Abstractions for Solving Consensus and Related Problems with Byzantine Faults

THÈSE N^{O} 5975 (2013)

PRÉSENTÉE LE 14 FÉVRIER 2014

À LA FACULTÉ INFORMATIQUE ET COMMUNICATIONS
LABORATOIRE DE SYSTÈMES RÉPARTIS
PROGRAMME DOCTORAL EN INFORMATIQUE, COMMUNICATIONS ET INFORMATION

ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

POUR L'OBTENTION DU GRADE DE DOCTEUR ÈS SCIENCES

PAR

Žarko MILOŠEVIĆ

acceptée sur proposition du jury:

Prof. W. Zwaenepoel, président du jury
Prof. A. Schiper, directeur de thèse
Prof. A. Ailamaki, rapporteur
Prof. R. Rodrigues, rapporteur
Prof. R. van Renesse, rapporteur

ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Suisse
2013

To my family

Abstract

We become increasingly dependent on online services; therefore, their availability and correct behavior become increasingly important. Software replication is a popular technique for ensuring that computer systems continue to provide a correct service even when some of their components fail. By replicating a service on multiple servers, clients are guaranteed that even if some replica fails, the service is still available.

At the core of software replication is the consensus problem, where a set of processes has to agree on a single value. A large number of consensus algorithms for different system models have been proposed. The most general system models (for which consensus is solvable) do not make strong assumptions on the synchrony (allow period of asynchrony) and assume that a subset of processes can fail completely arbitrarily (Byzantine faults). However, solving consensus in the presence of arbitrary faults and asynchrony is hard and demands sophisticated algorithms. Most of the existing consensus algorithms that deal with arbitrary faults are monolithic and developed from scratch, or by modifying existing algorithms in a non-modular manner. As a consequence, these algorithms are rather complex and hard to understand. We impute this complexity to the lack of adequate abstractions.

The motivation of this thesis is suggesting *abstractions* that simplify the understanding of existing consensus algorithms with arbitrary faults and allow modular design of novel algorithms. The thesis also aims to clarify relations between consensus and the total-order broadcast problem in the presence of arbitrary faults.

In the context of the consensus problem with arbitrary process faults, the literature distinguishes (1) *authenticated Byzantine* faults, where messages can be signed by the sending process, and (2) *Byzantine* faults, where there is no mechanism for signatures. Consensus protocols that assume *Byzantine* faults (without authentication) are harder to develop and prove correct than algorithms that consider *authenticated Byzantine* faults, even when they are based on the same idea. We propose an abstraction called *weak interactive consistency* (or *WIC*), that allows us to design consensus algorithms that can be instantiated into algorithms for authenticated Byzantine faults (signed messages) and algorithms for Byzantine faults. In other words, WIC unifies Byzantine consensus algorithms with and without signatures. This is illustrated on two seminal Byzantine consensus algorithms: the Castro-Liskov PBFT algorithm (no signatures) and the Martin-Alvisi FaB Paxos algorithms (signatures). WIC allows a very concise expression of these two algorithms. Furthermore, WIC turns out to be fundamental abstraction for solving consensus in the transmission fault model. The transmission fault model captures faults without blaming a specific component for the fault, and it is well-adapted to *dynamic* and *transient* faults. Using WIC we designed a consensus algorithm that overcomes limitations of all existing solutions to consensus in this model, which assume the synchronous system model, or require strong conditions for termination that exclude the case where all messages of a process can be corrupted.

Then we go one step further in unifying consensus algorithms by proposing a *generic consensus algorithm* that highlights, through well chosen parameters, the core mechanisms of a number of well-known consensus algorithms including Paxos, OneThirdRule, PBFT and FaB Paxos. Interestingly, the generic algorithm allows us to identify a new Byzantine consensus algorithm that requires $n > 4b$, in-between the requirement $n > 5b$ of FaB Paxos and $n > 3b$ of PBFT (b is the maximum number of Byzantine processes).

Afterwards, we study the relation between consensus and total-order broadcast in the presence of Byzantine faults. Total-order broadcast is defined for a set of processes, where each process can broadcast messages, with the guarantee that all processes in this set see the same sequence of messages. Among the several definitions of Byzantine consensus that differ only by their validity property, we identify those equivalent to total-order broadcast. We also give the first deterministic total-order broadcast reduction to consensus with constant time complexity with respect to consensus.

Finally, we consider state-machine replication (SMR) with Byzantine faults. State-machine replication is a general approach for replicating services that can be modeled as a state machine. The key idea of this approach is to guarantee that all replicas start in the same state and then apply requests from clients in the same order, thereby guaranteeing that the replica states do not diverge. Recent studies has shown that most BFT-SMR algorithms do not actually perform well under performance attacks by Byzantine processes. We propose a new BFT-SMR algorithm, called *BFT-Mencius*, that guarantees, assuming a partially synchronous system model, that the latency of updates of correct processes is eventually upper-bounded, even under performance attacks by Byzantine processes. BFT-Mencius is a modular, signature-free algorithm based on a new communication primitive called *Abortable Timely Announced Broadcast* (ATAB). We evaluate the performance of BFT-Mencius in cluster settings, and show that it performs comparably to the state-of-the-art algorithms such as PBFT and Spinning in fault-free configurations and outperforms these algorithms under performance attacks by Byzantine processes.

Keywords: Distributed Algorithms, Consensus, Unification, Authentication, Byzantine Fault Tolerance, Total-Order Broadcast, Reduction, State Machine Replication, Performance Attack, Bounded Delay.

Résumé

Nous sommes de plus en plus dépendants des services en ligne, et donc, de leur bon fonctionnement et de leur disponibilité. La réplication logicielle est une solution couramment utilisée pour assurer qu'un système informatique continue à fournir un service correct en dépit de la défaillance de certains de ses composants. En répliquant un service sur plusieurs serveurs, les clients ont l'assurance que le service reste disponible même si certaines répliques subissent une défaillance.

Le problème de consensus, où un ensemble de processus doivent se mettre d'accord sur une valeur commune, est au cœur de la réplication logicielle. Un grand nombre d'algorithmes de consensus ont été proposés avec différentes hypothèses sur les caractéristiques du système considéré. Le modèle le plus général (dans lequel le problème de consensus peut être résolu) ne fait pas de suppositions fortes sur le synchronisme du système (le système peut se comporter de manière asynchrone temporairement), et suppose qu'un sous-ensemble des processus peut subir des défaillances arbitraires (fautes byzantines). Cependant, résoudre le problème de consensus en présence d'asynchronisme et de fautes byzantines est difficile et repose sur des algorithmes sophistiqués. La plupart des algorithmes de consensus qui peuvent traiter des fautes byzantines sont monolithiques. Ils ont été développés soit en partant d'une feuille blanche, soit en modifiant des algorithmes existants d'une manière non-modulaire. Ainsi, ces algorithmes sont complexes et difficiles à comprendre. Nous imputons cette complexité à l'absence d'abstractions adéquates.

L'objectif de cette thèse est de proposer des abstractions que simplifient la compréhension des algorithmes de consensus existants traitant les fautes byzantines, et de permettre la conception de nouveaux algorithmes de manière modulaire. Cette thèse a aussi pour objectif de clarifier la relation entre les problèmes de consensus et de diffusion atomique en présence de fautes byzantines.

Dans le contexte du problème de consensus avec des fautes arbitraires de processus, la littérature distingue (1) les fautes byzantines *authentifiées*, où les messages peuvent être signés par le processus émetteur, et (2) les fautes byzantines, où il n'y a pas de mécanisme de signature. Les protocoles de consensus qui supposent des fautes byzantines (sans authentification) sont plus complexes à développer et à prouver que les algorithmes qui considèrent des fautes byzantines authentifiées, même quand ils sont fondés sur la même idée. Nous proposons une abstraction nommée *Consistance Interactive Faible* (Weak Interactive Consistency, WIC).

Cette abstraction nous permet de concevoir des algorithmes de consensus qui peuvent être instanciés pour des fautes byzantines authentifiées (avec messages signés) ou pour des fautes byzantines. Autrement dit, WIC unifie les algorithmes de consensus pour fautes byzantines avec et sans signatures. Nous illustrons ceci avec deux algorithmes de consensus fondateurs : l'algorithme PBFT de Castro et Liskov (sans signatures) et l'algorithme FaB Paxos de Martin et Alvisi (avec signatures). WIC permet de décrire ces deux algorithmes de manière très concise. De plus, WIC est une abstraction fondamentale pour résoudre le problème de consensus dans le modèle de faute de transmissions (transmission fault model). Le modèle de fautes de transmissions exprime les fautes sans mettre en cause de composants spécifiques. Il est adapté aux fautes dynamiques et transitoires. Nous avons conçu un algorithme de consensus utilisant WIC qui s'affranchit des limitations de tous les algorithmes de consensus proposés dans ce modèle, c'est-à-dire qu'il ne suppose pas un système synchrone, et qu'il peut terminer même si tous les messages d'un processus peuvent être corrompus.

Nous faisons ensuite un pas de plus dans l'unification des algorithmes de consensus. Nous proposons un algorithme de consensus générique qui met en évidence au travers d'un ensemble de paramètres, les mécanismes centraux de plusieurs protocoles de consensus dont Paxos, OneThirdRule, PBFT, et FaB Paxos. Cet algorithme générique nous permet d'identifier un nouvel algorithme de consensus pour fautes byzantines qui nécessite n>4b, quand FaB Paxos nécessite n>5b et PBFT nécessite n>3b (b étant le nombre maximum de processus byzantins).

Dans le chapitre suivant, nous étudions la relation entre consensus et diffusion atomique en présence de fautes byzantines. Dans un ensemble de processus où chaque processus peut diffuser des messages, la diffusion atomique assure que tous les processus de l'ensemble reçoivent la même séquence de messages. Sur l'ensemble des définitions du problème de consensus avec des fautes byzantines qui diffèrent seulement par leur propriété de validité, nous identifions celles qui sont équivalentes au problème de diffusion atomique. Nous donnons aussi la première réduction déterministe de problème de diffusion atomique au problème de consensus avec une complexité en temps constante par rapport au problème de consensus.

Finalement, nous étudions la réplication de machine à états (state machine replication, SMR) dans le contexte des fautes byzantines. La réplication de machines à états est une approche générale pour répliquer des services qui peuvent être modélisés comme une machine à états. Le principe de cette approche est de garantir que toutes les répliques démarrent dans le même état et exécutent les requêtes des clients dans le même ordre, pour assurer que l'état des répliques ne diverge pas. Des études récentes ont montré que la plupart des algorithmes BFT-SMR ont de mauvaises performances en cas d'attaque sur les performances par des processus byzantins. Nous proposons un nouvel algorithme BFT-SMR, appelé BFT-Mencius, qui garantit dans le cas d'un système partiellement synchrone, que la latence sur les mises à jour des processus corrects est éventuellement bornée, même en cas d'attaque sur les performances par des processus byzantins. BFT-Mencius est en algorithme modulaire ne nécessitant pas de signatures. Il est fondé sur une nouvelle primitive de communication appelée *Abordable Timely Announced Broadcast*. Les performances de BFT-Mencius, évaluées

sur une grappe de calcul, sont comparables à celles des algorithmes de l'état de l'art tels que PBFT et Spinning dans les exécutions sans fautes et sont meilleures que celles de ces algorithmes en cas d'attaque sur les performances par des processus byzantins.

Mots clés : Algorithmes distribués, problème de consensus, unification, authentification, fautes byzantines, diffusion atomique, réduction, réplication de machine à états, attaque sur les performances, délais borné.

Acknowledgements

First and foremost, I would like to thank my advisor, Prof. André Schiper, for his constant support and guidance, and, in particular, for always being available for answering my questions and exchanging thoughts. I appreciate the research freedom he offered, always giving me a chance to choose.
I was honored to have Prof. Anastasia Ailamaki, Prof. Rodrigo Rodrigues, Prof Robbert van Renesse and Prof. Christoph Koch as the members of the thesis jury, and Prof. Willy Zwaenepoel for chairing the jury. Thank you for carefully evaluating my thesis and providing the valuable feedback.
I am also grateful to my colleagues at the Distributed Systems Laboratory (LSR). I am particularly appreciative to four colleagues: Martin Hutle, Olivier Rütti, Nuno Santos and Martin Biely. To Martin Hutle for spending precious time with me while I was making my first steps in the scientific world, and later for co-authoring most of the work presented in this thesis. To Olivier for initiating an interesting work that led to Chapter 5. To Nuno for advices and help with the implementations and experiments. To Martin Biely for help at the end of my PhD journey, most particular, for implementing Distal, a framework that was invaluable tool for the practical contribution of the thesis. My gratitude also goes to all other members of the LSR, for interesting discussions, providing precious feedback on the drafts of the papers and finally for creating a great atmosphere at LSR. In particular I would like to thank Fatemeh Borran, Omid Shahmirzadi, Darko Petrović, Thomas Ropars, Cendrine Favez and Sergio Mena. Special thanks goes to France Faille, for always caring.
I am deeply grateful to my parents, Radica and Mile, my sister Bojana and my brother Nenad for their unconditional love and support throughout my whole life. At the point I am finishing my formal education, I would like to thank my parents for teaching me the true values in life, for always believing in me and pushing me forward.
Finally, and above all, the warmest gratitude goes to my loving wife Ana for her infinite love, patience, understanding and support, and to our daughters Mila and Dana, for bringing the immense joy to my life and making this period of my life so special and unforgettable.

Contents

List of Figures

List of Tables

1 Introduction

1.1 Thesis Context

We leave in an era in which computing is ubiquitous, where most people use online services everyday. As we become highly dependent on these services, their availability and correct functioning are extremely important. The nature of some services such as banking applications or applications that store medical records, makes outage or malfunction expensive or unacceptable. Therefore, it is not surprising that a lot of effort is put to ensure high availability and reliability of these services. However, because of the importance that online services have in modern society, we have similar expectation for our favorite email service, search service, social application, even our favorite online game.

However computers fail. The causes of faults are various, including faulty hardware, bit flips caused by ionizing radiation, overheating, human error and software bugs. Finally, computers malfunction because of the presence of malicious software such as viruses, worms, etc, or because they are under attack by hackers. Therefore, the important research question is how to build reliable systems from components that are inherently unreliable.

There are several approaches to this problem. For instance, the likelihood of failures can be decreased by designing better hardware and software (*fault avoidance*). However, this approach has high costs and can only decrease the likelihood of faults, it can never eliminate them completely. Another approach, which is the topic of this thesis, is called fault-tolerance.

A system is said to be *fault-tolerant* if it will continue operating in spite of the failure of some of its components. A popular way of achieving fault tolerance is software replication: By replicating a service on multiple servers, clients are guaranteed that even if some replica fails, the service is still available. Software replication is widely used because of its generality (can be applied to most services) and its low cost (use of the off-the-shelf hardware).

State Machine Replication State machine replication (SMR) is a general approach for replicating services that can be modeled as a deterministic state machine [Lam78, Sch90]. The key idea of this approach is to guarantee that all replicas start in the same state and then apply requests from clients in the same order, thereby guaranteeing that the replicas' states will not diverge.

The problem of agreeing on a order for a sequence of requests is an example of the *total-order broadcast* (or atomic broadcast) problem. Total-order broadcast is defined for a set of processes, where each process can broadcast messages, with the guarantee that all processes in this set see the same sequence of messages. At the heart of atomic broadcast, and therefore of state machine replication, is the *consensus* problem, where a set of processes has to agree on a single value. The two problems are in fact closely related, meaning that a solution to one can be used to solve the other.

Consensus Consensus is probably the most fundamental problem in fault-tolerant distributed computing. As such, it has been heavily studied for the last 30 years, both from a theoretical and practical perspective. This explains the numerous consensus algorithms that have been published, with different features and for different fault models.

The difficulty of solving consensus depends both on the failure assumptions (type and number of failures) and on the degree of *synchrony* of the system. The two main models considered in distributed computing are: the *synchronous* system model and the *asynchronous* system model. In a synchronous system model there is (1) a known bound Δ on the transmission delay of messages, and (2) a known bound Φ on the relative speed of processes. On the other hand, in an asynchronous system there is no bound on the transmission delay of messages and no bound on the relative speed of processes. This typically models a system with unpredictable load on the network and on the CPU.

Most research on consensus algorithms is considering *component fault models*, where faults are attached to a component that is either a process or a link. With respect to process/link faults, consensus can be considered with different fault assumptions. On the one end of the spectrum, processes/links can commit so called *benign* faults (processes fail only by crashing and links only loose messages); on the other end, faulty processes/links can exhibit an *arbitrary* behavior. Furthermore, in the context of a component fault model, faults are mainly *permanent* (as opposed to *transient* faults): if a process or link commits a fault, the process/link is considered to be faulty during whole execution. It follows that not all components can be faulty (at most f out of n per run), which is referred to as *static* faults (as opposed to dynamic faults that can affect any component).

In the context of arbitrary process faults, the literature distinguishes *authenticated Byzantine* faults, where messages can be signed by the sending process (with the assumption that the signature cannot be forged by any other process), and *Byzantine* faults, where there is no mechanism for signatures (but the receiver of a message knows the identity of the sender)[1]. Consensus protocols that assume *Byzantine* faults (without authentication) are harder to develop and prove correct [ST87].

An alternative approach to the component fault models is the *transmission fault model* that captures faults without blaming a specific component for the fault [SW89]. The transmission fault model is well-adapted to *dynamic* and *transient* faults.

Unfortunately, consensus is impossible to solve with a deterministic algorithm[2] in an asynchronous system even if only single process may crash [FLP85]. Although it is possible to solve consensus in the synchronous system model with Byzantine processes, it is not considered as a good idea from a practical point of view. The reason is that the synchronous system model requires to be pessimistic when defining the bounds on message transmission delays (and process relative speeds). Pessimistic bounds have a negative impact on the performance of consensus algorithms. Furthermore, considering synchronous solutions in the context of arbitrary faults might be dangerous: an attacker may simply target the timely delivery of messages in order to compromise the correctness of the protocol.

Therefore, the research community has focused its attention on intermediate models, with weaker assumptions than the synchronous model, but where consensus is still solvable. One such model is the *partially synchronous model* [DLS88], which relaxes the assumptions of the synchronous system by requiring that the timing bounds hold only eventually. It is possible to solve consensus in the partially synchronous system model, and contrary to the synchronous system model, the partially synchronous system model does not require being too pessimistic when defining the bounds on message transmission delays and process relative speeds.

Total-Order Broadcast The relation of consensus and total-order broadcast (called also atomic broadcast), including the reduction of total-order broadcast to consensus, is well understood in the case of benign faults [CT96a]. On the contrary, little is known on the relation between total-order broadcast and the consensus problem in the context of Byzantine faults. One can also observe that there exist several definitions of consensus with Byzantine faults (which differ in the validity property), and it is not clear at all which one should be considered for reduction of total-order broadcast.

Byzantine Fault Tolerance under Attack Byzantine fault-tolerant (BFT) state machine replication algorithms allow computer systems to continue to provide a correct service even when some of their components behave in an arbitrary way, either due to faults or due to a malicious intruder. However, recent studies have shown that most BFT-SMR systems do not actually tolerate Byzantine faults well [CWA$^+$09, ACKL11]. More precisely, it has been shown that a faulty process exhibiting performance failures can delay the ordering of requests, causing a

[1]In [LSP82], the latter is called Byzantine faults with *oral messages*.
[2]A deterministic algorithm is an algorithm that does not use randomization (random number generation).

considerable increase in latency and a great reduction in throughput. Performance failures were defined as Byzantine processes behaving as a "correct but very slow" servers.

This led to the definition of a new performance criterion, called *bounded-delay* [ACKL11]. Bounded-delay requires that, in a (long enough) period of synchrony the latency of updates initiated by correct processes is eventually upper-bounded[3], even in the presence of Byzantine processes. In a similar spirit, Clement et al. [CWA+09] have advocated what they called robust BFT. That is, they propose to shift the focus from algorithms that optimize only best case performance, and to design algorithms that can offer predictable performance under the broadest possible set of circumstances—including when faults occur.

1.2 Thesis Motivation

Dealing with arbitrary faults and asynchrony is hard and requires sophisticated algorithms. Most of the existing algorithms (that deal with arbitrary faults) are monolithic and developed from scratch, or obtained by modifying existing algorithms in a non-modular manner. As a consequence, these algorithms are rather complex and hard to understand. We impute this complexity to the lack of adequate abstractions.

1.3 Thesis Goal

The goal of this thesis is suggesting *abstractions* that improve the understanding of existing fault tolerant algorithms (that deal with arbitrary faults) and allow modular design of novel algorithms. The thesis also aims to clarify relations between fundamental problems, such as consensus and total-order broadcast, in the presence of arbitrary faults.

1.4 Thesis Contribution

The thesis makes the following contributions:

Unifying Byzantine Consensus Algorithms with Weak Interactive Consistency Consensus protocols that assume *Byzantine* faults (without authentication) are harder to develop and prove correct [ST87] than algorithms that consider *authenticated Byzantine* faults, even when they are based on the same idea. We propose an abstraction called *weak interactive consistency* (or *WIC*), that allows us to design a consensus algorithm that can be instantiated into an algorithm for authenticated Byzantine faults (signed messages) and algorithm for Byzantine faults, i.e., it unifies Byzantine consensus algorithms with and without signatures. The power of WIC is illustrated on two seminal Byzantine consensus algorithms: the Castro-Liskov PBFT algorithm [CL02] (no signatures) and the Martin-Alvisi FaB Paxos algorithm [MA06]

[3]The additional assumption is that a system that is not overloaded and where processes have sufficient bandwidth to communicate. Without such assumptions it is hard to provide any guarantees.

(signatures). WIC allows a very concise expression of these two algorithms.

Tolerating Permanent and Transient Faults Transmission faults allow us to reason about permanent and transient faults in a uniform way. However, all existing solutions to consensus in this model are either in the synchronous system model, or require strong conditions for termination, that exclude the case where all messages of a process can be corrupted. We propose a consensus algorithm for the transmission fault model that does not have those limitations. The algorithm considers a system parameterized with α and f. In every round *each process* can receive up to α corrupted messages; eventually rounds are synchronous and the messages sent by at most f processes are corrupted. Before these synchronous rounds, any number of benign faults is tolerated. Depending on the nature and number of permanent and transient transmission faults, we obtain different conditions on n (number of processes) for solving consensus.

Generic Consensus Algorithm for Benign and Byzantine Faults Numerous consensus algorithms have been published, with different features and for different fault models. Understanding these numerous algorithms could be made easier by identifying the core mechanisms on which these algorithms rely. We propose a *generic consensus algorithm* that highlights, through well chosen parameters, the core mechanisms of a number of well-known consensus algorithms including Paxos [Lam98], OneThirdRule [CBS09a], PBFT [CL02] and FaB Paxos [MA06]. Interestingly, the generic algorithm allowed us to identify a new Byzantine consensus algorithm that requires $n > 4b$, in-between the requirement $n > 5b$ of FaB Paxos and $n > 3b$ of PBFT (b is the maximum number of Byzantine processes). The generic consensus algorithm contributes to identify key similarities rather than non fundamental differences between consensus algorithms.

On the Reduction of Total-Order Broadcast to Consensus with Byzantine Faults We investigate the reduction of total-order broadcast to consensus in systems with Byzantine faults. Among the several definitions of Byzantine consensus that differ only by their validity property, we identify those equivalent to total-order broadcast. Finally, we give the first total-order broadcast reduction algorithm to consensus that has constant time complexity with respect to consensus.

Bounded Delay in Byzantine Tolerant State Machine Replication We propose a new state machine replication protocol for the partially synchronous system with Byzantine faults. The algorithm, called *BFT-Mencius*, guarantees that the latency of updates initiated by correct processes is eventually upper-bounded, even in the presence of Byzantine processes. BFT-Mencius is based on a new communication primitive, *Abortable Timely Announced Broadcast* (ATAB), and does not use signatures. We evaluate the performance of BFT-Mencius in the cluster settings, and show that it performs comparably to the state-of-the-art algorithms in fault-free configurations and outperforms them under performance attacks by Byzantine processes.

1.5 Outline

Chapter 2 provides the background material, including definitions and problem statements. Chapter 3 introduces weak interactive consistency abstraction (WIC) and show how it can be used to express concisely two well-known consensus algorithms. Chapter 4 shows the usefulness of WIC abstraction in the context of transmission fault model, where it is the key abstraction for solving consensus in a very weak model that considers both transient and permanent faults. in Chapter 5 we present our generic consensus algorithm that captures the core mechanisms of numerous consensus algorithms. In Chapter 6 we study the relation between total-order broadcast and consensus in systems with Byzantine faults. Chapter 7 presents new BFT state machine replication protocol that performs well under performance attacks by Byzantine processes. The thesis concludes in Chapter 8 that summarizes main results and identifies areas for future research.

2 Preliminaries

The chapter introduces the problems considered in this thesis, and defines the partially synchronous system model and the basic round model.

2.1 Consensus Problem

The consensus problem is defined over a set of processes Π, where each process $p \in \Pi$ starts with a given initial value, and later *decides* on a common value. In this thesis we will consider the consensus problem in the context of arbitrary process faults and in the context of the transmission fault model. The consensus definitions are slightly different in these two cases.

2.1.1 Arbitrary Process faults

In the context of arbitrary process faults, we differentiate *honest* processes that execute algorithms faithfully, from *Byzantine* processes [LSP82], that exhibit arbitrary behavior. Honest processes can be *correct* or *faulty*. An honest process is faulty if it eventually crashes, and is correct otherwise. The set of honest processes is denoted by $\mathscr{H}$ and the set of correct processes by $\mathscr{C}$. The consensus problem is formally specified by the following properties:

- *Agreement:* No two honest processes decide differently.
- *Termination:* All correct processes eventually decide.
- *Weak Validity:* If all processes are honest and if an honest process decides v, then v is the initial value of some process.
- *Strong Unanimity:* If all honest processes have the same initial value v and an honest process decides, then it decides v.

There are other definitions, which instead of Weak Validity and Strong Unanimity, consider a different validity property. They will be discussed in Chapter 6.

2.1.2 Transmission Faults

In the context of the transmission fault model, we consider the following specification of the consensus problem:

- *Agreement:* No two processes decide differently.
- *Termination:* All processes eventually decide.
- *Integrity:* If all processes have the same initial value this is the only possible decision value.

As in the transmission fault model there are no faulty processes, all processes must decide the initial value in the Integrity clause, and all processes must make a decision by the Termination clause.

2.2 Total-order Broadcast

Total-Order Broadcast is defined in terms of two primitives, to-broadcast and to-deliver. A process p that wishes to broadcast a message m taken from the set of messages $\mathcal{M}$ invokes to-broadcast(m). A message m is delivered by process q by executing to-deliver(m). We assume that the sender of a message can be determined from the message (denoted by $sender(m)$) and that all messages are unique. Both can be easily achieved by adding process identifiers and sequence numbers to messages. Total-order broadcast fulfills the following properties [HT94]:

- *TO-Validity*: If a correct process p invokes to-broadcast(m), then p eventually executes to-deliver(m).
- *TO-Agreement*: If a correct process p executes to-deliver(m), then every correct process q eventually executes to-deliver(m).
- *TO-Integrity*: For any message m, every correct process p executes to-deliver(m) at most once. Moreover, if $sender(m)$ is correct, then it previously invoked to-broadcast(m).
- *TO-Order*: If correct processes p and q execute to-deliver(m) and to-deliver(m'), then p delivers m before m' if and only if q delivers m before m'.

2.3 State Machine Replication

State machine replication (SMR) is a general approach for replicating services that can be modeled as a deterministic state machine [Lam78, Sch90]. The key idea of this approach is to guarantee that all replicas start in the same state and then apply requests from clients in the same order, thereby guaranteeing that the replicas' states will not diverge. Following Schneider [Sch90], we note that the following is key for implementing a replicated state machine tolerant to (Byzantine) faults:

- *Replica Coordination.* All [non-faulty] replicas receive and process the same sequence of requests.

Moreover, as Schneider also notes this property can be decomposed into two parts, *Agreement* and *Order*: Agreement requires all (non-faulty) replicas to receive all requests, and Order requires that the order of received requests is the same at all replicas.

There is an additional requirement that needs to be ensured by Byzantine tolerant state machine replication: only requests proposed by clients are executed. This requirement is trivially ensured by using cryptographic signatures to sign client requests. Request authentication can also be achieved using message authentication codes (MAC) [CL02, AABC08a].

2.4 Partially Synchronous System Model

The partially synchronous system model [DLS88] lies between a synchronous system and an asynchronous system. It relaxes the assumptions of the synchronous system, by requiring that the timing bounds hold only eventually. We can distinguish partial synchrony for processes and partial synchrony for communication. There are two variants of the partially synchronous model, depending on the assumptions on the bound Δ on message transmission delay and Φ on relative process speed:

Unknown bounds The bounds Δ and Φ hold from the beginning but are unknown (*i.e.*, Δ and Φ depend on the run).

Known bounds The bounds Δ and Φ are known but hold only after an unknown time called *Global Stabilization Time (GST)* (*i.e.*, GST varies from run to run). It is also assumed that channels may lose messages before GST but the channels among correct processes are reliable after GST.

2.5 The Basic Round Model

The basic round model is a computational model that was introduced in [DLS88] as a convenient abstraction on top of the partially synchronous system model. Using this abstraction, rather than the raw system model, improves the clarity of the algorithms and simplifies the proofs. In the basic round model, distributed algorithms are expressed as a sequence of rounds. Each round r consists of a sending step, a receive step, and a state transition step:

1. In the sending step of round r, each process p sends a message to each process according to a "sending" function S_p^r. [1]
2. In the receive step of round r, each process q receives a subset of all messages sent in round r (it can be the empty set); messages received by process p in round r are denoted by $\vec{\mu}_p^r$ ($\vec{\mu}_p^r[q]$ is the message received from q). The receive step is implicit, i.e., it does not appear in the algorithm. Note that this implies that a message sent in round r can only be received in round r (rounds are *closed*).
3. In the state transition step of round r (that takes place at the end of round r), each process p computes a new state according to a "transition" function T_p^r that takes as input the vector of messages it received at round r and its current state.

[DLS88] shows how to implement rounds that are eventually synchronous in the presence of Byzantine processes, i.e., eventually, for all rounds r, all messages sent in a round r by a correct process are received in the round r by all correct processes.

The state[2] of process p in round r is denoted by s_p^r; the message sent by an honest process is denoted by $S_p^r(s_p^r)$. We will refer to some field *fld* of a message m using notation *m.fld*.

[1]Without loss of generality, the same message is sent to all.
[2] Note that referring to the state of a Byzantine process does not make sense.

3 Unifying Byzantine Consensus Algorithms with WIC

As explained in Chapter 1, in the context of arbitrary process faults, the literature distinguishes *authenticated Byzantine* faults, where messages can be signed by the sending process (with the assumption that the signature cannot be forged by any other process), and *Byzantine* faults, where there is no mechanism for signatures (but the receiver of a message knows the identity of the sender). In this chapter we introduce an abstraction called *weak interactive consistency* (*WIC*) that unifies consensus algorithms with and without signed messages. WIC can be implemented with and without signatures.

The power of WIC is illustrated on two seminal Byzantine consensus algorithms: the Castro-Liskov PBFT algorithm [CL02] (no signatures) and the Martin-Alvisi FaB Paxos algorithm [MA06] (signatures). WIC allows a very concise expression of these two algorithms.

Publication: Zarko Milosevic, Martin Hutle and André Schiper. Unifying Byzantine Consensus Algorithms with Weak Interactive Consistency. In *13th International Conference On Principle Of Distributed Systems (OPODIS 2009)*, Nimes, France, December 15 - December 18, 2009.

3.1 Introduction

Consensus protocols that assume *Byzantine* faults (without authentication) are harder to develop and prove correct [ST87]. As a consequence, they tend to be more complicated and harder to understand than the protocols that assume *authenticated Byzantine* faults, even when they are based on the same idea. The existence of these two fault models raises the following question: is it possible to design a consensus algorithm such that it can be instantiated into an algorithm for authenticated Byzantine faults and an algorithm for Byzantine faults?

This question has been addressed by Srikanth and Toueg in [ST87] for the Byzantine agreement problem,[1] by defining the *authenticated broadcast* primitive. Authenticated broadcast is a communication primitive that provides additional guarantees compared to, *e.g.*, a normal (unreliable) broadcast. Srikanth and Toueg solve Byzantine agreement using authenticated broadcast, and show that authenticated broadcast can be implemented with and without signatures.

However, authenticated broadcast does not encapsulate all the possible uses of signed messages when solving consensus. One typical example is the Fast Byzantine Paxos algorithm [MA06], which relies on signed messages whenever the coordinator changes.

Complementing the approach of [ST87], we define an abstraction different from authenticated broadcast that we call *weak interactive consistency.* [2] Interactive consistency is defined in [PSL80] as a problem where correct processes must agree on a vector such that the ith element of this vector is the initial value of the ith process if this process is correct. Our abstraction is a weaker variant of interactive consistency, hence the name “weak” interactive consistency. Similarly to authenticated broadcast, weak interactive consistency can be implemented with and without signatures. We illustrate the power of weak interactive consistency by reexamining two seminal Byzantine consensus algorithms: the Castro-Liskov PBFT algorithm, which does not use signatures [CL02], and the Martin-Alvisi FaB Paxos algorithm, which relies on signatures [MA06]. We show how to express these two algorithms using the weak interactive consistency abstraction, and call these two algorithms CL (for Castro-Liskov), resp. MA (for Martin-Alvisi).

Both CL and MA are very concise algorithms. Moreover, replacing in CL weak interactive consistency with a signature-free implementation basically leads to the original signature-free PBFT algorithm, while replacing in MA weak interactive consistency with a signature-based implementation basically leads to the original signature-based FaB Paxos algorithm. In the latter case, the algorithm obtained is almost identical to the original algorithm; in the former case, the differences are slightly more important. In addition, using MA with a signature-free implementation of WIC allows us to derive a signature-free variant of FaB Paxos.

Roadmap The rest of the chapter is structured as follows. Weak interactive consistency is informally introduced in Section 3.2, and then formally defined in Section 3.3. In Section 3.4 we show that weak interactive consistency can be implemented with and without signatures. Section 3.5 describes the MA consensus algorithm (FaB Paxos expressed using weak interactive consistency) and the CL consensus algorithm (PBFT expressed using weak interactive consistency). Section 3.6 discusses related work, and Section 3.7 concludes the chapter.

[1] In this problem, a transmitter sends a message to a set of processes, all processes eventually deliver a single message, and (i) all correct processes agree on the same message, (ii) if the transmitter is correct, then all correct processes agree on the message of the transmitter.

[2] In [Lam83], Lamport defines "Weak Interactive Consistency Problem", as a general problem of reaching agreement. In [DGG00a], Doudou et al. define an abstraction called "Weak Interactive Consistency" (or WIConsistency), with a different definition than ours. We explain the difference in Section 3.6.

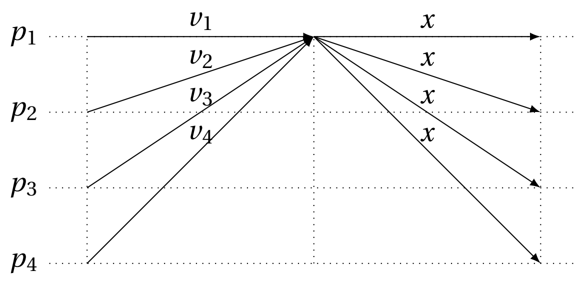


Figure 3.1: Coordinator change: $\boldsymbol{p_1}$ is the new coordinator.

3.2 Weak interactive consistency: an informal introduction

3.2.1 On the use of signatures

We start by addressing the following question: where are signatures used in coordinator based consensus algorithms? Signatures are typically used each time the coordinator changes, as done for example in the FaB Paxos algorithm [MA06]. The corresponding communication pattern is illustrated in Figure 3.1, and addresses the following issue. Assume that the previous coordinator has brought the system into a configuration where a process already decided v; in this case, in order to ensure safety (*i.e.*, agreement) the new coordinator can only propose v. This is done as follows. First every process sends its current estimate to the new coordinator (v_i sent by p_i to p_1 in Figure 3.1). Second, if the coordinator p_1 receives a quorum of messages, then p_1 applies a function f that returns some value x. The quorum ensures that if a process has already decided v, then f returns v. Finally, the value returned by f is then sent to all (x sent by p_1 in Figure 3.1).

This solution does not work with a Byzantine coordinator: the value sent by the coordinator p_1 might not be the value returned by f. Safety can here be ensured using signatures: Processes p_i sign the estimates v_i sent to the coordinator p_1, and p_1 sends x together with the quorum of signed estimates it received. This allows a correct process p_i, receiving x from p_1, to verify whether x is consistent with the function f. If not, then p_i ignores x.

Are signatures mandatory here? We investigate this question, first addressing safety and then liveness.

3.2.2 Safe updates requires neither signatures nor a coordinator

As said, safety means that if a process has decided v, and thus a quorum of processes had v as their estimate at the beginning of the two rounds of Figure 3.1, then each process can only update its estimate to v. This property can be ensured without signatures and without coordinator: each process p_i simply sends v_i to all, and each process p_i behaves like the coordinator: if p_i receives a quorum of messages, it updates its estimate with the value returned by f.

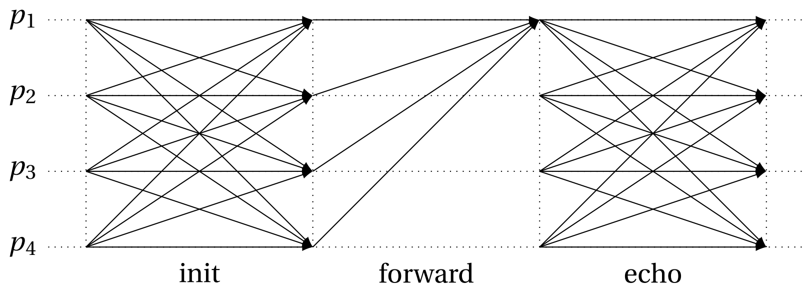


Figure 3.2: Three rounds to get rid of signatures when changing coordinator to $\boldsymbol{p_1}$.

This shows that updating the estimate maintaining safety does not require a coordinator. However, as we show in the next section, a coordinator is reintroduced for liveness.

3.2.3 Coordinator for liveness

The coordinator in Figure 3.1 has two roles: (i) it ensures safety (using signatures), and (ii) it tries to bring the system into a univalent configuration (if not yet so), in order to ensure liveness (*i.e.*, termination) of the consensus algorithm. A configuration typically becomes v-valent as soon as a quorum of correct processes update their estimate to v. This is ensured by a correct coordinator, if its message is received by a quorum of correct processes. Ensuring that a quorum of correct processes update their estimate to the same value v can also be implemented without signatures with an all-to-all communication schema, *if all correct processes receive the same set (of quorum size) of values.* Indeed, if two correct processes apply f to the same set of values, they update their estimate to the same value.

However, ensuring that all correct processes receive the same set of messages is problematic in the presence of Byzantine processes: (i) a Byzantine process can send v to some correct process p_i and v' to some other correct process p_j, and (ii) a Byzantine process can send v to some correct process p_i and nothing to some other correct process p_j.

These problems can be addressed using two all-to-all rounds and one all-to-coordinator rounds, as shown in Figure 3.2 (to be compared with the "init" round followed by the "echo" round of authenticated broadcast [ST87], see Figure 3.3).

These three rounds can be seen as one all-to-all *macro-round* that "always" satisfies integrity and "eventually" satisfies consistency:

Integrity. If a correct process p receives v from a correct process q in super-round r, then v was sent by q in super-round r.

Consistency. (i) If a correct process p_i sends v in super-round r, then every correct process receives v from p_i in super-round r, and (ii) all correct processes receive the same set of messages in super-round r.

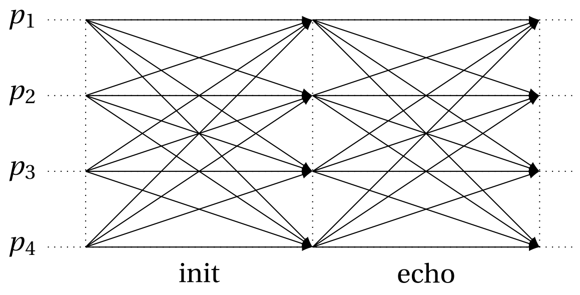


Figure 3.3: Two rounds to get rid of signatures for authenticated broadcast [ST87].

As noted in Section 3.2.2, integrity ensures safety. As noted at the beginning of this section, eventual consistency allows us to eventually bring the system into a univalent configuration, thus ensuring liveness.

In the scheme of Figure 3.2 we combine the concept of a coordinator as depicted in Figure 3.1 with the authentication scheme of [ST87] depicted in Figure 3.3.

This schema ensures that in synchronous rounds (which eventually exist in a partially synchronous model, see Section 2.4), messages received by a correct coordinator in the "forward" round (see Figure 3.2), are received by all correct processes in the "echo" round (see Figure 3.2). [3] Note that without having the coordinator, the authentication scheme of [ST87] is not able to provide a super-round such that all processes receive the same set of messages at the end of this super-round, since a Byzantine process can always prevent this from happening.

We call the problem of always ensuring integrity and eventually consistency the *weak interactive consistency* problem, or simply *WIC*.[4] We show below that WIC is a unifying concept for Byzantine consensus algorithms. WIC can be implemented with signatures in two rounds (Figure 3.1), or without signatures in three rounds (Figure 3.2), as shown in Section 3.4.

3.3 Definition of WIC

Assuming synchronous rounds is a strong assumption that we do not want to consider here. On the other side, an asynchronous system is not strong enough: WIC is not implementable in such a system. We consider a third option, *i.e.*, the partially synchronous system model (see Section 2.4). More precisely we consider an abstraction on top of the partially synchronous system model, namely the basic round model (see Section 2.5). Among the n processes in our system, we assume that at most b are Byzantine. We do not make any assumption about the behavior of Byzantine processes.

[3]The authentication scheme of [ST87] ensures that, during the synchronous rounds, if a message is received by a correct process in some round r', then it is received by all correct processes the latest in round $r'+1$.

[4]The relation with "interactive consistency" [PSL80], is explained in Section 3.1.

In every round of the basic round model, if an honest process sends v, then every honest process receives v or nothing. This can formally be expressed by the following predicate ($\perp$ represents no message reception):

$$\mathscr{P}_{int}(r) \equiv \forall p,q \in \mathscr{H} : (\vec{\mu}_p^r[q] = S_q^r(s_q^r)) \lor (\vec{\mu}_p^r[q] = \perp)$$

The basic round model ensures that rounds are eventually synchronous, i.e., eventually, for all rounds r, messages sent in round r by a correct process are received in round r by all correct processes. The fact that a round r is synchronous is formally expressed by the following predicate:

$$\mathscr{P}_{good}(r) \equiv \forall p,q \in \mathscr{C} : \vec{\mu}_p^r[q] = S_q^r(s_q^r)$$

We have informally defined WIC by an integrity property and by a consistency property that must hold "eventually". The integrity property is expressed by the predicate $\mathscr{P}_{int}$. "Eventual" consistency formally means that there exists a synchronous round r in which consistency holds:

$$\mathscr{P}_{cons}(r) \equiv \mathscr{P}_{good}(r) \land \forall p,q \in \mathscr{C} : \vec{\mu}_p^r = \vec{\mu}_q^r$$

Therefore, WIC is formally expressed by the following predicate:

$$\boxed{\forall r : \mathscr{P}_{int}(r) \land \exists r : \mathscr{P}_{cons}(r)}$$

Note that $\mathscr{P}_{cons}(r)$ is stronger than $\mathscr{P}_{good}(r)$. Consider two correct processes p and q, and a Byzantine process sending a message m to all processes in round r: $\mathscr{P}_{good}(r)$ allows m to be received by p and not by q; $\mathscr{P}_{cons}(r)$ does not allow this.

3.4 Implementing WIC

For implementing WIC, we show in this section that rounds that satisfy $\mathscr{P}_{good}$ can be transformed into a round that satisfies $\mathscr{P}_{cons}$. This transformation can be formally expressed thanks to the notion of *predicate simulation*. Intuitively, an algorithm A is a predicate simulation of $\mathscr{P}'$ from $\mathscr{P}$, if several rounds where $\mathscr{P}$ holds simulate one round where $\mathscr{P}'$ holds. Formally, given (macro) round r, we say that an algorithm A is a k-round simulation of predicate $\mathscr{P}'$ (*e.g.*, $\mathscr{P}_{cons}$) from predicate $\mathscr{P}$ (*e.g.*, $\mathscr{P}_{good}$), if there is a sequence of k rounds $\langle r,1\rangle$ to $\langle r,k\rangle$ (rounds $\langle r,1\rangle$ to $\langle r,k\rangle$ define a macro-round r) such that: (i) $\mathscr{P}$ holds for each round $\langle r,i\rangle$, $i \in [1,k]$; (ii) each process p executes A in each round $\langle r,i\rangle$, $i \in [1,k]$; (iii) for each process p, the message m_p sent by p in round $\langle r,1\rangle$ is the message sent by p in (macro) round r; (iv) for each process p, the messages received by p in (macro) round r are computed by p at the end of round $\langle r,k\rangle$; and (v) $\mathscr{P}'$ holds for (macro) round r. We also say that (macro) round r is *simulated* by the k rounds $\langle r,1\rangle$ to $\langle r,k\rangle$.

Algorithm 3.1 Getting WIC from $\mathscr{P}_{good} \wedge \mathscr{P}_{int}$ with signatures

```
Initialization:
  $\forall q \in \Pi$ : $received_p[q] \leftarrow \bot$
Round $\rho = \langle r, 1 \rangle$:
  $S_p^\rho$:
    send $\sigma_p(m_p, r)$ to $coord(r)$
  $T_p^\rho$:
    if $p = coord(r)$ then
      $received_p \leftarrow \vec{\mu}_p^\rho$
Round $\rho = \langle r, 2 \rangle$:
  $S_p^\rho$:
    if $p = coord(r)$ then
      send $received_p$ to all
  $T_p^\rho$:
    for all $q \in \Pi$ do
      $\vec{M}_p[q] \leftarrow \bot$
      if signature of $\vec{\mu}_p^\rho[coord(r)][q]$ is valid then
        $(msg, round) \leftarrow \sigma^{-1}(\vec{\mu}_p^\rho[coord(r)][q])$
        if $round = r$ then
          $\vec{M}_p[q] \leftarrow msg$
```

We give two simulations, one with and one without digital signatures. Both simulations rely on a coordinator. The simulation with signatures requires two rounds with the communication pattern of Figure 3.1, whereas the simulation without signatures requires three rounds with the communication pattern of Figure 3.2. The coordinator of macro-round r is denoted by $coord(r)$. Interestingly, there is also a decentralized (i.e., coordinator-free) simulation of $\mathscr{P}_{cons}$, that is signature-free, and that requires $b+1$ rounds [BS10].

We will analyze the two simulations in the following cases: (i) $coord(r)$ is correct and the rounds satisfy $\mathscr{P}_{good}$, and (ii) $coord(r)$ may be faulty and only $\mathscr{P}_{int}$ holds for the rounds. In case (i), we have a translation of $\mathscr{P}_{cons}$ from $\mathscr{P}_{good}$. Case (ii) ensures that the translation is always harmless, even if the coordinator is faulty.

Therefore, the big picture is the following. [DLS88] shows how to implement rounds for which $\mathscr{P}_{good}$ eventually holds. Moreover, the rotating coordinator paradigm eventually ensures macro-rounds with a correct coordinator. Together, this eventually ensures case (i).

3.4.1 Simulation with signatures

Algorithm 3.1 is a 2-round simulation with signatures that preserves $\mathscr{P}_{int}$ (*i.e.*, if $\mathscr{P}_{int}$ holds for every round, then $\mathscr{P}_{int}$ holds for the macro-round). Moreover, when $coord(r)$ is correct, it simulated $\mathscr{P}_{cons}$ from $\mathscr{P}_{good}$. At the beginning of Algorithm 3.1 every process p has a message m_p (line 5); at the end every process p has a vector $\vec{M}_p$ of received messages (lines 15, 19) [5].

[5]When round r is simulated using Algorithm 3.1, m_p is initially set to the $S_p^r(s_p^r)$ and in the end $\vec{\mu}_p^r$ is set to $\vec{M}_p$.

Vector *received*$_p$ (line 8) represents the messages that p received (one element per process). Message m signed by p is denoted by $\sigma_p(m)$. The function σ^{-1} allows us to get back the original message out of a signed message.

Algorithm 3.1 is straightforward: each process p sends its signed message m_p to the coordinator (line 5) in round $\langle r, 1\rangle$. In round $\langle r, 2\rangle$, the coordinator forwards all messages received (line 12).

Proposition 3.1. *Algorithm 3.1 preserves* $\mathscr{P}_{int}(r)$.

Proof. Every process checks at lines 16 and 18 whether the signature and the macro-round number of the message are valid. Since signatures cannot be forged, for all correct processes p, q, if p receives $m \neq \perp$ from q, then q has sent m. □

Proposition 3.2. *If* $coord(r)$ *is correct, then Algorithm 3.1 simulates* $\mathscr{P}_{cons}$ *from* $\mathscr{P}_{good}$.

Proof. Since we have $\mathscr{P}_{good}(\langle r, 1\rangle)$, the correct coordinator receives in round $\langle r, 1\rangle$ the message from all correct processes, and possibly from some faulty processes. Since the coordinator is correct and we have $\mathscr{P}_{good}(\langle r, 2\rangle)$, all messages received by the coordinator are forwarded in round $\langle r, 2\rangle$, and received by all correct processes. □

From Propositions 3.1 and 3.2 follows directly that repeating Algorithm 3.1 with a rotating coordinator ensures WIC (after GST). Indeed, by Proposition 3.1, predicate $\mathscr{P}_{int}(r)$ holds for each (simulated) round r. During a synchronous period (after GST) we have $\mathscr{P}_{good}(r)$ for every r, and eventually a correct coordinator, *i.e.*, the assumptions of Proposition 3.2. By Proposition 3.2 we have $\mathscr{P}_{cons}(r)$ for some (simulated) round r. Thus WIC holds.

3.4.2 Simulation without signatures

Algorithm 3.2 is a 3-round simulation with signatures, inspired by [CL02], that preserves $\mathscr{P}_{int}$ (*i.e.*, if $\mathscr{P}_{int}$ holds for every round, then $\mathscr{P}_{int}$ holds for the macro-round). Moreover, when $coord(r)$ is correct, it simulated $\mathscr{P}_{cons}$ from $\mathscr{P}_{good}$. It requires $n \geq 3b + 1$. At the beginning of Algorithm 3.2 every process p has a message m_p (line 7); at the end every process p has a vector $\vec{M}_p$ of received messages (lines 22, 24) [6].

We informally explain Algorithm 3.2 using Figure 3.4. Compared to Figure 3.2, Figure 3.4 shows only the messages relevant to v_2 sent by p_2. Process p_1 is the coordinator. In round $\langle r, 1\rangle$, process p_2 sends v_2 to all. In round $\langle r, 2\rangle$, all processes send the value received from p_2 to the coordinator. The coordinator then compares the value received from p_2 in round $\langle r, 1\rangle$, say v_2, with the value indirectly received from the other processes. If at least $2b + 1$ values v_2 have been received by the coordinator p_1, then p_1 keeps v_2 as the value received from p_2.

[6] When round r is simulated using Algorithm 3.2, m_p is initially set to the $S_p^r(s_p^r)$ and in the end $\vec{\mu}_p^r$ is set to $\vec{M}_p$.

Algorithm 3.2 Getting WIC from $\mathscr{P}_{good} \wedge \mathscr{P}_{int}$ without signatures

```
Initialization:
  $\forall q \in \Pi : received_p[q] \leftarrow \bot$
Round $\rho = \langle r, 1 \rangle$:
  $S_p^\rho$:
    send $m_p$ to all
  $T_p^\rho$:
    $received_p \leftarrow \vec{\mu}_p^\rho$
Round $\rho = \langle r, 2 \rangle$:
  $S_p^\rho$:
    send $received_p$ to $coord(r)$
  $T_p^\rho$:
    if $p = coord(r)$ then
      for all $q \in \Pi$ do
        if $\left|\{q' \in \Pi : \vec{\mu}_p^\rho[q'][q] = received_p[q]\}\right| < 2b+1$ then
          $received_p[q] \leftarrow \bot$
Round $\rho = \langle r, 3 \rangle$:
  $S_p^\rho$:
    send $\langle received_p \rangle$ to all
  $T_p^\rho$:
    for all $q \in \Pi$ do
      if $(\vec{\mu}_p^\rho[coord(r)][q] \neq \bot) \wedge \left|\{i \in \Pi : \vec{\mu}_p^\rho[i][q] = \vec{\mu}_p^\rho[coord(r)][q]\}\right| \geq b+1$ then
        $\vec{M}_p[q] \leftarrow \vec{\mu}_p^\rho[coord(r)][q]$
      else
        $\vec{M}_p[q] \leftarrow \bot$
```

Otherwise p_1 sets the value received from p_2 to $\bot$. This guarantees that, if p_1 keeps v_2, then at least $b+1$ correct processes have received v_2 from p_2 in round $\langle r, 1\rangle$.

Finally, in round $\langle r, 3\rangle$ every process sends the value received from p_2 in round $\langle r, 1\rangle$ to all. The final value received from p_2 at the end of round $\langle r, 3\rangle$ is computed as follows at each process p_i. Let val_i be the value received by p_i from coordinator p_1 in round $\langle r, 3\rangle$. If val_i is $\bot$ then p_i receives $\bot$ from p_2. Process p_i receives $\bot$ from p_2 in another case: if p_i did not receive $b+1$ values equal to val_i in round $\langle r, 3\rangle$. Otherwise, at least $b+1$ values received by p_i in round $\langle r, 3\rangle$ are equal to val_i, and p_i receives val_i from p_2.

Proposition 3.3. *Algorithm 3.2 preserves $\mathscr{P}_{int}(r)$.*

Proof. Let p, q be two correct processes. Assume for the sake of contradiction that $S_p^r(s_p^r) = v$, $\vec{M}_q[p] = v'$, where $v' \neq v$, $v' \neq \bot$. Therefore, by line 21, we have $\left|\left\{i : \vec{\mu}_q^\rho[i][p] = v'\right\}\right| \geq b+1$. Consequently, for at least one correct process c we have $\vec{\mu}_q^\rho[c][p] = v'$. Element $\vec{\mu}_q^\rho[c][p]$ is the message received by c from p in round $\langle r, 1\rangle$, which is $received_c[p]$. However, $received_c[p] = v'$ is in contradiction with the assumption that p and c are correct. □

Proposition 3.4. *If $coord(r)$ is correct, then Algorithm 3.2 simulates $\mathscr{P}_{cons}$ from $\mathscr{P}_{good}$.*

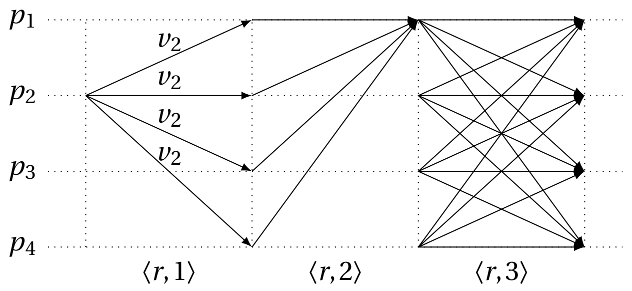


Figure 3.4: Simulation without signatures from the point of view of $\boldsymbol{v_2}$ sent by $\boldsymbol{p_2}$ ($\boldsymbol{p_1}$ is the coordinator).

Proof. Let p, q be two correct processes, and s some other process (not necessarily correct). Let c be the correct coordinator. Let $\mathscr{P}_{good}(\langle r,1\rangle)$, $\mathscr{P}_{good}(\langle r,2\rangle)$ and $\mathscr{P}_{good}(\langle r,3\rangle)$ hold. We first show (i) $\vec{M}_p[q] = S_q^r(s_q^r)$, and then (ii) $(\vec{M}_p[s] = v \neq \bot) \Rightarrow (\vec{M}_q[s] = v)$. Note that from (ii) it follows directly that $(\vec{M}_p[s] = \bot) \Rightarrow (\vec{M}_q[s] = \bot)$.

(i): In round $\langle r,1\rangle$, process q sends $v = S_q^r(s_q^r)$ to all, and because of $\mathscr{P}_{good}(\langle r,1\rangle)$, v is received by all correct processes. For all those correct processes i, we have $received_i[q] = v$ (*). In round $\langle r,2\rangle$, every correct process forwards v to the coordinator c, and c receives all these messages. Since $n \geq 3b+1$ there are at least $2b+1$ correct processes. Therefore the condition of line 14 is false for q because $\left|\{q' \in \Pi : \vec{\mu}_c^\rho[q'][q] = received_c[q]\}\right| \geq 2b+1$, *i.e.*, $received_c[q]$ is not set to $\bot$. By (*) above, we have $received_c[q] = v$. Because of $\mathscr{P}_{good}(\langle r,3\rangle)$ all messages sent by correct processes in round $\langle r,3\rangle$ are received by all correct processes. Thus, for p at line 21, we have $\vec{\mu}_p^\rho[coord(r)][q] \neq \bot$. Moreover, by (*), condition $\left|\{i \in \Pi : \vec{\mu}_p^\rho[i][q] = \vec{\mu}_p^\rho[coord(r)][q]\}\right| \geq b+1$ is true. This leads p to execute line 22, *i.e.*, assign v to $\vec{M}_p[q]$.

(ii): Let us assume $\vec{M}_p[s] = v \neq \bot$, and consider Algorithm 3.2 from the point of view of p. Consider the loop at line 20 for process s. By line 22, we have $\vec{\mu}_p^\rho[coord(r)][s] = v$. Since the coordinator is correct, in order to have $\vec{\mu}_p^\rho[coord(r)][s] = v$, the condition of line 14 is true at c for process s, *i.e.*, $\left|\{q' \in \Pi : \vec{\mu}_c^\rho[q'][s] = received_c[s]\}\right| \geq 2b+1$. This means that at least $2b+1$ processes, including at least $b+1$ correct processes, have received from s in round $\langle r,1\rangle$ the same message that c received from s, namely v ($\star$). In round $\langle r,3\rangle$, these $b+1$ correct processes send *received* to all. Because $\mathscr{P}_{good}(\langle r,3\rangle)$ holds, all these messages are received by q in round $\langle r,3\rangle$ ($\star\star$). Consider now Algorithm 3.2 from the point of view of q, and again the loop at line 20 for process s. Since the coordinator is correct, it sends at line 18 the same message to p and to q, *i.e.*, at q we also have $\vec{\mu}_q^\rho[coord(r)][s] = v$. By ($\star$) and ($\star\star$), the condition $\left|\left\{i \in \Pi : \vec{\mu}_q^\rho[i][s] = \vec{\mu}_q^\rho[coord(r)][s]\right\}\right| \geq b+1$ is true. Therefore q executes line 22 with $\vec{\mu}_p^\rho[coord(r)][s] = v$. □

From Propositions 3.3 and 3.4 follows directly that repeating Algorithm 3.2 in the basic round model with a rotating coordinator ensures WIC.

3.5 Achieving Consensus with WIC

In this section we show how to express the consensus algorithms of Castro-Liskov [CL02] and Martin-Alvisi [MA06] using WIC. The algorithm of Castro and Liskov solves a sequence of instances of consensus (state machine replication). For simplicity, we consider only one instance of consensus.

Both, [CL02] and [MA06] achieve only weak validity (see Section 2.1). Weak validity allows correct processes to decide on the value that is not initial value of some correct process. With strong unanimity (see Section 2.1), however, this is only possible if not all correct processes have the same initial value. We give algorithms for both, weak validity and strong unanimity, and show that strong unanimity is in fact easy to ensure.

3.5.1 On the use of WIC

We express the algorithms of this section in the round model defined in Section 2.5. All rounds of MA and CL require $\mathscr{P}_{int}$ to hold. Some of the rounds require $\mathscr{P}_{cons}$ to eventually hold. These rounds can be simulated using, *e.g.*, Algorithm 3.1 or Algorithm 3.2. We explicitly mention those rounds of MA and CL as rounds "in which $\mathscr{P}_{cons}$ must eventually hold". The other rounds of MA and CL are ordinary rounds.

3.5.2 MA algorithm

The algorithm of Martin and Alvisi [MA06] is expressed in the context of "proposers", "acceptors" and "learners". For simplicity, we express here consensus without considering these roles.

We give two algorithms. The first solves consensus with weak validity and is given as Algorithm 3.3. In the first phase it corresponds to the "common case" protocol of [MA06]. All later phases correspond to the "recovery protocol" of [MA06] (see Algorithm 3.4). The second algorithm solves consensus with strong unanimity, and is even simpler: all phases are identical, see Algorithm 3.4. In both algorithms, the notation $\#(v)$ is used to denote the number of messages received with value v, *i.e.*, $\#(v) \equiv \left|\left\{q \in \Pi : \vec{\mu}_p^r[q] = v\right\}\right|$.

For MA with weak validity, the first phase needs an initial coordinator, which is denoted by *coord*. Note that WIC is relevant only to rounds $2\phi - 1$, $\phi > 1$, of Algorithm 3.4. If rounds $2\phi - 1$ are simulated using Algorithm 3.1, we get the original algorithm of [MA06]. If rounds $2\phi - 1$ are simulated using Algorithm 3.2, we get a new algorithm. In this new algorithm, similarly to the algorithm in [MA06], fast decision is possible in two rounds; however, signatures are not used in the recovery protocol.

The basic technique of these algorithm is that a value that is decided is locked in the sense that a sufficiently high quorum of processes retain this value as estimate. A similar algorithmic

Algorithm 3.3 MA (weak validity)

```
Initialization:
    $x_p \leftarrow v_p \in V$                    /* $v_p$ is $p$'s initial value */
Round $r = 1$:
    $S_p^r$:
        if $p = coord$ then
            send $x_p$ to all
    $T_p^r$:
        if $\vec{\mu}_p^r[coord] \neq \perp$ then
            $x_p \leftarrow \vec{\mu}_p^r[coord]$
Round $r = 2$:
    $S_p^r$:
        send $x_p$ to all
    $T_p^r$:
        if $\exists \bar{v} \neq \perp : \#(\bar{v}) \geq \lceil (n+3b+1)/2 \rceil$ then
            DECIDE $\bar{v}$
Round $r \geq 3$:
    Same as Algorithm 3.4 without Initialization
```

schema can be found in algorithms for benign [BGMR01, PSUC02, Lam05, CBS09b] and arbitrary [BCBG$^+$07] faults.

The algorithms consist of a sequence of phases, where each phase ϕ has two rounds $2\phi - 1$ and 2ϕ. Every process p maintains a single variable x_p initialized to p's initial value. In round $2\phi - 1$ of Algorithm 3.4 (that is used also in phases $\phi > 1$ for Algorithm 3.3), if a process p receives at least $n - b$ messages then it updates x_p, and sets x_p to *the smallest most often received value* of the current round. In round 2ϕ, if a process p receives at least $\lceil (n+3b+1)/2 \rceil$ times the same value v then it decides on v. The values of thresholds are chosen such that if some process decides v in a round r, then in any round $r' > r$, at all correct processes p only v can be assigned to x_p. This is the case as in any set of $n - b$ messages received in round $r' > r$ by a correct process, v is always the most often received value. Therefore, in rounds $r' > r$ only v can be decided.

Both algorithms require $n \geq 5b + 1$. Agreement, weak validity and strong unanimity hold without synchrony assumptions. Termination requires (i) one phase ϕ such that $\mathscr{P}_{cons}(2\phi - 1)$ holds, and (ii) one phase $\phi' \geq \phi$ such that $\mathscr{P}_{good}(2\phi')$ holds. [7]

The first part of the predicate ensures the existence of a round in which correct processes receive the same set of at least $n - b$ messages which includes messages from all correct processes. This guarantees that at the end of round $2\phi - 1$ all correct processes adopt the same value for x_p. The second part of the predicate forces every correct process to make a decision at the end of round $2\phi'$.

[7] For simplicity, we have not included a boolean to prevent a process from deciding more than once, *e.g.*, Algorithm 3.4, line 14.

Algorithm 3.4 MA (strong unanimity)

```
Initialization:
  $x_p \leftarrow v_p \in V$                                   /* $v_p$ is $p$'s initial value */
Round $r = 2\phi - 1$:                         /* round in which $\mathcal{P}_{cons}$ must eventually hold */
  $S_p^r$:
    send $x_p$ to all
  $T_p^r$:
    if $\#(\bot) \le b$ then
      $x_p \leftarrow \min\{v : \nexists v' \in V\ s.t.\ \#(v') > \#(v)\}$
Round $r = 2\phi$:
  $S_p^r$:
    send $x_p$ to all
  $T_p^r$:
    if $\exists \bar{v} \neq \bot : \#(\bar{v}) \ge \lceil (n+3b+1)/2 \rceil$ then
      DECIDE $\bar{v}$
```

Theorem 3.1. *If $n \ge 5b+1$ then Algorithm 3.3 (resp. Algorithm 3.4) ensures weak validity (resp. strong unanimity) and agreement. Termination holds if in addition the following condition holds:*

$$\exists \phi : \mathcal{P}_{cons}(2\phi - 1) \land \exists \phi' \ge \phi : \mathcal{P}_{good}(2\phi')$$

Proof.
The proofs for termination with weak validity and strong unanimity are the same, and the proofs for agreement are almost identical. Weak validity is trivially satisfied. Therefore, we prove only MA with strong unanimity (Algorithm 3.4).

Agreement: Assume for a contradiction that process p decides v in round $r = 2\phi$, and process p' decides $v' \neq v$ in round $r' = 2\phi'$. W.l.o.g. assume $\phi' \ge \phi$.

If $\phi = \phi'$ then $\lceil (n+3b+1)/2 \rceil - b$ correct processes have sent v to p and $\lceil (n+3b+1)/2 \rceil - b$ correct processes have sent v' to p'. Since $2(\lceil (n+3b+1)/2 \rceil - b) + b > n$, there is one correct process q that has sent v to p and v' to p'. A contradiction with the assumption that q is correct.

Else, we have $\phi' > \phi$. By line 13, at least $\lceil (n+3b+1)/2 \rceil - b$ correct processes p have $x_p = v$ at the end of phase ϕ. We show now that for all phases $\phi'' > \phi$, every time line 8 is executed at some correct process q, x_q is updated only to v. By the condition of line 7, q has received at least $n-b$ values different from $\bot$. In any subset of size $\ge n-b$, at least $\lceil (n+3b+1)/2 \rceil - 2b$ values are v and at most $n - \lceil (n+3b+1)/2 \rceil + b$ are $\neq v$; thus because of $\lceil (n+3b+1)/2 \rceil - 2b > n - \lceil (n+3b+1)/2 \rceil + b$ no value can occur more often in $\vec{\mu}$ than v.

Therefore, in all phases $\phi'' > \phi$, at least $\lceil (n+3b+1)/2 \rceil - b$ correct processes p have $x_p = v$. It follows directly that only v can be decided in these phases, and thus also in ϕ'.

Strong unanimity: If $n \ge 3b+1$, then $n - b \ge \lceil (n+3b+1)/2 \rceil - b$. Therefore if all correct processes have the same initial value v, we have initially at least $\lceil (n+3b+1)/2 \rceil - b$ processes p with $x_p = v$. By an argument used in the proof of agreement, only v can be decided.

Termination: Let ϕ_0 be such that $\mathscr{P}_{cons}(2\phi_0 - 1)$ holds. Therefore the condition of line 7 is true for all correct processes. Moreover, $\mathscr{P}_{cons}(2\phi_0 - 1)$ ensures that all correct processes p, when executing line 8, set x_p to the same value, say v. By an argument used in the proof of agreement, after phase ϕ_0, correct processes p can only update x_p to v at line 8.

Let $\phi_0' \geq \phi_0$ such that $\mathscr{P}_{good}(2\phi_0')$ holds. In round $2\phi_0'$, $n-b$ correct processes send v. If $n \geq 5b+1$, then $n-b \geq \lceil (n+3b+1)/2 \rceil$; therefore the condition of line 13 is true for all correct processes, which decide at line 14. □

Note that $n \geq 5b+1$ is only needed for termination, while only $n \geq 3b+1$ is needed for agreement and strong unanimity.

3.5.3 CL algorithm

The algorithm of Castro and Liskov [CL02] solves a sequence of instances of consensus (state machine replication). For simplicity, we consider only one instance of consensus. As for MA, we give two algorithms.

The first solves consensus with weak validity and is given as Algorithm 3.5. The first phase corresponds to the "normal case" protocol of [CL02]. All later phases correspond to the "view change protocol" of [CL02] (cf. Algorithm 3.6). The second algorithm solves consensus with strong unanimity, and is even simpler: all phases are identical, see Algorithm 3.6. In both algorithms, the notation $\#(v)$ is used to denote the number of messages received with value v, *i.e.*, $\#(v) \equiv \left| \left\{ q \in \Pi : \vec{\mu}_p^r[q] = v \right\} \right|$.

For CL with weak validity, the first phase needs an initial coordinator, which is denoted by *coord*. WIC is relevant only to rounds $3\phi - 2$, $\phi > 1$ (cf. Algorithm 3.6). If rounds $3\phi - 2$, $\phi > 1$ are simulated using Algorithm 3.2, we get an algorithm close to the original algorithm of [CL02]. If rounds $3\phi - 2$, $\phi > 1$ are simulated using Algorithm 3.1, we get a variant of PBFT with signatures.

Both algorithms (CL with weak validity and CL with strong unanimity) require $n \geq 3b+1$. Agreement, weak validity and strong unanimity hold without synchrony assumptions. Termination requires (i) one phase ϕ such that $\mathscr{P}_{cons}(3\phi - 2)$, $\mathscr{P}_{good}(3\phi - 1)$ and $\mathscr{P}_{good}(3\phi)$ hold. In the following we explain the basic mechanisms behind CL with strong unanimity (the same mechanisms are also used in CL with weak validity).

De discuss now Algorithm 3.6. Algorithm 3.6 consists of a sequence of phases, where each phase ϕ has three rounds $3\phi - 2$, $3\phi - 1$ and 3ϕ. The algorithm is based on the *last voting* mechanism [CBS09b] that was first introduced in the Paxos algorithm by Lamport [Lam98] for benign faults. More precisely, the algorithm uses a timestamp variable *tVote* in addition to

Algorithm 3.5 CL (weak validity)

```
Initialization:
  $x_p \leftarrow v_p \in V$                                  /* $v_p$ is the initial value of $p$ */
  $pre\text{-}vote_p \leftarrow \emptyset$                     /* see Algorithm 3.6 */
  $vote_p \leftarrow \perp$                                    /* see Algorithm 3.6 */
  $tVote_p \leftarrow 0$                                       /* see Algorithm 3.6 */
Round $r = 3\phi - 2 = 1$:
  $S_p^r$:
    if $p = coord$ then
      send $\langle x_p \rangle$ to all
  $T_p^r$:
    if $\vec{\mu}_p^r[coord] \neq \perp$ then
      add $(\vec{\mu}_p^r[coord], \phi)$ to $pre\text{-}vote_p$
Round $r = 3\phi - 1 = 2$:
  $S_p^r$:
    if $\exists (v, \phi) \in pre\text{-}vote_p$ then
      send $\langle v \rangle$ to all
  $T_p^r$:
    if $\#(v) \geq \lceil (n + b + 1)/2 \rceil$ then
      $vote_p \leftarrow v$
      $tVote_p \leftarrow \phi$
Round $r = 3\phi = 3$:
  $S_p^r$:
    if $tVote_p = \phi$ then
      send $\langle vote_p \rangle$ to all
  $T_p^r$:
    if $\exists \bar{v} \neq \perp : \#(\bar{v}) \geq \lceil (n + b + 1)/2 \rceil$ then
      DECIDE $\bar{v}$
Round $r \geq 4$:
  Same as Algorithm 3.6 without Initialization
```

to the variable *vote*. Whenever a process p updates $vote_p$ in round $3\phi - 1$, $tVote_p$ is set to ϕ (line 31 and 32). If enough processes update *vote* in round $3\phi - 1$, then a decision is possible in round 3ϕ. Note the condition at line 30: It ensures that in round $3\phi - 1$, all processes that update *vote*, update it to the same value. This ensures that in round 3ϕ, processes attempt to decide on one single value, which is necessary for agreement.

In order to deal with invalid votes sent by Byzantine processes, CL maintains also a *pre-vote* variable, which stores pairs (v, ϕ). Having $(v, \phi) \in pre\text{-}vote_p$ means that p added (v, ϕ) to $pre\text{-}vote_p$ in phase ϕ (at line 20 or line 24). The *pre-vote* variable ensures that a message with invalid values for *vote* and *tVote* will not affect the safety properties of the algorithm. It is mainly used in round $3\phi - 2$, which has two roles, the first related to agreement and strong unanimity, and the second related to termination:

Algorithm 3.6 CL (strong unanimity)

```
1:  Initialization:
2:    $x_p \leftarrow v_p \in V$                    /* $v_p$ is the initial value of $p$ */
3:    $pre\text{-}vote_p \leftarrow \emptyset$        /* set of $(v,\phi)$, where $\phi$ is the phase in which $v$ is added to $pre\text{-}vote_p$ */
4:    $vote_p \leftarrow \bot$                       /* the most recent vote */
5:    $tVote_p \leftarrow 0$                         /* phase in which $vote_p$ was last updated */
6:  Procedure $pre\text{-}vote_p.add(v,\phi)$ :
7:    if $\exists (v,\phi') \in pre\text{-}vote_p$ then
8:      remove $(v,\phi')$ from $pre\text{-}vote_p$
9:    add $(v,\phi)$ to $pre\text{-}vote_p$
10: Round $r = 3\phi - 2$:                           /* round in which $\mathscr{P}_{cons}$ must eventually hold */
11:   $S_p^r$:
12:     send $\langle vote_p, tVote_p, pre\text{-}vote_p, x_p \rangle$ to all
13:   $T_p^r$:
14:     $proposals_p \leftarrow \emptyset$ ; $I_p \leftarrow \emptyset$          /* temporary variables */
15:     if $\vec{\mu}_p^r$ contains at least $\lceil (n+b+1)/2 \rceil$ messages $\langle vote, tVote, pre\text{-}vote, x \rangle$ then
16:       for all $m \in \vec{\mu}_p^r$ do
17:         if $\left|\left\{m' \in \vec{\mu}_p^r : (m'.tVote < m.tVote) \lor (m'.tVote = m.tVote \land m'.vote = m.vote)\right\}\right| \geq$
            $\lceil (n+b+1)/2 \rceil$ and
            $\left|\left\{m' \in \vec{\mu}_p^r : \exists (v,\phi') \in m'.pre\text{-}vote \; s.t. \; \phi' \geq m.tVote \land v = m.vote\right\}\right| \geq b+1$ then
18:           $proposals_p \leftarrow proposals_p \cup m.vote$
19:       if $\left|proposals_p\right| > 0$ then
20:         $pre\text{-}vote_p.add(\min(proposals_p), \phi)$
21:       else if exist at least $\lceil (n+b+1)/2 \rceil$ messages $m' \in \vec{\mu}_p^r : m'.vote = \bot$ then
22:         $I_p \leftarrow \left\{m.x \; s.t. \; m \in \vec{\mu}_p^r\right\}$
23:         $\overline{x} \leftarrow \min\left\{v : \nexists v' \in I_p \; s.t. \; \#(v') > \#(v)\right\}$
24:         $pre\text{-}vote_p.add(\overline{x}, \phi)$
25: Round $r = 3\phi - 1$:
26:   $S_p^r$:
27:     if $\exists (v,\phi) \in pre\text{-}vote_p$ then
28:       send $\langle v \rangle$ to all
29:   $T_p^r$:
30:     if $\#(v) \geq \lceil (n+b+1)/2 \rceil$ then
31:       $vote_p \leftarrow v$
32:       $tVote_p \leftarrow \phi$
33: Round $r = 3\phi$:
34:   $S_p^r$:
35:     if $tVote_p = \phi$ then
36:       send $\langle vote_p \rangle$ to all
37:   $T_p^r$:
38:     if $\exists \bar{v} \neq \bot : \#(\bar{v}) \geq \lceil (n+b+1)/2 \rceil$ then
39:       DECIDE $\bar{v}$
```

1. *Safety role of round* $3\phi - 2$*:*

   (a) *For Agreement*: If a process p has decided v in some phase ϕ_0, then for any process q, only v can be added to $pre\text{-}vote_q$ (at line 20 or line 24) in phases $\phi > \phi_0$.

(b) *For Strong Unanimity*: If all processes have the same initial value v, then only v can be added to $pre\text{-}vote_q$ (at line 20 or line 24).

2. *Termination role of round* $3\phi - 2$: In a consistent round where correct processes receives messages from all correct processes, all processes add the same value to $pre\text{-}vote_q$ (at line 20 or line 24).

We explain now the lines 15-24 of Algorithm 3.6:

- Lines 15-18 ensures 1a. More precisely, they ensure the selection of the most recent vote in the set *pre-vote* of some correct process. This is basically the same mechanism as in Paxos, adapted to tolerate invalid votes sent by Byzantine faults. Selecting the most recent vote among the set of majority processes (in Paxos) can be expressed as follows:

  $$mostRecentV \leftarrow \left\{ v : \ (v, tVote) \in \vec{\mu}_p^r \ \wedge \left| \left\{ q : \vec{\mu}_p^r[q] = (v', tVote') \wedge tVote \geq tVote' \right\} \right| > \frac{n}{2} \right\}$$

  In Paxos, this selection rule ensures agreement since the most recent vote is always v if some process decided v before. In CL, a message sent by a Byzantine process can contain $(vote, tVote)$ with $tVote$ equal (or higher) to the highest timestamp of a process, but with a $vote$ that is different than some value v decided by some correct process before. Therefore, the above selection rule does not ensure 1a, since several values can satisfy the condition. More precisely, in case some correct process has decided a value v before, the invalid vote $(v', tVote')$ with $v' \neq v$ sent by a Byzantine process might be selected if $tVote$ is high enough. Therefore, in order to transform the condition to ensure 1a even in the presence of Byzantine processes, it is necessary to: (i) transform condition $tVote \geq tVote'$ into

  $$tVote > tVote' \vee (v = v' \wedge tVote = tVote')$$

  and to use a higher threshold (namely $\lceil (n+b+1)/2 \rceil$) as done by the first part of the condition at line 17, and (ii) to prevent selection of a value v' from an invalid pair $(v', tVote)$ sent by a Byzantine process. This is achieved by the second part of the condition at line 17 that requires that selected pair must be in the *pre-vote* of at least one correct process (ensured by the threshold $b+1$). With this, if a process has previously decided $\bar{v}$, then only $\bar{v}$ can be selected by the condition at line 17 and added to $proposals_p$ at line 18. [8]

[8] Consider two phases ϕ_0 and $\phi_0 + 1$, such that a process has decided $\bar{v}$ in phase ϕ_0. We assume that $n = 4$ and $b = 1$. This means that at least $\lceil (n+b+1)/2 \rceil - b = 2$ correct processes have $tVote = \phi_0$ and $vote = \bar{v}$. Consider in phase $\phi_0 + 1$ that $(v, tVote) \in proposals_p$ at a correct process p with $v \neq \bar{v}$. This means that p, in round $3(\phi_0+1) - 2$, received $\lceil (n+b+1)/2 \rceil = 3$ messages with either $(v, tVote, -, -)$, or $(-, tVote', -, -)$ and $tVote' < tVote$. Since $n = 4$ and $b = 1$, at least one of these messages is from a correct process c such that $vote_c = \bar{v}$ and $tVote_c = \phi_0$. Since $v \neq \bar{v}$, we must have $\phi_0 < tVote$. However, in phase $\phi_0 + 1$, no correct process p can have $(v, tVote)$ with $ts > \phi_0$ in $pre\text{-}vote_p$. Therefore, by the second part of the condition at line 17, we will not have $v \in proposals_p$.

We consider now lines 19 and 20 of Algorithm 3.6. As we just explained, if a correct process has previously decided $\bar{v}$, then at a correct process p only $\bar{v}$ can be in $proposals_p$, that is, $|proposals| = 1$. In this case, by line 20, the function $(\bar{v}, \phi)$ is added to $pre\text{-}vote_p$. If no correct process has decided, we can have $|proposals| > 1$. In this case, if some round $3\phi - 2$ is a consistent round, then all processes consider the same set *proposals*, which ensures 2.

Lines 21 and 21 ensure 1b: in case no $(vote, tVote)$ pair is selected by the condition at line 17 ($|proposals_p| = 0$) [9], it is ensured that in case all correct processes have the same initial value v, v is always selected. This is achieved by selecting the smallest most often received initial value (line 23).

Theorem 3.2. *If $n \geq 3b + 1$ then Algorithm 3.5 (resp. Algorithm 3.6) ensures weak validity (resp. strong unanimity) and agreement. Termination holds if in addition the following condition holds:*

$$\exists \phi : \mathscr{P}_{cons}(3\phi - 2) \wedge \mathscr{P}_{good}(3\phi - 1) \wedge \mathscr{P}_{good}(3\phi).$$

The result follows from the proof of agreement, strong unanimity and termination. Weak validity is trivially satisfied. We start with two definitions.

Definition 3.1. *Correct process p has* pre-prepared *value v in phase ϕ if $(v, \phi) \in pre\text{-}vote_p$ at the end of phase ϕ.*

Definition 3.2. *Correct process p has* prepared *value v in some phase ϕ if $vote_p = v$ and $tVote_p = \phi$ at the end of phase ϕ.*

Proof of agreement

Lemma 3.1. *For all $b \geq 0$, any two sets of size $\lceil (n + b + 1)/2 \rceil$ have at least one correct process in common.*

Proof. We have $2\lceil (n + b + 1)/2 \rceil \geq n + b + 1$. This means that the intersection of two sets of size $\lceil (n + b + 1)/2 \rceil$ contains at least $b + 1$ processes, *i.e.*, at least one correct process. The result follows directly from this. □

Lemma 3.2. *If some correct process q decides v in phase ϕ_0, then in all phases $\phi > \phi_0$, all correct processes can only pre-prepare value v.*

Proof. We prove the result by induction on ϕ.

Base step $\phi = \phi_0 + 1$: Assume by contradiction that p is some correct process that pre-prepares $v' \neq v$ in phase $\phi_0 + 1$. This implies that either (i) line 20 or (ii) line 24 was executed by p in phase $\phi_0 + 1$ where v' was pre-prepared by p.

[9]For instance this is the case initially as $vote = \perp$ and $pre\text{-}vote = \emptyset$ at all correct processes.

For (ii), the conditions of line 15 and line 21 have both to be true. If the condition of line 15 is true, this implies that $\vec{\mu}_p^r$ contains at least $\lceil (n+b+1)/2 \rceil$ messages. Since q has decided in phase ϕ_0, q received at least $\lceil (n+b+1)/2 \rceil$ messages with v at line 38. All correct processes c who sent a message with v have prepared v in phase ϕ_0 (see lines 31, 32 and 35), *i.e.*, $vote_c = v$ and $tVote_c = \phi_0$. Let us denote this set of correct processes with Q_c. By Lemma 3.1 the intersection of two sets of size $\lceil (n+b+1)/2 \rceil$ contains at least one correct process. Therefore, in the $\lceil (n+b+1)/2 \rceil$ messages received (line 15) there is at least one message sent by process from Q_c, *i.e.*, the condition at line 21 cannot be true. So line 20 (case (i)) was executed by p.

For (i), the conditions at line 15, line 17 and line 19 have to be true. We show that if the condition at line 15 is true, and the first part of the condition at line 17 is true, then the second part of the condition at line 17 is false, which establishes the contradiction. Let us denote by $m_{v'}$ the message that leads p to pre-prepare $v' \neq v$, *i.e.*, $m_{v'} \in \vec{\mu}_p^r$ and $m_{v'}.vote = v'$. By Lemma 3.1, $\vec{\mu}_p^r$ at line 16 contains at least one message m' sent by a process in Q_c, *i.e.*, $m'.vote = v$ and $m'.tVote = \phi_0$. So the first part of the condition at line 17 can only be true for $m_{v'}$ if $\left|\left\{m' \in \vec{\mu}_p^r : (m'.tVote < m_{v'}.tVote)\right\}\right| \geq \lceil (n+b+1)/2 \rceil$. This holds only if $m_{v'}.tVote > \phi_0$ (*), since (as shown above) any set of size $\lceil (n+b+1)/2 \rceil$ contains at least one message m sent by a process in Q_c, *i.e.*, $m.tVote = \phi_0$.

The second part of the condition at line 17, because of the condition $\geq b+1$, can only be true for $m_{v'}$ if there is a message $\overline{m}$ in $\vec{\mu}_p^r$ sent by a correct process $\overline{c}$ such that: $(\overline{v}, \overline{\phi}) \in pre\text{-}vote_{\overline{c}}$ (**) and $\overline{\phi} \geq m_{v'}.tVote$ and $\overline{v} = m_{v'}.vote$. However, for any correct process $\overline{c}$, if $(\overline{v}, \overline{\phi}) \in pre\text{-}vote_{\overline{c}}$, then $\overline{\phi} \leq \phi_0$ (***). From (**) and (***) we get $\phi_0 \geq m_{v'}.tVote$: a contradiction with $m_{v'}.tVote > \phi_0$, see (*).

Induction step from ϕ to $\phi+1$: Arguments similar to the base step can be used to prove the induction step. □

Lemma 3.3. *If v is the only value that can be pre-prepared by correct processes in phase ϕ, then v is the only value that can be prepared in phase ϕ.*

Proof. If v is the only value that can be pre-prepared by correct processes in phase ϕ, then v is the only value that can be sent by correct process at line 28 in phase ϕ. Because there are at most t Byzantine processes, and $t < \lceil (n+b+1)/2 \rceil$, for all correct processes holds that if exists some value that satisfies the condition at line 30, then it must be v. So v is the only value that can be prepared by correct processes at line 31 in phase ϕ. □

Proposition 3.5. *Algorithm 3.6 ensures agreement if $n \geq 3b+1$.*

Proof. Let ϕ_0 be the first phase in which some correct process decides v. Since $b < n/3$, line 38 ensures that another correct process that decides in phase ϕ_0 also decides v. By Lemma 3.2 and Lemma 3.3, in all phases $\phi > \phi_0$, all correct processes can only set $vote_p$ to v. So in round $r = 3\phi$, correct processes cannot decide a value different from v. □

Proof of strong unanimity

Strong unanimity follows from the following two lemmas.

Lemma 3.4. *If $n \geq 3b+1$, then any set of $\lceil (n+b+1)/2 \rceil$ processes contains a majority of correct processes.*

Proof. We have $\lceil (n+b+1)/2 \rceil \geq (n+b+1)/2$. If $n \geq 3b+1$, then $(n+b+1)/2 \geq (3b+1+b+1)/2 = 2b+1$. Therefore, $\lceil (n+b+1)/2 \rceil \geq 2b+1$. □

Lemma 3.5. *If all correct processes have the same initial value v, then in all phases ϕ, v is the only value that can be pre-prepared by correct processes.*

Proof. Assume by contradiction that ϕ is the first round where a value different from v is pre-prepared at some correct process p. This implies that either (i) line 20 or (ii) line 24 was executed. By assumption, we have $(v, --) \in \textit{pre-vote}_p$ or $\textit{pre-vote}_p = \emptyset$.

For (i), line 19, line 17 and line 15 have to be true. If $\textit{pre-vote}_p = \emptyset$, the second part of the condition at line 17 is always false. If $\textit{pre-vote}_p \neq \emptyset$, only values $(v, --)$ are in $\textit{pre-vote}_p$, and thus the second part of the condition at line 17 can be true only for message $m \in \vec{\mu}_p^r$ such that $m.vote = v$.

For (ii), line 24 is executed, *i.e.*, the conditions at line 21 and line 15 have to be true. This means that $\vec{\mu}_p^r$ contains at least $\lceil (n+b+1)/2 \rceil$ messages. By Lemma 3.4, there is a majority of messages sent by correct processes in $\vec{\mu}_p^r$. Since all correct processes have the same initial value v, $\overline{x}$ is set to v at line 23, and p pre-prepares v.

So v is the only value that can be pre-prepared by correct processes in phase ϕ. Contradiction. □

Proposition 3.6. *If $n \geq 3b+1$, Algorithm 3.6 ensures strong unanimity.*

Proof. Assume that all correct processes have the same initial value v. By Lemma 3.5, v is the only value that can be prepared by correct processes. By Lemma 3.3, v is the only value that can be prepared by correct processes. Therefore, v is the only value that can be sent by correct processes at line 36 (*). If $n > b$, we have $\lceil (n+b+1)/2 \rceil > b$ (**). From (*) and (**), it follows that the condition at line 38 can only be true for v, *i.e.*, v is the only value that can be decided at line 39. □

Proof of termination

Proposition 3.7. *If $n \geq 3b+1$ and $\exists \phi_0 : \mathcal{P}_{cons}(3\phi_0 - 2) \wedge \mathcal{P}_{good}(3\phi_0 - 1) \wedge \mathcal{P}_{good}(3\phi_0)$, then Algorithm 3.6 ensures termination.*

Proof. Predicate $\mathscr{P}_{cons}(3\phi_0 - 2)$ ensures that, in round $3\phi_0 - 2$, for any two correct processes p and q, we have $\vec{\mu}_p^r = \vec{\mu}_q^r$, with at least $n - b$ messages in $\vec{\mu}_p^r$ (1). If $n \geq 3b+1$, we have $n - b \geq \lceil (n+b+1)/2 \rceil$ (2). (1) and (2) ensure that the condition of line 15 is true at each correct process in phase ϕ_0.

Part A: We prove that all correct processes will pre-prepare the same value at line 20 or 24 in phase ϕ_0. There are two cases to consider: (i) some correct process prepared a value in some phase smaller than ϕ_0, or (ii) there is no such process.

Case (i): Let $\phi < \phi_0$ be the largest phase in which some correct process prepared some value v (line 31). By the condition of line 30, if $n > b$ then all correct processes that prepare a value in phase ϕ, prepare the same value v. If $n \geq 3b+1$, we have $n - t \geq \lceil (n+b+1)/2 \rceil$. It follows that in case (i) the first part of the condition at line 17 holds for at least one message m (3).

We consider now the second part (*i.e.*, the second line) of that condition. If $n \geq 3b+1$, we have $\lceil (n+b+1)/2 \rceil - b \geq b+1$. Therefore if p prepares v in phase ϕ, by the condition of line 30, at least $b+1$ correct processes have pre-prepared v in phase ϕ. If v is pre-prepared by p in phase ϕ, then v stays pre-prepared by p (see lines 7–9). Therefore the second part of the condition at line 17 holds for at least one message m (4).

From (3) and (4), it follows that the condition of line 19 is true at all correct processes in phase ϕ_0. Moreover, predicate $\mathscr{P}_{cons}(3\phi_0 - 2)$ ensures that for two correct processes p and q, we have $proposals_p = proposals_q$. Therefore p and q pre-prepare the same value at line 20.

Case (ii): By hypothesis, for all correct processes p, we have $vote_p = \perp$. Predicate $\mathscr{P}_{cons}(3\phi - 2)$ ensures that $\vec{\mu}_p^r$ contains the message of all correct processes. If $n \geq 3b+1$, we have $n - b \geq \lceil (n+b+1)/2 \rceil$. Therefore the condition at line 21 is true at each correct process. Moreover, since for any two correct process p and q we have $\vec{\mu}_p^r = \vec{\mu}_q^r$, all correct processes will assign the same value to $\overline{x}$ (line 23), and pre-prepare the same value at line 24.

Part B: From Part A, there exists a value v such that all correct processes p have $(v, \phi_0) \in$ $pre\text{-}vote_p$ at the beginning of round $3\phi_0 - 1$. Therefore all correct processes send v to all at line 28. The predicate $\mathscr{P}_{good}(3\phi_0 - 1)$ ensures that all correct processes receive all these messages, set $vote_p$ to v (line 31), and send v to all at line 36. The predicate $\mathscr{P}_{good}(3\phi_0)$ ensures that all correct processes receive all these messages, and decide at line 39 in phase ϕ_0. □

CL *vs.* PBFT

As mentioned in Section 3.1, replacing in CL round $3\phi - 2$ with a signature-free WIC implementation basically leads to the original signature-free PBFT algorithm. There are a few differences.

1. CL assumes $n \geq 3b+1$ while PBFT assumes for simplicity $n = 3b+1$. This explains why $\lceil (n+b+1)/2 \rceil$ appears in CL instead of $2b+1$ in PBFT.

2. In PBFT a process p may wait for more the $n-b$ messages. This happens each time p can know, based on the content of messages, that it received messages from Byzantine processes. Indeed, if p knows that x messages are from Byzantine processes, and since channels are reliable, it is safe for p to wait for $n-(b-x)$ messages. Such mechanism in which a process looks at the content of the message is not needed in CL.
3. In PFBT the decision can be on a special "null" value, while in CL the decision is always on a “real” value.
4. Consider finally round $3\phi-2$ of Algorithm 3.6, and our signature-free implementation of WIC, see Figure 3.4 and Algorithm 3.2. Messages of round $\langle r,1\rangle$ basically correspond to the “view-change” messages of PBFT. Messages of round $\langle r,2\rangle$ basically correspond to the “view-change-ack” messages of PBFT. The difference is in round $\langle r,3\rangle$: (i) in PBFT only the coordinator (p_1 in Figure 3.4) sends its message, say $m_{p_1}^{\langle r,1\rangle}$, and piggybacks on it the hashes of the messages p_1 received in round $\langle r,1\rangle$. Let p_2 receive $m_{p_1}^{\langle r,3\rangle}$. If $m_{p_1}^{\langle r,3\rangle}$ piggybacks the hash of some message $m_{p_3}^{\langle r,1\rangle}$ that is not received by p_2 in round $\langle r,2\rangle$, then p_2 sends a request to get $m_{p_3}^{\langle r,2\rangle}$. If p_3 is Byzantine, it might not resend the message. Therefore the coordinator resends the requested message, and the correct processes that has received this message will resend a “view-change-ack” message. Process p_2 can accept the message if it receives b corresponding “view-change-ack” messages. This “pull” strategy avoids sending messages that are not needed. For simplicity, we did not include such an optimization in CL.

3.6 Related work

As already said in Section 3.1, we are not the first one to consider a weaker variant of interactive consistency. Doudou et al. [DGG00b] define an abstraction called weak interactive consistency (WIConsistency) with a definition different from ours. With WIConsistency each correct process proposes its initial value, and processes must eventually decide on the same vector of values which contains at least one value corresponding to the initial value of a correct process. They use this abstraction to derive a state machine replication protocol resilient to authenticated Byzantine faults. However, while to go from WIC to consensus is not trivial, going from WIConsistency to consensus is straightforward [10]. In this sense, WIC is a better building block for consensus.

To the best of our knowledge, there is little work that has proposed consensus algorithms based on abstractions that can be instantiated into (i) algorithms that use signatures and (ii) algorithms that do not use signatures. We are only aware of the work of Skrikanth and Toueg [ST87] related to *authenticated broadcast* (already mentioned in Section 3.1).

On the other hand, several papers consider the idea of automatically translated benign protocols into the protocols that tolerates Byzantine faults. In the context of synchronous systems,

[10]In order to solve consensus using WIConsistency, processes propose their initial values to WIConsistency and then apply deterministic function on the decided vector to get a decision value for consensus.

Neiger, Toueg and Bazzi [NT90, BN01] developed methods to automatically translate protocols tolerant to benign faults into ones tolerant to more severe faults, including Byzantine faults. In the context of asynchronous systems, Bracha [Bra87] and Coan [Coa88] show how to translate round-based algorithms, where each process waits for $n-b$ messages before proceeding to the next round. More recently, Ho et al. [HDVR07, HvRBD08] introduce an abstraction called ordered authenticated reliable broadcast (OARcast) to translate a crash-tolerant algorithm into an algorithm tolerating the same number of Byzantine faults. Compared to [Bra87] and [Coa88], Ho et al. are able to translate more algorithms. In [HDVR07] the authors illustrate their approach by translating the normal case of the Oki and Liskov Viewstamped Replication protocol [OL88] into a protocol that closely resembles the normal case of PBFT. Finally, Clement et al. [CJKR12] have shown a generic translation from a crash-tolerant algorithm into a Byzantine-tolerant one, using non-equivocation and transferable authentication; the translation requires the same number of processes as crash-tolerant algorithm. This work is inspired by several proposals [CMSK07, LDLM09] to add a trusted hardware component to each process, in order to make Byzantine fault tolerant algorithms more efficient in terms of number of processes required to tolerate b Byzantine faults (the number of processes is decreased from $3b+1$ to $2b+1$).

What is common to all these approaches is that transformations permanently restrict the behaviour of Byzantine processes to resemble honest (but potentially crashed) processes. On the other hand, WIC restricts the behavior of Byzantine processes only eventually and only for some rounds (so called WIC rounds). On the other hand, WIC does not restrict the content of the message received from a Byzantine process. Contrary to the translation approaches mentioned above, the message content does not need to correspond to the correct execution of an honest process, i.e., it can be arbitrary.

Orthogonal to our approach, several authors proposed authentication schemes that use symmetric message authentication codes (MACs) in order to achieve properties similar to public-key signature schemes, e.g., [AABC08b, RPS12].

3.7 Conclusion

The chapter has introduced the *weak interactive consistency* (or *WIC*) abstraction, and has shown that WIC allows to unify Byzantine consensus algorithms with and without signatures. This has been illustrated on two seminal Byzantine consensus algorithm, namely on the FaB Paxos algorithm [MA06] and on the PBFT algorithm [CL02]. In both cases this leads to a very concise algorithm.

4 Tolerating Permanent and Transient Value Faults

Transmission faults allow us to reason about permanent and transient value faults in a uniform way. However, all existing solutions to consensus in this model are either in the synchronous system, or require strong conditions for termination, that exclude the case where all messages of a process can be corrupted. In this chapter we overcome this limitation by relying on the weak interactive consistency abstraction introduced in Chapter 3.

The system considered in this chapter is parameterized with α and f. In every round *each process* can receive up to α corrupted messages; eventually rounds are synchronous and the messages sent by at most f processes are corrupted. Before these synchronous rounds, any number of benign faults is tolerated. The chapter presents an algorithm that solves consensus if either $n > 3\alpha + 3f$, or $n > 4$ and $\alpha = f = 1$, or $n > 2\alpha$ and $f = 0$.

Publication: Zarko Milosevic, Martin Hutle and André Schiper. Tolerating Permanent and Transient Value Faults. In *Distributed Computing*, DOI: 10.1007/s00446-013-0199-7.
Brief announcement at the *31st Annual ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing (PODC 2012)*, Madeira, Portugal, July 16 - July 18, 2012.

4.1 Introduction

As explained in Chapter 1, most research on consensus algorithms is considering *component fault models*, where faults are attached to a component that is either a process or a link. Furthermore, in the context of a component fault model, faults are mainly *permanent* (as opposed to *transient* faults): if a process or link commits a fault, the process/link is considered to be faulty during the whole execution. It follows that not all components can be faulty (at most f out of n per run), which is referred to as *static* faults (as opposed to dynamic faults that can affect any component).

Most research on consensus is about tolerating *permanent* and *static* process and/or link faults. While processes and links can be considered faulty, most of the literature considers only process faults. In the context of Byzantine faults, where at most f processes can behave arbitrarily, we can cite the early work of Lamport, Shostak and Pease [PSL80, LSP82] for a synchronous system. Consensus in a partially synchronous system with Byzantine faults is considered in [DLS88, ADGFT06, MA06]. Byzantine variants of Paxos [Lam98] include [CL02, Lam01, ACKM06, Lam11]. Only few authors solve consensus in the synchronous system model where, in addition to Byzantine processes, a small number of links connecting correct processes may be arbitrary faulty during the entire execution of a consensus algorithm [PS85, SAAAA95, SCY98]. However, only a very limited number of links can be faulty.

There are two major problems of a priori blaming some component for the failure [SW89, SW07, CBS09b]. First, it may lead to undesirable consequences if faults are permanent: for example, in the classical Byzantine fault model, where a bounded number of processes can behave arbitrarily (even maliciously), the entire system will be considered faulty even if only one message from each process is received corrupted. Second, when solving consensus, faulty processes are typically not obliged to make a decision or they are allowed to decide differently than correct processes.

Some work in the component fault model has addressed transient and dynamic faults [BSW11]. This paper solves consensus in the hybrid fault model for synchronous systems, where every process is allowed to commit up to f_l^{sa} arbitrary send link failures and experience up to f_l^{ra} arbitrary receive link failures without being considered as arbitrary faulty. Tolerating additional f_s send and f_r receive omissions (i.e., message loss) requires to increase the number of processes by small multiples of f_s and f_r.

Finally, note that when a process q receives a corrupted message from p, it makes no difference for q whether p is faulty and therefore sends a message that was not consistent with the protocol, or the message is corrupted by the link between p and q. Actually, for q these two cases are indistinguishable. Nevertheless, these two cases are not equivalent in the component fault model.

Alternative approach: Transmission fault model. These observations led to the definition of the *transmission fault model* that captures faults without blaming a specific component for the fault [SW89]. The transmission fault model is well-adapted to *dynamic* and *transient* faults.

Consensus under transmission faults in a synchronous system has been considered initially in [SW89]. In [CBS09b], this work combined with ideas from [Gaf98], is extended to non-synchronous systems with only benign transmission faults, leading to the *Heard-Of Model* (HO model). The paper gives several consensus algorithms under benign transmission faults.

In [BCBG$^+$07], the HO model for benign faults is extended to value faults. There, consensus under transmission faults (both benign and value faults) is solved for the first time in a non-synchronous setting. For safety, only the number of corrupted messages is restricted, that is, in each round r of the round based model, every process p receives at most α corrupted messages.[1] However, for liveness, some additional assumptions are necessary, namely *rounds in which some subset of processes does not receive any corrupted messages.*[2] This means that, despite the possibility to handle dynamic and transient value faults in a non-synchronous system, [BCBG+07] cannot tolerate *permanent faults located at some process* p, where all messages from p might be (always) corrupted.

This raises the following question: is it possible to design a consensus algorithm in the general transmission fault model, with non-synchronous assumptions, that does not require such a strong condition for liveness?

In this chapter we give a positive answer to the above question by presenting a consensus algorithm for transmission faults (both benign and value faults) that does not exclude permanent faults. The key insight in achieving this goal is using the *weak interactive consistency* (WIC) abstraction introduced in Chapter 3 in the context of the classical component fault model. It turns out that WIC is also a fundamental building block for solving consensus under transmission value faults.

The consensus algorithm presented in this chapter is inspired by the CL algorithm (Algorithm 3.6 from Section 3.5.3) for the classical Byzantine fault model, which we have adapted to the transmission fault model. The algorithm requires a round in which consistency eventually holds (processes receive the same set of messages). This round is used to bring the system in the univalent configuration, and later rounds are used to “detect” that the system entered a univalent configuration and allows processes to decide. So the key is to achieve WIC in the weak model that we consider in this chapter. This is the most important contribution of the chapter. We show that WIC can be simulated from eventually synchronous rounds in the presence of both static and dynamic value faults. The benefits of this approach are the following:

- First, contrary to most of the related work on transmission faults and on the hybrid fault model (where both processes and links can be arbitrary faulty), which considers the synchronous system model, our consensus algorithm can also be used in systems, where synchrony assumptions hold only eventually.
- Second, contrary to the algorithms in [BCBG+07], our algorithm can also be used in systems with permanent faults located at a process p, where all messages from p might be (always) corrupted.
- Third, by considering the transmission fault model, the algorithm can tolerate dynamic and transient value faults in addition to only permanent and static faults of the component fault model. As we explain in Section 4.7, considering (only) transmission faults allows a variety of interpretations, making it possible to apply our algorithms to a variety

[1] This assumption potentially allows corrupted messages on all links in a run; therefore it models dynamic faults.
[2] This assumption makes sense in the context of transient faults.

of system models: partially synchronous system with Byzantine processes, partially synchronous system with Byzantine processes eventually restricted to ”symmetrical faults” [SWR02], partially synchronous system with Byzantine processes, where, before stabilization time, in every round processes can receive some (bounded) number of corrupted messages from correct processes, etc.

Remark. Note that despite the similarity in title, [AH93] addresses a different topic. The paper investigates the possibility of designing protocols that are both self-stabilizing and fault-tolerant in an asynchronous system. A self-stabilizing distributed algorithm is an algorithm that, when started in an arbitrary state, guarantees to converge to a legitimate state and then forever remains in a legitimate state. Solving one-shot consensus, which is the subject of this chapter, is impossible in the context of self-stabilization, because a process can start in any state, i.e., its first step can be $decide(v)$, where v is an arbitrary value.

In the model considered in this chapter, (transmission) faults do not corrupt the initial configuration (the system starts in a pre-defined state) but may disturb the execution of the protocol. Therefore, the protocols presented in this chapter cannot deal with an arbitrary initial configuration.

Roadmap The rest of the chapter is structured as follows. We describe the transmission fault model we consider in Section 4.2. In Section 4.3 we introduce the communication predicates that we consider in this chapter. Section 4.4 shows how to simulate WIC under weak communication predicates considered in this chapter, while Section 4.5 shows how to solve consensus with WIC in the weak model considered in this chapter. In Section 4.6 we discuss in detail the combination of the consensus algorithm and the WIC simulation. In Section 4.7 we argue that Byzantine faults and permanent value faults located at a process are indistinguishable, and thus our algorithms also work (but not only) in a partial synchronous model with Byzantine processes. We conclude the chapter in Section 4.8.

4.2 Model

We use a slightly extended version of the round-based model of [BCBG$^+$07]. In this model, we reason about faults only as *transmission faults*, without looking for a “culprit” for the fault [BCBG$^+$07]. Therefore there are no “faulty” processes and no state corruption in our model, but messages can be arbitrarily corrupted (or lost) before reception. Nevertheless, as we explain in Section 4.7, the model can be used to reason about classical Byzantine faults.

Computations in this model are structured in rounds, which are communication-closed layers in the sense that any message sent in a round can be received only in that round. As messages can be lost, this does not imply that the system is synchronous. An algorithm $\mathscr{A}$ is specified by sending function S_p^r and transition function T_p^r for each round r and process p. We now give a formal definition of the round-based model considered, and introduce the notions of (i) the *heard-of set* $HO(p,r)$, which captures synchrony and benign faults, (ii) the *safe heard-of set* $SHO(p,r)$, which handles corruptions, i.e., captures communication safety properties, and (iii) *consistency* $CONS(r)$, which is true in round r, if all processes receive the same set of messages at round r.

4.2.1 Heard-Of Sets and Consistent Rounds

Let Π be a finite non-empty set of cardinality n, and let $\vec{M}$ be a set of messages (optionally including a *null* placeholder indicating the empty message). To each p in Π, we associate a *process*, which consists of the following components: A set of states denoted by $states_p$, a subset $init_p$ of initial states, and for each positive integer r called *round number*, a message-sending function S_p^r mapping $states_p$ to a unique message from $\vec{M}$, and a state-transition function T_p^r mapping $states_p$ and partial vectors (indexed by Π) of elements of $\vec{M}$ to $states_p$. The collection of processes is called an *algorithm on* Π.

In each round r, a process p:

1. applies S_p^r to the current state and sends the message returned to each process,[3]
2. determines the partial vector $\vec{\mu}_p^r$, formed by the messages that p receives at round r, and
3. applies T_p^r to its current state and $\vec{\mu}_p^r$.

The partial vector $\vec{\mu}_p^r$ is called the *reception vector of p at round r*.

Computation evolves in an infinite sequence of rounds. For each process p and each round r, we introduce two subsets of Π. The first subset is the *heard-of* set, denoted $HO(p,r)$, which is the support of $\vec{\mu}_p^r$, i.e.,

$$HO(p,r) = \left\{q \in \Pi : \vec{\mu}_p^r[q] \text{ is defined}\right\}.$$

A process q is in the set $HO(p,r)$ if p receives a message from process p in round r. Note that the message received may be corrupted. The second subset is the *safe heard-of* set, denoted $SHO(p,r)$, and defined by

$$SHO(p,r) = \left\{q \in \Pi : \vec{\mu}_p^r[q] = S_q^r(s_q)\right\},$$

where s_q is q's state at the beginning of round r. A process q is in the set $SHO(p,r)$ if the message received by p is not corrupted. In addition, for each round r, we define the *consistency* flag, denoted $CONS(r)$, which is true if all processes receive the same set of messages in round

[3]W.l.o.g., the same message is sent to all. Because of transmission faults, this does not prevent two processes p and q from receiving different messages from some process s.

r, i.e.,

$$CONS(r) = \forall p, q \in \Pi^2 : \vec{\mu}_p^r = \vec{\mu}_q^r.$$

From the sets $HO(p,r)$ and $SHO(p,r)$, we form the *altered heard-of set* denoted $AHO(p,r)$ as follows:

$$AHO(p,r) = HO(p,r) \setminus SHO(p,r).$$

For any round r, and for any set of rounds Φ, we further define the *safe kernel* of r resp. Φ:

$$SK(r) = \bigcap_{p \in \Pi} SHO(p,r) \qquad SK(\Phi) = \bigcap_{r \in \Phi} SK(r)$$

The safe kernel consists of all processes whose messages were received correctly by all processes. We use also $SK = SK(\mathbb{N})$. Similarly, the *altered span* (of round r) denotes the set of processes from which at least one process received a corrupted message (at round r):

$$AS(r) = \bigcup_{p \in \Pi} AHO(p,r) \qquad AS = \bigcup_{r>0} AS(r)$$

We also extend the notion of $CONS$ in a natural way to a set Φ of rounds, i.e., $CONS(\Phi) = \bigwedge_{r \in \Phi} CONS(r)$.

4.2.2 HO Machines

A *heard-of machine* for a set of processes Π is a pair $(\mathscr{A}, \mathscr{P})$, where $\mathscr{A}$ is an algorithm on Π, and $\mathscr{P}$ is a *communication predicate*, i.e., a predicate over the collection

$$(\big(HO(p,r), SHO(p,r)\big)_{p \in \Pi}, CONS(r))_{r>0}$$

A *run* of an HO machine M is entirely determined by the initial configuration (i.e., the collection of process initial states), and the collection of the reception vectors $\left(\vec{\mu}_p^r\right)_{p \in \Pi,\ r>0}$.

4.2.3 Simulation of communication predicates

We have already introduced the notion of predicate simulation in Section 3.4. In this chapter we will need to simulate[4] communication predicates $\mathscr{P}'$ using some HO machine $M = (\mathscr{A}, \mathscr{P})$. Intuitively, in such a simulation, several rounds of M will be used to simulate one round in which predicate $\mathscr{P}'$ holds. If the run of M consists of k rounds, then algorithm $\mathscr{A}$ is a k round simulation of $\mathscr{P}'$ from $\mathscr{P}$.

In the following we formally define the predicate simulation in the context of the HO model.

[4]The notion of a simulation differs from the notion of a *translation* of the HO model for benign faults. A translation establishes a relation purely based on connectivity, while with value faults, also some computation is involved.

Let k be any positive integer, and let $\mathscr{A}$ be an algorithm that maintains a variable $m_p \in \vec{M}$ and $Msg_p \in \vec{M}^n$ at every process p. We call *macro-round* ρ the sequence of the k consecutive round $k(\rho-1)+1,\ldots,k\rho$. The variable m_p is an input variable that can be set externally in every macro-round.[5]

The value of m_p at the beginning of macro-round ρ is denoted $m_p^{(\rho)}$, and the value of Msg_p at the end of macro-round ρ is denoted $Msg_p^{(\rho)}$.

For the macro-round ρ, we define in analogy to the definitions of Section 4.2.1:

$$HO(p,\rho) = \left\{q \in \Pi : Msg_p^{(\rho)}[q] \text{ is defined}\right\}$$

$$SHO(p,\rho) = \left\{q \in \Pi : Msg_p^{(\rho)}[q] = m_q^{(\rho)}\right\}$$

$$CONS(\rho) = (\forall p,q \in \Pi^2 : Msg_p^{(\rho)} = Msg_q^{(\rho)})$$

We say that the HO machine $M = (\mathscr{A},\mathscr{P})$ simulates the communication predicate $\mathscr{P}'$ in k rounds if for any run of M, the collection $(HO(p,\rho), SHO(p,\rho))_{p\in\Pi}, CONS(\rho))_{\rho>0}$ satisfies predicate $\mathscr{P}'$.

Given a simulation $\mathscr{A}$ of $\mathscr{P}'$ from $\mathscr{P}$, any problem that can be solved with $\mathscr{P}'$ by algorithm $\mathscr{A}'$ can be solved with $\mathscr{P}$ instead by simply simulating rounds of the algorithm $\mathscr{A}'$ using algorithm $\mathscr{A}$. In such a composed algorithm, the input variable $m_p^{(\rho)}$ of algorithm $\mathscr{A}$ is set at each macro-round ρ to the value returned by the sending function of $\mathscr{A}'$, and the transition function of $\mathscr{A}'$ is applied to the output $Msg_p^{(\rho)}$ of algorithm $\mathscr{A}$.

4.2.4 Consensus

Formally, an HO machine $(\mathscr{A},\mathscr{P})$ solves consensus, if any run for which $\mathscr{P}$ holds, satisfies Agreement, Termination and Integrity (see Section 2.1). To make this definition non-trivial, we assume that the set of *HO* and *SHO* collections for which $\mathscr{P}$ holds is non-empty.

4.3 Communication predicates

In this section we introduce the communication predicates that will be used in the chapter. As already mentioned, we reason about faults only as transmission faults. This allows us to deal with both permanent and transient faults, but also with static and dynamic faults.

[5]The sending function in a simulation algorithm is thus a function that maps $states_p$ and the input from $\vec{M}$ to a unique message from $\vec{M}$; while the state-transition function T_p^r is a function that maps $states_p$, the input from $\vec{M}$, and a partial vector (indexed by Π) of elements of $\vec{M}$ to $states_p$.

4.3.1 Predicates that capture static and dynamic value faults

A *dynamic* fault is a fault that can affect any link in the system — as opposed to *static* faults that affect the links of at most f out of n processes per run [BCBG+07]. We start with static faults:

$$\mathscr{P}^{f}_{perm} :: |AS| \leq f \tag{4.1}$$

with $f \in N$ and $N = \{0,\ldots,n\}$. $\mathscr{P}_{perm}$ is the name of the predicate, and f is a free parameter. $\mathscr{P}^{f}_{perm}$ is a safety predicate that models static faults, where corrupted messages are received only from a set of f processes. In Section 4.7 we will argue that such an assumption corresponds to a system with at most f Byzantine processes.

For algorithms in this chapter we will consider the weaker safety predicate $\mathscr{P}^{f}_{dyn}$ ($\forall f \in N$, $\mathscr{P}^{f}_{perm}$ implies $\mathscr{P}^{f}_{dyn}$) that restricts the number of corrupted messages only per round and per process:

$$\mathscr{P}^{f}_{dyn} :: \forall r > 0, \forall p \in \Pi : |AHO(p,r)| \leq f$$

with $f \in N$ and $0 \leq f \leq n$. Predicate $\mathscr{P}^{f}_{dyn}$ potentially allows corrupted messages on all links in a run, it therefore models dynamic value faults.

4.3.2 Predicates that restrict asynchrony of communication and dynamism of faults

Predicates $\mathscr{P}_{perm}$ and $\mathscr{P}_{dyn}$ only restrict the number of value faults; however, it does not tell us anything about liveness of communication. From [FLP85] we know that we cannot solve consensus in an asynchronous system if all messages sent by one process may be lost. On the other hand, Santoro and Widmayer [SW89] showed that consensus is impossible to solve in a synchronous system if, at each time unit, there is one process whose messages may be lost. Therefore, in order to solve consensus we need to restrict asynchrony of communication and dynamism of faults.

A synchronous system could be modeled as follows:

$$\mathscr{P}^{f}_{SK} :: |SK| \geq n - f \tag{4.2}$$

$\mathscr{P}^{f}_{SK}$ requires that there is a set of processes (safe kernel) of size $n - f$ whose messages are correctly received in every round. From

$$\forall f \in N,\ \mathscr{P}^{f}_{SK} \Rightarrow \mathscr{P}^{f}_{perm}$$

it follows that $\mathscr{P}_{SK}^{f}$ implies static faults only. However, we want to study consensus with dynamic faults. We consider therefore the following predicate:

$$\mathscr{P}_{\diamond SK}^{f,k} :: \forall r > 0\ \exists r_o > r,\ \Phi = \{r_0, \dots, r_0 + k - 1\} : |SK(\Phi)| \geq n - f \tag{4.3}$$

with $f \in N$ and $k > 0$. This predicate (repeatedly) requires a safe kernel of size $n - f$ only eventually and only for k rounds. It also restrict the dynamism of value faults during these k round; i.e., corrupted messages can only be received from at most f processes.

In the chapter we will consider $\mathscr{P}_{\diamond SK}$ always in conjunction, either with $\mathscr{P}_{perm}$ or $\mathscr{P}_{dyn}$. When we assume $\mathscr{P}_{\diamond SK}$ with $\mathscr{P}_{perm}$, i.e., $\mathscr{P}_{\diamond SK}^{f,k} \wedge \mathscr{P}_{perm}^{f}$, transmission value faults are static (benign transmission faults are not restricted, so they can be dynamic). On the other hand, when we assume $\mathscr{P}_{\diamond SK}$ with $\mathscr{P}_{dyn}$, i.e., $\mathscr{P}_{\diamond SK}^{f,k} \wedge \mathscr{P}_{dyn}^{\alpha}$ with $f \leq \alpha$, transmission value faults are no more static: $\mathscr{P}_{\diamond SK}^{f,k}$ alone does not imply $\mathscr{P}_{perm}^{f'}$ for any $f' < n$.

The implementation of the predicate $\mathscr{P}_{\diamond SK}$ in a partially synchronous system (in conjunction, either with $\mathscr{P}_{perm}$ or $\mathscr{P}_{dyn}$) is not discussed in this chapter. The reader is referred to [DLS88, BHSS12].

4.3.3 Permanent versus Transient Faults

Both predicates, $\mathscr{P}_{perm} \wedge \mathscr{P}_{\diamond SK}$ and $\mathscr{P}_{dyn} \wedge \mathscr{P}_{\diamond SK}$ allow *permanent faults*. Consider for example a run and a process p, where every process receives a corrupted message from p in every round:

$$\forall q \in \Pi, r > 0 : \ p \notin SHO(q, r)$$

and all other messages are received correctly. Such a run is included in the set of runs given by $\mathscr{P}_{perm} \wedge \mathscr{P}_{\diamond SK}$ and $\mathscr{P}_{dyn} \wedge \mathscr{P}_{\diamond SK}$, and thus the algorithms given later in the chapter can solve consensus in such a run. More precisely, $\mathscr{P}_{perm}^{f} \wedge \mathscr{P}_{\diamond SK}^{f}$ and $\mathscr{P}_{dyn}^{f} \wedge \mathscr{P}_{\diamond SK}^{f}$ permit the existence of up to f such processes. As pointed out in Section 4.7, this allows our algorithms to solve consensus also, e.g., in classical models with Byzantine faults, and addresses the question raised in the introduction. Indeed, this contrasts with [BCBG+07], where, although also $\mathscr{P}_{dyn}$ is considered (named $\mathscr{P}_{\alpha}$ there), eventually there has to be a round, where a sufficiently large subset of processes do not receive any corrupted messages. There, (most) faults have to be *transient*.

4.3.4 Weak Interactive Consistency

Informally speaking, weak interactive consistency combines the requirement of a consistent round ($CONS(r)$ in our model) with some requirements on liveness and safety of communication. As explained in Section 3.3, it can be seen as a *weaker* version of *interactive consistency* [PSL80]. In a component fault model, an algorithm that solves interactive consistency allows correct processes to agree on a vector, where at least $n-f$ entries correspond to the initial values of the corresponding correct processes (f is the maximum number of faulty processes).

Interactive consistency, when seen as a communication primitive, can be captured by the following predicate:

$$\mathscr{P}_{IC}^{f} :: \ |SK| \geq n-f \wedge \forall r > 0 : CONS(r)$$

When we express the result of [PSL80] in our model, their algorithm allows a $f+1$ round simulation of $\mathscr{P}_{IC}^{f}$ from $\mathscr{P}_{SK}^{f}$ if $n > 3f$. Note that $\forall f \in N$, $\mathscr{P}_{IC}^{f} \Rightarrow \mathscr{P}_{perm}^{f}$.

We define the weak interactive consistency in this chapter with the following predicate:

$$\mathscr{P}_{\diamond cons}^{f} :: \ \forall r > 0 \ \exists r_o > r : \ |SK(r_0)| \geq n-f \wedge CONS(r_0)$$

This predicate requires that there is always eventually a consistent round with a safe kernel of size $n-f$. In contrast to $\mathscr{P}_{SK}^{f}$ and $\mathscr{P}_{IC}$, this predicate requires these safe kernels only eventually and then only for a single round. Also faults are no more static: $\mathscr{P}_{\diamond cons}^{f}$ alone does not imply $\mathscr{P}_{perm}^{f'}$ for any $f' < n$. Note that $\mathscr{P}_{\diamond cons}^{f}$ is a stronger predicate than $\mathscr{P}_{\diamond SK}^{f,1}$: although both predicates require a safe kernel of size $n-f$ and both restrict the dynamism of value faults for a single round, $\mathscr{P}_{\diamond cons}^{f}$ in addition requires that consistency holds during this round, i.e., for any two processes p and q we have $\vec{\mu}_p = \vec{\mu}_q$.

However, $\mathscr{P}_{\diamond cons}$ can be simulated from $\mathscr{P}_{\diamond SK}$. In the next section, we give such simulation, and then establish the link to solving consensus.

4.4 Simulating weak interactive consistency $\mathscr{P}_{\diamond cons}$ from eventually safe kernels $\mathscr{P}_{\diamond SK}$

In this section we give a simulation of $\mathscr{P}_{\diamond cons}$ from $\mathscr{P}_{\diamond SK}$ in the presence of dynamic (and static) value faults ($\mathscr{P}_{dyn}$). Then we introduce a *generic predicate* $\mathscr{P}_{\diamond cons \oplus SK}$ that can be simulated from $\mathscr{P}_{\diamond SK}$. The predicate $\mathscr{P}_{\diamond cons \oplus SK}$, in conjunction with $\mathscr{P}_{dyn}$ or $\mathscr{P}_{perm}$, is later used in Section 4.5 to solve consensus.

In Section 3.4.2 we have shown a signature-free WIC simulation from eventually synchronous rounds in the context of classical Byzantine faults (Algorithm 3.2). This result can be expressed in the framework of this chapter by the following theorem:

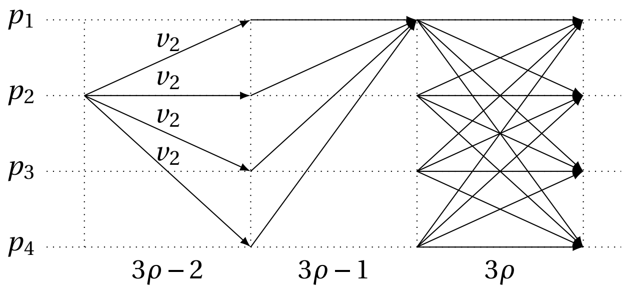


Figure 4.1: The communication scheme of the simulation of $\mathscr{P}^{f}_{\diamond cons} \wedge \mathscr{P}^{f}_{perm}$ from $\mathscr{P}^{f,3(f+1)}_{\diamond SK} \wedge \mathscr{P}^{f}_{perm}$ (Algorithm 3.2 from Section 3.4.2) from the point of view of v_2 sent by p_2 (p_1 is the coordinator, $n = 4$, $f = 1$).

Theorem 4.1. *If $n > 3f$ and $\alpha = f$, Algorithm 3.2 (from Section 3.4.2) is a simulation of $\mathscr{P}^{f}_{\diamond cons} \wedge \mathscr{P}^{f}_{perm}$ from $\mathscr{P}^{f,3(f+1)}_{\diamond SK} \wedge \mathscr{P}^{f}_{perm}$.*

Figure 4.1 gives a communication pattern of the Algorithm 3.2 from the point of view of messages sent by process p_2.

In this section we show a simulation of $\mathscr{P}_{\diamond cons}$ from $\mathscr{P}_{\diamond SK}$ and the weaker predicate $\mathscr{P}_{dyn}$ that (partially) preserves $\mathscr{P}_{dyn}$. More precisely, we show a simulation from $\mathscr{P}^{f}_{\diamond SK} \wedge \mathscr{P}^{\alpha}_{dyn}$ into $\mathscr{P}^{f}_{\diamond cons} \wedge \mathscr{P}^{\beta}_{dyn}$ with $\beta \geq \alpha$: the simulation may only partially preserve $\mathscr{P}^{\alpha}_{dyn}$ in the sense that the number of corruptions in the simulated rounds may increase from α to $\beta \geq \alpha$, depending on n. As already mentioned in Section 4.2.3, a simulation is an algorithm that maintains at each process p two variables: an input variable m_p that is set at the beginning of every macro-round ρ (Algorithm 4.1, line 6), and an output variable Msg_p whose value is considered at the end of every macro-round ρ (Algorithm 4.1, lines 31 and 33). The special value $\perp$ represents the case when a (reception) vector does not contain a message from the respective process.

The simulation requires four rounds, as shown by Algorithm 4.1. As we can see, β is not a parameter of the algorithm. Fixing β leads to some requirement on n. More precisely, given f, $\alpha \geq f$, $\beta \geq \alpha$, Algorithm 4.1 requires $n > \frac{(\beta+1)(\alpha+f)}{\beta-\alpha+1}$. Similarly to Algorithm 3.2 in Section 3.4.2, it is coordinator-based.

The communication pattern of Algorithm 4.1 is very similar to Algorithm 3.2 with the addition of one "all-to-all" round (see Figure 4.2, to be compared with Figure 4.1). We explain Algorithm 4.1 from the point of view of the message sent by process p_2. In round $4\rho - 3$, process p_2 sends message v_2 to all.[6] In round $4\rho - 2$, all processes send to all the value received from p_2, and then compare the value v_2 received from p_2 in round $4\rho - 3$ with the value indirectly received from the other processes in round $4\rho - 2$. If at least $n - f$ values v_2 have been received

[6]In the description of Algorithm 4.1, in case of messages that contain a vector of messages, we focus only on those elements of the vectors that are related to the message sent by process p_2.

Algorithm 4.1 Simulation of $\mathscr{P}^{f}_{\diamond cons} \wedge \mathscr{P}^{\beta}_{dyn}$ from $\mathscr{P}^{f,4(f+1)}_{\diamond SK} \wedge \mathscr{P}^{\alpha}_{dyn}$

```
1:  Initialization:
2:    $Msg_p \leftarrow (\bot, \dots, \bot)$                    /* $Msg_p$ is the output variable */
3:    $coord_p = \rho \bmod n + 1$

4:  Round $r = 4\rho - 3$:
5:    $S_p^r$:
6:      send $m_p$ to all                                  /* $m_p$ is the input variable */
7:    $T_p^r$:
8:      $first_p \leftarrow \vec{\mu}_p^r$
9:      $conf_p \leftarrow (\bot, \dots, \bot)$

10: Round $r = 4\rho - 2$:
11:   $S_p^r$:
12:     send $first_p$ to all
13:   $T_p^r$:
14:     for all $q \in \Pi$ do
15:       if $\left|\left\{i \in \Pi : \vec{\mu}_p^r[i][q] = first_p[q]\right\}\right| \geq n - f$ then
16:         $conf_p[q] \leftarrow first_p[q]$

17: Round $r = 4\rho - 1$:
18:   $S_p^r$:
19:     send $conf_p$ to all
20:   $T_p^r$:
21:     if $p = coord_p$ then
22:       for all $q \in \Pi$ do
23:         if $\left|\left\{i \in \Pi : \vec{\mu}_p^r[i][q] = conf_p[q]\right\}\right| < \alpha + f + 1$ then
24:           $conf_p[q] \leftarrow \bot$

25: Round $r = 4\rho$:
26:   $S_p^r$:
27:     send $conf_p$ to all
28:   $T_p^r$:
29:     for all $q \in \Pi$ do
30:       if $\left|\left\{i \in \Pi : \vec{\mu}_p^r[i][q] = \vec{\mu}_p^r[coord_p][q]\right\}\right| \geq \alpha + 1$ then
31:         $Msg_p[q] \leftarrow \vec{\mu}_p^r[coord_p][q]$
32:       else
33:         $Msg_p[q] \leftarrow \bot$
```

by process p, then p keeps v_2 as the message received from p_2. Otherwise, the message received from p_2 is $\bot$ (line 9). As explained later, rounds $4\rho - 3$ and $4\rho - 2$ filter the values for rounds $4\rho - 1$ and 4ρ in order to ensure $\mathscr{P}^{\beta}_{dyn}$ from $\mathscr{P}^{\alpha}_{dyn}$. Rounds $4\rho - 1$ and 4ρ are very similar to rounds $3\rho - 1$ and 3ρ in Algorithm 3.2.

Algorithm 4.1 relies on a coordinator for ensuring $\mathscr{P}^{f}_{\diamond cons}$: all processes assign to Msg_p the value received from the coordinator in round 4ρ (see line 31). This is achieved during a macro-round in which the size of the safe kernel is at least $n - f$, with the coordinator in the safe kernel. Since consistency is ensured under the same conditions as with Algorithm 3.2, we use exactly the same mechanism in Algorithm 4.1.

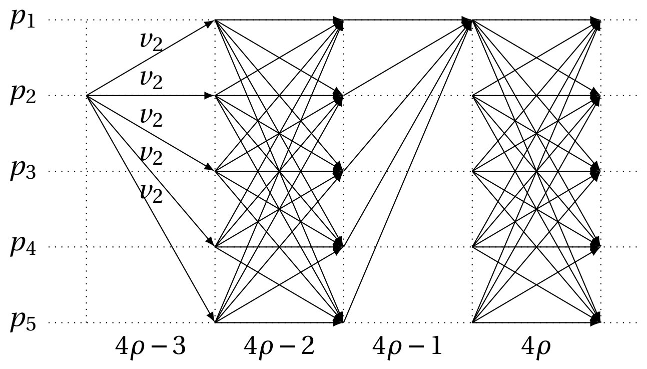


Figure 4.2: Algorithm 4.1 from the point of view of v_2 sent by p_2; p_1 is the coordinator, $n = 5$, $f = 1$.

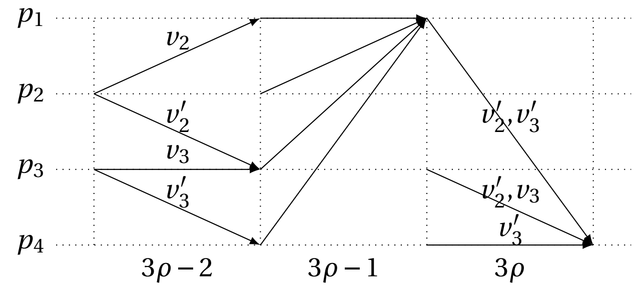


Figure 4.3: Algorithm 3.2 does not preserves $\mathscr{P}^{\alpha}_{dyn}$; from the point of view of p_2 and p_3, that sends correspondingly v_2 and v_3 in round $3\rho-2$, and reception of process p_4 of messages sent by p_2 and p_3. $n = 4$, $f = \alpha = \beta = 1$ and p_1 is coordinator. Message received by p_4 from coordinator in round 3ρ is corrupted; other messages are correctly received. Absence of arrows represents message loss.

The additional complexity of Algorithm 4.1 comes from the part responsible for ensuring $\mathscr{P}^{\beta}_{dyn}$. We start by explaining on Figure 4.3 why Algorithm 3.2 does not preserve $\mathscr{P}^{\alpha}_{dyn}$ for the simplest case $f = \alpha = 1$, $n = 4$. According to $\mathscr{P}^{1}_{dyn}$, every process can receive at most one corrupted message per round. In round $3\rho-2$, process p_3 receives the corrupted message v'_2 from p_2, and p_4 receives the corrupted value v'_3 from p_3. These values are sent to the coordinator p_1 in round $3\rho-1$. Finally, in round 3ρ, process p_4 receives v'_2, v'_3 from p_1, v'_2, v_3 from p_3, and v'_3 from itself. Since there are $f+1$ values equal to those sent by coordinator, p_4 considers v'_2, respect. v'_3, as messages received from p_2, respect. p_3, in macro-round ρ, violating $\mathscr{P}^{1}_{dyn}$. The problem comes from the fact that dynamic faults have a cumulative effect, i.e., messages that are corrupted in round $3\rho-2$ add to corrupted messages from round 3ρ.

We now explain why the addition of round $4\rho-2$ allows us to cope with this issue. Informally speaking, the role of round $4\rho-2$ in Algorithm 4.1 is to transform dynamic faults into some maximum number of static faults, i.e., into some maximum number of faults localized at some

first_{p_1}	×			
first_{p_2}		×		
first_{p_3}			×	
first_{p_4}				×

first_{p_1}	×			
first_{p_2}	×			
first_{p_3}		×		
first_{p_4}		×		

Figure 4.4: After round $4\rho-3$: Two examples (left and right) of corrupted values received (represented by ×).

fixed set of processes. Consider rounds $4\rho-3$ and $4\rho-2$, with $n=4$, $\alpha=f=1$. In round $4\rho-3$, predicate $\mathcal{P}^{\alpha}_{dyn}$ ensures that, in total, at most $n\cdot\alpha=4$ corrupted values are received. In other words, among the vectors first_{p_1} to first_{p_4} received (line 8), at most $n\cdot\alpha=4$ elements can be corrupted (see Figure 4.4, where × represents possible corrupted values). In round $4\rho-2$, each process p_i sends vector first_{p_i} to all processes. Consider the reception of these four vectors by some process p_j. Since $\alpha=1$, one of these vectors can be received corrupted at p_j. Figure 4.5 shows four examples, two starting from Figure 4.4 left, two starting from Figure 4.4 right.

To understand which value p adopts from q (lines 15 and 16) we need to look at column q in Figure 4.5. From line 16, p adopts a corrupted value from q only if column q contains at least $n-f=3$ corrupted values. In Figure 4.5 (a), no column satisfies this condition, i.e., p adopts no corrupted value. In Figure 4.5 (b), columns 2 and 1 satisfy this condition, i.e., corrupted values can be adopted from p_2 or p_1. It is easy to see that in the case $n=4$, $f=\alpha=1$, corrupted values can be adopted from at most one process. In other words, rounds $4\rho-3$ and $4\rho-2$ has transformed $\alpha=1$ dynamic fault into at most $\beta=1$ static faults. In the case $n=5$, $f=\alpha=2$, rounds $4\rho-3$ and $4\rho-2$ transform $\alpha=2$ dynamic fault into at most $\beta=3$ static fault.

Transforming α dynamic faults into $\beta\geq\alpha$ static faults allows us to rely on the same mechanism as in Algorithm 3.2 for the last two rounds of the simulation. Note that in rounds $4\rho-1$ and 4ρ of Algorithm 4.1 we have dynamic faults, while in rounds $3\rho-1$ and 3ρ of Algorithm 3.2 faults were static. Nevertheless the same mechanisms can be used in both cases.

Theorem 4.2. *If $n>\frac{(\beta+1)(\alpha+f)}{\beta-\alpha+1}$, $n>\alpha+f$, $\alpha\geq f$, and $\beta\geq\alpha$, then Algorithm 4.1 simulates $\mathcal{P}^{f}_{\diamond cons}\wedge\mathcal{P}^{\beta}_{corr}$ from $\mathcal{P}^{f,4(f+1)}_{\diamond SK}\wedge\mathcal{P}^{\alpha}_{dyn}$.*

The theorem follows directly from Lemmas 4.1 and 4.2: the first lemma considers $\mathcal{P}^{\beta}_{corr}$ and $\mathcal{P}^{\alpha}_{dyn}$, the second $\mathcal{P}^{f}_{\diamond cons}$ and $\mathcal{P}^{f,4(f+1)}_{\diamond SK}$.

Lemma 4.1. *If $n>\frac{(\beta+1)(\alpha+f)}{\beta-\alpha+1}$ and $\beta\geq\alpha$, then Algorithm 4.1 simulates $\mathcal{P}^{\beta}_{corr}$ from $\mathcal{P}^{\alpha}_{dyn}$.*

Proof. We need to show that for every macro-round ρ, and every process p, we have $|AHO(p,\rho)|\leq\beta$, i.e., at most β messages are corrupted.

$\vec{\mu}_{p_j}[p_1]$	×	×	×	×
$\vec{\mu}_{p_j}[p_2]$		×		
$\vec{\mu}_{p_j}[p_3]$			×	
$\vec{\mu}_{p_j}[p_4]$				×

$\vec{\mu}_{p_j}[p_1]$	×			
$\vec{\mu}_{p_j}[p_2]$		×		
$\vec{\mu}_{p_j}[p_3]$			×	
$\vec{\mu}_{p_j}[p_4]$	×	×	×	×

(a) Starting from Figure 4.4 left

$\vec{\mu}_{p_j}[p_1]$	×	×	×	×
$\vec{\mu}_{p_j}[p_2]$	×			
$\vec{\mu}_{p_j}[p_3]$		×		
$\vec{\mu}_{p_j}[p_4]$		×		

$\vec{\mu}_{p_j}[p_1]$	×			
$\vec{\mu}_{p_j}[p_2]$	×			
$\vec{\mu}_{p_j}[p_3]$		×		
$\vec{\mu}_{p_j}[p_4]$	×	×	×	×

(b) Starting from Figure 4.4 right

Figure 4.5: After round $4\rho-2$: (a) Two examples of vectors received by some p_j starting from Fig. 4.4 left; (b) Two examples of vectors received by some p_j starting from Fig. 4.4 right (corrupted values are represented by $\times$).

Assume by contradiction that there is a process p so that $|AHO(p,\rho)| > \beta$. That is, we have $|S| \geq \beta + 1$ for

$$S = \left\{s \in \Pi : Msg_p[s] \neq m_s \text{ and } Msg_p[s] \neq \bot\right\}$$

For all $s \in S$, let m'_s denote $Msg_p[s]$. The output $Msg_p[s]$ is set at line 31. Because of line 30, this implies that

$$\forall s \in S : \left|\left\{i \in \Pi : \vec{\mu}_p^{4\rho}[i][s] = m'_s\right\}\right| \geq \alpha + 1.$$

Because of $|AHO(p,4\rho)| \leq \alpha$, at the end of round $4\rho-1$ we have

$$\forall s \in S, \exists i_s \in \Pi : conf_{i_s}[s] = m'_s.$$

Since in round $4\rho-1$ the elements of *conf* can only be set to $\bot$, the same condition needs to holds also at the end of round $4\rho-2$. Because of line 15, this implies

$$\forall s \in S, \exists i_s \in \Pi, \exists Q_s \subseteq \Pi, |Q_s| \geq n-f, \forall q \in Q_s :$$

$$\vec{\mu}_{i_s}^{4\rho-2}[q][s] = m'_s.$$

Because of $|AHO(p,2)| \leq \alpha$, at the end of round $4\rho-3$ we have

$$\forall s \in S, \exists Q'_s \subseteq \Pi, |Q'_s| \geq n-f-\alpha : \forall q \in Q'_s :$$

$$first_q[s] = m'_s.$$

Note that $first_q = \vec{\mu}_q^{4\rho-3}$. The number of tuples (q,s) such that $\vec{\mu}_q^{4\rho-3}[s] = m'_s$ is thus at least

$(\beta+1)(n-f-\alpha)$. From this it follows that there is at least one process q_0 where the number of corrupted messages in the first round is

$$\left\lceil \frac{(\beta+1)(n-f-\alpha)}{n} \right\rceil = \alpha + \left\lceil \frac{(\beta+1)(n-f-\alpha)-n\alpha}{n} \right\rceil > \alpha$$

where the last inequation follows from $n > \frac{(\beta+1)(\alpha+f)}{\beta-\alpha+1}$ and $\beta \geq \alpha$, which ensures $(\beta+1)(n-f-\alpha)-n\alpha > 0$. Therefore $AHO(q_0, 4\rho-3) > \alpha$, which contradicts the assumption $AHO(q_0, 4\rho-3) \leq \alpha$. □

Lemma 4.2. *If $n > \alpha + f$ and $\alpha \geq f$, then Algorithm 4.1 simulates $\mathscr{P}^{f}_{\diamond cons}$ from $\mathscr{P}^{f,4(f+1)}_{\diamond SK}$.*

Proof. Let ρ denote a macro-round, let $\Phi = \{4\rho-3, \ldots, 4\rho\}$ be the set of rounds of ρ, and let $c_0 = \rho \bmod n+1$ be the coordinator of ρ such that

$$c_0 \in SK(\Phi) \ \wedge |SK(\Phi)| \geq n-f.$$

Such a macro-round exists, because *(i)* $\mathscr{P}^{f,4(f+1)}_{\diamond SK}$ holds and *(ii)* the coordinator is chosen using a rotating coordinator scheme (the coordinator of macro-round ρ is process $\rho \bmod n+1$). We show that with Algorithm 4.1 (i) $CONS(\rho)$ and (ii) $|SK(\rho)| \geq n-f$.

(i) Assume by contradiction that for two processes p and q, $Msg_p^{(\rho)}$ and $Msg_q^{(\rho)}$ differ by the message of process $s \in \Pi$, that is $Msg_p^{(\rho)}[s] \neq Msg_q^{(\rho)}[s]$. By round 4ρ, every process adopts the value of c_0 or sets $Msg^{(\rho)}[s]$ to $\perp$; when $c_0 \in SK(\Phi)$ it follows that $Msg_p^{(\rho)}[s]$ or $Msg_q^{(\rho)}[s]$ is $\perp$. W.l.o.g. assume that $Msg_p^{(\rho)}[s] = v$ and $Msg_q^{(\rho)}[s] = \perp$. For rounds $r \in [4\rho-1, 4\rho]$, let $R_p^r(v,s) := \left\{q \in \Pi : \vec{\mu}_p^r[q][s] = v\right\}$ represent the set of processes from which p receives v at position s. Similarly, for rounds $r \in [4\rho-1, 4\rho]$, let

$$Q^r(v,s) := \left\{q \in \Pi : S_q^r(s_q^r)[s] = v\right\}$$

represent the set of processes that sent v at position s.

By line 30, if $Msg_p^{(\rho)}[s] = v$, then $|R_p^{4\rho}(v,s)| \geq \alpha+1$, and $c_0 \in R_p^{4\rho}(v,s)$. Since $c_0 \in SK(\Phi)$, we have $c_0 \in Q^{4\rho}(v,s)$ and thus, by line 23, $|R_{c_0}^{4\rho-1}(v,s)| \geq \alpha+f+1$. From this and $|SK(\Phi)| \geq n-f$, we have $|R_{c_0}^{4\rho-1}(v,s) \cap SK(\Phi)| \geq \alpha+1$. Therefore, at least $\alpha+1$ processes p' in $SK(\Phi)$, including c_0, have $conf_{p'}[s] = v$. It follows that $|R_q^{4\rho}(v,s)| \geq \alpha+1$, and $c_0 \in R_q^{4\rho}(v,s)$. This contradicts the assumption that the condition in line 30 is false for process q.

(ii) For every processes $p \in \Pi$ and $q \in SK(\Phi)$, by definition we have $first_p[q] = m_q$ at the end of round $4\rho-3$. In round $4\rho-2$, for every process $s \in SK(\Phi)$, $first_s$ is received. Therefore, by line 15 since $|SK(\Phi)| \geq n-f$, at every process $p \in \Pi$ we have $conf_p[q] = m_q$, for all $q \in SK(\Phi)$ (*). In round $4\rho-1$, c_0 receives $conf_{q'}[q] = m_q$ from every process $q' \in SK(\Phi)$, and thus there is no $q \in SK(\Phi)$ s.t. c_0 sets $conf_{c_0}[q]$ to $\perp$ (**). In round 4ρ, since $c_0 \in SK(\Phi)$, every process p receives the message from c_0. In addition, since $n \geq f+\alpha+1$ and $|SK(\Phi)| \geq n-f$, every

process receives the message from $n-f \geq \alpha+1$ processes in $SK(\Phi)$. By (*), (**) and line 30, for all processes p and all $q \in SK(\Phi)$, we have $Msg_p[q] = m_q$. Thus $SK(\Phi) \subseteq SK(\rho)$, which shows that $SK(\rho) \geq n-f$. □

Corollaries 4.1 and 4.2 follow from Lemma 4.1.

Corollary 4.1. *If $n > (\alpha+1)(\alpha+f)$, then Algorithm 4.1 preserves $\mathscr{P}^{\alpha}_{dyn}$.*

By Corollary 4.1, preserving $\mathscr{P}^{\alpha}_{dyn}$ leads to a quadratic dependency between n and α. Corollary 4.2 shows the surprising result that, allowing more than α corruptions in the simulated round, leads instead to a linear dependency between n and α. Note that the simulation mentioned in Corollary 4.2 is not useful if $\left\lfloor \frac{\eta}{\eta-1}\alpha \right\rfloor \geq n$.

Corollary 4.2. *For any $\eta \in \mathbb{R}$, $\eta > 1$, if $n > \eta(\alpha+f)$, then Algorithm 4.1 simulates $\mathscr{P}^{\left\lfloor \frac{\eta}{\eta-1}\alpha \right\rfloor}_{dyn}$ from $\mathscr{P}^{\alpha}_{dyn}$.*

Proof. Let $\xi = \frac{\eta}{\eta-1}$. From $\lfloor \xi\alpha \rfloor > \xi\alpha - 1 = \alpha\frac{\eta}{\eta-1} - 1 = \frac{\alpha\eta-\eta+1}{\eta-1}$ it follows that $\frac{\lfloor \xi\alpha \rfloor + 1}{\lfloor \xi\alpha \rfloor - \alpha + 1} < \eta$. The corollary follows from Lemma 4.1 by setting $\beta = \lfloor \xi\alpha \rfloor$. □

4.4.1 Generic predicate

In Section 4.5 we solve consensus using the following generic predicate, which combines $\mathscr{P}_{\diamond cons}$ and $\mathscr{P}_{\diamond SK}$:

$$\begin{aligned}\mathscr{P}^{f,b,k}_{\diamond cons \oplus SK} :: &\forall \phi > 0, \exists \phi_0 \geq \phi,\ CONS((\phi_0 - 1)k + 1)\ \wedge |SK(\Phi)| \geq n-f,\\ &\text{where } \Phi = \left\{(\phi_0-1)k+1-b, \ldots, \phi_0 k\right\}\end{aligned}$$

It defines a phase with k rounds, where the first round of some phase ϕ_0 is consistent and all rounds of phase ϕ_0 plus the preceding b rounds have safe kernel of size at least equal to $n-f$.

Obviously, $\mathscr{P}_{\diamond cons \oplus SK}$ can be simulated from $\mathscr{P}_{\diamond SK}$ and $\mathscr{P}_{dyn}$ using Algorithm 4.1. Algorithm 4.1 simulates the first round of a phase, and a trivial simulation (where messages are just delivered as received) are used for the other rounds. Ensuring that the coordinator is in the safe kernel requires $f+1$ phases. The first macro-round of a phase requires 4 rounds, and the others $k-1$ only 1 round. Therefore $f+1$ phases correspond to $(k+3)(f+1)$ rounds. This leads to:

Corollary 4.3. *If $n > \frac{(\beta+1)(\alpha+f)}{\beta-\alpha+1}$, $n > \alpha+f$, $\alpha \geq f$, and $\beta \geq \alpha$, then $\mathscr{P}^{f,b,k}_{\diamond cons \oplus SK} \wedge \mathscr{P}^{\beta}_{dyn}$ can be simulated from $\mathscr{P}^{f,K}_{\diamond SK} \wedge \mathscr{P}^{\alpha}_{dyn}$, where $K = (k+3)(f+1) + b + (k+2)$.*

Note that the additional term $k+2$ for K stems from the fact that the rounds with a safe kernel are not necessarily aligned to the phases of $\mathscr{P}^{f,b,k}_{\diamond cons \oplus SK}$.

4.5 Solving consensus with eventual consistency

In this section we use the generic predicate $\mathscr{P}_{\diamond cons \oplus SK}$ to solve consensus. In a consensus algorithm below, the notation $\#(v)$ is used to denote the number of messages received with value v, *i.e.*,

$$\#(v) \equiv \left|\left\{q \in \Pi : \vec{\mu}_p^r[q] = v\right\}\right|.$$

4.5.1 The BLV algorithm

Algorithm 4.2 BLV algorithm

```
Initialization:
    $vote_p \leftarrow init_p \in V$
    $ts_p \leftarrow 0$
    $history_p \leftarrow \{(init_p, 0)\}$
Round $r = 3\phi - 2$:
    $S_p^r$:
        send $\langle vote_p, ts_p, history_p \rangle$ to all
    $T_p^r$:
        $select_p \leftarrow \mathscr{FBLV}_{T,\beta}(\vec{\mu}_p^r)$
        if $select_p \neq null$ then
            $history_p \leftarrow history_p \cup \{(select_p, \phi)\}$
Round $r = 3\phi - 1$:
    $S_p^r$:
        if $\exists(v, \phi) \in history_p$ then
            send $\langle v \rangle$ to all
    $T_p^r$:
        if $\#(v) \geq T$ then
            $vote_p \leftarrow v$
            $ts_p \leftarrow \phi$
Round $r = 3\phi$:
    $S_p^r$:
        if $ts_p = \phi$ then
            send $\langle vote_p \rangle$ to all
    $T_p^r$:
        if $\exists \bar{v} \neq \perp : \#(\bar{v}) \geq T$ then
            DECIDE $\bar{v}$
```

The algorithm we present is called BLV. It is inspired by CL algorithm (Algorithm 4.2) presented in Section 3.6, that we adapted for transmission value faults. BLV is designed to work under $\mathscr{P}_{dyn}^{\alpha}$. For safety, BLV requires $n > 2(\alpha + f)$ and $T > \frac{n}{2} + \alpha$. Termination is achieved if in addition the following predicate holds:

Algorithm 4.3 Function $\mathscr{FBLV}_{T,\beta}(\vec{M})$

```
27: $possibleV \leftarrow \{(v,ts) : \exists i \in \Pi : (v,ts) = \vec{M}[i] \wedge |\{q : \vec{M}[q] = (v',ts',-) \wedge ((v = v' \wedge ts = ts') \vee ts > ts')\}| \geq T\}$
28: $confirmedV \leftarrow \{v : (v,ts) \in possibleV \wedge |\{q : \vec{M}[q] = (-,-,history) \wedge (v,ts) \in history\}| > \alpha\}$
29: if $|confirmedV| \geq 1$ then
30:     return $\min(confirmedV)$
31: else if $\{q : \vec{M}[q] = (-,0,-)\} \geq T$ then
32:     return minimal $v$, such that $\exists(v,0,-) \in \vec{M}$ and $\nexists(v',0,-) \in \vec{M}\ s.t.\ \#((v',0,-)) > \#((v,0,-))$
33: else
34:     return $null$
```

$$\mathscr{P}^{f}_{BLV} :: \forall \phi > 0, \exists \phi_0 > \phi : CONS(3\phi_0 - 2) \wedge \forall r \in \{3\phi_0 - 2, \ldots, 3\phi_0\} : |SK(r)| \geq n - f$$

The $\mathscr{P}^{f}_{BLV}$ predicate ensures the existence of a phase ϕ_0 such that: (i) in the first round of ϕ_0 processes receive the same set of messages, and (ii) in all three rounds of ϕ_0 processes receive at least $n - f$ uncorrupted messages.

Obviously, $\mathscr{P}^{f,0,3}_{\diamond cons \oplus SK}$ implies $\mathscr{P}^{f}_{BLV}$. Weak interactive consistency ensures that at the end of round $3\phi_0 - 2$, all processes select the same value. The condition that there exists a large enough safe kernel in phase ϕ_0 finally forces every process to make a decision at the end of round $3\phi_0$.

The code of BLV is given as Algorithm 4.2. It consists of a sequence of phases, where each phase ϕ has three rounds $3\phi - 2$, $3\phi - 1$ and 3ϕ. The algorithm uses a timestamp variable *ts* in addition to to the variable *vote*. Whenever a process p updates $vote_p$ in round $3\phi - 1$, ts_p is set to ϕ (line 18 and 19). If enough processes update *vote* in round $3\phi - 1$, then a decision is possible in round 3ϕ. The condition at line 17 ensures that in round $3\phi - 1$, all processes that update $vote$, update it to the same value. This ensures that in round 3ϕ, processes attempt to decide on one single value, which is necessary for agreement.

In order to deal with value faults, BLV maintains also a *history* variable, which stores pairs (v, ϕ). Having $(v, \phi) \in history_p$ means that p added (v, ϕ) to $history_p$ in phase ϕ (line 11). The *history* variable ensures that a corrupted message with invalid values for *vote* and ts_p will not affect the safety properties of the algorithm.

Similarly as in Algorithm 4.2 round $3\phi - 2$ has two roles, the first related to agreement and integrity, and the second related to termination:

1. *Safety role:*

   (a) *For Agreement*: If a process p has decided v in some phase ϕ_0, then for any process q, only v can be assigned to $select_q$ at line 9 in phases $\phi > \phi_0$.

(b) *For Integrity*: If all processes have the same initial value v, then only v can be assigned to $select_p$ at line 9.

2. *Termination role*: In a consistent round with safe kernel of size $n - f$, all processes must assign the same value to *select* at line 9.

Line 9 refers to the selection function $\mathcal{FBLV}_{T,\beta}$, which takes as input the messages received in round $3\phi - 2$. We explain now this function (Algorithm 4.3):

- Line 27 (together with line 28) ensures 1a. More precisely, it ensures selection of the most recent vote in the history of some process. If a process has previously decided $\bar{v}$, then only $\bar{v}$ can be in *confirmedV*.[7]
- Line 28 prevents from returning a value v from a pair (v, ts) that is from a corrupted message: the pair must be in the history of at least one process. Therefore, a pair (v, ts) is considered only if it is part of the history in at least $\alpha + 1$ messages received. Together with line 27, it also ensures 1b: when all processes have the same initial value, no other value is in the $history_p$ variable of processes.

We consider now lines 29 and 30 of Algorithm 4.3. As we just explained, if a process has previously decided $\bar{v}$, then only $\bar{v}$ can be in *confirmedV*, that is, $|confirmedV| = 1$. In this case, by line 30, the function $\mathcal{FBLV}_{T,\beta}$ returns $\bar{v}$. If no correct process has decided, we can have $|confirmedV| > 1$. In this case, if some round $3\phi - 2$ is a consistent round with safe kernel of size $n - f$, then all processes consider the same set *confirmedV*, which ensures 2. Lines 31 and 32 are for the case where not all processes have the same initial value. Termination would be violated without these lines.

Correctness of the *BLV* algorithm

First we introduce some notation. For any variable x local to process p, we denote $x_p^{(r)}$ the value of x_p at the end of round r. For any value $v \in V$ and any process p, at any round $r > 0$, we define the sets $R_p^r(v)$ and $Q_p^r(v)$ as follows:

$$R_p^r(v) := \left\{q \in \Pi : \vec{\mu}_p^r[q] = v\right\}$$

$$Q_p^r(v) := \left\{q \in \Pi : S_q^r(p, s_q) = v\right\},$$

[7]Consider two phases ϕ_0 and $\phi_0 + 1$, such that a process has decided $\bar{v}$ in phase ϕ_0. We consider the more general case with dynamic faults, and we assume that $n = 5$, $f = \alpha = 1$ and $T = 4$. This means that at least $T - \alpha = 3$ processes have $ts = \phi_0$ and $vote = \bar{v}$. Consider in phase $\phi_0 + 1$ that $(v, ts) \in possibleV_p$ at p with $v \neq \bar{v}$. This means that p, in round $3(\phi_0 + 1) - 2$, has received $T = 4$ messages with either $(v, ts, -)$, or $(-, ts', -)$ and $ts' < ts$. Since $n = 5$ and $T = 4$, at least one of these messages is from a process c such that $vote_c = \bar{v}$ and $ts_c = \phi_0$. Since $v \neq \bar{v}$, we must have $\phi_0 < ts$. However, in phase $\phi_0 + 1$, no process p can have (v, ts) with $ts > \phi_0$ in $history_p$. Therefore, by line 28, we will not have $v \in confirmedV$.

where s_q denotes q's state at the beginning of round r. The set $R_p^r(v)$ (resp. $Q_p^r(v)$) represents the set of processes from which p receives v (resp. which ought to send v to p) at round r. Since at each round of the consensus algorithm, every process sends the same message to all, the sets $Q_p^r(v)$ do not depend on p, and so can be just denoted by $Q^r(v)$ without any ambiguity.

We start our correctness proof with a general basic lemma:

Lemma 4.3. *For any process p and any value v, at any round r, we have:*

$$|R_p^r(v)| \le |Q^r(v)| + |AHO(p,r)|$$

Proof. Suppose that process p receives a message with value v at round $r > 0$ from process q. Then, either the code of q prescribes it to send v to p at round r, i.e., q belongs to $Q^r(v)$ and thus q is also in $SHO(p,r)$, or the message has been corrupted and q is in $AHO(p,r)$. It follows that $R_p^r(v) \subseteq Q^r(v) \cup AHO(p,r)$, which implies $|R_p^r(v)| \le |Q^r(v)| + |AHO(p,r)|$. □

Definition 4.1. *A value v is locked in a phase ϕ by process p if $vote_p = v$ and $ts_p = \phi$ at the end of round $3\phi - 1$.*

Lemma 4.4. *If $T > \frac{n}{2} + \alpha$, then in any run of the HO machine $(BLV, \mathcal{P}_{dyn}^{\alpha})$ there is at most one locked value per phase.*

Proof. Assume by contradiction that there exist two processes p and q that lock different values v and v' in some phase $\phi_0 > 0$. From line 17 we deduce that $|R_p^{3\phi_0}(v)| \ge T$ and $|R_q^{3\phi_0}(v')| \ge T$. Then Lemma 4.3 (note that this lemma holds also for BLV) ensures that $|Q^{3\phi_0}(v)| \ge T - \alpha$ and $|Q^{3\phi_0}(v')| \ge T - \alpha$ when $\mathcal{P}_{dyn}^{\alpha}$ holds.

Since each process sends the same value to all at each round, the sets $Q^{3\phi_0}(v)$ and $Q^{3\phi_0}(v')$ are disjoint if v and v' are distinct values. Hence,

$$|Q^{3\phi_0}(v) \cup Q^{3\phi_0}(v')| = |Q^{3\phi_0}(v)| + |Q^{3\phi_0}(v')| \ge 2T - 2\alpha.$$

Consequently, since $T > \frac{n}{2} + \alpha$, we derive that $|Q^{3\phi_0}(v) \cup Q^{3\phi_0}(v')| > n$, a contradiction. □

Lemma 4.5. *If $T > \alpha$, then in any run of the HO machine $(BLV, \mathcal{P}_{dyn}^{\alpha})$ there is at most one possible decision value per phase.*

Proof. Assume by contradiction that there exist two processes p and q that decide on different values v and v' in some phase $\phi_0 > 0$. From line 25 we deduce that $|R_p^{3\phi_0}(v)| \ge T$ and $|R_q^{3\phi_0}(v')| \ge T$. Then Lemma 4.3 and $\mathcal{P}_{dyn}^{\alpha}$ ensure that $|Q^{3\phi_0}(v)| \ge T - \alpha$ and $|Q^{3\phi_0}(v')| \ge T - \alpha$.

Since each process sends the same value to all at each round, the sets $Q^{3\phi_0}(v)$ and $Q^{3\phi_0}(v')$ are disjoint since v and v' are distinct values. Hence, when $T > \alpha$, the sets $Q^{3\phi_0}(v)$ and $Q^{3\phi_0}(v')$

are not empty, and so by line 22, there exist two processes p' and q' that have $vote_{p'} = v$, $ts_{p'} = \phi_0$, $vote_{q'} = v'$ and $ts_{q'} = \phi_0$. A contradiction with Lemma 4.4. □

Lemma 4.6. *If $T > \frac{n}{2} + \alpha$, then in any run of the HO machine $(BLV, \mathcal{P}^{\alpha}_{dyn})$, if process p decides v in phase $\phi_0 > 0$, then for all later phases $\phi > \phi_0$ and all processes q, (v, ϕ) is the only pair that can be added to $history_q$.*

Proof. Assume by contradiction that $\phi_1 > \phi_0$ is the first phase where a pair (v_1, ϕ_1) with $v_1 \neq v$ is added to the $history_q$ at process q. This implies that if *history* at some process contains a pair (v', ϕ') with $v' \neq v$, then $\phi' \leq \phi_0$ (*).

Since by our assumption q added $(v', -)$ to $history_q$ in phase ϕ_1, this implies that $\mathcal{FBLV}_{T,\beta}$ returns v' at line 9 in phase ϕ_1. Therefore, either (i) line 30 or (ii) line 32 of Algorithm 4.3 was executed by q in phase ϕ_1.

In case (ii), the condition of line 31 has to be true. This implies that $|R_q^{3\phi_1-2}((-,0,-))| \geq T$, and thus, by Lemma 4.3, $|Q^{3\phi_1-2}((-,0,-))| \geq T - \alpha$.

> We prove an intermediate result: In phases ϕ such that $\phi_0 \leq \phi < \phi_1$, we have $|\{q \in \Pi : vote_q^{3\phi-1} = v \wedge ts_q^{3\phi-1} \geq \phi_0\}| \geq T - \alpha$. Since p decides v in phase ϕ_0, $|R_p^{3\phi_0}(v)| \geq T$, and thus by Lemma 4.3, we have $|Q^{3\phi_0}(v)| \geq T - \alpha$. From the code of the BLV algorithm, we have $Q^{3\phi_0}(v) = \{q : vote_q^{3\phi_0-1} = v \wedge ts_q^{3\phi_0-1} = \phi_0\}$, therefore the claim holds for phase ϕ_0.
>
> We now show that any process that locked value v in phase ϕ_0 (see Definition 4.1) and updates *vote* in phase ϕ such that $\phi_0 < \phi < \phi_1$, sets it to v. This ensures the claim. Assume by contradiction that one of these processes q' sets $vote_q$ to v' in round $3\phi - 1$. By line 17, $|R_q^{3\phi-1}(v')| \geq T$. Then Lemma 4.3 ensures that $|Q^{3\phi-1}(v')| \geq T - \alpha$. Since $T > \alpha$, we have $|Q^{3\phi-1}(v')| > 0$, i.e. at least one process sent v' at line 15. Therefore, by line 14 at least one process has $(v', \phi_0 + 1)$ in *history*, a contradiction with the assumption that ϕ_1 is the first phase where a pair $(v', -)$ is added to *history* at some process.

So we have also $|\bigcup_{ts \geq \phi_0} Q^{3\phi_1-2}((v, ts, -))| \geq T - \alpha$. Since in each round, every process sends the same value to all, and $\phi_0 > 0$, the sets $X(v) = \bigcup_{ts \geq \phi_0} Q^{3\phi_1-2}((v, ts, -))$ and $Q^{3\phi_1-2}((-,0,-))$ are disjoint. Hence,

$$|X(v) \cup Q^{3\phi_1-2}((-,0,-))| = |X(v)| + |Q^{3\phi_1-2}((-,0,-))| \geq 2T - 2\alpha.$$

Together with $T > \frac{n}{2} + \alpha$, we derive that $|\bigcup_{ts \geq \phi_0} Q^{3\phi_1-2}((v, ts, -)) \cup Q^{3\phi_1-2}((-,0,-))| > n$, a contradiction.

In case (i), the condition at line 29, has to be true, i.e., v' need to be part of *confirmedV* set at line 28. Value v' can be part of the set *confirmedV* only if (v', ts') is part of the set *possibleV* at line 27. We show that if (v', ts') is part of the set *possibleV* at line 27, v' cannot be part of the set *confirmedV* at line 28, which establishes the contradiction.

If the pair (v', ts') is added to the set *possibleV* at line 27, then $HO(q, 3\phi_1 - 2) \geq T$. Since $2T - \alpha > n + \alpha$, $|HO(q, 3\phi_1 - 2) \cap \bigcup_{ts \geq \phi_0} Q^{3\phi_1 - 2}((v, ts, -))| > \alpha$. Therefore, since $\mathcal{P}^{\alpha}_{dyn}$ holds, any set of messages of size T contains at least one message m with $m.vote = v$ and $m.ts \geq \phi_0$ (**). So we have $|\{m' \in \vec{\mu}^{r}_{q} : (m'.ts < ts')\}| \geq T$ and, because of (**), $ts' > \phi_0$.

The value v' is added to the set *confirmedV* at line 28 only if there are at least $\alpha + 1$ messages $\overline{m}$ in $\vec{\mu}^{3\phi-2}_{q}$ such that: $(\overline{v}, \overline{\phi}) \in \overline{m}.history$ and $\overline{\phi} \geq ts'$ and $\overline{v} = v'$. Since $ts' > \phi_0$, by (*), q receives at most α such messages, a contradiction. □

Proposition 4.1 (Agreement)**.** *If $T > \frac{n}{2} + \alpha$, then no two processes can decide differently in any run of the HO machine $(BLV, \mathcal{P}^{\alpha}_{dyn})$.*

Proof. Let a phase $\phi_0 > 0$ be the first phase at which some process p makes a decision, and let v be the p's decision value. Assume that process q decides v' at phase ϕ'. By definition of ϕ_0, we have $\phi' \geq \phi_0$.

We proceed by contradiction and assume that $v \neq v'$. By Lemma 4.5, we derive that $\phi' > \phi_0$. Since q decides at round $3\phi'$, by line 25 we have $|R^{3\phi'}_{q}(v')| \geq T$. By Lemma 4.3, we have $|Q^{3\phi'}(v')| \geq T - \alpha$. Since $T > \alpha$, there is at least one process p' that sends v' in round $3\phi'$. By line 22 and line 19 we have that process p' sends it's current *vote* in round $3\phi'$ only if *vote* is updated in round $3\phi' - 1$. Therefore, $|R^{3\phi'-1}_{p'}(v')| \geq T$, i.e. by Lemma 4.3, we have $|Q^{3\phi'-1}(v')| \geq T - \alpha$. Since $T > \alpha$, at least one process q' sends v' in round $3\phi' - 1$. By line 14, if q' sends v' in round $3\phi' - 1$, then $\exists (v', \phi') \in history^{3\phi'-2}_{q'}$, a contradiction with Lemma 4.6. □

Lemma 4.7. *If $T > 2\alpha$, then in any run of the HO machine $(BLV, \mathcal{P}^{\alpha}_{dyn})$ where all the initial values are equal to some value v, for all processes q, $history_q$ contains only pairs $(v, -)$.*

Proof. Since all processes have v as their initial value, *history* at all processes is initialized to $(v, 0)$. Assume by contradiction that ϕ_0 is the first phase where a pair $(v', -)$ is added to $history_p$ at some process p (*). This implies that $\mathcal{FBLV}_{T,\beta}$ returns v' at line 9. Therefore, either (i) line 30 or (ii) line 32 of Algorithm 4.3 was executed by p in phase ϕ_0.

For (i), the condition at line 29 has to be true, i.e., v' needs to be in *confirmedV* at line 28. This means that p received more than α messages $m = (-, -, history_m)$ with $(v', ts) \in history_m$ in round $3\phi_0 - 2$. By Lemma 4.3 and $\mathcal{P}^{\alpha}_{dyn}$, at least one process sends a message $m = \langle -, -, history_m \rangle$ with $(v', ts) \in history_m$ in round $3\phi_0 - 2$, a contradiction with (*).

For (ii), the condition of line 31 has to be true. If this condition is true, this implies that $|HO(p, 3\phi_0 - 2)| \geq T$. Since $T > 2\alpha$, $\mathcal{P}^{\alpha}_{dyn}$ holds, and all processes have the same initial value v,

v is returned at line 32 and (v, ϕ_0) is added to the $history_p$. A contradiction. □

Proposition 4.2 (Integrity)**.** *If $T > 2\alpha$, then in any run of the HO machine $(BLV, \mathscr{P}^{\alpha}_{dyn})$ where all the initial values are equal to some value v, the only possible decision value is v.*

Proof. By contradiction, assume that phase $\phi_0 > 0$ is the first phase in which some process p decides $v' \neq v$.

Since p decides at round $3\phi_0$, by line 25 we have $|R_p^{3\phi_0}(v')| \geq T$. By Lemma 4.3 and $\mathscr{P}^{\alpha}_{dyn}$, we have $|Q^{3\phi_0}(v')| \geq T - \alpha$. Since $T > \alpha$, there is at least one process q that sends v' in round $3\phi_0$. By line 22 and line 19, we have that process q sends it's current *vote* in round $3\phi_0$ only if *vote* is updated in round $3\phi_0 - 1$. Therefore, $|R_q^{3\phi_0-1}(v')| \geq T$, i.e., by Lemma 4.3 and $\mathscr{P}^{\alpha}_{dyn}$, we have $|Q^{3\phi_0-1}(v')| \geq T - \alpha$. Since $T > \alpha$, at least one process q' sends v' in round $3\phi_0 - 1$. By line 14, if q' sends v' in round $3\phi_0 - 1$, then $\exists (v', \phi_0) \in history_{q'}^{3\phi_0-2}$, a contradiction with Lemma 4.7. □

Proposition 4.3 (Termination)**.** *If $n > 2(f + \alpha)$, $T > \frac{n}{2} + \alpha$ and $f \leq \alpha$, then any run of of the HO machine $(BLV, \mathscr{P}^{\alpha}_{dyn} \wedge \mathscr{P}^{f}_{BLV})$ satisfies the Termination clause of consensus.*

Proof. By $\mathscr{P}^{f}_{BLV}$, there exists a phase ϕ_0 such that

$$CONS(3\phi_0 - 2) \wedge \forall r \in \{3\phi_0 - 2, \ldots, 3\phi_0\} : SK(r) \geq n - f.$$

Therefore, in round $3\phi_0 - 2$, for any two processes p and q, we have $\vec{\mu}_p^r = \vec{\mu}_q^r$, and $|SHO(p, 3\phi_0 - 2) \cap SHO(q, 3\phi_0 - 2)| \geq n - f$.

Part A. We now prove that $select_p^{3\phi_0-2}$ will be the same at all processes p, i.e., that $\mathscr{FBLV}_{T,\beta}$ returns the same value at all processes, and all processes add the same pair to *history* in round $3\phi_0 - 2$. There are two cases to consider: (i) some process $p \in SK(\phi_0)$ locked a value in some phase smaller than ϕ_0, or (ii) there is no such process in $SK(\phi_0)$.

Case (i): Let $\phi < \phi_0$ be the largest phase in which some process p locked some value v (line 18). By Lemma 4.4 and since $Q > \frac{n}{2} + \alpha$, all processes that lock a value in phase ϕ, lock the same value v. Since $n > 2(f + \alpha)$ and $T > \frac{n}{2} + \alpha$, $n - f \geq T$; therefore in case (i) at least a pair $(v, -)$ is added to the set *possibleV* at line 27 of Algorithm 4.3 (*).

We consider now line 28 of Algorithm 4.3. If p locked value v in phase ϕ, then $|R_p^{3\phi-1}(v)| \geq T$, i.e., by Lemma 4.3, we have $|Q^{3\phi-1}(v)| \geq T - \alpha$ when $\mathscr{P}^{\alpha}_{dyn}$ holds. Because of line 14 of Algorithm 4.2, at least $T - \alpha$ processes have (v, ϕ) in *history*. By assumption, $n > 2(f + \alpha)$ and $T > \frac{n}{2} + \alpha$, therefore $n - f + T > n + \alpha$. Therefore, because of $|SK(\phi_0)| \geq n - f$, any set of messages received in round $3\phi_0 - 2$ contains more than α messages m with $(v, \phi) \in m.history$. Since $n > 2(f + \alpha)$ and $T > \frac{n}{2} + \alpha$, $n - f \geq T$ (***), and therefore v is added to the set *confirmedV* at line 28 of Algorithm 4.3 (**).

From (*) and (**), it follows that the condition of line 29 of Algorithm 4.3 is true at all processes in phase ϕ_0. Moreover, since function $\mathscr{FBLV}_{T,\beta}$ is deterministic and $CONS(3\phi_0 - 2)$ holds, for any two processes p and q, we have $select_p = select_q$ at line 9. Therefore p and q add the same pair to *history* at line 11.

Case (ii): By hypothesis, for all processes $p \in SK(\phi_0)$, we have $ts_p = 0$. By (***) $n - f \geq T$ and therefore the condition at line 31 of Algorithm 4.3 is true at each process. Moreover, by $CONS(3\phi_0 - 2)$ we have for any two processes p and q $\vec{\mu}^r_p = \vec{\mu}^r_q$. Therefore, the value returned at line 32 of Algorithm 4.3 is the same at all processes, and they will add the same pair to $history$ at line 11 of Algorithm 4.2.

Part B. From Part A, there exists a value v such that at all processes p we have $(v, \phi_0) \in history_p$ at the beginning of round $3\phi_0 - 1$. Therefore all processes send v to all at line 15. By $|SK(3\phi_0 - 1)| \geq n - f$ we have that all processes receive at least $n - f$ messages equal to v, and since by (***) $n - f \geq T$, they all set $vote_p$ to v (line 18) and send v to all at line 23. By $|SK(3\phi_0)| \geq n - f$ and the same reasoning we can show that all processes receive $n - f$ messages equal to v in round $3\phi_0$, and since by (***) $n - f \geq T$, decide v at line 26 in phase ϕ_0. □

Combining Propositions 4.1, 4.2, and 4.3, we get the following theorem:

Theorem 4.3. *If $n > 2(\alpha + f)$ and $T > \frac{n}{2} + \alpha$, then the HO machine $\langle BLV, \mathscr{P}^f_{BLV} \wedge \mathscr{P}^{\alpha}_{dyn} \rangle$ solves consensus.*

Note that the BLV algorithm can also be used in the model considered in [BCBG+07], where all faults are transient. By Theorem 4.3, the BLV algorithm solves consensus in this model if $n > 2\alpha$ $(f = 0)$, in contrast to algorithm $\mathscr{A}_{T,E}$ in [BCBG+07], which requires $n > 4\alpha$. Algorithm $\mathscr{U}_{T,E,\alpha}$ in [BCBG+07] requires $n > 2\alpha$ but, contrary to BLV, requires for safety that in every round every process receives a sufficient number of correct messages. This is not required by BLV, which is still correct even if processes do not receive any correct message in some rounds.

4.6 Deriving the overall resilience of BLV

In this section we look at the overall resilience of the BLV consensus algorithm together with the $\mathscr{P}^f_{BLV}$ predicate simulation algorithm.

When solving consensus in the presence of dynamic value faults ($\mathscr{P}_{\diamond SK} \wedge \mathscr{P}^f_{dyn}$), the BLV and the simulation algorithm have different requirements on n.

From Corollary 4.3 and the fact that $\mathscr{P}^{f,0,3}_{\diamond cons \oplus SK}$ implies $\mathscr{P}^f_{BLV}$ we get:

Corollary 4.4. *If $n > \frac{(\beta+1)(\alpha+f)}{\beta-\alpha+1}$, $n > \alpha + f$, $\alpha \geq f$, and $\beta \geq \alpha$, then Algorithm 4.1 simulates $\mathscr{P}^f_{BLV} \wedge \mathscr{P}^{\beta}_{dyn}$ from $\mathscr{P}^{6(f+1)+5}_{\diamond SK} \wedge \mathscr{P}^{\alpha}_{dyn}$.*

From Corollary 4.4 for any $\beta \geq \alpha$, we can simulate $\mathscr{P}^{f}_{BLV} \wedge \mathscr{P}^{\beta}_{corr}$ from $\mathscr{P}^{f,6(f+1)+5}_{\diamond SK} \wedge \mathscr{P}^{\alpha}_{dyn}$ if

$$n > \frac{(\beta+1)(\alpha+f)}{\beta-\alpha+1} \quad \wedge \quad n > \alpha + f$$

On the other hand, from Theorem 4.3 we know that we can solve consensus with BLV under $\mathscr{P}^{f}_{BLV} \wedge \mathscr{P}^{\beta}_{corr}$ if

$$n > 2(\beta + f).$$

Combining these conditions and setting $\beta = k\alpha$, where $k \in \mathbb{R}$, $k \geq 1$, we can solve consensus with Algorithm 4.2 and Algorithm 4.1 under $\mathscr{P}^{f,6(f+1)+5}_{\diamond SK} \wedge \mathscr{P}^{\alpha}_{dyn}$ if the following two conditions hold:

$$n > \frac{(k\alpha+1)(\alpha+f)}{k\alpha-\alpha+1} \tag{4.4}$$

$$n > 2(k-1)\alpha + 2(\alpha+f). \tag{4.5}$$

We first consider $\alpha > 1$, then $\alpha = 1$.

Case $\alpha > 1$: We can obtain different resilience bounds depending on the choice of k.

Choosing $k = 1$ leads to the quadratic dependency from Corollary 4.1, and is thus not what we want to achieve here.

For $k \geq 2$, condition (4.5) implies condition (4.4) for any $\alpha > 1$, because $\frac{k\alpha+1}{k\alpha-\alpha+1} \leq 2$. Thus, when choosing $k \geq 2$, the smallest n is obtained with $k = 2$:

$$n > 4\alpha + 2f.$$

In case $1 < k < 2$, the optimal choice of k depends on α and f. As special case we get for $k = 1.5$ from condition (4.4), $n > \frac{3\alpha+2}{\alpha+2}(\alpha+f)$, i.e.,

$$n > 3(\alpha + f)$$

while from condition (4.5) we get

$$n > 3\alpha + 2f$$

Since both conditions should hold, it follows that $n > 3(\alpha + f)$.

Case $\alpha = 1$: For the special case $\alpha = 1$ and $f = 1$, conditions (4.5) and (4.4) become $n > 2(k-1) + 4$ and $n > \frac{2(k+1)}{k}$. We obtain the smallest value for n by choosing $k = 1$, which leads to $n > 4$.

Discussion: The results show that $k = 1$ (i.e., $\beta = \alpha$) leads to the smallest value of n only when $\alpha = 1$. In cases where $\alpha > 1$, a better choice is $k = 1.5$ (i.e., $\beta = 1.5\alpha$). This is a non intuitive result.

4.7 Communication predicates and corresponding systems

In the HO model, there are no faulty processes and no state corruption. Nevertheless, for predicates that characterize permanent faults, the model can be used to reason about classical Byzantine faults. This implies that the algorithm in this paper can be used also to solve consensus in the classical Byzantine fault model. We develop this observation first for a synchronous system (for simplicity), and then extend it to our model.[8]

Let S_f denote a synchronous system with reliable links and at most f Byzantine processes, and consider on the other hand an HO machine with $|SK| \geq n - f$. For correct processes, a run in S_f is indistinguishable from a run of the HO machine. Therefore, an algorithm that solves consensus with $|SK| \geq n - f$ allows in S_f correct processes to solve consensus. Note that in S_f faulty processes do not follow the protocol. It is then natural that they do not follow the specification of consensus.

The same indistinguishability argument can be applied to (i) the weaker partial synchronous system [DLS88] with at most f Byzantine processes and (ii) the HO model with $\mathscr{P}^{f}_{perm} \wedge \mathscr{P}^{f,\infty}_{\diamond SK}$. For correct processes in the model (i), a run is indistinguishable from a run in model (ii), and so an HO algorithm that solves consensus allows correct processes in the fault-prone system to solve consensus.

The predicate $\mathscr{P}^{\alpha}_{dyn} \wedge \mathscr{P}^{f,k}_{\diamond SK}$, $\alpha \geq f$, can correspond to a partially synchronous system with at most f Byzantine processes, where in addition, before stabilization time, in every round processes can receive $\alpha - f$ corrupted messages from correct processes (*). This spectrum of interpretations, which includes permanent faults (see Sect. 4.3.3) contrary to [BCBG⁺07], shows the benefit of considering the consensus problem in a model with (only) transmission faults.

Further, these interpretations show that the BLV consensus algorithm presented in this chapter can be used in classical system models. This allows us to compare BLV with existing consensus algorithms, specifically consensus algorithms that tolerate arbitrary faults (process and/or link faults). The comparison appears in Table 4.1. For BLV we assume the interpretation (*) in the preceding paragraph.

[8]This observation was made already in [Lam01] and [BCBG$^+$07], but without giving algorithms supporting the observation.

Algorithm	*Synchrony*	*Static and Permanent*	*Dynamic and Transient*	*Process faults*	*Link faults*	*Resilience*
[PSL80, LSP82]	synchronous	✓		✓		$n > 3f$
[SAAAA95]	synchronous	✓			✓	$n > 2l_a + 2$
[YCW92, SCY98]	synchronous	✓		✓	✓	$n > 3f$ and $c > 2f + l_a$
OMH [BSW11]	synchronous	✓	✓	✓	✓	$n > 3f + f_l^{ra} + 2f_l^s + f_l^r + 2f_s + 2f_o + f_m$
FaB Paxos [MA06]	partially synchronous	✓		✓		$n > 5f$
PBFT [CL02]	partially synchronous	✓		✓		$n > 3f$
DLS [DLS88]	partially synchronous	✓		✓		$n > 3f$
$\mathscr{A}_{T,E}$ [BCBG+07]	partially synchronous		✓		✓	$n > 4\alpha$
$\mathscr{U}_{T,E,\alpha}$ [BCBG+07]	partially synchronous		✓		✓	$n > 2\alpha$
BLV	partially synchronous	✓	✓	✓	✓	$n > 3f + 3\alpha$

Table 4.1: Summary of consensus algorithms that tolerate arbitrary faults. The parameter f denotes the number of Byzantine faulty processes; l_a denotes the number of links subjected to arbitrary faults. In [YCW92, SCY98], the parameter c denotes the network connectivity (value c means that there exists at least c disjoint paths between any pairs of processes). In OMH, f_l^{ra} is the number arbitrary receive link failures, f_l^s the number of send link failures, f_l^r the number of receive link failures, f_s the number of symmetrical Byzantine faulty processes, f_o the number of omission faulty processes and f_m the number of manifest faulty processes. The parameter α denotes the maximum number of corrupted messages a process can receive per round.

4.8 Conclusion

The transmission fault model allows us to reason about permanent and transient value faults in a uniform way, which makes the model very attractive. However, all existing solutions to consensus in this model are either in the synchronous system model, or require strong conditions for termination that exclude the case where all messages of a process can be corrupted. The chapter has shown that this limitation can be overcome thanks to the weak interactive consistency predicate that states the existence of a round where all processes receive the same set of messages. The simulation of weak interactive consistency from a predicate that corresponds to a partially synchronous system parameterized with α (in every round each process can receive up to α corrupted messages) and f (at most f processes are corrupted) has been given. The simulation is compatible with permanent and transient faults. The chapter has pointed out two options for the simulation: preserving or not the number of corrupted messages in each round. The first option requires $n > (\alpha+1)(\alpha+f)$. The second option requires $n > \eta(\alpha+f)$. Combining the BLV consensus algorithm with this second simulation leads to $n > 3(\alpha+f)$ for $\alpha > 1$ and $n > 4$ for $\alpha = 1$.

5 Generic Consensus Algorithm for Benign and Byzantine Faults

In the Chapter 3 we introduced the *weak interactive consistency* (*WIC*) abstraction, and have shown how it allows us to unify Byzantine consensus algorithms with and without signatures. In this chapter we go one step further (in unifying consensus algorithms) by proposing a *generic consensus algorithm* that highlights, through well chosen parameters, the core mechanisms of a number of well-known consensus algorithms including Paxos [Lam98], OneThirdRule [CBS09a], PBFT [CL02] and FaB Paxos [MA06]. Interestingly, the generic algorithm allows us to identify a new Byzantine consensus algorithm that requires $n > 4b$, in-between the requirement $n > 5b$ of FaB Paxos and $n > 3b$ of PBFT (b is the maximum number of Byzantine processes). The chapter contributes to identify key similarities rather than non fundamental differences between consensus algorithms.

Publication: Olivier Rütti, Zarko Milosevic, André Schiper. Generic construction of consensus algorithms for benign and Byzantine faults. In *The 40th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2010)*, Chicago, Illinois, USA, June 28 - July 1, 2010.

5.1 Introduction

Our generic consensus algorithm assumes a partially synchronous system model [DLS88] (see Section 2.4). However, in order to improve the clarity of the algorithms and simplify the proofs, as in [DLS88], we consider an abstraction on top of the system model, namely the round model (see Section 2.5). Expressing a consensus algorithm in the round model or directly in the partially synchronous system, does not change its core mechanisms.

The generic algorithm consists of successive phases, where each phase is composed of three rounds: a *selection* round, a *validation* round and a *decision* round. The validation round may be skipped by some algorithms, which introduces a first dichotomy among consensus algorithms: those that require the validation round, and the others for which the validation round is not necessary. We further subdivide the algorithms that require a validation round in two, based on the state variables required. This lead us to identify three classes of consensus algorithms: OneThirdRule and FaB Paxos belong to class 1, Paxos to class 2 and PBFT to class 3.

Our generic algorithm is based on four parameters: two functions (*FLV* and *Validator*), the threshold parameter T_D, and a *FLAG* ($FLAG = *$ or $FLAG = \phi$). The functions *FLV* and *Validator* are characterized by abstract properties; T_D is defined with respect to n (number of processes), f (maximum number of honest faulty processes, see Section 2.1) and b (maximum number of Byzantine processes). We prove correctness of the generic consensus algorithm by referring only to the abstract properties of our parameters. The correctness proof of any specific instantiated consensus algorithm consists simply in proving that the instantiations satisfy the abstract properties of *FLV* and *Validator*.

This is not the first tentative to propose a generic consensus algorithm, but it goes significantly beyond previous approaches. Mostéfaoui *et al.* [MRR02] propose a consensus framework restricted to benign faults, which allows unification of leader oracle, random oracle and failure detector oracle. Guerraoui and Raynal [GR04] propose a generic consensus algorithm, where the generality is encapsulated in a function called $Lambda$. The $Lambda$ function encapsulates our selection and validation rounds. This does not allow the paper to identify the differences between two of our three classes of consensus algorithms. Moreover, as for [MRR02], the paper is restricted to benign faults. Later, Guerrraoui and Raynal [GR07] proposed a generic version of Paxos in which communication (using shared memory, storage area networks, message passing channels or active disks) is encapsulated in the *Omega* abstraction. The paper is also restricted to benign faults. Apart from this work, several other authors proposed abstractions related to Paxos-like protocols, e.g., [Lam01, LCAA07, Cac10, Lam11].

More generally, Song et al. [SvRSD08] proposed building blocks that are used to construct a skeletal consensus algorithm. The skeletal algorithm is then used to instantiate three consensus algorithms: Paxos, Chandra-Toueg [CT96a] and the Ben-Or consensus algorithm for Byzantine faults [BO83]. The skeletal algorithm is expressed in terms of several process roles, selectors, archivers and deciders, and parameterized with two quorum systems, the selectors quorum system $\mathscr{S}^r$ and archivers quorums system $\mathscr{A}$. The algorithm proceeds in multiple instances (corresponds to rounds, phases or ballots in other approaches) where each instance corresponds to a sub-protocol that itself might decide. The agreement among multiple instances is ensured by proposing a "safe" value, captured by the notion of *guarded proposal*, that is similar to our *FLV* function. The difference is that contrary to the *FLV* function, that is defined with the abstract properties, the guarded proposal is not a parameter, but already represents an instantiation that covers three consensus algorithms considered. This is the main limitation of the approach, as the instantiation of the guarded proposal does not generalize to some seminal consensus algorithms such as PBFT and FaB Paxos. The role of the $\mathscr{S}^r$ quorum is to prevent archivers from archiving different suggestions in the same instance and is similar to our *Validator* parameter. The instantiation of the $\mathscr{A}$ quorum system is the same for all three consensus algorithms considered and consists of all processes, while algorithms use different instantiations of $\mathscr{S}^r$.

Finally, Zieliński [Zie06] proposed an *agreement framework* based on the optimistically terminating consensus (OTC) abstraction. OTC can be seen as a weaker variant of consensus that terminates only if all correct processes propose the same value (and no correct process stop an OTC instance)[1]). Reconstructing consensus algorithm within the OTC framework means obtaining an algorithm that matches latency (number of communication steps) in favorable scenarios (when the first coordinator is correct) and the number of required processes of the original algorithm. The paper shows how multiple well-known consensus algorithms can be reconstructed from these algorithms, e.g., CT, Paxos, OTR, FaB Paxos and PBFT. The reconstruction is based on the two coordinator-based consensus algorithms based on the OTC abstraction, one for benign faults and the other for Byzantine faults that uses signatures. Informally speaking, these algorithms capture the "good case" (called also normal case) of consensus algorithms[2], but simplifies the "recovery" part (as the focus is on the best-case latency). Therefore, the framework does not correctly capture the recovery part of consensus algorithms (considered as the most complex part of the algorithms). For example the algorithm for benign case does not have the timestamp variable required by Paxos, or does not capture correctly the "view change" protocol of PBFT.

Roadmap The rest of the chapter is organized as follows. Section 5.2 introduces the system model. We derive our generic consensus algorithm and prove its correctness in Section 5.3. In Section 5.4 we present three instantiations of the *FLV* function that lead to the three classes of consensus algorithms. Section 5.5 gives examples of instantiations and Section 5.6 concludes the chapter.

5.2 Model

Unless stated otherwise, in the rest of the chapter we consider the partially synchronous system model. As explained in Section 2.5, we consider an abstraction on top of the system model, namely the basic round model. Among the n processes in our system, we assume at most b Byzantine processes and at most f faulty (honest) processes (see Section 2.1). We do not make any assumption about the behavior of Byzantine processes.

We consider the communication predicates introduced in Section 3.3: $\mathscr{P}_{int}$, $\mathscr{P}_{good}$ and $\mathscr{P}_{cons}$.

In the benign fault model (i.e., $b = 0$), $\mathscr{P}_{cons}$ can be implemented using the implementation of $\mathscr{P}_{good}$ described in [DLS88] if we assume that no crash occurs in good periods. In Section 3.4 two coordinator-based implementations of $\mathscr{P}_{cons}$ have been proposed, for Byzantine faults and authenticated Byzantine faults. As already mentioned in Section 3.4, there is also a

[1]Apart from the $propose(v)$ primitive that process uses to propose a value v to OTC, there is also a $stop$ primitive to stop processing messages in the OTC instance.

[2]The algorithm differ in OTC implementations.

decentralized (i.e., coordinator-free) implementation of $\mathscr{P}_{cons}$ for the Byzantine fault model that requires $b+1$ micro-rounds [BS10]. The predicate $\mathscr{P}_{cons}$ allows us to describe consensus algorithms without making difference between authenticated Byzantine faults and Byzantine faults. Therefore, in this chapter we use the term Byzantine faults for both fault models, except if explicitly mentioned.

A *phase* is a sequence of rounds. We define a *good phase* of k rounds as a phase such that $\mathscr{P}_{cons}$ holds in the first round, and $\mathscr{P}_{good}$ holds in the remaining $k-1$ rounds.

5.3 Deriving a generic consensus algorithm

The goal of this section is to understand what are the core mechanisms (sometimes also called "building blocks" [SvRSD08]) present in existing consensus algorithms. Identifying these mechanisms and the properties they provide will allow us to derive a generic consensus algorithm.

5.3.1 Very simple consensus algorithm

We start our quest for a generic consensus algorithm with a very simple consensus algorithm (code shown as Algorithm 5.1). Algorithm 5.1 consists of a single round in which each process collects initial values from all processes, then applies a deterministic function to choose some value v (line 5) and then decides on v. The notation $\#(v)$ is used to denote the number of messages received with value v, *i.e.*, $\#(v) \equiv \left|\left\{q \in \Pi : \vec{\mu}_p^r[q] = v\right\}\right|$.

Algorithm 5.1 Very simple consensus algorithm

```
1: **Round** $r = 1$:
2:   $S_p^r$:
3:     send $\langle init_p \rangle$ to all
4:   $T_p^r$:
5:     $v \leftarrow \min\{v : \nexists v' \in V \ s.t. \#(v') > \#(v)\}$
6:     DECIDE $v$
```

Theorem 5.1. *Algorithm 5.1 solves consensus if* $n > 2b + f$ *and* $\mathscr{P}_{cons}(1)$ *holds.*

Proof. Termination and Weak Validity trivially holds. Agreement holds from $\mathscr{P}_{cons}(1)$ and the fact that all processes choose the decision values using the deterministic *min* function at line 5. We now prove that Strong Unanimity also holds.

We assume that initial value of all honest processes is v. The proof is by contradiction. We assume by contradiction that there is an honest process p that decides $v' \neq v$. Therefore, v' is the smallest most frequent value received by p in round 1 at line 5. By $\mathscr{P}_{cons}(1)$, process p receives at least $n-b-f$ messages equal to v in round 1. Furthermore, since there are at most b Byzantine processes, p received at most b messages equal to v'. Since $n > 2b+f$, p received more than b messages equal to v, and at most b messages equal to v'. Therefore, the value selected at line 5 is v and p decided v. A contradiction. □

5.3.2 Generic Algorithm: Draft 1

Algorithm 5.1 is not correct in the partially synchronous system model because $\mathscr{P}_{cons}$ cannot be ensured from the beginning. To remedy this, a consensus protocol has to invoke multiple instances of a sub-protocol. Such sub-protocols have been called rounds, phases, views or ballots. In this chapter, we use the term *phase*, where a phase itself consists of *rounds*. Using this terminology, we can say that Algorithm 5.1 consists of a single phase with a single round. In the partially synchronous system model, algorithms consist of a sequence of phases, where each phase contains some fixed number of rounds.

Having multiple phases requires additional mechanisms compared to Algorithm 5.1:

(i) A mechanism to detect that a decision is possible in a given phase. Clearly, this mechanism must ensure that two honest processes that decide in a given phase, decide the same value.

(ii) A mechanism to ensure consistency among decisions made by honest processes in different phases.

(iii) A mechanism to ensure that all correct processes eventually decide.

Concerning (i), we introduce the notion of *decision quorum* captured by parameter T_D. The parameter T_D defines the number of identical votes required to decide. More precisely, once a process p observes that T_D processes (decision quorum) have voted for some value v, then it can decide on v. There are some obvious restrictions on the value of T_D:

- To ensure strong unanimity, we must have $T_D > b$, i.e., a decision quorum must contain at least one honest process.
- To ensure termination, the votes of faulty (honest) and Byzantine processes must not be required to decide. Hence, $T_D \leq n - b - f$.

Concerning (ii), we introduce the notion of *locked value* [3] and the function $FLV(\vec{\mu}_p^r)$ (stands for "*F*ind the *L*ocked *V*alue") used to retrieve locked value (if there is some) from a set of messages received.[4] A value v is locked in round r if:

1. An honest process has decided v in round $r' < r$, or

[3] The definition of *locked value* in some other works, e.g. [DLS88], differs from our definition.

[4] *FLV* is not really a function. It is rather a problem defined by properties. However, calling it a function is more intuitive.

2. All honest processes have the same initial value v.

Item 2 is meaningful only if strong unanimity has to be ensured, or if all processes are honest. In all other cases, item 2 can be ignored. From this definition it follows that, if v is locked in the context of a consensus algorithm then the configuration is v-valent. However, the opposite is not true (e.g., if a configuration is v-valent in round r, and the first honest process p decides v in round $r' \geq r$, then v is not locked in round r, but only in round $r'+1 > r$).

The basic idea for ensuring agreement among different phases is the following. If some value v is locked in round r, then any honest process p that updates its variable $vote_p$[5] in round r, can only, thanks to the function $FLV(\vec{\mu}_p^r)$, update it to v. In addition to normal values, the function FLV may return the following special values:

- ? if no value is locked, i.e., any value can be assigned to $vote_p$
- $null$ if not enough information is provided to FLV through $\vec{\mu}_p^r$

The $FLV(\vec{\mu}_p^r)$ function is defined by the following three properties:

- *FLV-validity*: If all processes are honest and $FLV(\vec{\mu}_p^r)$ returns v such that $v \neq ?$ and $v \neq null$, then $\exists$ process q such that $v = \vec{\mu}_p^r[q].vote$.
- *FLV-agreement*: If value v is locked in round r, only v or $null$ can be returned.
- *FLV-liveness*: If $\forall q \in \mathscr{C} : \vec{\mu}_p^r[q] \neq \perp$, then $null$ cannot be returned.

FLV-validity and *FLV-agreement* are for safety, while *FLV-liveness* is for liveness. Note that, as FLV is used to find the locked value, its instantiations depend on the T_D parameter (since T_D defines when a value becomes locked).

Starting from Algorithm 5.1 and using parameters T_D and $FLV(\vec{\mu}_p^r)$, we obtain a first draft of our generic consensus algorithm, see Algorithm 5.2. Algorithm 5.2 consists of a sequence of *phases* that can be seen as successive trials to decide on a value. Each phase ϕ consists of two rounds, respectively called *selection round* ($r = 2\phi - 1$) and *decision round* ($r = 2\phi$).

The selection round ($r = 2\phi - 1$) selects a value that will be considered for the decision. Each process p first sends its state ($vote_p$) to all processes. Based on the set of messages received, each honest process selects a value. If any value can be selected (i.e., $FLV(\vec{\mu}_p^r)$ returns ?), the selected value is deterministically chosen among $\vec{\mu}_p^r$. If $FLV(\vec{\mu}_p^r)$ returns neither ? nor $null$, then the returned value is selected. If $FLV(\vec{\mu}_p^r)$ returns $null$, then $vote_p$ is not updated.

The decision round ($r = 2\phi$) determines the conditions that must hold for a process to decide. Each process starts by sending its vote to all processes. A process then decides if it receives a

[5]Variable $vote_p$ is p's estimate of the decision value.

Algorithm 5.2 Generic Algorithm – Draft 1 (boxes represent parameters)

```
1:  Initialization:
2:    $vote_p := init_p$                                   /* value considered for consensus */
3:  Selection Round $r = 2\phi - 1$:
4:    $S_p^r$:
5:      send $\langle vote_p \rangle$ to all
6:    $T_p^r$:
7:      $select_p \leftarrow \boxed{FLV(\vec{\mu}_p^r)}$
8:      if $select_p = ?$ then
9:        $select_p \leftarrow$ choose deterministically a value among the votes received
10:     if $select_p \neq null$ then
11:       $vote_p \leftarrow select_p$
12: Decision Round $r = 2\phi$:
13:   $S_p^r$:
14:     send $\langle vote_p \rangle$ to all
15:   $T_p^r$:
16:     if received at least $\boxed{T_D}$ messages with the same vote $\langle v \rangle$ then
17:       DECIDE $v$
```

threshold number T_D of identical votes. To ensure agreement we require $T_D > \frac{n+b}{2}$ (majority despite Byzantine processes [MR97]), so that two honest processes that decide in the same phase decide the same value.

In order to guarantee that all correct processes eventually decide we rely on the notion of *good phase* (Sect. 5.2). A good phase ensures that all correct processes receive the same set of messages in the selection round, and therefore select the same value.[6] Since all correct processes update $vote$ to the same value, they all decide in the decision round of the good phase.

5.3.3 Generic Algorithm: Draft 2

With Draft 2 we manage to reduce T_D. Remember that $T_D \leq n - b - f$, i.e., $n \geq T_D + b + f$, which means that a smaller T_D leads to a smaller n.

In the decision round of Draft 1 (Algorithm 5.2), the votes sent by honest processes can be different. Therefore, to prevent two honest processes from deciding different values in the same phase, we must have $T_D > \frac{n+b}{2}$. This condition can be relaxed if honest processes would vote for at most one value in a phase—the *validated vote*. In this case, $T_D > b$ is enough, since it ensures that the decision quorum contains at least one honest process.

To ensure that honest processes vote for at most one value in a phase, we need to add one more round to Algorithm 5.2 — the *validation round* — and to introduce the timestamp variable

[6] FLV-liveness ensures that the selected value cannot be $null$.

ts_p. For every process p, the timestamp ts_p represents the most recent phase in which the vote of process p ($vote_p$) has been *validated* in the validation round. The second draft of our generic algorithm is presented as Algorithm 5.3.

The validation round is executed as follows. Each process p first sends the value selected in the selection round ($select_p$), if non-$null$. Based on the set of messages received, each process tries to determine a validated value v. If it observes that a majority despite Byzantine processes (i.e., more than $\frac{n+b}{2}$) have selected the same value v, then $vote_p$ is set to v and ts_p is updated to the current phase number ϕ. This mechanism ensures that all honest processes that validate some value v in phase ϕ, consider the same value. The role of line 24 is explained later.

Algorithm 5.3 Generic Algorithm – Draft 2 (boxes represent parameters)

```
1:  Initialization:
2:    $vote_p := init_p \in V$                          /* value considered for consensus */
3:    $ts_p := 0$                                        /* last phase in which $vote_p$ has been validated */
4:    $history_p := \{(init_p, 0)\}$                     /* updates to the variable $vote_p$ */
5:  Selection Round $r = 3\phi - 2$:
6:    $S_p^r$:
7:      send $\langle vote_p, ts_p, history_p \rangle$ to all
8:    $T_p^r$:
9:      $select_p \leftarrow \boxed{FLV(\vec{\mu}_p^r)}$
10:     if $select_p = ?$ then
11:       $select_p \leftarrow$ choose deterministically a value among the votes received
12:     if $select_p \neq null$ then
13:       $vote_p \leftarrow select_p$
14:       $history_p \leftarrow history_p \cup \{(vote_p, \phi)\}$
15: Validation Round $r = 3\phi - 1$:                    /* executed only if $FLAG = \phi$ */
16:   $S_p^r$:
17:     if $select_p \neq null$ then
18:       send $\langle vote_p \rangle$ to all
19:   $T_p^r$:
20:     if there is a value $v$ such that $|\left\{q \in \Pi : \vec{\mu}_p^r[q] = \langle v \rangle\right\}| > \frac{n+b}{2}$ then
21:       $vote_p \leftarrow v$
22:       $ts_p \leftarrow \phi$
23:     else
24:       $vote_p \leftarrow v$ such that $(v, ts_p) \in history_p$   /* revert the value of $vote_p$ to ensure consistency
          with $ts_p$ */
25: Decision Round $r = 3\phi$:
26:   $S_p^r$:
27:     send $\langle vote_p, ts_p \rangle$ to all
28:   $T_p^r$:
29:     if received at least $\boxed{T_D}$ messages with the same value $\langle v, \boxed{FLAG} \rangle$ then
30:       DECIDE $v$
```

Introducing the validation round and the timestamp variable requires changes in the decision round. In the decision round we need to distinguish two cases: (i) honest processes vote for at most one value (thanks to the validation round), and (ii) honest processes can vote for different values (no validation round) [7]. We introduce the parameter *FLAG* to distinguish between these two cases. In case (i) *FLAG* is an integer (the current phase number); in case (ii) *FLAG* is the special wildcard value $*$. In line 29, if $FLAG = *$ then all votes are taken into account, and the validation round is not needed. Otherwise, $FLAG = \phi$ (current phase number), and only the votes with $ts_p = \phi$ are taken into account.

The $vote_p$ and ts_p variables are not only used within one phase, but also between phases, in order to ensure that if one honest process decides v in phase ϕ, honest processes select v in the selection round of phase $\phi + 1$. In the context of Byzantine faults, we need a mechanism to prove that some value v may have been validated in some previous phase (to filter out invalid votes sent by Byzantine processes). The mechanism is based on an additional variable $history_p$, which is a list of pairs (v, ϕ): each pair denotes that $vote$ has been set to v in the selection round of phase ϕ, i.e., that value v may have been validated in phase ϕ. Variable $history$ is sent together with $vote$ and ts in the selection round, where it is used by the *FLV* function: a pair $(vote, ts)$ is considered valid if at least $b + 1$ processes sent it in their $history$ variable.[8] Furthermore, $history_p$ is updated at line 14 in order for process p to remember what was the value chosen in the selection round of phase ϕ. The $history_p$ is also used at line 24 to revert the value of the $vote_p$ to be consistent with the current value of ts_p in case no value has been validated during the current validation round. [9] In the context of (only) benign faults, variable $history$ can be ignored.

5.3.4 Generic Algorithm: final version

In Algorithm 5.3 in every round all processes send messages to all processes. This can be avoided by introducing the notion of *validator*. Validators are processes that have a special role in the validation round. The intuition is the following: instead of all processes being involved in the selection of a validated vote, this task is devoted to a subset of processes called validators. The question is how to select the validators. We express this through the function *Validator*(p,ϕ)[10] that returns a set of processes $S \subseteq \Pi$ representing the validators for process p in phase ϕ. Note that in the benign fault model, *Validator*(p,ϕ) can be one single process, namely the *leader* process or the process selected by the *rotating coordinator* paradigm.

Validator(p,ϕ) is defined by the following three properties:

[7] In case (ii), an honest process p votes for at most one value v per phase, but another honest process q can vote for v' in the same phase. This is not possible in case (i).

[8] Another solution is to rely on authentication, i.e., to consider the *authenticated Byzantine fault model.*

[9] Note that $vote_p$ is updated both in the selection and the validation round (see line 13 and line 21) with the value selected in the selection round. On the other hand ts_p is updated only during the validation round (line 22) if some value has been validated. Therefore, there is a need for a mechanism at line 24 to ensure that, at the end of the validation round, ts_p represents the last phase in which $vote_p$ has been validated.

[10] As for *FLV*, *Validator* is not really a function. It is rather a problem defined by properties. However, calling it a function is somehow more intuitive.

Algorithm 5.4 Generic Algorithm (boxes represent parameters)

```
1:  Initialization:
2:    $vote_p := init_p$                                   /* value considered for consensus */
3:    $ts_p := 0$                           /* last phase in which $vote_p$ has been validated */
4:    $history_p := \{(init_p, 0)\}$                       /* updates to the variable $vote_p$ */
5:  Selection Round $r = 3\phi - 2$:            /* round in which $\mathcal{P}_{cons}$ must eventually hold */
6:    $S_p^r$:
7:      send $\langle vote_p, ts_p, history_p \rangle$ to all
8:    $T_p^r$:
9:      $select_p \leftarrow \boxed{FLV(\vec{\mu}_p^r)}$
10:     if $select_p = ?$ then
11:       $select_p \leftarrow$ choose deterministically a value among the votes received
12:     if $select_p \neq null$ then
13:       $vote_p \leftarrow select_p$
14:       $history_p \leftarrow history_p \cup \{(vote_p, \phi)\}$
15: Validation Round $r = 3\phi - 1$: /* executed only if $FLAG = \phi$; round in which $\mathcal{P}_{good}$ must eventually
    hold */
16:   $S_p^r$:
17:     if $p \in \boxed{Validator(p,\phi)}$ and $select_p \neq null$ then
18:       send $\langle select_p \rangle$ to all
19:   $T_p^r$:
20:     if there is a value $v$ such that $\left|\left\{q \in \boxed{Validator(p,\phi)} : \vec{\mu}_p^r[q] = \langle v \rangle\right\}\right| > \frac{|\boxed{Validator(p,\phi)}| + b}{2}$
        then
21:       $vote_p \leftarrow v$
22:       $ts_p \leftarrow \phi$
23:     else
24:       $vote_p \leftarrow v$ such that $(v, ts_p) \in history_p$   /* revert the value of $vote_p$ to ensure consistency
          with $ts_p$ */
25: Decision Round $r = 3\phi$:                 /* round in which $\mathcal{P}_{good}$ must eventually hold */
26:   $S_p^r$:
27:     send $\langle vote_p, ts_p \rangle$ to all
28:   $T_p^r$:
29:     if received at least $\boxed{T_D}$ messages with the same value $\langle v, \boxed{FLAG} \rangle$ then
30:       DECIDE $v$
```

- *Validator-validity*:
  If $|Validator(p,\phi)| > 0$ then $|Validator(p,\phi)| > b$.

- *Validator-agreement*:
  $\forall p, q \in \mathcal{H}$ and $\forall \phi$, if $|Validator(p,\phi)| > 0$ and $|Validator(q,\phi)| > 0$, then $Validator(p,\phi) = Validator(q,\phi)$.

- *Validator-liveness*:
  There exists a good phase ϕ_0 such that:
  $\forall p \in \mathcal{C} : |Validator(p,\phi_0) \cap \mathcal{C}| > \frac{|Validator(p,\phi_0)| + b}{2}$.

Validator-validity ensures at least one non-Byzantine validator in the validator set in the presence of Byzantine processes ($b > 0$), while *Validator*-agreement requires that honest processes consider the same non-empty set of validators. Without this requirement, honest processes might consider a different set of validators, and potentially consider a different value as a valid vote. *Validator*-liveness is a non-triviality requirement: without this property returning always empty set would be a possible instantiation of the *Validator* parameter. As for the progress of the generic algorithm it is necessary that correct processes vote for the same value in the decision round, *Validator*-liveness requires that eventually all correct processes consider the same set of validators (non-empty) such that a majority despite Byzantine faults (namely $\frac{|Validator(p,\phi_0)|+b}{2}$) are correct processes.

Introducing $Validator(p,\phi)$ leads to Algorithm 5.4. In the validation round, only validators send messages. Line 20 matches line 20 of Algorithm 5.3. Specifically, expression $\frac{|validators_p|+b}{2}$ of Algorithm 5.4 matches expression $\frac{n+b}{2}$ of Algorithm 5.3. If p observes that a majority despite Byzantine faults have selected the same value v, then v is a validated value, and p sets $vote_p$ to v, and updates its timestamp ts_p to ϕ. Otherwise, the vote is reverted to the value corresponding to ts_p (line 24).[11]

5.3.5 Correctness of the Generic Algorithm

We now prove that the generic algorithm (Algorithm 5.4) solves consensus. Our proof is based on two lemmas.

Lemma 5.1. *If Validator-validity holds, then the following property holds on every honest process h and in every phase ϕ: if process h set $vote_h$ to v and ts_h to ϕ at lines 21-22, then at least one honest process has sent $\langle v\rangle$ at line 18.*

Proof. Assume that a process h set $vote_h$ to v and ts_h to ϕ at lines 21-22. Therefore, $Validator(h,\phi)$ is non empty at line 20 in phase ϕ. By *Validator*-validity, we have $|Validator(h,\phi)| > b$ and $\frac{|Validator(h,\phi)|+b}{2} > b$. Therefore, condition at line 20 can only be true for v if an honest process has sent $\langle v\rangle$ at line 18. □

Lemma 5.2. *In every phase ϕ, if (i) Validator-validity and Validator-agreement hold, (ii) an honest process p updates $vote_p$ to v and ts_p to ϕ, and (iii) another honest process q updates $vote_q$ to v' and ts_q to ϕ (lines 21-22), then $v = v'$.*

Proof. Assume for a contradiction that in some phase ϕ_0 two honest processes p and q have respectively $vote_p = v$ and $ts_p = \phi_0$, and $vote_q = v'$ and $ts_q = \phi_0$. This means that in round $3\phi_0 - 1$ at least $x - b$ honest processes ($x > \frac{|Validator(p,\phi_0)|+b}{2}$) send message $\langle v, -\rangle$ and at least $y - b$ honest processes ($y > \frac{|Validator(q,\phi_0)|+b}{2}$) send message $\langle v', -\rangle$. By *Validator*-agreement we have that $Validator(p,\phi_0) = Validator(q,\phi_0)$. Therefore, $x - b + y - b > |Validator(p,\phi_0)| - b$.

[11] Line 24 is not mandatory, but it allows us to simplify the instantiation of function $FLV(\vec{\mu}^r_p)$.

By *Validator*-validity, it follows that at least one honest process h sent $\langle v, -\rangle$ to one process and $\langle v', -\rangle$ to another process. A contradiction with the fact that h is an honest process.

□

Theorem 5.2. *If (i) function FLV*$(\vec{\mu}_p^r)$ *satisfies FLV-validity and FLV-agreement, (ii) function Validator*(p, ϕ) *satisfies Validator-validity and Validator-agreement, (iii-a)* $FLAG = \phi$ *and* $T_D > b$ *or (iii-b)* $FLAG = *$ *and* $T_D > \frac{n+b}{2}$*, then Algorithm 5.4 ensures weak validity, strong unanimity and agreement.*

Termination holds if (iv) $T_D \leq n - b - f$*, (v) function FLV*$(\vec{\mu}_p^r)$ *satisfies FLV-liveness, and (vi) there is a good phase* ϕ_0 *in which Validator-liveness holds.*

Proof. *(a) Agreement:* Assume for a contradiction that process p decides v in round $r = 3\phi$, and process p' decides $v' \neq v$ in round $r' = 3\phi'$. We consider the following two cases for line 29: $FLAG = *$ and $FLAG = \phi$.

$FLAG = *$: This means that at least T_D processes (at least $T_D - b$ honest) sent $\langle v, \rangle$ in round $r = 3\phi$, and at least T_D processes (at least $T_D - b$ honest) sent $\langle v', \rangle$ in round $r' = 3\phi'$ (*). We have two cases to consider: $\phi = \phi'$ and $\phi > \phi'$.

• $\phi = \phi'$: By (*), $T_D - b$ honest processes sent $\langle v, \rangle$ and $T_D - b$ honest processes sent $\langle v', \rangle$ in round $r = 3\phi$. From (iii-b), $(T_D - b) + (T_D - b) > n - b$. It follows that one honest process h sent $\langle v, \rangle$ to one process and $\langle v', \rangle$ to another process. A contradiction with the fact that h is an honest process.

• $\phi' > \phi$: Let ϕ' be the smallest phase $> \phi$ in which some honest process decides $v' \neq v$. By definition of a locked value, v is locked in all phases $> \phi$. Together with the *FLV*-agreement property, no honest process updates its vote with a value $\hat{v} \neq v$ in the selection round of a phase $> \phi$. Since there is no validation round (i.e., $FLAG = *$), no honest process updates its vote with a value $\hat{v} \neq v$ after phase ϕ. Together with (*), $T_D - b$ honest processes sent $\langle v, \rangle$, and $T_D - b$ honest processes sent $\langle v', \rangle$ in round $r' = 3\phi'$. From (iii-b), $(T_D - b) + (T_D - b) > n - b$. It follows that one honest process h sent $\langle v, \rangle$ to one process and $\langle v', \rangle$ to another process. A contradiction with the fact that h is an honest process.

$FLAG = \phi$: This means that at least T_D processes (at least $T_D - b$ honest) sent $\langle v, \phi\rangle$ in round $r = 3\phi$, and at least T_D processes (at least $T_D - b$ honest) sent $\langle v', \phi'\rangle$ in round $r' = 3\phi'$ (**). We have two cases to consider: $\phi = \phi'$ and $\phi > \phi'$.

• $\phi = \phi'$: By (**), $T_D - b$ honest processes sent $\langle v, \phi\rangle$ and $T_D - b$ honest processes sent $\langle v', \phi\rangle$. From (iii-a), there is an honest process h that validates v (set $vote_h$ to v and ts_h to ϕ) and an honest process h' that validates v' (set $vote_{h'}$ to v' and $ts_{h'}$ to ϕ) at lines 21-22 of round $\hat{r} = 3\phi - 1$. A contradiction with Lemma 5.2 and (ii).

• $\phi' > \phi$: Let ϕ' be the smallest phase $> \phi$ in which some honest process decides $v' \neq v$. By definition of a locked value, v is locked in all phases $> \phi$. By (**), $T_D - b$ honest processes sent $\langle v', \phi\rangle$. From (iii-a), there is an honest process h that validates v' (set $vote_{h'}$ to v' and $ts_{h'}$ to

ϕ) at lines 21-22 of round $\hat{r}' = 3\phi' - 1$. By (ii) and Lemma 5.1, there is an honest process h' that sent $\langle v', - \rangle$ at line 18. Therefore, the function $FLV(\vec{\mu}_p^r)$ returns v' or ? at line 9 on process h'. A contradiction with the *FLV*-agreement property and the fact that v is locked.

(b) Weak Validity: Follows from the *FLV*-validity property and the assumption that all processes are honest.

(c) Strong Unanimity: Unanimity follows from Lemma 5.1 together with (ii), the *FLV*-agreement property, (iii-a) and (iii-b).

(d) Termination: Let ϕ_0 be the good phase in which *Validator*-liveness holds. By $\mathscr{P}_{cons}(3\phi_0 - 2)$, all correct processes receive the same set of messages in round $3\phi_0 - 2$ and therefore select the same value. By *FLV*-liveness and $\mathscr{P}_{cons}(3\phi_0 - 2)$, the value selected cannot be $null$. Thus, at the end of round $r = 3\phi_0 - 2$, all correct processes have the same value for $select_p$, and $vote_p$ (***). Let denote this value with v. We have two cases to consider: $FLAG = *$ and $FLAG = \phi$.

$FLAG = *$: The validation $3\phi_0 - 1$ round is skipped. Together with (***), all correct processes send the same message $\langle v, - \rangle$ at line 27. By $\mathscr{P}_{good}(3\phi_0)$ and (iv), all correct processes receives at least T_D messages $\langle v, - \rangle$ in round $r = 3\phi_0$, and therefore decide.

$FLAG = \phi$: Let us call *validator* any process that is in a set $Validator(p, \phi_0)$ where p is a correct process. By *Validator*-liveness and *Validator*-agreement, all correct processes p consider the same set $Validator(p, \phi_0)$ at line 20. By *Validator*-liveness, the set $Validator(p, \phi_0)$ contains more than $\frac{|Validator(p,\phi_0)|+b}{2}$ correct processes (****). Since $FLAG = \phi$, the validation round $3\phi_0 - 1$ is executed. Together with (****), $\mathscr{P}_{good}(3\phi_0 - 1)$ and lines 20-22, all correct processes update $vote_p$ to v and ts_p to ϕ_0. By $\mathscr{P}_{good}(3\phi_0)$ and (iv), all correct processes receives at least T_D messages $\langle v, \phi_0 \rangle$ in round $r = 3\phi_0$, and therefore decide. □

5.3.6 Optimizations

We point out here several simple optimizations of Algorithm 5.4, to which we will refer later when discussing instantiations of our generic algorithm.

(i) If $FLAG = \phi$, processes in the selection round can send their message only to the processes in the *Validator* set (instead to all processes).

(ii) The selection round can be suppressed in the first phase. As a consequence, if $FLAG = *$ then a decision is possible in one round if all correct processes have the same initial value and $\mathscr{P}_{good}$ holds in the first round. If $FLAG = \phi$, suppressing the selection round in the first phase requires to initialize the variable $selected_p$ to $init_p$.[12]

[12]Note that it is safe to select $init_p$ at the first round for the following reason. If no value is initially locked, then any value may be selected by honest process. If some value v is initially locked, then by definition all honest processes have $init_p = v$, and all honest processes select v.

(iii) If Unanimity is not considered, then the selection round of the first phase can be simplified by having a predetermined process (an initial coordinator) that sends its initial value to all processes. Processes set $selected_p$ to the received value without executing the $FLV(\vec{\mu}_p^r)$ function. If the initial coordinator is a correct process, then all correct processes might set $selected_p$ to the same value, and a decision in possible in the first phase.

(iv) The functionality of the decision round of phase ϕ and of the selection round of phase $\phi+1$ can be provided in one single round.

5.4 Instantiations of Parameters and Classification of Algorithms

We present now instantiations of *FLV* and *Validator*. The *FLV* function is used to find the locked value, therefore depends on the decision mechanism, i.e., on T_D and *FLAG*. We identify three instantiations of the *FLV* function (see Table 5.1). The first one is for the case $FLAG = *$ and $T_D > \frac{n+3b+f}{2}$, and uses only variable $vote_p$; the second one is for the case $FLAG = \phi$ and $T_D > 3b+f$, and uses variables $vote_p$ and ts_p; the last one is for the case $FLAG = \phi$ and $T_D > 2b+f$, and uses all three variables $vote_p$, ts_p and $history_p$. This leads to three classes of consensus algorithms, as shown in Table 5.1. Algorithms that belong to the same class have the same values for the parameters *FLAG* and T_D. Therefore algorithms from the same class have the same constraint on n (follows from $n \geq T_D + b + f$) and have the same number of rounds per phase (depending on the value of *FLAG*). Note that for the first round of each phase, $\mathscr{P}_{cons}$ is required to eventually hold, while for the other rounds of each phase (only) $\mathscr{P}_{good}$ should eventually hold.

One can observe the following tradeoff among these three classes. When $FLAG = *$ and $T_D > \frac{n+3b+f}{2}$ (class 1), only two rounds per phase are needed and the process state is the smallest, but class 1 requires the largest n ($n > 5b+3f$). Table 5.1 lists well-known algorithms that correspond to a given class. These examples are discussed in Section 5.5.

We can make the following comments. First, to the best of our knowledge, no existing algorithm corresponds to class 2, case $f=0$ (Byzantine faults). We call this new algorithm MQB (*Masking Quorum Byzantine* consensus algorithm).[13] Second, Table 5.1 shows that despite its name, the FaB Paxos algorithm does not belong to the same class as the Paxos algorithm.

We now present the three instantiations of the *FLV* function that lead to the three classes of consensus algorithms. Instantiations of the *Validator* function are discussed later.

5.4.1 Instantiations of $FLV(\vec{\mu}_p^r)$

We give here the instantiations of *FLV* for the three classes of algorithm. The differences between the three instantiations have their roots in the use of a different class of Byzantine

[13]The quorums used in this algorithm satisfy the property of *masking quorums* [MR97].

Class	*FLAG*	T_D	n	Variables	Rounds per phase	Examples
1	$*$	$> \frac{n+3b+f}{2}$	$> 5b+3f$	$(vote_p)$	2 $(\mathscr{P}_{cons};\mathscr{P}_{good})$	OneThirdRule [CBS09a] ($b=0$) FaB Paxos [MA06] ($f=0$)
2	ϕ	$> 3b+f$	$> 4b+2f$	$(vote_p, ts_p)$	3 $(\mathscr{P}_{cons};\mathscr{P}_{good};\mathscr{P}_{good})$	Paxos [Lam98] ($b=0$), CT [CT96b] ($b=0$) MR [MR99] ($b=0$), b-DLS [DLS88] ($b=0$) MQB ($f=0$) (*new alg*)
3	ϕ	$> 2b+f$	$> 3b+2f$	$(vote_p, ts_p, history_p)$	3 $(\mathscr{P}_{cons};\mathscr{P}_{good};\mathscr{P}_{good})$	PBFT [CL02] ($f=0$) B-DLS [DLS88] ($f=0$)

Table 5.1: The three classes of consensus algorithms.

quorum systems [MR97]: class 1 uses opaque quorums, class 2 masking quorums and class 3 dissemination quorums. In addition to the *decision quorum*, defined by the decision mechanism, i.e., the parameters T_D and *FLAG*, to simplify description of the *FLV* instantiations, we introduce also the *selection quorum* used in the *FLV* instantiations to select a "safe" value. The complexity of the *FLV* function depends on the *intersection* property of the decision and selection quorums: the instantiation of the *FLV* function of class 1 is the simplest, while the instantiation of the *FLV* function of class 3 is the most complex as it uses the weakest dissemination quorums. The three classes of quorum systems have different requirements on the number of processes (see Table 5.1). Before discussing the *FLV* instantiations, we provide a short background on Byzantine quorum systems.

Byzantine quorum systems

We define a quorum system over a set of processes Π as a collection $\mathcal{Q} = \{Q_1, ..., Q_n\}$ such that

$$\forall Q_1, Q_2 \in \mathcal{Q} : |Q_1 \cap Q_2| > 0$$

Malkhi and Reiter [MR97] proposed three ways of strengthening the basic intersection property to enable quorum systems to be used in systems with Byzantine faults. For the following definitions we assume that there is a (unknown) set B of up to b processes that are Byzantine.

We start with the simplest variant of Byzantine quorum systems called *dissemination quorums*, where the intersection of any two quorums contains at least one honest process:

$$|Q_1 \cap Q_2 \setminus B| > 0$$

We illustrate the dissemination quorums on Figure 5.1.

Masking quorum systems are second alternative, where the intersection of any two quorums contains more honest processes than the number of Byzantine processes in either quorum:

$$|Q_1 \cap Q_2 \setminus B| > |Q_2 \cap B|$$

We illustrate the masking quorums on Figure 5.2.

Finally, with *opaque quorum systems* the number of honest processes in the intersection of any two quorums Q_1 and Q_2 exceeds the number of Byzantine processes together with the number of processes in Q_2 but not in Q_1:

$$|Q_1 \cap Q_2 \setminus B| > |(Q_1 \cap B) \cup (Q_2 \setminus Q_1)|$$

We illustrate the opaque quorums on Figure 5.3.

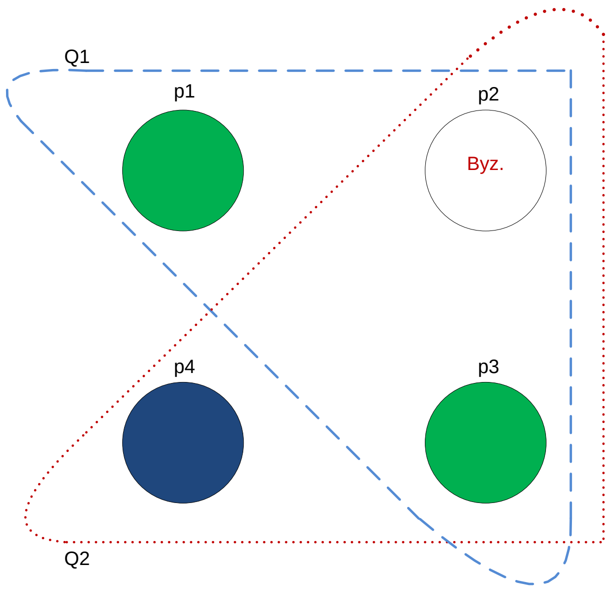


Figure 5.1: Dissemination quorums. The system consists of four processes p_1 to p_4, p_2 is a Byzantine process.

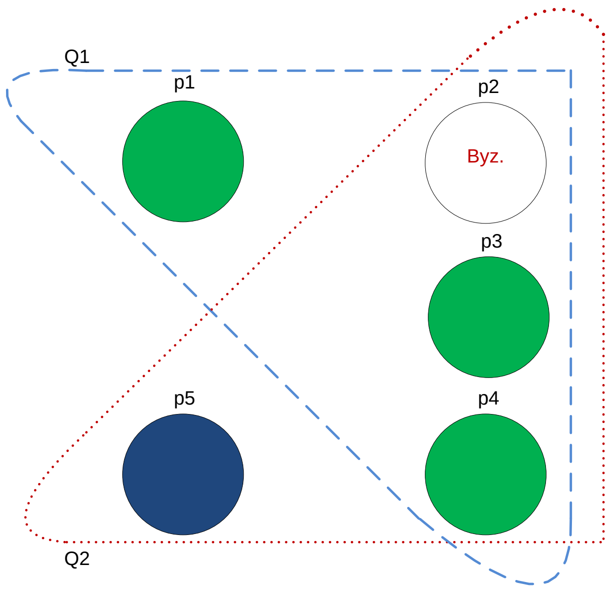


Figure 5.2: Masking quorums. The system consists of five processes p_1 to p_5, p_2 is a Byzantine process.

$FLV(\vec{\mu}_p^r)$ **for class 1**

We start with the *FLV* function for class 1 ($FLAG = *$ and $T_D > \frac{n+3b+f}{2}$), see Algorithm 5.5. The selection quorum is any set of processes of size bigger than $2(n - T_D + b)$.

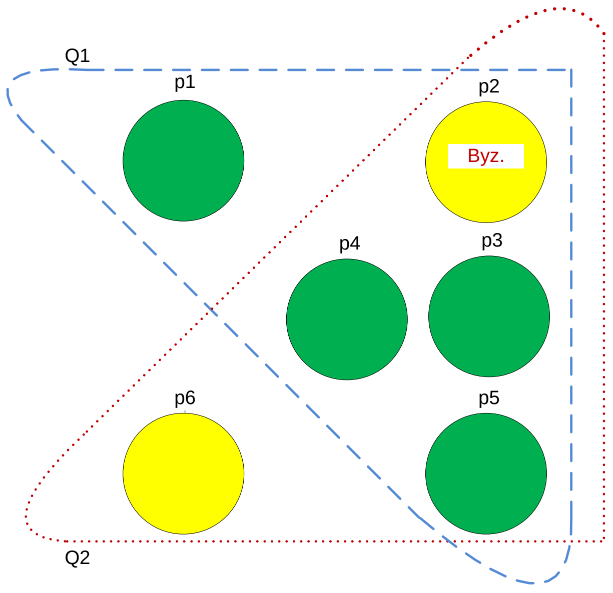


Figure 5.3: Opaque quorums. The system consists of six processes p_1 to p_6, p_2 is a Byzantine process.

Algorithm 5.5 $FLV(\vec{\mu}_p^r)$ for class 1

```
1: $correctVotes_p \leftarrow \left\{v : \left|\left\{(v,-,-) \in \vec{\mu}_p^r\right\}\right| > n - T_D + b\right\}$
2: if $|correctVotes_p| = 1$ then
3:     return $v$ s.t. $v \in correctVotes_p$
4: else if $|\vec{\mu}_p^r| > 2(n - T_D + b)$ then
5:     return ?
6: else
7:     return $null$
```

Algorithms of class 1 relies on *opaque* quorums. Therefore, if some value v_1 is stored by the decision quorum (because some honest process decided v_1), then in any selection quorum, the votes with the value v_1 always outnumbers votes sent by Byzantine processes and votes (potentially different than v_1) sent by honest processes not part of the decision quorum (see Figure 5.3 where v_1 is denoted with *green* color).

We now explain Algorithm 5.5 in more details. Line 1 of Algorithm 5.5 is for *FLV*-agreement as we now explain with a simple example. Let v_1 be locked in round r because some honest process p has decided v_1 in round $r-1$. By Algorithm 5.4, p has received in the decision round $r-1$ at least T_D votes v_1. At least $T_D - b$ votes v_1 are from honest processes (see $\textcircled{1}$ at Figure 5.4), i.e., a process can receive at most $n-(T_D-b)$ votes equal to $v_2 \neq v_1$ (*) (see $\textcircled{2}$ at Figure 5.4). Therefore, the condition of line 1 can only hold for v_1, i.e., among the values different from ? and $null$, *FLV* can only return v_1. For *FLV*-agreement to hold, Algorithm 5.5 must also prevent ? to be returned when v_1 is locked. The condition of line 4 ensures this. Here is why. Assume that the condition of line 4 holds. This means that $\vec{\mu}_p^r$ contains more

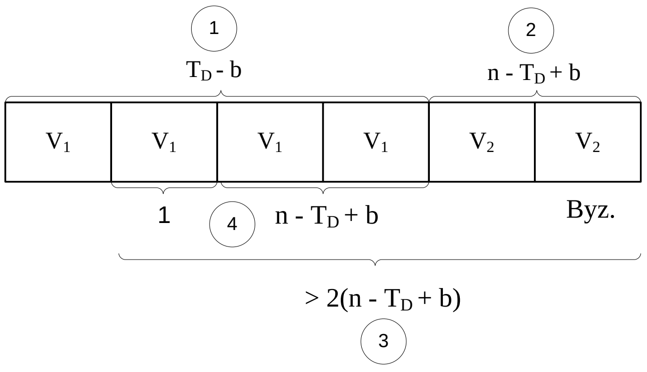


Figure 5.4: Illustration of *FLV* for class 1 with v_1 locked ($n = 6$, $b = 1$, $f = 0$, $T_D = 5$)

than $2(n - T_D + b)$ messages (see (3) at Figure 5.4). With (*), $\vec{\mu}_p^r$ contains more than $n - T_D + b$ messages equal to v_1 (see (4) at Figure 5.4). By line 1, we have $v_1 \in correctVotes_p$, and as shown above, only v_1 can be in $correctVotes_p$. Therefore, the condition of line 2 holds: Algorithm 5.5 cannot return ? when v_1 is locked.

Property *FLV*-liveness is ensured by lines 4 and 5 of Algorithm 5.5. This is because when $T_D > \frac{n+3b+f}{2}$, we have $n - b - f > 2(n - T_D + b)$. Therefore, receiving a message from all correct processes (i.e., $|\vec{\mu}_p^r| \geq n - b - f$) implies that the condition of line 4 holds, i.e., $null$ is not returned. Property *FLV*-validity is trivially ensured by lines 1-3.

Theorem 5.3. *If FLAG* $= *$, *then Algorithm 5.5 ensures FLV-validity and FLV-agreement. Moreover, FLV-liveness holds if* $T_D > \frac{n+3b+f}{2}$.

Proof.
FLV-validity: FLV-validity follows from lines 1-3.

FLV-agreement: Let $r = 3\phi - 2$ be the smallest selection round in which value v is locked. By definition of a locked value, we have two cases to consider (1) all honest processes have $vote_p = v$ and unanimity must be ensured (and $r = 1$), or (2) v has been decided in round $r' = 3\phi - 3$ by some honest process p. We now show that for both cases at least $T_D - b$ honest processes sent $\langle v, -, - \rangle$ in round r (*).

Case 1: Trivially follows from initialization and $T_D \leq n - b - f$.

Case 2: By Algorithm 5.4, the process p received at least T_D messages $\langle v, - \rangle$ in round r'. Therefore, at least $T_D - b$ honest processes send $\langle v, - \rangle$ in round r', i.e., at least $T_D - b$ honest processes have their vote set to v, and send $\langle v, -, - \rangle$ in round r.

We now show that when property (*) holds, Algorithm 5.5 ensures *FLV*-agreement. Assume for the contradiction that a non null value $v' \neq v$ is returned. Two cases must be considered.

v' is returned at line 3: Because $correctVotes_p$ is not empty, the set $\vec{\mu}_p^r$ contains more than $n - T_D + b$ messages $\langle v', -, - \rangle$. A contradiction with (*).

? is returned at line 5: This means that $\vec{\mu}_p^r$ contains more than $2(n - T_D + b)$ messages. By (*), $\vec{\mu}_p^r$ contains at most $n - T_D + b$ messages $\langle v' \neq v, -, - \rangle$, and therefore, more than $n - T_D + b$ messages $\langle v, -, - \rangle$. By line 1, the set $correctVotes_p$ is not empty. By lines 2-3, value v is returned. A contradiction.

This shows that Algorithm 5.5 ensures *FLV*-agreement in round r. Therefore, no honest process p updates its variable $vote_p$ and $select_p$ to a value $v' \neq v$ in selection round r. Because *FLAG* $= -$, the validation round is skipped. Therefore, property (*) holds in selection round $r'' = 3\phi + 1$. With similar arguments as above, we can show that Algorithm 5.5 ensures *FLV*-agreement in round r''. By a simple induction on ϕ, we can show that Algorithm 5.5 ensures *FLV*-agreement in all rounds.

FLV-liveness: Property *FLV*-liveness is ensured by lines 4-5. This is because when $T_D > \frac{n+3b+f}{2}$, we have $n - b - f > 2(n - T_D + b)$. Therefore, receiving messages from all correct processes (i.e., $|\vec{\mu}_p^r| \geq n - b - f$) implies that the condition of line 4 holds. □

FLV($\vec{\mu}_p^r$) **for class 2**

For class 2 we have $T_D \leq \frac{n+3b+f}{2}$ $(n < 5b + 3f)$, which means that opaque quorums cannot be used: the locked value does not necessarily outnumber other values. Therefore, an additional mechanism is needed: the timestamp ts_p. By considering a vote and timestamp pair (instead of only vote), it is possible to ignore "outdated" votes sent by honest processes. As the validation round ensures only one valid vote per phase, the locked value can only be the vote with the highest timestamp. This mechanism can be denotes as the "last voting rule" [CBS09a].

However, as a Byzantine process can always send a vote with an arbitrary large timestamp, it is necessary to authenticate the vote-and-timestamp pairs. Algorithms of class 2 relies on the properties of *masking* quorums to authenticate vote-and-timestamp pairs. As with the masking quorums, the intersection of any two quorums contains more honest processes than Byzantine processes in either quorum, an honest process considers a vote-and-timestamp pair as valid only if it is received from at least $b + 1$ processes. Therefore, once an honest process decides some value v_1 in phase ϕ_1, any selection quorum (any set of processes of size bigger than $n - T_D + 2b$) contains at least $b + 1$ messages with $vote = v_1$ and $ts = \phi_1$.

The *FLV* function for class 2 (*FLAG* $= \phi$ and $T_D > 3b + f$) is shown in Algorithm 5.6.

Lines 1 (where $\{\#\ldots\#\}$ denotes a multiset) and 2 Algorithm 5.6 are for *FLV*-agreement, as we

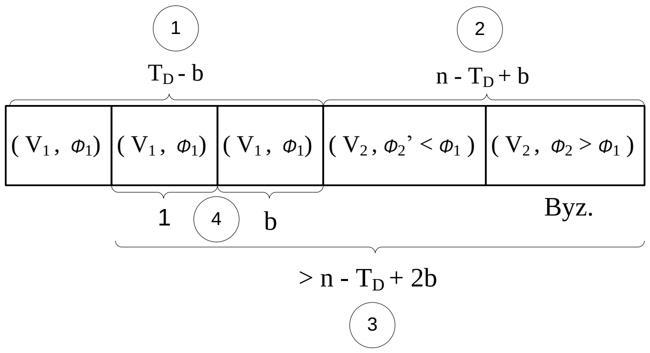


Figure 5.5: Illustration of *FLV* for class 2 with v_1 locked ($n = 5$, $b = 1$, $f = 0$, $T_D = 4$)

Algorithm 5.6 *FLV*$(\vec{\mu}_p^r)$ for class 2

```
1: $possibleVotes_p \leftarrow \{\#(vote, ts, -) \in \vec{\mu}_p^r :$
     $|\{(vote', ts', -) \in \vec{\mu}_p^r : vote = vote' \lor ts > ts'\}|$
                                          $> n - T_D + b\#\}$
2: $correctVotes_p \leftarrow \{(vote,-,-) \in possibleVotes_p :$
     $|\{\#(vote',-,-) \in possibleVotes_p : vote = vote'\#\}| > b\}$
3: if $|correctVotes_p| = 1$ then
4:     return $v$ s.t. $(v, -, -) \in correctVotes_p$
5: else if $|\vec{\mu}_p^r| > n - T_D + 2b$ then
6:     return ?
7: else
8:     return $null$
```

now explain with the simple example. Let v_1 be locked in round r, phase $\phi_1 + 1$, because some honest process p has decided v_1 in round $r-1$, phase ϕ_1. By Algorithm 5.4, p has received in the decision round $r-1$ at least T_D messages $\langle v_1, \phi_1 \rangle$. At least $T_D - b$ messages are from honest processes that have $vote_p = v_1$ and $ts_p = \phi_1$ (see $\textcircled{1}$ at Figure 5.5), i.e., at most $n - b - (T_D - b) = n - T_D$ honest processes have $vote_p = v_2 \neq v_1$ (*) (see $\textcircled{2}$ at Figure 5.5). Because only one value can be validated by honest processes in phase ϕ_1 (Lemma 5.2), all honest processes with $vote_p = v_2 \neq v_1$ have $ts_p < \phi_1$. It follows that for every honest process p, we have $vote_p = v_1$ or $ts_p < \phi_1$ (**). Together with (*), no message $\langle v_2 \neq v_1, -, - \rangle$ sent by an *honest* process can satisfy the condition of line 1. In other words, the set $possibleVotes_p$ may contain at most b messages $\langle v_2 \neq v_1, -, - \rangle$, namely the messages sent by Byzantine processes. Line 2 prevents such messages to be in $correctVotes_p$. This shows that among the values different from ? and $null$, only v_1 can be returned.

For *FLV*-agreement to hold, Algorithm 5.6 must also prevent ? to be returned when v_1 is locked. The condition of line 5 ensures this. Here is why. Assume that the condition of line 5 holds. This means that $\vec{\mu}_p^r$ contains more than $n - T_D + 2b$ messages (see $\textcircled{3}$ at Figure 5.5). From (*), a process can receive at most $n - T_D + b$ messages with $vote = v_2 \neq v_1$. It follows that the set $\vec{\mu}_p^r$ contains at least $b+1$ messages $\langle v_1, \phi_1, - \rangle$ from honest processes (see $\textcircled{4}$ at Figure 5.5). With (**) and the fact that $\vec{\mu}_p^r$ contains more than $n - T_D + b$ messages from honest processes (see Figure 5.5), the $b+1$ messages $\langle v_1, \phi_1, - \rangle$ satisfy the condition of line 1.

By line 2, $\langle v_1, \phi_1, - \rangle$ is in $correctVotes_p$. Moreover, as discussed above, only v_1 can be in $correctVotes_p$. Therefore, the condition of line 3 holds: Algorithm 5.6 cannot return ? when v_1 is locked.

Property *FLV*-liveness is ensured by lines 5 and 6 of Algorithm 5.6. This is because when $T_D > 3b + f$, we have $n - b - f > n - T_D + 2b$. Therefore, receiving a message from all correct processes (i.e., $|\vec{\mu}_p^r| \geq n - b - f$) ensures that the condition of line 5 holds, i.e., $null$ cannot be returned. Property *FLV*-validity is trivially ensured by lines 1-4.

Theorem 5.4. *If FLAG* $= \phi$, *Validator*(p, ϕ)*-validity and Validator*(p, ϕ)*-agreement hold, then Algorithm 5.6 ensures FLV-validity and FLV-agreement. FLV-liveness holds if in addition* $T_D > 3b + f$.

Proof.
FLV-validity: FLV-validity follows from lines 1-4.

FLV-agreement: Let $r = 3\phi - 2$ be the smallest selection round in which value v is locked. By definition of a locked value, we have two cases to consider (1) all honest processes have $vote_p = v$ and unanimity must be ensured (and $r = 1$), or (2) v has been decided in round $r' = 3\phi - 3$ by some honest process p. We now show that for both cases in round r at least $T_D - b$ honest processes sent $\langle v, \hat{\phi} \geq \phi - 1, - \rangle$ (*), and for all honest processes q, we have $(vote_q = v \vee ts_q < \phi - 1)$ (**).

Case 1: Trivially follows from initialization and $T_D \leq n - b - f$.

Case 2: By Algorithm 5.4, the process p received at least T_D messages $\langle v, \phi - 1 \rangle$ in round r'. Therefore, at least $T_D - b$ honest processes send $\langle v, \phi - 1 \rangle$ in round r', i.e., at least $T_D - b$ honest processes have their vote set to v, and send $\langle v, \phi - 1, - \rangle$ in round r (which shows (*)). This means that an honest process p updates $vote_p$ to v and ts_p to ϕ in the validation round of phase $\phi - 1$. By *Validator*(p, ϕ)-validity, *Validator*(p, ϕ)-agreement and Lemma 5.2, no honest process update its vote to a value $v' \neq v$ in this validation round (which shows (**)).

We now show that when properties (*) and (**) hold, Algorithm 5.6 ensures *FLV*-agreement. Assume for the contradiction that a non null value $v' \neq v$ is returned. Two cases must be considered.

v' is returned at line 4: Because $correctVotes_p$ is not empty, the multi-set $possibleVotes_p$ contains more than b messages $\langle v', -, - \rangle$. It follows that an honest process sent a message $\langle v', \phi', - \rangle$. By (**), $\phi' < \phi - 1$. By line 1, more than $n - T_D$ honest processes sent a message $\langle v'', \phi'', - \rangle$ with $v'' = v'$ or $\phi'' < \phi' < \phi - 1$. A contradiction with (*).

? is returned at line 6: This means that $\vec{\mu}_p^r$ contains more than $n - T_D + 2b$ messages (and more than $n - T_D + b$ messages from honest processes (***)). By (*), $\vec{\mu}_p^r$ contains at most $n - T_D + b$ messages different from $\langle v, \hat{\phi} \geq \phi - 1, - \rangle$, and therefore, more than b messages $\langle v, \hat{\phi} \geq \phi - 1, - \rangle$. By line 1, (**) and (***), the multi-set $possibleVotes_p$ contains more than b messages $\langle v, -, - \rangle$. Therefore, the set $correctVotes_p$ contains a message $\langle v, -, - \rangle$. By lines 3-4, value v is returned. A contradiction.

This shows that Algorithm 5.6 ensures *FLV*-agreement in round r. Therefore, no honest process p updates its variable $vote_p$ and $select_p$ to a value $v' \neq v$ in selection round r. By *Validator*(p, ϕ)-validity and Lemma 5.1, no honest process p updates its vote to a value $v' \neq v$ in the validation round $r+1$. Therefore, properties (*) and (**) hold in selection round $r'' = 3\phi + 1$. With similar arguments as above, we can show that Algorithm 5.6 ensures *FLV*-agreement in round r''. By a simple induction on ϕ, we can show that Algorithm 5.6 ensures *FLV*-agreement in all rounds.

FLV-liveness: Property *FLV*-liveness follows from lines 5 and 6. This is because when $T_D > 3b + f$, we have $n - b - f > n - T_D + 2b$. Therefore, receiving messages from all correct processes (i.e., $\vec{\mu}_p^r \geq n - b - f$) ensures that the condition of line 5 holds. □

FLV$(\vec{\mu}_p^r)$ for class 3

The *FLV*$(\vec{\mu}_p^r)$ function for class 3 also relies on the "last Voting" mechanism. However, for class 3 we have $T_D \leq 3b + f$ $(n < 4b + 2f)$, which means that the masking quorums cannot be used and a different mechanism for authenticating vote-and-timestamp pairs is needed. The idea is to use the $history$ log. As explained in Section 5.3.3, a pair $(vote, ts)$ is considered valid if at least $b + 1$ processes have it in their $history$ variable.

The *FLV* function for class 3 (*FLAG* $= \phi$ and $T_D > 2b + f$) is shown in Algorithm 5.7. The selection quorum is any set of processes of size bigger than $n - T_D + b$. Algorithms of class 3 rely on *dissemination* quorums.

Similarly to Algorithm 5.6, lines 1 and 2 of Algorithm 5.7 are for *FLV*-agreement, as we now explain with a simple example. Let v_1 be locked in round r of phase $\phi_1 + 1$, because some honest process p has decided v_1 in round $r - 1$ of phase ϕ_1. Consider Figure 5.6. For the same reason as for Algorithm 5.6, at least $T_D - b$ honest processes have $vote_p = v_1$ and $ts_p = \phi_1$ (*), as shown by $\textcircled{1}$ at Figure 5.6. Furthermore, at most $n - T_D$ honest processes have $vote_p = v_2 \neq v_1$ (see $\textcircled{2}$ at Figure 5.6). As for class 2, for every honest process p, we have $vote_p = v_1$ or $ts_p < \phi_1$ (**). Together with (*), no message $\langle v_2 \neq v_1, -, - \rangle$ sent by an *honest* process can satisfy the condition of line 1(see Figure 5.6). Said differently, apart from messages $\langle v_1, -, - \rangle$, only messages $\langle v_2 \neq v_1, \phi_2, - \rangle$ sent by Byzantine processes can be in the set $possibleVotes_p$. Because honest processes can only update history at line 14 of Algorithm 5.4, no honest process has a pair $(-, \phi_2 > \phi_1)$ in its history in the sending step of

Algorithm 5.7 *FLV*$(\vec{\mu}_p^r)$ for class 3

```
1: $possibleVotes_p \leftarrow \{(vote, ts, -) \in \vec{\mu}_p^r :$
      $|\{(vote', ts', -) \in \vec{\mu}_p^r : vote = vote' \vee ts > ts'\}|$
                                                    $> n - T_D + b\}$
2: $correctVotes_p \leftarrow \{v : (v, ts, -) \in possibleVotes_p \wedge$
      $|\{(vote', ts', history') \in \vec{\mu}_p^r : (v, ts) \in history'\}| > b\}$
3: if $|correctVotes_p| = 1$ then
4:    return $v$ $s.t.$ $(v, -, -) \in correctVotes_p$
5: else if $|correctVotes_p| > 1$ then
6:    return ?
7: else if $|\left\{(vote, ts, -) \in \vec{\mu}_p^r : ts = 0\right\}| > n - T_D + b$ then
8:    if there is a value $v$ such that $\vec{\mu}_p^r$ contains a majority of messages $(v,-,-)$ then      /* only for unanimity */
9:       return $v$
10:   else
11:      return ?
12: else
13:   return $null$
```

round r. It follows that only messages $\langle v_1, -, -\rangle$ can be in $correctVotes_p$ at line 2. Therefore, when a value v_1 is locked, lines 1 and 2 prevent any value $v \neq v_1$ or ? to be returned at lines 4 and 6. By (*) together with $\phi_1 > 0$, condition of line 7 never holds in our example.

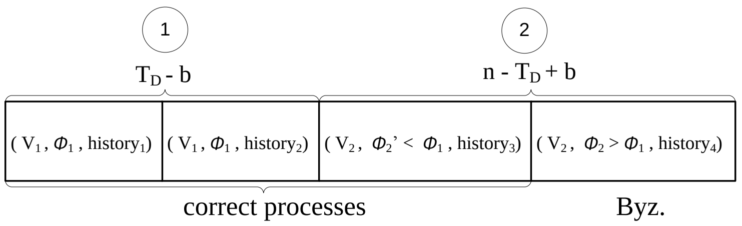


Figure 5.6: Illustration of *FLV* for class 3 ($n = 4$, $b = 1$, $f = 0$, $T_D = 3$)

The role of lines 8-11 is to ensure *FLV*-agreement when strong unanimity is considered. Let all honest processes have initially $vote_p = v_1$. With the same arguments as above, it follows that no value different from v_1 can be in $correctVotes_p$ at line 2, and therefore no value different than v_1 cannot be returned at line 4. As $correctVotes_p$ contains at most one value, the condition of line 5 is never `true` so ? is never returned at line 6. However, the condition of line 7 might hold. In this case, $\vec{\mu}_p^r$ contains more than $n - T_D$ messages $\langle v_1, 0, -\rangle$ from honest processes, and at most b messages $\langle v_2 \neq v_1, 0, -\rangle$ from Byzantine processes. Because $T_D \leq n - b - f$, we have $n - T_D \geq b$, and v_1 is returned at line 9, which ensures *FLV*-agreement.

Let us now discuss *FLV*-liveness. For this property to hold, we need a stronger variant of *Validator-validity*:[14]

[14]This stronger variant was not introduced in Section 5.3.4, since the proof of the generic Algorithm 5.4 does not require the stronger variant. In the proof of Algorithm 5.4, the stronger variant is hidden in the *FLV-liveness* property.

- *Validator-$strongValidity$*:
  If $|Validator(p,\phi)| > 0$ then $|Validator(p,\phi)| > 3b$.

With *Validator-$strongValidity$* the following Lemma 5.1 holds:

Lemma 5.3. *If Validator-$strongValidity$ holds, then the following property holds on every honest process h and in every phase ϕ: if process h set $vote_h$ to v and ts_h to ϕ at lines 21-22 of Algorithm 5.4, then at least $b+1$ honest process has sent $\langle v\rangle$ at line 18.*

Proof. Assume that a process h set $vote_h$ to v and ts_h to ϕ at lines 21-22 of Algorithm 5.4. Therefore, $Validator(h,\phi)$ is non empty at line 20 in phase ϕ. By *Validator-$strongValidity$*, we have $|Validator(h,\phi)| > 3b$ and $\frac{|Validator(h,\phi)|+b}{2} > 2b$. Therefore, condition at line 20 can only be true for v if an at least $b+1$ honest process has sent $\langle v\rangle$ at line 18. □

We now explain how Algorithm 5.7 ensures *FLV*-liveness when *Validator-$strongValidity$* holds. Let $\vec{\mu}_p^r$ contains the messages from all the $n-b-f$ correct processes. There are two cases to consider: (1) correct processes sent only $\langle -,0,-\rangle$, (2) at least one correct process sent $\langle -,ts>0,-\rangle$. Note that $T_D > 2b+f$ ensures $n-b-f > n-T_D+b$ (*). In case (1), by (*) the condition of line 7 holds, and $null$ cannot be returned at line 13 of Algorithm 5.7. In case (2), let v denote the subset of messages in $\vec{\mu}_p^r$ that are from correct processes, and let ts_{v} be the highest timestamp in v. By Lemma 5.2 there is a unique value v_{v} such that $\langle v_{\mathsf{v}}, ts_{\mathsf{v}}, -\rangle \in \mathsf{v}$. Together with (*), this ensures that the set $possibleVotes_p$ is not empty, and contains $\langle v_{\mathsf{v}}, ts_{\mathsf{v}}, -\rangle$. By Lemma 5.3, any correct process that validates v_{v} in the validation round $3\,ts_{\mathsf{v}}-1$ received v_{v} from at least $b+1$ correct processes. Therefore, at least $b+1$ correct processes have selected v_{v} in round $3\,ts_{\mathsf{v}}-2$, and these processes have $(v_{\mathsf{v}}, ts_{\mathsf{v}})$ is their history. This implies that the set $correctVotes_p$ is non empty, and a non-$null$ value is returned at line 4 or 6, which ensures *FLV*-liveness.

Theorem 5.5. *If FLAG = ϕ, Validator(p,ϕ)-validity and Validator(p,ϕ)-agreement hold, then Algorithm 5.7 ensures FLV-validity and FLV-agreement. FLV-liveness holds if in addition $T_D > 2b+f$ and Validator-strongValidity holds.*

Proof.
FLV-validity: *FLV*-validity follows from the lines 1-4 and 8-9.

FLV-agreement: Let $r = 3\phi-2$ be the smallest selection round in which value v is locked. By definition of a locked value, we have two cases to consider (1) all honest processes have $vote_p = v$ and unanimity must be ensured (and $r=1$), or (2) v has been decided in round $r' = 3\phi-3$ (and thus, $\phi-1 \geq 1$) by some honest process p. We now show that for both cases in round r at least $T_D - b$ honest processes sent $\langle v, \hat{\phi} \geq \phi-1, -\rangle$ (*), for all honest processes q, we have $(vote_q = v \vee ts_q < \phi-1)$ (**), and for any element $(vote, ts)$ in the set $history_q$ of any honest process q, we have $(ts \le \phi - 1)$ (***). In addition, if less than $T_D - b$ honest processes have $ts_p > 0$, then all honest processes have $vote_p = v$ (****).

Case 1: Trivially follows from initialization and $T_D \le n - b - f$.

Case 2: By Algorithm 5.4, the process p received at least T_D messages $\langle v, \phi - 1\rangle$ in round r'. Therefore, at least $T_D - b$ processes send $\langle v, \phi - 1\rangle$ in round r', i.e., at least $T_D - b$ honest processes have their vote set to v, and send $\langle v, \phi - 1, -\rangle$ in round r (which shows (*)). This means that an honest process p updates $vote_p$ to v and ts_p to ϕ in the validation round of phase $\phi - 1$. By *Validator*(p,ϕ)-validity, *Validator*(p,ϕ)-agreement and Lemma 5.2, no honest process update its vote to a value $v' \neq v$ in this validation round (which shows (**)).

Property (***) trivially follows from the fact that for each honest process the last update of history occured in round $3(\phi - 1) - 2$. Property (****) trivially follows from $\hat{\phi} \ge \phi - 1 \ge 1$ and (*), which implies that the precondition of (****) cannot be true.

We now show that when properties (*), (**), (***) and (****) hold, Algorithm 5.7 ensures *FLV*-agreement. Assume for the contradiction that a non null value $v' \neq v$ is returned. Four cases must be considered.

v' is returned at line 4: Because $correctVotes_p$ is not empty, the set $\vec{\mu}_p^r$ contains a message $\langle v', \phi', -\rangle$ in $possibleVotes_p$ such that an honest process h has (v', ϕ') in $history_h$ (see line 2). By (***), $\phi' \le \phi - 1$. By line 1, more than $n - T_D$ honest processes sent a message $\langle v'', \phi'', -\rangle$ with $v'' = v'$ or $\phi'' < \phi' \le \phi - 1$. A contradiction with (*).

? is returned at line 6: Same arguments as the case v' is returned at line 4.

v' is returned at line 9: By line 7, the set $\vec{\mu}_p^r$ contains more than $n - T_D + b$ messages $\langle -, 0, -\rangle$. Therefore, less than $T_D - b$ honest processes has $ts_p > 0$. By (****), all honest processes has $vote_p = v$. Therefore $\vec{\mu}_p^r$ contains more than $n - T_D$ messages $\langle v, 0, -\rangle$ and at most b messages $\langle v', 0, -\rangle$. Because $T_D \le n - b - f$, there is a majority of messages $\langle v, 0, -\rangle$ in $\vec{\mu}_p^r$. A contradiction with line 8 and the fact that v' is returned at line 9.

? is returned at line 11: Same arguments as the case v' is returned at line 9.

This shows that Algorithm 5.7 ensures *FLV*-agreement in round r. Therefore, no honest process p updates its variable $vote_p$ and $select_p$ to a value $v' \neq v$ in selection round r. Furthermore, no honest process p adds a tuple $(v' \neq v, \phi)$ in selection round r. By *Validator*(p,ϕ)-validity and Lemma 5.1, no honest process p updates its vote to a value $v' \neq v$ in the validation round $r+1$. Therefore, properties (*), (**), (***) and (****) hold in selection round $r'' = 3\phi + 1$. With similar arguments as above, we can show that Algorithm 5.7 ensures *FLV*-agreement in round r''. By a simple induction on ϕ, we can show that Algorithm 5.7 ensures *FLV*-agreement in all rounds.

FLV-liveness: Let $\vec{\mu}_p^r$ contain the messages from all the $n-b-f$ correct processes. There are two cases to consider: (1) correct processes sent only $\langle -, 0, - \rangle$, (2) at least one correct process sent $\langle -, ts > 0, - \rangle$. Note that $T_D > 2b+f$ ensures $n-b-f > n-T_D+b$ (*). In case (1), by (*) the condition of line 7 holds, and $null$ cannot be returned at line 13. In case (2), let S denote the subset of messages in $\vec{\mu}_p^r$ that are from correct processes, and let ts_S be the highest timestamp in S. By *Validator*(p,ϕ)-validity, *Validator*(p,ϕ)-agreement and Lemma 5.2, there is a unique value v_S such that $\langle v_S, ts_S, - \rangle \in S$. Together with (*), this ensures that the set $possibleVotes_p$ is not empty, and contains $\langle v_S, ts_S, - \rangle$. *Validator*-strongValidity ensures that $|validators_p| > 0$ implies $|validators_p| > 3b$. As a result, any correct process that validates v_S in the validation round $3\,ts_S - 1$ received $\langle v_S, - \rangle$ from more than $\frac{(3b)+b}{2} = 2b$ processes. Therefore, at least $b+1$ correct processes have selected v_S in round $3\,ts_S - 2$, and these processes have (v_S, ts_S) in their history. This implies that the set $correctVotes_p$ is non empty, and a non-$null$ value is returned at line 4 or 6. □

5.4.2 Instantiations of *Validator*(p,ϕ)

A trivial instantiation of the *Validator* function consists in always returning the whole set of processes Π. This trivially satisfies *Validator*-validity, *Validator*-strongValidity, *Validator*-agreement and *Validator*-liveness. To our knowledge, this instantiation is used in all algorithms for Byzantine faults. However, another possible instantiation can be considered in the Byzantine fault model: it consists in returning the same set S of $b+1$ processes at every process, e.g., S defined by a deterministic function of the phase ϕ.[15]

In the benign fault model, it is sufficient that the *Validator* function always returns a single process rather than a set of processes. One such instantiation is the well known *rotating coordinator* function used in [CT96b]. Another example involves message exchange (these messages can be piggybacked on existing messages). In each phase every process chooses a potential validator q and informs q by sending him a message. If some process q receives messages from a majority, then q becomes the validator, and q informs all processes that the output of *Validator* function is q. If a process does not receive such a message within some timeout, the *Validator* function returns $\emptyset$. It can easily be shown that this instantiation satisfies *Validator*-validity, *Validator*-agreement and *Validator*-liveness.[16] We call this instantiation *majority voting validator selection*; it is used for example in the *prepare phase* of Paxos [Lam98].

[15]Note that this instantiation does not satisfy *Validator*-strongValidity.

[16]For *Validator*-liveness to hold, it is necessary to have a phase in which all correct processes choose the same validator.

5.5 Instantiations of Algorithm 5.4

In this section we show several well-known consensus algorithms derived from Algorithm 5.4. Note that even though the instantiated algorithms are expressed in the round model, which is not the case of many well-known consensus algorithms, the core mechanisms are the same.[17]

5.5.1 Class 1 algorithms

OneThirdRule [CBS09a] The OneThirdRule algorithm assumes benign faults only ($b = 0$) and requires $n > 3f$ to tolerate f benign faults.

The instantiated version of the OneThirdRule algorithm (that we call *Inst-OneThirdRule*), is obtained from Algorithm 5.4 with the following parametrization: $T_D = \lceil \frac{2n+1}{3} \rceil$,[18] $FLAG = *$ and Algorithm 5.5 with $T_D = \lceil \frac{2n+1}{3} \rceil$ as the *FLV* instantiation.

Algorithm 5.8 OneThirdRule algorithm ($n > 3f$) [CBS09a]

```
1: Initialization:
2:   $vote_p := init_p$
3: Round $r$:
4:   $S_p^r$:
5:     send $\langle vote_p \rangle$ to all
6:   $T_p^r$:
7:     if received more than $2n/3$ messages then
8:       $x_p :=$ the smallest most often received value
9:       if more than $2n/3$ values received are equal to $v$ then
10:        DECIDE $v$
```

We now compare Inst-OneThirdRule with the original algorithm (Algorithm 5.8). In Algorithm 5.8, the functionality of the selection round and of the decision round are merged into one single round (see optimization (iv), Sect. 5.3.6). With $T_D = \lceil \frac{2n+1}{3} \rceil$, it is easy to see that the condition for deciding is the same in the two algorithms (compare line 29 of Algorithm 5.4 and line 9 of Algorithm 5.8). However, the selection condition of the two algorithms have (minor) differences. Specifically, it is easy to see that whenever some value is selected by Algorithm 5.8 (lines 7 and 8), then some value (not necessarily the same) is also selected by Algorithm 5.5. The opposite is not true. If the number of messages received is not larger than $2n/3$, Algorithm 5.8 will not select any value, while Algorithm 5.5 may still select a value by line 3.

FaB Paxos [MA06] The FaB Paxos algorithm is designed for the Byzantine fault model ($f = 0$) and requires $n > 5b$ to tolerate b Byzantine faults. The algorithm is expressed in the context

[17]We ignore differences in the way algorithms implement phase change (timeout based mechanisms, failure detector based approaches, sending NACK messages, etc), message acceptance policies, retransmission rules, etc.

[18]T_D is chosen such that the same number of messages allow the condition at line 29 of Algorithm 5.4 and the condition at line 4 of Algorithm 5.5 to hold.

of "proposers", "acceptors" and "learners". For simplicity, in our framework, consensus algorithms are expressed without considering these roles.

The following parametrization leads to *Inst-FaB Paxos*: $T_D = \lceil (n+3b+1)/2 \rceil$, $FLAG = *$ and Algorithm 5.5 with $T_D = \lceil (n+3b+1)/2 \rceil$ as the *FLV* instantiation.

We now compare Inst-FaB Paxos with FaB Paxos. With $T_D = \lceil (n+3b+1)/2 \rceil$, the deciding condition is the same in both algorithms. However, the selection condition of the two algorithms have (minor) differences. With FaB Paxos, the selection rule is applied when $n-b$ messages are received. In that case, a value v is selected if it appears at least $\lceil (n-b+1)/2 \rceil$ times in the set of received messages; otherwise any value can be selected.[19] Therefore, if a number of received messages is smaller than $n-b$, FaB Paxos will not select any value, while Inst-FaB Paxos may still select a value by line 3 of Algorithm 5.5. Another difference is that FaB Paxos does not ensure the strong unanimity property, which allows a simpler selection round in the first phase (see Optimization (iii), Sect. 5.3.6).

5.5.2 Class 2 algorithms

Algorithm 5.9 *FLV* for class 2 with $b = 0$, $T_D = \lceil \frac{n+1}{2} \rceil$

```
1: $possibleVotes_p \leftarrow \{(vote, ts, -) \in \vec{\mu}_p^r :$
       $|\{(vote', ts', -) \in \vec{\mu}_p^r : vote = vote' \vee ts > ts'\}| > \frac{n}{2}$
2: **if** $|possibleVotes| = 1$ **then**
3:     **return** $v \; s.t. \; (v, -, -) \in possibleVotes$
4: **else if** $|\vec{\mu}_p^r| > \frac{n}{2}$ **then**
5:     **return** ?
6: **else**
7:     **return** $\perp$
```

Paxos [Lam98] Paxos assumes benign faults only ($b = 0$) and requires $n > 2f$.

We get *Inst-Paxos* from Algorithm 5.4 with the following parametrization: $T_D = \lceil \frac{n+1}{2} \rceil$,[20] $FLAG = \phi$, $Validator(p, \phi)$ implemented by majority voting (see Section 5.4.2), and Algorithm 5.9 as the *FLV* instantiation. Although we can use Algorithm 5.6 with $T_D = \lceil \frac{n+1}{2} \rceil$ as an instantiation of the *FLV* function, with only benign faults, Algorithm 5.6 can be simplified as we explain now. In addition, we apply Optimization (i), Sect. 5.3.6.

We now explain how to get Algorithm 5.9 from Algorithm 5.6. When $b = 0$, the set $correctVotes_p$ is the same as the set $possibleVotes_p$, which means that the set $correctVotes_p$ is not needed.[21]

[19]Note that when $T_D = \lceil (n+3b+1)/2 \rceil$, the condition at line 1 of Algorithm 5.5 for selecting a value v requires a smaller number of messages to be received than in FaB Paxos. For example, when $n = 7$ and $b = 1$, FaB Paxos requires at least 4 messages equal to v to be received (at least $\lceil (n-b+1)/2 \rceil (= 4)$), while Algorithm 5.5 with $T_D = \lceil (n+3b+1)/2 \rceil$ requires 3 messages (more than $\frac{n-b-1}{2} (= 2)$).

[20]Same argument as in footnote 18: same value for T_D in the decision round and in the *FLV* function.

[21]We can also use a set instead of a multiset for $possibleVotes$.

We now compare Inst-Paxos with Paxos. The decision rule is the same in both algorithms. The selection conditions are not necessarily the same. Paxos selects the vote with the highest timestamp, or any value if there are no votes with $ts > 0$. On the other hand, Inst-Paxos selects the value with the highest timestamp that is locked (returned by $FLV(\vec{\mu}_p^r)$ function, see Algorithm 5.9). Otherwise, if the $FLV(\vec{\mu}_p^r)$ function returns ?, Inst-Paxos selects any value chosen by some deterministic function (see line 11 of Algorithm 5.4). If the deterministic function at line 11 returns the value with the highest timestamp, then the selection condition of Inst-Paxos and Paxos are the same.

CT [CT96b]

Like Paxos, CT — the Chandra-Toueg consensus algorithm using the $\Diamond\mathcal{S}$ failure detector — assumes benign faults only ($b = 0$) and requires $n > 2f$. Paxos and CT use the same selection and decision conditions, and from this point of view rely on the same core mechanisms. The difference is in the *Validator*(p, ϕ) implementation: CT is based on a rotating coordinator.

MR [MR99] The Mostéfaoui-Raynal algorithm (MR) is designed for benign faults and requires $n > 2f$. It assumes "reliable channels", which allows for some optimizations. The reliable channel assumption can be expressed in the round model by the following predicate:

$$\mathcal{P}_{rel}(r) \equiv \forall p \in \mathcal{C} : |\{m \in \vec{\mu}_p^r : m \neq \bot\}| \geq n - f.$$

Let us consider an instantiation of Algorithm 5.4, with $\mathcal{P}_{rel}$ in every round, and the following parametrization: $T_D = \lceil \frac{n+1}{2} \rceil$, *Validator*$(p, \phi)$ implementing the *rotating coordinator function* and Algorithm 5.10 as the *FLV* instantiation. Algorithm 5.10 is a simplification of Algorithm 5.6 with only benign faults and reliable channels. We call this algorithm *Inst-MR*.

We now compare Inst-MR with MR. The validation and the decision condition are the same in both algorithms.[22] Furthermore, in both algorithms the validation round for phase $\phi + 1$ is executed in parallel with decision (and selection) round of phase ϕ. The selection condition of the MR algorithm expressed as a $FLV(\vec{\mu}_p^r)$ function (Algorithm 5.10) is a variant of Algorithm 5.6. We now explain how to get Algorithm 5.10 from Algorithm 5.6. Because $n - f > n - T_D$ and $\mathcal{P}_{rel}$ hold in every round, we have that $|\vec{\mu}_p^r| \geq n - f$ in all rounds and lines 5, 7 and 8 of Algorithm 5.6 can be suppressed. Since the algorithm considers only benign faults and assumes reliable channels, line 1 of Algorithm 5.10 is equivalent to lines 1-3 of Algorithm 5.6.

Note that in the original MR algorithm, a variable ts_p is not explicitly used. Basically, MR needs to distinguish only two cases, $ts_p = \phi$ and $ts_p < \phi$. Instead of $(vote_p, ts_p)$, the first case can be represented by $vote_p$, while the second case can be represented by $vote_p = \bot$, where $\bot$ is a special value different from all "normal" values of $vote_p$.

[22]In MR algorithm the functionalities of the selection and decision rounds are provided in the same round (see Optimization (iv)).

Algorithm 5.10 Instantiation of *FLV* function based on MR algorithm [MR99]

```
if received message $\langle v, \phi - 1\rangle$ then
    return $v$
else
    return ?
```

DLS algorithms [DLS88] There are three consensus algorithms in [DLS88]: one for benign faults (requires $n > 2f$) one for authenticated Byzantine faults ($n > 3b$) and one for Byzantine faults (also $n > 3b$). Let us denote these three algorithms by b-DLS (*benign*), a-DLS (*authenticated*) and B-DLS (*Byzantine*) and let us concentrate only on the former and the latter. The algorithm B-DLS belongs to class 3, and b-DLS to class 2. We discuss only b-DLS in more details since it is simpler.

Let us consider an instantiation of Algorithm 5.4, with $T_D = f+1$, $FLAG = \phi$ and $Validator(p, \phi)$ implementation based on the rotating coordinator paradigm and Algorithm 5.11 as the instantiation of the *FLV* function. We call this algorithm *Inst-b-DLS*.

We now compare Inst-b-DLS with b-DLS. The validation and the decision condition are the same in both algorithms. Although the selection conditions are the same, there is a small difference in how the selection logic is executed. Namely, the b-DLS algorithm relies on a mechanism called *locking* (which is different from the notion of locking used in this chapter): Whenever $vote_p = v$ with v different from a special value $\mathscr{A}$, then p has *locked* value v; If $vote_p = \mathscr{A}$ (special value) then p has not locked any value. Furthermore, b-DLS relies on a lock-release mechanism that takes place in the additional round (called lock-release round). The lock-release round is executed immediately before the selection round of the next phase, in which processes exchange messages ($vote$ and ts) to possibly unlock locked values, i.e., reset $vote_p$ to $\mathscr{A}$. When $\mathscr{P}_{good}$ holds in the lock-release round r, then at most one value is locked in round $r+1$.

By contrast, in Inst-b-DLS the lock-release mechanism is simulated inside the *FLV* function, i.e., there is no need for the additional round. Lines 1-4 of Algorithm 5.11 correspond to the lock-release mechanism.

We now explain with a simple example how Algorithm 5.11 ensures *FLV*-agreement. Let v_1 be locked in round r of phase $\phi_1 + 1$, because some honest process p has decided v_1 in round $r-1$ of phase ϕ_1. Then at least $T_D = f+1$ honest processes have $vote_p = v_1$ and $ts_p = \phi_1$ (*), as shown by ① at Figure 5.7. Furthermore, at most $n - T_D$ processes have $vote_p = v_2 \neq v_1$ (see ② at Figure 5.6). For every process p, we have $vote_p = v_1$ or $ts_p < \phi_1$ (*).

If some value $v \neq null$ is returned at line 7 or 9, this implies that vector V_p contains at least $n-f$ messages. As $T_D = f+1$, then $n \geq 2f+1$, and therefore any set of $n-f$ messages contains at least one message with $vote = v_1$ and $ts = \phi_1$ (see ③ at Figure 5.7). Because of (*), after lines 1-4 are executed, v_1 is the only value different from $\mathscr{A}$ in the vector V_p. Therefore, the set $possibleVotes_p$ contains only (v_1, ϕ_1) and v_1 is returned at line 7, which ensures

FLV-agreement.

When $\mathscr{P}_{good}$ holds, lines 8-9 ensure *FLV*-liveness as $\vec{\mu}^r_p$ contains at least $n-f$ messages.

Algorithm 5.11 Instantiation of *FLV* function based on [DLS88]

```
1: $V_p \leftarrow \vec{\mu}^r_p$
2: for $i = 1$ to n do
3:     if $\exists(vote, ts) \in \vec{\mu}^r_p$ $s.t.$ $ts > V_p[i].ts$ then
4:         $V_p[i] \leftarrow (\mathscr{A}, -)$
5: $possibleVotes_p \leftarrow \{(vote, -) \in V_p :$
       $|\{(vote', -) \in V^r_p : vote = vote' \vee vote' = \mathscr{A}\}| \geq n - f$
6: if $|possibleVotes_p| = 1$ then
7:     return $v$ $s.t.$ $(v, -) \in possibleVotes_p$
8: else if $|\vec{\mu}^r_p| \geq n - f$ then
9:     return ?
10: else
11:     return $null$
```

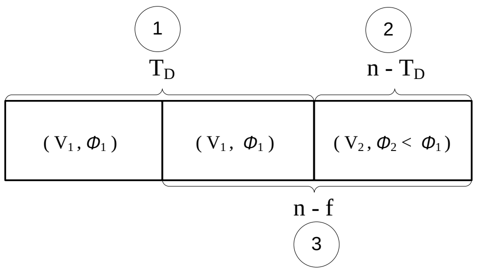


Figure 5.7: Illustration of *FLV* function based on [DLS88] ($n = 3$, $b = 0$, $f = 1$, $T_D = 2$)

MQB MQB is a *new* Byzantine consensus algorithm that requires $n > 4b$. It is obtained by instantiation of Algorithm 5.4 with *FLAG* $= \phi$ and *Validator*(p, ϕ) returning always Π; this corresponds to Algorithm 5.3. We consider the *FLV* instantiation given by Algorithm 5.6 and $T_D = \lceil \frac{n+2b+1}{2} \rceil$.[23]

Compared to PBFT, MQB has the advantage not to need the variable $history_p$, at the cost of requiring $n > 4b$ instead of $n > 3b$ (for PBFT).

5.5.3 Class 3 algorithms

PBFT [CL02]

PBFT is an algorithm that solves a sequence of instances of consensus (in the context of state

[23]Same argument as in footnote 18: same value for T_D in the decision round and in the *FLV* function.

machine replication). We consider here the instantiation of a single instance of consensus that represents the “core” of the PBFT algorithm. PBFT is designed for Byzantine faults ($f = 0$) and requires $n > 3b$.

We get *Inst-PBFT* from Algorithm 5.4 with the following parametrization: $T_D = 2b+1$, $FLAG = \phi$, $Validator(p,\phi) = \Pi$ and Algorithm 5.12 as the *FLV* instantiation. Algorithm 5.12 is a simplification of Algorithm 5.7 when strong unanimity is not considered (which is the case for PBFT). We also set $n = 3b + 1$, as in PBFT.

We now show how to get the *FLV* Algorithm 5.12 from Algorithm 5.7 if strong unanimity is not considered. Indeed, in this case, lines 8-9 of Algorithm 5.7 can be removed, and the conditions of line 5 and line 7 of Algorithm 5.7 can be merged into line 5 of Algorithm 5.12.

Algorithm 5.12 *FLV* for class 3 with $T_D = 2b + 1$ and $n = 3b + 1$

```
1: $possibleVotes_p \leftarrow \{(vote, ts, -) \in \vec{\mu}_p^r :$
     $|\{(vote', ts', -) \in \vec{\mu}_p^r : vote = vote' \vee ts > ts'\}| > 2b$
2: $correctVotes_p \leftarrow \{v : (v, ts, -) \in possibleVotes_p \wedge$
       $|\{(vote', ts', history') \in \vec{\mu}_p^r : (v, ts) \in history'\}| > b\}$
3: if $|correctVotes_p| = 1$ then
4:    return $v$ s.t. $(v, -, -) \in correctVotes_p$
5: else if $|correctVotes_p| > 1$ or $|\left\{(vote, ts, -) \in \vec{\mu}_p^r : ts = 0\right\}| > 2b$ then
6:    return ?
7: else
8:    return $null$
```

We now compare Inst-PBFT with PBFT. The validation and decision rounds are the same in both algorithms. There is a small difference in the selection condition of the selection round: whenever Inst-PBFT selects any value using some deterministic function (see line 11 of Algorithm 5.4), PBFT selects a special "null" value. Therefore, in PFBT the decision can be on a special "null" value, while in Inst-PBFT the decision is always on a "real" value.

Since the strong unanimity property is not considered, we can apply optimization (iii) (Sect. 5.3.6) to the selection round of Inst-PBFT. With this modification, the first phase of Inst-PBFT corresponds to the "common case" and all later phases correspond to the "view change protocol" of PBFT.

5.5.4 Where is the leader in the generic algorithm ?

Most of the consensus algorithms discussed (e.g., Paxos, CT, PBFT, FaB Paxos) are so called leader-based protocol, i.e., there is a designated process called leader (sometimes called also coordinator or primary) that has a special role in the execution of the algorithm. The leader role does not explicitly appear in the Algorithm 5.4. It is hidden either in the implementation of the $\mathscr{P}_{cons}$ predicate, e.g., in case of PBFT and FaB Paxos (see Section 3.4), or as an instantiation of the *Validator* parameter, for example in Paxos.

5.6 Conclusion

The chapter has presented a generic consensus algorithm that highlights the core mechanisms of various consensus algorithms for benign and Byzantine faults. The generic algorithm has four parameters: T_D, *FLAG*, *Validator* and *FLV*. Instantiation of these parameters led us to distinguish three classes of consensus algorithms into which well-known algorithms fit. It allowed us also to identify the new MQB algorithm. We believe that our classification should contribute to a better understanding of the jungle of consensus algorithms.

6 On the Reduction of Total-Order Broadcast to Consensus

In this chapter we investigate the reduction of total-order broadcast to consensus in systems with Byzantine faults. Among the several definitions of Byzantine consensus that differ only by their validity property, we identify those equivalent to total-order broadcast. Finally, we give the first total-order broadcast reduction algorithm to consensus with a constant time complexity with respect to consensus.

Publication: Zarko Milosevic, Martin Hutle and André Schiper. On the Reduction of Atomic Broadcast to Consensus with Byzantine Faults. In *30th International Symposium On Reliable Distributed Systems (SRDS 2011)*, Madrid, Spain, October 4 - October 18, 2011.

6.1 Introduction

The relation of consensus and total-order broadcast (called also atomic broadcast), including the reduction of total-order broadcast to consensus, is well understood in the case of crash faults [CT96a]. On the contrary, little is known about the relation between total-order broadcast and the consensus problem in the context of Byzantine faults. Although there is a huge amount of literature on the implementation of total-order broadcast with Byzantine faults, and some of them even use some sort of agreement as a building block, most of these papers do not—strictly speaking—reduce total-order broadcast to consensus.[1] In fact, we are aware only of two *reductions* of total-order broadcast to consensus with Byzantine faults [CNV06, DFK06], and in our opinion this work does not fully clarify the relation of consensus and total-order broadcast.

It is easy to observe that the standard reduction of total-order broadcast to consensus with benign faults does not work with Byzantine faults. One can also observe that there exist several definitions of consensus with Byzantine faults (which differ in the validity property), and it is not clear at all which one should be considered for a total-order broadcast reduction. In

[1]We discuss these papers in Section 6.7.

addition, the relation between these definitions of consensus and total-order broadcast is only partially understood.

Contribution. The first question addressed in the chapter is the relation between these various definitions of consensus for Byzantine faults and the total-order broadcast problem. We show that only some of the consensus validity properties in the literature lead to consensus problems that are equivalent to total-order broadcast. We also show that consensus with *weak unanimity* [DLS88] is not sufficient to solve total-order broadcast, while with *strong validity* [FG03] consensus is harder than total-order broadcast.

After that, we give a reduction of total-order broadcast to *range validity* consensus [DH08].[2] The reduction has a constant time complexity with respect to consensus, which is not the case for the reduction in [CNV06, DFK06].

After the reduction of total-order broadcast to range validity consensus, we give a reduction of range validity consensus to binary consensus that has constant time complexity (with respect to binary consensus). This implies that total-order broadcast can be reduced to binary consensus with constant time complexity (with respect to binary consensus).

About constant time reductions. The key feature of range validity consensus that allows us to achieve constant time reduction to consensus is the fact that it constrains the decision value even if not all initial values of correct processes are the same. This is not the case with the *strong unanimity* [DLS88] consensus, which was used in the reduction in [DFK06]. The reduction in [DFK06] requires, as written by the authors, a "deterministic and fair rule" for choosing which messages to propose to consensus such that eventually all correct processes propose the same set of messages. However, [DFK06] gives no examples of such a rule. Moreover, relying on such a mechanism clearly leads to a reduction algorithm that does not have constant time complexity with respect to consensus. The reduction in [CNV06] relies on *abortable consensus* [MR10, CNV06],[3] which restricts a decision value to the default value $\perp$ if not all initial values of correct processes are the same. The consequence is that the reduction algorithm only makes progress when the initial values of correct processes are the same. This leads to an algorithm that does not have constant time complexity with respect to consensus.

Roadmap The system model and problem definitions are given in Section 6.2. We then compare the different definitions of consensus and total-order broadcast in Section 6.3. In Section 6.4, the reduction of total-order broadcast to range validity consensus is presented. Solving range validity consensus is addressed in Section 6.5. The reduction of total-order broadcast to binary consensus with constant time complexity with respect to consensus is

[2]We denote with *range validity* the validity property used in [DH08], which requires the decision value to be in the range of initial values of correct processes.

[3]In [MR10] abortable validity is called *non-intrusion* validity, and the corresponding consensus *intrusion-tolerant Byzantine consensus*. In [CNV06] it is called *multi-valued consensus validity*.

discussed in Section 6.6. Section 6.7 is devoted to related work, and Section 6.8 concludes the chapter.

6.2 Definitions

We consider a system of n processes $\Pi = \{1, \dots, n\}$ that are connected with asynchronous reliable links. At most b processes may fail in an arbitrary way (Byzantine faults). A correct process is the one that is not Byzantine. Links satisfy the integrity property, i.e., a message that is received from a process was also sent by this process. We do not use signatures.

6.2.1 Reliable Unique Broadcast

In our reduction we use *reliable unique broadcast* [ADGFT06], [4] a broadcast primitive slightly different from *reliable broadcast* [HT94, CKPS01]. As shown in [ADGFT06], the primitive can be implemented in an asynchronous system with reliable links. Reliable unique broadcast is defined in terms of two primitives, rubcast and rubdeliver. A process p that wants to broadcast a message m invokes $\mathsf{rubcast}_p(k, m)$ with a tag k it has not used before. This message is then delivered by executing $\mathsf{rubdeliver}_q(k, m, p)$ at process q. Reliable unique broadcast fulfills the following properties:

- *Validity:* If a correct process p invokes $\mathsf{rubcast}_p(k, m)$, and this is the only invocation of p for index k, then p eventually executes $\mathsf{rubdeliver}_p(k, m, p)$.
- *Agreement:* For any two correct processes p and q, if p executes $\mathsf{rubdeliver}_p(k, m, s)$, then q eventually executes $\mathsf{rubdeliver}_q(k, m, s)$.
- *Integrity:* For any index k and process q, a correct process p executes $\mathsf{rubdeliver}_p(k, m, q)$ at most once. Moreover, if q is correct, then q previously invoked $\mathsf{rubcast}_q(k, m)$.

Observe that $\mathsf{rubcast}_q(k, m)$ provides stronger guarantees than $\mathsf{rbcast}_q(m')$ (reliable broadcast) with $m' = \langle k, m \rangle$. With the latter, a correct process could deliver both $m' = \langle k, m \rangle$ and $m'' = \langle k, \overline{m} \rangle$. With the former, because of the integrity property, a correct process cannot deliver (k, m) and $(k, \overline{m})$.

6.2.2 Consensus

As explained in Section 2.1, in the consensus problem, every process has an initial value from a set $\mathcal{V}$ and has eventually to irrevocably decide on a value from $\mathcal{V}$. The problem is defined by an agreement, a termination, and a validity property. As will be explained in the next section, there are several variants of the validity property.

[4] In [ADGFT06] the primitive is called *consistent unique broadcast.*

The interface to consensus uses two primitives, $\mathsf{propose}_p(i,v)$ to denote the start of a consensus instance i at p with the initial value v, and $\mathsf{decide}_p(i,d)$ to denote termination at p of consensus instance i with the decision value d.

6.3 Total-order broadcast reduction and the validity property of consensus

Consensus is defined by three properties, *validity*, *agreement* and *termination.* With benign faults, agreement can be uniform or non uniform, termination requires that all correct processes decide, and the standard validity property states that the decision must be the initial value (taken from a set $\mathcal{V}$) of some process. Other validity properties have also been considered [Lyn96]. However, it is well understood that the standard validity property above allows the reduction of total-order broadcast to consensus in an asynchronous system with benign faults.

With Byzantine faults, more definitions of consensus have been proposed, which differ only by the validity property (with Byzantine faults, only non uniform agreement makes sense). Depending on the validity property, the definition of consensus can be weaker, stronger or equivalent to total-order broadcast, as we now show. We consider the following definitions:

- *Strong validity* ([Nei94, FG03]): If a correct process decides on v, then v is the initial value of some *correct* process.[5]
- *Weak unanimity* [DLS88]: If all correct processes have the same initial value, and no process is faulty, then this is only possible decision value.
- *Strong unanimity* [DLS88]: If all correct processes have the same initial value, then this is the only possible decision value.

Abortable validity considers a special decision value $\perp$. This special value can be decided if not all correct processes have the same initial value:

- *Abortable validity* ([CNV06, MR10]):[6]
  (a) If all correct processes have the same initial value, then this is the only possible decision value.

[5]Strong validity can be seen as the counterpart, in the context of Byzantine faults, of the standard validity property with benign faults: *If a correct process decides v, then v is the initial value of some process.* Indeed, since it does not make sense to refer to the initial value of a Byzantine process, the definition becomes: *If a correct process decides v, then v is the initial value of some correct process.*

[6]In [CNV06] this is called *multi-valued consensus validity,* and is defined by three validity properties MVC1 to MVC3. The MVC1 validity property corresponds to abortable validity (a). The MVC2 validity property can be rephrased as follows: *If a correct process decides v, then v was proposed by some correct process.* Together with MVC3, we get abortable validity (b).

(b) If a correct process decides v, then v is the initial value of some correct process, or $v = \bot$.

The last validity property considered is introduced (without name) in [DH08]. We call it *range validity* since the decision value is in the range of initial values of correct processes. Range validity requires the domain $\mathcal{V}$ of initial values to be totally ordered:

- *Range validity* [DH08]: There exist two distinct correct processes p_1, p_2, so that every decision value d_q of a correct process q is between the initial values of p_1 and p_2 ($\mathcal{C}$ is the set of correct processes, v_p denotes the initial value of process p):

$$\exists p_1, p_2 \in \mathcal{C} : \forall q \in \mathcal{C} : v_{p_1} \leq d_q \leq v_{p_2}.$$

We have the following relations between these validity properties: strong validity implies range validity and abortable validity; both range validity and abortable validity imply strong unanimity; strong unanimity implies weak unanimity.

The special case $|\mathcal{V}| = 2$ is called *binary consensus*. For $|\mathcal{V}| = 2$, strong validity, range validity, strong unanimity and abortable validity are equivalent. In our proofs, when referring to binary consensus, we will assume the "strong unanimity" formulation.

In the rest of the section, we compare the difficulty of these various consensus problems with total-order broadcast.

6.3.1 Total-order broadcast is harder than weak unanimity consensus

We show that with $n > 3b$, total-order broadcast is harder than weak unanimity consensus.[7] To do so, we first show that whenever total-order broadcast is solvable, range validity consensus is also solvable.

Lemma 6.1. *If $n > 3b$, then range validity consensus can be reduced to total-order broadcast.*

Proof. The reduction is as follows. First, every process total-order broadcasts its initial value, and waits until $n - b$ messages are delivered. When a process has to-delivered $n - b$ messages, it orders the values of these messages in ascending order and decides on the value at $(b+1)$st position.

Termination. Since total-order broadcast guarantees the delivery of all messages from correct processes, and there are at least $n - b$ correct processes, all correct processes decide.

Agreement. Since total-order broadcast ensures that all messages are delivered in total-order, the set of the $n - b$ first messages is the same at all correct processes, and thus all correct processes decide on the same value.

[7] Consensus with validity property X will simply be called *"X consensus"* or *"X validity consensus"*.

Range Validity. After values from the first $n - b$ messages are ordered, we have two cases to consider: (1) the b first values are from faulty processes, (2) the b first values contain at least one value from a correct process. It is easy to see that in both cases range validity holds. □

Since range validity implies strong unanimity validity, we have the following corollary:

Corollary 6.1. *If $n > 3b$, then strong unanimity consensus can be reduced to total-order broadcast.*

Now we show that for $n > 3b$, the solvability of weak unanimity consensus does not imply solvability of total-order broadcast. We start with a lemma.

Lemma 6.2. *Strong unanimity consensus cannot be reduced to weak unanimity consensus.*

Proof. Assume by contradiction that such a reduction T exists. Then T, together with weak unanimity consensus, solves strong unanimity consensus, and in particular in all runs where at least one process is faulty and weak unanimity consensus always decides a fixed value v_0 (independently of the initial values of correct processes). Now consider a modified algorithm T' that uses always v_0 instead of the output of weak unanimity consensus. Clearly, for all runs of T' there is a run of T that is indistinguishable to the correct processes, and thus they decide as in T. This means that T' is a deterministic algorithm that solves strong unanimity consensus in an asynchronous systems with at least one faulty process. A contradiction with the FLP impossibility result [FLP85]. □

We can now prove the above claim.

Proposition 6.1. *If $n > 3b$, total-order broadcast cannot be reduced to weak unanimity consensus.*

Proof. Assume by contradiction that total-order broadcast can be reduced to weak unanimity consensus. If $n > 3b$, by Corollary 6.1, consensus with strong unanimity can be reduced to total-order broadcast. Thus, strong unanimity consensus is reducible to consensus with weak unanimity. A contradiction with Lemma 6.2. □

From Corollary 6.1 and Proposition 6.1 it follows that with $n > 3b$ total-order broadcast is harder than weak unanimity consensus.

6.3.2 Strong validity consensus is harder than total-order broadcast

We show that strong validity consensus with more than three values ($|\mathcal{V}| > 3$) is harder than total-order broadcast. To do so, we will use the result from [CNV06] that total-order broadcast can be reduced to binary consensus.

Proposition 6.2. *If $n > 3b$, then total-order broadcast can be reduced to binary consensus.*

Proof. The result is proven in [CNV06]. □

Proposition 6.3. *If $n > 3b$, then total-order broadcast can be reduced to strong validity consensus.*

Proof. If $n > 3b$ then by Proposition 6.2 total-order broadcast can be reduced to binary consensus, that is strong validity consensus with $|V| = 2$. □

We now show that the opposite does not hold in general.

Proposition 6.4. *For $|\mathcal{V}| > 3$, in general strong validity consensus cannot be reduced to total-order broadcast.*

Proof. The proof is by contradiction. Consider a synchronous system with $n = 4$ and $b = 1$. We first show that in this setting, total-order broadcast is solvable. Indeed, in this setting binary consensus is solvable. If binary consensus is solvable, by Proposition 6.2 total order-broadcast is also solvable.

Let us assume by contradiction that there is a reduction of strong validity consensus to total-order broadcast in an asynchronous system. With the result of the previous paragraph, this means strong validity consensus is solvable in a synchronous system with $n = 4$ and $b = 1$. This contradicts the results from [FG03], where it is shown that solving strong validity consensus in a synchronous system requires $n > b \cdot \max(3, |\mathcal{V}|)$. □

6.3.3 Consensus problems equivalent to total-order broadcast

The other consensus problems introduced, namely strong unanimity consensus, abortable validity consensus and range validity consensus are all equivalent to total-order broadcast. This result follows from three propositions:

Proposition 6.5. *If $n > 3b$, total-order broadcast and range validity consensus are equivalent.*

Proof. From Proposition 6.2 it follows that if $n > 3b$ total-order broadcast can be reduced to binary consensus, that is range validity consensus with $|V| = 2$. On the other hand, by Lemma 6.1, if $n > 3b$ range validity consensus can be reduced to total-order broadcast. □

Proposition 6.6. *If $n > 3b$, total-order broadcast and abortable validity consensus are equivalent.*

Proof. This result is proven in [CNV06]. □

Proposition 6.7. *If $n > 3b$, total-order broadcast and strong unanimity consensus are equivalent.*

Proof. From Corollary 6.1 it follows that if $n > 3b$ strong unanimity consensus can be reduced to total-order broadcast. On the other hand, by Proposition 6.2 if $n > 3b$ then total-order broadcast can be reduced to binary consensus, that is strong unanimity consensus with $|V| = 2$. This shows that total-order broadcast and strong unanimity consensus are equivalent. □

6.4 Reducing total-order broadcast to consensus with range validity

In this section we give a reduction of total-order broadcast to consensus with range validity, the first constant time reduction of total-order broadcast to consensus for Byzantine faults. For range validity consensus we assume that $\mathcal{V}$ is a countable set, and without loss of generalization, we assume that $\mathcal{V}$ is the set $\{0, \dots, |\mathcal{V}| - 1\}$ if $\mathcal{V}$ is finite, or $\mathbb{N}_0$ if $\mathcal{V}$ is infinite.

6.4.1 Reduction algorithm

The reduction is given as Algorithm 6.1. It uses *reliable unique broadcast.* In order to to-broadcast a message, the message is first rubcast together with a local sequence number *lsn* (line 8). When this message is rubdelivered it is stored in the variable *rubdelivered* for further processing (line 10).

The main loop (lines 14-26) is executed when there are messages that are rubdelivered but not yet to-delivered. The code between two wait statements (e.g., lines 13-18) and the "upon" blocks are executed atomically. We call each iteration of the main loop a "round". In the sequel, we denote with x_p the value of a variable x at process p, and for any round $r > 0$ we denote with x_p^r the value of x_p at the beginning of round r (a round begins at line 14). In every round r, Algorithm 6.1 executes n instances of range validity consensus in parallel, see lines 14 and 18. Consensus instance $(r-1) \cdot n + \pi$ is used to agree which messages to-broadcasted by process π are to-delivered in round r. A process p proposes in consensus instance $(r-1) \cdot n + \pi$ the difference between $\mathit{rdel\_sn}_p[\pi]$ and $\mathit{decided\_sn}_p[\pi]$, where

- $\mathit{rdel\_sn}_p[\pi]$ contains the highest sequence number, such that all messages from π with a smaller sequence number are rubdelivered at p, and
- $\mathit{decided\_sn}_p[\pi]$ contains the sequence number of the last message from π that is to-delivered by p.

The decision value of consensus instance $(r-1) \cdot n + \pi$ determines the number of messages from π that will be to-delivered in round r. Finally, all messages from π are to-delivered by all correct processes in the order of their sequence numbers *lsn.*

Algorithm 6.1 Total-order broadcast by reduction to range-validity consensus

```
$r \leftarrow 0$
to-delivered $\leftarrow \emptyset$
$rubdelivered \leftarrow \emptyset$
$lsn \leftarrow 0$
$decided\_sn[1 \dots n] \leftarrow [0 \dots 0]$

upon to-broadcast($m$) do
  $lsn \leftarrow lsn + 1$
  rubcast($lsn, m$)

upon rubdeliver($i, m, q$) do
  $rubdelivered \leftarrow rubdelivered \cup \{\langle m, i, q \rangle\}$

loop
  wait until $rubdelivered$ − to-delivered $\neq \emptyset$
  $r \leftarrow r + 1$
  for $\pi = 1$ to $n$ do
    $rdel\_sn[\pi] \leftarrow$ largest $\ell$, so that $\forall j, 1 \le j \le \ell : \langle -, j, \pi \rangle \in rdelivered$
    $v[\pi] \leftarrow \min(rdel\_sn[\pi] - decided\_sn[\pi], \max(\mathcal{V}))$
    range-propose($(r-1) \cdot n + \pi, v[\pi]$)
  for $\pi = 1$ to $n$ do
    wait until range-decide($(r-1) \cdot n + \pi, decision[\pi]$)
    if $decision[\pi] > 0$ then
      for $\ell = decided\_sn[\pi] + 1$ to $decided\_sn[\pi] + decision[\pi]$ do
        wait until $\exists msg = \langle m, \ell, \pi \rangle : msg \in rubdelivered$
        if $\langle m, -, - \rangle \notin$ to-delivered then
          to-deliver($m$)
          to-delivered $\leftarrow$ to-delivered $\cup\ msg$
      $decided\_sn[\pi] \leftarrow decided\_sn[\pi] + decision[\pi]$
```

We illustrate Algorithm 6.1 on a simple example using Figure 6.1. We consider a scenario with four processes (p4 is the Byzantine process), where process p1 initially executes to-broadcast($m1$). The execution of the algorithm starts by process p1 executing rubcast($1, m1$). Once a correct process executes rubdeliver($1, m1, p1$) it starts in parallel next four range validity consensus instances. In the scenario considered on Figure 6.1 correct processes start consensus instances after executing rubdeliver($1, m1, p1$). As this is the first message rubdelivered from process p1, processes propose 1 in the consensus instance that "belongs" to process p1. As correct processes have not rubdelivered messages from other processes they propose 0 in the corresponding consensus instances of other processes. Eventually, all correct processes decide in the consensus instances started. As all correct processes propose 1 in the consensus instance that belongs to process p1, the decision value must be 1. At this point correct processes to-deliver the next not delivered message from process p1, i.e., message m1 in this case.

Remark Executing range validity consensus n times in parallel, as done in Algorithm 6.1, has a high message complexity. This cost can easily be reduced to the message complexity of one instance of range validity, with larger messages. The solution consists of executing *one* instance of consensus on a “vector of n elements” instead of “n instances of range consensus”

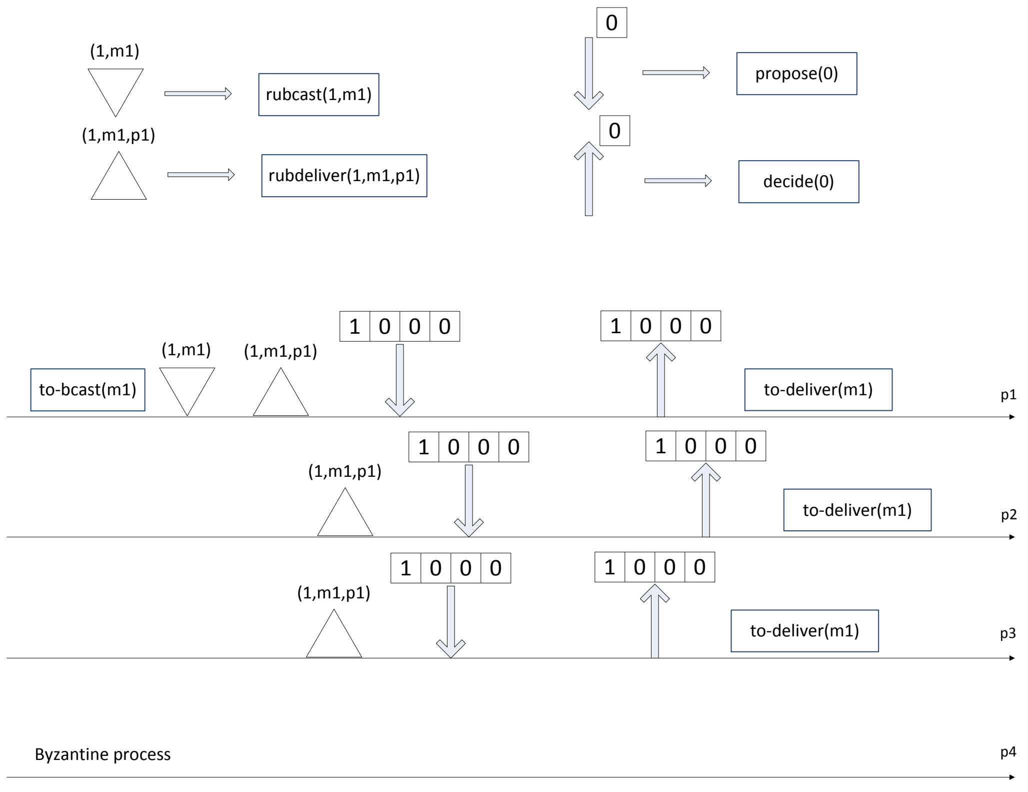


Figure 6.1: Illustration for Algorithm 6.1, $n = 4, b = 1$, process p4 is a Byzantine process. Process p1 executes to-broadcast$(m1)$.

in parallel. The consensus then operates simultaneously on each entry of the vector, providing an independent element-wise range consensus semantics for each of the vector's elements.

6.4.2 Proof of Algorithm 6.1

Lemma 6.3. *For all correct processes p and q, and all $r > 0$, $decided\_sn_p^r = decided\_sn_q^r$.*

Proof. The proof is by induction on r. For $r = 1$, at all processes $decided\_sn^r = [0\ldots0]$, and thus the lemma holds. Assume now that the lemma is true for $r-1$. When the correct processes decide consensus instances $(r-2)n+1$ to $(r-2)+n$ in round $r-1$, because of the agreement property of range validity consensus they all decide the same values. Thus, at the end of round $r-1$ (at line 26), all processes update $decided\_sn$ consistently. Therefore, $decided\_sn_p^r = decided\_sn_q^r$ and the lemma holds also for r. □

Lemma 6.4. *For all correct processes p and q, and all $r > 0$, $\text{to-delivered}_p^r = \text{to-delivered}_q^r$.*

Proof. The proof is by induction on r. For $r = 1$, at all processes to-delivered$^r = \emptyset$, and thus the lemma holds. Assume now that the lemma is true for $r-1$. By the agreement property of consensus, all correct processes terminate consensus instance $(r-2)n+\pi$ for $\pi = 1..n$ in round $r-1$ with the same decision *decision*(π) $(\star)$. The value of *decided_sn* in round $r-1$ at line 21 is equal to *decided_sn*$^{r-1}$, therefore by Lemma 6.3, *decided_sn* has the same value at line 21 in round $r-1$ at all correct processes $(\star\star)$. By $(\star)$ and $(\star\star)$, for every instance $\pi = 1..n$ of loop at line 18 all correct processes will execute the same number of iterations at line 21. For each iteration they will consider the same message due to the agreement property of reliable unique broadcast. By induction assumption we have $\text{to-delivered}_p^{r-1} = \text{to-delivered}_q^{r-1}$, and since the value of to-delivered at line 23 at round $r-1$ is equal to $\text{to-delivered}^{r-1}$, the condition at p and q at line 23 of round $r-1$ will evaluate to the same value. Therefore, p and q will update to-delivered in round $r-1$ at line 25 to the same value, so the lemma holds also for r. □

Lemma 6.5. *If a message m is to-delivered at a correct process p in round r, it is also to-delivered in round r at any correct process q.*

Proof. By Lemma 6.4 we have $\text{to-delivered}_p^{r-1} = \text{to-delivered}_q^{r-1}$ and $\text{to-delivered}_p^{r} = \text{to-delivered}_q^{r}$ (for $r = 1$ the result follows also from line 2 of the reduction algorithm) (*). We have

$$\text{to-delivered}_p^{r} = \text{to-delivered}_p^{r-1} \cup todel_p^r$$

where $todel_p^r$ denotes the messages to-delivered by p in round r (**). From (*) and (**) we have $todel_p^r = todel_q^r$. If m is to-delivered by p in round r, we have $m \in todel_p^r$. So we also have $m \in todel_q^r$, i.e., m is to-delivered by q in round r. □

Lemma 6.6 (TO-Validity)**.** *If a correct process p invokes* $\text{to-broadcast}_p(m)$*, then p eventually executes* $\text{to-deliver}_p(m)$.

Proof. Assume by contradiction that a message m to-bcast by p is the first message to-bcast by p that is not to-delivered by some correct process q. When process p invokes $\text{to-broadcast}_p(m)$, the local sequence number *lsn* is incremented and the message $\langle lsn+1, m\rangle$ is rubcast. Messages that are to-bcast by p before m are rubcast with sequence number smaller than $lsn+1$ and by assumption all these messages are to-delivered. By line 26 (update of *decided_sn*$_q[p]$), $\exists r$ s.t. eventually *decided_sn*$_q^r[p]$ = *lsn*. From Lemma 6.3 follows that at all correct processes we have *decided_sn*$[p]^r$ = *lsn*.

By the termination property of reliable unique broadcast, all messages $\langle i, -\rangle$ with $i = 1..lsn+1$ are eventually rubdelivered at all correct processes. Because $\langle lsn+1, m\rangle$ was rubcast by a correct process, by the agreement and integrity property of reliable unique broadcast, all correct processes eventually rubdeliver $\langle lsn+1, m\rangle$, and have $\langle m, lsn+1, p\rangle$ in *rubdelivered*. Therefore, in round r all correct processes propose in the consensus instance $(r-1)\cdot n+p$ a value larger or equal to 1 (see lines 16-17).[8] By the range validity property of consensus, *decision*$_q[p]$ is therefore larger or equal to 1 and m is to-delivered. A contradiction. □

[8]For simplicity we assume that $\mathcal{V}$ is infinite: otherwise the same reasoning will apply for some round $r' > r$.

Lemma 6.7 (TO-Agreement). *For any two correct processes p and q, if p executes* $\text{to-deliver}_p(m)$, *then q eventually executes* $\text{to-deliver}_q(m)$.

Proof. Follows directly from Lemma 6.5. □

Lemma 6.8 (TO-Integrity). *For any message m, every correct process p executes* $\text{to-deliver}_p(m)$ *at most once. Moreover, if the sender q of message m is correct, then q previously invoked* $\text{to-broadcast}_q(m)$.

Proof. The first part of the claim follows directly from line 23. For the second part, a message is to-delivered at a correct process p only if p rubdelivered $\langle i, m, q\rangle$ before. If the sender q is correct, by the integrity property of reliable unique broadcast, q previously invoked $\text{rubcast}_q(-, m)$. By lines 6 and 8, q previously invoked $\text{to-broadcast}_q(m)$. □

Lemma 6.9 (TO-Order). *If some correct process p executes* $\text{to-deliver}_p(m)$ *before* $\text{to-deliver}_p(m')$, *then every correct process q executes* $\text{to-deliver}_q(m')$ *only after it has executed* $\text{to-deliver}_q(m)$.

Proof. Let p and q be correct processes such that $\text{to-deliver}_p(m)$, $\text{to-deliver}_p(m')$, $\text{to-deliver}_q(m)$, and $\text{to-deliver}_q(m')$ are executed. Let further r_x, resp. r'_x, denote the value of r when message m, resp. m', is delivered at process $x \in \{p, q\}$.

By Lemma 6.5, messages m and m' are each delivered in the same rounds by all correct processes, i.e., $r_p = r_q$ and $r'_p = r'_q$. If messages m and m' are delivered in different rounds, then either $r_p < r'_p$ and $r_q < r'_q$, or $r_p > r'_p$ and $r_q > r'_q$, and the order property holds. If m and m' are delivered in the same round r, then the messages are delivered in the order of the process IDs and lsn, and the TO-Order property holds. □

From Lemmas 6.6–6.9 and the fact that reliable unique broadcast can be implemented in an asynchronous system with reliable links when $n > 3b$ [ADGFT06], we get the following theorem:

Theorem 6.1. *If $n > 3b$, then Algorithm 6.1 is a reduction of total-order broadcast to range validity consensus in an asynchronous system with reliable links.*

6.4.3 Why *reliable unique broadcast*?

We illustrate now on an example why *reliable unique broadcast* [ADGFT06] (rather than reliable broadcast [HT94, CKPS01]) is needed in Algorithm 6.1. Consider $n = 4, b = 1$, processes denoted by $i \in [1, 4]$, and process 4 being the faulty process. If *reliable broadcast* is used in Algorithm 6.1, then processes 1 and 2 may rdeliver, from process 4 at line 9, $m' = \langle 1, m\rangle$ resp. $m'' = \langle 1, \overline{m}\rangle$. By the *Agreement* property of *reliable broadcast,* all correct processes eventually *rdeliver* $m' = \langle 1, m\rangle$ and $m'' = \langle 1, \overline{m}\rangle$ from process 4. Therefore, all correct processes will eventually start round k proposing 1 for consensus instance $(k-1)\cdot n + 4$. By the range validity property of consensus, all processes will decide 1 in the consensus instance $(k-1)\cdot n+4$. Therefore, process 1 will *adeliver* m and process 2 will *adeliver* $\overline{m}$, violating the TO-order property.

6.4.4 Time complexity of Algorithm 6.1

We express the time complexity τ_m^{TO} of our reduction algorithm as the upper bound of the duration between the total-order broadcast of some message m by a correct process and the delivery of m at all correct processes.[9] We express τ_m^{TO} in terms of the maximum execution time of consensus — denoted by τ^C — and the maximum communication delay δ. For τ^C, we consider the starting time of an instance of consensus to be the time at which the last input event of a correct process occurs, and the ending time to be the time at which the last output event (decision value) of a correct process occurs. We define δ as the maximum transmission delay on messages exchanged among correct processes. We have the following result:

Theorem 6.2. *[Time complexity] If $|\mathcal{V}| = \infty$, for Algorithm 6.1 we have $\tau_m^{TO} \leq 2\tau^C + 3\delta$.*

Proof. Let τ_{rv}^C denote the maximum execution time of range validity consensus. We show that ordering an arbitrary message from a correct process takes at most $2\tau_{rv}^C + 3\delta$ time. , i.e., for any message m we have $\tau_m^{TO} = e_m^{TO} - s_m^{TO} \leq 2\tau_{rv}^C + 3\delta$ (s_m^{TO} being time when m is to-broadcast and e_m^{TO} being time when last correct process to-deliver m). In the proof we assume that reliable unique broadcast is implemented as in [ADGFT06] and therefore has time complexity of 3δ.

Assume that a correct process p to-broadcasts message m at time s_m^{TO}. Then, also at time s_m^{TO}, message m is rubcast with the next index ℓ. Since p is correct, for all j, $0 < j < \ell$, messages have been rubcast before. By time $s_m^{TO} + 3\delta$, all these messages have been rubdelivered at all correct processes. Therefore, at every correct process q, the set $rubdelivered_q$ contains a message $\langle -, j, p\rangle$ for all $0 < j \leq \ell$ $(\star)$. At most τ_{rv}^C later, every process has started a new consensus instance with an index k such that $k = (r-1)\cdot n + p$. By $(\star)$, $rdel\_sn[p] \geq \ell$ at line 15 of round r at all correct processes. Now, if $decided\_sn[p] \geq \ell$, all correct processes already to-delivered message m before, and because of $\tau_m^{TO} \leq 2\tau_{rv}^C + 3\delta$ we are done. Othewise, $decided\_sn[p] < \ell$ at line 16 in round r. Then because $|\mathcal{V}| = \infty$, at all correct processes we have that $v[p] = rdel\_sn[p] - decided\_sn[p]$ at line 16, and this value is proposed to consensus instance $(r-1)\cdot n+p$. By the range validity property of consensus, $decision[p]$, the decision value of consensus instance $(r-1)\cdot n+p$ is greater or equal to $v[p]$. Therefore, at all correct processes $decided\_sn[p] + decision[p] \geq decided\_sn[p] + rdel\_sn[p] - decided\_sn[p] \geq rdel\_sn[p] \geq \ell$. Therefore, message $\langle m, p\rangle$ will be to-delivered at line 24. This will happen at all correct processes at latest at time $s_m^{TO} + 2\tau_{rv}^C + 3\delta$. The theorem follows. □

[9]Since communication delays are unbounded in an asynchronous system, the analysis is done for finite runs, where a maximum value exists for every run. To simplify the notation, we drop the reference to runs.

Algorithm 6.2 Reduction of range validity consensus to binary consensus (code of process p)

```
1:  $r \leftarrow 0$
2:  $rubdelivered \leftarrow \emptyset$
3:  upon range-propose($v$) do
4:    rubcast($1, v$)    /* 1 is the local index number, see Sect.6.2.1; each process rubcasts only once. */
5:  upon rubdeliver($1, v, q$) do
6:    $rubdelivered \leftarrow rubdelivered \cup \{\langle 1, v, q\rangle\}$
7:  wait until $|rubdelivered| \geq n - b$
8:  loop
9:    $r \leftarrow r + 1$
10:   for $\pi = 1$ to $n$ do
11:     $v[\pi] \leftarrow 1$ if $\langle -, \pi\rangle \in rdelivered$ otherwise 0
12:     bin-propose($(r-1) \cdot n + \pi, v[\pi]$)
13:   for $\pi = 1$ to $n$ do
14:     wait until bin-decide($(r-1) \cdot n + \pi, decision[\pi]$)
15:   $\Pi_1 \leftarrow \{q \in \Pi : decision[q] = 1\}$
16:   if $|\Pi_1| \geq n - b$ then
17:     for all $i \in \Pi_1$ do
18:       wait until $\exists m :\ msg = \langle 1, m, i\rangle \in rubdelivered$
19:     $dec \leftarrow$ max $v$ $s.t.$
        $\exists q_v \in \Pi_1 : (1, v, q_v) \in rubdelivered \wedge |\{q \in \Pi_1 : (1, v', q) \in rubdelivered\ s.t.\ v' \geq v\}| \geq b + 1$
20:     range-decide($dec$)
```

6.5 Solving range validity consensus

In order to complete the contribution of the previous section, we discuss now the solution of range validity consensus. In Section 6.5.1 we show that range validity consensus is reducible to binary consensus with constant time complexity. The reduction is given by Algorithm 6.2. Solving range validity consensus by reduction to binary consensus with constant time complexity allows us to obtain reduction of total order broadcast to binary consensus that has constant time complexity, as we will show in Section 6.6. Then, in Section 6.5.2 we explain briefly how range validity consensus can be solved directly in the partially synchronous model [DLS88].

6.5.1 Reduction of range validity consensus to binary consensus

Algorithm 6.2 starts by having processes *rubcast* their initial value (line 4). After delivery of $n - b$ initial values (line 7), processes execute a sequence of rounds: in each round r, n binary consensus instances, namely instances $(r-1) \cdot n + i$, $i = 1..n$, are executed in parallel. A correct process proposes 1 in consensus instance $(r-1) \cdot n + i$ if it has *rubdelivered* the initial value of process i; otherwise it proposes 0 (line 11). The algorithm terminates in a round in which at least $n - b$ binary consensus instances decide 1 (line 16). Line 19 ensures that the decision value is greater or equal than the initial value of at least one correct process. Moreover, since $n > 3b$, line 19 also ensures that the decision value is smaller or equal than the initial value of at least one correct process.

The next theorem establishes the correctness of the reduction:

Theorem 6.3. *If $n > 3b$ then Algorithm 6.2 reduces range validity consensus to binary consensus.*

Proof. To avoid ambiguity we use the prefix range resp. bin to distinguish the proposal and decision events of the two consensus specifications.

Agreement. Because of the agreement property of binary consensus, for any round r, the set Π_1 is the same at all correct processes p. By the agreement and integrity property of reliable unique broadcast, if two correct processes rubdeliver a message with the same tag from some process, they rubdeliver the same message. Since the deterministic rules at lines 15-19 are based only on the set Π_1 and messages rubbcast by processes from the set Π_1, it follows that all correct processes decide the same value.

Termination. By the validity and agreement properties of reliable unique broadcast all correct processes eventually rubdeliver messages from all correct processes ($\star$). Since there are at most b Byzantine processes, no correct process waits forever at line 7. By the termination property of binary consensus, no correct process waits forever at line 14. Furthermore, by ($\star$), eventually all correct processes have received messages from all correct processes, so they propose 1 for all consensus instances $(r-1)\cdot n+\pi$ that correspond to correct processes π. By the agreement property of binary consensus, these instances of binary consensus decide 1. Therefore, the set Π_1 eventually has $n-b$ elements and the condition at line 16 evaluates to *true*.

If a process waits at line 18 to rubdeliver $(1, m, q)$, this means that the decision value of binary consensus instance $(r-1)\cdot n+q$ was 1. By the validity property of binary consensus, at least one correct process proposed 1 for this binary consensus instance. Therefore, by line 11, this process rubdelivered $(1, m, q)$. By the agreement property of reliable unique broadcast, all correct processes eventually rubdeliver $(1, m, q)$, so no process waits forever at line 18.

Range Validity. A value v is decided if (i) there are at least $b+1$ values v' such that $v' \geq v$, and (ii) if v is the maximum value among the values that satisfies condition (i) (see line 19). Condition (i) ensures that there is at least one initial value of a correct process that is greater or equal than the decision value ($\star$). Further, a value is decided only if the set Π_1 contains at least $n-b$ values. Since we assume that $n > 3b$, the number of values that satisfies condition (i) is $n-2b > b$, so at least one among these values is an initial value of a correct process. Therefore, condition (ii) ensures that selected value is greater or equal to at least one initial value of correct process ($\star\star$). From ($\star$) and ($\star\star$) it follows that the decision value is between the initial values of correct processes. □

As in in Section 6.4.4, the time complexity of Algorithm 6.2 is expressed in terms of maximum execution time of binary consensus τ^C_{bin} and the maximum communication delay δ. We have the following result:

Theorem 6.4. *The reduction given by Algorithm 6.2 has a time complexity* $\tau_k^{RV} \leq 2\tau_{bin}^C + 3\delta$.

Proof. We show that an execution of every instance k of range validity consensus takes at most $2\tau_{bin}^C + 3\delta$ time i.e., $\tau_k^{RV} = e_k^{RV} - s_k^{RV} \leq 2\tau_{bin}^C + 3\delta$ for any range validity consensus instance (s_k^{RV} being time when the last input event of a correct process for range validity consensus instance k occurs, while e_k^{RV} is the time where the last output event of a correct process occurs). In the proof we assume that reliable unique broadcast is implemented as in [ADGFT06] and therefore has time complexity of 3δ.

By definition of s_k^{RV}, at time $s_k^{RV} + 3\delta$, all messages rubcast by correct processes at line 4 are rubdelivered by all correct processes (see line 6). By definition of τ_{bin}^C, the next n binary consensus instances are started at line 12 the latest at time $s_k^{RV} + 3\delta + \tau_{bin}^C$. Since at this time, all correct processes have received messages from all correct processes, they will propose 1 for all consensus instances $(r-1) \cdot n + i$, for all correct processes i. By the validity property of binary consensus, all these instances will decide 1. Since there are at least $n - b$ correct processes, at least $n - b$ consensus instances will return 1 and the condition at line 16 evaluates to true at all correct processes. Therefore, they all decide at line 20 the latest at time $s_k^{RV} + 3\delta + 2\tau_{bin}^C$. Therefore, the execution time of any range validity consensus instance k is $\tau_k^{RV} \leq 2\tau_{bin}^C + 3\delta$, and Algorithm 6.2 has a constant time complexity with respect to both binary consensus and communication. □

6.5.2 Solving range validity consensus in the partially synchronous system model

Range validity consensus is easy to solve in the partially synchronous model [DLS88]. For example, this is achievable by modifying one single line of the CL consensus algorithm that solves strong unanimity consensus (Algorithm 3.6) from Section 3.5.3. The condition at the line 23 of Algorithm 3.6 ensures that a value considered for a decision satisfies strong unanimity constraint. More precisely, according to this rule, the most frequent initial value is chosen. This ensures that in case all correct processes have the same initial value v, the value v is selected. If instead of this condition, we use the one from the line 19 of Algorithm 6.2, we obtain the consensus algorithm that solves range validity consensus. As already explained above, the condition at line 19 of Algorithm 6.2 ensures the selected value v is in the range of initial values of correct processes, i.e., there exist values v_1 and v_2 that are initial values of correct processes such that $v_1 \leq v \leq v_2$.

6.6 Reducing total-order broadcast to binary consensus

If we use Algorithm 6.1 with $|\mathcal{V}| = 2$, we get a reduction of total-order broadcast to binary consensus. Unfortunately, this reduction does not have constant time complexity. For a finite domain, in particular for $|\mathcal{V}| = \{0,1\}$, the reduction shown by Algorithm 6.1 has a constant time complexity only if the number of to-broadcast invocation by a correct process is bounded:

Algorithm 6.1 can order at most $|\mathcal{V}|$ messages per round and process. Since every round takes at least the execution time of consensus, ordering $n|\mathcal{V}|$ requests requires $O(n)$ time.

Constant time complexity can be obtained by combining Algorithm 6.1 with Algorithm 6.2 (to implement range validity consensus). This leads to a reduction of total-order broadcast to binary consensus with constant time complexity. From Theorem 6.1 and Theorem 6.2 follow:

Corollary 6.2. *The total-order broadcast reduction to binary consensus, obtained by combining Algorithm 6.1 (with $|\mathcal{V}| = \infty$) with Algorithm 6.2, has a time complexity $\tau_m^{TO} \leq 2(2\tau_{bin}^C + 3\delta) + 3\delta$.*

6.7 Related Work

Total-order broadcast Algorithms for total-order broadcast have attracted a lot of attention, both for the benign and for the Byzantine fault model. In this chapter, we focus on algorithms that solve total-order broadcast by reduction to consensus.

In [CT96a], Chandra and Toueg solve total-order broadcast by reduction to consensus for benign faults. The reduction has constant time complexity with respect to consensus. The authors also mention that total-order broadcast is reducible to consensus in the Byzantine fault model, but no reduction is given.

In [CNV06, MNCV11a], Correira *et al.* give reductions of total-order broadcast to consensus in the Byzantine fault model. They give two reductions, the first to abortable consensus and the second to binary consensus. However, as we point out below, there is some confusion between the use of reliable broadcast and reliable unique broadcast in the reduction. Moreover, the reductions do not have constant time complexity with respect to consensus, a property of our reduction. Indeed, the reduction to abortable consensus has a time complexity $\tau_m^{TO} \leq (b+1)\tau^C + 6\delta$, while the time complexity of the reduction to binary consensus is $\tau_m^{TO} \leq (b+1)(\tau_{bin}^C + 6\delta) + 6\delta$.

In [CKPS01], Cachin *et al.* give a modular total-order broadcast algorithm for Byzantine faults with authentication. As an intermediate module in this algorithm, they use multi-valued Byzantine agreement with *external validity*. External validity allows an application that requests agreement to specify the decision values that are acceptable. Although this algorithm is an interesting solution for total-order broadcast, strictly speaking it is not a reduction to consensus (since consensus depends here on the application by the external validity property). Nevertheless, the reduction algorithm has a constant time complexity with respect to multi-valued Byzantine agreement with *external validity*. The approach also relies on the use of cryptographic tools, which is not the case for our reduction. The approach was later followed in several papers [KS01, RC05, DRS07].

Total-order broadcast is part of several Byzantine-tolerant group communication systems, where the implementation is either monolithic [MMS99] or it uses *view synchrony* [Rei94a, KMMS01, RPCS08]. All these protocols rely on the use of signatures, which is not the case for our reduction.

Although total-order broadcast can be used to implement state machine replication, several algorithms directly implement Byzantine state machine replication, *e.g.*, [CL02, KAD$^+$07, ACKL08]. The algorithm in [ACKL08] proceeds in two phases, pre-agreement and agreement phase, and in the agreement phase, similarly to our Algorithm 6.1, processes agree on message IDs instead on full messages. This idea of agreeing on message IDs was discussed by Ekwall and Schiper in [ES06], where the authors give a reduction of total-order broadcast to consensus in the benign fault model.

Reliable broadcast vs Reliable unique broadcast We have pointed out the difference between reliable broadcast and reliable unique broadcast, and the need to consider the latter in the reduction of total-order broadcast to consensus. It is not sure that this was clear for the authors of [CNV06, MNCV11a]. Indeed, in [CNV06] the authors use a primitive whose specification corresponds to reliable unique broadcast, but cite papers that refer to reliable broadcast [Bra84, HT94]. Moreover in [MNCV11a], where the authors refer to their protocol stack of [CNV06], they incorrectly refer to the reliable broadcast protocol of [Bra84]. Indeed, it can be shown (as done in Section 6.4.3) that the use of reliable broadcast instead of reliable *unique* broadcast in the reductions in [CNV06, MNCV11a] leads both to safety issues (violation of agreement and total-order) and liveness issues (violation of termination).

Validity property of consensus Besides the validity properties discussed in the chapter, other validity properties have been proposed. In [FG03], *differential consensus* is defined by introducing δ-differential validity, which requires that the decision value is of a certain plurality among the correct processes. It is not clear that such a validity property helps in the reduction of total-order broadcast.

Several papers have considered the *vector consensus* problem [DS97] as an intermediate step in solving total-order broadcast [CNV06, DS97]. Contrary to the consensus problem, in vector consensus the type of the decision differs from the type of the initial values (initial values of type T, decision of type $\mathsf{vector\ of\ T}$). As noted in [CNV06], vector consensus is an adaptation for asynchronous systems of *interactive consistency*, defined for synchronous systems [PSL80]. A similar approach is taken by [DGG00b] where total-order broadcast is solved by reduction to an abstraction called WIConsistency, a weaker variant of interactive consistency, which is itself implemented using the abstraction of muteness failure detectors [DS97].

An interesting observation is that range validity can be seen as a special case of the validity condition in approximate agreement [DLP$^+$86]. We can think about range validity consensus as a "perfect" approximate agreement problem with $\epsilon = 0$ where ϵ defines allowed difference among decision values. Interestingly in SIFT, a fault tolerant system for aircraft control [WLG$^+$78], range validity consensus would naturally fit in the algorithms for clock synchronization, stabilization of input from sensors, and agreement on results of diagnostic tests, where *interactive* *consistency* was used.

Consensus Reductions In [TC84], Turpin and Coan give an algorithm that reduces abortable or strong unanimity multi-valued consensus to binary consensus. By this algorithm, processes decide on a “default” value if correct processes do not have the same initial value. Therefore, it cannot be used to reduce multi-valued range validity consensus to binary consensus.

6.8 Conclusion

The chapter has discussed the relation between total-order broadcast and different variants of consensus in systems with Byzantine faults. It has shown that consensus with *weak unanimity* is not sufficient to solve total-order broadcast, while consensus with *strong validity* is harder than total-order broadcast. Furthermore, the chapter has shown that total-order broadcast is equivalent to consensus with *strong unanimity*, consensus with *abortable validity*, and consensus with *range validity*.

The chapter has also given a reduction of total-order broadcast to range validity consensus with constant time complexity with respect to consensus. Range validity consensus has been then reduced to binary consensus, also with constant time complexity. Together, this leads to a reduction of total-order broadcast to binary consensus, with constant time complexity with respect to binary consensus. To the best of our knowledge, these are the first total-order broadcast reductions to consensus with the constant time complexity with respect to consensus in the Byzantine fault model.

Acknowledgement We would like to thank Sam Toueg for having shared with us, a long time ago, a reduction of total-order broadcast to binary consensus in the context of Byzantine faults.

7 Bounded Delay in Byzantine-Tolerant State Machine Replication

The chapter proposes a new state machine replication protocol for the partially synchronous system model with Byzantine faults. The algorithm, called *BFT-Mencius*, guarantees that the latency of updates initiated by correct processes is eventually upper-bounded, even in the presence of Byzantine processes. BFT-Mencius is based on a new communication primitive, *Abortable Timely Announced Broadcast* (ATAB), and does not use signatures. We evaluate the performance of BFT-Mencius in the cluster settings, and show that it performs comparably to the state-of-the-art algorithms such as PBFT and Spinning in fault-free configurations, and outperforms these algorithms under performance attacks by Byzantine processes.

Publication: Zarko Milosevic, Martin Biely and André Schiper. Bounded Delay in Byzantine-Tolerant State Machine Replication. In *32th International Symposium On Reliable Distributed Systems (SRDS 2013)*, Braga, Portugal, September 30 - October 2, 2013.

7.1 Introduction

As explained in Section 2.3, state machine replication (SMR) is a general approach for replicating services that can be modeled as a deterministic state machine [Lam78, Sch90]. The key idea of this approach is to guarantee that all replicas start in the same state and then apply requests from clients in the same order, thereby guaranteeing that the replicas' state will not diverge.

Current deployments of SMR in industry handle benign faults only, with all major players employing some sort of replication in their infrastructure (e.g., Zookeeper [HKJR10], Chubby [Bur06], Dynamo [DHJ$^+$07]). However, SMR protocols that tolerate arbitrary failures (Byzantine faults) have started to be considered as a viable option for ensuring continuous operation of critical infrastructure control systems for electricity distribution, water treatment and traffic control [KGAS11].

Although Byzantine faults have already been introduced in 1980 [PSL80], Byzantine fault tolerant (BFT) replication protocols were considered too expensive to be practical [Rei94b]. This changed when Castro and Liskov introduced "Practical Byzantine faulty tolerance" (PBFT) [CL02]. They showed that in the fault-free settings BFT replication protocols can achieve performance that is close to that of non-replicated systems. The key observation that made PBFT practical was using MACs instead of signatures, which was the main performance bottleneck in previous systems [CL02]. Indeed, while our implementation of PBFT (cf. Section 7.7 for details) achieves peak throughput of 52K requests per second with average client-latency of 4.5 ms, once we introduced RSA [RSA78] signatures to PBFT, peak throughput drops to 6K requests per second with 20ms latency.

After PBFT, several similar approaches continued to improve performance in the fault-free case (e.g., Q/U [AEMGG$^+$05], HQ [CML$^+$06], Zyzzyva [KAD$^+$07], Aliph [GKQV10] and Zzyzx [HSGR10]). However, like PBFT, most of these protocols are fixed sequencer protocols [DSU04], i.e., a single server has a special role, namely to propose the order of requests. As already said in Chapter 1, Amir et al. showed in [ACKL11] that this class of protocols is vulnerable to *performance attacks*. The key observation is that a malicious sequencer can delay the ordering of requests, causing a considerable increase in latency and a great reduction in throughput. More precisely, a Byzantine server exhibiting performance failures sends messages according to the protocol, but delayed—typically just in time to avoid triggering protocol timeouts that will get them demoted. This makes it very hard to detect a faulty server and apply some kind of reconfiguration mechanism, in order to reduce the impact on the protocol. Bounding the service response time (or having other performance guarantees) is not only of theoretical interest. For instance, in Amazon's Dynamo [DHJ$^+$07], there is a formal Service Level Agreement (SLA) where a client and a service agree on the client's expected request rate distribution and the expected service latency under those conditions.

In order to design algorithms that tolerate performance attacks, Amir et al. [ACKL11] proposed a new performance criterion, called *bounded-delay* (see Chapter 1). Ensuring bounded-delay could be considered as capturing what one would informally describe as tolerating performance attacks. Unfortunately, this is not entirely true, because the definition of *bounded-delay* (see Chapter 1) does not tell us how big the upper-bound should be. Consider for example protocols that continuously rotate the leader, such as Aardvark [CWA$^+$09] or Spinning [VCBL09] (which do not consider bounded-delay as a criterion). It can be easily seen that such protocols are able to guarantee an upper-bound of $b \cdot T$, where b is the number of faulty servers and T is a protocol timeout whose expiration triggers reconfiguration mechanism such as view change. The intuition behind such claim is simple: in the worst case, from the time a correct server p has a request to propose (e.g., because it received it from a client) until it can propose the request, the server needs to wait until $n-1$ instances of other servers have terminated. If a process forwards the request to other processes when its not its turn, then it could be proposed faster by some other process (it does not need to wait $n-1$ instances before the request is proposed). However, even in this case, in the worst case, the next b instances may be coordinated by faulty servers, so the request will be proposed the earliest after b instances (owned by faulty processes) terminate. As the timeout values are normally chosen rather conservatively[1], the guaranteed bound that is roughly $b \cdot T$ can be significantly higher than the latency in the failure-free case. Thus, providing such bound does not necessarily match the intuition one has for *tolerating performance attacks*. One would rather expect that algorithms that tolerate performance attacks achieve the same order of performance as in the failure-free case, where latency is in the order of the communication delay among correct servers.

Apart from defining new performance criteria, Amir et al. also proposed in [ACKL11] a new BFT algorithm called *Prime*. Note that what Prime provides is in accordance with our arguments above, as its upper bound on latency is in the order of the communication delay among correct processes. Prime is derived from PBFT. Besides heavily relying on signatures, Prime adds a pre-agreement phase to PBFT. That is, servers exchange requests using a reliable broadcast protocol before the requests are actually ordered by what is essentially PBFT. While increasing the protocol complexity and the number of communication steps on the critical path, having this pre-agreement phase allows servers to compute a threshold of acceptable performance. The computed threshold is then used to judge if the sequencer is faulty. More precisely, executing reliable broadcast allows correct servers to come to a consistent view of what the current sequencer should do, for example, when the next instance should start and what requests should be proposed in the next instance. Without pre-agreement phase, i.e., in PBFT and similar protocols, this knowledge is only available at the sequencer.[2] Therefore, it is very hard (if not impossible) for a correct server to determine a bound that defines an acceptable level of performance in such centralized protocols. In a sense, adding a pre-agreement phase makes Prime a less centralized protocol, as in the pre-agreement phase all servers have the same role. So although still based on the fixed-sequencer scheme, being more decentralized allows Prime to reduce the impact a faulty sequencer can have on the protocol, by demoting the sequencer if it does not provide the acceptable level of performance. When comparing Prime with PBFT we see that ensuring the stronger performance criteria comes at a price: use of signatures makes it costly for cluster settings, and adding the pre-agreement phase increases protocol complexity and strongly affects performance in the failure-free case.

In this chapter, we propose a new BFT SMR protocol, BFT-Mencius, that ensures bounded-delay in the order of communication delay among correct servers, but does not incur additional costs (as Prime). The key to do so is to go one step further: Instead of adding a decentralized pre-agreement phase to PBFT as in Prime, BFT-Mencius is a fully decentralized protocol where all servers concurrently propose in different instances of a sub-protocol called ATAB (explained below). Because these instances are tightly coupled, servers can use their own progress in proposing requests to estimate the progress others should make.

In more detail, BFT-Mencius is inspired by Mencius [MJM08], an efficient multi-leader SMR protocol that tolerates (only) benign faults, originally designed for WAN settings. Mencius uses an (infinite) sequence of instances of a subprotocol referred to as *simple consensus*, where

[1]For example, the timeout in Aardvark is set to 40ms, which is order of magnitude bigger than a duration of a single instance during synchronous period.

[2]Note that rotating the sequencer is of no help as a correct server has a complete picture only while being the sequencer.

only a pre-determined initial leader can propose any value; the others can only propose a special *no-op* value. A basic idea of Mencius is to partition the sequence of instances among the servers such that each server is the coordinator in an infinite number of instances. For instance, servers can take the role of coordinating instances in a round-robin fashion. The difficult part of Mencius is (i) preventing servers that do not have requests to propose from blocking the protocol and (ii) dealing with instances where the coordinator is a faulty server. While the latter is handled inside *simple consensus*, the former is handled by servers allowing to skip their turns by proposing a special *no-op* request. The key to Mencius' performance is that simple consensus allows servers to skip their turns without having to execute the full agreement protocol, thereby requiring to wait for messages from a majority of servers. In fact, Mencius even allows servers to skip implicitly by participating in higher numbered instances started by "faster" servers.

As Mao et al. [MJM08] pointed out, such mechanism does not work in the Byzantine tolerant systems, because not all decisions are communicated through a quorum. Therefore, we designed BFT-Mencius using an abstraction that we call *Abortable Timely Announced Broadcast (ATAB)*. ATAB is a new broadcast primitive, that is similar to *Timely Announced Broadcast* [ABH$^+$11] and *Terminating Reliable Broadcast* (TRB). In contrast to these two, it is specified such that it can be implemented in the partially synchronous system model. Like these two primitives (owing to the fact that it is a broadcast abstraction) each instance of ATAB has a dedicated sender. BFT-Mencius let servers skip their turns by proposing *no-op* requests, and relies on ATAB to terminate instances with faulty dedicated sender within bounded time. Furthermore, ATAB allows to tie the start time of different instances together. This enables correct servers to compute (during the synchronous period) a threshold for "acceptable speed" with which servers should start and terminate their ATAB instances. This knowledge is used in the blacklisting mechanism, which ensures that faulty servers behave according to this bound (otherwise they get blacklisted and their subsequent ATAB instances ignored).

As we mentioned before, although BFT-Mencius provides strong performance guarantees, this does not penalize performance in the failure-free case, where the latency and throughput are comparable to state-of-the-art algorithms such as PBFT and Spinning [VCBL09]. We implemented a prototype of BFT-Mencius and evaluated its performance in cluster settings. We show that it achieves good performance, both in fault-free configurations and under performance attacks by Byzantine servers. For example, BFT-Mencius achieves a throughput of 45K requests (20B of payload) per second with latency always below 5ms even under performance attacks.

Contribution

We propose a new BFT state machine replication protocol, BFT-Mencius, that guarantees a bounded-delay in the order of PBFT's latency in the failure-free case. Thus it *tolerates performance attacks.* Key to achieving this is by concurrently running multiple instances of a new broadcast-abstraction called ATAB, which *does not use signatures.* This makes BFT-

Mencius a *modular* protocol, which, we think, makes it simpler to understand, implement and test than other BFT SMR protocols (which are in general monolithic and often rather complex).

Roadmap The remainder of the chapter is as follows: We discuss related work in Section 7.2. Section 7.3 defines the system model considered. ATAB is introduced in Section 7.4, and the total-order broadcast algorithm based on ATAB in Section 7.5. The complete BFT-Mencius algorithm is given in Section 7.6. We evaluate the performance of BFT-Mencius in Section 7.7 and conclude in Section 7.8.

7.2 Related work

As already noted in Section 7.1, BFT-Mencius is inspired by Mencius [MJM08]. In a recent position paper [MJM09], the same authors discuss the design of RAM, which is a variant of Mencius that tolerates Byzantine faults by relying on trusted components. However, the paper does not consider performance attacks and does not ensure bounded-delay. Since tolerating performance attacks is central to achieving bounded delay, in the following we focus on work that considers performance attacks.

Amir et al. [ACKL11] showed that malicious processes can significantly reduce throughput and increase the service latency in previous BFT protocols, by sending valid messages but as slow as possible without triggering timeouts. Moreover, they also introduced a protocol called Prime that ensures bounded delay. We have already discussed Prime in Section 7.1 in detail.

Although Prime is the only protocol (before BFT-Mencius) that considers bounded-delay, it is not the only work that considers performance attacks by Byzantine processes. In a similar spirit, Clement et al. [CWA$^+$09] have advocated what they called robust BFT. That is, they propose to shift the focus from algorithms that optimize only best case performance to algorithms that can offer predictable performance under the broadest possible set of circumstances—including when faults occur. To this end they propose a set of mechanisms to increase robustness of PBFT, in a system called Aardvark. Aardvark decreases the impact of slow leader by constantly monitoring the throughput sustained in the current view and by regularly performing view-changes. The idea of always rotating the sequencer was also used in BAR-B [AAC$^+$05], but with a different purpose. BAR-B is a cooperative backup system designed for the Byzantine-Altruistic-Rational model, where the leader is always changed so all nodes have equal opportunity to submit proposals to the system. Continuously changing the leader is also used in Spinning [VCBL09], with the goal to reduce the effect of performance attacks by Byzantine processes. In Spinning, the leader is changed after it defines the order of a single batch of requests, making leader change more efficient under performance attacks than with Aardvark, since there is no need for a complex view change protocol[3]. Spinning

[3]The equivalent of the view change protocol in Spinning is called *merge* and it is executed when a sufficient

is also more efficient than BAR-B because it does not use signatures and ordering a request takes three communication steps compared to six with BAR-B. In order to prevent a faulty process from periodically impairing the protocol performance by requiring merge phases, Spinning introduces a blacklisting mechanism. Processes that are in the blacklist loose the privilege to propose, i.e., instances where they are primary are skipped. After a successful merge phase in view v, the primary of the view $v-1$ is added to the blacklist. As the merge phase is triggered once the timeout expires in view $v-1$, this blacklisting mechanism is still vulnerable to performance attacks. BFT-Mencius also contains a blacklisting mechanism, but it uses a different detection mechanism that makes it very effective in detecting processes exhibiting performance attacks.

The common property of all three protocols, Aardvark, BAR-B and Spinning, is that all servers are allowed to propose requests only once it is their turn. As already explained in Section 7.1, these algorithms ensure bounded-delay, but with an upper-bound in the order of $b \cdot T$. As we show in Section 7.6, BFT-Mencius does not have such limitation: its latency does not depend on number of faulty processes, and is in the order of the actual communication delay among correct processes.

More recently, Aublin et al. proposed a new approach called RBFT (Redundant-BFT) [ABQ13] where $b+1$ instances of the same BFT-SMR protocol (in their case PBFT) are executed in parallel, each with a primary executed on a different server. Although all these instances order requests, only requests ordered by one instance (called master instance) are executed. The other instances are used only to monitor whether the master instance provides adequate performance. In order to be able to monitor performance of the master instance, the load on all servers need to be the same. Therefore, RBFT adds a dissemination phase (called PROPAGATE phase) to the underlying BFT-SMR protocol, in which a server, upon receiving some client request for the first time, forwards it to all servers. This ensures that a request received by a correct server is eventually received by all correct servers. RBFT is more robust to performance failures than Prime, Aardvark and Spinning, and performs comparably to these algorithms in the failure-free case. Compared to BFT-Mencius, RBFT uses more hardware resources as the underlying BFT-SMR protocol is executed $b+1$ times in parallel [4], and it adds the PROPAGATE phase, increasing therefore latency (even during failure-free runs) and protocol complexity of the underlying BFT-SMR protocol.

As mentioned above, performance attacks exploit the fact that there is a process with a special role (leader, coordinator, sequencer). So in order to reduce the impact of performance attacks, one approach is to make the protocol decentralized (i.e., leader-free). We are aware of only one deterministic BFT protocols for partially synchronous systems that has this property [BS10]. However, the algorithm terminates only in the order of $b \cdot T$ even in the failure-free case. The same applies to protocols for synchronous systems like [PSL80].

number of correct processes suspect the current leader as faulty.

[4]The experiments shown in [ABQ13] are for cluster settings where nodes have two quad-core CPUs with ten network interfaces. It is not clear what performance can be expected from RBFT in more "standard" cluster settings where nodes have only one or two network interfaces.

Contrary to deterministic BFT protocols, most randomized BFT protocols (e.g., [CKPS01, MNCV11b]) are decentralized (there are no processes with a special role) and work in the asynchronous systems. Therefore faulty processes cannot prevent correct processes from moving forward by delaying messages. However, these protocols are normally more costly in the failure-free case than deterministic protocols, due to a higher number of communication steps.

7.3 Definitions

7.3.1 Model

We consider a system composed of n server processes $\Pi = \{1, \dots, n\}$ and a finite number of clients processes connected by point-to-point channels. Servers and clients can be *correct* or *faulty*, where correct servers and clients follow the algorithm and the faulty one may behave in an arbitrary way, i.e., we consider Byzantine faults. We assume than any number of clients can be faulty, but the number of faulty servers is limited to b.

We assume integrity of channels, that is if a process p received a message m from process q, then q sent message m to p before. We consider a partially synchronous system model: in all executions of the system, there is a bound Δ and an instant GST (Global Stabilization Time) such that all communication among correct processes after GST is reliable and Δ-timely, i.e., if a correct process p sends message m at time $t \geq GST$ to some correct process q, then q receives m before $t + \Delta$. We do not make any assumption (on the communication among correct processes) before GST. For example, messages among correct processes can be delayed, dropped or duplicated before GST. Spoofing/impersonation attacks are assumed to be impossible also before GST.

We assume that process steps (which might include sending and receiving messages) take zero time. Processes are equipped with clocks able to measure local timeouts.

7.4 Abortable Timely Announced Broadcast

In this section we introduce a new broadcast primitive called *abortable timely announced broadcast* (ATAB), that we will use later to solve Total-Order broadcast. ATAB is defined in terms of four primitives: *atab-cast*(m), *atab-abort*, *atab-announce*, *atab-deliver*(m). The first two primitives are invoked by processes, while the latter two are triggered by the protocol that implements ATAB. Moreover, each ATAB instance has a dedicated sending process[5] s and no correct process $p \neq s$ executes *atab-cast* for that instance. When a process delivers a message m it executes *atab-deliver*(m). Like with *timely announced broadcast* (TAB) [ABH$^+$11] a process is notified of an ongoing broadcast by *atab-announce* before a message is actually delivered. In contrast to TAB we require all correct processes to eventually execute *atab-deliver*,

[5]If a correct process has a message m to broadcast, it executes *atab-cast*(m).

much like *terminating reliable broadcast* (TRB). Like TRB we also allow delivery of a special value $\perp$. Unlike TRB which is designed for synchronous systems[6] and benign faults, ATAB can be solved in the partially synchronous system model with Byzantine faults. Therefore, we allow $\perp$ to be delivered only if some correct process aborts the broadcast by invoking *atab-abort*. Typically, this occurs when the sending process s is suspected of being faulty. It is known that in the presence of Byzantine faults failure detection requires application knowledge [DS98]. The idea of *atab-abort* is to make this knowledge explicit, and the idea of *atab-announce* is to help with failure detection.

An algorithm solves ATAB with parameters d_1 and d_2, such that $d_1 \geq d_2$, if the following properties hold:

- *ATAB-Agreement*: If a correct process executes *atab-deliver*(m), then every correct process eventually executes *atab-deliver*(m).
- *ATAB-Integrity*: A correct process executes *atab-deliver*(m) at most once. Furthermore, if s is a correct process and s executed *atab-cast*(b) then $m \in \{\perp, b\}$.
- *ATAB-Termination*: If all correct processes execute either *atab-cast*, or *atab-announce*, or *atab-abort* then every correct process eventually executes *atab-deliver*.
- *ATAB-Validity*: If a correct process s executes *atab-cast*(m) at time $T \geq GST$ then all correct processes execute *atab-deliver*(m) before $T + d_1$, unless *atab-abort* has been executed before by a correct process.
- *ATAB-Announcement*: If a correct process p executes *atab-deliver* at time T, then it executed *atab-announce* before T. Furthermore, if at time T, p executes *atab-deliver* or *atab-cast*, then every correct process executes *atab-announce* before $\max\{T, GST\} + d_2$.

7.4.1 Solving ATAB

Algorithms that solve ATAB have similarities with algorithms that solve consensus. The sender process execute *atab-cast*(m) where it would execute $propose(m)$ in the consensus protocol, while *atab-deliver*(m) is triggered at the point the consensus protocol invokes $decide(m)$. The main difference is adding the *atab-announce* up-call and reacting upon the *atab-abort* down-call. As consensus protocols normally proceeds in a sequence of views (sometime also called rounds, ballots or phases), the protocol would normally react on *atab-abort* call by changing the view.

We now informally explain the algorithm that solves ATAB, which is close to the CL consensus algorithm, Algorithm 3.5 (itself inspired by the PBFT SMR algorithm by Castro and Liskov). We call it CL-ATAB, and it requires $n \geq 3b + 1$ processes to tolerate at most b Byzantine faults. The full algorithm and description (with proofs and timing analysis) is given in Appendix A.

[6]TRB has been shown to require synchronous systems or an asynchronous system with a perfect failure detector.

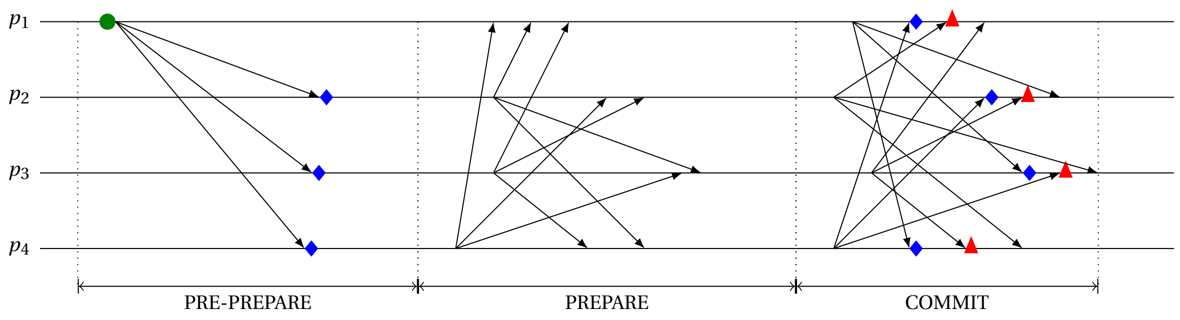


Figure 7.1: Solving ATAB: message pattern of initial view of the protocol. Process p_1 is the (correct) sender; its sending event (*atab-cast*) is indicated by the circle. Delivery of messages (*atab-deliver*) is indicated by triangles, while diamonds indicate *atab-announce* events. $n = 4$, $b = 1$.

Contrary to PBFT which solves SMR (multiple instances problem), CL-ATAB solves a single instance problem ATAB.[7] The algorithm proceeds in views, such that in every view there is a single process that is the coordinator (of the view). The assignment scheme of views to coordinators is known to all processes. The sender s is the coordinator of the initial view. The initial view of the protocol is very similar to the "normal case" protocol of PBFT with addition of *atab-announce* upcalls as shown in Figure 7.1. A process executes *atab-announce* once it receives the `PRE-PREPARE` message from the sender, or once it receives `COMMIT` messages from $b+1$ processes. The latter is related to *ATAB-Announcement*. It ensures that once a correct process executes *atab-deliver*, all correct processes eventually execute *atab-announce*. The reason is the following: Once a correct process executes *atab-deliver*, it received a `COMMIT` message from a quorum of processes (in case $n = 3b+1$, the size of quorum is $2b+1$). Since we assume that $n \geq 3b+1$, at least $b+1$ correct processes sent `COMMIT` to all. In CL-ATAB, we assume that messages sent by correct processes are periodically retransmitted, so all correct processes eventually receive $b+1$ `COMMIT` and execute *atab-announce*.

The protocol of all subsequent views of CL-ATAB is very similar to the "view change protocol" of PBFT, with the difference that CL-ATAB deals only with a single instance problem. A process p advances from view v to view $v+1$ for a number of reasons:

- p leaves the initial view $v = 1$ upon execution of *atab-abort*.
- p enters view $v+1$ after the timeout expires, and it has not yet learned what message should be delivered. The timeout is triggered by the execution of *atab-announce*, or when p receives `VIEWCHANGE` for view v from quorum of processes.
- Finally, p moves to view $v+1$ if it receives `VIEWCHANGE` from a correct process that is in view $v+1$.

[7]Thus, the relation between ATAB and PBFT can be considered similar to that of the Synod and Parliament protocols of [Lam98].

The following result is proven in Appendix A:

Theorem 7.1. *If $n \geq 3b+1$, then the CL-ATAB algorithm solves ATAB in the partially synchronous system model with known Δ, $d_1 = 3\Delta$ and $d_2 = 2\Delta$.*

7.5 Solving Total-Order Broadcast with ATAB

In this section, we present the central part of our BFT SMR protocol, which solves total-order broadcast, see Algorithm 7.1. It is inspired by Mencius [MJM08], an efficient multi-leader SMR. The algorithm relies on the ATAB primitive for sending messages and refers to the parameters d_1 and d_2.

7.5.1 Basic idea

Algorithm 7.1 runs an infinite sequence of ATAB instances. We add a number i to ATAB calls to refer to ATAB instance i, e.g., *atab-cast*(i, m). These instances are evenly partitioned among the servers. Function $owner(i)$, known to all processes, returns the sender for instance i.

For every process p, $index_p$ is the next ATAB instance in which p is the sender. In order to to-broadcast a message m a process p executes *atab-cast*$(index_p, m)$ in the ATAB instance $index_p$ (line 9) and then updates $index_p$ (line 10).

Once a process p learns that ATAB instance i terminated (execution of *atab-deliver*(i, m), line 11), it executes the following steps:

- If (i) p was the sender in instance i, i.e., $owner(i) = p$, (ii) p has sent m and (iii) p learns that $m' \neq m$ is delivered (necessarily $m' = \perp$), then p to-broadcasts m again (line 17).

- Process p executes the $CheckCommit$ procedure (lines 24–30), which uses the $expected_p$ variable to keep track of the lowest yet undecided ATAB instance. Inside $CheckCommit$, p increases $expected_p$ as far as possible, executing to-deliver for all messages that are not *noop*.

Note that according to the $CheckCommit$ procedure, p executes to-deliver(m) in the order of ATAB instances. Therefore the message delivered in instance i cannot be adelivered before all instances $j < i$ have terminated. Since processes might to-broadcast at different rates, we need a mechanism to allow processes to fill the gaps so that the message delivered by ATAB instance i is not delayed because another process has nothing to broadcast in instance $j < i$. This is discussed in the two next paragraphs.

7.5.2 Process p skipping its own instances

In order to fill these gaps, a correct process will *skip* its instance j by broadcasting a special message *noop*. A process could execute *atab-cast*$(j, noop)$ when it sees that there is a decision in some instance $i > j$. However, skipping instance only once a higher number instance i terminates is unnecessarily late as processes know that instance i is in progress already before decision. More precisely, in case a correct process p sees that some other process, say q, broadcasted in instance $i > index_p$ (by executing *atab-announce* in instance i, line 20), then p skips all instances j with $index_p \leq j < i$ where p is the sender (lines 21-23). Here, the other correct processes handle instance j like any other instance owned by p. This is not the case in the next paragraph.

7.5.3 Process p skipping instances of other processes

The skipping mechanism of the previous paragraph is able to fill only those gaps caused by correct processes not broadcasting. Indeed, we cannot require a (Byzantine) faulty process to skip its instances. Therefore, a different mechanism is needed for the instances owned by a faulty process. Put differently, correct processes need a means to ensure that instances owned by faulty processes will terminate. ATAB provides the *atab-abort* primitive to this end. However, executing *atab-abort* too early might lead to deliver $\perp$ in ATAB instances owned by correct processes. This can be avoided during a synchronous period, i.e., after GST, with a timeout of $d_2 + d_1$, see lines 6 and 19. Consider some process p that terminates instance i at time T. By *ATAB-Announcement*, all correct processes execute *atab-announce* for instance i at latest at $T + d_2$. By line 22, these processes execute *atab-cast* in all instances $j < i$ for which they are sender (if they did not already *atab-cast*). By *ATAB-Validity*, these instances will all terminate within d_1, that is before $T' = T + d_2 + d_1$. Since process p does not execute *atab-abort*$_p(j)$ for the yet undecided instances j before T', instances owned by correct processes are indeed not aborted too early after GST (see lines 19 and 31–33).

7.5.4 Correctness proof

Lemma 7.1. *Assume a correct process s calls atab-cast(i, m) (in line 9) at time σ. If $\sigma > GST + d_2$ then no correct process aborts instance i.*

Proof. Assume by contradiction that some correct processes aborted before deciding and let q be the first to abort at time t_q (line 33). Since the process q aborted in instance i, this means that q decided in some instance $j > i$ at time $t_q - timeout$. By *ATAB-Announcement*, all correct processes (including s) announced in instance $j > i$ the latest at time $\max\{t_q - timeout, GST\} + d_2$. When s executes *atab-announce*(j) (lines 20ff.) it updates its $index$ until it reaches some $k > j$ (lines 21 and 23). If $t_q - timeout < GST$, then s announced at $GST + d_2$, which means that at time $\sigma > GST + d_2$ it $index_s$ is at least k, such that $i \geq k > j > i$. A contradiction. Otherwise, we have two cases, either s announced before executing

Algorithm 7.1 Total Order Broadcast with ATAB

```
Initialization:
    $proposed_p[] := \perp$                                  /* initially, for all $i$, $proposed_p[i] = \perp$ */
    $decided_p[] := \perp$                                   /* initially, for all $i$, $decided_p[i] = \perp$ */
    $expected_p := 0$                                        /* lowest undecided ATAB instance */
    $index_p := min\{i : owner(i) = p\}$                     /* next instance owned by $p$ */
    $timeout_p := d_1 + d_2$

upon to-broadcast($m$) do
    $proposed_p[index_p] = m$
    $atab\text{-}cast(index_p, m)$
    $index_p \leftarrow min\{i : owner(i) = p \wedge i > index_p\}$

upon $atab\text{-}deliver(i, m)$ do
    if $m \neq \perp \wedge owner(i) = sender(m)$ then
        $decided_p[i] \leftarrow m$
    else
        $decided_p[i] \leftarrow noop$
    if $p = owner(i) \wedge proposed_p[i] \notin \{m, noop\}$ then
        to-broadcast($proposed_p[i]$)
    $CheckCommit$
    after $timeout_p$ execute $OnTimeout(i)$

upon $atab\text{-}announce(i)$ do
    while $index_p \leq i$ do
        $atab\text{-}cast(index_p, noop)$
        $index_p \leftarrow min\{i : owner(i, 1) = p \wedge i > index_p\}$

Procedure $CheckCommit$:
    while $decided_p[expected_p] \neq \perp$ do
        $m \leftarrow decided_p[expected_p]$
        $o \leftarrow owner(expected_p)$
        if $m \notin \{noop\} \cup \{decided_p[i] : i < expected_p\}$ then
            to-deliver($m$)
        $expected_p \leftarrow expected_p + 1$

Procedure $OnTimeout(i)$:
    for each $k \in \{j \in [expected_p, i] : decided_p[j] = \perp \wedge owner(j) \neq p\}$ do
        $atab\text{-}abort_p(k)$
```

atab-cast(i, m), then $i \geq k > j > i$ as above and we have reached a contradiction again; or s announced after executing *atab-cast*(i, m), then as s announced before $t_q - timeout + d_2 = t_q - d_1$, it also called *atab-cast*(i, m) before this point in time, i.e., $\sigma < t_q - d_1$ Since q is the first to abort no process aborts before t_q, that is no process aborts before $\sigma + d_1$. Therefore by *ATAB-Validity* all correct processes decide before t_q. Also a contradiction. □

Lemma 7.2. *If a correct process s calls* to-broadcast(m), *then all correct processes eventually* to-deliver m.

Proof. When s calls to-broadcast(m), it executes *atab-cast*(i, m) for some i. If it does so after $GST + d_2$, then from Lemma 7.1 it follows that instance i will not be aborted, and therefore *ATAB-Validity*, ensures that all processes will execute *atab-deliver*(i, m) and thus set $decided[i] \leftarrow m$. Eventually, all instances $j < i$ will terminate as well, so processes will execute $CheckCommit$ with $decided[j] \neq \perp$ for all $j \leq i$, and will thus to-deliver m.

Otherwise, if s calls *atab-cast*(i, m) before $GST + d_2$, instance i might get aborted and $\perp$ may be delivered. If this is the case, processes will set $decided[i] \leftarrow noop$. Upon doing so, s will by line 17,re-call to-broadcast(m) implying some $i' > i$ for which s executes *atab-cast*(i', m). What, therefore, remains to be shown is that there is some k such that s executes *atab-cast*(k, m) and *atab-deliver*(k, m). (*ATAB-Agreement* implies that all others will also execute *atab-deliver*(k, m).)

We show this by contradiction and assume that there is no k such that s executes *atab-deliver*(k, m). Then line 17 entails that there is an infinite sequence of instances that all decide $\perp$ although m was proposed. Clearly, one of them must start at some time $\sigma > GST + d_2$, which by Lemma 7.1 implies that s will *atab-deliver* within d_1. A contradiction. □

Proposition 7.1. *Algorithm 7.1 reduces Total-Order Broadcast to ATAB.*

Proof. *TO-Validity* follows from Lemma A.12.

From the *ATAB-Agreement* and *ATAB-Integrity* properties it follows that if p executes 13 for some i and $v \in \mathcal{M}$ then so will any correct q, and both will do so exactly once. Since these non-$\perp$ values of $decided_p$ determine for which messages m and processes s that p executes to-deliver$_p(m)$ for, *TO-Agreement*follows.

We now turn to the first part of *TO-Integrity*: The while-loop of $CheckCommit$ ensures that p executes to-deliver(m) for every value of expected at most once. The condition of line 28 then entails that m was not adelivered for a previous value of $expected$. Thus every to-deliver$_p(m)$ is executed exactly once. Since we have already shown *TO-Agreement*, it suffices to show for the second part of *TO-Integrity*, that the correct process $s = sender(m)$ will only to-deliver$_s(m)$ if it previously executed to-broadcast$_s(m)$. Since s executes to-deliver(m) with $m = decided_s[expected]$ it follows that $m \neq noop$. Further, since $decided_s[i]$ with $i = expected$ can have been set to a non-*noop* message only in line 13 it follows that s executed *atab-deliver*(i, m) such that $owner(i) = s$ before. Since s is correct and $m \neq noop$, it follows from *ATAB-Integrity*of that s executed *atab-cast*(i, m), which it must have done in line 9, that is in a call of to-broadcast(m).

TO-Order follows from the fact that if p executes to-deliver$_p(m, s)$ before to-deliver$_p(m', s')$ then there are j and j' such that $j < j'$, $decided_p[j] = m$, $decided[j'] = m'$, $s = owner(j)$ and $s' = owner(j')$. Now when q executes to-deliver$_q(m', s')$ it does so when $expected = j'$, since $j < j'$ it follows that it executed line 29 with $expected = j$ before. From the argument on *TO-Agreement* above, it is clear that at this point $decided_p[j] = m$, thus q executed to-deliver$_q(m, s)$ before. Now *TO-Order* follows from *TO-Integrity*. □

7.6 BFT-Mencius

In this section we describe the complete BFT-Mencius protocol for SMR that is based on the total-order broadcast algorithm 7.1.

7.6.1 Generalities

In BFT-Mencius clients send requests to servers (details below), which use total-order broadcast to order requests. After a request is executed by some server, the server sends the reply to the corresponding client. A client accepts a response only once it received $b+1$ identical responses from $b+1$ servers.

In Byzantine fault tolerant state machine replication only requests proposed by clients should be executed. This requirement is trivially ensured by using cryptographic signatures to sign client requests. Request authentication can also be achieved using MACs [CL02, AABC08a]. Although BFT-Mencius would in principle also work with other ways of request authentication, we assume in this chapter that request authentication is done using MACs as in [CL02] and other protocols that will be compared experimentally with BFT-Mencius.

In BFT-Mencius each server is able to propose requests, thus different variants of load balancing of client requests can be used. However, finding the optimal load balancing scheme is outside the scope of this chapter. For simplicity, we assume here a static assignment of client to servers based on their id. Servers propose requests they receive from clients assigned by this scheme.

7.6.2 Dealing with faulty servers

In the presence of faulty servers some clients are assigned to faulty servers. In order to ensure that requests from such clients will be ordered and executed, an additional mechanism is necessary: clients send each request to all servers,[8] and servers keep track of requests not assigned to them. They propose any requests that are not executed within some time. For instance, if a server finds a request req that is not executed after the server has terminated k of its own ATAB instances, the server proposes req.[9] In our experiments we have set $k = 3$. Thus faulty servers cannot starve clients by ignoring their requests.

Moreover, since we use a (numbered) sequence of instances we have to prevent a faulty server from exhausting the space of sequence numbers by starting ATAB instance with a very large instance number. To this end, every server can have at most one outstanding non-decided instance. Put differently, a server will not react on messages received for some instance j owned by a process q if it has not terminated in all instance $k < j$ owned by q.

7.6.3 Dealing with slow servers

The mechanism presented in the previous section addresses the problem of requests sent by clients assigned to faulty servers. Here we address the problem of faulty servers slowing down the ordering of requests assigned to *correct* servers. In the total-order broadcast algorithm

[8]Since messages might be lost before GST, we actually assume clients periodically retransmit.

[9]In fact it is sufficient if requests assigned to a certain server are tracked only by b other servers (instead of all).

Algorithm 7.2 Blacklisting mechanism

```
1:  Initialization:
2:     $blacklist_p := \emptyset$
3:     $\forall q \in \Pi : suspects[q]_p := \emptyset$
4:     /* see also Algorithm 7.1 */

5:  upon $suspect(q)$ do
6:     if $q \notin blacklist_p$ then
7:        $abcast(\langle \text{SUSPECT}, q \rangle)$

8:  Procedure $CheckCommit$:
9:     while $decided_p[expected_p] \neq \bot$ or $owner(expected_p) \in blacklist_p$ do
10:       $m \leftarrow decided_p[expected_p]$
11:       $o \leftarrow owner(expected_p)$
12:       if $m = \langle \text{SUSPECT}, q \rangle$ then
13:          $UpdateBlackList(q, o)$
14:       else if $m \notin \{noop\} \cup \{decided[i] : i < expected_p\}$ then
15:          $adeliver(m, owner(expected_p))$
16:       $expected_p \leftarrow expected_p + 1$

17: Procedure $UpdateBlackList(q, o)$ :
18:    if $q \notin blacklist_p$ then
19:       add $o$ to $suspects_p[q]$
20:       if $|blacklist_p[q]| \geq b + 1$ then
21:          add $q$ to $blacklist_p$
22:          $suspects_p[q] \leftarrow \emptyset$
```

(Algorithm 7.1) a faulty server can do a performance attack by delaying its own instances. This is because the message of ATAB instance i is not delivered until all instances $j < i$ have terminated. We address this issue by introducing a *blacklisting mechanism* used as follows (we discuss when to suspect servers in Section 7.6.4): ATAB instance i waits for the termination of only those instances $j < i$ that are not owned by servers on the blacklist. That is, instances whose owners are in the blacklist are skipped, i.e., the effect is equivalent to the case where *noop* was decided. Therefore, it is important that the blacklist is kept consistent among all servers. Because the blacklist may be seen as a state machine, we can use our SMR protocol to ensure consistency (similar to how reconfiguration in benign systems can be done [Lam98]).

The blacklist is implemented as a circular buffer of size b, thus adding the $(b+1)$-st server will rehabilitate the server that is longest in the list. In order for server p to consistently inform other servers that it suspects q to be faulty, p to-broadcasts the special request $\langle \text{SUSPECT}, q \rangle$. According to Algorithm 7.2, this request is then broadcasted using p's next ATAB instance, and finally stored in all correct servers $decided$ list at the same position. Note that SUSPECT messages are actually piggy-backed on the normal messages or *noop* messages that are to-broadcast. A server p adds a server q to its (server of the) blacklist once $b+1$ servers suspected q.

The blacklisting mechanism is part of Algorithm 7.2. It includes: (i) the $suspect(q)$ function, used by server p to locally trigger the blacklisting mechanism once it suspects some server q, (ii) a modified version of the $CheckCommit$ function of Algorithm 7.1, and (iii) function $UpdateBlackList(q, o)$ called by $CheckCommit$. The function $UpdateBlackList(q, o)$ (line 17) maintains the blacklist at server p: it is executed whenever p learns that server o suspects server q. The function first checks if q is not already in the blacklist. If not, o is added to $suspects[q]$. If at this point there are $b+1$ different servers that suspect q, server q is added to the $blacklist$ and $suspects[q]$ is cleared. Note that, since $UpdateBlacklist(q, o)$ is called by $CheckCommit$ when $\langle \text{SUSPECT}, q\rangle$ is decided, the correct servers always have a consistent view of the blacklist. That is, for each value of $expected$, the blacklist is the same at all servers.

Since ATAB instances of server p are ignored while p is on the blacklist, messages to-broadcast by p cannot be delivered. Thus once p is added to the blacklist, p's clients are reassigned to servers not in the blacklist. At this point all requests from reassigned clients not executed will be proposed by the newly assigned server.[10]

7.6.4 Ensuring Bounded Delay

The blacklisting mechanism is very general, i.e., it can be used to report any suspicious behaviour. As we are interested in ensuring bounded delay, we use it to report when a server is slow. To do so we rely on the properties of ATAB that hold during a synchronous period, i.e., after GST. More precisely, once a correct server executes *atab-cast*(i, m) at time t in line 9 of Algorithm 7.1, by the *ATAB-Announcement* property we know that all correct servers execute *atab-announce*(i) the latest at time $t + d_2$. Therefore, all correct servers start their instances j, with $j < i$, the latest at time $t + d_2$. By *ATAB-Validity* all such instances terminate the latest at time $t + d_2 + d_1$. Therefore, if at time $t' > t + d_2 + d_1$, some instance $j < i$ has not terminated, the owner of instance j is suspected and $suspect(owner(j))$ is executed. The owner of some instance i is also suspected if a server executes *atab-abort* for instance i (line 33 of Algorithm 7.1). This solution guarantees bounded delay that does not depend on the number of faulty servers.

7.6.5 Improving the Delay Bound

However, d_1 and d_2 are worst case bounds. For blacklisting,[11] we would like to replace the $d_1 + d_2$ timeout by a smaller value, that is in the order of the real communication delay. To do so, similarly to Prime [ACKL11], we assume that network eventually meets certain stability conditions. More precisely, we assume that after GST the duration of ATAB instances running concurrently do not differ substantially between owning servers. Let d_{ATAB} be the duration of ATAB instances measured by some server. This leads us to estimate the "actual value" of d_1 in the run as d_{ATAB}, and since $d_2 < d_1$, we (conservatively) also estimate d_2 as d_{ATAB}. This leads us to use $2 \cdot K_{lat} \cdot d_{ATAB}$ as a timeout to suspect servers, where K_{lat} is a system parameter that accounts for variability in latencies on the network. As we will show in Section 7.7, in the cluster settings, $K_{lat} = 1$ is sufficient to reliably detect a faulty server doing a performance

[10]This mechanism is different from the delayed re-proposing of requests mentioned at the beginning of Section 7.6: the reassignment mentioned here causes instantaneous re-proposing when a server is added to the blacklist.

[11]Note that we only change the timeout for suspecting processes in the blacklisting mechanism. The timeout in Algorithm 7.1 is not modified.

attack. In less homogeneous settings, it could be preferable to use a more complex way to determine this timeout, for example by taking the median of recent durations of ATAB instances owned by different servers, by adding additional weight factors, or just by using a bigger value for K_{lat}.

In addition to suspecting servers based on latency, it is possible to use additional strategies for detecting faulty servers. For example, one could measure the number of requests executed in the last n instances owned by each server, and suspect servers whose number is less than 50 percent of the average number of requests executed by servers. Aardvark [CWA+09] uses a similar idea to suspect the current leader. While this and other techniques of suspecting the current leader based on performance or fairness criteria easily carry over from Aardvark and other protocols, we did not consider them for BFT-Mencius, because our main goal is the ability to ensure bounded delay. In any case, if the suspicion mechanism used leads to frequent changes of the blacklist, this can only lead to requests being proposed by multiple servers thereby causing performance degradation.

7.6.6 Blacklisting cannot impair liveness

In this section we show that the blacklisting mechanism, even with a suspect timeout of 0 (meaning that correct processes are continuously being suspected by other correct processes), cannot prevent the progress of our SMR algorithm. Let $n = 3b+1$, and assume by contradiction that there is a request req which is never executed. Then after the re-propose timeout (see beginning of the Section 7.6) expired at all correct processes, every correct process will propose req in it's next ATAB instances. Let us assume that at time t all re-propose timeouts for req have expired at all correct processes. As the request req is never executed, all instances started after time t are either skipped or owned by a faulty process. As there are at most b processes in the blacklist, but $2b+1$ correct processes, which have to be in the blacklist whenever their instances would be decided, the blacklist needs to be continuously updated after time t (otherwise req would be delivered by the first instance started after t and owned by a correct process).

So each correct process q is repeatedly added to the blacklist. Since $suspected[q]$ is reset when q is added to the blacklist, in order for q to be added again later, it is necessary that $b+1$ processes suspect q, i.e., $b+1$ ATAB instances owned by different processes need to deliver $\langle \mathsf{SUSPECT}, q\rangle$ without being skipped (see Algorithm 7.2). As there are at most b faulty processes, this means a correct process p has to suspect q in a successful instance i started after t. But then p also proposed req in that instance, and req is executed. A contradiction.

7.7 Evaluation

We have implemented a prototype of BFT-Mencius in Scala using the Distal framework [BDMS13]. Distal is a new framework that allows writing code in a domain specific language (DSL) that

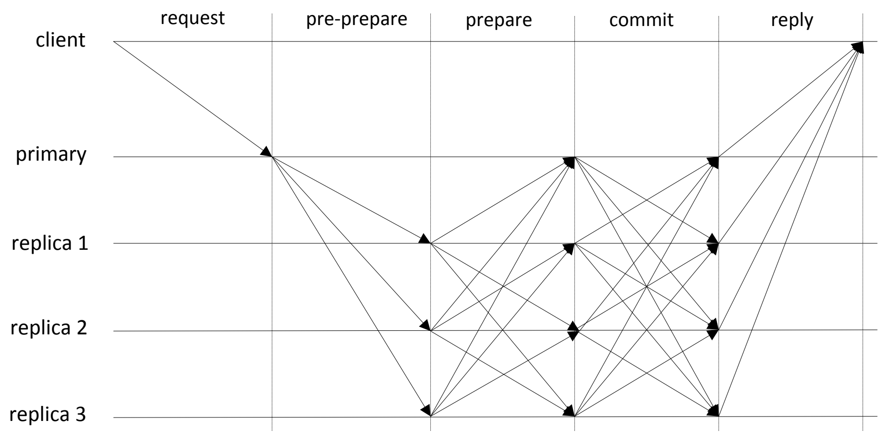


Figure 7.2: Communication pattern of the normal case of PBFT. When the primary receives a client request, it starts the three-round protocol to assign the sequence number for the received request. The order is defined at the end of "commit" round; at this point the request is executed and the reply is sent to the client. The client accepts a reply only once it received $b+1$ identical replies from $b+1$ servers.

is close to the protocol description. Therefore, it leads to implementations that matches the protocol specification on paper. It also leads to efficient implementations. As Distal currently does not provide authenticated channels, we extended the Distal's messaging layer to support message authentication based on SHA-1 HMACs.

In order to compare BFT-Mencius with the state-of-the-art protocols based on the same code base, we have also implemented the normal case of PBFT (see Figure 7.2) and Spinning using Distal. We have chosen PBFT as it is often considered to be the baseline for BFT algorithms, while Spinning is one of the most efficient existing algorithm designed to tolerate performance attacks. Furthermore, these protocols are interesting to compare with BFT-Mencius as they have different core mechanisms. PBFT has a single primary process that is responsible for proposing an order for client requests; the primary is changed only if enough processes suspect it. With Spinning the primary role rotates among processes, while with BFT-Mencius we have multiple primaries (as each process is primary in ATAB instances it owns).

We implemented the variant of Spinning denoted Spinning(LS) [VCBL09], where LS stands for Lock Step. With Spinning(LS) at most one agreement instance is initiated by the current leader. The communication pattern of the Spinning(LS) during the normal case is similar to the PBFT pattern shown on Figure 7.2 with the difference that once the current primary defines the order for some request[12] (at the end of COMMIT round), the next server becomes the primary for the next instance. There is also a variant in which the current primary can initiate multiple instances in parallel (without waiting that already started instances terminate); however,

[12]As will be explained below, in fact, the ordering is defined for a batch of requests.

[VCBL09] shows that this variant is less resilient to performance attacks (although it achieves slightly better performance than Spinning(LS) in the failure-free case). We implemented request batching[13] in all algorithms as it is known to be an essential optimization. PBFT and BFT-Mencius also allow multiple protocol instances to be run in parallel, i.e., a primary in PBFT (or any server in BFT-Mencius) can start a protocol instance (e.g., upon receiving a client request) before already started instances terminate. Running multiple instances in parallel might lead to better performance.

We do not experimentally compare BFT-Mencius with Prime because—as mentioned in Section 7.1—adding signatures to PBFT has already lead to a significant drop in performance even without adding the pre-agreement phase.

We make the following two points with the experimental evaluation. First, we show that the modular BFT-Mencius protocol has performance comparable to PBFT and Spinning in the failure-free case. Second, we show that BFT-Mencius is able to sustain good performance (similar to the failure-free case) even under performance attack.

7.7.1 Experimental setup and methodology

The experiments were run in the *Suno* cluster of the Grid5000 testbed. This cluster consists of nodes with dual 2.26GHz Intel Xeon E5520 processors, 32GB of memory, and 1Gb/s Ethernet connections. Nodes were running Linux, kernel version 2.6.32-5, and Oracle's Java 64-Bit Server VM version 1.6.0_26.

The workload was generated by nodes located in the same cluster as the servers. Clients send requests in a closed loop, waiting for the answer to the current request before sending the next one. The clients are evenly distributed over 15 nodes. We consider a setup with $n = 4$ servers that can tolerate one faulty server ($b = 1$), as we believe this to be a typical deployment setting. Each experiment was run for 3 minutes, with the first minute ignored in the calculation of the results. The service is a simple (stateless) echoing service that sends back the request as the response.

We use as metrics (i) the throughput in requests per second and (ii) the client response time. The *client response time* is the time from the point the client sends a request until it receives the corresponding reply from $b+1$ servers. Note that the client response time includes delays incurred by queuing of requests at servers. As mentioned in Section 7.1, guarantees for client latency assume a maximum client load.

[13]Instead of running protocol instance for a single request, multiple requests are packed in a batch, and then an instance is run for a batch. Request batching reduces the protocol overhead and leads to significant performance improvements.

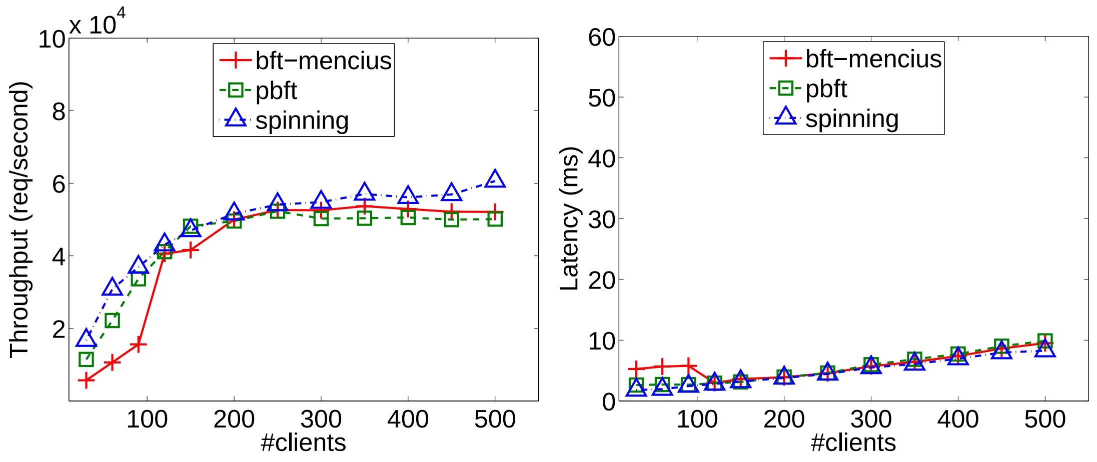


Figure 7.3: Throughput and latency of BFT-Mencius, PBFT and Spinning for different client load in the failure-free case for small requests (20B payload).

7.7.2 Failure-free executions

In the failure-free executions, we are interested in the maximum throughput and the corresponding response time under different load (number of clients). We run experiments for two sizes of requests: (i) 20B of payload and (ii) 8KB of payload. In case (i) at the point algorithms achieves maximum performance, the CPU is the bottleneck, while in case (ii) the network is the bottleneck. The number of concurrent protocol instances was set to 4 for PBFT, and to 1 for BFT-Mencius and Spinning. These values led to the best results.

As we can see on Figure 7.3, in case of small requests (20B payload), BFT-Mencius has slightly lower throughput and higher latency until 120 clients. This is due to the batching policy used, as the number of clients is not enough to fill the batch. We set the batch size to 1350B so that the size of the frame matches the natural limits of the underlying Ethernet network. The server timeout for proposing the batch was set to 3ms. The same batching policy is used with PBFT, while for Spinning we used the adaptive batching strategy used also in the original Spinning implementation [VCBL09][14]. With BFT-Mencius 120 clients or less is not enough to fill the batch, i.e., the latency is dominated by the batch timeout (3ms). With more than 120 clients, BFT-Mencius performs comparably to PBFT and Spinning.

In case of big requests (see Figure 7.4), we set the batch size to 60KB, and server timeout for proposing the batch to 3ms. These values led to the best results. BFT-Mencius performs very similarly to PBFT. On the other hand, Spinning legs behind. We believe that the reason is the fact that Spinning executes only one instance at a time, while with PBFT and BFT-Mencius there are multiple instances taking place in parallel. Executing multiple instances in parallel would be possible with Spinning, but this would require using a different batching strategy; we have not experimented with this option. Note that latency with big requests is higher than the

[14]Using adaptive batching with BFT-Mencius and PBFT led to similar results.

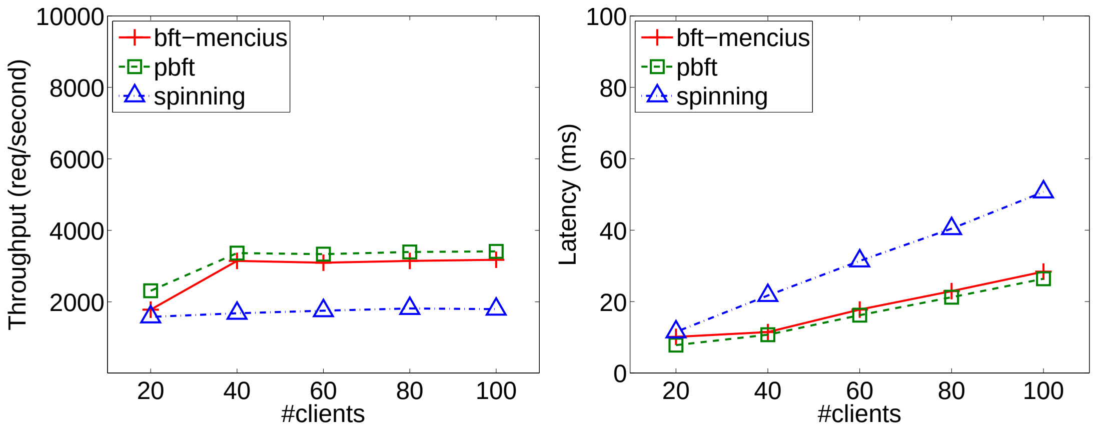


Figure 7.4: Throughput and latency of BFT-Mencius, PBFT and Spinning for different client load in the failure-free case for big requests (8KB payload).

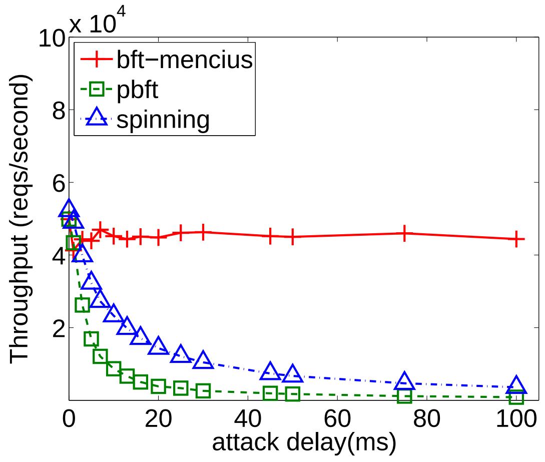
latency with small requests (compare Figure 7.3 with Figure 7.4). We believe that the difference comes from the cost of calculating MACs, which depends on the message size.

7.7.3 Executions with performance attacks by faulty servers

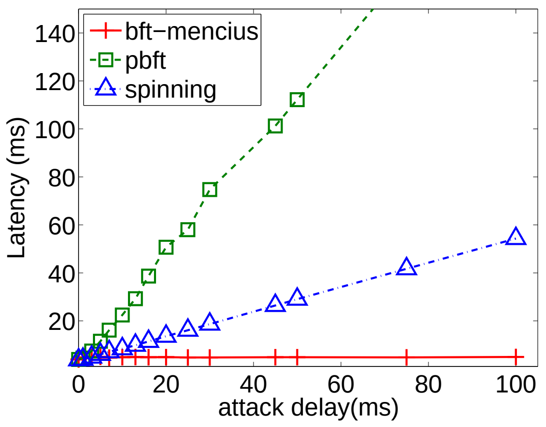
In order to measure the performance of BFT-Mencius under performance attacks, we run experiments where the faulty server delays starting ATAB instances it owns. Similarly, for Spinning and PBFT, the faulty server delays starting protocol instances (sending `PRE-PREPARE` message) whenever it is coordinator. Otherwise, the server normally participates in all algorithms. The number of clients for these experiments was set to 200 for experiments with small requests, and 60 for big requests, for all algorithms. We have chosen the number of clients based on the failure-free case where performance started to level off for all algorithms. For BFT-Mencius, we set K_{lat} (see Section 7.6.4) to 1. Thus we use $2 \cdot d_{ATAB}$ as timeout.

We have measured the performance of the three protocols by varying values for the attack delay. As we can observe on Figure 7.5 (for small requests) and Figure 7.6 (for big requests), the performance of PBFT and Spinning, compared to the failure-free case, can be significantly degraded in the presence of performance attack. We can also observe that always rotating the primary makes Spinning more robust to performance attacks than PBFT: its average latency is significantly better, albeit still dependent on the attack delay.

On the other hand, once the attack delay is above the suspicion timeout, the blacklisting mechanism of BFT-Mencius allows us to skip instances of blacklisted servers. Setting the suspicion timeout as explained in Section 7.6.5, faulty servers are blacklisted already with an attack-delay of 3ms for small requests and above 5ms with big requests. This allows BFT-Mencius to achieve latency similar to the failure-free case, and throughput close to the peak throughput achieved in the failure-free case.

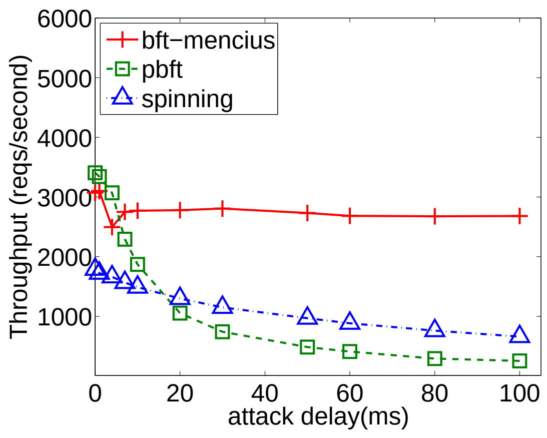


(a) Throughput for different client load

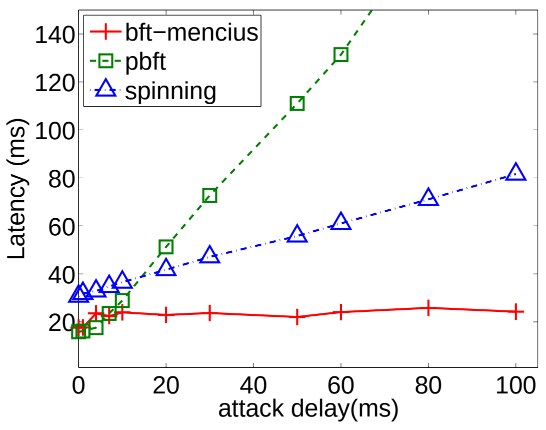


(b) Latency for different client load

Figure 7.5: Throughput and latency of BFT-Mencius, PBFT and Spinning under performance attacks with small requests (20B of payload).



(a) Throughput for different client load



(b) Latency for different client load

Figure 7.6: Throughput and latency of BFT-Mencius, PBFT and Spinning under performance attacks with big requests (8KB of payload).

7.8 Conclusion

We have proposed a new state machine replication protocol for the partially synchronous system model with Byzantine faults. The algorithm, called *BFT-Mencius*, is a modular, signature-free SMR protocol that ensures bounded-delay, i.e., eventual bounded latency during periods of synchrony, even in the presence of Byzantine processes. BFT-Mencius is based on a new communication primitive, *Abortable Timely Announced Broadcast* (ATAB). In cluster settings, BFT-Mencius performs comparably to state-of-the-art algorithms such as PBFT and Spinning in fault-free configurations. Contrary to these protocols, BFT-Mencius is able to maintain the same performance under performance attacks.

Acknowledgement Experiments presented in this chapter were carried out using the Grid'5000 experimental testbed, being developed under the INRIA ALADDIN development action with support from CNRS, RENATER and several Universities as well as other funding bodies (see https://www.grid5000.fr).

8 Concluding Remarks

The thesis has proposed several abstractions that allows us to express consensus algorithms that tolerate arbitrary faults in a concise and modular way. A number of existing and several novel consensus algorithms are expressed using these abstractions. The thesis also clarifies relations among total-ordered broadcast and consensus in the presence of Byzantine faults. Finally, the thesis proposes a modular BFT state machine replication algorithm that performs well even under performance attacks by Byzantine processes. We summarize each of these contributions and discuss directions for future research.

Weak Interactive Consistency. Consensus protocols that assume *Byzantine* faults (without authentication) are harder to develop and prove correct [ST87] than algorithms that consider *authenticated Byzantine* faults, even when they are based on the same idea. We have introduced an abstraction called *weak interactive consistency* (or *WIC*), that allows us to design a consensus algorithm that can be instantiated into an algorithm for authenticated Byzantine faults (signed messages) and an algorithm for Byzantine faults, i.e., WIC unifies Byzantine consensus algorithms with and without signatures. This has been illustrated on two seminal Byzantine consensus algorithm, namely on the FaB Paxos algorithm (signatures) and on the PBFT algorithm (no signatures). In both cases this leads to a very concise algorithm. We presented in thesis two leader-based implementations of WIC, one that requires signatures and needs two rounds of communication and the other that does not use signatures but needs three rounds of communication. The message complexity of the two implementations is also different, namely $O(n)$ for the former and $O(n^2)$ for the latter. Furthermore, there exists a leader-free implementation of WIC [BS10] that needs $b+1$ rounds (where b is a maximum number of faulty processes) and has $O(n^2)$ message complexity. Although some work analytically compare different WIC implementations in the round-based model [BHS12], it would be worth doing an experimental study of different WIC implementations in various settings: different network setup (LAN and WAN), different number of processes and different message sizes.

Tolerating Permanent and Transient Value Faults. Transmission faults allow us to reason about permanent and transient faults in a uniform way. However, all existing solutions to consensus in this model assume either the synchronous system, or require strong conditions for termination that exclude the case where all messages of a process can be corrupted. Using the WIC abstraction, we propose a consensus algorithm for the transmission fault model that does not have these limitations. The algorithm considers a system parameterized with α and f. In every round *each process* can receive up to α corrupted messages; eventually rounds are synchronous and the messages sent by at most f processes are corrupted. Before these synchronous rounds, any number of benign faults is tolerated. The thesis proposes a leader-based simulation of weak interactive consistency in this system model, which is compatible with permanent and transient faults. Depending on the nature and number of permanent and transient transmission faults, we obtain different conditions on n (number of processes) for solving consensus. Our work opens several research directions:

- The consensus algorithm presented requires for termination that eventually messages of at most f processes are corrupted. Is it possible solving consensus if we further weaken this condition?
- Is there a way to adapt the leader-free implementation of WIC [BS10] for this system model?
- Can we solve other problems, such as total-order broadcast or state machine replication in this model?

Generic Consensus Algorithm for Benign and Byzantine Faults. Numerous consensus algorithms have been published, with different features and for different fault models. We have presented a generic consensus algorithm that highlights the core mechanisms of various consensus algorithms for benign and Byzantine faults. The generic algorithm has four parameters: T_D, *FLAG*, *Validator* and *FLV*. Instantiation of these parameters led us to distinguish three classes of consensus algorithms into which well-known algorithms fit. It allowed us also to identify the new MQB algorithm that requires $n > 4b$, in-between the requirement $n > 5b$ of FaB Paxos and $n > 3b$ of PBFT (b is the maximum number of Byzantine processes). We believe that a generic algorithm allows people familiar with a particular consensus algorithm to more easily expend their knowledge to other consensus algorithms. As the generic algorithm focuses (only) on deterministic algorithms for the partially synchronous system model, an interesting research direction could be considering also randomized consensus algorithms and algorithms designed for synchronous and semi-synchronous system models.

On the Reduction of Total-Order Broadcast to Consensus with Byzantine Faults. The relation of consensus and total order broadcast, including the reduction of total order broadcast to consensus, is well understood in the case of crash faults [CT96a]. We have studied the relation between total order broadcast and different variants of consensus in systems with Byzantine faults. We have shown that consensus with *weak unanimity* is not sufficient to solve total order broadcast, while consensus with *strong validity* is harder than total order broadcast. Furthermore, we have shown that total order broadcast is equivalent to consensus with *strong unanimity*, consensus with *abortable validity*, and consensus with *range validity*. We have also proposed a reduction of total order broadcast to range validity consensus with constant time complexity with respect to consensus. To the best of our knowledge, this is the first total order broadcast reductions to consensus with constant time complexity with respect to consensus in the Byzantine fault model.

The total-order broadcast algorithm presented separates message dissemination (done using reliable unique broadcast) from defining the order of messages (done using instances of range validity consensus). Furthermore, range validity consensus instances operate only on integers that corresponds to message ids. The same architecture is used by the S-Paxos SMR algorithm [BMSS12] for benign faults. S-Paxos achieves between 2 and 3 times better throughput than Paxos, and the difference further increases for higher number of replicas. It would be interesting to experimentally evaluate whether the total-order broadcast algorithm presented could have similar performance gains for the Byzantine model when compared to the classical approaches (e.g., PBFT or BFT-Mencius), where request dissemination and ordering are not separated.

Bounded Delay in Byzantine Tolerant State Machine Replication. Recent studies have shown that most BFT-SMR systems do not actually perform well under performance attacks by Byzantine processes. This led to the definition of a new performance criterion, called *bounded-delay*, which requires that the latency of updates initiated by correct processes is eventually upper-bounded, even under performance attacks by Byzantine processes. We have proposed a new state machine replication protocol for partially synchronous systems with Byzantine faults. The algorithm, called *BFT-Mencius*, is a modular, signature-free SMR protocol that ensures bounded-delay, i.e., eventual bounded latency during periods of synchrony, even under performance attacks by Byzantine processes. BFT-Mencius is based on a new communication primitive, *Abortable Timely Announced Broadcast* (ATAB). In cluster settings, BFT-Mencius performs comparably to state-of-the-art algorithms such as PBFT and Spinning in the fault-free case. Contrary to these protocols, BFT-Mencius is able to maintain the same performance under performance attacks.

We have evaluated BFT-Mencius in cluster settings, as this is the most common context where state machine replication is used. However, BFT-Mencius also works in a WAN, as it does not depend on any cluster-specific functionality. Many of the techniques that allow BFT-Mencius to perform well in a cluster should work equally well in a WAN; nevertheless it is worth investigating this question. Furthermore, it would be possible to consider different options for some mechanisms used by BFT-Mencius that might be more suitable for a WAN: different mechanisms for suspecting, different client load balancing schema or different request authentication schema. Finally, it would be interesting considering BFT-Mencius with different ATAB implementations, for example one based on FaB Paxos.

A CL-ATAB

In this chapter we present the CL-ATAB algorithm that solves ATAB in the partially synchronous system model with Byzantine faults. The algorithm requires $n \geq 3b+1$ processes to tolerate at most b Byzantine faults. The code of the algorithm is given as Algorithms A.1 and A.2. The *upon rules* of Algorithm A.1 and Algorithm A.2 are executed atomically. Since we allow message loss before GST, we use the p-broadcast primitive which denotes periodic send to all processes with Δ as an interval between two broadcast events.

As already said in Section 7.4.1, the algorithm proceeds in views, such that in every view there is a single process that is the coordinator (of the view). The assignment scheme of views to coordinators is known to all processes and is given as a function *coord*(v) returning the coordinator for view v. The sender s is the coordinator of the initial view ($view = 1$).

A.1 Normal case

We now explain the protocol in the initial view $v = 1$ (Algorithm A.1); the procedure for changing view is discussed for Section A.2.

Once a process p wants to broadcast a message m using ATAB, it executes *atab-cast*(m) (line 10 of Algorithm A.1). Upon *atab-cast*(m), a process sends message $\langle \mathsf{INIT}, 1, m \rangle$ to all processes (line 11 of Algorithm A.1). Once a process receives $\langle \mathsf{INIT}, 1, m \rangle$ from the sender for the first time, and it is in the view $v = 1$, it sends $\langle \mathsf{ECHO}, 1, m \rangle$ to all processes (line 14).

Once a process p receives $\langle \mathsf{ECHO}, 1, m \rangle$ from $\lceil (n+b+1)/2 \rceil$ processes, and it is in the view $v = 1$, it updates $vote_p$ and ts_p (lines 17-18) and sends $\langle \mathsf{COMMIT}, 1, m \rangle$ to all processes (line 19).

A process receiving $\langle \mathsf{COMMIT}, 1, m \rangle$ from $\lceil (n+b+1)/2 \rceil$ processes, while being in the view $v = 1$, it executes *atab-deliver*(m) (line 23). A process can also *atab-deliver*(m) by receiving $b+1$ messages $\langle \mathsf{DEC}, m \rangle$ (line 20).

The algorithm also needs to execute *atab-announce* such that the *ATAB-Announcement* property of ATAB is fulfilled. A correct process always executes *atab-announce* before it executes *atab-deliver* (line 21). Furthermore, the algorithm needs to ensure that once a correct process executes *atab-deliver*, all correct processes will eventually execute *atab-announce*. This is ensured by the second part of line 24. Once a correct process executes *atab-deliver* (line 23), then at least one correct process received $\langle \mathsf{COMMIT}, v, m\rangle$ from $\lceil (n+b+1)/2\rceil$ processes. Since messages are sent using p-broadcast, and $n \geq 3b+1$, at least $b+1$ correct processes sent $\langle \mathsf{COMMIT}, v, m\rangle$ to all. Therefore, all correct processes eventually also receive this messages and execute *atab-announce* at line 24.

Algorithm A.1 CL-ATAB (part A)

```
1: Initialization:
2:    $vote_p := noop$
3:    $ts_p := 0$
4:    $history_p := \emptyset$
5:    $view_p := 1$
6:    $state_p \in \{init = 1, echoed = 2, changingView = 3\}$, initially $init$
7:    $decision_p = null$
8:    $timeout1 := 3\Delta$
9:    $timeout2 := 6\Delta$

10: upon atab-cast($m$) do
11:    p-broadcast $\langle \mathsf{INIT}, 1, m\rangle$

12: upon receiving $\langle \mathsf{INIT}, view_p, m\rangle$ from coord(1) while $state_p = init$ do
13:    $history_p \leftarrow \{(m, view_p)\}$
14:    p-broadcast $\langle \mathsf{ECHO}, view_p, m\rangle$
15:    $state_p \leftarrow echoed$

16: upon receiving $\langle \mathsf{ECHO}, view_p, m\rangle$ from $\lceil (n+b+1)/2\rceil$ processes while $state_p \leq echoed$ do
17:    $vote_p \leftarrow m$
18:    $ts_p \leftarrow view_p$
19:    p-broadcast $\langle \mathsf{COMMIT}, view_p, m\rangle$

20: upon receiving ($\langle \mathsf{COMMIT}, view, m\rangle$ from $\lceil (n+b+1)/2\rceil$ processes or $\langle \mathsf{DEC}, m\rangle$ from $b+1$ processes) while
    $decision_p \neq null$ do
21:    $Announce()$
22:    $decision_p \leftarrow m$
23:    atab-deliver($m$)

24: upon receiving $\langle \mathsf{INIT}, 1, v\rangle$ from coord(1) or $\langle \mathsf{COMMIT}, v, m\rangle$ from $b+1$ processes do
25:    $Announce()$

26: upon receiving any message $\neq \langle \mathsf{DEC}, -\rangle$ from $q$ while $decision_p \neq null$ do
27:    send $\langle \mathsf{DEC}, decision_p\rangle$ to $q$

28: Procedure Announce :
29:    atab-announce()
30:    if $view_p = 1$ then
31:       after $timeout1$ execute $OnTimeout(1)$

32: Procedure $OnTimeout(v)$ :
33:    if $decision_p = \perp$ then
34:       $ProgressToView(v+1)$
```

A.2 Changing view

In this section we explain the part of the protocol that processes execute when changing view. The protocol is given as Algorithm A.2. As mentioned in Section 7.4.1 changing the view is handled in a similar way as in PBFT. The processes exchange their state by broadcasting $\langle \mathsf{VC}, v, vote, ts, history\rangle$ upon entering view v (line 59). When a process p receives $\langle \mathsf{VC}, view_p, -, -, -\rangle$ from a process q for the first time, it acknowledges reception by sending $\langle \mathsf{VC\text{-}ACK}, -, q\rangle$ to all (lines 41-44). The coordinator of the new view considers only $\langle \mathsf{VC}, view_p, -, -, -\rangle$ messages for which it is received acknowledgments by at least $2b+1$ processes. This ensures that at least $b+1$ correct processes received the same $\langle \mathsf{VC}, view_p, -, -, -\rangle$ message from some process q (that could be Byzantine). The array of messages that satisfies this condition is passed to the FLV function (line 83) which is responsible for selecting a value that the new coordinator will propose, ensuring that the *ATAB-Agreement* and *ATAB-Integrity* properties are not violated.

The FLV function ensures that in case some correct process has executed *atab-deliver*(m) in the previous views, it can only select m or $null$. The value $null$ is returned to indicate that not enough information was provided to the FLV function. As processes receive more information, i.e., more view-change messages, they will retry to obtain a previous decision value by calling FLV again. If there was a decision, FLV is guaranteed to eventually return the value decided. In case no correct process decided in the previous views, FLV returns $\perp$.

Once the FLV function returns a non $null$ value, the coordinator of the new view sends $\langle \mathsf{VC\text{-}INIT}, -, -, -\rangle$ to all processes (line 69). The other processes verify if $\langle \mathsf{VC\text{-}INIT}, -, -, -\rangle$ sent by the new coordinator is valid using the $isValid$ function (line 75). This mechanism is needed because the new coordinator can be a faulty process. In case $isValid$ function returns `true`, the process sends $\langle \mathsf{ECHO}, -, -\rangle$ to all (line 70) and the protocol continues as in view $v = 1$.

A.3 Proof of correctness

Lemma A.1. *Let v_m be the highest view entered by some correct process up to time t. If $n \geq 3b+1$, then at at time t at least $b+1$ correct processes are either in view v or in view $v-1$.*

Proof. Let denote with p a first correct process that started view v_m. There are two cases to consider: (i) $v_m = 2$ or (ii) $v_m > 2$. In case (i), the lemma trivially follows since all correct processes are in at least view 1. In case (ii), since v_m is the highest view started by some correct process, the process p entered view v_m upon timeout expiration. Since $v_m > 2$, the timeout is set at line 40. By line 39, p received $\langle \mathsf{VC}, v_m - 1, -, -, -\rangle$ from $\lceil (n+b+1)/2 \rceil$ processes. Since $n \geq 3b+1$, $\lceil (n+b+1)/2 \rceil > b$. Therefore at least $b+1$ correct processes sent $\langle \mathsf{VC}, v_m - 1, -, -, -\rangle$ message, i.e., at least $b+1$ correct processes are at least in the view $v_m - 1$ at time t. □

Lemma A.2. *Let $t > GST$ be such that at time t $b+1$ correct processes have started view $v \geq 2$, and no correct process has started view $v' > v$ and $owner(v)$ is a correct process. If $n \geq 3b+1$ then all correct processes decide in view v the latest at time $t + 6\Delta$.*

Proof. Let denote with C_f the set of $b+1$ correct processes that start view v, and let assume that p is the last among correct processes from C_f that starts view v. Furthermore, let assume that it starts view v at time $t > GST$. Then the latest at time $t + \Delta$ all correct processes receive $\langle \mathsf{VC}, v, -, -, - \rangle$ message from processes C_f. Because of rule at line 37, all correct processes then start view v at time $t + \Delta$, and send $\langle \mathsf{VC}, v, -, -, - \rangle$ message. Note that process start timeout for view v once it receives $\langle \mathsf{VC}, v, -, -, - \rangle$ messages from $\lceil (n + b + 1)/2 \rceil$ processes (line 39). Therefore, the earliest time when timeout for view v is started at some correct process is t.

At time $t + 2\Delta$, all correct processes are in view v and receive $\langle \mathsf{VC}, v, -, -, - \rangle$ message from all correct processes. Upon receipt of $\langle \mathsf{VC}, v, -, -, - \rangle$ message, $\langle \mathsf{VC\text{-}ACK}, -, - \rangle$ message is sent (line 44). Therefore, at time $t + 2\Delta$ all correct processes acknowledge receipt of $\langle \mathsf{VC}, v, -, -, - \rangle$ messages from correct processes by sending the corresponding $\langle \mathsf{VC\text{-}ACK}, -, - \rangle$ message. This mean that $owner(v)$ send $\langle \mathsf{VC\text{-}INIT}, v, m, - \rangle$ message before $t + 3\Delta$ (line 69) and all correct processes receive it and have the condition at line 71 of Algorithm A.2 evaluates to `true` at latest at $t + 4\Delta$ (at processes that are not $owner(v)$). Then they send $\langle \mathsf{ECHO}, v, m \rangle$ message (see line 73 of Algorithm A.2) that is received before $t + 5\Delta$. Since $n \geq 3b + 1$, $n - b \geq \lceil (n + b + 1)/2 \rceil$, so once a correct process receives $\langle \mathsf{ECHO}, v, m \rangle$ message from all correct processes, it sends $\langle \mathsf{COMMIT}, v, m \rangle$ message (line 19). All correct processes then decide before $t + 6\Delta$. Since the value of $timeout$ is 6Δ, no correct process will leave view v before $t + 6\Delta$ and therefore all correct processes will decide in view v. □

Lemma A.3. *If up to time T, all correct processes have executed either atab-announce or atab-abort, then all correct processes are in view $v \geq 2$ the latest at time $T + 3\Delta$.*

Proof. The correct process that executed *atab-abort* move in view 2 the latest at time T (line 36). After correct process executes *atab-announce*, the timer is started (line 31). Once timeout expires, the process move in view 2 (if it is not already in view higher than 1). Therefore, all correct processes are in view 2 the latest at time $T + 3\Delta$. □

Lemma A.4. *If at least $b + 1$ correct processes are in view $v \geq 2$ at time $t > GST$, then $\exists t' \geq t$, $t' \leq t + 16\Delta$, such that all correct processes are in view $v' \geq v$ at time t'.*

Proof. Let denote with p the correct process that is in the highest view v_m at time t. We have two cases to consider: (i) there are at most $b - 1$ other correct processes p in view v at time t or (ii) there are more than b other correct processes in view v at time t.

In case (i), at time $t + \Delta$ (due to retransmission that takes place every Δ time) all correct processes resend their $\langle \mathsf{VC}, -, -, -, - \rangle$ messages, so all correct processes receive those messages before $t + 2\Delta$. Because of rule at line 37, all correct processes enter at least view $v - 1$ at time $t + 2\Delta$ and send their $\langle \mathsf{VC}, -, -, -, - \rangle$ messages for view $v - 1$ at that point. They start timer for view $v - 1$ at time $t + 3\Delta$ (line 39). Therefore, they will enter view v the latest at time $t + 9\Delta$ since timeout value is 6Δ. Note however, that at time $t + \epsilon$ a correct process from view $v - 1$ can move to view v. Therefore, it can happen that some correct process receives $\langle \mathsf{VC}, v, -, -, - \rangle$ message from $\lceil (n + b + 1)/2 \rceil$ processes at time $t + \epsilon$ and start timer for view v. Therefore, at time $t+6\Delta$ he will leave view v and start view $v+1$. Therefore, we need to calculate the point in time when other correct processes will start view $v+1$. Since all correct processes start view v the latest at time $t+9\Delta$, they will receive $\langle \mathsf{VC}, v, -, -, -\rangle$ message from $\lceil (n+b+1)/2\rceil$ processes before time $t+10\Delta$. Therefore, they will start timer for view v the latest at $t+10\Delta$, and therefore start view $v+1$ the latest at time $t+16\Delta$.

In case (ii), at time $t+\Delta$ (due to retransmission that takes place every Δ time) $b+1$ correct processes resend their $\langle \mathsf{VC}, v, -, -, -\rangle$ messages, so all correct processes will receive them before time $t+2\Delta$ and enter view v. They start timer for view v the latest at time $t+3\Delta$. Since at any time after t there can be correct process that leaves view v and start view $v+1$ we need to calculate what is the latest point when other correct processes will enter view $v+1$. Since they start timer for view v the latest at time $t+3\Delta$, they will start view $v+1$ the latest at time $t+9\Delta$.

Therefore, all correct processes will start the same view v', such that no correct process is in the higher view, the latest at time $t+16\Delta$. □

Lemma A.5. *If at least $b+1$ correct processes are in view $v \geq 2$ at time $t > GST$, then all correct processes decide the latest at time $t+16\Delta+b\cdot 7\Delta+6\Delta$.*

Proof. By Lemma A.4, all correct processes start the same view v, such that no process is in the higher view, the latest at time $t+16\Delta$. If $owner(v)$ is a correct process, then by Lemma A.2 all correct processes decide the latest at time $t+16\Delta+6\Delta$. In case $owner(v)$ is a faulty process, then all correct processes will start timeout for view v the latest at time $t+17\Delta$. Therefore, they will start the view $v+1$ the latest at $t+17\Delta+6\Delta$. Since we apply rotating coordinator strategy for $owner(v)$ function, we will have correct process being owner of the view $v+b$. The correct processes start view $v+b$ the latest at time $t+16\Delta+b\cdot 7\Delta$. According to Lemma A.2, all correct processes decide in the view $v+b$ the latest at time $t+16\Delta+b\cdot 7\Delta+6\Delta$. □

Lemma A.6. *For all $b \geq 0$, any two sets of size $\lceil (n+b+1)/2\rceil$ have at least one correct process in common.*

Proof. We have $2\lceil (n+b+1)/2\rceil \geq n+b+1$. This means that the intersection of two sets of size $\lceil (n+b+1)/2\rceil$ contains at least $b+1$ processes, *i.e.*, at least one correct process. The result follows directly from this. □

Lemma A.7. *If $m \neq null$ is the only value that can be returned by FLV function at correct processes in view v, then in view v a correct process p can set $vote_p$ only to m.*

Proof. If m is the only not-$null$ value that can be that can returned by FLV function at correct processes in view v, then if a correct process sends $\langle \mathsf{ECHO}, v, value\rangle$, $value = m$. Because there are at most b Byzantine processes, and $b < \lceil (n+b+1)/2\rceil$, for all correct processes holds that if exists some value that satisfies the condition at line 16, then it must be m. So if a correct process p set $vote_p$ in view v, it set it to m. □

Lemma A.8. *If some correct process q executes atab-deliver(m) in view v_0, then in all views $v > v_0$, FLV returns either m or $null$ at all correct processes.*

Proof. We prove the result by induction on v.

Base step $v = v_0 + 1$: Assume by contradiction that p is some correct process where FLV function returns $m' \neq m \wedge m' \neq null$ in view $v_0 + 1$. This implies that either (i) line 87 or (ii) line 89 was executed by p in phase $v_0 + 1$.

For (ii), the condition of line 88 has to be true. If the condition of line 88 is true, this implies that either (i) $|correctVotes_p| > 1$ or (ii) there are at least $\lceil (n+b+1)/2 \rceil$ messages with $ts = 0$ in the array V. Since q has decided in view v_0, by LemmaX at least one correct process received at least $\lceil (n+b+1)/2 \rceil$ messages $\langle \mathsf{COMMIT}, v_0, m \rangle$ at line 20. All correct processes c who sent a message $\langle \mathsf{COMMIT}, v_0, m \rangle$ have set $vote_c = v$ and $ts_c = v_0$ in view v_0. Let us denote this set of correct processes with Q_c. By Lemma A.6 the intersection of two sets of size $\lceil (n+b+1)/2 \rceil$ contains at least one correct process. Therefore, in the $\lceil (n+b+1)/2 \rceil$ messages received there is at least one message sent by process from Q_c, *i.e.*, the second part of the condition at line 88 cannot be true. So the case (i) was executed by p.

For (i), the condition at line 86 have to be true, i.e., the $|correctVotes_p| > 0$, there exists a $(m', ts', history')$ such that m' is in $correctVotes_p$ and because of line 85 $(m', ts', history')$ in $possibleVotes_p$. We now show that if $(m', ts', history') \in possibleVotes_p$, then m' does not satisfy the condition at line 85 to be added to the $correctVotes_p$. This establishes the contradiction.

Since the parameter passed to the FLV function consists of $\langle \mathsf{VC}, -, -, - \rangle$ messages, by Lemma A.6, the array V contains at least one message $\langle \mathsf{VC}, v_0 + 1, m, v_0, - \rangle$ sent by a process in Q_c. So the $(m', ts', history')$ can only be added to the set $possibleVotes_p$ if $ts' > v_0$.

In order to have m' in the set $correctVotes_p$ it is necessary to have at least $b+1$ messages $\langle \mathsf{VC}, v_0+1, -, -, history \rangle$ in V such that $\exists (m', ts') \in history$, i.e., that there is a correct process c_1 that sends such a message. A contradiction with the assumption that c_1 is a correct process.

Induction step from ϕ to $\phi + 1$: Lemma A.7 and the arguments similar to the base step can be used to prove the induction step. □

Lemma A.9. *If $n \geq 3b+1$, Algorithm A.1 ensures* ATAB-Agreement.

Proof. Let view v_0 be the first view in which some correct process p executes *atab-deliver*(m). By Lemma A.14, there exists a correct process c that received $\langle \mathsf{COMMIT}, v, m \rangle$ message from $\lceil (n+b+1)/2 \rceil$ processes in some view $v \leq v_0$. Since $n \geq 3b+1$, $\lceil (n+b+1)/2 \rceil > b$, so at least one correct process c_1 sent $\langle \mathsf{COMMIT}, v, m \rangle$ message in the view v. Therefore c_1 received at least

$\lceil (n+b+1)/2 \rceil$ messages $\langle \mathsf{ECHO}, v, m\rangle$ (*). We prove now that if some correct process q executes *atab-deliver*(m') in some view $v \geq v_0$, then $m = m'$. In case $v = v_0$, by Lemma A.14, there exists a correct process c' that that received $\langle \mathsf{COMMIT}, v, m'\rangle$ message from $\lceil (n+b+1)/2 \rceil$ processes. Since $n \geq 3b+1$, $\lceil (n+b+1)/2 \rceil > b$, so at least one correct process sent $\langle \mathsf{COMMIT}, v, m'\rangle$ message in the view v. Therefore it received least $\lceil (n+b+1)/2 \rceil$ messages $\langle \mathsf{ECHO}, v, m'\rangle$. By Lemma A.6, any two sets of messages of size $\lceil (n+b+1)/2 \rceil$, contains at least one correct process in intersection. Therefore, there exists a correct process c_1 that sends $\langle \mathsf{ECHO}, v, m\rangle$ and $\langle \mathsf{ECHO}, v, m'\rangle$ message. A contradiction with the assumption that c_1 is a correct process and the rule at line 12 and lines 73-74.

In case $v > v_0$, by Lemma A.8 and Lemma A.7, all correct processes can only set $vote$ to m in views bigger than v_0. Since $n \geq 3b+1$, $b < \lceil (n+b+1)/2 \rceil$, so the first part of the condition at line 20 can be `true` only for m. This is in contradiction with Lemma A.14, so correct process q cannot *atab-deliver* message different than m. □

Lemma A.10. *If s is a correct process and s executes atab-cast(m), then later the FLV function at all correct processes can return only b such that $b \in \{m, \perp, null\}$.*

Proof. Assume by contradiction that view v is the first view where *FLV* function at a correct process q returns m' such that $m' \notin \{m, \perp, null\}$. This implies that line 87 is executed by q. By assumption we have that for all messages $\langle \mathsf{VC}, v, -, -, history\rangle$ sent by correct processes, $history = \{(v,t) : v = m \lor v = \perp\}$ or $history = \emptyset$.

In order to have the condition at line 86 to be true, i.e., the $|correctVotes_q| > 0$, there exists m' in $correctVotes_q$.

In order to have m' in the set $correctVotes_p$ it is necessary to have at least $b+1$ messages $\langle \mathsf{VC}, v, -, -, history\rangle$ in V such that $\exists (m', ts') \in history$, i.e., that there is a correct process c_1 that sends such a message, i.e., *FLV* returns m' at c_1 in the view $v' < v$. A contradiction. □

Lemma A.11. *If $n \geq 3b+1$, Algorithm A.1 ensures* ATAB-Integrity.

Proof. A correct process p executes *atab-deliver* only once because the rule in which *atab-deliver* is executed is triggered only if $decision_p = null$ (line 20). After process p executes line *atab-deliver* for the first time, $decision_p$ is set to some value $m \neq null$.

Now assume that s is a correct process and executes *atab-cast*(m) and there is a correct process q that executes *atab-deliver*(m') such that $m' \notin \{m, \perp\}$. By Lemma A.14, there is a correct process c that received $\langle \mathsf{COMMIT}, v', m'\rangle$ messages from $\lceil (n+b+1)/2 \rceil$ in some view v' when $view_c = v'$. Since $n \geq 3b+1$, $\lceil (n+b+1)/2 \rceil > b$, there is at least one correct process c_1 that sends $\langle \mathsf{COMMIT}, v', m'\rangle$ message in view v'. Therefore, c_1 received $\lceil (n+b+1)/2 \rceil > b$ messages $\langle \mathsf{ECHO}, v', m'\rangle$. Since $n \geq 3b+1$, $\lceil (n+b+1)/2 \rceil > b$, i.e., there is at least single correct process c_2 that sent $\langle \mathsf{ECHO}, v', m'\rangle$ in the view v'. There are two cases to consider: (i) $v' = 1$ and (ii) $v' > 1$. In case (i), the process c_2 received $\langle \mathsf{INIT}, v', m'\rangle$ message from s. A contradiction with the assumption that s is a correct process that executed *atab-cast*(m). In case (ii), the process c_2 received $\langle\text{VC-INIT}, v', m', -\rangle$ from some process that is $owner(v')$.

Since process c_2 sent $\langle\text{ECHO}, v', m'\rangle$ after receiving $\langle\text{VC-INIT}, v', m', -\rangle$, this implies that the function $isValid$ at process c_2 returns `true` for message $\langle\text{VC-INIT}, v', m', -\rangle$ (line 73). By line 79, the function FLV returns m' at process c_2 in view v'. A contradiction with Lemma A.10. □

Lemma A.12. *If $n \geq 3b+1$, Algorithm A.1 ensures* ATAB-Validity *with $d_1 = 3\Delta$.*

Proof. If s executes *atab-cast*(m) at time T, it sends $\langle\text{INIT}, 1, m\rangle$ message to all (line 11), and all correct processes receive it before time $T+\Delta$. Since no correct process calls *atab-abort* before $T+3\Delta$, $state$ variable at all correct processes is $init$ and $view = 1$, so the condition at line 12 evaluates to `true`, and all correct processes send $\langle\text{ECHO}, 1, m\rangle$ to all (line 14 and set $state$ to *echoed* (line 15). Before time $T+2\Delta$, all correct processes receive $\langle\text{ECHO}, 1, m\rangle$ message from all correct processes. Since $n \geq 3b+1$ and no correct process *atab-abort* before $T+3\Delta$, all correct processes receive $\langle\text{COMMIT}, 1, m\rangle$ message from $\lceil(n+b+1)/2\rceil$ processes, so the condition at line 16 evaluates to `true`, and every correct process sends $\langle\text{COMMIT}, 1, m\rangle$ message to all (line 19) the latest at time $T+2\Delta$. If a correct process has not decided up to time $T+3\Delta$, it will decide upon receiving $\langle\text{COMMIT}, 1, m\rangle$ message from all correct processes since the condition at line 20 evaluates to `true`. Therefore, if no correct process *atab-abort* before time $T+3\Delta$, all correct processes will *atab-deliver*(m) the latest at time $T+3\Delta$. □

Lemma A.13. *Algorithm A.1 ensures* ATAB-Termination.

Proof. If all correct processes execute *atab-announce* or *atab-abort* up to time T, by Lemma A.3 all correct processes enter view 2 the latest at time $T+3\Delta$. By Lemma A.5 all correct processes will decide the latest at time $\max\{T+3\Delta, GST\}+16\Delta+b\cdot 7\Delta+6\Delta$. □

Lemma A.14. *If a correct process p executes atab-deliver(m) at time T, then at least one correct process q received $\langle\text{COMMIT}, v, m\rangle$ from $\lceil(n+b+1)/2\rceil$ processes in view v at time $t \leq T$.*

Proof. Assume by contradiction that a correct process p executes *atab-deliver*(m) at time T, and that no correct process received $\langle\text{COMMIT}, v, m\rangle$ message from $\lceil(n+b+1)/2\rceil$ processes at time $t \leq T$. Furthermore, assume that a process p is a first correct process that executed *atab-deliver*(m). This implies that a correct process that executes *atab-deliver*(m) execute it after time T.

Since a correct process p executes *atab-deliver*(m) at time T, this mean that the condition at line 20 evaluates to `true`. There are two cases to consider: (i) p received $\langle\text{COMMIT}, view_p, m\rangle$ message from $\lceil(n+b+1)/2\rceil$ processes, or (ii) p received $\langle\text{DEC}, m\rangle$ message from $b+1$ processes.

In case (ii), p received $\langle \mathsf{DEC}, m\rangle$ message from $b+1$ processes. This mean that there is at least one correct process c that sent $\langle \mathsf{DEC}, m\rangle$ message before time T. A contradiction with the assumption that p is the first correct process that executed *atab-deliver*(m). Therefore by (i), p received $\langle \mathsf{COMMIT}, view_p, m\rangle$ message from $\lceil (n+b+1)/2\rceil$ processes. A contradiction. □

Lemma A.15. *If* $n \geq 3b+1$*, Algorithm A.1 ensures* ATAB-Announcement *with* $d_2 = 2\Delta$.

Proof. The first part of *ATAB-Announcement* property is trivially ensured by lines 21 and 23. We now prove the second part, i.e., (i) if a correct process p executes *atab-deliver*(m) at time T or (ii) a correct process s executes *atab-cast*(m') at time T, then every correct process executes *atab-announce* before $\max\{T, GST\} + d_2$, where $d_2 = 2\Delta$.

In case (i), if a correct process p executes *atab-deliver*(m) at time T, then by Lemma A.14, there is at least one correct process c that received $\langle \mathsf{COMMIT}, view_c, m\rangle$ message from $\lceil (n+b+1)/2\rceil$ processes at time $t \leq T$. Since $n \geq 3b+1$, $\lceil (n+b+1)/2\rceil - b > b$, therefore at least $b+1$ correct processes sent $\langle \mathsf{COMMIT}, view_c, m\rangle$ message. Since messages are sent using p-broadcastprimitive (and therefore resent every Δ time units), all correct processes will receive $\langle \mathsf{COMMIT}, view_c, m\rangle$ message from at least $b+1$ process the latest at time $\max\{T, GST\} + 2\Delta$, and *atab-announce* because of the rule at line 24).

In case (ii), a correct process s executes *atab-cast*(m') at time T. By line 11, s sends $\langle \mathsf{INIT}, 1, m'\rangle$ message to all at time T. Since the message is sent using p-broadcastprimitive, it is retransmitted every Δ time units, so all correct processes receive $\langle \mathsf{INIT}, 1, m'\rangle$ the latest at time $\max\{T, GST\} + 2\Delta$. By line 24 all correct processes execute *atab-announce* the latest at time $\max\{T, GST\} + 2\Delta$. □

Theorem A.1. *If* $n \geq 3b+1$*, then the CL-ATAB algorithm solves ATAB in the partially synchronous system model with known* Δ*,* $d_1 = 3\Delta$ *and* $d_2 = 2\Delta$*.*

Proof. Follows from Lemma A.9, Lemma A.11, Lemma A.13, Lemma A.12 and Lemma A.15. □

Algorithm A.2 CL-ATAB (part B) — Changing view

```
upon atab-abort() do
  $ProgressToView(2)$

upon receiving ⟨VC, $view, -, -, -$⟩ with $view > view_p$ from $b+1$ processes do
  $ProgressToView(view)$

upon receiving ⟨VC, $view_p, -, -, -$⟩ from $\lceil (n+b+1)/2 \rceil$ processes do
  after $timeout2$ execute $OnTimeout(view_p)$

upon receiving ⟨VC, $view_p, v, ts, history$⟩ from $q$ do
  if $viewChange_p[q] = \perp$ then
    $viewChange[q] \leftarrow (view_p, v, ts, history)$
    p-broadcast ⟨VC-ACK, $viewChange[q], q$⟩
    $CheckFLV$

upon receiving ⟨VC-ACK, $view_p, v, ts, history, q$⟩ from $r$ do
  if $viewChangeAck_p[q][r] = \perp$ then
    $viewChangeAck[q][r] \leftarrow (view, v, ts, history)$
    $CheckFLV$

upon receiving ⟨VC-INIT, $view_p, v, VC[]$⟩ from $coord(view_p)$ do
  if $state_p = changingView \wedge newPrePrepare_p = \perp$ then
    $newPrePrepare_p \leftarrow (view_p, v, VC[])$
    $CheckFLV$

Procedure $ProgressToView(v)$ :
  if $view_p < v$ then
    $view_p \leftarrow v$
    $viewChange[] := \perp$; $VC[] := \perp$; $viewChangeAck[][] := \perp$; $newPrePrepare_p := \perp$
    $state_p \leftarrow changingView$
    p-broadcast ⟨VC, $view_p, v_p, ts_p, history_p$⟩

Procedure $CheckFLV$ :
  for $i = 1$ to $n$ do
    if $|\{j : viewChangeAck[i][j] = viewChange[i]\}| > 2b+1$ then
      $VC[i] = viewChange[i]$
  if $p = coord(view_p)$ then
    $select \leftarrow FLV(VC[])$
    if $select \neq null$ then
      $history_p \leftarrow history_p \cup \{(select, view_p)\}$
      $state_p \leftarrow prePrepared$
      p-broadcast ⟨VC-INIT, $view_p, select, VC[]$⟩
      p-broadcast ⟨ECHO, $view_p, select$⟩
  else if $newPrepare_p \neq \perp$ and $isValid(newPrepare_p)$ then
    $history_p \leftarrow history_p \cup \{(newPrepare_p, view_p)\}$
    p-broadcast ⟨ECHO, $view_p, newPrepare_p$⟩
    $state_p \leftarrow echoed$

Procedure $isValid(m)$ :
  for $i = 1$ to $n$ do
    if $m.VC[i] \neq \perp \wedge \neq viewChange_p[i]$ or $|\{q : viewChangeAck[i][q] = m.VC[i]\}| < b+1$ then
      return false
  if $FLV(m.VC) = m.v$ then
    return true
  else
    return false

Procedure $FLV(V[])$ :
  $possibleVotes_p \leftarrow \{(vote, ts, -) \in V :$
      $|\{(vote', ts', -) \in V : \ vote = vote' \vee ts > ts'\}| \geq \lceil (n+b+1)/2 \rceil$
  $correctVotes_p \leftarrow \{v : (v, ts, -) \in possibleVotes_p \wedge$
      $|\{(vote', ts', history') \in V : (v, ts) \in history'\}| > b\}$
  if $|correctVotes_p| > 0$ then
    return $\min\{v \ s.t. \ (v, -, -) \in correctVotes_p\}$
  else if $|\{(vote, ts, -) \in V : \ ts = 0\}| \geq \lceil (n+b+1)/2 \rceil$ then
    return $\perp$
  else
    return $null$
```

Bibliography

[AABC08a] Amitanand S. Aiyer, Lorenzo Alvisi, Rida A. Bazzi, and Allen Clement. Matrix signatures: From macs to digital signatures in distributed systems. In *DISC*, pages 16–31, 2008.

[AABC08b] Amitanand S. Aiyer, Lorenzo Alvisi, Rida A. Bazzi, and Allen Clement. Matrix signatures: From macs to digital signatures in distributed systems. In *DISC*, pages 16–31. Springer-Verlag, 2008.

[AAC$^+$05] Amitanand S. Aiyer, Lorenzo Alvisi, Allen Clement, Mike Dahlin, Jean-Philippe Martin, and Carl Porth. Bar fault tolerance for cooperative services. In *SOSP*, pages 45–58, 2005.

[ABH$^+$11] Hagit Attiya, Fatemeh Borran, Martin Hutle, Zarko Milosevic, and André Schiper. Structured derivation of semi-synchronous algorithms. In *DISC*, pages 374–388, 2011.

[ABQ13] Pierre-Louis Aublin, Sonia Ben Mokhtar, and Vivien Quema. RBFT: Redundant Byzantine Fault Tolerance. In *The 33rd International Conference on Distributed Computing Systems (ICDCS)*, 2013.

[ACKL08] Yair Amir, Brian A. Coan, Jonathan Kirsch, and John Lane. Byzantine replication under attack. In *DSN*, 2008.

[ACKL11] Yair Amir, Brian Coan, Jonathan Kirsch, and John Lane. Prime: Byzantine replication under attack. *IEEE Trans. Dependable Secur. Comput.*, 8(4):564–577, July 2011.

[ACKM06] Ittai Abraham, Gregory Chockler, Idit Keidar, and Dahlia Malkhi. Byzantine disk paxos: optimal resilience with byzantine shared memory. *Distributed Computing*, 18(5):387–408, 2006.

[ADGFT06] Marcos Kawazoe Aguilera, Carole Delporte-Gallet, Hugues Fauconnier, and Sam Toueg. Consensus with byzantine failures and little system synchrony. In *DSN*, pages 147–155, 2006.

[AEMGG$^+$05] Michael Abd-El-Malek, Gregory R. Ganger, Garth R. Goodson, Michael K. Reiter, and Jay J. Wylie. Fault-scalable byzantine fault-tolerant services. *SIGOPS Oper. Syst. Rev.*, 39(5):59–74, October 2005.

[AH93] Efthymios Anagnostou and Vassos Hadzilacos. Tolerating transient and permanent failures (extended abstract). In *Proceedings of the 7th International Workshop on Distributed Algorithms*, WDAG '93, pages 174–188. Springer-Verlag, 1993.

[BCBG$^+$07] Martin Biely, Bernadette Charron-Bost, Antoine Gaillard, Martin Hutle, André Schiper, and Josef Widder. Tolerating corrupted communication. In *Proceedings of the 26th Annual ACM Symposium on Principles of Distributed Computing (PODC'07)*. ACM Press, 2007.

[BDMS13] Martin Biely, Pamela Delgado, Zarko Milosevic, and Andre Schiper. Distal: A framework for implementing fault-tolerant distributed algorithms. In *Dependable Systems and Networks (DSN), 2013 43rd Annual IEEE/IFIP International Conference on*, pages 1–8, 2013.

[BGMR01] Francisco V. Brasileiro, Fabíola Greve, Achour Mostéfaoui, and Michel Raynal. Consensus in one communication step. In *Proceedings of the 6th International Conference on Parallel Computing Technologies*, PaCT '01, pages 42–50, London, UK, 2001. Springer-Verlag.

[BHS12] Fatemeh Borran, Martin Hutle, and André Schiper. Timing analysis of leader-based and decentralized byzantine consensus algorithms. *J. Braz. Comp. Soc.*, 18(1):29–42, 2012.

[BHSS12] Fatemeh Borran, Martin Hutle, Nuno Santos, and André Schiper. Quantitative analysis of consensus algorithms. *IEEE Trans. Dependable Sec. Comput.*, 9(2):236–249, 2012.

[BMSS12] Martin Biely, Zarko Milosevic, Nuno Santos, and André Schiper. S-paxos: Offloading the leader for high throughput state machine replication. In *SRDS*, pages 111–120, 2012.

[BN01] Rida A. Bazzi and Gil Neiger. Simplifying fault-tolerance: providing the abstraction of crash failures. *J. ACM*, 48(3):499–554, May 2001.

[BO83] Michael Ben-Or. Another advantage of free choice: Completely asynchronous agreement protocols. In *PODC*, pages 27–29. ACM, 1983.

[Bra84] Gabriel Bracha. An asynchronous [(n - 1)/3]-resilient consensus protocol. In *PODC*, 1984.

[Bra87] Gabriel Bracha. Asynchronous byzantine agreement protocols. *Inf. Comput.*, 75(2):130–143, November 1987.

[BS10] Fatemeh Borran and André Schiper. A Leader-Free Byzantine Consensus Algorithm. In *ICDCN*, pages 67–78, 2010.

[BSW11] Martin Biely, Ulrich Schmid, and Bettina Weiss. Synchronous consensus under hybrid process and link failures. *Theor. Comput. Sci.*, 412(40):5602–5630, 2011.

[Bur06] Mike Burrows. The chubby lock service for loosely-coupled distributed systems. In *OSDI*, pages 335–350, 2006.

[Cac10] Christian Cachin. Yet another visit to paxos. Technical report, IBM, 2010.

[CBS09a] B. Charron-Bost and A. Schiper. The Heard-Of model: computing in distributed systems with benign failures. *Distributed Computing*, 22(1):49–71, 2009.

[CBS09b] Bernadette Charron-Bost and André Schiper. The heard-of model: computing in distributed systems with benign faults. *Distributed Computing*, 22(1):49–71, 2009.

[CJKR12] Allen Clement, Flavio Junqueira, Aniket Kate, and Rodrigo Rodrigues. On the (limited) power of non-equivocation. In *Proceedings of the 2012 ACM symposium on Principles of distributed computing*, PODC '12, pages 301–308, New York, NY, USA, 2012. ACM.

[CKPS01] Christian Cachin, Klaus Kursawe, Frank Petzold, and Victor Shoup. Secure and efficient asynchronous broadcast protocols (extended abstract). In *in Advances in Cryptology: CRYPTO*, 2001.

[CL02] Miguel Castro and Barbara Liskov. Practical byzantine fault tolerance and proactive recovery. *ACM Transactions on Computer Systems*, 20(4):398–461, 2002.

[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 implementation*, OSDI '06, pages 177–190, Berkeley, CA, USA, 2006. USENIX Association.

[CMSK07] Byung-Gon Chun, Petros Maniatis, Scott Shenker, and John Kubiatowicz. Attested append-only memory: making adversaries stick to their word. In *Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, SOSP '07, pages 189–204, New York, NY, USA, 2007. ACM.

[CNV06] Miguel Correia, Nuno Ferreira Neves, and Paulo Veríssimo. From consensus to atomic broadcast: Time-free byzantine-resistant protocols without signatures. *Comput. J.*, 2006.

[Coa88] B. A. Coan. A compiler that increases the fault tolerance of asynchronous protocols. *IEEE Trans. Comput.*, 37(12):1541–1553, December 1988.

[CT96a] Tushar Deepak Chandra and Sam Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2):225–267, March 1996.

[CT96b] Tushar Deepak Chandra and Sam Toueg. Unreliable failure detectors for reliable distributed systems. *JACM*, 43(2):225–267, March 1996.

[CWA$^+$09] Allen Clement, Edmund Wong, Lorenzo Alvisi, Mike Dahlin, and Mirco Marchetti. Making byzantine fault tolerant systems tolerate byzantine faults. In *NSDI*, pages 153–168, 2009.

[DFK06] Vadim Drabkin, Roy Friedman, and Alon Kama. Practical byzantine group communication. *Distributed Computing Systems, International Conference on*, 2006.

[DGG00a] Assia Doudou, Rachid Guerraoui, and Benoít Garbinato. Abstractions for devising byzantine-resilient state machine replication. In *SRDS*, 2000.

[DGG00b] Assia Doudou, Rachid Guerraoui, and Benoît Garbinato. Abstractions for devising byzantine-resilient state machine replication. In *SRDS*, pages 144–153, 2000.

[DH08] Danny Dolev and Ezra N. Hoch. Constant-space localized byzantine consensus. DISC, 2008.

[DHJ$^+$07] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: amazon's highly available key-value store. *SIGOPS Oper. Syst. Rev.*, 41(6):205–220, October 2007.

[DLP$^+$86] Danny Dolev, Nancy A. Lynch, Shlomit S. Pinter, Eugene W. Stark, and William E. Weihl. Reaching approximate agreement in the presence of faults. *Journal of the ACM*, 1986.

[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.

[DRS07] Dan Dobre, HariGovind V. Ramasamy, and Neeraj Suri. On the latency efficiency of message-parsimonious asynchronous atomic broadcast. In *SRDS*, 2007.

[DS97] A. Doudou and A. Schiper. Muteness failure detectors for consensus with Byzantine processes. TR 97/230, EPFL, Dept d'Informatique, October 1997.

[DS98] A. Doudou and A. Schiper. Muteness detectors for consensus with Byzantine processes (Brief Announcement). In *Proceeding of the 16th Annual ACM Symposium on Principles of Distributed Computing (PODC'98)*, Puerto Vallarta, Mexico, July 1998.

[DSU04] Xavier Défago, André Schiper, and Péter Urbán. Total order broadcast and multicast algorithms: Taxonomy and survey. *ACM Comput. Surv.*, 36(4):372–421, December 2004.

[ES06] Richard Ekwall and Andre Schiper. Solving atomic broadcast with indirect consensus. In *DSN*, 2006.

[FG03] Matthias Fitzi and Juan A. Garay. Efficient player-optimal protocols for strong and differential consensus. In *PODC*, 2003.

[FLP85] Michael J. Fischer, Nancy A. Lynch, and M. 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 *Proceeding of the 16th Annual ACM Symposium on Principles of Distributed Computing (PODC'98)*, pages 143–152, Puerto Vallarta, Mexico, 1998. ACM Press.

[GKQV10] Rachid Guerraoui, Nikola Knežević, Vivien Quéma, and Marko Vukolić. The next 700 bft protocols. In *Proceedings of the 5th European conference on Computer systems*, EuroSys '10, pages 363–376, New York, NY, USA, 2010. ACM.

[GR04] Rachid Guerraoui and Michel Raynal. The information structure of indulgent consensus. *IEEE Trans. Comput.*, 53(4):453–466, April 2004.

[GR07] Rachid Guerraoui and Michel Raynal. The alpha of indulgent consensus. *Comput. J.*, 50(1):53–67, January 2007.

[HDVR07] Chi Ho, Danny Dolev, and Robbert Van Renesse. Making distributed applications robust. In *Proceedings of the 11th international conference on Principles of distributed systems*, OPODIS'07, pages 232–246, Berlin, Heidelberg, 2007. Springer-Verlag.

[HKJR10] Patrick Hunt, Mahadev Konar, Flavio P. Junqueira, and Benjamin Reed. Zookeeper: wait-free coordination for internet-scale systems. In *USENIXATC*, page 11, 2010.

[HSGR10] James Hendricks, Shafeeq Sinnamohideen, Gregory Ganger, and Michael Reiter. Zzyzx: Scalable fault tolerance through byzantine locking. In *DSN*, pages 363–372, 2010.

[HT94] Vassos Hadzilacos and Sam Toueg. A modular approach to fault-tolerant broadcasts and related problems. Technical report, Cornell University, Ithaca, NY, USA, 1994.

[HvRBD08] Chi Ho, Robbert van Renesse, Mark Bickford, and Danny Dolev. Nysiad: practical protocol transformation to tolerate byzantine failures. In *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation*, NSDI'08, pages 175–188, Berkeley, CA, USA, 2008. USENIX Association.

[KAD$^+$07] Ramakrishna Kotla, Lorenzo Alvisi, Mike Dahlin, Allen Clement, and Edmund Wong. Zyzzyva: speculative byzantine fault tolerance. In *SOSP*, 2007.

[KGAS11] Jonathan Kirsch, Stuart Goose, Yair Amir, and Paul Skare. Toward survivable scada. In *CSIIRW*, pages 21:1–21:1, 2011.

[KMMS01] Kim Potter Kihlstrom, L. E. Moser, and P. M. Melliar-Smith. The securering group communication system. *ACM Trans. Inf. Syst. Secur.*, 2001.

[KS01] Klaus Kursawe and Victor Shoup. Optimistic asynchronous atomic broadcast. In *in the Proceedings of International Colloqium on Automata, Languages and Programming (ICALP05) (L Caires, G.F. Italiano, L. Monteiro, Eds.) LNCS 3580*, 2001.

[Lam78] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, 1978.

[Lam83] L. Lamport. The weak byzantine generals problem. *JACM*, 30(3):668–676, 1983.

[Lam98] Leslie Lamport. The part-time parliament. *ACM Transactions on Computer Systems*, 16(2):133–169, May 1998.

[Lam01] Butler Lampson. The abcd's of paxos. In *PODC*, page 13. ACM Press, 2001.

[Lam05] Leslie Lamport. Fast paxos. Technical Report MSR-TR-2005-12, Microsoft Research, 2005.

[Lam11] Leslie Lamport. Byzantizing Paxos by refinement. In *DISC*, pages 211–224, 2011.

[LCAA07] Harry C. Li, Allen Clement, Amitanand S. Aiyer, and Lorenzo Alvisi. The paxos register. In *SRDS*, pages 114–126, 2007.

[LDLM09] Dave Levin, John R. Douceur, Jacob R. Lorch, and Thomas Moscibroda. Trinc: small trusted hardware for large distributed systems. In *Proceedings of the 6th USENIX symposium on Networked systems design and implementation*, NSDI'09, pages 1–14, Berkeley, CA, USA, 2009. USENIX Association.

[LSP82] Leslie Lamport, Robert Shostak, and Marshall Pease. The byzantine generals problem. *ACM Trans. Program. Lang. Syst.*, 4(3):382–401, 1982.

[Lyn96] Nancy Lynch. *Distributed Algorithms.* Morgan Kaufman, 1996.

[MA06] J.-P. Martin and L. Alvisi. Fast byzantine consensus. *Transactions on Dependable and Secure Computing*, 3(3):202–214, 2006.

[MJM08] Yanhua Mao, Flavio P. Junqueira, and Keith Marzullo. Mencius: building efficient replicated state machines for wans. In *OSDI*, pages 369–384, 2008.

[MJM09] Yanhua Mao, Flavio P. Junqueira, and Keith Marzullo. Towards low latency state machine replication for uncivil wide-area networks. In *HotDep*, 2009.

[MMS99] Louise E. Moser and P. M. Melliar-Smith. Byzantine-resistant total ordering algorithms. *Inf. Comput.*, 1999.

[MNCV11a] Henrique Moniz, Nuno Ferreria Neves, Miguel Correia, and Paulo Verissimo. Ritas: Services for randomized intrusion tolerance. *IEEE Transactions on Dependable and Secure Computing*, 2011.

[MNCV11b] Henrique Moniz, Nuno Ferreria Neves, Miguel Correia, and Paulo Verissimo. Ritas: Services for randomized intrusion tolerance. *IEEE Trans. Dependable Secur. Comput.*, 8(1):122–136, January 2011.

[MR97] Dahlia Malkhi and Michael Reiter. Byzantine quorum systems. In *STOC*, pages 569–578, 1997.

[MR99] Achour Mostéfaoui and Michel Raynal. Solving consensus using chandra-toueg's unreliable failure detectors: A general quorum-based approach. *Distributed Computing*, 1693:847–847, 1999.

[MR10] Achour Mostéfaoui and Michel Raynal. Signature-free broadcast-based intrusion tolerance: never decide a byzantine value. OPODIS, 2010.

[MRR02] Achour Mostéfaoui, Sergio Rajsbaum, and Michel Raynal. A versatile and modular consensus protoco. In *DSN*, pages 364–373, 2002.

[Nei94] Gil Neiger. Distributed consensus revisited. *Inf. Process. Lett.*, 1994.

[NT90] Gil Neiger and Sam Toueg. Automatically increasing the fault-tolerance of distributed algorithms. *J. Algorithms*, 11(3):374–419, 1990.

[OL88] Brian M. Oki and Barbara H. Liskov. Viewstamped replication: A new primary copy method to support highly-available distributed systems. In *Proceedings of the seventh annual ACM Symposium on Principles of distributed computing*, PODC '88, pages 8–17, New York, NY, USA, 1988. ACM.

[PS85] S. S. Pinter and I. Shinahr. Distributed agreement in the presence of communication and process failures. In *Proceedings of the 14th IEEE Convention of Electrical & Electronics Engineers in Israel*. IEEE, March 1985.

[PSL80] M. Pease, R. Shostak, and L. Lamport. Reaching agreement in the presence of faults. *Journal of the ACM*, 27(2):228–234, 1980.

[PSUC02] Fernando Pedone, André Schiper, Péter Urbán, and David Cavin. Solving agreement problems with weak ordering oracles. In *Proceedings of the 4th European Dependable Computing Conference on Dependable Computing*, EDCC-4, pages 44–61, London, UK, 2002. Springer-Verlag.

[RC05] HariGovind V. Ramasamy and Christian Cachin. Parsimonious asynchronous byzantine-fault-tolerant atomic broadcast. In *OPODIS*, 2005.

[Rei94a] Michael K. Reiter. Secure agreement protocols: Reliable and atomic group multicast in rampart. In *In Proceedings of the 2nd ACM Conference on Computer and Communications Security*, 1994.

[Rei94b] Michael K. Reiter. Secure agreement protocols: reliable and atomic group multicast in rampart. In *CCS*, pages 68–80, 1994.

[RPCS08] HariGovind V. Ramasamy, Prashant Pandey, Michel Cukier, and William H. Sanders. Experiences with building an intrusion-tolerant group communication system. *Softw. Pract. Exper.*, 2008.

[RPS12] Tom Roeder, Rafael Pass, and Fred B. Schneider. Multi-verifier signatures. *J. Cryptology*, 25(2):310–348, 2012.

[RSA78] R. L. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Commun. ACM*, 21(2):120–126, February 1978.

[SAAAA95] Hasan M. Sayeed, Marwan Abu-Amara, and Hosame Abu-Amara. Optimal asynchronous agreement and leader election algorithm for complete networks with byzantine faulty links. *Distrib. Comput.*, 9:147–156, December 1995.

[Sch90] Fred B. Schneider. Implementing fault-tolerant services using the state machine approach: a tutorial. *ACM Comput. Surv.*, 22(4), December 1990.

[SCY98] Hin-Sing Siu, Yeh-Hao Chin, and W.-P. Yang. Byzantine agreement in the presence of mixed faults on processors and links. *IEEE Trans. Parallel Distrib. Syst.*, 9:335–345, April 1998.

[ST87] T. K. Srikanth and Sam Toueg. Simulating authenticated broadcasts to derive simple fault-tolerant algorithms. *Distributed Computing*, 2(2):80–94, 1987.

[SvRSD08] Yee Jiun Song, Robbert van Renesse, Fred B. Schneider, and Danny Dolev. The building blocks of consensus. In *ICDCN*, pages 54–72, 2008.

[SW89] Nicola Santoro and Peter Widmayer. Time is not a healer. In *Proc. 6th Annual Symposium on Theor. Aspects of Computer Science (STACS'89)*, volume 349 of *LNCS*, pages 304–313, Paderborn, Germany, February 1989. Springer-Verlag.

[SW07] Nicola Santoro and Peter Widmayer. Agreement in synchronous networks with ubiquitous faults. *Theor. Comput. Sci.*, 384(2-3):232–249, October 2007.

[SWR02] Ulrich Schmid, Bettina Weiss, and John Rushby. Formally verified byzantine agreement in presence of link faults. In *22nd International Conference on Distributed Computing Systems (ICDCS'02)*, pages 608–616, Vienna, Austria, July 2-5, 2002.

[TC84] R. Turpin and B. A. Coan. Extending binary byzantine agreement to multivalued byzantine agreement. *Information Processing Letters*, 1984.

[VCBL09] Giuliana Santos Veronese, Miguel Correia, Alysson Neves Bessani, and Lau Cheuk Lung. Spin one's wheels? byzantine fault tolerance with a spinning primary. In *SRDS*, 2009.

[WLG$^+$78] John H. Wensley, Leslie Lamport, Jack Goldberg, Senior Member, Milton W. Green, Karl N. Levi'it, P. M. Melliar-smith, Robert E. Shostak, Charles, and B. Weinstock. Sift: Design and analysis of a fault-tolerant computer for aircraft control. In *Proceedings of the IEEE*, pages 1240–1255, 1978.

[YCW92] K. Q. Yan, Y. H. Chin, and S. C. Wang. Optimal agreement protocol in malicious faulty processors and faulty links. *IEEE Trans. on Knowl. and Data Eng.*, 4(3):266–280, June 1992.

[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*, ISPDC '06, pages 24–33, Washington, DC, USA, 2006. IEEE Computer Society.

Publications

Chapter 3

Zarko Milosevic, Martin Hutle and André Schiper. Unifying Byzantine Consensus Algorithms with Weak Interactive Consistency. In *13th International Conference On Principle Of Distributed Systems (OPODIS 2009)*, Nimes, France, December 15 - December 18, 2009.

Chapter 4

Zarko Milosevic, Martin Hutle and André Schiper. Tolerating Permanent and Transient Value Faults. In *Distributed Computing*, DOI: 10.1007/s00446-013-0199-7.
Brief announcement at the *31st Annual ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing (PODC 2012)*, Madeira, Portugal, July 16 - July 18, 2012. Extended version under submission.

Chapter 5

Olivier Rütti, Zarko Milosevic, André Schiper. Generic construction of consensus algorithms for benign and Byzantine faults. In *The 40th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2010)*, Chicago, Illinois, USA, June 28 - July 1, 2010.

Chapter 6

Zarko Milosevic, Martin Hutle and André Schiper. On the Reduction of Atomic Broadcast to Consensus with Byzantine Faults. In *30th International Symposium On Reliable Distributed Systems (SRDS 2011)*, Madrid, Spain, October 4 - October 18, 2011.

Chapter 7

Zarko Milosevic, Martin Biely and André Schiper. Bounded Delay in Byzantine-Tolerant State Machine Replication. In *32th International Symposium On Reliable Distributed Systems (SRDS 2013)*, Braga, Portugal, September 30 - October 2, 2013.

Curriculum Vitae

Zarko Milosevic was born in Cacak, Serbia, on September 26th, 1982. He obtained his graduated engineer (dipl.ing.) degree in Computer Science in 2006. As part of his diploma project he has designed and implemented a tool for network monitoring based on Cisco IOS Netflow technology. The project has been done during his internship at the Belgrade University Computing Center (Serbia).

In 2006, he continued his graduate studies at the Swiss Federal Institute of Technology in Lausanne, Switzerland (EPFL) in the School of Computer and Communication Sciences (IC). After obtaining M.Sc. degree in Computer Science from EPFL in 2008, he started his doctoral studies under the supervision of Prof. André Schiper at EPFL IC in the Distributed Systems Laboratory. His main interests are: (Byzantine) fault tolerance, dependable distributed computing and security. His research focuses on abstractions for Byzantine Fault Tolerant (BFT) protocols.