_{rd1,a}} \cap \Sigma_{rd2}$). For any benign object $s_i \in \Sigma_{\mu_{rd1,a}} \cap \Sigma_{rd2}$, let m_1 and m_2 be the messages sent by s_i in μ_1 and μ_2 respectively. Since we know that $m_1.ts_i = m_2.ts_i = ret_{rd1}$ and since m_1 is sent before m_2, we have $m_1.updated \subseteq m_2.updated$. Hence, $\bigcap_{m \in \mu_1} m.updated \subseteq \bigcap_{m \in \mu_2} m.updated$. Since every benign base object which replies to r_2 in $rd2$, adds r_2 to its *updated* set before replying to r_2, $r_2 \in \bigcap_{m \in \mu_2} m.updated$.

Since $r_2 \notin \bigcap_{m \in \mu_1} m.updated$, we have $|\bigcap_{m \in \mu_2} m.updated| \geq |\bigcap_{m \in \mu_1} m.updated| + 1 \geq a + 1$. Since $|\Sigma_{\mu_{rd1,a}} \cap \Sigma_{rd2} \cap \Sigma^{NB}| \geq S - (a+1)t - ab$, the number of messages in μ_2 is at least $S - (a+1)t - ab$. Finally, since $a + 1 \leq R + 1$, $ret_{rd1} = ret_{rd2} + 1$ is admissible in $rd2$ with degree $a + 1$. Hence, the timestamp returned by $rd2$ is $ret_{rd1} = ret_{rd2} + 1$, a contradiction (with the assumption that $rd2$ returns ret_{rd2}).

$\langle 4 \rangle .2$ $r_2 \in \bigcap_{m \in \mu_1} m.updated$

Denote by X the set $\Sigma_{\mu_{rd1,a}} \cap \Sigma_{rd2} \cap \Sigma_{NB}$ and notice that all base objects in X are benign. By definition of μ_1, messages in μ_1 are sent by processes in X. By Lemma 38, $|X| = |\Sigma_{\mu_{rd1,a}} \cap \Sigma_{rd2}| \geq S - (a+1)t - ab$. Since $a \leq R$, it follows that $|X| > t + b$. Hence, by Lemma 39, $ret_{rd2} \geq maxTS_{rd1}$. A contradiction with $ret_{rd_2} = maxTS_{rd1} - 1$.

$\langle 3 \rangle .2$ $a = R + 1$

Since $|\{w, r_1, ..., r_R\}| = R + 1$ and $|\bigcap_{m \in \mu_{rd1,a}} m.updated| \geq a = R + 1$, we have $r_2 \in \bigcap_{m \in \mu_{rd1,a}} m.updated$. Observe that $|\Sigma_{\mu_{rd1,a}}| \geq S - at - (a-1)b > t + b$. Hence, $|\Sigma_{\mu_{rd1,a}} \cap \Sigma^{NB}| > t$ Replacing X with $\Sigma_{\mu_{rd1,a}} \cap \Sigma^{NB}$ in Lemma 39, it follows that $ret_{rd2} \geq maxTS_{rd1}$. A contradiction with $ret_{rd_2} = maxTS_{rd1} - 1$.

□

5.2 Lower Bound

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

Proposition 41. *Let $t \geq 1$, $b \geq 0$ and $R \geq 2$. If $(R+2)t + (R+1)b \geq S$, then there is no fast atomic storage implementation.*

Preliminaries.

In our proof, we suppose by contradiction that $(R+2)t + (R+1)b \geq S$ and that there is a fast implementation I of an atomic storage in the authenticated model. We construct a partial execution of I that violates atomicity: a partial execution in which some read returns 1 and a subsequent read returns an older value, namely, the initial storage value, $\perp$.

Recall first that w denotes the writer, r_i for $1 \leq i \leq R$ denote the readers, and s_i for $1 \leq i \leq S$ denote the base objects. Recall also, that we assume the authenticated model, i.e., that the writer writes digitally signed information to base objects, and that Byzantine base objects cannot forge the writers signature. Suppose by contradiction that $(R+2)t + (R+1)b \geq S$ and that there is fast implementation I of atomic storage. Given that $(R+2)t + (R+1)b \geq S$, we

can partition the set of base objects into $2R + 3$ subsets (which we call *blocks*), denoted by T_i $(1 \leq i \leq R + 2)$ and B_j $(1 \leq j \leq R + 1)$, such that each of the blocks T_i (resp., B_j) is of size less than or equal to t (resp., b).

Notice that our model does not require that a message sent by a faulty process is received by the receiver. Hence, in any (partial) execution, it is possible any message sent by a faulty process remains in transit (i.e., that such a message is never received by the receiver. In our proof we construct (partial) executions in which, unless explicitly stated otherwise, all messages sent by faulty processes are in transit.

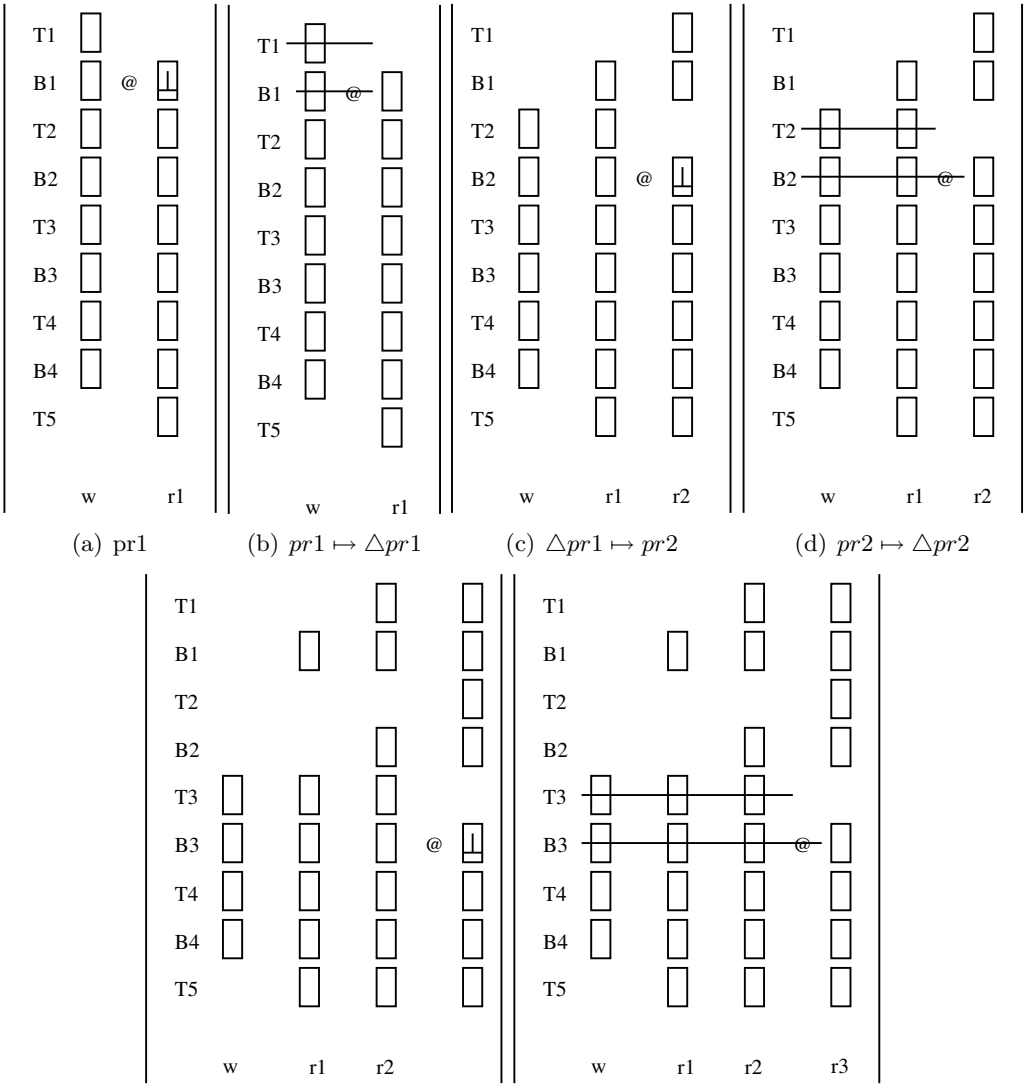
Similarly as in Chapter 4, we say that an *incomplete invocation inv skips* a set of blocks BS in a partial execution, where $BS \subseteq \{T_1, \ldots, T_{R+2}, B_1, \ldots B_{R+1}\}$, if (1) no base object in any block $BL_i \in BS$ receives any READ or WRITE message from *inv* in that partial execution, (2) all other base objects receive the READ or the WRITE message from *inv* and reply to that message, and (3) *all these reply messages are in transit.* We also say that a *complete invocation inv skips* a block BL_i in a partial execution, if (1) no base object in BL_i receives any READ or WRITE message from *inv* in that partial execution, (2) all base objects that are not in BL_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.*

To depict our proof, we use block diagrams similar to those used in previous chapters. We depict an invocation *inv* in a given partial execution through a set of rectangles arranged in a single column, where each row is attributed to a unique block of base objects. In the column corresponding to some invocation *inv*, we draw a rectangle in the respective row, if all base objects in block BL_i have received the READ or WRITE message from *inv* and have sent reply messages, i.e., we draw a rectangle in the respective row if *inv* does not skip BL_i. In partial execution pr_i, we denote a Byzantine failure of BL_i by "@".

We illustrate a particular instance of the proof in Figure 5.2 and Figure 5.3, where $R = 3$ and the set of base objects are partitioned into nine blocks, T_1 to T_5 and B_1 to B_4.

Proof. **Partial writes.** Consider a partial execution wr in which w completes write(1). The invocation skips T_{R+2}. We define a series of partial executions each of which can be extended to wr. Let wr_{R+2} be the partial execution 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 execution which contains an incomplete invocation write(1) that skips $\{T_{R+2}\} \cup \{T_j | 1 \leq j \leq i - 1\} \cup \{B_j | 1 \leq j \leq i - 1\}$. We make the following simple observations: (1) for $1 \leq i \leq R$, wr_i and wr_{i+1} differ only at base objects in $T_i \cup B_i$, (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.

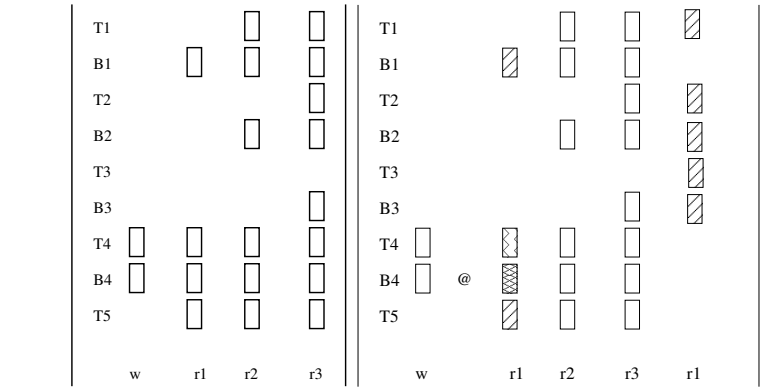
Appending reads. Partial execution pr_1 extends wr by having block B_1 failing



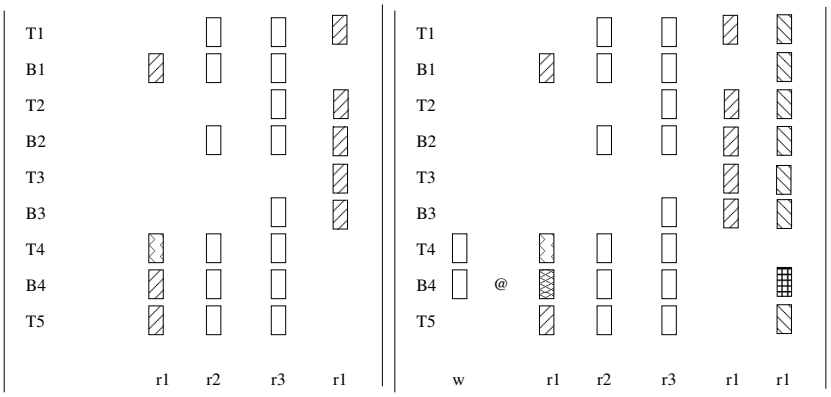
(a) pr1 (b) $pr1 \mapsto \triangle pr1$ (c) $\triangle pr1 \mapsto pr2$ (d) $pr2 \mapsto \triangle pr2$

(e) $\triangle pr2 \mapsto pr3$ (f) $pr3 \mapsto \triangle pr3$

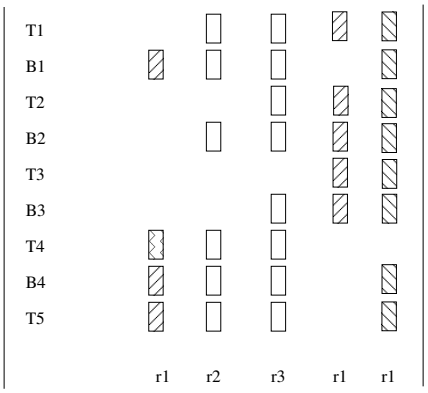
Figure 5.2: Partial executions pr_i and $\triangle pr_i$



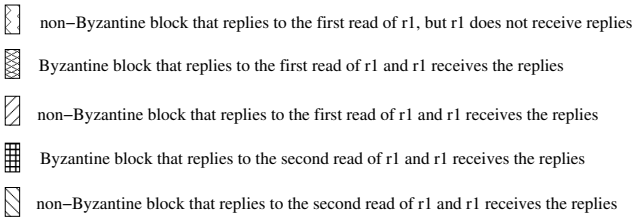
(a) $\triangle pr3$ (b) pr^A



(c) pr^B (d) pr^C



(e) pr^D



(f) Legend

Figure 5.3: Partial executions: pr^A, pr^B, pr^C and pr^D

upon completion of write(1) and appending a complete read by r_1 that skips block T_1 (Fig. 5.2(a)). B_1 fails in such a way that it behaves as if it never received any write message (i.e., a message from invocation write(1)). We say that B_1 fails and loses its memory. Observe that r_1 cannot distinguish pr_1 from some partial execution $\triangle pr_1$, that extends wr_2 by appending a complete read by r_1 that skips T_1. To see why, notice that (a) wr and wr_2 differ at w and at blocks T_1 and B_1, (b) r_1 does not receive any message from writer w and block T_1 in both executions and (c) r_1 received the same message from block B_1 in both executions. By wait-freedom property and since w can fail by crashing, r_1's read in $\triangle pr_1$ must return some value x, since it cannot wait for the completion of the writer's invocation, nor a message from w. Since r_1 cannot distinguish $\triangle pr_1$ from pr_1, r_1 returns the same value x in pr_1 as well, and by atomicity, in pr_1, x must equal 1. Therefore, in $\triangle pr_1$, r_1 also returns 1.

Starting from $\triangle pr_1$, we iteratively define the following partial executions for $2 \leq i \leq R$. Partial execution pr_i extends $\triangle pr_{i-1}$ by: (1) block B_i failing in such a way that it behaves as if it never received any message (loses memory) and (2) appending a complete read by r_i that skips T_i. Partial execution $\triangle pr_i$ is constructed by deleting from pr_i all steps of the base objects in block T_i and all steps of base objects in block B_i up to the instant in which B_i lost its memory (including that particular step). Since the last read in pr_i by reader r_i skips block T_i, r_i cannot distinguish pr_i from $\triangle pr_i$, as in both executions r_i receives the same messages from B_i. More precisely, partial execution $\triangle pr_i$ extends wr_{i+1} by appending the following i reads one after the other: for $1 \leq h \leq i-1$, r_h does a read that skips $\{T_j | h \leq j \leq i\} \cup \{B_j | h+1 \leq j \leq i\}$ and r_i does a (complete) read that skips T_i. Here, the first $i-1$ appended reads are incomplete whereas the last one is complete. Figure 5.2 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. By wait-freedom, in $\triangle pr_2$ r_2 must return some value, say x_2. However, since r_2 cannot distinguish pr_2 from $\triangle pr_2$, r_2 must return a value x_2 in pr_2 as well. Since pr_2 extends $\triangle pr_1$, by atomicity, r_2's read in pr_2 must return $x_2 = 1$. Therefore, 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 $2 \leq i \leq R$), in which it must return a value, it follows by trivial induction that r_i's read in $\triangle pr_i$ returns 1. In particular, r_R reads 1 in $\triangle pr_R$. Moreover, note that in $\triangle pr_R$ no object is faulty.

Partial execution pr^A. Consider again partial execution $\triangle pr_R$, i.e., partial execution wr_{R+1} extended by appending R reads by each reader r_h ($1 \leq h \leq R-1$) such that r_h's read skips $\{T_j | h \leq j \leq R-1\} \cup \{B_j | h+1 \leq j \leq R-1\}$, whereas a read by reader r_R skips T_R only. The read by r_1 is incomplete in $\triangle pr_R$: only base objects in B_1, T_{R+1}, B_{R+1} and T_{R+2} send replies to r_1, and those reply messages are in transit. Observe that, in $\triangle pr_R$, only base objects in T_{R+1} and B_{R+1} receive the WRITE message from write(1). Consider the following partial execution pr^A which differs from $\triangle pr_R$ in the following:

1. Upon reception of message from write(1) invocation, B_{R+1} fails arbitrarily

in such a way that, from that point on, it sends replies to all processes but r_1 as if it was not faulty, and to r_1 as if it never received a write(1) message. Moreover, after completion of read by r_R,

2. r_1 receives the READACK messages from T_{R+2} and B_1 (that were in transit in $\triangle pr_R$) and B_{R+1} (i.e., from the Byzantine faulty objects),

3. base objects in T_1 to T_R and B_2 to B_R receive the READ message from r_1 (that were in transit in $\triangle pr_R$) and reply to r_1, and

4. reader r_1 receives these replies from base objects in T_1 to T_R and B_2 to B_R, and then r_1 returns from the read invocation.

Notice that r_1 received replies from all blocks but T_{R+1}, and so, must return from the read; however, r_1 does not receive the replies from base objects in T_{R+1}, i.e., from the only benign base objects whose state was modified by write(1).

Partial execution pr^B. Consider another partial execution pr^B with the same communication pattern as pr^A, except that write(1) is not invoked at all and block B_{R+1} is not faulty. Hence, base objects in T_{R+1} do not receive any WRITE message (Figure 5.3). Clearly, only base objects in T_{R+1}, 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 executions because it does not receive any message from the base objects in T_{R+1}, the writer, or other readers and it receives the same message from the base objects in B_{R+1} in both executions. By atomicity, r_1's read returns (the initial storage value) $\perp$ 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 executions 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 executions: (1) pr^C is constructed by extending pr^A with another complete read by r_1, which skips T_{R+1} (as in pr^A, in pr^C B_{R+1} always replies to r_1 as if it never received any WRITE message), and (2) pr^D is constructed by extending pr^B with another complete read by r_1, which skips T_{R+1} (Figure 5.3). Since r_1 cannot distinguish pr^A from pr^B, and r_1's second read skips T_{R+1} (i.e., base objects 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 follows r_R's read which returns 1. Clearly, partial execution pr^C violates atomicity. □

6 ABsTRACT

6.1 Introduction

6.1.1 Motivation

In this chapter, we focus on the most general replication technique - state machine replication (SMR). SMR is a software technique for making any critical service (be it storage, consensus, or any other service) fault-tolerant, in which the critical service is modeled by a state machine. Several, possibly different, copies of the state machine are then placed on different nodes. Clients of the service access the replicas through a replication protocol which ensures that, despite concurrency and failures, replicas perform client requests in the same order.

Two objectives underly the design and implementation of a SMR algorithm: *robustness* and *performance*. Robustness conveys the ability to ensure availability (liveness) and one-copy semantics (safety) despite failures and asynchrony. The most robust SMR algorithms are those that tolerate (a) arbitrarily large periods of asynchrony, during which communication delays and process relative speeds are unbounded, (b) Byzantine failures of any client and (c) as many Byzantine replica failures as possible (optimal resilience). These are called Byzantine fault-tolerant state machine replication algorithms, or simply *BFT-SMR* algorithms.[1] The ultimate goal of a system designer is to build BFT-SMR algorithms that exhibit comparable performance to a non replicated server under "normal" circumstances that are considered the most frequent in practice. The notion of "normal" circumstance might depend on the application and underlying network but it usually means network synchrony, rare failures, and sometimes also the absence of contention among concurrent clients' requests. Examples of such BFT-SMR algorithms include PBFT, HQ, Q/U and Zyzyyva [AGG+05, CL99, CML+06, KAD+07].

Not surprisingly, even under the same notion of "normal" case, there is no "one size that fits all" BFT-SMR algorithm. Our experience reveals that the perfor-

[1] Frequently, these algorithms are simply referred to as BFT algorithms/protocols. In this thesis, we prefer to use the term BFT-SMR algorithms, to prevent confusion with other, more specialized, Byzantine fault tolerant algorithms, including consensus and storage algorithms given earlier in this thesis.

mance differences between BFT-SMR algorithms can be heavily impacted by the actual network, the size of the messages, the very nature of the "normal" case (e.g, contention or not); the actual number of clients, the total number of replicas as well the cost of the cryptographic libraries being used. This echoes some of the observations of [SDM+08]: *"For instance, PBFT offers more predictable performance and scales better with payload size compared to Zyzzyva; in contrast, Zyzzyva offers greater absolute throughput in wider-area, lossy networks"*. So determining the best existing algorithm seems clearly impossible. In fact, besides all BFT-SMR algorithms mentioned above, there are good reasons to believe that new algorithms could be designed that outperform all others under specific circumstances. As a matter of fact, in this Chapter, we do indeed present examples of such algorithms.

On the one hand, we might be tempted to devise a new BFT-SMR algorithm, or to modify an existing one, for each and every situation in order to achieve the best performance. But this can turn into a nightmare. All algorithm implementations we looked at involve around 20.000 lines of (non-trivial) C++ code, e.g., PBFT and Zyzzyva. Any change to an algorithm, although algorithmically intuitive, is extremely painful. In some cases, the changes of the algorithm needed to optimize for the "normal" case have strong impacts on the part of the algorithm used in other cases [KAD+07]. In other words, developing new and *maintaining* [ISO06] existing BFT-SMR algorithms is a very complex. Clearly, the size of BFT-SMR algorithms and the impossibility of exhaustive testing in distributed computing [CGR07] would rather plead for never changing an algorithm that revealed to be stable so far, e.g., PBFT.

6.1.2 Contributions

We propose in this thesis a way to have the cake and eat a big chunk of it. We present *A*BsTRACT — Abortable Byzantine faulT-toleRant stAte maChine replicaTion (simply written *Abstract*). As we already mentioned in the introduction in Chapter 1, *Abstract* significantly reduces the development cost of BFT-SMR algorithms and makes it significantly easier to develop efficient ones. *Abstract* looks like state machine replication and it can be used to make any shared service fault-tolerant, with one exception: it may sometimes *abort* a client request. The (*non-triviality*) condition under which *Abstract* cannot abort is a generic parameter. From this perspective, *Abstract* can be viewed as a *virtual type*; each specification of the non-triviality parameter defines a concrete type. [2]

Different instances of *Abstract* may be *composed* together; we expect many of such compositions to lead to interesting flexible BFT-SMR algorithms. *Abstract* instances are composed by using *unforgeable* histories; when a particular instance of *Abstract* aborts a client request, *Abstract* returns an unforgeable request history that can be used by the client to "recover" using another instance of *Abstract*.

To illustrate this composability, we present *Modular BFT-SMR*: a BFT-SMR algorithm built using two *Abstract* instances: (i) the first, which we denote *any Abstract*, would typically be an *Abstract* with a weak non-triviality condition that

[2] These genericity ideas date back to the seminal paper of Landin: *The Next 700 Programming Languages* (CACM, March 1966).

can be implemented very efficiently, whereas (ii) the second is a stronger *Abstract* (called *Backup*) with a non-triviality property that guarantees to commit a certain number of requests k; this can easily be implemented on top of *any* BFT-SMR algorithm (e.g., [CL99, AGG+05, CML+06, KAD+07]). Such a modular approach allows for "black-box" code reuse and can significantly reduce the development cost of new BFT-SMR algorithms, as well as maintenance of the existing ones (used as the basis for *Backup*).

We illustrate this concretely by exhibiting three specific *Abstract* instances with weak non-triviality and use them in *Modular BFT* in place of the *any Abstract* instance: (1) a "quorum"-based algorithm, (2) a "chain"-based algorithm, and (3) a "primary"-based algorithm that mimics Zyzzyva [KAD+07] in synchronous and failure-free executions. When implemented in C++, these algorithms require 6000, 4000 and 5000 lines of C++ code (respectively), which is, in average, less than 25% of the size of state of the art BFT-SMR algorithms like PBFT [CL99] or Zyzzyva.

While the third, Zyzzyva-like algorithm (called *AZyzzyva*), illustrates how our approach can help modularly derive algorithms that mimic existing ones, the first two algorithms are interesting in their own right. Namely, our "quorum"-based algorithm (called *Decentralized Abstract, or simply, DEC*)[3] outperforms all algorithms we know of in terms of time complexity (latency) when there is no concurrency, asynchrony or failures. Its C++ implementation improves the latency of Q/U [AGG+05] and Zyzzyva up to more than 33%. It implements the message exchange pattern used in optimistic executions of Q/U with two very important differences:

- First, our *DEC* algorithm requires only $3t+1$ servers to tolerate t Byzantine replica failures (optimal resilience), whereas Q/U requires $5t + 1$ servers to ensure a correct execution. To achieve this lower number of servers, we leverage our experience with refined quorum systems (see Chapter 3), which allows us to make use of class 1 quorums even with as few as $3t + 1$ servers. In addition, we combine this with the idea of speculative execution of requests [KAD+07]. Intuitively, the smaller number of servers *DEC* uses (in comparison to Q/U) brings itself a latency improvement in practice, since, in Q/U, it takes longer to wait for a roundtrip from a larger number of servers.

- Second, clients and replicas do not need to piggyback replica histories to requests and replies. Rather, they send Zyzzyva-style history digests, which significantly reduces the amount of data to be sent, as well as the number of message authentication codes (MAC) operations that are performed by clients and servers. Note that this latter improvement was intuited in [SDM+08].

[3] Decentralized algorithms, in which clients directly access servers implementing a particular service, are in literature often referred to as *quorum* algorithms (e.g., [CML+06, SDM+08]). This practice is somewhat misleading since quorums (as sets with particular intersection properties) are much wider a concept and underly all robust replication algorithms we know of.

On the other hand, our "chain"-based algorithm (called simply *Chain*) outperforms all algorithms we know of in terms of peak throughput in synchronous and failure-free executions. Its C++ implementation improves the performance of Zyzzyva and PBFT by up to 375%. It is the first BFT-SMR "chain"-based algorithm we know of. It implements a pipelining pattern for broadcasting client requests that is very similar to the one implemented in [vRS04]. Nevertheless, unlike in [vRS04], *Chain* tolerates the Byzantine failure of up to one third of the servers. This is realized using *chain authenticators* that prevent bypassing any process in the chain, using a fraction of message authentication codes (MACs) required by the MAC authenticators used in state of the art BFT-SMR algorithms. In short, a process in the chain uses a chain authenticator to authenticate the message (using MACs) for the next $t + 1$ processes in the chain.

Finally, besides facilitating the design of new BFT-SMR algorithms, our *Abstract* framework significantly simplifies their correctness proofs, thanks to the composability of *Abstracts*. We were able for instance to implement our *Modular BFT-SMR* in +CAL [Lam06a], and to model check its correctness (in small configurations).

6.1.3 Roadmap

The rest of the Chapter is organized as follows. First, in Section 6.2 we overview the related work. Then, after augmenting the system model of Chapter 2 in Section 6.3, we define *Abstract* and present our *Modular BFT-SMR* algorithm in Section 6.4. In Section 6.5 we give the implementations of the four *Abstract* instances: *DEC*, *Chain*, *AZyzzyva* and *Backup*. The proofs of correctness of our new *Abstract* implementations can be found is Section 6.6.

Finally, in Section 6.7, we evaluate the performance of two different *Modular BFT-SMR* C++ implementations obtained by substituting every instance of *aAbstract* in *Modular BFT-SMR* with *DEC* (resp., *Chain*) and by using *Backup* implemented over the PBFT algorithm [CL99].[4]

6.2 Related Work

Several BFT-SMR algorithms have been proposed during the last ten years: PBFT [CL99] was the first to propose the use of vectors of MACs rather than signature, which significantly improves performance. Then, Q/ U [AGG+05] proposed a decentralized algorithm only requiring one-phase in the fault and contention-free case. Q/U requires $5t + 1$ servers to tolerate t Byzantine server failures, whereas other BFT-SMR algorithms surveyed in this paragraph requires $3t + 1$ servers. HQ [CML+06] is another decentralized algorithm which, unlike Q/U, requires multiple rounds to complete a request (2 rounds in the best-case). Zyzzyva [KAD+07] used *speculation* to improve performance: replicas optimistically execute requests according to an order that is given by a primary server. Clients are in charge of detecting inconsistencies and helping servers converge

[4] The author is very grateful to Vivien Quéma, who wrote C++ implementations, ran the evaluation experiments and helped in interpreting evaluation results presented in Section 6.7.

to a correct state. Essentially, in Zyzzyva's speculative approach only clients are viewed as consensus learners; this is in contrast to previous BFT-SMR algorithms in which servers play all three consensus roles (i.e., in which servers are proposers, acceptors, as well as learners). As a result, Zyzzyva significantly improved the latency and throughput of all previous algorithms.

Several examples of composing an optimistic algorithm with a backup one was discussed in the survey of Pedone [Ped01]. In [PS98], for instance, an optimistic total order (atomic) broadcast algorithm is described: if messages are spontaneously totally ordered by the network, then they are delivered in one phase. Else, a consensus algorithm is used as a backup. The similar approach was pursued in [KS05], where the optimistic part of the atomic broadcast algorithm consists of Bracha broadcast [Bra84, BT85] (used implicitly in [CL99] as well) and where the backup algorithm relies on the (probabilistic) consensus algorithm. Unlike in this thesis, the optimistic parts in these algorithms are not modular and cannot easily be replaced with a different optimistic algorithm. The idea was also implicitly used in the context of Byzantine state machine replication in HQ [CML+06] and Zyzzyva [KAD+07]. In both of these cases, the algorithm consists of an optimistic phase and then the invocation of a second, recovery phase. Unlike in this thesis, the recovery phase is not encapsulated within a first class state machine replication abstraction; furthermore, the optimistic part is not encapsulated either. In Zyzzyva [KAD+07] for instance, speculation had a profound impact on the *view-change* (recovery) mechanism and this seemed to indicate that one could not fully decouple optimism from recovery.

The idea of aborting if "something goes wrong" is an old one in distributed computing. It underlies for instance the seminal two-phase commit algorithm [Gra78]: abort can be decided if there is a failure or some database server votes "no". The idea was also explored in the context of mutual exclusion: a process in the *entry section* can abort if it cannot enter the critical section [Jay03].

The idea of an abortable abstraction was proposed in the context of consensus in [Che07] and [BDFG03]. In the first case a process can abort if a majority of processes cannot be reached whereas, in the second, a process can abort if there is contention. The latter idea was generalized for arbitrary shared objects in [AGK05] and then [AFH+07]. In particular, in [AFH+07], a process can abort and then query the object to seek whether the last query of the process was performed. This query can however abort if there is contention.

Our notion of abortable state machine replication is different in two senses. First, the condition under which *Abstract* can abort is a generic parameter: it can express for instance contention, synchrony or failures. Second, in case of abort, *Abstract* returns (without any further query) what is needed for recovery in a Byzantine context; namely, an unforgeable history. This, in turn, can then be used to invoke another, possibly stronger, *Abstract*. This ability is key to the composability of *Abstract*.

Furthermore, non-abortable abstractions for deconstruction and modularization of BFT consensus and BFT-SMR algorithms have been proposed. For example, in the BFT-SMR modularization proposed in [Dou00], a BFT-SMR algorithm is viewed as a series of consensus instances, each reduced to the problem called *weak-interactive consistency* which is itself implemented using the abstrac-

tion of *muteness* failure detectors [DS97]. The ideas presented in [Dou00] are orthogonal to *Abstract*; muteness failure detectors could be for example used to implement *Abstract* instances with the strong non-triviality. On the other hand, and unlike *Abstract*, the framework of [Dou00] neither facilitates the design nor supports the composability of optimistic BFT-SMR algorithms. A similar conclusion can be drawn when our work is compared to that of [CKPS01], where an atomic broadcast algorithm is built modularly using the underlying abstraction of *multi-valued Byzantine agreement* with *external validity*.

Another related abstraction is optimistically terminating consensus (OTC) framework proposed in [Zie05, Zie06]. An OTC instance captures the notion of a round of communication and allows for deconstruction and reconstruction of (single-instance) BFT consensus algorithm. Unlike *Abstract*, OTC framework does not target specifically SMR algorithms which include multiple consensus instances. Moreover, this framework is latency-only oriented and reconstruction of the algorithm in OTC means matching the latency and the number of processes of the reconstructed algorithm. This is in contrast with our generic *Abstract* abstractions that is oblivious with respect to any particular complexity metric; indeed, in this thesis we propose both latency-efficient and throughout-efficient BFT-SMR algorithms based on *Abstract*.

6.3 System Model

In this Chapter, we assume the eventually synchronous, authenticated model, as defined in Chapter 2, complemented as follows.

We assume a distributed system with a fully connected network among processes: clients and servers (also called replicas). The links among processes are unreliable (before GST): messages may be delayed, dropped or duplicated (we speak of network, or link failures). However, we assume *fair-loss channels*, i.e., a message sent an infinite number of times between two correct processes is eventually received by the receiver (in other words, communication between 2 correct processes can de delayed only for finite periods of time). Moreover, we assume that any number of clients and less then one third of the servers can be Byzantine (i.e., our algorithms tolerate t server failures, using $3t + 1$ servers).

Furthermore, a process p can use vectors of MACs (called authenticators [CL99]) to authenticate the message m for multiple recipients belonging to the set S in a single physical message; we denote such a message, which contains a message m and the corresponding MAC for every process $q \in S$, by $\langle m \rangle_{\alpha_{p,S}}$. Moreover, we explicitly denote a message m sent by process p to process authenticated using a MAC by $\langle m \rangle_{\mu_{p,q}}$). We say that a MAC is *valid* if q can successfully verify a MAC and authenticate a message.

Finally, we assume ideal collision-resistant hash functions used to compute message digests. The digest function D maps a bit string to a short, unique representation; its invocation takes a bit string m as parameter and returns a value $d = D(m)$.

6.4 Abstract

We give in this section the specification of *Abstract* and describe how several *Abstract* instances can be combined in a modular way to implement an effective full-fledged BFT system. We give the *Abstract* specification both informally in plain English (for convenience) as well as a set of TLA+ [Lam02] predicates (for rigorousness). Similarly, we give both an informal explanation of our *Modular BFT-SMR* algorithm as well as its +CAL code.

For presentation simplicity, we first assume (in Sections 6.4.1– 6.4.3) that clients cannot be Byzantine (but can fail by crashing). We proceed through the following steps: (a) we first describe an *Abstract* specification that does not account for explicit initialization: we assume *Abstract* to be already initialized with an empty history (Sec. 6.4.1), (b) we then explain how to dynamically initialize an *Abstract* instance to enable composition (Sec. 6.4.2), (c) then, we give (one possible) implementation of a full-fledged BFT-SMR out of different *Abstract* boxes: we call the result of the composition *Modular BFT-SMR*.

In Section 6.4.4, we will explain the additional guarantees that *Abstract* must provide assuming clients can be Byzantine. These are mainly related to the unforgeability guarantees for certain messages returned by *Abstract* and are orthogonal to the *Abstract* properties we present beforehand. All our algorithms (presented in Section 6.5 and afterwards) provide these additional *Abstract* properties and tolerate Byzantine clients.

Still to simplify our specifications, we assume that the state of the replicated object is modeled by a history of requests. Such a history is also returned to the client by the replicas after a request, and can be used by the client to compute a reply. Of course, this can be in practice implemented by having the replicas compute themselves the reply and return it to the clients.

6.4.1 Abstract specification

Abstract (without initialization) is defined as follows:

Definition 11. ***(Abstract)***
Abstract *exports one operation:*

- $Invoke(m)$ — *we say the client invokes the request* m.

Abstract *returns two indications:*

- $Commit(m, h)$, *and*
- $Abort(m, h)$.

We say the client commits *(resp.,* aborts*) the request* m *with history* h*, where* h *is a sequence of requests that the client can use to compute a reply (resp., to recover). If the client commits (resp., aborts)* m *with history* h*, we refer to* h *as the* commit history *(resp.,* abort history*).*

Abstract *ensures the following properties.*

1. (Termination) *If a correct client invokes a request m, it eventually commits or aborts m with h, and h contains m.*

2. (Commit Ordering) *Let h and h' be any two commit histories: either h is a prefix of h' or vice versa.*

3. (Abort Ordering) *Every commit history is a prefix of every abort history.*

4. (Validity) *In every commit/abort history h, no request appears twice and every request was invoked by some client.*

5. (Non-Triviality) *If a correct client invokes a request m and some predicate NT is satisfied, the client commits m.*

The Non-Triviality property is generic; the undefined predicate NT may vary depending on the design goals and the environment in which a particular *Abstract* implementation is to be deployed.

The property that defines the behavior of *Abstract* in the case of *abort* is *Abort Ordering*. Intuitively, *Abstract* returns histories that represent the ordering of the clients requests. In case of a commit, this ordering is definitive and the reply of the implemented object is uniquely determined by the order of the requests in the history. This is not the case with the abort history. As specified by Abort Ordering, every abort history contains every commit history as its (non-strict) prefix; i.e., Abort Ordering prevents any new request, invoked after some request is aborted, from being committed.

6.4.2 Abstract initialization and composition

An instance of *Abstract* could be used on its own. However, a particular *Abstract* becomes practically useless after aborting even a single request, since it must also abort every subsequent request. Therefore, an *Abstract* instance is much more interesting when composed with other *Abstract* instances.

Hereafter, we consider multiple *Abstract* instances, combined to work for a "common good", with the ultimate goal of producing a flexible full-fledged BFT-SMR algorithm that can benefit from good performance under different scenarios. We assume a fixed, predetermined, ordering among *Abstract* instances, known by all processes in the system. This ordering is used by clients to know which *Abstract* instance I' to invoke after aborting from a specific *Abstract* instance I; we talk about a client *switching* from *Abstract* instance I to I'. We call I the *preceding Abstract* for I' (resp., I' is the *succeeding Abstract* for I).

Clients use abort histories received from the preceding *Abstract* to invoke a special INIT request, used to *initialize* a specific instance of *Abstract* (Fig. 6.1). INIT request invocations have the form $Invoke(m, h_I)$, where m is the request aborted by the preceding *Abstract* I, and where h_I is the corresponding abort history; in the context of the INIT request invocation, h_I is called *init history*.

We enhance *Abstract* properties of Definition 11 to account for INIT requests by: (a) slightly modifying the notion of an *invoked request* in the Validity property to include any request contained in an init history h_I of any INIT request invocation $Invoke(m, h_I)$, and (b) by adding the following property:

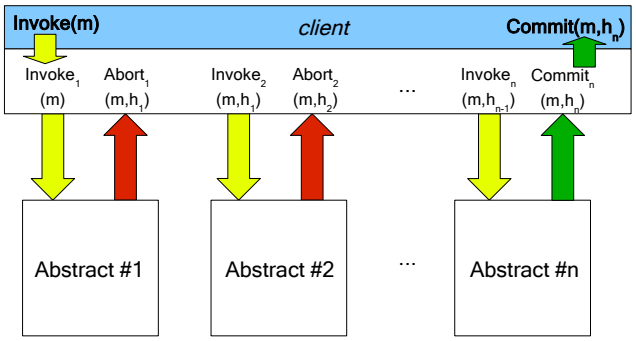


Figure 6.1: Composition of *Abstract* instances: an abort history of the preceding *Abstract* is used as an init history for a subsequent *Abstract*.

6. *(Init Ordering)* Any common prefix of init histories of all invoked INIT requests is a prefix of any commit or abort history.

Moreover, client invocations must be *well-formed*, meaning that before invoking a "non-INIT" request, correct client c must invoke an INIT request (recall that we discuss Byzantine clients in Section 6.4.4). Notice that an INIT request may be committed or aborted (according to the specification of a particular *Abstract*), just like any other request.

6.4.3 Building BFT-SMR using Abstract(s)

In the following, we describe a possible BFT-SMR implementation using multiple *Abstract* boxes called *Modular BFT-SMR*. *Modular BFT-SMR* algorithm uses:

1. *any Abstract* (simply referred to as *aAbstract*). Typically, *aAbstract* would be an *Abstract* with *weak* Non-Triviality, that can be implemented very efficiently; we refer to such an *Abstract* as the *weak* Abstract. Roughly, weak Non-Triviality means here that the notion of "something going wrong" applies to many executions; the strongest Non-Triviality property would be the one in which "nothing ever goes wrong" (such an *Abstract* is precisely BFT-SMR). We will come back to efficient implementations of weak *Abstract* in Section 6.5.

2. an *Abstract* (called *Backup*) with a strong Non-Triviality property that guarantees that at least $k \geq 1$ requests will be committed by any instance of *Backup* (where k is generic). *Backup* can be implemented over any BFT-SMR algorithm, as explained in Section 6.5.4. In this sense, *Modular BFT-SMR* can be seen as a BFT-SMR self-implementation (i.e., *Modular BFT-SMR* is a BFT-SMR algorithm implemented using some (any) other BFT-SMR algorithm).

Modular BFT-SMR uses multiple instances of both *aAbstract* and *Backup*, switching between them alternatively. Namely, according to a deterministic order

of *Abstract* instances, every odd (resp., even) instance is that of *aAbstract* (resp., *Backup*).[5] The result of the composition is a full-fledged BFT-SMR algorithm with the important feature that *Backup* is never invoked as long as the Non-Triviality conditions of *aAbstract* are satisfied.

On invoking *Modular BFT-SMR*, clients first invoke an instance of *aAbstract*. The first instance of *aAbstract* does not need to be initialized with special INIT requests. If the client obtains a commit indication from *aAbstract* then it returns the reply to the BFT-SMR client with the commit history it obtained from *aAbstract* (see Fig. 6.2(a)). Else, if the client's request aborts (in the i^{th} instance of *aAbstract*), the client switches to the i^{th} instance of *Backup* using the obtained abort history (from the i^{th} instance of *aAbstract*) as the init history. It is important to notice here that, as long as the conditions of the *aAbstract* Non-Triviality are satisfied, *Backup* is never invoked.

In *Modular BFT-SMR*, the i^{th} instance of *Backup* is configured to commit exactly $k = SP(i)$ requests, where $SP()$ is a *switching policy* function that maps the set of natural numbers into the set of natural numbers (including ∞). The switching policy can be tuned to allow reuse of *aAbstract* (Fig. 6.2(c)). Intuitively, if $SP(1) = \infty$ then all requests aborted by *aAbstract* will be committed by the first instance of *Backup*. However, in case some transient bad conditions caused *aAbstract* to abort it would be desirable to *switch-back* to *aAbstract*, in order to benefit from its performance. To this end, $SP()$ function can be tuned to reflect different switching policies.

Finally, in *Modular BFT-SMR* a client always invokes the instance of *aAbstract* or *Backup* that committed client's last request. In a sense, a client does not need to invoke all the time the entire sequence of *Abstract* instances to find a "working" one.

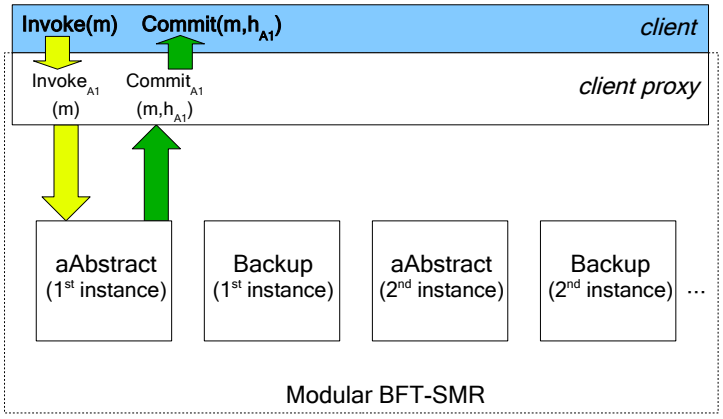
Model checking Modular BFT-SMR

We argue for the correctness of our *Modular BFT-SMR* construction by specifying it in +CAL/TLA+ and by model-checking it using the TLC model checker [Lam02].

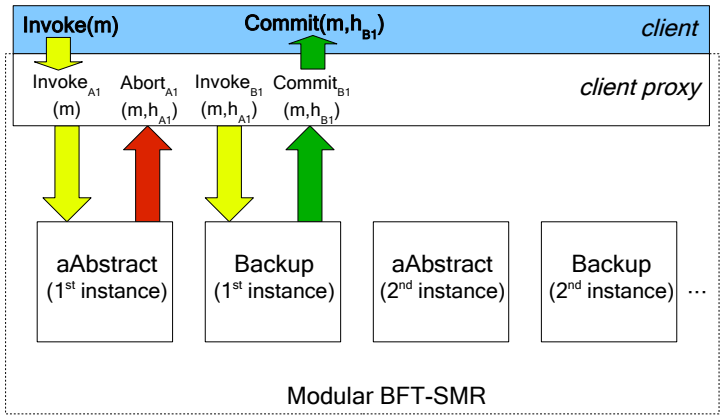
The +CAL pseudocode of our *Modular BFT-SMR* algorithm along with the TLA+ predicates that specify *Abstract* is given in Figures 6.3 and 6.4. The TLA+ predicates (Fig. 6.3) consist of *Abstract* safety predicates (lines 12–21,Fig. 6.3), as well as some auxiliary predicates used either in *Abstract* safety predicates or in the main +CAL code given in Figure 6.4.

To simplify the distinction among clients request, our +CAL code of Figure 6.4 uses the centralized counter $UniqueReqID$; in practical *Abstract* implementations, this distinction is made using local timestamps at clients and their IDs (see Section 6.5). Moreover, to reduce the number of +CAL labels and generated states in the TLC model checker, the switching policy function for *Backup* is slightly different; the i^{th} instance of *Backup* commits exactly $SP(i) - SP(i-1)$ requests (where $SP(0) = 0$), instead of $SP(i)$ requests as described previously. These simplifications do not impact the correctness of our *Modular BFT-SMR*.

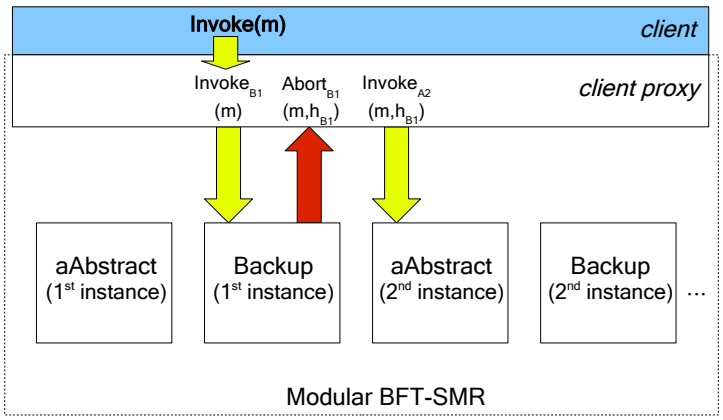
[5] It is important to notice that different instances of *aAbstract* and *Backup* can be implemented over the same set of processes.



(a) If *aAbstract* Non-Triviality conditions are satisfied, clients access only the *aAbstract* module, which (in this case) behaves like a full-fledged SMR.



(b) If *aAbstract* aborts, clients use the abort history as init history to switch to *Backup*. *Backup* is a powerful *Abstract* that guarantees to commit a certain number of requests.



(c) After a certain number of requests is committed within *Backup*, a client may be switched-back to try again (some) *aAbstract*.

Figure 6.2: Modular BFT-SMR.

In the configuration with 2 clients, 2 requests per client, each instance of *Backup* committing 1 request, TLC v2.0 finds 14107516 distinct states with the depth of the complete state graph search of 61. It takes 1h 8min 1s to model check the algorithm and verify its correctness on a 4xdual core Opteron 8216 with 8 GB RAM. The absence of deadlock during the model check asserts BFT-SMR Termination property in a given configuration. BFT-SMR Termination in any configuration is satisfied with any switching policy function $SP()$ in case we consider only executions with an arbitrarily large, yet finite number of requests; in the worst-case, all requests will be committed by (some instance of) *Backup*. In case the number of requests can be infinite, Termination can be guaranteed by configuring the N^{th} instance of *Backup* to commit an infinite number of requests, for an arbitrary natural number N.

001: ——— MODULE ModularBFTSMR ———
002: EXTENDS Integers, TLC, FiniteSets, Sequences
003: CONSTANT aAbstract, Backup, Client, TotReq, MaxReq, SP(_), NoOfInst

004: $Fresh \triangleq [req : TotReq]$
005: $getFreshReq(m) \triangleq$ IF $m \in Fresh$ THEN m ELSE $getFreshReq(m.req)$

006: $prefix(m, h) \triangleq (Len(m) \leq Len(h)) \wedge (\forall i \in 1 \ldots Len(m) : m[i] = h[i])$
007: $SetPermutations(len, S) \triangleq \{f \in [1..len \rightarrow S] : \forall v, w \in 1..len : v = w \vee f[w] \neq f[v]\}$
008: $Set2Sequence(len, S) \triangleq$ IF $len = 0$ THEN $\{\}$
ELSE $SetPermutations(len, S) \cup Set2Sequence(len - 1, S)$
009: $InitHistories(inv) \triangleq$ IF $\forall mm \in inv : mm \in Fresh \vee mm.type \neq "INIT"$ THEN $\{\}$
ELSE LET $init \triangleq$ CHOOSE $m \in inv : m \notin Fresh \wedge m.type = "INIT"$
IN $\{init.history\} \cup InitHistories(inv \setminus \{init\})$
010: $Tails(inv) \triangleq$ IF $inv = \{\}$ THEN $\{\}$
ELSE LET $h \triangleq$ CHOOSE $m \in inv : m \in inv$ IN $\{Tail(h)\} \cup Tails(inv \setminus \{h\})$
011: $LCPrefix(inv) \triangleq$ IF $inv = \{\}$ THEN $\langle\rangle$
ELSE LET $hist \triangleq$ CHOOSE $m \in inv : \forall mm \in inv : Len(m) \leq Len(mm)$
IN IF $(hist = \langle\rangle) \vee (\exists m1 \in inv : Head(hist) \neq Head(m1))$ THEN $\langle\rangle$
ELSE $\langle Head(hist)\rangle \circ LCPrefix(Tails(inv))$

012: $CommitOrdering(h, ex) \triangleq$
$\forall m \in ex : m.type = "COMMIT" \Rightarrow prefix(m.history, h) \vee prefix(h, m.history)$
013: $AbortOrdering(h, ex) \triangleq \forall m \in ex : m.type = "COMMIT" \Rightarrow prefix(m.history, h)$
014: $AbortOK(h, ex) \triangleq \forall m \in ex : m.type = "ABORT" \Rightarrow prefix(h, m.history)$
015: $InitOrdering(h, inv) \triangleq prefix(LCPrefix(InitHistories(inv)), h)$
016: $ValidHistory(inv) \triangleq Set2Sequence(Cardinality(inv), inv)$
017: $Validity(h, inv) \triangleq \forall i \in 1..Len(h) : \exists j \in inv : h[i] = j \wedge (\forall k \in i + 1..Len(h) : h[i] \neq h[k])$
018: $Exists(req, h) \triangleq \exists i \in 1..Len(h) : h[i] = req$
019: $Reqs(inv) \triangleq \{req \in Fresh : \exists mm \in inv :$
$(getFreshReq(mm) = req) \vee (mm \notin Fresh \wedge mm.type = "INIT" \wedge Exists(req, mm.history))\}$
020: $validCommitHistories(newReq, inv, ex) \triangleq \{h \in ValidHistory(Reqs(inv)) :$
$\bigwedge Exists(getFreshReq(newReq), h) \wedge CommitOrdering(h, ex)$
$\bigwedge AbortOK(h, ex) \wedge InitOrdering(h, inv)\}$
021: $validAbortHistories(newReq, inv, ex) \triangleq \{h \in ValidHistory(Reqs(inv)) :$
$Exists(getFreshReq(newReq), h) \wedge AbortOrdering(h, ex) \wedge InitOrdering(h, inv)\}$

022:$NewReq(inv, ex) \triangleq \{m \in inv : (\forall mm \in ex : m \neq mm.req)\}$
023:$Response(m, ex) \triangleq \{mm \in ex : m = mm.req\}$

Figure 6.3: Modular BFT-SMR: TLA+ predicates

```
024: (* - -algorithm ModularBFTSMR {
025:   variables
026:     Inv = [i ∈ {aAbstract, Backup} ↦ [j ∈ 1..NoOfInst ↦ {}]];
027:     Exec = [i ∈ {aAbstract, Backup} ↦ [j ∈ 1..NoOfInst ↦ {}]];
028:     finished = [i ∈ Client ↦ FALSE];
029:     InvBFT = {}; ExecBFT = {}; UniqueReqID = {};

030:   macro Invoke(box, inst, m){
031:     Inv[box][inst] := Inv[box][inst] ∪ {m}}
032:   macro Return(box, inst, m){
033:     Exec[box][inst] := Exec[box][inst] ∪ {m}}
034:   macro next(box, inst){
035:     if (box = aAbstract) box := Backup
036:     else {box := aAbstract; inst := inst + 1}}

037:   process (clnt ∈ Client)
038:     variables curReq; absReq; localts = 0; lastbox = aAbstract;
039:               lastInst = 1;
040:     {C: while (localts < MaxReq){
041:           localts := localts + 1;
042:           UniqueReqID := UniqueReqID + 1;
043:           curReq := [req ↦ UniqueReqID];
044:           absReq := curReq;
045:           InvBFT := InvBFT ∪ {curReq};
046:cloop:    while (Response(curReq, ExecBFT) = {}){
047:             Invoke(lastbox, lastInst, absReq);
048:Return:      with (m ∈ Response(absReq, Exec[lastbox][lastInst]))
049:               if (m.type = "COMMIT"){
050:                 ExecBFT := ExecBFT ∪ {[type ↦ "COMMIT",
051:                                        req ↦ curReq,
052:                                        history ↦ m.history]};
053:                 assert(⋀ Validity(m.history, InvBFT)
054:                        ⋀ Exists(curReq, m.history)
055:                        ⋀ CommitOrdering(m.history, ExecBFT))}
056:               else {next(lastbox, lastInst);
057:                     absReq := [type ↦ "INIT", req ↦ m.req, history ↦ m.history]}}};
058:         finished[self] := TRUE}


059: process (abs ∈ {aAbstract, Backup})
060:   variables cnt = 0; \* used only by Backup
061:   {A: while (∃c ∈ Client : finished[c] = FALSE){
062:         with (inst ∈ 1..NoOfInst;
063:               newReq ∈ NewReq(Inv[self][inst], Exec[self][inst]))
064:           either {when (⋁(self = aAbstract);
065:                         ⋁(cnt < SP(inst) ∧ self = Backup));
066:                   with (hist ∈ validCommitHistories(newReq,
067:                                                     Inv[self][inst],
068:                                                     Exec[self][inst]))
069:                     Return(self, inst, [type ↦ "COMMIT",
070:                                         req ↦ newReq,
071:                                         history ↦ hist]);
072:                   cnt := cnt + 1}
073:           or {when (⋁(self = aAbstract);
074:                     ⋁(cnt ≥ SP(inst) ∧ self = Backup));
075:       with (hist ∈ validAbortHistories(newReq,
076:                                        Inv[self][inst],
077:                                        Exec[self][inst]))
078:         Return(self, inst, [type ↦ "ABORT",
079:                             req ↦ newReq,
080:                             history ↦ hist])}}}}*)
```

Figure 6.4: Modular BFT-SMR: +CAL code

6.4.4 Byzantine clients

So far, we presented *Abstract* and *Modular BFT-SMR* assuming that no client is Byzantine. In a real environment, where clients can be Byzantine, we need to enforce *Abstract* with some additional properties.

Abstract composability (like in *Modular BFT-SMR*) depends on clients using abort histories generated by the preceding *Abstract* to initialize the particular instance of *Abstract*. This can be ignored by Byzantine clients, who may generate their own INIT request with forged init histories to corrupt the service. We thus assume that *Abstract* can produce *unforgeable* abort histories (an unforgeable abort history cannot be generated by an adversary) and require all *Abstract* abort histories to be unforgeable. In practice (see Section 6.5), the *Abstract* abort histories are implemented using digital signatures, as well as message digests.

Moreover, we assume every *Abstract* to be able to verify the integrity of an abort history returned by the preceding *Abstract*. An init history of an *Abstract* INIT request is said to be *valid* if it is indeed an unforgeable abort history of the preceding *Abstract*.

Finally, Byzantine clients' invocations might not be well-formed, i.e., Byzantine clients cannot be expected to first invoke an INIT request on a given *Abstract*. Despite this, a Byzantine-fault tolerant *Abstract* must provide correct service to correct clients. However, we formally require *Abstract Validity* to hold only in the case in which no client is Byzantine, since it is difficult to establish the meaning of an "invoked request" when the invoking client is Byzantine.

6.5 Abstract implementations

We gave in the last section the specification of *Abstract* and explained how, using different *Abstract* instances, one can implement a full-fledged BFT-SMR. In this section, we first show how to efficiently implement *Abstract* instances with weak Non-Triviality by describing two novel implementations *DEC* and *Chain*, as well as *AZyzzyva*, an *Abstract* implementation that mimics Zyzzyva [KAD+07] when the system is synchronous and there are no failures. Then, we show how to implement *Backup* using (any) BFT-SMR.

Our efficient *Abstract* implementations, *DEC* and *Chain* are interesting in their own right. *DEC* has the lowest latency among all BFT-SMR algorithms we know of, when the system is synchronous and there are no server failures or contention. In short, (no) contention means that (no) two requests are concurrently pending (i.e., invoked but not committed/aborted). On the other hand, *Chain* has the highest peak throughput when the system is synchronous and there are no server failures. Both *DEC* and *Chain* are *Abstract* implementations over a set of $n = 3t+1$ servers (denoted hereafter by Σ), out of which t can be Byzantine (i.e., controlled by an adversary). Finally, a very important feature of *DEC* and *Chain* is that they do not use signatures (which introduce large overhead) for authentication, but rather MAC authenticators, as long as there are no failures and the system is synchronous (and, in the case of *DEC*, when there is no contention).

In the rest of this section, we present first *DEC* in Section 6.5.1 and then *Chain* in Section 6.5.2 (for pedagogical reasons, since *DEC* is simpler of the two). We

present both *DEC* and *Chain* in the vein of Zyzzyva [KAD+07], by following a client's request throughout the algorithm. This is followed by the presentation of *AZyzzyva* (Sec. 6.5.3) which we obtain by modifying *Chain*. Then, in Section 6.5.4, we show how to simply implement *Backup* using any BFT-SMR. For better readability, the correctness proofs of *DEC* and *Chain* (correctness of our *Backup* implementation using any BFT-SMR is straightforward) are postponed to Section 6.6.

6.5.1 Decentralized Abstract (DEC)

As we mentioned earlier, *DEC* is an *Abstract* implementation over a subset of $3t + 1$ servers. *DEC* uses a simple refined quorum system (Section 3), in which the set that contains all servers is a class 1 quorum, whereas all sets that contain $2t + 1$ servers are ordinary quorums. Strictly speaking, in this case, quorums that contain $2t + 1$ servers are class 2 quorums (see Example 6, Section 3.2.2); however, to keep *DEC* simple, we are not interested in graceful degradation in time complexity when a class 1 quorum cannot be accessed. We require *DEC* to achieve best possible latency when a class 1 quorum is available.

DEC is an implementation of *Abstract* with the following Non-Triviality property:

(DEC Non-Triviality) If (a) a correct client c invokes request m, (b) there are no server failures, (c) the system is synchronous and (d) there is no contention, then client c commits m.

DEC is very simple; it consists of only two round-trips of message exchange between a client and servers. Importantly, in case the conditions in the *DEC* Non-Triviality property are satisfied, only the first round-trip is executed. If these conditions are not satisfied, the second round-trip might be executed; in the second round-trip the client is provided with an abort history necessary for recovery using some stronger *Abstract*.

t_c - local timestamp at client c
o - operation invoked by the client
c/s_i - client (resp., server) ID
LH_i - a local history at server s_i
$ts_i[c]$ - the highest t_c seen by server s_i

Figure 6.5: *DEC* message fields and process local variables.

As in related algorithms (e.g., [CL99, KAD+07]), to help distinguish clients' requests for the same operations, we assume that client c, on requesting an execution of operation o, invokes *DEC*[6] by executing $Invoke(req)$, where $req = \langle o, t_c, c \rangle$ and where t_c is a unique, monotonically increasing client's timestamp (the message fields and process local variables we use are explained in Fig. 6.5). In the

[6]We extend this assumption to *Chain* and *AZyzzyva* as well.

following description, the algorithmic part marked with Handling INIT Requests is relevant only if a particular DEC needs to be initialized.

1. Client sends a request to all servers.

Client c, on invoking $req = \langle o, t_c, c\rangle$, sends message $\langle \mathsf{REQ}, req\rangle_{\mu_{c,s_i}}$ to every server s_i. Upon sending REQ messages, the client triggers a timer T_{DEC} (set to 2Δ). We assume that correct client c does not invoke another request before it commits/aborts req.

Handling INIT Requests. The first message sent by the client must also contain an INIT tag and a valid init history IH (recall that a valid init history is an unforgeable abort history from the preceding Abstract).

2. Server receives the request, appends it to its local history (i.e., executes it speculatively) and sends the resulting history to the client.

Server s_i on receiving $\langle \mathsf{REQ}, req\rangle_{\mu_{c,s_i}}$, if $req.t_c$ is higher than $ts_i[c]$, (i) updates $ts_i[c]$ to $req.t_c$, (ii) appends req to its local history LH_i and (iii) sends $\langle \mathsf{ACK}, LH_i, req.t_c, s_i\rangle_{\mu_{s_i,c}}$ to c.

In fact, and in order to improve performance, only one designated server (say s_1) sends its full local history to the client; other servers send an ACK message containing a digest of of their local history LH_i, $D(LH_i)$.

Handling INIT Requests. If the local history of s_i is empty, s_i may only execute the INIT request with a valid init history IH. More precisely, s_i executes the entire IH, by appending the entire IH (instead only req) to its (empty) history LH_i (we say s_i *initializes* its local history). All (non-INIT) requests received before are discarded. If LH_i is not empty, s_i neglects IH and executes only req.

3. Client gathers matching server responses.

Client c gathers matching server responses, until the timer T_{DEC} (set in Step 1) expires, or until c collects $3t+1$ matching responses from different servers (i.e., until c collects matching responses from a class 1 quorum). A matching response for req is an ACK message (with a valid MAC) containing $req.t_c$.

3a. Client receives $3t + 1$ matching responses (i.e., from a class 1 quorum) with identical histories and commits.

If client c receives $3t + 1$ $\langle \mathsf{ACK}, LH, req.t_c, *\rangle$ messages from different servers, with identical history LH (in fact, with the identical history digests equal to $D(LH)$), then the client commits the request by returning $Commit(req, LH)$, i.e., LH is the DEC commit history for request req.

3b. Client *does not* receive $3t + 1$ matching responses with identical histories and panics.

If client c does not receive $3t + 1$ matching responses by the expiration of the timer T_{DEC}), the client panics, i.e., it sends a $\langle \mathsf{PANIC}, req\rangle_{\mu_{c,s_i}}$ to every server s_i.

Notice that, since a PANIC message may be lost, we assume that the client periodically re-sends the PANIC message to servers, until it commits or aborts the request.

3b.1. Server receives a panic message, stops accepting new requests and sends an abort message containing a signed local history to the client.

Server s_i, on receiving a $\langle \mathsf{PANIC}, req \rangle_{\mu_{c,s_i}}$ message, stops accepting new REQ messages (i.e., stops executing Step 2) and sends $\langle \mathsf{ABORT}, LH_i, req.t_c, s_i \rangle_{\sigma_{s_i}}$ message to c.

Handling INIT Requests. If the history LH_i of server s_i is empty, s_i acts in this step only on a PANIC message for an INIT request (with a valid init history IH). In this case, upon receiving the first such PANIC message, s_i, before sending an ABORT message sets LH_i to IH (i.e., s_i initializes its local history). The following PANIC messages for INIT requests are treated as described above, neglecting init histories.

3b.2. Client receives $2t+1$ matching ABORT messages (i.e., from any quorum), extracts the abort history and aborts the request.

A matching ABORT message for a $\langle \mathsf{PANIC}, req \rangle$ is any ABORT message containing $req.t_c$. When the client receives a matching ABORT message from $2t+1$ different servers, the client extracts the abort history AH in the following way:

- (i) the client generates the history AH_1 such that $AH_1[j]$ equals the value that appears at position $j \geq 1$ of $t+1$ different histories LH_i received in ABORT messages. If such a value does not exists for a position k then AH_1 does not contain a value at position k or higher,
- (ii) the longest prefix AH_2 of AH_1 is selected such that no request appears in AH_2 twice,
- (iii) if $req = \langle o, t_c, c \rangle$ does not exists in AH_2, it is appended to AH_2. The resulting sequence of requests is an abort history AH.

Then, c aborts req by returning $Abort(req, AH)$. AH must always be accompanied with the set of $2t+1$ ABORT messages signed by the servers (these signatured make AH unforgeable).

Notice that a Byzantine client could always force *DEC* to abort all requests by sending PANIC messages prematurely. However, a Byzantine client cannot be prevented from constantly sending perfectly regular requests (hence creating contention and forcing *DEC* to abort). Therefore, it is reasonable to assume that the existence of Byzantine client(s) implies contention. Hence, for simplicity, we choose not to include the solution to premature panicking in the description of *DEC*, since *DEC* is not required to commit requests under contention; we however include this solution in *Chain* (that should commit requests despite contention).

```
001: ---------- MODULE DEC ----------
002: EXTENDS Integers, TLC, FiniteSets, Sequences
003: CONSTANT Server, Client, Operation, Req, MaxReq

004: Reqs ≜ [type : {"REQ"}, c : Client, tc : Req, o : Operation]


005: validReq(m, ts) ≜ (m.type = "REQ") ∧ (m.c ∈ Client) ∧ (m.tc > ts[m.c])
006: validPanic(m, ts, sent, srv) ≜ ⋀(m.type = "PANIC" ∧ m.c ∈ Client ∧ m.tc ≥ ts[m.c])
                                   ⋀¬(∃m1 ∈ sent : (⋀ m1.type = "ABORT"
                                                   ⋀ m1.server = srv
                                                   ⋀ m1.tc = m.tc
                                                   ⋀ m1.c = m.c))

007: Clean(pos, MM) ≜ {m ∈ MM : Len(m.history) ≥ pos}
008: getAbortHistory(pos, MM) ≜
       IF (∃req ∈ Reqs :Cardinality({m ∈ Clean(pos, MM) : m.history[pos] = req}) >
                        Cardinality(Server) ÷ 3)
       THEN LET req1 ≜ CHOOSE req ∈ Reqs :
                  Cardinality({m ∈ Clean(pos, MM) : m.history[pos] = req}) >
                  Cardinality(Server) ÷ 3
            IN ⟨req1⟩ ∘ getAbortHistory(pos + 1, MM)
       ELSE ⟨⟩

009: Valid(h) ≜ ∀i ∈ 1..Len(h) : ∀k ∈ i + 1..Len(h) : h[i] ≠ h[k]
010: SetHist(h) ≜ IF h = ⟨⟩ THEN {h} ELSE {h} ∪ SetHist(SubSeq(h, 1, Len(h) − 1))
011: LVP(h) ≜ CHOOSE m ∈ SetHist(h) :⋀Valid(m)
                                    ⋀(∀n ∈ SetHist(h) : Valid(n) ⇒ Len(m) ≥ Len(n))
```

Figure 6.6: DEC +CAL code: TLA+ predicates

```
012: (* - -algorithm DEC {
013:   variables msgs = {}; stop = FALSE;: Invoked = {}; Returned = {}; finished = [i ∈ Client ↦ FALSE];
014:   macro Send(m) {msgs := msgs ∪ {m}}

015:   process (clnt ∈ Client)
016:     variables localts = 0; invokedOp = 0; panic = FALSE;{
017:C:   while (localts < MaxReq) {
018:       localts := localts + 1;
019:       with (op ∈ Operation) {
020:         Send([type ↦ "REQ", c ↦ self, tc ↦ localts, o ↦ op]);
021:         Invoked := Invoked ∪ {[type ↦ "REQ", c ↦ self, tc ↦ localts, o ↦ op]};
022:         invokedOp := op}
023:cl:  while (invokedOp ≠ 0) {
024:       either {with (M = {m ∈ msgs : (m.type = "ACK") ∧ (m.c = self) ∧ (m.tc = localts)};
025:                     mmsg = {m ∈ M : ∀m2 ∈ M : (m.history = m2.history)};
026:                     A = {m.server : m ∈ mmsg};
027:                     m1 ∈ mmsg) {
028:                       when A = Server;
029:                       Returned := Returned ∪ {[type ↦ "COMMIT",
                                                    req ↦ [type ↦ "REQ",
                                                           c ↦ self,
                                                           tc ↦ localts,
                                                           o ↦ invokedOp],
                                                    history ↦ m1.history]};
030:                       invokedOp := 0;
031:                       panic := FALSE}}
032:       or {when panic = FALSE;
033:           panic := TRUE;
034:           Send([type ↦ "PANIC", c ↦ self, tc ↦ localts, o ↦ invokedOp])}
035:       or {with (MM = {m ∈ msgs : (m.type = "ABORT") ∧ (m.c = self) ∧ (m.tc = localts)}) {
036:             when (Cardinality(MM) > (2 ∗ Cardinality(Server)) ÷ 3);
037:             Returned := Returned ∪ {[type ↦ "ABORT",
                                          req ↦ [type ↦ "REQ", c ↦ self, tc ↦ localts, o ↦ invokedOp],
                                          history ↦ LVP(getAbortHistory(1, MM))]};
038:             invokedOp := 0;
039:             panic := FALSE}}}};
040:   finished[self] := TRUE}

041:   process (serv ∈ Server)
042:     variables stoplocal = FALSE; ts = [i ∈ Client ↦ 0]; LH = ⟨⟩;{
043:S:   while (∃c ∈ Client : finished[c] = FALSE) {
044:       either {when (stoplocal = FALSE);
045:               with (m1 ∈ {m ∈ msgs : validReq(m, ts)}) {
046:                 ts[m1.c] := m1.tc;
047:                 LH := LH ∘ ⟨m1⟩;
048:                 Send([type ↦ "ACK", history ↦ LH, c ↦ m1.c, tc ↦ m1.tc, server ↦ self])}}
049:       or {with (m1 ∈ {m ∈ msgs : validPanic(m, ts, msgs, self)}) {
050:           stoplocal := TRUE;
051:           Send([type ↦ "ABORT", history ↦ LH, c ↦ m1.c, tc ↦ m1.tc, server ↦ self])}}}}} *)
```

Figure 6.7: DEC client/server +CAL code

DEC +CAL code

Since *DEC* is a very simple algorithm (in particular when compared to full-fledged BFT-SMR algorithms), we were able to implement it in +CAL and to partially verify its correctness using the TLC model checker. Here, the partial verification means that we model check *DEC* only in executions in which no server (or client) fails. However, our machine verification using TLC, verifies *DEC* correctness in all (failure-free) executions, including those that exhibit asynchrony and contention. To our knowledge, this is the first +CAL implementation of a (portion of a) BFT-SMR algorithm and we feel it represents a (perhaps modest) pioneer step in machine verification of BFT-SMR algorithms. Naturally, in order to fully prove *DEC* correct, we give a traditional, "hand-written", proof of *DEC* correctness in Section 6.6.1.

The *DEC* +CAL code is given in Figures 6.6 and 6.7. For simplicity and better readability, the case of a *DEC* that does not need to be initialized is shown; it is not difficult to add the details on handling INIT requests along the lines of *DEC* description in Section 6.5.1. Predicates in lines 7-8 (resp., 9-11), in Figure 6.6, serve for computing Step 3b.2.(i) (resp., 3b.2.(ii)) of the algorithm. The (trivial) Step 3b.2.(iii) is not shown for better readability of the code. Finally, and still to simplify our +CAL code, we assume that all servers (in addition to server s_1) send their full local histories in Step 2 (rather than history digests).

The code shown in Figures 6.6 and 6.7, when ran in TLC v2.0 in the model check mode, generates 69729 distinct states with the depth of the state graph search of 31 in the configuration with 4 servers, 2 clients, 2 different SMR operations and 1 request per client ($MaxReq = 1$), by having TLC make use of symmetry in sets *Server*, *Client* and *Operation*. It takes only 48 seconds to model check the algorithm and verify its correctness in this configuration on a 4xdual core Opteron 8216 with 8 GB RAM. In a similar configuration, with 2 requests per client ($MaxReq = 2$), TLC generates 56613741 distinct states with the depth of the state graph search of 46. However, this model check takes as much as 22h 32mins (on the above mentioned machine). This depicts the exponential growth of verification complexity (with a rather large exponent) when increasing the size of a verified configuration.

6.5.2 Chain Abstract

Chain is an implementation of *Abstract* with the following Non-Triviality property:

(Chain Non-Triviality) If (a) a correct client c invokes a request m, (b) there are no server failures and (c) the set of servers (Σ) is synchronous, then client c commits m.

Notice two differences between *DEC* and *Chain* Non-Triviality: (1) *DEC* Non-Triviality does not allow for contention, and (2) *DEC* Non-Triviality requires the system (i.e., both clients and servers) to be synchronous whereas *Chain* Non-Triviality requires only the set of servers to be synchronous. Roughly speaking,

Chain shares with *DEC* the ideas of committing requests when receiving identical histories, panicking, extracting abort histories and handling of INIT requests. However, *Chain* employs a radically different message pattern (in the best-case, before a client panics), as explained below.

Every instance of *Chain* has two particular servers designated as the *head* and the *tail* and the fixed ordering of server IDs (called *chain order*), known to all processes such that the head precedes all servers in the chain order, whereas the tail is preceded by all servers; without loss of generality we assume that the head (resp., tail) is server s_1 (resp., s_{3t+1}).[7]

In *Chain*, a client on invoking a request sends the request *req* to the head, who is responsible for assigning sequence numbers to requests. Then, each server passes the request (possibly after adding some data) to its *successor*, i.e., server s_i sends a message to s_{i+1}, whereas the tail sends the message (reply) to the client (Fig. 6.8). Similarly, a server in *Chain* accepts a CHAIN message only if sent by its *predecessor* (defined dually to successor), or from the client in case the server is the head.

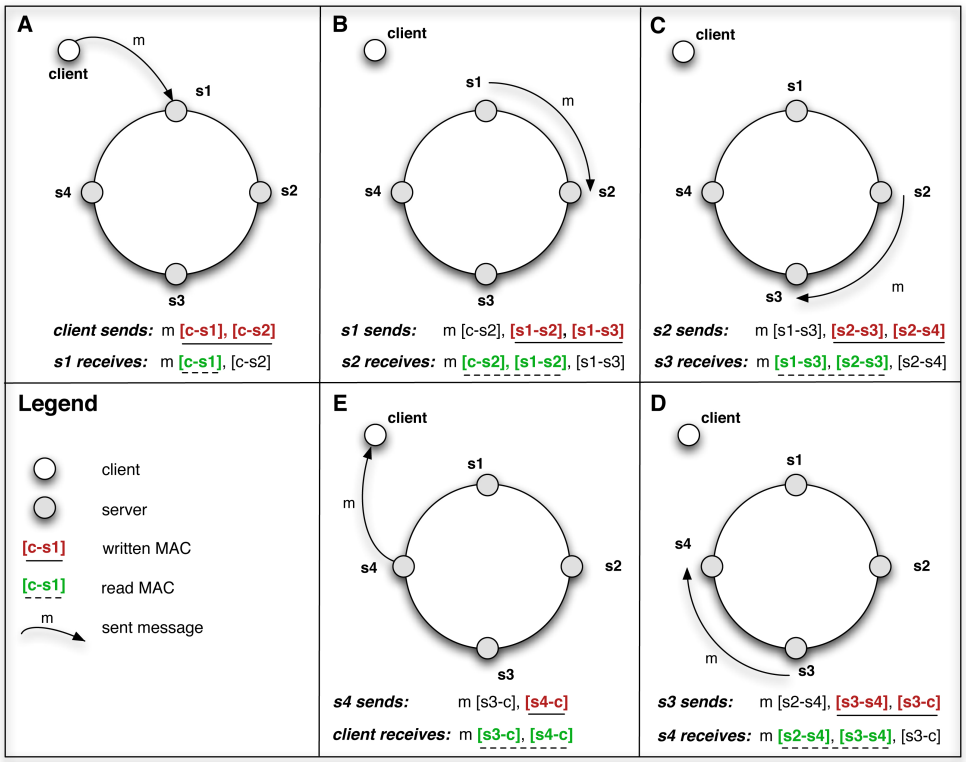


Figure 6.8: Chain authenticators ($t = 1$).

To ensure safety in presence of Byzantine clients and servers, *Chain* relies on *chain authenticators (CA)*. Roughly speaking, CAs are lightweight MAC authenticators that guarantee that, if a client's request *req* commits, (1) no server in chain is bypassed, and (2) a correct server receives *req* and not some other request.

[7]The head, the tail and chain order do not have to be the same in every instance of *Chain*.

Chain authenticators are depicted in Figure 6.8 (for the case $t = 1$). Server s_i uses a chain authenticator to authenticate a message for all servers in its *successor set*, denoted by $\overrightarrow{s_i}$ (i.e., in our notation defined in Section 6.3, s_i uses authenticator $\alpha_{s_i,\overrightarrow{s_i}}$). A successor set for server s_i depends on the position i of the server in the chain order: (a) for the first $2t$ servers (i.e., when $i \leq 2t$), a successor set contains the next $t+1$ servers, i.e, $\overrightarrow{s_i} = \{s_{i+1} \dots s_{i+t+1}\}$, whereas (b) for other servers (i.e., when $i > 2t$), a successor set contains all subsequent servers, i.e., $\overrightarrow{s_i} = \{s_{i+1}, \dots s_{3t+1}\}$.

t_c - local timestamp at client c
o - operation invoked by the client
c/s_i - client (resp., server) ID
LH_i - a local history at server s_i
sn_i - request sequence number at server s_i
$lastreq_i[c]$ - the last request req invoked by client c executed by server s_i
$lastsn_i[c]$ - the sequence number associated to $lastreq_i[c]$
$lasthist_i[c]$ - copy of the state of LH_i upon s_i executes $lastreq_i[c]$
$CASET$ - the set of chain authenticators (used only in *Chain*)
$MACSET$ - the set of MACs destined to a client of a chain (used only in *Chain*)

Figure 6.9: *Chain* and *AZyzzyva* message fields and process local variables.

Dually, when a server in *Chain* receives a message m it *verifies* m, i.e., it checks whether m is correctly authenticated by the preceding servers as described above. Namely, server s_i must be able to verify the MACs issued by: (a) if $i \leq t+1$, all servers from the set $\overleftarrow{s_i} = \{s_1 \dots s_{i-1}\}$, and (b) if $i > t+1$, all servers from the set $\overleftarrow{s_i} = \{s_{i-(t+1)}, \dots s_{i-1}\}$. We refer to set $\overleftarrow{s_i}$ as the *preceding* set for s_i.

We give *Chain* pseudocode in Figure 6.10 (client code) and Figure 6.11 (server code). For simplicity and better readability, these pseudocodes do not contain the part of the algorithm related to INIT messages; this is fairly straightforward to do following the description of *Chain* we give below. In the following we describe *Chain* while following a client's request throughout the algorithm (the message fields and process local variables we use in *Chain*, as well as in *AZyzzyva* are explained in Fig. 6.9).

1. Client sends a request to head.

On invoking request $req = \langle o, t_c, c \rangle$ (lines 3-4, Fig. 6.10) client c sends a CHAIN message to the head (line 7, Fig. 6.10). All CHAIN messages in *Chain* consist of the following fields: (1) client's request req, (2) request *sequence number* sn, (3) a set of chain authenticators $CASET$; roughly, these are destined to servers and used to authenticate req and sn and (4) a set of MACs $MACSET$; these are destined to the client and authenticate the servers' response. For uniformity (and cleaner pseudocode), all of the above 4 fields are included in every CHAIN message. However, a client, on sending a request to the head in line 7, Fig. 6.10, populates only fields req and $CASET$. The sequence number sn is populated by

1: **procedure** initialization:
2: $t_c := 0$; $T_{Chain} := (3t + 3)\Delta$

3: **procedure** INVOKE(o)
4: $t_c := t_c + 1$; $req := \langle o, t_c, self \rangle$
5: **for** $i = 1$ to $t + 1$ **do**
6: $CA[i] := MAC(self, s_i, req)$
7: send $\langle \mathsf{CHAIN}, req, nil, \{CA\}, \emptyset \rangle$ to the head (s_1)
8: trigger(T_{Chain})

9: **upon** received $\langle \langle \mathsf{CHAIN}, req, *, *, MACSET \rangle, LH \rangle$ from the tail (s_{3t+1}) **and**
$\forall i : (2t < i \leq 3t + 1) \Rightarrow (MAC(s_i, self, \langle req, D(LH) \rangle) \in MACSET)$ **do**
10: trigger(COMMIT(req, LH)); **return**

11: **upon** T_{Chain} expires **do**
12: send $\langle \mathsf{PANIC}, req_{\sigma_c} \rangle$ to (any) $2t + 1$ servers

13: **upon** received $\langle \mathsf{GET\text{-}A\text{-}GRIP}, h, req \rangle_{\mu_{s_i,self}}$ from $t + 1$ diff. servers s_i with the same h **do**
14: trigger(COMMIT(req, h)); **return**

15: **upon** received $\langle \mathsf{ABORT}, AH_{i\sigma_{s_i}}, t_c, s_i \rangle_{\mu_{s_i,self}}$ from $2t + 1$ different servers s_i **do**
16: $AH' := \langle\rangle$; $j := 1$; $hist := \bigcup_i AH_i$
17: **while** $\exists req', \exists AH^j \subset hist : (|AH^j| \geq t + 1) \wedge (\forall h, h' \in AH^j : h[j] = h'[j] = req')$ **do**
18: $AH' := AH' \circ \langle req' \rangle$
19: $j := j + 1$
20: $abortH :=$ **choose** $AH'' : isPrefix(AH'', AH') \wedge (\forall req_1, req_2 \in AH'' : req_1 \neq req_2)$
21: **if** $req \notin abortH$ **then**
22: $abortH := abortH \circ \langle req \rangle$
23: trigger(ABORT($req, abortH$)); **return**

Figure 6.10: *Chain*: client pseudocode

1: **procedure** initialization:
2: $LH := \langle\rangle$; $stopChain := FALSE$; $T_{OBR_*} := (3t+3)\Delta$; $sn := 0$; $OBRPending := \emptyset$
3: $\forall c \in Clients : (lastreq[c] := nil;\ lastsn[c] := nil;\ lasthist[c] := nil)$

4: **function** updateCAs($CASET, req, sn$)
5: $myCASET := \emptyset$; $myCA := nil$
6: **for** all $CA \in CASET$ **do**
7: $CA' := CA$; $CA'[i] := nil$; $myCASET := myCASET \cup CA'$
8: **if** $i \leq 2t$ **then** $end := i + t + 1$ **else** $end := 3t + 1$
9: **for** $j = i + 1$ to end **do**
10: $myCA[j] := MAC(self, s_j, \langle req, sn\rangle)$
11: $myCASET := myCASET \cup \{myCA\}$; **return** $myCASET$

12: **function** updateMACs($MACSET, req, c, LH$)
13: $myMACSET := MACSET$;
14: **if** $i > 2t$ **then** $myMACSET := myMACSET \cup MAC(self, c, \langle req, D(LH)\rangle)$
15: **return** $myMACSET$

16: **procedure** execute(req)
17: **if** $i = 1$ **then** $sn := sn + 1$ **else** $sn := sn'$
18: $LH := LH \circ \langle req\rangle$; $lastreq[req.c] := req$; $lastsn[req.c] := sn$; $lasthist[req.c] := LH$
19: **for** all $req' \in OBRPending \wedge req'.c = req.c$ **do**
20: **if** $req'.t_c < lastreq[req.c].t_c$ **then** $OBRPending := OBRPending \setminus \{req'\}$; stop($T_{OBR_{req'}}$)

21: **upon** received $\langle$CHAIN$, req, sn', CASET, MACSET\rangle$ from s_{i-1} (or a client, if $i = 1$) **do**
22: **if** $\bigwedge i \leq t+1 \Rightarrow \exists CA \in CASET : CA[i] = MAC(req.c, self, req)$
$\bigwedge \forall s_j \in \overleftarrow{self} : \exists CA \in CASET : CA[i] = MAC(s_j, self, \langle req, sn'\rangle)$
$\bigwedge req.t_c > lastreq[req.c].t_c \wedge \neg stopChain \wedge (i > 1 \Rightarrow sn' = sn + 1)$ **then**
23: execute(req)
24: **if** $i = 3t + 1$ **then**
25: send $\langle\langle$CHAIN$, req, nil, \emptyset, updateMACs(MACSET, req, req.c, LH)\rangle, LH\rangle$ to $req.c$
26: **else**
27: send $\langle$CHAIN$, req, sn, updateCAs(CASET, req, sn)$
$updateMACs(MACSET, req, req.c, LH)\rangle$ to s_{i+1}

28: **upon** received valid $\langle$PANIC$, req_{\sigma_c}\rangle$ from client $req.c$ such that $req.t_c \geq lastreq[req.c].t_c$ **do**
29: $OBRPending := OBRPending \cup \{req\}$
30: send $\langle OBR, self, req_{\sigma_c}, nil, \emptyset, \emptyset\rangle_{\mu_{self,s_1}}$ to the head s_1
31: trigger($T_{OBR_{req}}$)

32: **upon** received $\langle$OBR$, s_j, req_{\sigma_{req.c}}, sn', CASET, MACSET\rangle_{\mu_{s_k,self}}$ from $s_k = s_{i-1}$
(or from $s_k = s_j$, if $i = 1$) **do**
33: **if** $\bigwedge \neg stopChain \wedge req.t_c \geq lastreq[req.c].t_c \wedge req_{\sigma_{req.c}}$ is valid
$\bigwedge \forall s_j \in \overleftarrow{self} : \exists CA \in CASET : CA[i] = MAC(s_j, self, \langle req, sn'\rangle)$ **then**
34: **if** req=$lastreq[req.c]$ **then**
35: $sn_{OBR} := lastsn[req.c]$; $LH_{OBR} := lasthist[req.c]$
36: **else**
37: **if** $\neg(req.t_c > lastreq[req.c].t_c \wedge (i > 1 \Rightarrow sn' = sn + 1))$ **then break upon**
38: execute(req)
39: $sn_{OBR} := sn$; $LH_{OBR} := LH$
40: **if** $i = 3t + 1$ **then**
41: send $\langle\langle$OBR$, s_j, req_{\sigma_{req.c}}, nil, \emptyset, updateMACs(MACSET, req, s_j, LH_{OBR})\rangle, LH_{OBR}\rangle$ to s_j
42: **else**
43: send $\langle$OBR$, s_j, req_{\sigma_{req.c}}, sn_{OBR}, updateCAs(CASET, req, sn_{OBR})$,
$updateMACs(MACSET, req, s_j, LH_{OBR})\rangle_{\mu_{s_i,s_{i+1}}}$ to s_{i+1}

44: **upon** received $\langle\langle$OBR$, self, req_{\sigma_{req.c}}, *, *, MACSET\rangle, h\rangle$ from the tail (s_{3t+1}) **and**
$\forall i : (2t < i \leq 3t + 1) \Rightarrow (MAC(s_i, self, D(h)) \in MACSET)$ **do**
45: send $\langle$GET-A-GRIP$, h, req\rangle_{\mu_{self,req.c}}$ to $req.c$
46: stop($T_{OBR_{req'}}$)

47: **upon** $T_{OBR_{req}}$ expires for some req **or** received $\langle$STOP$, req\rangle_{\mu_{s_j,self}}$ from some s_j **do**
48: **if** $\neg stopChain$ **then** $stopChain := TRUE$;
49: send $\langle$ABORT$, LH_{\sigma_{s_i}}, req.t_c, self\rangle_{\mu_{self,req.c}}$ to $req.c$
50: send $\langle$STOP$, req\rangle_{\mu_{self,s_j}}$ to every server s_j

Figure 6.11: *Chain*: server s_i pseudocode

the head, whereas $MACSET$ is populated by the last $t+1$ servers in the chain order.

$CASET$ sent by the client in line 7, Fig. 6.10, contains a MAC authenticating req for the first $t+1$ servers in the chain order (lines 5-6, Fig. 6.10). In our pseudocode $MAC(p, q, msg)$ (see, e.g., line 6, Fig. 6.10) represents a MAC generated by process q and destined to process q that authenticates msg.

Upon sending a CHAIN message to the head, the client triggers the timer T_{Chain} (set to more than $(3t+3)\Delta$). We assume that correct client c does not invoke another request before it commits/aborts req.

Handling INIT Requests. The first message sent by the client must contain also the INIT tag and a valid init history IH (recall that a valid init history is an unforgeable abort history from the preceding Abstract).

2. A server in chain receives a CHAIN message, updates the message fields and forwards it to its successor.

Server s_i, on receiving a $\langle \mathsf{CHAIN}, req, sn', CASET, MACSET \rangle$ message from its predecessor (or from the client in case s_i is the head, i.e., in case $s_i = s_1$), first checks whether it can successfully authenticate and accept the message. This check consists of several conditions, captured by the predicate in line 22, Fig. 6.11. Namely:

1. The first $t+1$ servers check whether some chain authenticator (CA) in $CASET$ contains a valid MAC issued by the client that authenticates req,
2. every server s_i checks whether $CASET$ contains a CA with a valid MAC for every server s_j from its preceding set, $\overleftarrow{s_i}$ authenticating req and sn' (notice that the preceding set for the head s_1, $\overleftarrow{s_1} = \emptyset$),
3. every server accepts a CHAIN message only if the client's timestamp of the request req, $req.t_c$ is higher than $lastreq_i[req.c].t_c$, i.e., only if $req.t_c$ is higher than the timestamp of the last request invoked by the same client and executed by s_i (as described below).
4. finally, every server but the head accepts a CHAIN message only if the sequence number of the message, sn', exactly equals $sn_i + 1$.

If this check succeeds (we assume, for the time being, that the boolean $stopChain$ is false at every correct server), then s_i *executes* req (line 23, Fig. 6.11), by invoking the procedure given in lines 16-20, Figure 6.11 and explained in the following. First, if s_i is the head (i.e., if $s_i = s_1$), it increments its local sequence number sn_1. On the other hand, if s_i is not the head, it stores sn' into its local variable sn_i (line 17, Fig. 6.11). Moreover, every server s_i (line 18, Fig. 6.11): (1) appends req to its local history LH_i and (2) updates the data that reflects the last request by the client $req.c$ by storing req, sn_i and LH_i respectively into $lastreq_i[req.c].t_c$, $lastsn_i[req.c]$ and $lasthist_i[req.c]$.

Upon executing req, s_i is ready to forward the CHAIN message. Server s_i sends a CHAIN message containing req and sn_i, as well as updated sets $CASET$ and $MACSET$. These sets are updated using (respectively) the functions $updateCAs()$ (lines 4-11, Fig. 6.11) and $updateMACs()$ (line 12-15, Fig. 6.11).

- Every server s_i executes $updateCAs()$. First, s_i removes from the $CASET$ (received in the CHAIN message) all the MACs destined to itself from all CAs in $CASET$ (lines 6-7, Fig. 6.11); this is done only as a performance optimization and can safely be omitted. Then, s_i adds a CA authenticating the pair $\langle req, sn_i \rangle$ for every server in its successor set, $\overrightarrow{s_i}$ (lines 8-11, Fig. 6.11).

- On the other hand, $updateMACs$ is effectively executed only by the last $t+1$ servers in the chain order. These servers authenticate the pair $\langle req, D(LH_i) \rangle$ with a MAC destined to the client $req.c$ (lines 13-15, Fig. 6.11). Here, $D(LH_i)$ denotes a digest of the server's local history.

Finally, every server sends a CHAIN message containing req, sn_i, as well as sets $CASET$ and $MACSET$ as described above, to its successor in the chain order (line 27, Fig. 6.11). The only exception is the tail (i.e., $s_i = s_{3t+1}$), that sends a CHAIN message as described above to the client $req.c$. In addition, the tail appends to the CHAIN message its full local history LH_i (line 25, Fig. 6.11).

CHAIN message verification failure. If a verification of the received CHAIN message in line 17, Fig. 6.11 fails, server s_i may safely discard the received message, except in one case. This case, is the one where all the predicates in line 17 are satisfied except the first one, i.e., if s_i is one of the first $t+1$ servers in the chain order and it cannot find a MAC issued by the client in $CASET$ that successfully authenticates request req.

More precisely, this is only the case if s_i is not the head (i.e., if $s_i \in \{s_2 \dots s_{t+1}\}$). Obviously, if s_i is the head and this case occurs, then the client is Byzantine and the head may safely discard the CHAIN message.

On the other hand, if $s_i \in \{s_2 \dots s_{t+1}\}$, a client or some server in $\overleftarrow{s_i}$ is Byzantine (a Byzantine client may not have included a correct MAC, or this may have been corrupted or removed by some preceding Byzantine server). The *Verification Failure Recovery* (VFR) subprotocol that guarantees algorithm correctness in this case is presented in details separately, later in this section (VFR subprotocol is based on a solution to a similar problem described in [KAD$^+$]).

Handling INIT Requests. The first message that the head can assign a sequence number to must contain the INIT tag and a valid init history IH. Moreover, if the local history LH_i of s_i is empty, s_i may only execute the INIT request with a valid init history IH. More precisely, s_i executes the entire IH, by appending the entire IH (instead only req) to its (empty) history LH_i. All (non-INIT) requests received before are discarded. If LH_i is not empty, s_i neglects IH and executes only req, as described above.

3a. Client receives the CHAIN message from the tail that it can successfully verify before the expiration of the timer and commits the request.

If client c receives $\langle \langle \mathsf{CHAIN}, req, *, *, MACSET \rangle, LH \rangle$ from the tail s_{3t+1}, that can be successfully verified, then c commits request req with *Chain* commit history LH (lines 9-10, Fig. 6.10). Successful verification means here that the set $MACSET$ contains valid MACs from the last $t+1$ servers in chain order destined

to c, that authenticate the pair $\langle req, d\rangle$ and if $d = D(LH)$.

3b. Client *does not* receive the CHAIN message from the tail that it can verify before the expiration of the timer and panics.

If the client does not the receive the message from the tail as described in Step 3a before T_{Chain} expires, the client panics, i.e., it sends a $\langle \mathsf{PANIC}, req_{\sigma_c}\rangle$ to (any) $2t + 1$ servers s_i (lines 11-12, Fig. 6.10). Notice here that, in a PANIC message, req is digitally signed by the client.

Moreover, as in Step 3b of *DEC*, the client periodically re-sends the PANIC message to servers, until it commits or aborts the request.

3b.1. Server receives a PANIC message, and retries *Chain* on behalf of the client.

Server s_i, on receiving a $\langle \mathsf{PANIC}, req_{\sigma_{req.c}}\rangle$ message (with a valid signature on req), tries to commit the client's requests by invoking Steps 1-3a on behalf of the client (lines 28-31, Fig. 6.11). Namely, s_i acts as a client (as well as a server) and sends the $\langle \mathsf{OBR}, s_i, req_{\sigma_{req.c}}, sn_{OBR} = nil, CASET_{OBR} = \emptyset, MACSET_{OBR} = \emptyset\rangle_{\mu_{s_i,s_1}}$ message to the head (line 30, Fig. 6.11) and triggers the OBR timer for req, $T_{OBR_{req}}$ (line 31, Fig. 6.11).

As can be seen from its format, an OBR message is very similar to a CHAIN message, with some differences:

- besides the fields req, sn, $CASET$ and $MACSET$, an OBR message contains an additional field which the server s_i (that invokes an on-behalf request OBR) populates with its own ID,
- in an OBR message, $CASET$ is initially empty (unlike when a client sends a CHAIN message to the head). Intuitively, given that the client's signature on req is included in an OBR message, additional MACs are not needed (notice that this also prevents the verification failure issue in the case of an OBR, described for the case of a CHAIN message in Step 2).

Finally, we note that a correct server s_i sends an OBR request to the head, only if the $req.t_c$ of the PANIC message is greater or equal than $lastreq_i[req.c].t_c$ (line 28, Fig. 6.11). Moreover, if in any point in time $ts_i(c)$ becomes greater than $req.tc$ (suggesting that c committed req and invoked another request), s_i abandons waiting for the CHAIN message from the tail, cancels its timer and does not send anything to the client (see *execute* procedure, lines 19-20, Fig. 6.11).

3b.2. A server in chain receives an OBR message and processes it in a similar way as the CHAIN message (Step 2).

On the server side, an OBR request is treated in the same way as a CHAIN request coming from the client in Step 1, except that (lines 30-45, Fig. 6.11):

- it may happen that the same request req has already been executed by server s_i and stored into $lastreq_i[req.c]$ (earlier requests from this client may be safely ignored by the server). In this case, the changes with respect to Step 2 are the following (lines 35-36, Fig. 6.11):

  - the server does not execute a request, but simply repeats the sequence number it used for request req, $lastsn_i[req.c]$,
  - moreover, if s_i is one of the last $t+1$ servers (i.e., if s_i should authenticate $D(LH_i)$), s_j replaces LH_i with $lasthist_i[req.c]$ (see also 41 and 43, Fig. 6.11).

- client c (i.e., $req.c$) does not receive any message; a message sent by the tail, normally destined to c, is sent to s_i (i.e., the server that invoked an on-behalf request). Analog holds for MACs placed in the field $MACSET$ that are destined in Step 2 to the client (lines 40-43, Fig. 6.11).

3b.2a. Server commits the request on behalf of the client and forwards the commit history to the client.

If s_i receives an OBR message from the tail containing MACs for the pair $\langle req, D(h)\rangle$ from the last $t+1$ servers in $MACSET$, as well as the full history h, then s_i sends $\langle \text{GET-A-GRIP}, h, req\rangle_{\mu_{s_i,req.c}}$ to $req.c$ (lines 44-46, Fig. 6.11). For simplicity of presentation, when server s_i takes this step, we say (with slight abuse of the language) that s_i commits the OBR for req with history h.

Again, to counter possible message losses, if a server receives a repeated PANIC message for req after committing an OBR for req (and if the flag $stopChain$ is false), the server replies to $PANIC$ by re-sending the GET-A-GRIP message to the client.

3b.2a.1. Client receives $t+1$ GET-A-GRIP messages with the same history and commits the request.

If the client received $t+1$ $\langle \text{GET-A-GRIP}, h, req\rangle$ messages from different servers, with the same history h, the client commits the request by returning $Commit(req, h)$ (lines 13-14, Fig. 6.10).

3b.2b. Server *does not* commit the request on behalf of the client, stops processing new requests and sends a signed history to the client.

If a server does not receive a commit history from its OBR request before the expiration of the timer, it acts similarly as the server receiving PANIC in Step 3b.1 of *DEC*. More specifically, (a) s_i stops accepting new CHAIN and OBR messages (i.e., stops executing Steps 2 and 3b.2), by setting the flag $stopChain$ to true (line 48, Fig. 6.11) and (b) sends a signed local history to the client using an $\langle \text{ABORT}, LH_{i\sigma_{s_i}}, req.t_c, s_i\rangle_{\mu_{s_i,req.c}}$ message to client $req.c$ (line 49, Fig. 6.11).

In addition, s_i sends $\langle \text{STOP}, req\rangle_{\mu_{s_i,s_j}}$ to every server s_j (line 50, Fig. 6.11). To counter possible message losses, we assume here that s_i periodically retransmits the STOP message.

Handling INIT Requests. If the history LH_i of server s_i is empty, s_i acts in this step only on a PANIC message for an INIT request (that contains correctly authenticated init history IH from preceding Abstract). In this case, upon receiving the first such PANIC message, s_i, before sending an ABORT message sets LH_i

to IH. The following PANIC messages for INIT requests are treated as described above, neglecting the init histories.

3b.2b.1 Server receives a STOP message from some other server, stops processing new requests and sends a signed history to the client.

Server s_j, upon receiving the STOP message, acts in exactly the same way as in Step 3b.1b (lines 44-46, Fig. 6.11).

3b.2b.2. Client receives $2t + 1$ matching ABORT messages, extracts the abort history and aborts the request.

This step is identical to Step 3b.2 of *DEC* and is depicted in lines 15-23, Fig. 6.10.

Verification failure recovery subprotocol

In *Chain*, in the case described in Step 2, $s_i \in \{s_2 \dots s_{t+1}\}$ initiates the *Verification Failure Recovery* (VFR) subprotocol[8] if s_i cannot find a MAC authenticating request req in the $CASET$ field of a CHAIN message. Basically, VFR is a variant of consensus ran among servers in the case of a verification failure to reach the agreement on whether req should appear in servers' local histories (e.g., in case req was committed), or should a special request *noop* appear in servers' local histories instead of req. Of course, VFR needs to ensure agreement only in case no server is faulty and the set of servers is synchronous; otherwise, VFR must only ensure that a correct server detects asynchrony or failure. Below, we explain VFR subprotocol in details; in short, VFR proceeds in following steps: (1) s_i requests the head to resolve this conflict, (2) the head asks all servers whether they have appended the request in question (req) to their local histories, (3) servers sign their answer and refuse to accept req until the issue is resolved, (4) the head collects all signatures and distributes them to servers and (5) if there are $t + 1$ servers that appended the request, a server appends the request and proceeds normally; otherwise, the request is erased from servers' local histories (i.e., it is replaced by *noop*) and client is asked to retransmit the request. In the latter case, all future request of client c are required to be signed, which prevents this conflict.

VFR1. A server requests verification failure recovery (VFR) from the head.

Upon a verification failure (described in Step 2 of our *Chain* description) occurs at correct server s_i, s_i sends a $\langle \mathsf{VFR\_REQ}, req, sn \rangle \mu_{s_i,h}$ message to the head. Here, req equals the req field of the CHAIN message that caused verification failure.

Moreover, the server triggers the timer T_{VFR} (set to more than 5Δ). Finally, the server s_i stops executing any new requests and suspends all OBR request

[8] Our VFR subprotocol is a variation of Fill Hole subprotocol of [KAD+] used if signatures are replaced with authenticators.

timers T_{OBR_*} until VFR completes (or T_{VFR} expires).

VFR2. The head receives the VFR request and asks all servers for the proof on whether they appended req.

On reception of a $\langle \mathsf{VFR\_REQ}, req, sn\rangle\mu_{s_i,h}$ message, the head sends a $\langle \mathsf{VFR\_ASK}, req, sn\rangle\mu_{h,s_i}$ to every server s_i.

VFR3. A server receives a VFR_ASK message from the head and replies with a signed answer on whether it appended req or not, and (if not) refuses to append req until VFR subprotocol completes.

Upon reception of $\langle \mathsf{VFR\_ASK}, req, sn\rangle\mu_{h,s_i}$, from the head, if s_i already appended req with sequence number sn to its local history, s_i replies to the head with $\langle \mathsf{VFR\_REP}, YES, req, sn\rangle\sigma_{s_i}$. Otherwise, s_i: (1) replies with $\langle \mathsf{VFR\_REP}, NO, req, sn\rangle\sigma_{s_i}$ to the head, (2)triggers the timer T'_{VFR} (set to more than 3Δ) and (3) refuses to execute new requests and suspends all OBR request timers T_{OBR_*} until VFR subprotocol completes, or until T'_{VFR} expires.

VFR4. The head receives all VFR_REP messages, generates the proof and sends the proof to all servers .

Upon the head receives a VFR_REP from *all* servers, it forms either (a) the P proof by selecting $t+1$ signed VFR_REP messages containing YES, or (b) the N proof by selecting signed VFR_REP $2t+1$ messages that contain NO. Then the head sends $\langle \mathsf{VFR\_PROOF}, P/N, req, sn\rangle\mu_{h,s_i}$ to every server s_i.

VFR5. A server receives a VFR_PROOF message m from the server and sends the MAC for m to every other server.

Every server s_i, on receiving $m = \langle \mathsf{VFR\_PROOF}, P/N, req, sn\rangle\mu_{h,s_i}$ (for the first time), sends the MAC for m to every other server . This prevents the Byzantine head to make two different correct servers act on different VFR_PROOF messages. A server may execute this step only if it already executed Step VFR3.

VFR6a. A server s_i receives the VFR proof from the head and the matching MACs from all other servers, before the expiration of any timer.

If the server s_i receives VFR_PROOF message m from the head, as well as MACs authenticating m from every other server, if m contains the P proof, s_i proceeds normally, as described in Section 6.5.2 (in particular, the server s_j that initiated VFR completes Step 3 by executing req). On the other hand, if the message m contains the N proof, then s_i inserts a special *noop* request in place with sequence number sn (possibly overwriting some request) and proceeds normally. Such *noop* requests are simply treated as non-existing in every operation on server histories described in Section 6.5.2. Moreover, s_i resumes all OBR request timers T_{OBR_*} suspended in Step VFR1 or Step VFR3.

Upon successful completion of the VFR subprotocol, the head, in case it sends

the N proof, informs the client c (that invoked req) to repeat the request using a signature (and a new client timestamp) and to authenticate all his future requests using signatures (which help avoid verification failure issue). Such a signed request within a CHAIN message is treated similarly to the signed request in an OBR message.

VFR6b. A server *does not* receive the VFR proof from the head and/or the matching MACs from all other servers before the expiration of a timer and stops processing CHAIN and VFR messages.

If a server s_j triggered a timer in Step VFR1 or Step VFR3 and did not receive a proof from the head before expiration of the timer, or s_j did not receive the corresponding MACs from all other servers as described in Step VFR6a, s_j stops processing any further CHAIN or VFR messages and expires all the timers suspended in Step VFR1 or Step VFR3.

6.5.3 Zyzzyva-like Abstract (AZyzzyva)

AZyzzyva is an *Abstract* implementation with the same Non-Triviality property as *Chain*. It mimics Zyzzyva [KAD+07] in case there are no server failures and the system is synchronous. It shares with *Chain* the panicking mechanism, on behalf requests (OBRs), handling INIT requests and extracting abort histories. Here, we skip all these details and highlight only the differences with respect to *Chain*, i.e., the messages exchanged in the best-case.

Every instance of *AZyzzyva* has one server designated as the *primary* (without loss of generality we assume that the primary is server s_1). The primary does not need to be the same in every instance of *AZyzzyva*.

1. Client sends a request to the primary.

A client c, on invoking *AZyzzyva* with $req = \langle o, t_c, c\rangle$, sends the message $m' = \langle \mathsf{REQ}, req\rangle_{\alpha_{c,\Sigma}}$ (notice that m' uses an authenticator with an entry for each server) to the primary s_1 and triggers the timer T set to 3Δ.

2. The primary receives a request, assigns it a sequence number and forwards it to other server(s).

The primary s_1 on receiving $m' = \langle \mathsf{REQ}, req\rangle_{\alpha_{c,\Sigma}}$, if t_c is higher than $ts_1(c)$, updates $ts_1(c)$ to t_c and assigns a (monotonically increasing) sequence number timestamp sn to req. Then, s_1 sends $\langle \mathsf{ORDER}, req, REQ_{\mu_{c,s_i}}, sn\rangle_{\mu_{s_1,s_i}}$ to every server s_i, where $REQ_{\mu_{c,s_i}}$ is the MAC entry of the authenticator of message m' for server s_i.

3. A server receives a forwarded request, appends it to its local history (i.e., executes it speculatively) and sends the reply to the client.

A server s_i, on receiving the $\langle \mathsf{ORDER}, req, REQ_{\mu_{c,s_i}}, sn'\rangle_{\mu_{s_1,s_i}}$ from the primary s_1, that s_i can verify (as described below), with timestamp $sn' = sn_i + 1$ (where

sn_i is the largest primary timestamp received by s_i), updates sn_i to sn and $ts[req.c]$ to $req.t_c$, appends $req = \langle o, t_c, c\rangle$ to its local history LH_i, and sends $\langle \mathsf{ACK}, LH_i, req.t_c, s_i\rangle_{\mu_{s_i,c}}$ to c.

Server s_i can verify the ORDER message if MAC $REQ_{\mu_{c,s_i}}$ matches the req field. If the case of verification failure, s_i initiates the VFR subprotocol (the same as in Step 2 of Chain), described in Section 6.5.2.

4. Client gathers matching server responses.

Client c gathers matching server responses, until the timer T (set in Step 1) expires, or until c collects $3t + 1$ matching responses from different servers. A matching response for req is an ACK message (with a valid MAC) containing $req.t_c$.

4a. Client receives $3t+1$ matching responses with identical histories and commits.

If client c receives $3t + 1$ $\langle \mathsf{ACK}, LH, req.t_c, *\rangle$ messages from different servers, with identical history LH, then the client commits the request by returning $Commit(req, LH)$.

The rest of *AZyzzyva* is the same as *Chain*: if client c does not execute Step 4a before the expiration of the timer, c panics and sends $\langle \mathsf{PANIC}, req, REQ\rangle$ to $2t+1$ servers, where REQ is its original request as in Step 1; REQ is then forwarded by the servers to the primary using an on behalf request (OBR). OBR requests are treated as regular requests with the difference highlighted in Step 3b.2 of *Chain*.

6.5.4 Implementing Backup using any BFT-SMR

We present now another implementation of *Abstract* we call *Backup* with Non-Triviality property that guarantees at least $k \geq 1$ requests to be committed where k is a generic parameter. *Backup* is used in *Modular BFT-SMR* to resolve request aborts from optimistic *Abstracts* (e.g., *DEC* and *Chain*). The +CAL code of this implementation using (any) BFT-SMR is shown in Figure 6.12. BFT-SMR in the code is implicit; the power of BFT-SMR reflects in the fact that the Backup can atomically and sequentially treat invoked requests. The simple code shown in Figure 6.12 can be directly plugged in the +CAL code of Modular BFT-SMR of Figure 6.4, Section 6.4.3. For the purpose of model checking, we defined the predicate $ValidInitHistory(h)$ to always evaluate to $TRUE$; in practice, this predicate evaluates whether the particular init history is indeed an unforgeable abort history of the preceding *Abstract*. Recall here that the ordering among Abstracts is global and hence the definition of a preceding Abstract is also global (Sec. 6.4.2).

In implementing *Backup* using BFT-SMR, we exploit the fact that any BFT-SMR can totally order requests submitted to it and implement any functionality on top of this total order. In our case, *Backup* is precisely this functionality. First, *Backup* ignores all the requests delivered by BFT-SMR until it encounters the INIT request invocation $Invoke(m', h_I)$ with the valid init history h_I. Here, a valid init history is an unforgeable abort history for request m' generated by

```
01: process (bkp ∈ {Backup})
02:   variables cnt = 0; commit = FALSE; discard = FALSE;localHistory =<<>>;
03:   {B: while (∃c ∈ Client : finished[c] = FALSE){
04:         with (inst ∈ 1..NoOfInst;
05:               newReq ∈ NewReq(Inv[self][inst], Exec[self][inst]))
06:           if (cnt = SP(inst − 1)) {
07:             if (newReq ∉ Fresh ∧ ValidInitHistory(newReq.history)) {
08:               commit := TRUE; discard := FALSE; localHistory := newReq.history; cnt := cnt + 1}
09:             else discard := TRUE}
10:           else {
11:             if (cnt < SP(inst)) {
12:               commit := TRUE; discard := FALSE; cnt := cnt + 1;
13:               if (¬Exists(getFreshReq(newReq), localHistory))
14:                 localHistory := Append(localHistory, getFreshReq(newReq))}
15:             else {commit := FALSE; discard := FALSE}};
16:           if (discard = FALSE)
17:             if (commit = TRUE)
18:               Return(self, inst, [type ↦ "COMMIT", req ↦ newReq, history ↦ localHistory]);
19:             else Return(self, inst, [type ↦ "ABORT", req ↦ newReq, history ↦ localHistory]);}}
```

Figure 6.12: Implementing *Backup* using BFT-SMR: +CAL code

the preceding *Abstract*. At this point, *Backup* sets its history H_{bkp} to h_I. Then, *Backup* simply appends the following $k-1$ invoked requests, ordered by BFT-SMR immediately after m', neglecting possible init histories and commits these requests (unless a particular request m is already in H_{bkp} when appending m one more time is avoided, and m is simply committed with (the prefix of) history H_{bkp}). Denoting by $H_{bkp}^{(k)}$ the *Backup* history after committing the k^{th} request, *Backup* aborts all subsequent requests with the (signed) abort history $H_{bkp}^{(k)}$.

In the C++ implementations of your algorithms (evaluated in Sec. 6.7), *Backup* is implemented over PBFT [CL99]. Specifically, *Backup* signing of abort histories is implemented by having every PBFT replica digitally sign its reply (containing $H_{bkp}^{(k)}$) to the client.

6.6 Implementation correctness

In this Section, we give the correctness proofs of *DEC*, and *Chain*.

6.6.1 DEC

In this Section, we prove that *DEC* implements *Abstract* with *DEC Non-Triviality*.

Validity. For any request *req* to appear in a commit or abort history, at least $t+1$ servers must have reported a history containing *req* to the client (see Step 3a. for commit histories, and Step 3b.2.(i) for abort histories). Hence, at least one correct server appended *req* to its local history. By Step 2, the correct server s_i appends *req* to its local history only if s_i receives a REQ message from a client with a valid MAC, i.e., only if some client invoked *req*, or if *req* is contained in some valid init history.

Moreover, by Step 2, no server executes the same request twice. Hence, no request appears twice in any local history of a correct process, and consequently,

no request appears twice in any commit history. In the case of abort histories, no request appears twice by construction (see Step 3b.2.(ii) Sec. 6.5.1). □

To prove Commit and Abort Ordering we first prove the following Lemma.

Lemma 42. *Denote the value of local history of correct server s_i upon appending request req to LH_i as LH_i^{req}. Then, for any* ACK *or* ABORT *message m sent by s_i upon appending req to LH_i with history LH_i^m, LH_i^{req} is a prefix of LH_i^m.*

Proof. A correct server s_i modifies its local history LH_i only in Step 2 by sequentially appending requests to LH_i. Hence, LH_i^{req} remains a prefix of LH_i forever. □

Commit Ordering. Assume, by contradiction, that there are two committed request req (by benign client c) and $req' \neq req$ (by benign client c') with different commit histories h_{req} and $h_{req'}$ such that neither is the prefix of the other. Since a benign client commits requests in *DEC* only if it receives in Step 3a identical histories (more precisely, identical history digests) from all servers, there must be a correct server s_i that sent h_{req} to c and $h_{req'}$ to c' such that $h(req)$ is not a prefix of $h_{req'}$ nor vice versa. A contradiction with Lemma 42. □

Abort Ordering. Let t be the time when the first correct server s_i executes Step 3b.1 and stops appending new requests to its local history LH_i and sends its first ABORT message m with abort history LH_i^m. By Lemma 42 and since a correct client needs to receive identical histories (history digests) from all servers to commit a request, no request req' that is not in LH_i^m can be committed. Moreover, let req be a committed request with history h_{req}, such that s_i appended req to its local history before time t. Again, by Lemma 42 and since a correct client needs to receive identical histories from all servers to commit a request, h_{req} is a prefix of every history LH_j^m sent in any ABORT message m by any correct server s_j. Hence, for every committed request req with the commit history h_{req} and any ABORT message m sent by a correct server s_j containing local history LH_j^m, h_{req} is a prefix of LH_j^m.

By Step 3b.2., a client that aborts a request waits for $2t+1$ ABORT messages including at least $t+1$ from correct servers. Since any commit history h_{req} is a prefix of every history sent by any correct server, at least $t+1$ received histories will contain h_{req} as a prefix, for any committed request req. Hence, by construction of abort histories (Step 3b.2.(i) Sec. 6.5.1) every commit history h_{req} is a prefix of every abort history. □

Termination. By assumption of a quorum of $2t+1$ correct servers and fair-loss channels: (1) correct servers eventually receive the PANIC message sent by correct client c and (2) c eventually receives $2t+1$ abort messages from correct servers. Hence, if correct client panics c, it eventually aborts invoked request req, in case c does not commit req beforehand.

Now we show that req is in any commit or abort history for req. This is immediate for abort histories (see Step 3b.2.(iii) Sec. 6.5.1). In the case of a commit history, by Step 2, a correct server s_i send ACK message with timestamp $req.ts$ and local history LH_i only upon appending req to LH_i. Moreover, by Step 3a., a client needs to receive the same history from all servers in order to commit req. Hence, if a client commits req with commit history h then req is in h. □

DEC Non-Triviality. Non-Triviality relies on the fact that client's timers are set such that they do not expire in case the system is synchronous. Assuming that the message processing at processes takes negligible time it is sufficient to set T_{DEC} not to expire before 2Δ. Since there is no contention and all servers are correct, all servers order all requests in the same way and send identical histories to the clients. □

Init Ordering. By Step 2 and Step 3b.1., every correct process must initialize its local history (with some valid init history) before sending any ACK or ABORT message. Since any common prefix CP of all valid init histories is a prefix of any particular init history I, CP is a prefix of every local history sent by a correct process in an ACK or ABORT message. Init Ordering for commit histories immediately follows. In the case of abort histories, notice that at least out of $2t+1$ ABORT messages received by a client on aborting a request in Step 3b.2 at least $t+1$ are sent by correct processes and contain local histories that have CP as a prefix. By Step 3b.2.(i), CP is a prefix of any abort history. □

6.6.2 Chain

In this Section, we prove that *Chain* implements *Abstract* with *Chain Non-Triviality*. We denote by Σ_{last} the set of the last $t+1$ servers in the chain order, i.e., $\Sigma_{last} = \{s_i \in \Sigma : i > 2t\}$. In addition, we say that correct server s_i executes *req at position pos* if the length of the LH_i upon appending *req* in line 18, Fig 6.11 is *pos*.[9]

Before proving *Abstract* properties, we first prove the following three lemmas.

Lemma 43. *If correct server s_i executes $req \neq noop$ (at position sn, at time t_1), then all correct servers s_j, $1 \leq j < i$ execute req (at position sn, before t_1).*

Proof. By contradiction, assume the lemma does not hold and fix s_i to be the first correct server that executes *req* (at position *sn*), such that there is a correct server s_j ($j < i$) that never executes *req* (at position *sn*); we say s_i is the first server for which *req* skips. Since CHAIN messages are authenticated using CAs, s_i executes *req* at position *sn* only if all servers from $\overleftarrow{s_i}$ authenticate (using a MAC) pair $\langle req, sn \rangle$, i.e., only *after* all correct servers from $\overleftarrow{s_i}$ execute *req* at position *sn*. Notice here that sequence number *sn* associated by the head to *req* is indeed equivalent to the position at which a executes *req*, since (1) if the server is the head, *sn* is always incremented by one and (2) if server is not the head, server s_i only accepts a CHAIN message with sn' if $sn' = sn + 1$ (Steps 2 and 3b.2). If $s_j \in \overleftarrow{s_i}$, s_j must have executed *req* at position *sn* — a contradiction. On the other hand, if $s_j \notin \overleftarrow{s_i}$, then s_i is not the first server for which *req* skips, since *req* skips for any correct server (at least one) from $\overleftarrow{s_i}$ — a contradiction. □

Lemma 44. *If a correct server $s_i \in \Sigma_{last}$ executes req, then no correct server inserts noop instead of req in the VFR subprotocol (Section 6.5.2).*

[9] Here, for simplicity of presentation, init histories are treated as single requests, i.e, the entire init history IH which is possibly appended to an empty local history is said to be executed at position 1. Otherwise, for a Chain instance that needs to be initialized, we would speak of a request executed at position *pos* when the corresponding length of LH_i is $pos+Length(IH)-1$.

Proof. By contradiction, assume that some correct server inserts *noop* in place of *req*, and fix s_j to be the first such server; moreover, denote by VFR' the instance of the VFR subprotocol in which this occurs. Then, in VFR', s_j receives a VFR_PROOF message m from the head containing the N proof, as well as MACs that authenticate m from all other servers, including s_i. We consider two cases:

1. s_i executes *req* before sending VFR_REP message in VFR' at time t_1. At time t_1, no correct server inserted *noop* in place of *req* (by our assumption that s_j is the first server to do so and only *after* time t_1.) Hence, by Lemma 43, at most t (Byzantine) servers and at most t servers with ID higher than i (i.e., a total of at most $2t$ servers) may send VFR_REP containing NO in VFR'. Hence, s_j cannot receive a VFR_PROOF message m from the head containing a valid N proof, a contradiction.

2. s_i executes *req* after sending VFR_REP message in VFR' at time t. By Step VFR3 of VFR subprotocol (Section 6.5.2) s_i does not execute *req* before VFR' is resolved. Notice that s_i may execute *req* only if its proceeds normally after VFR' i.e., if it receives VFR_PROOF message m from the head containing the P proof, as well as MACs authenticating m from all servers. This contradicts the assumption that s_j receives in VFR' a VFR_PROOF message m' from the head containing the N proof, with MACs authenticating m' from all servers, since no correct server will authenticate both m and m'.

□

Lemma 45. *If benign client (resp., server) c commits req (resp., the OBR for req) with history h (at time t_1), then all correct servers in Σ_{last} execute req (after t_1) and the state of their local history upon executing req is h.*

Proof. To prove this Lemma, notice that correct server $s_i \in \Sigma_{last}$ generates a MAC for a pair $\langle req, D(h')\rangle$ for some history h': (1) only after s_i executes *req* and (2) only if the state of LH_i upon execution of *req* equals h'. Moreover, by Steps 2 and 3b.2, no correct server executes the same request twice. From Steps 3a, 3b.2a and 3b.2a.1. it is immediate that a benign client (resp., server) cannot commit *req* (resp., the OBR for *req*) with h unless it receives a MAC for $\langle req, D(h)\rangle$ from every correct server in Σ_{last}. Hence the lemma.

□

Validity. For any request *req* to appear in a commit or abort history, at least $t+1$ servers must have: (a) reported a history containing *req* to the client, or (b) generated a MAC for a pair that consists of *req* and a digest of a (commit) history (see Step 3a and Step 3b.2a.1 for commit histories, and Step 3b.2b.2 (i.e., *DEC* Step 3b.2.(i)) for abort histories). Hence, at least one correct server executed *req*.

Now, we show that all correct servers execute only requests invoked by clients (with possible exception of a *noop* request which is treated as non-existent, per

Step VFR6a, Section 6.5.2). By contradiction, assume that some correct server executed a request not invoked by any client and let s_i be the first correct server to execute such a request $req' \neq noop$. We distinguish three exhaustive cases:

1. We first show that it is not possible that s_i executes req' in the VFR subprotocol (Section 6.5.2). By Step VFR4 and Step VFR6a, a correct server executes a request in VFR subprotocol only if it receives a VFR_PROOF message with the P proof containing the VFR_REP message from $t + 1$ servers, including at least one correct server s_j, asserting that s_j already executed req' to its local history — a contradiction.

2. Now, we show that it is not possible that s_i executes req' in Step 2 of *Chain*. In case $i < t + 1$, s_i executes req' only if s_i receives a CHAIN message with a MAC from the client, i.e., only if some client invoked req, or if req is contained in some valid init history. On the other hand, if $i > t + 1$, Lemma 43 yields a contradiction with our assumption that s_i is the first correct server to execute req'.

3. Finally, a correct server may execute request $req' \neq noop$, in Step 3b.2, when processing an OBR message. In this case, before executing req', correct server s_i verifies the signature of client $req'.c$ on req', which asserts that $req'.c$ indeed invoked req'.

Moreover, by Steps 2 and 3b.2, no server executes the same request twice. Hence, no request appears twice in any local history of a correct process, and consequently, no request appears twice in any commit history. In the case of abort histories, no request appears twice by construction (see Step 3b.2.(ii) Sec. 6.5.1). □

Commit Ordering. Assume, by contradiction, that there are two committed request req (by benign client c) and $req' \neq req$ (by benign client c') with different commit histories h_{req} and $h_{req'}$ such that neither is the prefix of the other. By Lemma 45, there is correct server $s_i \in \Sigma_{last}$ that executed req and req' such that the state of LH_i upon executing these requests is h_{req} and $h_{req'}$, respectively. By Lemma 44, s_i does not ever overwrite any of the requests in its local history with a *noop*, and by Steps 2 and 3b.2, s_i otherwise always simply appends the requests to LH_i. Hence it is not possible that neither the h_{req} is the prefix of $h_{req'}$ nor vice versa. A contradiction. □

Abort Ordering. Assume, by contradiction, that there is committed request req_C (by some benign client) with commit history h_{req_C} and aborted request req_A (by some benign client) with commit history h_{req_A}, such that h_{req_C} is not a prefix of h_{req_A}. By Lemma 45 and the assumption of at most t faulty servers, all correct servers (at least one) from Σ_{last} execute req_C and their state upon executing req_C is h_{req_C}. Let $s_i \in \Sigma_{last}$ be a correct server with the highest index i among all servers in Σ_{last}. By Lemma 43, all correct servers execute all the requests in h_{req_C} at the same positions these requests have in h_{req_C}. In addition, a correct server executes all the requests from h_{req_C} before sending any ABORT message; indeed, before sending any ABORT message, a correct server must set the flag *stopChain* to true which prevents further execution of requests.

Therefore, for every local history LH_i that a correct server sends in an ABORT message, h_{req_C} is a prefix of LH_i.

Finally, by Step 3b.2b.2 (Step 3b.2 of *DEC*), a client that aborts a request waits for $2t + 1$ ABORT messages including at least $t + 1$ from correct servers. By construction of abort histories (Step 3b.2.(i) Sec. 6.5.1) every commit history, including h_{req_C} is a prefix of every abort history, including h_{req_A}, a contradiction. □

Termination. A correct client that invokes *req* panics if it does not commit *req*. On panicking, the client sends a PANIC message to $2t + 1$ servers including at least $t + 1$ correct which eventually receive a PANIC message. Denote this set of $t + 1$ correct servers by Σ_P. We distinguish two cases:

- All servers from Σ_P commit their on-behalf request for *req* in Step 3b.2a. Then, by Lemma 45, assumption of at most t server failures, and the fact that no server executes the same request twice, we conclude that all servers from Σ_P send the same history h to *req.c* in a GET-A-GRIP message. By assumption of fair-loss channel and periodic retransmission of PANIC and, consequently, GET-A-GRIP messages (Steps 3b and 3b.2a) the client eventually commits *req* in Step 3b.2a.1.

- If some server $s_i \in \Sigma_P$ does not commit its OBR request for *req*, then its OBR timer for *req* eventually expires. This may not be evident in the case when some clients are Byzantine, since a chain verification failure may cause correct servers to suspend their OBR timers (Steps VFR1 and VFR3). However, these are eventually resumed or expired in Steps VFR6a and VFR6b. It is not possible that any (finite) number of Byzantine clients keep OBR timers suspended forever by repeatedly creating chain verification failures, since only one verification failure per client is possible; upon the first conflict, all subsequent requests by the conflicting client must be digitally signed (Step VFR6a).

  Hence, the OBR timer for *req* at s_i eventually expires and s_i periodically sends STOP message to all other servers (Step 3b.2b.1); hence, all correct servers periodically send ABORT to the client. In this case, by assumption of (1) $2t+1$ correct servers and (2) fair-loss channels, a correct client aborts the invoked request *req*.

Now we show that *req* is in any commit or abort history for *req*. This is immediate for abort histories (see Step 3b.2b.2, i.e., Step 3b.2.(iii) Sec. 6.5.1). In the case of a commit history, the proof follows from Lemma 45. □

Chain Non-Triviality. Non-Triviality relies on the fact that the OBR timers set by servers do not expire when the Σ is correct and synchronous. We distinguish two cases:

- In the first case, no client is Byzantine. In this case, since Σ is correct, all servers execute requests in the same order in which the head receives the requests and no chain verification failure occurs. Assume, by contradiction that some correct client invokes and aborts *req*. By Step 3b.2b.1, if some correct client aborts a request, a timer set by some server for the OBR for

req, $T_{Chain_{req}}$ must had expired. Denote, the earliest time at which this timer expires at some correct server s_i by t_{exp}. Notice that a client cannot abort req before t_{exp}. Denote the time at which s_i triggers $T_{Chain_{req}}$ by $t_{trig} < t_{exp} - (3t+3)\Delta$. Since the client $req.c$ is correct and does not invoke a subsequent request before req aborts, no server stores in $lastreq_i[req.c]$ a request with a higher timestamp than $req.t_c$ before t_{exp}. Hence, all servers process the OBR for req invoked by s_i (with the same sequence number and the same corresponding local histories). Since the set of servers is synchronous, after $3t + 3$ time units Δ s_i commits its OBR request for req before t_{exp}, i.e., $T_{OBR_{req}}$ never expires at s_i. A contradiction.

- in the case there some client is Byzantine, a chain verification failure may occur and a VFR subprotocol may be invoked in which a server may temporarily block the execution of new requests (invoked by, e.g., correct clients). To this end, servers suspend all the OBR timers upon blocking the execution of new requests at Steps VFR1 and VFR3. Upon the server receives a decision in VFR it resumes the suspended OBR timers (Step VFR6a). Notice that the server cannot execute Step VFR6b, by assumption on Σ being correct and synchronous and since timers triggered in VFR are chosen not to expire in this case. Moreover, VFR guarantees a unique decision on whether to overwrite a conflicting request or to execute it, which guarantees that all servers execute all the requests in the same order, as in the first case when no client is Byzantine. Hence, no OBR timer can expire at any server, and the client cannot abort the request. By Termination, the client commits the request.

Init Ordering. The proof is analog to the proof of *DEC* Init Ordering.

6.7 Evaluation

This section evaluates the performance characteristics of *Abstract*. We first compare the performance of *DEC* and *Chain* to that of PBFT [CL99], Q/U [AGG+05] and Zyzzyva [KAD+07]. We then assess the cost of the switching mechanism. As stated in [SDM+08], the motivations for comparing against PBFT and Zyzzyva are the following: PBFT is considered the "baseline" for practical BFT-SMR implementations, whereas Zyzzyva is considered state-of-the-art, and is known to outperform existing algorithms under most conditions. Finally, we benchmark Q/U as it is known to provide better latency than Zyzzyva under certain condition. Note that Q/U requires $5t + 1$ servers, whereas other algorithms we benchmark only require $3t + 1$ servers.

PBFT and Zyzzyva define two similar optimizations: (1) the *batching* optimization, in which servers can batch requests and (2) the *client broadcast* optimization, in which clients broadcast requests directly to all the servers (the primary thus just needs to send ordering messages). All measurements on PBFT are done with batching enabled, since this systematically improves performance. This is not the case with the batching mechanism implemented in Zyzzyva. Therefore, we assess Zyzzyva with and/or without batching depending on the experiment. Concerning

the client broadcast optimization, we show results for both configurations every time we observe an interesting behavior.

Note that all evaluations of *Chain* and *DEC* use several optimizations: (1) servers do not send local histories but computed replies, (2) In *DEC*, only one server sends the complete reply (corresponding to the complete local history), whereas the others send only the digest of the reply, and (3) to truncate histories we use lightweight checkpoint protocol triggered every 128 messages (as in PBFT and Zyzzyva [CL99, KAD+07]), explained in Section 6.7.6.

Zyzzyva, *DEC*, and *Chain* are built on the PBFT C++ code base. All algorithms are evaluated using UMAC [BHK+99] for MACs and MD5 [Riv85] for message digests (RSA digital signatures [RSA78] are used for signing messages where applicable). To ensure that the comparison is fair, we use for Q/U a simple best-case implementation realized with the same code base. This implementation of Q/U is similar to that described in [KAD+07]: (1) a client simply generates and sends $4t+1$ MACs with a request, (2) each replica verifies $4t+1$ MACs (1 to authenticate the client and $4t$ to validate the object history state (OHS)), each replica generates and sends (3) $4t+1$ MACs (1 to authenticate the reply to the client and $4t$ to authenticate OHS) with a reply to the client and (4) the client verifies $4t+1$ MACs.

We ran all our experiments on a cluster of 17 identical machines, each equipped with a 1.66GHz bi-processor and 2GB of RAM. Machines run the Linux 2.6.18 kernel and are connected using a Gigabit ethernet switch.

Finally, we use the microbenchmarks of [CL99]: in the x/y microbenchmark, a client sends a xKB request and receives a yKB reply.

6.7.1 Latency

Figure 6.13 shows the latencies of *DEC*, *Chain*, Q/U, PBFT, and Zyzzyva for the 0/0 microbenchmarks as a function of the number of tolerated failures t (ranging from 1 to 3). The results show that *DEC* outperforms all other algorithms. Q/U also achieves a good latency with $t = 1$ due to the fact that it uses the same communication pattern as *DEC*. Nevertheless, when t increases, its performance significantly decreases. This is explained by the fact that Q/U requires $5t + 1$ replicas and both clients and servers perform additional MAC computations with respect to *DEC*. Moreover the significant improvement of *DEC* over Zyzzyva (-31% at $t = 1$) can easily be explained by the fact that Zyzzyva uses an optimistic agreement algorithm requiring 3 message delays, whereas *DEC* only requires 2 message delays.

The results depicted in Figure 6.13 are slightly different from those published in [KAD+07]. Namely, the improvement of Zyzzyva over PBFT is slightly lower (+34 % at $t = 1$ vs. +50% in [KAD+07]); moreover, absolute values of measured latencies are higher (the published latency for Zyzzyva at $t = 1$ is 0.26ms against 0,51ms in our cluster). These differences can be explained by the fact that a ping between two machines on our cluster takes 0.288ms, which is already higher than the measured latency for Zyzzyva in [KAD+07].

Finally, in Figure 6.13 we can see that the latency of *Chain* is significantly higher than that of other algorithms, and linearly increases with the number of

tolerated faults. This can be explained by the fact that the *Chain* latency (in the best case) is equal to $2 + (3t + 1)$. The second observation is that the latency of *DEC*, PBFT and Zyzzyva at $t = 3$ is lower than that at $t = 2$. We do not have any explanation for this phenomenon. It was consistently observed in all the experiments we ran.

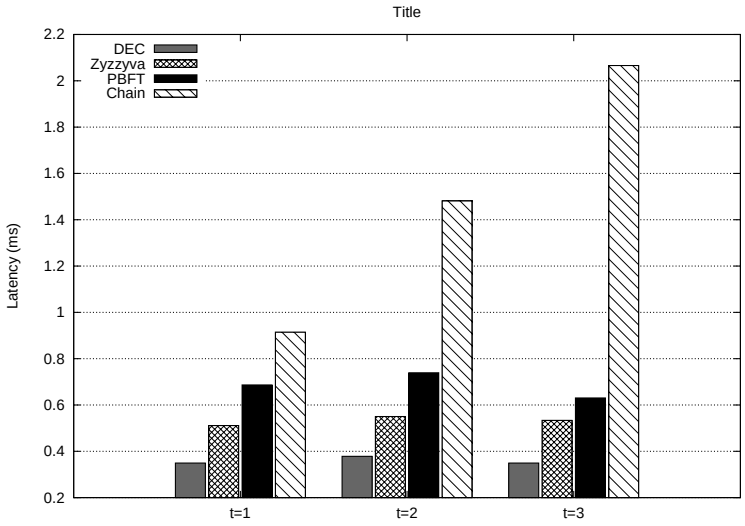


Figure 6.13: Latency of the various protocols for the 0/0 benchmark.

The latency results for all microbenchmarks (0/0, 0/4 and 4/0) are summarized in table 6.1. This table presents the latency improvement of *DEC* over Q/U, *Chain*, PBFT, and Zyzzyva. Our results show that *DEC* consistently outperforms other algorithms.

	0/0 benchmark			**4/0 benchmark**			**0/4 benchmark**		
	t=1	**t=2**	**t=3**	**t=1**	**t=2**	**t=3**	**t=1**	**t=2**	**t=3**
Q/U	8 %	14,9%	33,1%	6,5 %	13,6%	22,3%	4,7%	20,2%	26%
Zyzzyva	31,6 %	31,2%	34,5%	27,7 %	26,7%	15,6%	24,3%	26%	15,6%
PBFT	49,1%	48,8%	44,5%	36,6 %	38,4 %	26%	37,6%	38,2%	29%
Chain	61,8%	74,4%	83%	64,2%	76,4%	82,7%	61,6%	75,6%	79,5%

Table 6.1: Latency improvement of *DEC* over the various algorithms for the 0/0, 4/0, and 0/4 benchmarks.

6.7.2 Throughput

In this section, we present results obtained running the 0/0, 4/0 and 0/4 microbenchmarks. We do not present results for the Q/U and *DEC* algorithms since both algorithms are decentralized and thus perform poorly under contention.

Overall, our results show that *Chain* consistently and significantly outperforms other algorithms starting from a certain number of clients that varies for different benchmarks. Below this threshold, Zyzzyva achieves higher throughput than other algorithms.

0/0 benchmark

Figure 6.14 shows the throughput achieved by various algorithms in the 0/0 benchmark and when $t = 1$. We ran Zyzzyva with and without batching. In contrast to PBFT, which batches as many messages as possible, the size of a batch in Zyzzyva is bounded by a configuration parameter. We evaluated various batch sizes (from 2 to 10). The results we obtained show that there is no ideal batch size, which confirms the findings of [KAD+07]. Therefore, the "Zyzzyva with batching" curve is obtained by taking, independently for every number of clients, the best performance we obtained varying the batch size between 2 and 10.

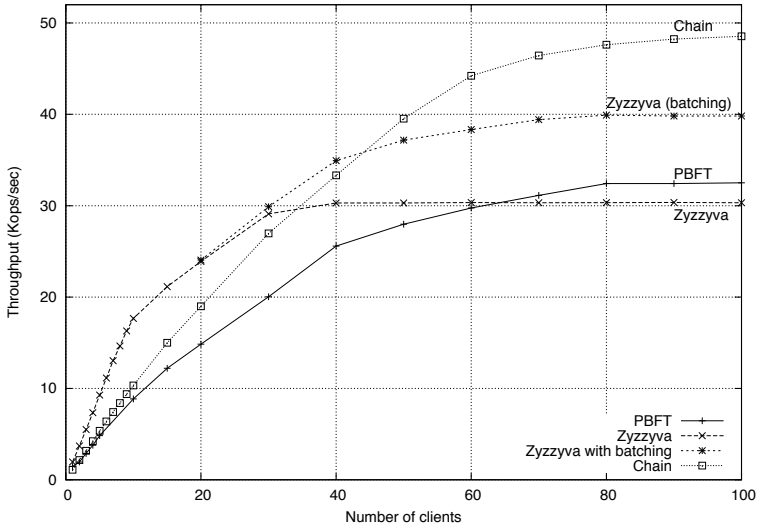


Figure 6.14: Throughput of the various protocols for the 0/0 benchmark ($t = 1$).

The results presented in Figure 6.14 are consistent with the ones published in [KAD+07, SDM+08]: batching in Zyzzyva improves performance and the performance of PBFT (with batching) are better than that of Zyzzyva without batching from a certain number of clients. Absolute values are nevertheless lower and the gain brought by batching requests in Zyzzyva is not as significant. As for the latency experiments, we attribute this lower performance to the fact that we used different hardware.

Moreover, Figure 6.14 shows that with up to 40 clients, Zyzzyva achieves the best throughput. With more than 40 clients, *Chain* starts to outperform Zyzzyva. The peak throughput achieved by *Chain* is 21% higher than that of Zyzzyva with batching. We explain this result as follows: the advantage of *Chain* over other algorithms resides in the pipelining pattern it uses for broadcasting requests and replying to clients. Thanks to this pipelining pattern, servers in *Chain* send

(resp., receive) messages to (resp., from) one server, which results in a better network usage and a significant decrease of packet losses.

Nevertheless, for *Chain* to be efficient, the pipeline must be fed, i.e. the link between any server and its successor in the chain must be saturated[10]. There are two ways to feed the pipeline: using large messages (see next section), or a large number of small messages; in the case of 0/0 benchmark, the latter applies. Moreover, as the microbenchmarks we are using implement closed-loop load injection (meaning that a client only issues one request when it gets a reply to its previous request), it is necessary to have a large number of clients to issue a large number of requests. This explains why *Chain* starts outperforming Zyzzyva when there are more than 40 clients.

0/4 benchmark

Figure 6.15 shows the throughput of the various algorithms for the 0/4 microbenchmark when $t = 1$. PBFT and Zyzzyva are using the client broadcast optimization. We observe that with up to 15 clients, Zyzzyva outperforms other algorithms. Starting from 20 clients, *Chain* outperforms PBFT and Zyzzyva. Nevertheless, the gain in peak throughput (+7,7% over PBFT and 9,8% over Zyzzyva) is lower than the gain we had with the 0/0 microbenchmark. This can be explained by the fact that the dominating cost is now sending replies to clients, partly masking the effect of request processing and request/sequence number forwarding. In all algorithms, there is only one server sending a full reply to the client (other servers send only a digest of the reply). We were expecting PBFT and Zyzzyva to outperform *Chain*, since the server that sends a full reply in PBFT and Zyzzyva changes on a per-request basis. Nevertheless, this is not the case. We again attribute this result to the fact that *Chain* uses a pipelining pattern: the last server in the chain (i.e., the tail) replies to clients at the throughput of about 391MB/sec.

4/0 benchmark

Figure 6.16 shows the results of *Chain*, PBFT and Zyzzyva for the 4/0 microbenchmark when $t = 1$. Note that we use a logarithmic scale for the X axis to better observe the behavior of the various algorithms with small numbers of clients. We only plot a curve for Zyzzyva with the client broadcast optimization enabled (explanation follows), whereas we plot two curves for PBFT (with and without client broadcast optimizations). The graph shows that with up to 3 clients, Zyzzyva outperforms other algorithms. With more than 3 clients, the *Chain* algorithm significantly outperforms other algorithms. Its peak throughput is about 375% higher than that of Zyzzyva. The reason why *Chain* is very efficient with large requests is explained in previous paragraphs.

[10]Saturation is reached when a server is sending the maximum number of bytes it can send provided it must also process requests (i.e. perform cryptographic operations and execute requests). Note that in the microbenchmarks we are using, this maximum outgoing number of bytes increases when the request size increases as the ratio processing time/sending time decreases.

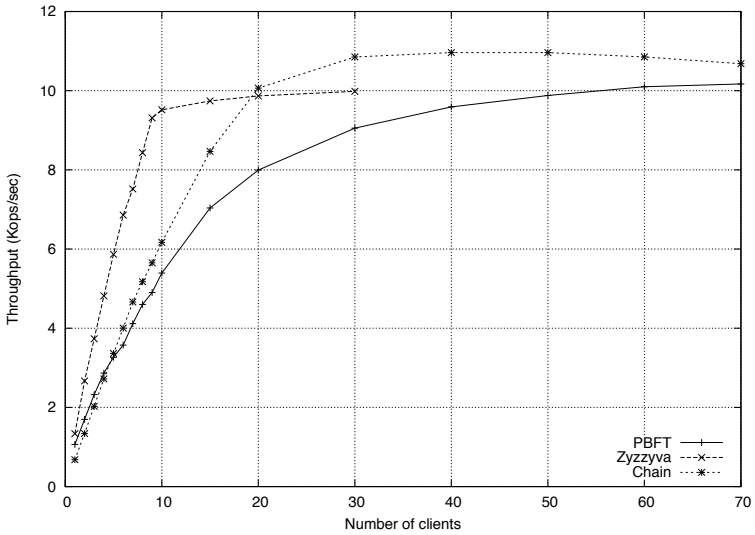


Figure 6.15: Throughput of the various protocols for the 0/4 benchmark ($t = 1$).

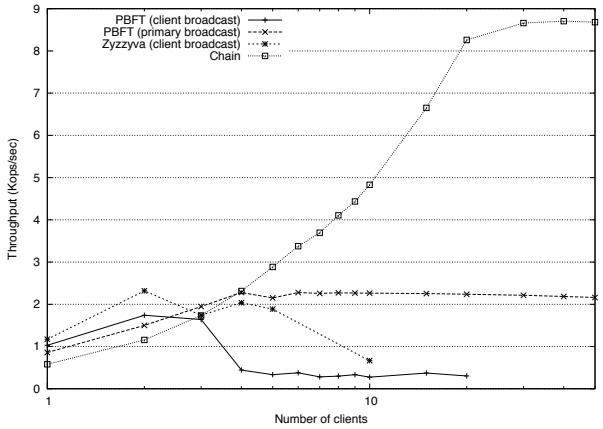


Figure 6.16: Throughput of the various protocols for the 4/0 benchmark ($t = 1$).

An interesting observation that can be made about Figure 6.16 is that the performance of both Zyzzyva and PBFT drops when the client broadcast optimization is used. In contrast, the performance of PBFT when the primary forwards requests remains almost constant after the peak throughput has been reached. This result may seem surprising given that [KAD+07, CL99] recommend to use the client broadcast optimizations when requests are large, in order to avoid to the primary the costly operation of forwarding requests. Nevertheless, while surprising at first, this result can be explained by the fact that simultaneous broadcast of large messages by different clients result in collisions and buffer overflows[11] (since there is no flow control in UDP which is used as a transport layer in the PBFT code base and, consequently, in all implementations evaluated here). This explains why enabling the clients to concurrently broadcast messages drastically reduces performance. On the contrary, when the primary forwards messages, there are fewer collisions, which explains the better performance we observe. We do not present results for Zyzzyva disabling the client broadcast optimizations as there is a bug in the current Zyzzyva implementation that prevented us from running experiments. We could have presented performance results obtained with our implementation of AZyzzyva; its throughput peak is of about 1,8kops/sec. However, in the case of message losses, clients in our AZyzzyva implementation simply re-issue requests. This results in a significant increase of the traffic and is probably not as efficient as the mechanism implemented in Zyzzyva to drive servers to a coherent state when messages are lost.

Impact of the request size

In this section we study how algorithms are impacted by the size of requests. Figure 6.17 shows the peak throughput of *Chain*, PBFT and Zyzzyva as a function of the request size for one tolerated fault. To obtain the peak throughput of PBFT and Zyzzyva, we benchmarked both algorithms with and without client broadcast optimizations and with different batching sizes for Zyzzyva. Interestingly, the behavior we observe is similar to that observed using simulations in [SDM+08]: payload increase diminishes differences between PBFT and Zyzzyva. Indeed, starting from 128B payloads, both algorithms have almost identical performance. Figure 6.17 also shows that *Chain* sustains high peak throughput with all message sizes.

6.7.3 Fault scalability

One important characteristic of BFT-SMR algorithms is their behavior when the number of tolerated faults increases. Figure 6.18 depicts the performance of *Chain* for the 4/0 benchmark when the number of faults varies between 1 and

[11] Note that similar performance drops with large UDP packets have already been observed in the context of broadcast algorithms [ACL04, Ban07]. For instance, a recent study made by the authors of the JGroups toolkit [Ban07] showed that with 5K messages, their TCP stack achieves up to 5 times the throughput of their UDP stack, even if the latter includes some flow control mechanisms.

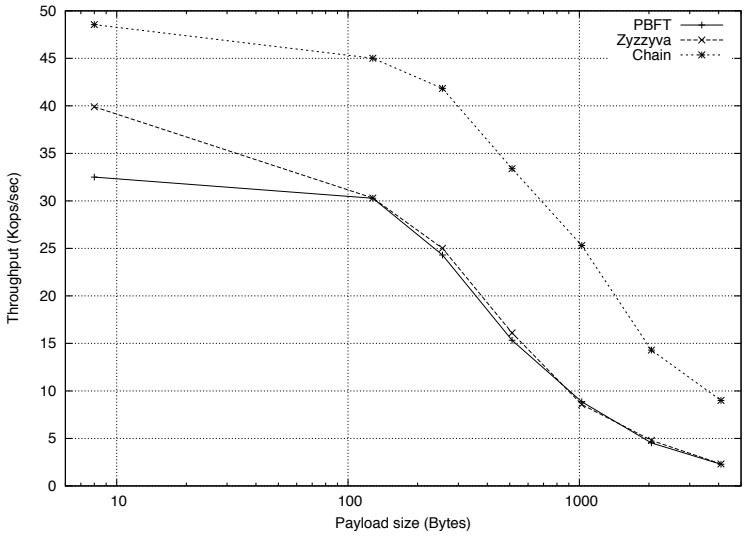


Figure 6.17: Peak throughput of the various protocols as a function of request size ($t = 1$).

3.[12] We do not present results for PBFT and Zyzzyva as it has been shown in [KAD+07] that their peak throughput suffers only a slight impact with the increase of t. Figure 6.18 shows that this is also the case for *Chain*. The peak throughput at $t = 3$ is only 3,5% lower than that achieved at $t = 1$. We also observe that when the number of tolerated faults increases, *Chain* requires more clients to reach its peak throughput. This can be explained by the fact that the chain length increases when f increases. Hence, more clients are needed to feed the *Chain*, which is necessary for *Chain* to reach its peak throughput, as explained in Section 6.7.2.

6.7.4 Impact of slow clients

In this section we examine the impact of having clients connected to servers through slow links. The motivation for performing this experiment is that it is sometimes the case that interserver communication is fast (e.g. when servers are organized in a cluster), whereas clients-to-servers communication is slow (e.g. when clients are remote). For that purpose, we connected all clients to a Fast Ethernet switch, which was itself connected to the Gigabit Ethernet switch interconnecting servers. Using this topology, all requests and replies are going through the same link and every client can only send and receive messages at a throughput of 100Mb/sec. Results presented in this section show that *Chain* still achieves higher peak throughput than PBFT and Zyzzyva. These results also show that PBFT and Zyzzyva exhibit better performance with 4k requests than with 0k requests. This result is explained by the fact that the biggest cost with 0k requests is that of replying to clients, which penalizes PBFT and Zyzzyva in

[12] The curve for $t = 1$ is the same as the one given in Figure 6.16, but without a logarithmic scale.

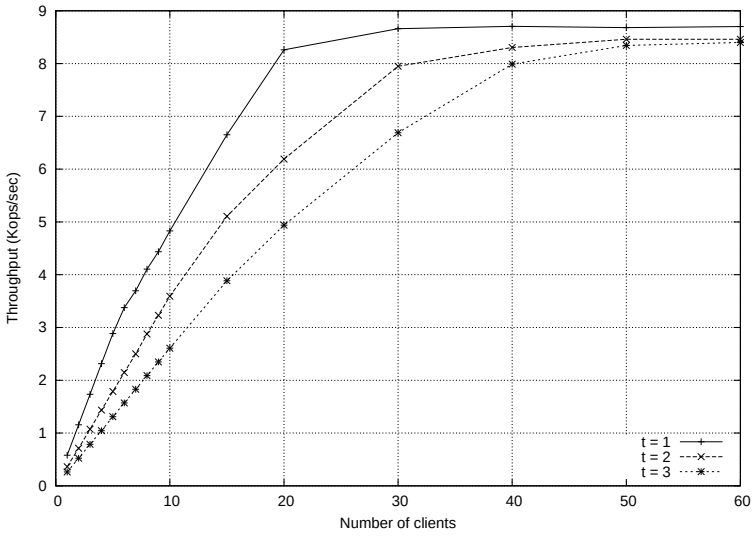


Figure 6.18: Impact of the number of tolerated faults on the throughput of the chain protocol.

which all replicas reply to clients. With 4k requests, the biggest cost is that of sending/receiving requests, masking the fact that all replicas send replies.

0/0 benchmark

Figure 6.19 shows the throughput of the various algorithms as a function of the number of clients for the 0/0 benchmark when one server failure is tolerated ($t = 1$). Interestingly, we see that the performance of *Chain* with up to 50 clients is very similar to that obtained with clients connected to the Gigabit switch (Figure 6.14). With more than 50 clients the performance is slightly worse (the peak throughput is 7,4% lower). We explain this by the fact that, for each request, *Chain* only requires two messages to be exchanged between a client and servers. In the case of 0KB requests (for which only a header is sent), slow clients can sustain the same sending rate as the one that they sustain when they are connected on the Gigabit switch. Indeed, the bottleneck in this case is not created by clients sending requests, but by servers needing to receive/forward messages and perform cryptographic computations on these requests.

Another observation we can make about Figure 6.19 is that Zyzzyva and PBFT are much more impacted by the presence of slow clients than *Chain* (the same observation holds when the client broadcast optimization is not used). For instance, the peak throughput of Zyzzyva is 55% lower than the one obtained with clients connected to the Gigabit switch. We explain this result by the fact that both Zyzzyva and PBFT require all servers to send a reply to the client. In the

considered microbenchmark (0/0), all servers send a reply with the same size[13]. These replies all transit the same slow network link, which explains the observed throughput decrease for PBFT and Zyzzyva. Moreover, we observe that PBFT and Zyzzyva achieve almost the same peak throughput. The reason for this is that the maximum throughput they can achieve is that of the slow network link interconnecting the clients and the servers. With a sufficiently large number of clients, both PBFT and Zyzzyva are able to saturate this link. Finally, we observe that batching in Zyzzyva (not shown in Fig. 6.19) does not improve its performance. This is due to the fact that the cost of communicating with clients dominates other costs (in particular request processing by servers and inter-servers communication).

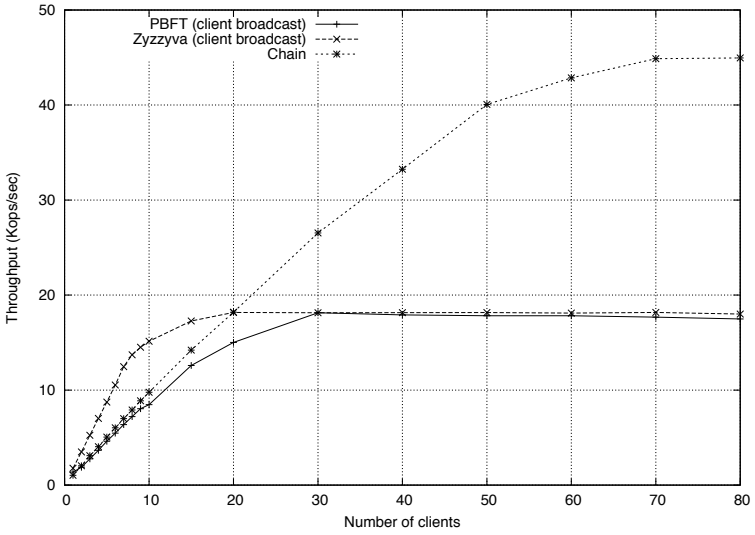


Figure 6.19: Throughput of the various protocols for the 0/0 benchmark with slow clients ($t = 1$).

4/0 benchmark

Figure 6.20 shows the throughput of various algorithms as a function of the number of clients, for the 4/0 benchmark and when one failure is tolerated ($t = 1$). The results are quite different from those obtained when clients are connected using the Gigabit switch. First of all, Zyzzyva (with client broadcast enabled) and *Chain* obtain the same peak throughput. This peak throughput (89Mb/sec) is very close to the link bandwidth (100Mb/sec). The good throughput achieved by *Chain* is again a consequence of the pipelining pattern *Chain* uses to broadcast messages. More interesting is the good throughput achieved by Zyzzyva. This shows that Zyzzyva achieves good performance when there is no contention on servers (i.e. when clients issue requests at a lower rate than that at which servers can handle them). This also confirms our interpretation that the bad throughput Zyzzyva achieved when clients were connected through a Gigabit

[13]The optimization that consists in having all servers but one only sending a digest of the reply does not improve performance as the payload of replies is 0KB.

switch (Figure 6.16) is probably due to packet losses in the UDP stack due to network congestion. Finally, we see that PBFT with the client broadcast optimization enabled suffers a significant performance drop when there are more than 8 clients. We have no explanation for this behavior.

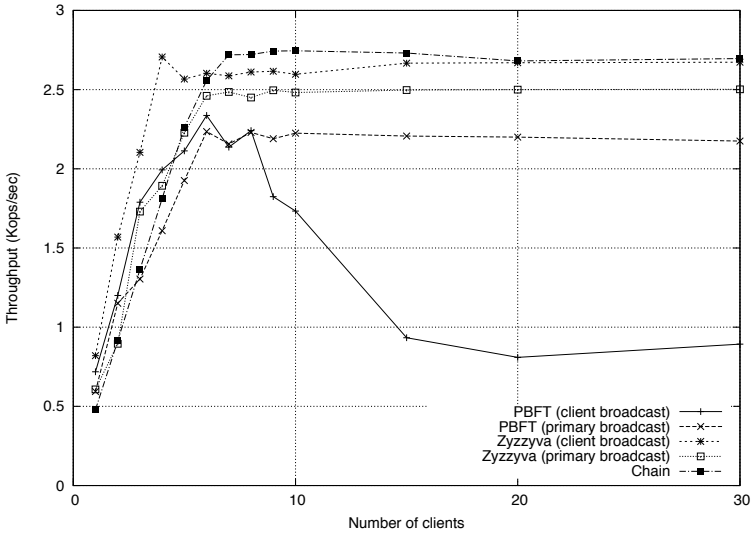


Figure 6.20: Throughput of the various protocols for the 4/0 benchmark with slow clients ($t = 1$).

6.7.5 Switching cost

This section assesses the cost of switching. This cost obviously depends on the two *Abstract* instances involved in the switching processes. Nevertheless, both instances do not contribute equally to this cost. Indeed, the cost of panicking/aborting can be considered identical for the various *Abstract* implementations. This is not the case with the *Abstract* initialization cost, as this requires the new *Abstract* instance to execute all requests contained in the init history. We decided to evaluate the cost of switching from *DEC* to *Backup* implemented over PBFT (denoted by *Backup-PBFT*).

To evaluate the switching cost, we perform the following experiments: we feed the local history of *DEC* servers with r requests of size s. We then issue 100 successive requests (requests are not concurrent). For every request, we force a *DEC* client to panic (without waiting for a timeout). Consequently, *DEC* servers generate, for each request, the same abort histories containing r requests of size s. Moreover, we reset the state of *Backup-PBFT* after every request to force it to process all abort histories identically. We evaluate the time it takes for the 100 successive requests to be processed and reproduce the experiment until the variance becomes negligible.

Figure 6.21 shows the switching cost (in sec) as a function of the history size. We run experiments with fast and slow clients (we use the corresponding network topology described previously) and with two request sizes: 0k and 4k. We perform checkpointing in *DEC* every 128 requests. To account for requests that might be

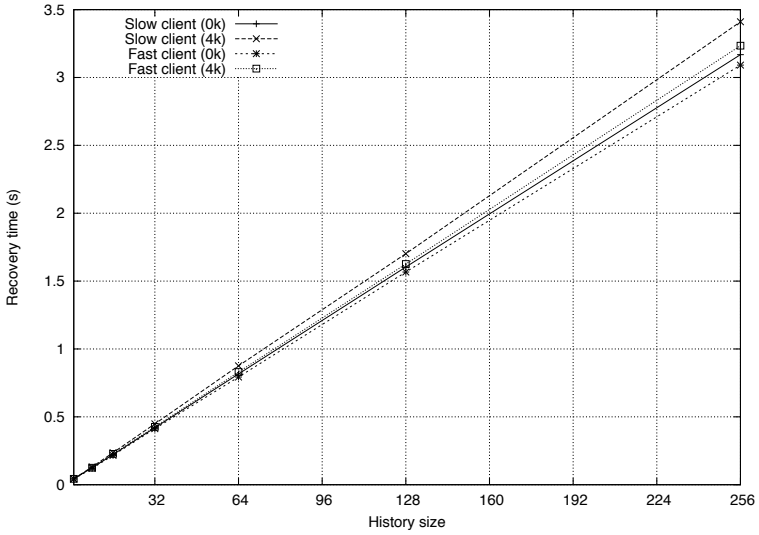


Figure 6.21: Switching time as a function of the history size ($t = 1$).

received by servers while they are performing a checkpoint, we consider that the history size can grow up to 256 requests. Not surprisingly, we observe that the switching cost is linear with respect to the history size in the 4 considered cases. Moreover, we observe that slow clients do not have a big impact on the switching cost. We attribute this to the fact that the biggest factor in the switching cost is the time it takes for *Backup-PBFT* to execute all requests in the abort histories. Our experiments (see Section 6.7.2 and 6.7.4) show that this time is very similar with slow and fast clients (for instance, with 0k requests, *Backup-PBFT* achieves a throughput of 1456req/s with one fast client against 1373req/s with one slow client).

Figure 6.22 shows the impact of the number of tolerated faults t on the switching cost with 0k requests for history sizes ranging from 1 to 256. Note that we use a logarithmic scale on both axis. We observe that the switching cost is proportional to t. This makes sense since increasing t also increases the number of servers, and thus also the number of abort histories that need to be sent and processed. Moreover, we observe that increasing f from 1 to 2 has a higher impact than increasing f from 2 to 3. We explain this by the fact that, as remarked in Section 6.7.1, PBFT achieves a better latency with $t = 3$ than with $t = 2$. *Backup-PBFT* can thus handle abort histories at a higher throughput (with one client sending 0k requests, *Backup-PBFT* achieves a throughput of 1353 req/s at $t = 2$ against 1586 req/s at $t = 3$).

Finally, we would like to comment on the absolute value of the switching cost. Figure 6.21 shows that when one fault is tolerated, this cost ranges from 42ms to 3,5s. This cost might seem high. Nevertheless, we don't believe this is an issue for the following reasons. First of all, we did not try to optimize the switching cost. All messages transfers between *DEC* and *Backup-PBFT* pass through the client. A possible way to optimize this would, for instance, be to facilitate *Abstract* to have servers directly communicate. Perhaps even better optimization would be to

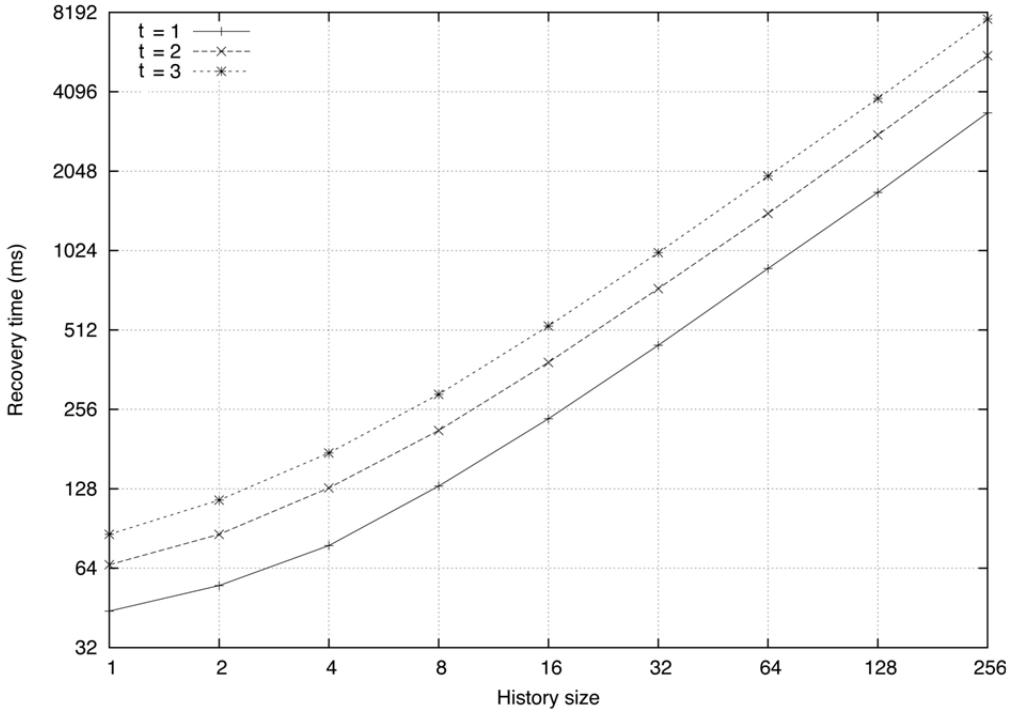


Figure 6.22: Switching time as a function of history size ($t = 1$, $t = 2$, and $t = 3$).

have servers of the two *Abstract* instances share a request history, thus avoiding most message exchanges. Moreover, there are two cases where switching should be used: to ensure progress when the first *Abstract* instance aborted, or to improve performance by switching to an *Abstract* instance achieving better performance. The former case should be very rare. Indeed, if failures were to be frequent, this would mean that only *Abstract* instances tolerating faults should be used. In the latter case, two remarks hold: (1) switching can be triggered when the history size is small, thus significantly decreasing its cost, and (2) it makes sense to spend up to 3s switching if the new *Abstract* instances improves performance significantly. For instance, our performance evaluation shows that it is worth switching from PBFT to *DEC* when there is no contention or from Zyzzyva to *Chain* when big requests are sent or many clients are simultaneously issuing requests.

6.7.6 Lightweight checkpointing subprotocol

In the above, we evaluated *DEC* and *Abstract* optimized with a *lightweight checkpoint subprotocol* (LCS) to truncate histories every CHK requests (where, in Section 6.7, $CHK = 128$). Here, we explain our simple LCS and its impact on our *Abstract* implementations presented in Section 6.5.

LCS consists in the following:

1. every server s_i increments the checkpoint counter cc and sends it along with the digest of its local state to every other server (using simple point-to-point MACs), when its (non-checkpointed suffix of) local history reaches CHK

requests. Then, s_i triggers a checkpoint timer.

2. if the timer expires and there is no checkpoint, the server aborts all future requests.

3. If server s_i receives the digest of the same state st with the same checkpoint counter number cc greater than $lastcc$ (initially $lastcc = 0$) from *all* servers, s_i: (a) truncates its local history and checkpoints its state to st and (b) stores cc to variable $lastcc$. Such a checkpointed state (referred to as st_{cc}) becomes a prefix of servers' local histories to which new requests are appended and is treated as such in all operations on local histories in our algorithms. Moreover, every abort or commit history of length at most $cc * CHK$ is considered to be a prefix of st_{cc}.

LCS has no impact on *DEC* and *Chain* as described in Sections 6.5.1 and 6.5.2, with a single exception, related to client extraction of abort histories from the received ABORT messages (see Step 3b.2, Sec. 6.5.1). Namely, if the client receives a history from some server s_i consisting of a checkpointed state followed by CHK requests, the client will first collapse all such histories into the single checkpointed state (i.e., the client will perform the checkpoint on behalf of the server). Only in case the client cannot retrieve $t+1$ confirmations of (some) checkpointed state when executing *DEC* Step 3b.2. (equivalent to Step 3b.2b.2 of *Chain*) in this way, the client will repeat the procedure described in this step with server histories as received from servers, i.e., precisely as described in Step 3b.2., Section 6.5.1.

It is not difficult to extend our proofs of Section 6.6 to account for LCS.

7 Concluding Remarks

This thesis proposed reusable abstractions for asynchronous distributed algorithms that tolerate malicious (Byzantine) failures (BFT algorithms). A number of read/write storage, consensus and state machine replication algorithms that use these abstractions and provide optimal resilience to malicious failures and/or optimal complexity were presented. We now briefly summarize our contributions and outline a few open issues and directions for future investigation.

Refined Quorum Systems. This thesis introduced the notion of *refined quorum systems* (RQS) and argued that this is a useful notion to reason about optimally resilient and best-case latency efficient distributed object implementations assuming general adversary structures. Refined quorum systems were shown to be necessary and sufficient (or, in a sense, minimal) for implementing an important class of *atomic* objects, namely single-writer multi-reader atomic storage and consensus. This minimality holds when we indeed require atomicity and do not rely on authentication primitives to cope with Byzantine failures in best-case executions.

Roughly speaking, denoting the best possible latency of an object implementation by l_1 (this can be measured by the best possible latency in synchronous, uncontended and failure-free situations), i.e., 1 round in the case of storage, or 2 message delays in the case of (Byzantine and indulgent) consensus, and by l_2 and l_3, incrementally, the next best possible latencies according to the corresponding metric, we proposed two RQS-based object implementations that achieve a latency of l_i whenever a quorum of class i is available and best-case conditions (namely, synchrony and no-contention) are met. Since Property 1 of RQS (defined on class 3 quorums) is anyway necessary for any resilient implementation of distributed storage and consensus in an asynchronous environment, there is no need for refining quorums further.

It might be important to notice here that the very notion of a refined quorum system helps highlight the information structure of optimally resilient and best-case efficient atomic object implementations (at least those implementing the abstractions of atomic storage or consensus). Basically, these implementations go through at most three "rounds" in best-case conditions and fall into a backup

subprotocol in case of asynchrony or contention. A novel algorithmic scheme we used in both algorithms consists of appending the ids of (class 2) quorums, to written/proposed values. This is key to combining graceful degradation (i.e., achieving both latencies l_1 and l_2) with optimal resilience.

Our study opens several research directions. For example, it is intriguing to determine:

- the load and availability of RQS [NW94],
- how RQS can be optimally placed in the network [GGM+06],
- the extension of RQS with respect to asymmetric read and write quorums [MAD02b], and
- how to devise algorithms that cope with unknown RQS/adversary structures.

High Resolution Timestamps. The second notion introduced in this thesis is that of *high resolution timestamps*, a timestamping mechanism based on matrix clocks. In a typical read/write storage implementation only timestamps of writer(s) are associated to a written value. A high resolution timestamp is used in conjunction with such classical timestamps, to allow detection and filtering of malicious processes. Our high resolution timestamping mechanism requires reader's to write (meta-data in the form of readers' local timestamps). Moreover, the thesis presented safe and regular Byzantine fault-tolerant storage algorithms based on high resolution timestamps; these algorithms are the first to combine optimal resilience with the worst-case time complexity of two communication round-trips, which we prove (in the combination with the results of [ACKM06]) optimal.

Using our results and the transformation from regular to atomic storage [GR06], we were able to narrow down the range for the optimal worst-case complexity of optimally resilient BFT atomic storage to between 2 and 4 communication round-trips. The exact complexity remains a challenging open problem.

To complement these results, we also analyzed the resilience/performance tradeoff in BFT atomic storage, by establishing a tight bound on *fast* single-writer multi-reader BFT atomic storage algorithms, i.e., the algorithms in which read and write operations always complete in a single communication round-trip. We show that this tradeoff depends on the number of readers R in the system (which is in line with the analog result in the crash-failure model [DGLC04]), even in the authenticated Byzantine failure model, in which processes can rely on digital signatures.

It should be emphasized again that the scope of this thesis covers single-writer multi-reader storage implementations (SWMR), typically used in construction of agreement algorithms (e.g., [GL03, ACKM06]). Whereas our optimal SWMR algorithms based on refined quorums and high resolution timestamps can be used as a building block in classical algorithms [AW98, GR06] to construct multi-writer multi-reader (MWMR) storage algorithms, this would yield only near-optimal solutions in terms of optimal latency (albeit while preserving optimal

resilience). While, for example, our algorithms given in the context of BFT state machine replication (see below) suggest that our abstractions (in this case refined quorums) can be successfully used even in the presence of multiple writers (that can be mapped to multiple state machine replication clients) to obtain algorithms with optimal latency, more research needs to be conducted in order to obtain the exact characterization of the optimal best-case and worst-case latency in the context of MWMR storage.

ABsTRACT. Finally, we introduced *Abstract*, a generic abstraction that simplifies the development and maintenance of BFT state machine replication algorithms. *Abstract* resembles state machine replication with one exception: it may sometimes abort a client request, depending on the generic *Abstract* Non-Triviality property. *Abstract* instances are composable; this allows many new BFT state machine replication algorithms to be built as a composition of independent instances of *Abstract*. To this end, we provide a well-defined interface to interconnect *Abstract* instances, which allows any ordering among *Abstract* instances. This permits, for example, to first try to replicate the request using an *optimistic Abstract* that can be implemented very efficiently and that optimizes performance under some best-case conditions, and then fall back to a slower and more faithful *Abstract*. Typically, an implementation of such an optimistic *Abstract* is significantly simpler then developing a full fledged BFT state machine replication algorithm.

To illustrate this approach, two new optimally resilient BFT state machine replication algorithms are described in this thesis. These algorithms use a novel generic BFT-SMR algorithm to compose different *Abstract* instances called *Modular BFT-SMR*. The first algorithm (called *DEC*), that makes use of the notion of refined quorums introduced earlier in this thesis, has the lowest time complexity among all BFT state machine replication algorithms we know of, in synchronous periods that are free from contention and failures. The second algorithm (called *Chain*) has the highest peak throughput in failure-free and synchronous periods.

Several directions can be interesting to explore with *Abstract* in mind. It would be interesting to devise *Abstract* implementations for other meaningful definitions of the non-triviality property. There is also room for optimizing the switching mechanism between *Abstract* instances. The switching mechanism could for instance be improved by facilitating inter-replica communication, rather than having all communication going through the client. Finally, we believe that a very interesting research challenge is to define and evaluate heuristics for dynamic establishment of the switching order among *Abstract* instances in order to improve performance.

Finally, we believe that machine verification of BFT algorithms (and BFT state machine replication algorithms, in particular) is a very challenging and interesting topic. While this thesis contains some basic steps in this direction (e.g., model checking *Modular BFT-SMR* and *DEC*), much work lies ahead.

Bibliography

[ABD95] Hagit Attiya, Amotz Bar-Noy, and Danny Dolev. Sharing memory robustly in message-passing systems. *Journal of the ACM*, 42(1):124–142, 1995.

[ACC+05] Michael Abd-El-Malek, William V. Courtright II, Chuck Cranor, Gregory R. Ganger, James Hendricks, Andrew J. Klosterman, Michael Mesnier, Manish Prasad, Brandon Salmon, Raja R. Sambasivan, Shafeeq Sinnamohideen, John D. Strunk, Eno Thereska, Matthew Wachs, and Jay J. Wylie. Ursa minor: versatile cluster-based storage. In *Proceedings of the 4th conference on USENIX Conference on File and Storage Technologies*, pages 5–5, Berkeley, CA, USA, 2005. USENIX Association.

[ACKM06] Ittai Abraham, Gregory V. Chockler, Idit Keidar, and Dahlia Malkhi. Byzantine disk paxos: optimal resilience with Byzantine shared memory. *Distributed Computing*, 18(5):387–408, 2006.

[ACKM07] Ittai Abraham, Gregory Chockler, Idit Keidar, and Dahlia Malkhi. Wait-free regular storage from Byzantine components. *Inf. Process. Lett.*, 101(2):60–65, 2007.

[ACL04] Takoua Abdellatif, Emmanuel Cecchet, and Renaud Lachaize. Evaluation of a group communication middleware for clustered J2EE application servers. In *International Symposium on Distributed Objects and Applications (DOA)*, pages 1571–1589, 2004.

[AFH+07] Marcos K. Aguilera, Svend Frolund, Vassos Hadzilacos, Stephanie L. Horn, and Sam Toueg. Abortable and query-abortable objects and their efficient implementation. In *Proceedings of the 26th annual ACM symposium on Principles of distributed computing*, pages 23–32, New York, NY, USA, 2007.

[AGG+05] Michael Abd-El-Malek, Gregory Ganger, Garth Goodson, Michael Reiter, and Jay Wylie. Fault-scalable Byzantine fault-tolerant ser-

vices. In *Proceedings of the 20th ACM symposium on Operating systems principles*, pages 59–74, October 2005.

[AGGV05] Gildas Avoine, Felix C. Gärtner, Rachid Guerraoui, and Marko Vukolić. Gracefully degrading fair exchange with security modules. In *Proceedings of the 5th European Dependable Computing Conference*, pages 55–71, 2005.

[AGK05] Hagit Attiya, Rachid Guerraoui, and Petr Kouznetsov. Computing with reads and writes in the absence of step contention. In *Proceedings of the 19th International Symposium on Distributed Computing*, pages 122–136, October 2005.

[AW98] Hagit Attiya and Jennifer Welch. *Distributed Computing. Fundamentals, Simulations, and Advanced Topics.* McGraw-Hill, 1998.

[Awe85] Baruch Awerbuch. Complexity of network synchronization. *Journal of the ACM*, 32(4):804–823, October 1985.

[Ban07] Bela Ban. Performance tests jgroups 2.5, June 2007. http://www.jgroups.org/javagroupsnew/perfnew/Report.html.

[BDFG03] Romain Boichat, Partha Dutta, Svend Frölund, and Rachid Guerraoui. Deconstructing Paxos. *SIGACT News in Distributed Computing*, 34(1):47–67, 2003.

[BGMR01] Francisco V. Brasileiro, Fabíola Greve, Achour Mostéfaoui, and Michel Raynal. Consensus in one communication step. In *PaCT '01: Proceedings of the 6th International Conference on Parallel Computing Technologies*, pages 42–50, London, UK, 2001. Springer-Verlag.

[BHK+99] John Black, Shai Halevi, Hugo Krawczyk, Ted Krovetz, and Phillip Rogaway. UMAC: Fast and secure message authentication. *Lecture Notes in Computer Science*, 1666:216–233, 1999.

[BJ95] K.P. Birman and T.A. Joseph. Reliable communication in the presence of failures. *ACM Transactions on Computer Systems*, 5(1):47–76, February 1995.

[BO83] Michael Ben-Or. Another advantage of free choice: Completely asynchronous agreement protocols. In *Proc. Second Ann. ACM Symp. on Principles of Distributed Computing*, pages 27–30, 1983.

[Bra84] Gabriel Bracha. An asynchronous [(n-1)/3]-resilient consensus protocol. In *Proceedings of the 3rd Annual Symposium on Primciples of Distributed Computing*, pages 154–162, 1984.

[BT85] Gabriel Bracha and Sam Toueg. Asynchronous consensus and broadcast protocols. *Journal of the ACM*, 32(4):824–840, October 1985.

[CBS06] Bernadette Charron-Bost and Andre Schiper. Improving fast Paxos: being optimistic with no overhead. In *Proceedings of the 12th Pacific Rim International Symposium on Dependable Computing*, pages 287–295, Washington, DC, USA, 2006. IEEE Computer Society.

[CDH+06] Byung-Gon Chun, Frank Dabek, Andreas Haeberlen, Emil Sit, Hakim Weatherspoon, M. Frans Kaashoek, John Kubiatowicz, and Robert Morris. Efficient replica maintenance for distributed storage systems. In *Proceedings of the 3rd conference on Networked Systems Design & Implementation*, pages 4–4, Berkeley, CA, USA, 2006. USENIX Association.

[CGK07] Gregory Chockler, Rachid Guerraoui, and Idit Keidar. Amnesic distributed storage. In *Proceedings of the 21st International Symposium on Distributed Computing*, pages 139–151, September 2007.

[CGKV] Grogory Chockler, Rachid Guerraoui, Idit Keidar, and Marko Vukolić. Reliable distributed storage. *IEEE Computer. To appear.*

[CGR07] Tushar D. Chandra, Robert Griesemer, and Joshua Redstone. Paxos made live: an engineering perspective. In *Proceedings of the 26th Annual ACM symposium on Principles of Distributed Computing*, pages 398–407, New York, NY, USA, 2007.

[Che07] Wei Chen. Abortable consensus and its application to probabilistic atomic broadcast. Technical Report MSR-TR-2006-135, Microsoft Research, September 2007.

[CKPS01] Christian Cachin, Klaus Kursawe, Frank Petzold, and Victor Shoup. Secure and efficient asynchronous broadcast protocols. In *Advances in Cryptology*, Lecture Notes in Computer Science. International Association for Cryptologic Research, Springer-Verlag, 2001.

[CKS00] Christian Cachin, Klaus Kursawe, and Victor Shoup. Random oracles in constantinople: Practical asynchronous Byzantine agreement using cryptography. In *Proceedings of the 19th Annual ACM Symposium on Principles of Distributed Computing*, pages 123–132, 2000.

[CL99] Miguel Castro and Barbara Liskov. Practical Byzantine fault tolerance. In *Proceedings of the 3rd Symposium on Operating Systems Design and Implementation*, February 1999.

[CM05] Gregory Chockler and Dahlia Malkhi. Active disk Paxos with infinitely many processes. *Distrib. Comput.*, 18(1):73–84, 2005.

[CML+06] James Cowling, Daniel Myers, Barbara Liskov, Rodrigo Rodrigues, and Liuba Shrira. HQ replication: A hybrid quorum protocol for Byzantine fault tolerance. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementations*, November 2006.

[CSS07] Christian Cachin, Abhi Shelat, and Alexander Shraer. Efficient fork-linearizable access to untrusted shared memory. In *Proceedings of the 26th annual ACM symposium on Principles of distributed computing*, pages 129–138, New York, NY, USA, 2007. ACM.

[CT96] Tushar Deepak Chandra and Sam Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2):225–267, March 1996.

[DGLC04] Partha Dutta, Rachid Guerraoui, Ron R. Levy, and Arindam Chakraborty. How fast can a distributed atomic read be? In *Proceedings of the 23rd annual ACM symposium on Principles of distributed computing*, pages 236–245, July 2004.

[DGLV05] Partha Dutta, Rachid Guerraoui, Ron R. Levy, and Marko Vukolić. How fast can a distributed atomic read be? Technical Report LPD-REPORT-2005-001, EPFL, School of Computer and Communication Sciences, Lausanne, Switzerland, 2005.

[DGV05] Partha Dutta, Rachid Guerraoui, and Marko Vukolić. Best-case complexity of asynchronous Byzantine consensus. Technical Report 200499, Swiss Federal Institute of Technology (EPFL), School of Computer and Communication Sciences, Lausanne, Switzerland, 2005.

[DLS88] Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. Consensus in the presence of partial synchrony. *Journal of the ACM*, 35(2):288–323, April 1988.

[Dou00] Assia Doudou. *Abstractions for Byzantine-resilient state machine replication.* PhD thesis, School of Computer and Communication Sciences, EPFL, Lausanne, Switzerland, February 2000.

[DS97] Assai Doudou and André Schiper. Muteness detectors for consensus with Byzantine processes. Technical report, EPFL – Département d'Informatique, Lausanne, Switzerland, 1997.

[Fid91] Colin Fidge. Logical time in distributed computing systems. *Computer*, 24(8):28–33, 1991.

[FLP85] Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374–382, April 1985.

[Gaf98] Eli Gafni. Round-by-round fault detectors (extended abstract): unifying synchrony and asynchrony. In *Proceedings of the 17th annual ACM symposium on Principles of distributed computing*, pages 143–152, June 1998.

[GGK07] Seth Gilbert, Rachid Guerraoui, and Dariusz R. Kowalski. On the message complexity of indulgent consensus. In *Proceedings of*

the 20th International Symposium on Distributed Computing, pages 283–297. Springer, 2007.

[GGL03] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The google file system. *SIGOPS Oper. Syst. Rev.*, 37(5):29–43, 2003.

[GGM+06] Daniel Golovin, Anupam Gupta, Bruce M. Maggs, Florian Oprea, and Michael K. Reiter. Quorum placement in networks: Minimizing network congestion. In *Proceedings of the 15th annual ACM symposium on Principles of distributed computing*, pages 16–25, July 2006.

[Gif79] David K. Gifford. Weighted voting for replicated data. In *Proceedings of the 7th ACM symposium on Operating systems principles*, pages 150–162, December 1979.

[GKLQ07] Rachid Guerraoui, Dejan Kostic, Ron R. Levy, and Vivien Quema. A high throughput atomic storage algorithm. In *Proceedings of the 27th International Conference on Distributed Computing Systems*, page 19, Washington, DC, USA, 2007. IEEE Computer Society.

[GL03] Eli Gafni and Leslie Lamport. Disk paxos. *Distributed Computing*, 16(1):1–20, 2003.

[GLPQ06] Rachid Guerraoui, Ron R. Levy, Bastian Pochon, and Vivien Quema. High throughput total order broadcast for cluster environments. In *Proceedings of the International Conference on Dependable Systems and Networks*, pages 549–557, Washington, DC, USA, 2006. IEEE Computer Society.

[GLV06] Rachid Guerraoui, Ron R. Levy, and Marko Vukolić. Lucky read/write access to robust atomic storage. In *Proceedings of the International Conference on Dependable Systems and Networks*, pages 125–136, Washington, DC, USA, 2006. IEEE Computer Society.

[GQV08] Rachid Guerraoui, Vivien Quéma, and Marko Vukolić. The next 700 BFT protocols. Technical Report LPD-REPORT-2008-008, EPFL, School of Computer and Communication Sciences, Lausanne, Switzerland, 2008.

[GR06] Rachid Guerraoui and Luís Rodrigues. *Introduction to Reliable Distributed Programming*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006.

[Gra78] Jim Gray. Notes on database operating systems. In *Operating Systems — An Advanced Course*, number 66 in Springer Verlag (LNCS). Springer-Verlag, 1978.

[Gue00] Rachid Guerraoui. Indulgent algorithms. In *Proceedings of the 19th Annual ACM Symposium on Principles of Distributed Computing*, pages 289–298, July 2000.

[GV06] Rachid Guerraoui and Marko Vukolić. How fast can a very robust read be? In *Proceedings of the 25th annual ACM symposium on Principles of distributed computing*, pages 248–257, New York, NY, USA, 2006. ACM.

[GV07] Rachid Guerraoui and Marko Vukolić. Refined quorum systems. In *Proceedings of the 26th annual ACM symposium on Principles of distributed computing*, pages 119–128, New York, NY, USA, 2007.

[GV08] Rachid Guerraoui and Marko Vukolić. A scalable and oblivious atomicity assertion. In *Proceedings of the 19th International Conference on Concurrency Theory*, pages 52–66, London, UK, 2008. Springer-Verlag.

[GWGR04] Garth Goodson, Jay Wylie, Gregory Ganger, and Michael Reiter. Efficient Byzantine-tolerant erasure-coded storage. In *Proceedings of the International Conference on Dependable Systems and Networks*, pages 135–144, 2004.

[Her91] Maurice Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems*, 13(1):124–149, January 1991.

[HM97] Martin Hirt and Ueli Maurer. Complete characterization of adversaries tolerable in secure multi-party computation (extended abstract). In *Proceedings of the 16th annual ACM symposium on Principles of distributed computing*, pages 25–34, 1997.

[HW90] Maurice Herlihy and Jeannette Wing. Linearizability: a correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, July 1990.

[ISO06] ISO/IEC 14764:2006, 2006.

[Jay03] Prasad Jayanti. Adaptive and efficient abortable mutual exclusion. In *Proceedings of the 22nd Annual ACM symposium on Principles of Distributed Computing*, pages 295–304, New York, NY, USA, 2003.

[JCT98] Prasad Jayanti, Tushar Deepak Chandra, and Sam Toueg. Fault-tolerant wait-free shared objects. *Journal of the ACM*, 45(3):451–500, 1998.

[JM03] Flavio P. Junqueira and Keith Marzullo. Synchronous consensus for dependent process failures. In *Proceedings of the 23rd IEEE International Conference on Distributed Computing Systems*, pages 274–283, May 2003.

[KAD+] Ramakrishna Kotla, Lorenzo Alvisi, Mike Dahlin, Allen Clement, and Edmund Wong. Zyzzyva: speculative Byzantine fault tolerance. Technical Report UTCS-TR-07-40, University of Texas at Austin, Austin, TX, USA.

[KAD+07] Ramakrishna Kotla, Lorenzo Alvisi, Mike Dahlin, Allen Clement, and Edmund Wong. Zyzzyva: speculative Byzantine fault tolerance. In *Proceedings of 21st ACM SIGOPS symposium on Operating systems principles*, pages 45–58, New York, NY, USA, 2007. ACM.

[KHGFZ04] Deepak R. Kenchammana-Hosekote, Richard A. Golding, Claudio Fleiner, and Omer A. Zaki. The design and evaluation of network raid protocols. Technical Report RJ 10316, IBM Almaden Research Center, 2004.

[KS05] Klaus Kursawe and Victor Shoup. Optimistic asynchronous atomic broadcast. In *Proceedings of 32nd International Colloquium on Automata, Languages and Programming*, pages 204–215, 2005.

[KS06] Idit Keidar and Alexander Shraer. Timeliness, failure-detectors, and consensus performance. In *Proceedings of the 25th annual ACM symposium on Principles of distributed computing*, pages 169–178, 2006.

[Lam78] Leslie Lamport. Time, clocks and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.

[Lam86] Leslie Lamport. On interprocess communication. *Distributed computing*, 1(1):77–101, May 1986.

[Lam98] Leslie Lamport. The part-time parliament. *ACM Transactions on Computer Systems*, 16(2):133–169, 1998.

[Lam02] Leslie Lamport. *Specifying Systems, The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley, 2002.

[Lam03] Leslie Lamport. Lower bounds for asynchronous consensus. In *Future Directions in Distributed Computing*, Springer Verlag (LNCS), pages 22–23, 2003.

[Lam06a] Leslie Lamport. The +CAL algorithm language. In *Proceedings of the Fifth IEEE International Symposium on Network Computing and Applications*, page 5, Washington, DC, USA, 2006.

[Lam06b] Leslie Lamport. Fast Paxos. *Distributed Computing*, 19(2):79–103, 2006.

[LR89] Nancy A. Lynch and Mark R.Tuttle. An introduction to input/output automata. *CWI Quarterly*, 2(3):219–246, 1989.

[LSP82] Leslie Lamport, Robert Shostak, and Marshall Pease. The Byzantine generals problem. *ACM Transactions on Programming Languages and Systems*, 4(3):382–401, July 1982.

[Lyn96] Nancy A. Lynch. *Distributed Algorithms*. Morgan-Kaufmann, 1996.

[MA06] Jean-Philippe Martin and Lorenzo Alvisi. Fast Byzantine consensus. *IEEE Transactions on Dependable and Secure Computing*, 3(3):202–215, 2006.

[MAD02a] Jean-Philippe Martin, Lorenzo Alvisi, and Michael Dahlin. Minimal Byzantine storage. In *Proceedings of the 16th International Conference on Distributed Computing*, pages 311–325, October 2002.

[MAD02b] Jean-Phillipe Martin, Lorenzo Alvisi, and Michael Dahlin. Small Byzantine quorum systems. In *Proceedings of the International Conference on Dependable Systems and Networks*, pages 374–383, June 2002.

[Mat89] Friedemann Mattern. Virtual time and global states of distributed systems. In *Proceedings of the International Workshop on Parallel and Distributed Algorithms*, pages 215–226, 1989.

[MR98] Dahlia Malkhi and Michael Reiter. Byzantine quorum systems. *Distributed Computing*, 11(4):203–213, 1998.

[MS02] David Mazières and Dennis Shasha. Building secure file systems out of Byzantine storage. In *Proceedings of the 21st annual symposium on Principles of distributed computing*, pages 108–117, New York, NY, USA, 2002. ACM.

[MVO96] Alfred J. Menezes, Scott A. Vanstone, and Paul C. Van Oorschot. *Handbook of Applied Cryptography*. CRC Press, Inc., Boca Raton, FL, USA, 1996.

[NW94] Moni Naor and Avishai Wool. The load, capacity and availability of quorum systems. In *Proceedings of the 35th IEEE Symposium on Foundations of Computer Science*, pages 214–225, 1994.

[Ped01] Fernando Pedone. Boosting system performance with optimistic distributed protocols. *Computer*, 34(12):80–86, 2001.

[PS98] Fernando Pedone and André Schiper. Optimistic atomic broadcast. In *Proceedings of the 12th International Symposium on Distributed Computing*, pages 318–332, 1998.

[PSL80] Marshall Pease, Robert Shostak, and Leslie Lamport. Reaching agreements in the presence of faults. *Journal of the ACM*, 27(2):228–234, April 1980.

[Rab83] Michael O. Rabin. Randomized Byzantine generals. In *Proceedings of the 24th Annual Symposium on Foundations of Computer Science*, pages 403–409, Washington, DC, USA, 1983. IEEE Computer Society.

[RC05] HariGovind V. Ramasamy and Christian Cachin. Parsimonious asynchronous Byzantine-fault-tolerant atomic broadcast. In *Proceedings of the 9th International Conference on Principles of Distributed Systems*, pages 88–102, December 2005.

[RFGN01] Erik Riedel, Christos Faloutsos, Garth A. Gibson, and David Nagle. Active disks for large-scale data processing. *Computer*, 34(6):68–74, 2001.

[Riv85] Ronald L. Rivest. The md5 message-digest algorithm. *Internet RFC-1321*, 1985.

[RS96] Michel Raynal and Mukesh Singhal. Logical time: Capturing causality in distributed systems. *Computer*, 29(2):49–56, 1996.

[RSA78] Ronald L. Rivest, Adi Shamir, and Leonard M. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2):120–126, 1978.

[Sch90] Fred B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys*, 22(4):299–319, December 1990.

[Sch97] André Schiper. Early consensus in an asynchronous system with a weak failure detector. *Distributed Computing*, 10(3):149–157, 1997.

[SDM+08] Atul Singh, Tathagata Das, Petros Maniatis, Peter Druschel, and Timothy Roscoe. BFT protocols under fire. In *Proceedings of the 5th Symposium on Networked Systems Design and Implementation*, 2008.

[SFV+04] Yasushi Saito, Svend Frolund, Alistair Veitch, Arif Merchant, and Susan Spence. Fab: building distributed enterprise disk arrays from commodity components. *SIGOPS Oper. Syst. Rev.*, 38(5):48–58, 2004.

[Tho79] Robert H. Thomas. A majority consensus approach to concurrency control for multiple copy databases. *ACM Trans. Database Syst.*, 4(2):180–209, 1979.

[TP88] Philip Thambidurai and You-Keun Park. Interactive consistency with multiple failure modes. In *Proceedings of the seventh symposium on Reliable distributed systems*, pages 93–100. IEEE Computer Society Press, 1988.

[vRS04] Robbert van Renesse and Fred B. Schneider. Chain replication for supporting high throughput and availability. In *Proceedings of the 6th conference on Symposium on Operating Systems Design & Implementation*, pages 91–104, Berkeley, CA, USA, 2004. USENIX Association.

[WB84] Gene T.J. Wuu and Arthur J. Bernstein. Efficient solutions to the replicated log and dictionary problems. In *Proceedings of the 3rd annual ACM symposium on Principles of distributed computing*, pages 233–242, New York, NY, USA, 1984. ACM.

[YMV+03] Jian Yin, Jean-Philippe Martin, Arun Venkataramani, Lorenzo Alvisi, and Mike Dahlin. Separating agreement from execution for Byzantine fault tolerant services. In *Proceedings of the 19th ACM symposium on Operating systems principles*, pages 253–267, 2003.

[Zie05] Piotr Zielinski. *Minimizing latency of agreement protocols*. PhD thesis, Computer Laboratory, University of Cambridge, Cambridge, UK, September 2005.

[Zie06] Piotr Zielinski. Optimistically terminating consensus: All asynchronous consensus protocols in one framework. In *Proceedings of the Proceedings of The Fifth International Symposium on Parallel and Distributed Computing*, pages 24–33, Washington, DC, USA, 2006. IEEE Computer Society.

List of Figures

List of Tables

Curriculum Vitæ

Marko Vukolić was born in Belgrade, Serbia on December 6th, 1977. He completed elementary school in 1992 and the Mathematical High School in Belgrade in 1996.

In the same year, he started his studies at the School of Electrical Engineering, University of Belgrade. He obtained his graduated engineer (dipl.ing.) degree in Telecommunications in 2001, as the best student in class in the Department of Telecommunications.

In 2002, he continued his graduate studies at the Doctoral School of the Swiss Federal Institute of Technology in Lausanne, Switzerland (EPFL) in the School of Computer and Communication Sciences (IC).

After graduating from EPFL IC Doctoral School in 2003, he started his doctoral studies under the supervision of Prof. Rachid Guerraoui at EPFL IC in the Distributed Programming Laboratory. In 2007 he spent three months as an intern in the IBM Zurich Research Laboratory, Rüschlikon, Switzerland, as a part of his doctoral studies.

Since June 2008, he conducts his research as a full-time employee (post-doc) in the IBM Zurich Research Laboratory, in the Storage Systems group.