# TIME-COMPLEXITY BOUNDS ON AGREEMENT PROBLEMS

THÈSE N$^{O}$ 3261 (2005)

PRÉSENTÉE À LA FACULTÉ INFORMATIQUE ET COMMUNICATIONS

Institut d'informatique fondamentale

SECTION DES SYSTÈMES DE COMMUNICATION

ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

POUR L'OBTENTION DU GRADE DE DOCTEUR ÈS SCIENCES

PAR

Partha DUTTA

Bachelor of Technology in Electrical Engineering, Indian Institute of Technology, Kanpur, Inde
et de nationalité indienne

acceptée sur proposition du jury:

Prof. R. Guerraoui, directeur de thèse
Prof. D. Dolev, rapporteur
Prof. B. Faltings, rapporteur
Prof. M. Raynal, rapporteur

Lausanne, EPFL
2005

# Abstract

In many distributed systems, designing an application that maintains consistency and availability despite failure of processes, involves solving some form of agreement. Not surprisingly, providing efficient agreement algorithms is critical for improving the performance of many distributed applications. This thesis studies how fast we can solve fundamental agreement problems like consensus, uniform consensus, and non-blocking atomic commit.

In an agreement problem, the processes are supposed to propose a value and eventually decide on a common value that depends on the proposed values. To study agreement problems, we consider two round-based message-passing models, the well-known synchronous model, and the eventually synchronous model. The second model is a partially synchronous model that remains asynchronous for an arbitrary number of rounds but eventually becomes synchronous.

We investigate two aspects of the performance of agreement algorithms. We first measure time-complexity using a finer-grained metric than what was considered so far in the literature. Then we optimize algorithms for subsets of executions that are considered to be common in practice.

Traditionally, the performance of agreement algorithms was measured in terms of *global decision*: the number of rounds required for *all* correct (non-faulty) processes to decide. However, in many settings, upon deciding, any correct process can provide the decision value to the process that is waiting for a decision. In this case, a more suitable performance metric is a *local decision*: the number of rounds required for *at least one* correct process to decide. We present tight bounds for local decisions in the synchronous and the eventually synchronous models. We also show that considering the local decision metric allows us to uncover fundamental differences between agreement problems, and between models, that were not apparent with previous metrics.

In the eventually synchronous model, we observe that, for many cases in practice, executions are frequently synchronous and only occasionally asynchronous. Thus we optimize algorithms for *synchronous executions*, and give matching lower bounds. We show that, in some sense, synchronous executions of algorithms designed for the eventually synchronous model are slower than executions of algorithms directly

designed for the synchronous model, i.e., there is an inherent price associated with tolerating arbitrary periods of asynchrony. Finally, we establish a tight bound on the number of rounds required to reach agreement once an execution becomes synchronous and no new failures occur.

# Résumé

Dans les systèmes répartis, la conception d'une application consistante et disponible malgré des erreurs de processus nécessite un protocole d'accord. Comme on peut s'y attendre, il est difficile de fournir des algorithmes d'accord efficaces. Cette thèse étudie la complexité des problèmes d'accord fondamentaux comme le consensus, le consensus uniforme, et la validation atomique non-bloquante.

Dans un problème d'accord, les processus sont supposés proposer une valeur et ensuite décider d'une valeur commune qui sera déterminée en fonction des valeurs proposées. Pour étudier les problèmes d'accord, nous considérons deux modèles de communication basés sur des rondes: le modèle synchrone bien connu, et le modèle finalement synchrone. Le second modèle est un modèle partiellement synchrone, qui reste asynchrone pour un nombre arbitraire de rondes, pour finalement devenir synchrone.

Nous étudions deux aspects de la performance d'algorithmes d'accord. D'abord, nous mesurons la complexité en temps avec une métrique plus fine que celle utilisée jusqu'à présent dans la littérature. Puis nous optimisons les algorithmes pour les sous-ensembles d'exécutions considérés comme les plus fréquentes dans la pratique.

Traditionnellement, la performance d'algorithmes d'accord est mesurée en termes de décision globale: le nombre de rondes requis pour que tous les processus corrects (sans défaillance) décident. Cependant, dans bien des contextes, n'importe quel processus correct peut fournir la valeur de décision au processus qui attend qu'une décision soit prise. Dans ce cas, une métrique de la performance mieux adaptée sera la décision locale: le nombre de rondes requis pour qu'au moins un processus correct puisse décider. Nous présentons des bornes exactes pour les décisions locales dans les modèles synchrone et finalement synchrone. Nous montrons également qu'en matière de décision locale, notre métrique nous permet de découvrir des différences fondamentales entre les problèmes d'accord, et entre les modèles eux-mêmes, différences qui n'apparaissaient pas avec les métriques précédentes.

Dans le modèle finalement synchrone, nous observons que dans bien des cas, les exécutions sont en pratique souvent synchrones, et seulement occasionnellement asynchrones. Nous optimisons donc les algorithmes pour des exécutions synchrones, et donnons des bornes inférieures adaptées. Nous montrons que d'une certaine façon,

les exécutions synchrones d'algorithmes conçus pour le modèle finalement synchrone sont plus lentes que les exécutions d'algorithmes conçus directement pour le modèle synchrone; cela signifie que la tolérance aux périodes asynchrones a un prix. Finalement, nous établissons une borne inférieure exacte sur le nombre de rondes requis pour chaque accord une fois qu'une exécution devient synchrone et qu'aucune nouvelle erreur n'apparaît.

# Acknowledgements

First of all, I would like to thank my advisor Rachid Guerraoui who introduced me to the wonderful world of distributed algorithm, and has been a constant source of inspiration and encouragement. Without his guidance this thesis would not have materialized.

I am grateful to Prof. Emre Telatar for presiding over my thesis exam and to the members of the jury, Prof. Danny Dolev, Prof. Boi Faltings, and Prof. Michel Raynal, for the time they spent examining my thesis.

I am indebted to my co-authors, in alphabetical order, Romain Boichat, Arindam Chakraborty, Svend Frølund, Idit Keidar, Leslie Lamport, Ron Levy, Bastian Pochon, and Marko Vukolić. Their ideas and comments were invaluable in writing the papers and this thesis.

I would like to thank Petr Kouznetsov for many discussions on distributed algorithms which improved my understanding of the topic. Many thanks also to Sidath Handurukande and Arnas Kupsys for many interesting coffee-sessions. The current and the past members of the Distributed Programming Laboratory provided a great working environment, and Kristine Verhamme took care of many important details — a big thank you to them. I would also like to express my thanks to all my friends in Lausanne, Martigny, and Zurich who made my Ph.D. days an enjoyable experience.

Finally, I want to thank my parents and sister for their love and support.

# Preface

This thesis concerns the Ph.D. work I did under the supervision of Prof. Rachid Guerraoui at the School of Computer and Communication Sciences, EPFL, from 2000 to 2005. During this period, I also worked on (1) a deconstruction of the Paxos algorithm of Lamport [DFGP02, BDFG03a, BDFG03b], (2) incompatibility results for non-blocking atomic commit [DGP04], and (3) time-complexity of atomic register implementations [DGLC04]. More recently, I also worked on the best-case time-complexity of Byzantine agreement [DGV04].

This thesis focuses on the time-complexity of agreement algorithms where processes may fail by crashing, and it is a composition of extended and revised version of four papers: [DG02a, DG02b, DGP03, DGK04].


[BDFG03a] R. Boichat, P. Dutta, S. Frølund, and R. Guerraoui. Deconstructing Paxos. *Distributed Computing Column of ACM SIGACT News*, 34(1):47–67, March 2003.

[BDFG03b] R. Boichat, P. Dutta, S. Frølund, and R. Guerraoui. Reconstructing Paxos. *Distributed Computing Column of ACM SIGACT News*, 34(2):42–57, June 2003.

[DFGP02] P. Dutta, S. Frolund, R. Guerraoui, and B. Pochon. An efficient universal construction for message-passing systems. In *Proceedings of the 16th International Symposium on Distributed Computing (DISC-16)*, number 2508 in Lecture Notes in Computer Science, pages 133–147. Springer-Verlag, October 2002.

[DG02a] P. Dutta and R. Guerraoui. Fast indulgent consensus with zero degradation. In *Proceedings of the Fourth European Dependable Computing Conference (EDCC-4)*, number 2485 in Lecture Notes in Computer Science, pages 191–208. Springer-Verlag, October 2002.

[DG02b] P. Dutta and R. Guerraoui. The inherent price of indulgence. In *Proceedings of the 21st ACM Symposium on Principles of Distributed Computing (PODC-21)*, pages 88–97, July 2002. To appear in Distributed Computing.

[DGK04] P. Dutta, R. Guerraoui, and I. Keidar. The overhead of consensus failure recovery. IC Technical Report 200456, École Polytechnique Fédérale de Lausanne, June 2004.

[DGLC04] P. Dutta, R. Guerraoui, R. Levy, and A. Chakraborty. How fast can a distributed atomic read be? In *Proceedings of the 23rd ACM Symposium on Principles of Distributed Computing (PODC-23)*, pages 236-245, July 2004.

[DGP03] P. Dutta, R. Guerraoui, and B. Pochon. Tight bounds on early local decisions. In *Proceedings of the 17th International Symposium on Distributed Computing (DISC-17)*, number 2848 in Lecture Notes in Computer Science, pages 264–278. Springer-Verlag, October 2003.

[DGP04] P. Dutta, R. Guerraoui, and B. Pochon. Fast non-blocking atomic commit: An inherent tradeoff. *Information Processing Letters (IPL)*, 91(4):195–200, August 2004.

[DGV04] P. Dutta, R. Guerraoui, and M. Vukolić. The complexity of asynchronous Byzantine consensus. IC Technical Report 200499, École Polytechnique Fédérale de Lausanne, November 2004.

# Contents

# List of Figures

# Chapter 1

# Introduction

Fault-tolerant agreement lies at the heart of many critical computer applications. For instance, one of the primary ways to simultaneously maintain application consistency and availability despite failures is state-machine replication [Lam78, Lam89, Sch90]: a central server is emulated by many replica servers, some of which may fail, and replicas *agree* on the order in which the requests are executed. Similarly, a distributed transaction service may require database servers to agree on whether to commit or abort a transaction [Gra78]. Indeed, providing a fault-tolerant service frequently reduces to solving some agreement problem, and the performance of the service is directly impacted by the efficiency of the underlying agreement algorithm. The goal of this thesis is to investigate how fast we can achieve agreement in a distributed system.

## 1.1 Context

### Distributed systems

A distributed system is a collection of computing entities (also called, processes) that communicate with each other. In a typical distributed computation, the processes perform local computation and exchange relevant information to achieve some global objective. Models of distributed systems differ according to how the processes communicate. In a message-passing model, the processes communicate by sending and receiving messages over point-to-point or multicast communication channels. In a shared-memory model, the processes communicate by reading or modifying shared objects. Models also differ in how processes fail, i.e., deviate from the assigned algorithms. In this work, we exclusively consider point-to-point message-passing models where a process may fail only by crashing, i.e., by prematurely stopping its execution, and crashed processes do not recover. We also assume that there is an upper bound on the number of processes that may crash in any execution. Non-faulty processes are also called correct processes.

### Agreement problems

An agreement algorithm enables a set of processes to agree on a common decision value (output) that depends on the proposal values (input) of the processes. A primary example of an agreement problem is consensus [LSP82, FLP85], which requires that no two correct processes decide differently (agreement), every correct process eventually decides (termination), and a decision value is the proposal value of some process (validity). Uniform consensus is variant of consensus that, in addition, requires that no two processes decide differently (uniform agreement) [Had87, HT93]. We also consider in this thesis two other closely related agreement problems: non-blocking atomic commit and interactive consistency.

### Synchrony assumptions

Agreement problems have been extensively studied in the *synchronous model* [Lyn96, AW98, Ray02], where computation proceeds in rounds of message exchange. In an execution of the synchronous model, for every round $k$, all messages sent by a process in round $k$ are received in the same round, unless the process crashes in round $k$, in which case any subset of messages sent by the process may be lost. Although the strong synchrony assumptions in the synchronous model allow us to design simple and efficient agreement algorithms, we would typically like our algorithm to work even with weak, or possibly no, timing assumptions. However, in one of the seminal results in distributed computing, Fischer et al. [FLP85] showed that if we make no timing assumptions (the asynchronous model) then consensus is impossible to solve even if a single process may crash. To circumvent this impossibility, agreement problems are frequently studied in partially synchronous models that make weak timing assumptions but still allow us to solve the problems [DDS87, DLS88]. We consider in this thesis, a natural partially synchronous model, called the *eventually synchronous model*, where any execution is asynchronous for an arbitrary period of time but eventually becomes synchronous. More precisely, every execution has an unknown round number, called the Global Stabilization Round ($GSR$), such that any message sent in a round lower than $GSR$ may be lost, but for every round $k \geq GSR$, all messages sent by a process in round $k$ are received in the same round, unless the process crashes in round $k$, in which case any subset of messages sent by the process may be lost (i.e., from round $GSR$, an execution behaves as in the synchronous model).

### Time-complexity of agreement

An obvious measure of the efficiency of an agreement algorithm is how quickly all correct processes can decide, or halt, called global decision and global halting, respectively. (We say that a process halts if the algorithm assigns no further computation step to the process.) Two points are worth noting, (1) only the time required by the correct processes is considered because faulty processes may crash before deciding, and (2) upon deciding, a process may continue to send messages to help other processes decide, and thus, may not halt immediately.

Let us consider the synchronous model and suppose that at most $t$ out of a total of $n$ processes, $p_1, ..., p_n$, may crash in any execution. One of the earliest lower bounds obtained on agreement algorithms is the number of rounds required in the worst-case: there is an execution of every consensus algorithm that takes at least $t + 1$ rounds for a global decision [FL82, DLM82, DS83]. (For ease of presentation, in this chapter, we ignore the boundary cases, e.g. when $t$ is close to $n$.) This lower bound is tight, i.e., there is a consensus algorithm that globally decides by round $t+1$ in every execution. The same tight bound also holds for global halting, and for the three other agreement problems that we consider, namely, uniform consensus, non-blocking atomic commit, and interactive consistency.

Worst-case executions are rare in practice. Thus, it is interesting to investigate whether algorithms can globally decide or halt before $t + 1$ rounds for some subset of executions. In particular, one immediate generalization of the above worst-case lower bound is to consider lower bounds for the subset of executions in which at most $f$ processes crash, for some given $f \leq t$. (The worst-case lower bound is simply the special case $f = t$.) These bounds are called early decision or early halting bounds and were first studied in [DRS90], where it was shown that the lower bound for early global halting is $f + 2$ rounds for consensus. Furthermore, a simple extension of the $t+1$ bound implies that the bound for a early global decision is $f+1$ rounds for consensus. In contrast, both early global decision and early global halting lower bounds are $f + 2$ rounds for uniform consensus, as well as for non-blocking atomic commit and interactive consistency [KR03, CBS04]. All these bounds are tight [CBS04, DGFHR03, CBF04].

Now let us consider the eventually synchronous model. In this model, any consensus algorithm also solves uniform consensus, and moreover, non-blocking atomic commit and interactive consistency are impossible to solve in the presence of crashes [Gue95]. Thus, we only consider uniform consensus in the eventually synchronous model.

As any message may be lost before *GSR*, clearly, the processes can not decide before *GSR*. The only tight bound known in this model was the special case where $GSR = 1$ and $f = 0$ (i.e., executions that are failure-free and synchronous from the very beginning): every uniform consensus algorithm in the eventually synchronous model has a failure-free synchronous execution which requires at least 2 rounds for a global decision [KR03]. The most efficient uniform consensus algorithm known in this model required $2f + 2$ rounds in synchronous executions with at most $f$ crashes [MR99].

## 1.2 Motivation

### Synchronous model

As we discussed above, lower bounds on the time complexity of agreement have been traditionally stated in terms of the time required for *all* correct processes to decide or halt. From a practical perspective, what we might sometimes want to measure and optimize, is the time needed for *at least one* correct process to decide, i.e., for a *local decision*. Indeed, a replicated service can respond to its clients as soon as a single replica decides on a reply (and knows that other replicas will reach the same decision). Similarly, the client of a transaction service might be happy to know the outcome once it has been determined, even if some database servers have yet to be informed of the outcome.

Surprisingly, despite the large body of work on the performance of agreement, so far, no study on local decision lower bounds has appeared in the literature. To get an intuition that the local decision bound may be different from the global decision bound, consider the following simple consensus algorithm in the synchronous model. Every process maintains an estimate value that is initialized to its proposal value. Process $p_1$ decides at the very beginning (which we call deciding at round 0) on its estimate, and then sends the decision value to all in round 1. In general, at the end of round $i - 1$, process $p_i$ decides on its estimate, and then sends its decision value to all in round $i$. Moreover, if a process receives a decision value, then it adopts that value as its estimate. It is easy to see that the algorithm satisfies the agreement property of consensus: if $p_i$ is the lowest correct process to decide, then at the end of round $i$, every alive (non-crashed) process has adopted the decision value of $p_i$ as its estimate. Thus, no correct process can decide on a different value. Observe that, in executions of this algorithm with at most $f \leq t$ crashes, at least one correct process decides by round $f$. Thus, for consensus, the early local decision tight bound cannot be more than $f$, whereas, we know that the early global decision tight bound is $f + 1$. In fact, we show that the early local decision tight bound for consensus is $f$. But this observation raises several new questions.

- Can we match both the local and the global decision lower bounds with the *same* algorithm? The consensus algorithm we sketched above matches the local decision lower bound but clearly does not match the global decision bound: it has an execution in which some correct process decides in round $n$. Is there any algorithm that matches both bounds?

- Does the lower bound change if we consider a more general metric, namely, the number of rounds required for at least $c$ correct processes to decide? (Recall that a local decision requires at least one correct process to decide, and a global decision requires all correct processes to decide.)

- What is the local decision bound for uniform consensus, non-blocking atomic commit and interactive consistency? Can their local and global decision bounds be matched by the same algorithm?

### Eventually synchronous model

Although algorithms in the eventually synchronous model tolerate arbitrary periods of asynchrony, for many cases in practice, executions are frequently synchronous and only occasionally asynchronous. Thus, it is important to optimize algorithms so that processes decide quickly in synchronous executions (i.e., in executions with $GSR = 1$). Recall that we only consider uniform consensus in this model.

Obviously, any lower bound in the synchronous model also holds in synchronous executions of the eventually synchronous model. But are these bounds tight? In particular,

- Does the $t + 1$ rounds worst-case bound in the synchronous model also holds in synchronous executions of the eventually synchronous model?
- What is the tight bound for early local and global decisions in synchronous executions?

More generally, how quickly can a uniform consensus algorithm globally decide after the system becomes synchronous?

## 1.3 Contributions

### Synchronous model

We show that the lower bound for early local decision is $f$ rounds for consensus, i.e., for every consensus algorithm that can tolerate $t$ crashes, there is an execution with at most $f \leq t$ crashes that takes at least $f$ rounds for even one correct process to decide. More interestingly, we show that no single consensus algorithm can match both the early local decision and the early global decision bounds for even two consecutive values of $f$. Furthermore, we show that the number of rounds needed for even two correct processes to decide is the same as that for a global decision; i.e., $f + 1$ rounds.

In the synchronous model, we also show that the early local decision lower bound for uniform consensus, non-blocking atomic commit and interactive consistency is

$f + 1$ rounds, except for the failure-free case ($f = 0$) where, for uniform consensus, the lower bound is 1 round, and for the other two problems, the lower bound is 2 rounds. For all the three problems, the number of rounds needed for even two correct processes to decide is same as that for a global decision; i.e., $f + 2$ rounds. However, unlike consensus, for each of the remaining three problems, we show that a single algorithm can match the local decision, global decision, and global halting bounds.

### Eventually synchronous model

Our results in the eventually synchronous model reveal that there is an inherent price of tolerating asynchrony (i.e., a price of indulgence [Gue00]). More precisely, for every uniform consensus algorithm in the eventually synchronous model where at most $t$ processes can crash in any execution, we show that:

- (worst-case global decision) there is a synchronous execution which requires at least $t + 2$ rounds for all correct processes to decide. (The corresponding bound for uniform consensus algorithms in the synchronous model is $t + 1$.)
- (early local decision) for every $f \leq t-3$, there is a synchronous execution with at most $f$ crashes which requires at least $f + 2$ rounds for even one correct process to decide. (The corresponding bound for uniform consensus algorithms in the synchronous model is $f + 1$.)

We then present a matching algorithm for the above two bounds: for every $f \leq t$, our uniform consensus algorithm globally decides (and therefore, locally decides) by round $f + 2$ in every synchronous execution with at most $f$ crashes. Thus, synchronous executions of algorithms designed in the eventually synchronous model are slower than executions of algorithms designed directly for the synchronous model.

Finally, we address the question of how fast can an algorithm globally decide after the system becomes synchronous. We consider the number of rounds required for a global decision after the system becomes synchronous and no new crash occurs. (Note that this bound is different from the number of rounds required after *GSR* because from *GSR* onwards, the system becomes synchronous but the processes may still crash.) Perhaps surprisingly, the bound depends on the total number of crashes that are tolerated, and in particular, whether $t \geq n/3$:

- For every uniform consensus algorithm in the eventually synchronous model, there is an execution that requires at least two rounds for a global decision after the system becomes synchronous and no new crash occurs.
- When $t \geq n/3$, for every uniform consensus algorithm in the eventually synchronous model, there is an execution that requires at least three rounds for a global decision after the system becomes synchronous and no new crash occurs.

We also give matching algorithms for both cases, $t < n/3$ and $t \geq n/3$. In addition, when $t < n/3$, we show that the tight bound on the number of rounds required for early global decision after the system becomes synchronous (i.e., after $GSR$) is $f+2$.

### Roadmap

This thesis is organized as follows. In Chapter 2, we give the definitions of the models and the agreement problems that we consider. We also define the time-complexity metrics for agreement, and a compact notation for presenting the lower bounds. Chapter 3 introduces the layering technique from [MR02, KR03] and some related results that are useful for proving our lower bounds. Chapter 4 presents our results in the synchronous model. Our results on uniform consensus in the eventually synchronous model are covered in the following two chapters. Chapter 5 focuses on the lower bounds in synchronous executions, whereas Chapter 6 gives the bounds for a global decision once the execution becomes synchronous and no new crash occurs. We conclude the thesis in Chapter 7 by summarizing our results and discussing some open issues.

# Chapter 2

# Background

## Part A — Definitions

### 2.1 Models

A distributed system is a collection of computing entities that may communicate with each other. In this section, we present some models of distributed systems that are relevant to this thesis.

**Round-Based Model *(RM)*.** We consider a distributed system model consisting of a finite and static set of processes, any pair of which may communicate by message-passing over a bi-directional communication channel. The set of processes is denoted by $\Pi = \{p_1, p_2, \ldots, p_n\}$, where $i$ is the process identifier (pid) of process $p_i$, and we consider $n \geq 3$.

Each process is assigned a deterministic state machine (with possibly infinite states). The set of possible states is denoted by $Q_i$, and the set of initial states by $Init_i \subseteq Q_i$. An algorithm $A$ specifies the state machine $A^i$ that is assigned to each process $p_i$. We model the channels using a single set *mset* (a message buffer), the state of which, at any given point of computation, is the set of messages that are sent but not yet received. We assume that every message $m$ has a unique message identification tag $m.id$, and two tags which identify the sender and the recipient of $m$, $m.sender$ and $m.recp$, respectively.

Given an algorithm $A$, a *run* of $A$ is an infinite sequence of rounds (of message-exchange). Rounds are identified by round numbers that are positive integers starting from 1. Each round consists of three subrounds executed one after the other. In each subround, processes *atomically* perform the following actions in lock-step:

- *Send subround*: each process $p_i$ sends a message to every process, i.e., puts $n$ messages in *mset*.

- *Receive subround*: each process $p_i$ receives some set of messages $M$, i.e., removes some messages from *mset*. ($M$ might be the emptyset $\emptyset$.)
- *Computation subround*: each process $p_i$ applies the set of messages $M$ received in the receive subround to the state machine $A^i$ assigned to $p_i$. $A^i$ changes its state accordingly and outputs a message for each process, to be send in the send subround of the next round.

A run of of $A$ satisfies the following properties for every process $p_i$:

1. Initially, $mset = \emptyset$.
2. The initial state of $p_i$ is in $Init_i$.
3. If $p_i$ receives a message $m$ in the receive subround of a round, then $m.rcpt = p_i$, and $m \in mset$ immediately before that subround
4. If $p_i$ does not execute a subround, then it does not execute any subsequent subround, or a higher round; i.e., the processes are crash-stop.

We now introduce some definitions on the runs in *RM*.

- We say that a process is *correct* in a run if it executes an infinite number of rounds in that run. Otherwise, the process is said to be *faulty*. If subround $sb$ of round $k$ is the last subround executed by a faulty process $p_i$, then we say that $p_i$ *crashes* at subround $sb$ of round $k$, or simply that $p_i$ crashes at round $k$.
- A process *enters* a round $k$ if it executes the send subround of round $k$, and a process *completes* round $k$ if it executes all subrounds of round $k$.
- If $p_i$ completes round $k-1$ but does execute any subround of round $k$, then we say that $p_i$ *crashes at the beginning of round* $k$. Furthermore, if $p_i$ crashes at the beginning of round 1, then we say that $p_i$ *crashes initially*.
- A *round* $k$ *message of* $p_i$ is a message sent by $p_i$ in the send subround of round $k$. We say that a message $m$ is *lost* in run $r$ if $m$ is sent but not received in run $r$.
- A *model* is a set of runs selected (from all possible runs of all possible algorithms in *RM*) by restricting when processes can crash, and specifying which messages are received. A model $M'$ is a *submodel* of model $M$ if $M' \subseteq M$.

A generic algorithm (modified from [Gaf98]) in *RM* is shown in Figure 2.1. A specific algorithm additionally describes the local computation done in lines 1 and 5.

```
at process p_i
initialize()
in round k                                    {rounds 1, 2,3 ...}
    send round k messages
    receive messages
    compute()
```

Figure 2.1: A generic algorithm in *RM*

**Eventually Synchronous Model (EM).** The *E*ventually synchronous *M*odel, denoted *EM*, is a submodel of *RM*. Every run $r$ in *EM* satisfies the following three properties:

1. *loopback* — if a process executes the receive subround in round $k$, then the process receives its own round $k$ message in that subround,

2. *communication closed rounds* — every message received in a receive subround is sent in the send subround of that round, and

3. *eventual synchrony* — there is an unknown but finite round number $GSR(r)$ (Global Stabilization Round of run $r$) such that, in every round $k \geq GSR(r)$, if a process $p_i$ executes the receive subround of round $k$, then every process $p_j$ that executes the receive subround of round $k$, receives in that subround, the round $k$ message sent by $p_i$ to $p_j$. If $p_i$ crashes in the send subround, i.e., does not execute the receive subround, then there are no delivery guarantees: any message sent by $p_i$ in round $k$ may be lost. (We sometimes drop the parameter $r$ in $GSR(r)$ when we make a general statement about all runs in the model.)

For $1 \leq t \leq n$, model $EM_t$ is a submodel of *EM* containing all runs of *EM* in which at most $t$ processes are faulty. Notice that, for $1 \leq t < w \leq n$, $EM_t \subset EM_w$, and $EM_n = EM$.

**Synchronous Model (SM).** The *S*ynchronous *M*odel, denoted by *SM*, is a submodel of *EM* that consists of all runs $r$ of *EM* such that $GSR(r) = 1$. For $1 \leq t \leq n$, model $SM_t$ is a submodel of *SM* that consists of all runs of *SM* in which at most $t$ processes are faulty. $SM1_t$ is a submodel of $SM_t$ that consists of all runs of $SM_t$ in which *at most one* process crashes in each round. Notice that, for $1 \leq t < w \leq n$, $SM_t \subset SM_w$, $SM_n = SM$, and $SM_t \subset EM_t$.

**Discussion.** We only consider models $EM_t$, $SM_t$, and $SM1_t$ ($1 \leq t \leq n$) in this thesis. Model *SM* is the well-known synchronous crash-stop model [Lyn96, AW98]. Model *EM* is simply a round based model that eventually provides the same guarantees as the synchronous model *SM*. Model *EM* is inspired by the fail-stop Basic

Round Model of [DLS88], which we denote here by *DLS*. In every run of *DLS*, there is a round $GST$ in every run such that, in round $GST$ and in higher rounds, messages sent from correct processes to correct processes are received in the same round in which they are sent. Thus, in *DLS*, messages sent by faulty processes are never guaranteed to be received. In $EM$ however, any message sent in round $k \geq GSR$ is guaranteed to be received if the sender and the recipient do not crash by round $k$.

## 2.2 Agreement problems

In the following, we give an overview of the agreement problems studied in this thesis. We assume that the state of every process $p_i$ contains two special components: $prop_i$ and $dec_i$. Component $prop_i$ cannot be modified, and component $dec_i$ can be modified at most once. In addition, there is a known set of values $V$ such that, the value $\perp$ is not in $V$, there are at least two distinct values in $V$, and in every state in $Init_i$, $prop_i$ equals a value from $V$ and $dec_i$ equals $\perp$. We say that the *proposal value* of $p_i$ in a run $r$ is $v$, or $p_i$ *proposes* $v$, if $prop_i$ equals $v$ in the initial state of $p_i$ in run $r$. We say that $p_i$ *decides* $d$ ($d$ is not necessarily in $V$) in $r$ if $p_i$ executes a subround in $r$ which sets $dec_i$ to $d$. (However we make an exception for the trivial case where a process decides before sending any message, i.e., the process computes the decision value without any external communication. In this trivial case, we assume that $dec_i$ contains the decision value at initialization, and we say that $p_i$ *decides initially*.)

This work is primarily concerned with determining time-complexity bounds for the consensus and the uniform consensus problems. Additionally, we prove our lower bound results in the synchronous model for weak binary agreement and weak binary uniform agreement to strengthen the results. We also contrast our bounds for uniform consensus with those for the non-blocking atomic commit and the interactive consistency problems. We now define these agreement problems.

An algorithm $A$ solves consensus, uniform consensus or interactive consistency, in model $M$, if all runs of $A$ in $M$ satisfy the following properties, respectively:

- **Consensus (denoted by NC)** [LSP82, FLP85] (a) (*agreement*) no two correct processes decide differently, (b) (*termination*) every correct process eventually decides, and (c) (*validity*) if a process decides $v$ then some process has proposed $v$.
- **Uniform consensus (denoted by UC)** [Had87, HT93] (a) (*uniform agreement*) no two processes decide differently, (b) (*termination*) every correct process eventually decides, and (c) (*validity*) if a process decides $v$ then some process has proposed $v$.
- **Interactive consistency (denoted by IC)** [PSL80] (a) (*uniform agreement*) no two processes decide differently, (b) (*termination*) every correct process

eventually decides, and (c) (*IC validity*) every process that decides, decides on an ordered $n$-tuple $D$ such that the $j^{th}$ element of $D$ is either the proposal value of $p_j$ or $\perp$, and may be $\perp$ only if $p_j$ is faulty.

An agreement problem is called binary if $V$ is fixed to $\{0, 1\}$. We consider three binary agreement problems. An algorithm $A$ solves non-blocking atomic commit, weak binary agreement or weak binary uniform agreement, in model $M$, if all runs of $A$ in $M$ satisfy the following properties, respectively:

- **Non-blocking atomic commit (denoted by NBAC)** [Gra78, Ske81] (a) (*uniform agreement*) no two processes decide differently, (b) (*termination*) every correct process eventually decides, (c) (*abort validity*) 0 can be decided only if some process proposes 0 or is faulty, and (d) (*commit validity*) 1 can be decided only if all processes propose 1. (Traditionally, the proposal values in NBAC are denoted by *yes* and *no* and the decision values by *abort* and *commit*.)
- **Weak binary agreement (denoted by WA)** [KR03] (a) (*agreement*) no two correct processes decide differently, (b) (*termination*) every correct process eventually decides, and (c) (*weak validity*) for each $v \in \{0, 1\}$, there is a failure-free run in which some process decides $v$.
- **Weak binary uniform agreement (denoted by UA)** [KR03] (a) (*uniform agreement*) no two processes decide differently, (b) (*termination*) every correct process eventually decides, and (c) (*weak validity*) for each $v \in \{0, 1\}$, there is a failure-free run in which some process decides $v$.

We study the time-complexity of these six agreement problems. For ease of presentation, we assume that $V$ always contains 0 and 1. We make the following observations which we use frequently in our lower bound proofs.

- Any uniform consensus algorithm also solves consensus because the uniform agreement property implies the agreement property.
- Any consensus, uniform consensus or non-blocking atomic commit algorithm also solves weak binary agreement. Consider any consensus or uniform consensus algorithm $A$. If all processes propose 1 and the run is failure-free, then from the validity property, all processes decide 1. Similarly, if all processes propose 0 and the run is failure-free, then all processes decide 0. Thus $A$ satisfies weak validity property, and hence, solves weak binary agreement. Now consider any non-blocking atomic commit algorithm $B$. If all processes are correct and propose 1, then from abort-validity property, all processes decide 1. If all process are correct and propose 0, then from commit-validity, all processes decide 0. Thus, $B$ satisfies weak validity property, and hence, solves weak binary agreement.

- Any uniform consensus or non-blocking atomic commit algorithm also solves weak uniform binary agreement. The argument underlying this observation is similar to that of the previous observation.

- If an algorithm $A$ solves NC, UC, NBAC or IC in any model $M1$ then $A$ solves that problem in any submodel $M2$ of $M1$. Notice that runs of $A$ in $M2$ are a subset of runs of $A$ in $M1$. Thus if, in every run in $M1$, $A$ satisfies one of the properties of the above agreement problems, then $A$ satisfies that property in every run in $M2$.

  However, the Weak validity property of WA and UA is special: it is a condition on a set of runs, whereas, other properties are conditions on a single run. Consider any algorithm $B$ that solves WA or UA in $SM_t$. Then, $B$ satisfies weak validity in $SM_t$, i.e., for each $v \in \{0, 1\}$, there is a failure-free run of $B$ in $SM_t$ in which some process decides $v$.

  Recall that, the only models that we consider in this thesis are $EM_t$, $SM_t$, and $SM1_t$ ($1 \leq t \leq n$). Thus, the submodels of $SM_t$ that we consider in this thesis are $SM1_w$ and $SM_w$ ($1 \leq w \leq t$). Notice that any failure-free run of $B$ in $SM_t$ is also a failure-free run of $A$ in any of those submodels. Thus, if $B$ has a failure-free run $r$ in $SM_t$ in which some process decides $v$, then $r$ is also a failure-free run of $B$ in any of the considered submodel of $SM_t$. Thus, if $B$ solves WA and UA in $SM_t$, then $B$ solves the problem in any of the considered submodel of $SM_t$. (The same statement does not hold for WA or UA algorithms in $EM_t$.)

## 2.3 Time complexity metrics

We now discuss some time-complexity metrics for agreement problems. Let $r$ be a run of any algorithm that solves an agreement problem in some model $M$. We say that a process $p_i$ *decides in round* $k > 0$ of a run $r$, if $p_i$ executes a subround in round $k$ that modifies the value in $dec_i$. We say that $p_i$ *decides in round 0*, if it decides initially.

Roughly speaking, a process is said to *halt* when its algorithm does not require the process to take any further steps. However, in the models we consider, a correct process executes an infinite number of rounds. So we define halting in a restricted and indirect manner (which is nevertheless sufficient for our purposes). Assume that every message has a binary tag *halted*, which is by default 0. Additionally, assume that any message that is received with *halted* tag set to 1, is ignored; i.e., all messages sent by a halted process are ignored. We say that $p_i$ *halts in round* $k$ if $p_i$ has decided in round $k$ or in a lower round, and every message sent by $p_i$ in a higher round has its *halted* tag set to 1.

We distinguish four time complexity metrics: *global decision*, *global halting*, *local decision* and *local halting*. We denote the four metrics by $gd$, $gh$, $ld$ and $lh$, respectively. Consider any run $r$ of an algorithm that solves an agreement problem.

- We say that run $r$ *globally decides* (resp. *globally halts*) in round $k$ if every correct process decides (resp. halts) in round $k$ or in a *lower* round, and some correct process decides (resp. halts) in round $k$ [FL82, DRS90, CBS04, KR03].
- We say that run $r$ *locally decides* (resp. *locally halts*) in round $k$ if every correct process decides (resp. halts) in round $k$ or in a *higher* round, and some correct process decides (resp. halts) in round $k$.

If a run $r$ globally decides at round $k$, we write $(r, gd) = k$. Similarly, the round at which $r$ globally halts, locally decides, and locally halts, are denoted by $(r, gh)$, $(r, ld)$, $(r, lh)$, respectively. Note that a local decision always occurs before a global decision (and similarly for halting). As every correct process decides before it halts, so $(r, ld) \leq (r, lh) \leq (r, gh)$, and $(r, ld) \leq (r, gd) \leq (r, gh)$.

Given a model M1, a submodel M2 of M1, an agreement problem P, and a time complexity metric T, we denote by the ordered tuple (M1, M2, P, T) the following tight bound.

**(M1, M2, P, T)** is the round number $k$ such that the following two conditions hold:

1. *(lower bound)* every algorithm that solves P in M1 has a run $r$ in M2 such that $(r, T) \geq k$, and
2. *(matching algorithm)* there is an algorithm $A$ that solves P in M1 such that, every run $r$ of $A$ in M2 has $(r, T) \leq k$.

In short, for algorithms that solve P in M1, (M1, M2, P, T) is the tight bound for achieving T in M2. The notation captures the common time-complexity tight bounds for agreement problems in round-based models, where submodel M2 denotes the set of runs (e.g., failure-free runs) for which we want to optimize the algorithms in M1. If we set M2=M1, the tuple denotes the worst-case bound in M1. Let us recall some known results on consensus (NC) and uniform consensus (UC) using our notation. (For every pair of real numbers $a \leq b$, $[a, b]$ denotes the set of integers $x$ such that $a \leq x \leq b$.)

**Theorem 1 (from [FL82, DLM82, DS83, DM90, Mer85, MT88])** $\forall t \in [0, n-2]$, $\forall f \in [0, t]$, $(SM_t,\ SM_f,\ \text{WA},\ gd) = f + 1$. Every weak binary agreement algorithm in $SM_t$ has a run in $SM_f$ in which some correct process decides in round $f+1$ or in a higher round, and there is a weak binary agreement algorithm $A$

in $SM_t$ such that, in every run of $A$ in $SM_f$, every correct process decides by round $f+1$.

**Theorem 2 (from [DRS90])** $\forall t \in [2, n-2]$, $\forall f \in [0, t-1]$, ($SM_t$, $SM_f$, WA, $gh$) $= f+2$. Every weak binary agreement algorithm in $SM_t$ has a run in $SM_f$ in which some correct process halts in round $f+2$ or in a higher round, and there is a weak binary agreement algorithm $A$ in $SM_t$ such that, in every run of $A$ in $SM_f$, every correct process halts by round $f+2$.

**Theorem 3 (from [CBS04, KR03])** $\forall t \in [2, n-1]$, $\forall f \in [0, t-2]$, ($SM_t$, $SM_f$, UA, $gd$) $= f+2$. Every weak binary uniform agreement algorithm in $SM_t$ has a run in $SM_f$ in which some correct process decides in round $f+2$ or in a higher round, and there is a weak binary uniform agreement algorithm $A$ in $SM_t$ such that, in every run of $A$ in $SM_f$, every correct process decides by round $f+2$.

## 2.4 Configurations and full-information algorithms

We introduce few additional notions on runs of an agreement algorithm. Fix any algorithm $A$ that solves an agreement problem in $EM$ or any of its submodel.

**Configurations.** A configuration captures the state of the system at the end of a round. For $k > 0$, a *round $k$ configuration of a run $r$* is an ordered $n$-tuple such that,

- if $p_j$ completes round $k$, then element $j$ of the tuple contains the state of $p_j$ after executing the computation subround of round $k$, else
- element $j$ of the tuple contains a special symbol $\top$.

We ignore the messages in *mset* at the end of round $k$, because from the communication closed round property of $EM$, we know that, those messages will be never received. A round 0 configuration of $r$ is an ordered $n$-tuple where element $j$ equals the proposal value of $p_j$. $C$ is a round $k$ configuration of $A$ if $C$ is the round $k$ configuration of some run of $A$.

**Extensions.** A run $r$ is an *extension* of a round $k$ configuration $C$, if $C$ is the round $k$ configuration of $r$. A round $k_1$ configuration $C'$ is an *extension* of $C$ if $k \leq k_1$ and there is a run $r$ such that the round $k$ configuration of $r$ is $C$ and round $k_1$ configuration of $r$ is $C'$. In a round $k$ configuration $C$, we say that $p_i$ is *alive* (respectively, *decided* or *halted*) in $C$ if $p_i$ completes round $k$ (respectively, decides or halts by round $k$) in $C$. If $p_i$ is not alive in $C$, we say that $p_i$ has *crashed* in $C$.

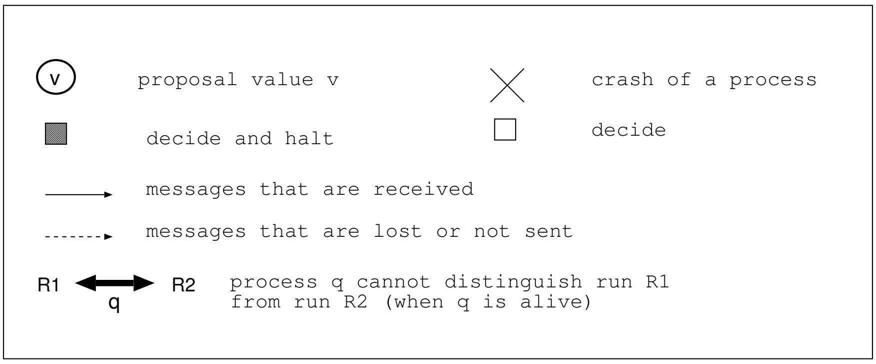


Figure 2.2: Common notations used in the diagrams

**Full-information algorithms.** In a full-information algorithm, every message includes the entire state of the sender, and the state of a process includes all previous states of the process (which in turn includes all received messages). *To strengthen our lower bound results, we always consider full-information algorithms in our lower bound proofs.*

**Valency [KR03].** Consider any full-information algorithm. For each run $r$, we denote by $val(r)$ the decision value of any correct process in $r$. (This definition is unambiguous because, in every agreement problem we consider, no two correct processes decide differently.) For any round $k$ configuration $C$, $r(C)$ denotes a run that is an extension of $C$ such that, every process that is alive is $C$ is correct in $r(C)$, and in every round higher than $k$, no message is lost (i.e., correct processes receive messages from all correct processes). Note that the run $r(C)$ is unambiguously defined by these conditions because, (1) as $A$ is a full-information algorithm, $C$ completely defines the run until round $k$, and (2) the message exchange pattern is completely defined from round $k+1$. We define $val(C)$ as $val(r(C))$.

**Notations in diagrams.** Figure 2.2 depicts the common notations that we use in the diagrams of this thesis. For clarity of presentation, in the runs presented in our diagrams, we only indicate the messages that assist in distinguishing the constructed runs from each other.

# Chapter 3

# Background

## Part B — Related Results

In this chapter we revisit some related results that are useful in showing our lower bounds. In particular, we frequently use a variant of Lemma 2.3 of [KR03] (Lemma 6 in this thesis) that captures the level of indistinguishability among runs that remain after a given number of rounds. As in [KR03], we use the layering technique, introduced in [MR02], to prove this lemma. (In the following, we point out when our notions differ from those in [KR03].)

## 3.1 The layering technique

Consider any weak binary agreement (WA) algorithm A in a synchronous model where at most $t$ processes may crash and at most one process crashes in each round ($SM1_t$). In any run of $SM1_t$, the round $k$ configuration is completely determined by the initial configuration and the failure pattern: the failure pattern for a run in $SM1_t$ specifies, for each round $k$, the process $p_i$ that crashes in round $k$ (or that no process crashes), and the set of processes which did not receive the round $k$ message from $p_i$.

**Extensions in $SM1_t$.** We denote an extension by one round, of a round $k$ configuration $C$, as follows: for $i \in [1, n]$ and $S \subseteq \Pi$, $C.(i, S)$ denotes the round $k+1$ configuration reached by crashing $p_i$ in (the send subround of) round $k+1$ in such a way that a process $p_j$ *does not* receive a round $k+1$ message from $p_i$ if and only if at least one of the following holds:

(1) $p_j = p_i$, (2) $p_j$ is crashed in $C$, or (3) $p_j \in S$.

Configuration $C.(0, \emptyset)$ denotes the one round extension of $C$ in which no process crashes. Clearly, $C.(i, S)$ for $i \in [1, n]$ and $S \subseteq \Pi$, is a possible extension of $C$ in $SM1_t$ if at most $t-1$ processes have crashed in $C$ and $p_i$ is alive in $C$ — we then say that $(i, S)$ is *applicable* to $C$. Obviously, $(0, \emptyset)$ is applicable to to any configuration.

**Layers.** A layer $L(C)$ is the set of configurations defined as $\{C.(i, S) \mid i \in [0, n], S \subseteq \Pi, (i, S) \text{ is applicable to } C\}$. (In other words, if $C$ is a round $k$ configuration, then $L(C)$ is the set of all round $k+1$ configurations that extend $C$ in $SM1_t$.) For a set of round $k$ configurations $SC$, $L(SC)$ is a set of round $k+1$ configurations defined as $\cup_{C \in SC} L(C)$. $L^k(SC)$ is recursively defined as follows: $L^0(SC) = SC$ and for $k > 0$, $L^k(SC) = L(L^{k-1}(SC))$. (In other words, if $SC$ is a set of round $l$ configurations, then $L^k(SC)$ is the set of all round $(l+k)$ configurations that extend any configuration in $SC$.)

**Similar configurations.** Two configurations $C$ and $D$ at the same round are said to be *similar*, denoted $C \sim D$, if they are identical or they differ at exactly one process. A pair of configurations $C$ and $D$ is *similarity connected* if there are configurations $C = C_0, \ldots, C_m = D$ such that $C_i \sim C_{i+1}$ for every $i$ in $[0, m-1]$. A set of configurations $SC$ is *similarity connected* if, every pair of configurations in $SC$ is similarly connected. (Our definition of similarity does not include the second property of the original definition in [KR03]: there exists a process that is alive in both $C$ and $D$, and has identical states in $C$ and $D$. When this property is required in our lower bound proofs, we derive it directly from our assumptions on $t$ and $n$.)

We now revisit Lemma 2.3 of [KR03]. Roughly speaking, this lemma says that, in $SM1_t$, if we start with a similarity connected set (of configurations) $SC$, we can keep the set of extensions from $SC$ similarity connected, provided we can crash one process in every round we extend.

**Lemma 4** *Let* $\mathrm{SC} = L^0(\mathrm{SC})$ *be a similarity connected set of configurations such that, in any configuration of* SC, *no process has crashed, then for all* $k \in [1, t]$, $L^k(\mathrm{SC})$ *is a similarity connected set of configurations in which no more than k processes are crashed in any configuration.*

**Proof**: The proof is by induction on the round number $k$. The base case $k = 0$ is immediate. For the inductive step, assume that $L^{k-1}(SC)$ is similarity connected and in every configuration of $L^{k-1}(SC)$ at most $k-1$ processes have crashed. Notice that, from the definition of $SM1_t$, in every extension by one round which is applicable to a configuration in $L^{k-1}(SC)$, at most one more process can crash. Therefore, in every configuration in $L^k(SC)$, at most $k$ processes have crashed. We now show that $L^k(SC)$ is similarity connected through the following three claims.

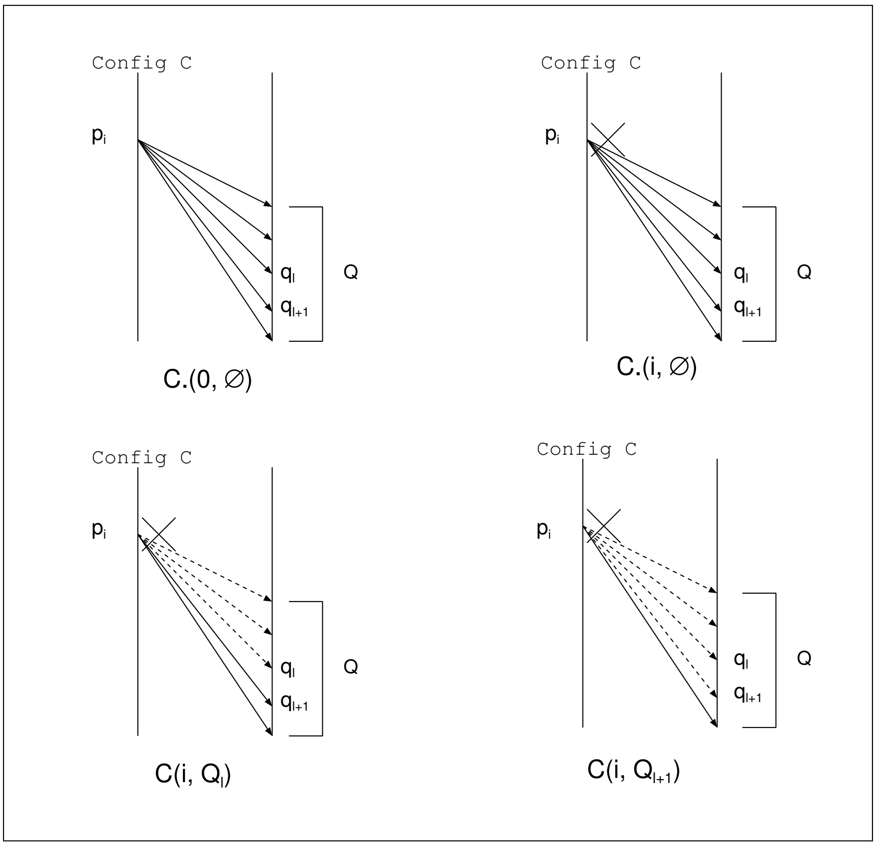


Figure 3.1: Round $k$ of Case 1, Lemma 4

1. F*or any configuration* $C \in L^{k-1}(\mathrm{SC})$, $L(C)$ *is similarity connected.* Consider any configuration in $L(C)$ that is different from $C.(0, \emptyset)$, say $C1 = C.(i, Q)$, where $Q \subseteq \Pi$, and $p_i$ is alive in $C$. We claim that $C1$ and $C.(0, \emptyset)$ are similarity connected. Since $C1$ is arbitrarily selected from $L(C)$, our claim implies that every configuration in $L(C)$ is similarity connected to $C.(0, \emptyset)$, and hence, $L(C)$ is similarity connected.

   Now we prove our claim. (Figure 3.1 depicts the runs with the relevant messages.) $C.(i, \emptyset) \sim C.(0, \emptyset)$ since the configurations differ only at $p_i$. If $Q = \emptyset$ then we are done. Hence, let $Q = \{q_1, q_2, \ldots, q_m\}$. For every $l$ in $[1, m]$, let $Q_l = \{q_1, \ldots, q_l\}$, and $Q_0 = \emptyset$. For every $l$ in $[0, m-1]$, $C.(i, Q_l) \sim C.(i, Q_{l+1})$ because the two configurations differ only at $q_{l+1}$. Thus, $C(i, \emptyset) = C.(i, Q_0)$ and $C1 = C.(i, Q_m)$ are similarly connected.

2. *For any pair of configurations* $C, D \in L^{k-1}(\mathrm{SC})$, *if* $C \sim D$ *then* $L(C) \cup L(D)$ *is similarity connected.* If $C$ and $D$ are identical then the claim immediately follows from claim 1. So consider the case where $C$ and $D$ are distinct. As

$C \sim D$, there is a process $p_i$ such that $C$ and $D$ are different only at $p_i$. Then, configurations $C.(i, \Pi)$ and $D.(i, \Pi)$ are identical because no process receives message from $p_i$ in round $k$, and $p_i$ has crashed. Hence, $C.(i, \Pi) \sim D.(i, \Pi)$. We know from claim 1 that $L(C)$ and $L(D)$ are each similarity connected. Thus every configuration in $L(C)$ is similarly connected to $C.(i, \Pi)$ and every configuration in $L(D)$ is similarity connected to $D.(i, \Pi)$. As, $C.(i, \Pi) \sim D.(i, \Pi)$, so every configuration in $L(C)$ is similarity connected to every configuration in $L(D)$. Thus, $L(C) \cup L(D)$ is similarly connected.

3. $L^k$(SC) *is similarity connected.* Consider any pair of configurations $C', D' \in L^k(SC)$. Thus, there are configurations $C, D \in L^{k-1}(SC)$ such that $C' \in L(C)$ and $D' \in L(D)$. As $L^{k-1}(SC)$ is similarity connected, so there is a chain of configurations $C = C_0, ..., C_m = D$ such that, for every $l \in [0, m-1]$, $C_l \sim C_{l+1}$. Thus, from claim 2, $L(C_l) \cup L(C_{l+1})$ is similarity connected. A simple induction using claim 2 shows that $L(C_1) \cup ... \cup L(C_m)$ is similarly connected. Thus $C' \in L(C = C_0)$ is similarity connected to $D' \in L(D = C_m)$. As $C'$ and $D'$ are arbitrarily selected from $L^k(SC)$, $L^k(SC)$ is similarity connected.

$\square$

**Remark.** Our lemma is a simple generalization of Lemma 2.3 of [KR03]. The statement of our lemma looks similar to that of [KR03], but the model considered in [KR03], say $SM1'_t$, is actually a submodel of the model $SM1_t$ that we consider here. In $SM1'_t$, $C.(i, S)$ is an extension of $C$ only if $S$ is a prefix of $\Pi = \{p_1, ..., p_n\}$, whereas, in $SM1_t$, $S$ can be any subset of $\Pi$. Thus $L^k(SC)$ in $SM1'_t$ is a subset of $L^k(SC)$ in $SM1_t$, and $L^k(SC)$ being connected in $SM1_t$ implies that $L^k(SC)$ is connected in $SM1'_t$.

Informally, the next lemma says that, for any WA algorithm in $SM1_t$ and any $f \in [0, t]$, there are two $f$ round configurations which are almost identical (differ at only one process) but have different decision values in failure-free extensions.

**Lemma 5** *Let* $t \in [1, n-1]$. *Consider any WA algorithm A in* $\mathrm{SM1}_t$. *For every* $f \in [0, t]$, *there are two runs of A in* $\mathrm{SM1}_t$ *such that their round* $f$ *configurations,* $y$ *and* $y'$, *satisfy the following: (1) at most* $f$ *processes have crashed in each configuration, (2) the configurations differ at exactly one process, and (3)* $val(y) = 0$, *whereas* $val(y') = 1$.

**Proof**: Consider any initial configuration $C'$ of algorithm A in model $SM1_t$. Let $C$ be the configuration in which all processes propose 0. Consider the following $n-1$ (not necessarily distinct) initial configurations: for every $i \in [1, n-1]$, in configuration $C_i$, processes $p_1, ..., p_i$ propose the same value as in $C'$, and the remaining processes propose 0. Notice that, for every $i \in [1, n-2]$, $C_i$ and $C_{i+1}$ may differ only at $p_{i+1}$.

Furthermore, $C_1$ and $C$ may differ only at $p_1$, and $C'$ and $C_{n-1}$ may differ only at $p_n$. Thus $C$ and $C'$ are connected through a chain of configurations, such that, any two adjacent configurations in the chain are similar. Since $C'$ was arbitrarily selected, the set of initial configurations of A in $SM1_t$ is similarity connected. Let $I$ be the set of initial configurations of A in $SM1_t$. From the definition of $L^f(I)$, it follows that $L^f(I)$ is the set of round $f$ configurations of A in $SM1_t$. Then from Lemma 4, it follows that the set of round $f$ configurations of A in $SM1_t$ is similarly connected.

Consider any failure-free run $r0$ of algorithm A in which correct processes decide 0. (From the validity property of WA, such a run of A exists.) We denote by $z$, the round $f$ configuration of $r0$. Similarly, consider any failure-free run $r1$ of A in which correct processes decide 1. We denote by $z'$, the round $f$ configuration of $r1$. Obviously, $val(z) = 0$ and $val(z') = 1$.

As the set of round $f$ configurations of A in $SM1_t$ is similarly connected, so there are some round $f$ configurations of $A$ in $SM1_t$, $z = y_0, y_1, \ldots, y_m = z'$, such that $y_j \sim y_{j+1}$ for every $j$ in $[0, m-1]$. Clearly, there is some $y_i \in \{y_0, \ldots, y_{m-1}\}$ such that, $val(y_0) = \ldots = val(y_i) \neq val(y_{i+1})$. (Otherwise, $val(z) = val(y_0) = val(y_1) = \ldots = val(y_m) = val(z')$; a contradiction.)

As $val(y_i) = val(y_0)$ and $z = y_0$, so $val(y_i) = 0$. Therefore, $val(y_{i+1}) = 1$. Since both $y_i$ and $y_{i+1}$ are round $f$ configurations in $SM1_t$, at most $f$ processes have crashed in each configuration. As $y_i \sim y_{i+1}$, the two configurations are either identical or differ at exactly one process. Since $val(y_i) \neq val(y_{i+1})$, the configurations cannot be identical, i.e., they differ at exactly one process. □

As any NC, UC, NBAC, and UA algorithm is also a WA algorithm, and $SM1_t$ is a submodel of $SM_t$, so the following lemma immediately follows from Lemma 5.

**Lemma 6** *Let $t \in [1, n-1]$ and $V = \{0, 1\}$. Consider any algorithm $A$ in* $\mathrm{SM}_t$ *that solves NC, UC, NBAC, WA or UA. For every $f \in [0, t]$, there are two runs of $A$ in* $\mathrm{SM}_t$ *such that their round $f$ configurations, $y$ and $y'$, satisfy the following: (1) at most $f$ processes have crashed in each configuration, (2) the configurations differ at exactly one process, and (3) $val(y) = 0$, whereas $val(y') = 1$.*

We will use Lemma 6 frequently as the starting point for our lower bound proofs. Informally, an agreement algorithm divides the set of runs into two subsets (e.g., 0-deciding runs and 1-deciding runs), such that every correct process can distinguish a run of one subset from any run of the other subset. Lemma 6 captures the level of indistinguishability between the runs from different subsets that remains after $f$ rounds.

## 3.2 A warm-up example

**Early global halting lower bound.** The lower bound part of Theorems 1, 2 and 3 can be derived from Lemma 6. We revisit Theorems 1 and 3 in Chapter 4. As an example, we now show how to derive the lower bound of Theorem 2: every weak binary agreement algorithm in $SM_t$ has a run in $SM_f$ in which some correct process halts in round $f + 2$ or in a higher round.

**Theorem 2 (from [DRS90])** (lower bound) $\forall t \in [2, n-2]$, $\forall f \in [0, t-1]$, ($SM_t$, $SM_f$, WA, $gh$) $\geq f + 2$.

**Proof**: Suppose by contradiction that there is a WA algorithm $A$ in $SM_t$ and an integer $f$ in $[0, t-1]$ such that, in every run of $A$ with at most $f$ crashes, all correct processes halt by round $f + 1$.

It follows from Lemma 6 that, there are two runs of $A$ in $SM_t$ such that their round $f$ configurations, $y$ and $y'$, satisfy the following: (1) at most $f$ processes have crashed in each configuration, (2) the configurations differ at exactly one process, say $p_i$, and (3) $val(y) = 0$ and $val(y') = 1$. Let $z$ and $z'$ denote the configurations at the end of round $f + 1$ of $r(y)$ and $r(y')$, respectively.

As $r(y)$ is a run with at most $f$ crashes, it follows from our initial assumption on $A$ that all correct processes have decided $val(y) = 0$ and halted by round $f+1$. As $z$ is the round $f+1$ configuration of $r(y)$, and all correct processes in $r(y)$ are alive in $z$, it follows that, in $z$, all alive processes have decided $val(y) = 0$ and halted. Similarly, in $z'$, all alive processes have decided $val(y') = 1$ and halted. Since $f \leq n-3$, there are two processes $q_1$ and $q_2$ that are distinct from $p_i$, and which have halted in both $z$ and $z'$. There are two cases to consider.

*Case 1.* Process $p_i$ is alive in $y$ and $y'$. Consider the following two runs of $A$. (Figure 3.2 depicts the runs with the relevant messages.)

**R1** is a run such that (1) the round $f$ configuration is $y$, (2) $p_i$ crashes in round $f+1$ such that only $q_1$ receives the message from $p_i$, (3) no process distinct from $p_i$ crashes after round $f$. Notice that $q_1$ cannot distinguish the round $f + 1$ configuration of $R1$ from $z$, and therefore, $q_1$ decides 0 and halts at the end of round $f + 1$ in $R1$. By agreement, every correct process decides 0 in $R1$.

**R2** is a run such that (1) the round $f$ configuration is $y'$, (2) $p_i$ crashes in round $f+1$ such that only $q_1$ receives the message from $p_i$, (3) no process distinct from $p_i$ crashes after round $f$. Notice that $q_1$ cannot distinguish the round $f + 1$ configuration of $R2$ from $z'$, and therefore, $q_1$ decides 1 and halts at the end of round $f + 1$ in $R2$. By agreement, every correct process decides 1 in $R1$. However, notice that

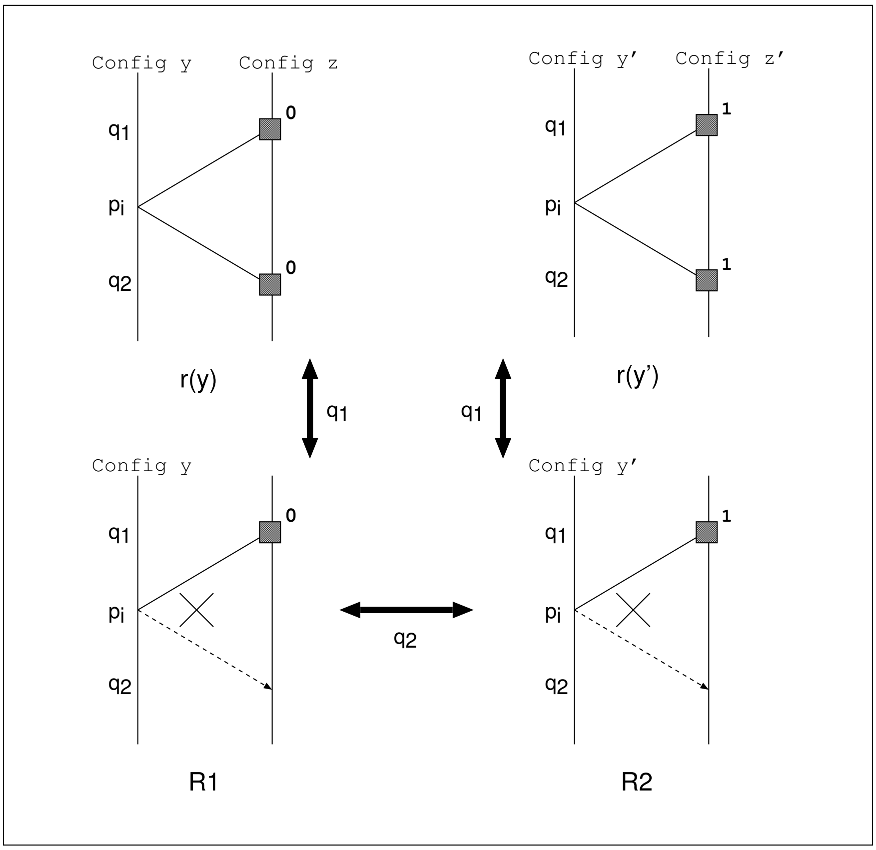


Figure 3.2: Round $f + 1$ of Case 1, Theorem 2

$q_2$ cannot distinguish $R1$ from $R2$: at the end of round $f$, the two runs are different only at $p_i$, only $q_1$ receives a message from $p_i$ in round $f + 1$, and $q_1$ halts in round $f + 1$. Thus, $q_2$ is correct and decides the same value in both runs; a contradiction.

*Case 2.* (See Figure 3.3.) Process $p_i$ has crashed in either $y$ or $y'$. (Process $p_i$ has not crashed in both $y$ and $y'$ because $p_i$ has different states in $y$ and $y'$.) Without loss of generality, we can assume that $p_i$ has crashed in $y'$, and hence, $p_i$ is alive in $y$. Let us reconsider run $R1$ in this setting. Process $q_1$ receives round $f+1$ messages from $p_i$, decides 0 and halts. Whereas process $q_2$ does not receive any message, and hence, cannot distinguish round $f+1$ configuration of $R1$ from $z'$. (Recall that, now in $y'$, process $p_i$ is crashed.) Thus, as in $z'$, $q_2$ decides 1, a violation of agreement. $\square$

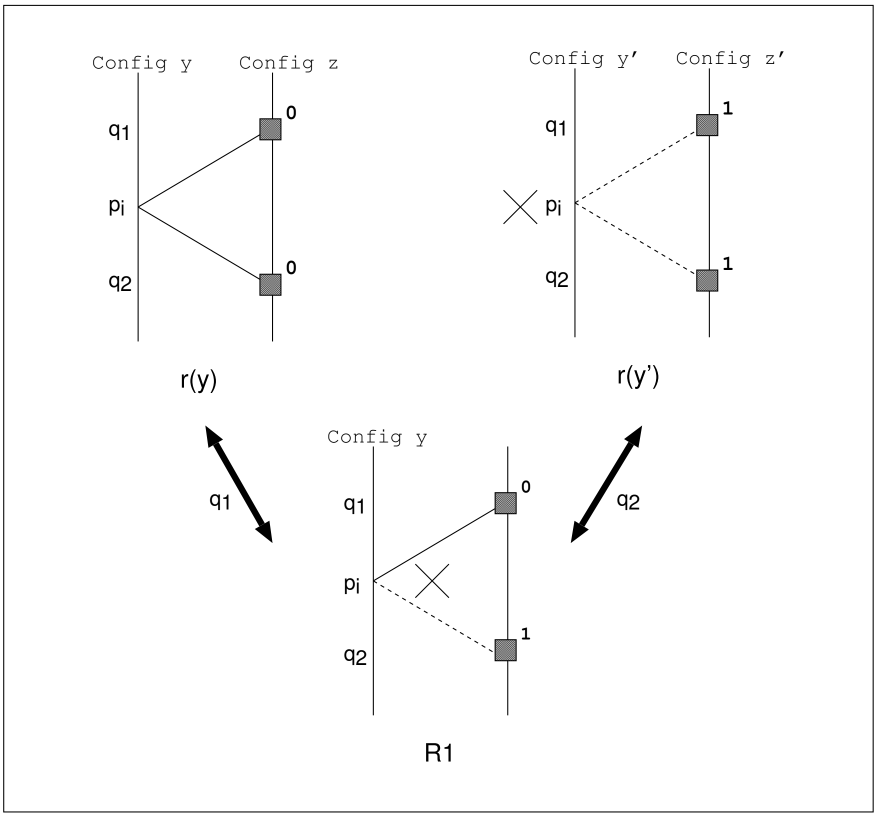


Figure 3.3: Round $f + 1$ of Case 2, Theorem 2

**Byzantine agreement – an informal discussion.** Roughly speaking, in the synchronous model with Byzantine failures (called Byzantine model, hereafter), the processes may fail by behaving arbitrarily [LSP82], i.e., a faulty process might replace its assigned algorithm with any arbitrary algorithm. Several algorithms solve some variant of WA in this setting, sometimes called Byzantine agreement. In the Byzantine model, just as in the crash-stop one, it is possible to match the $f + 2$ round lower bound for early global halting [DRS90]. However, the early global decision lower bound has not been considered in this setting. From Theorem 1 and Theorem 3, we know that the tight bound for early global decision in the crash-stop model is $f + 1$ for WA and $f + 2$ for UA. However, UA is not considered in a Byzantine model because we cannot impose any requirement on the decision value of the processes that behave arbitrarily. In the Byzantine model, one might wonder whether we can match the $f + 1$ global decision lower bound of WA.

A simple variant of the proof of Theorem 2 above shows that, in the Byzantine model, the lower bound for a early global decision of WA can be improved to $f + 2$, the same as that for global halting. Suppose that all correct processes decide by round $f+1$ in every run in which $f$ processes fail. Consider configurations $z$ and $z'$, in the above proof. Suppose that, if $p_i$ is not Byzantine faulty, and it is alive in both $y$ and $y'$, then $p_i$ sends in round $f+1$, message $m$ to $q_1$ in $z$, and message $m'$ to $q_2$ in $z'$. (No process distinct from $p_i$ is Byzantine faulty in the runs we consider.) Consider a run $R$ in which $p_i$ is Byzantine faulty and sends $m$ to $q_1$ and $m'$ to $q_2$, and no other process fails, thereafter. Then, in $R$, $q_1$ decides 0 (as in $z$), and $q_2$ decides 1 (as in $z'$); a violation of consensus agreement. The proof for the case where $p_i$ has crashed in either $y$ or $y'$ does not need to be modified. Notice that our proof does not need processes to halt in round $f+1$, because $q_1$ and $q_2$ decide differently (i.e., disagreement occurs) at round $f+1$ itself, and not at some later round. Thus, the tight bound for a WA early global decision is lower in the crash-stop model than in the Byzantine model. (We do not consider Byzantine failure elsewhere in this work.)

## 3.3 Some useful results in the eventually synchronous model

In this section, we recall three well-known results in the eventually synchronous model $EM_t$. (Recall that we always assume $n \geq 3$.)

**Lemma 7** *(from [Gue95]) If* $t \geq 1$*, then there is no NBAC algorithm in* $\mathrm{EM}_t$*.*

**Proof**: Suppose by contradiction that $t \geq 1$ and there is a NBAC algorithm $A$ in $EM_t$. Consider two runs of $A$, run $R_0$ and run $R_1$ such that (1) $p_1$ crashes initially, (2) all other processes are correct, (3) $GSR = 1$, and (4) $p_1$ proposes 0 in $R_0$ and 1 in $R_1$, and other processes propose 1 in both runs.

As $p_1$ crashes before sending any message in both $R_0$ and $R_1$, no process distinct from $p_1$, say $p_2$, can distinguish $R_0$ from $R_1$. From commit-validity, all correct processes decide 0 in in $R_0$, and hence, in $R_1$. Suppose $p_2$ decides 0 in $R_1$ in round $k'$.

Consider run $R_2$ such that (1) all processes are correct and propose 1, (2) until round $k'$, every messages that is sent is received in the same round in which it is sent, except that the messages sent by $p_1$ in the first $k'$ rounds to other processes are lost, and (3) $GSR(R_2) = k' + 1$. Note that, before round $k'+1$, no process distinct from $p_1$ can distinguish $R_2$ from $R_1$. Thus, $p_2$ decides 0 in $R_2$. However, as all processes are correct and propose 1, from the abort-validity property, correct processes must decide 1 in $R_2$; a contradiction. □

**Lemma 8** *(from [Gue95]) If* $t \geq 1$*, then in* $\mathrm{EM}_t$*, every NC algorithm is also a UC algorithm.*

**Proof**: Suppose by contradiction that there is a NC algorithm $A$ in $EM_t$ that does not satisfy the uniform agreement property. Thus there is a run $r$ of $A$ such that some process $p_i$ decides $v$, another process $p_j$ decides $v' \neq v$, and at least one process from $\{p_i, p_j\}$ crashes in $r$. Let $p_i$ and $p_j$ decide at rounds $k_0$ and $k_1$, respectively. Without loss of generality, assume that $k_0 \leq k_1$. There are two cases to consider:

1. Process $p_i$ crashes in some round $k_2 \in [k_0 + 1, k_1]$. Consider another run $r'$ such that (1) $r'$ is identical to $r$ until round $k_1$ except that $p_i$ is correct in $r'$, and in rounds $[k_2, k_1]$, all messages sent by $p_i$ to processes distinct from $p_i$ are lost, but $p_i$ receives all the messages sent to itself, and (2) no process crashes after round $k_1$, and (3) $GSR(r') = k_1 + 1$.

   Clearly, the round $k_0$ configuration in $r$ and $r'$ are identical, and thus, $p_i$ decides $v$ in $r'$. Until round $k_1$, process $p_j$ cannot distinguish $r$ from $r'$, because in both runs, no round $k''$ messages from $p_i$, such that $k'' \in [k_2, k_1]$, is received by any process distinct from $p_i$. Thus, in round $k_1$ of $r'$, $p_j$ decides $v'$. Since both $p_i$ and $p_j$ are correct in $r'$, the run violates agreement.

2. Process $p_i$ crashes at some round higher than $k_1$ or does not crash in run $r$. Consider another run $r''$ such that (1) $r''$ is identical to $r$ until round $k_1$, (2) no process crashes after round $k_1$, and (3) $GSR(r'') = k_1 + 1$.

   As $r$ and $r''$ are identical until round $k_1$, in $r''$, $p_i$ decides $v$ and $p_j$ decides $v'$. Since both $p_i$ and $p_j$ are correct in $r''$, the run violates agreement.

□

**Lemma 9** *(from [DLS88, CT96]) If $t \geq n/2$, then there is no UC algorithm in* EM$_t$.

**Proof**: Suppose by contradiction that $t \geq n/2$ and there is a UC algorithm $A$ in $EM_t$. Consider the set $P$ that contains every process $p_i$ in $\Pi$ such that $i \in [1, n/2]$, and $Q = \Pi \backslash P$. As $t$ is an integer, so $t \geq \lceil \frac{n}{2} \rceil$, and $|P|, |Q| \leq \lceil \frac{n}{2} \rceil \leq t$.

Consider a run $R_P$ of $A$ such that (1) all processes propose 0, (2) processes in $Q$ crash initially, (3) all processes in $P$ are correct, and (4) $GSR(R_P) = 1$. From the validity property, all processes in $P$ decide 0. Suppose run $R_P$ globally decides at round $k_P$. Consider another run $R_Q$ of $A$ such that (1) all processes propose 1, (2) processes in $P$ crash initially, (3) all processes in $Q$ are correct, and (4) $GSR(R_Q) = 1$. From the validity property, all processes in $Q$ decide 1. Suppose run $R_Q$ globally decides at round $k_Q$. Let $k' = max\{k_P, k_Q\}$.

We now construct another run $R_{PQ}$ as follows: (1) all processes are correct, (2) in the first $k'$ rounds, any message sent from a process in $P$ to a process in $Q$, and from a process in $Q$ to a process in $P$, is lost, all other messages are received, and (3) $GSR(R_{PQ}) = k' + 1$.

It is easy to see that, in the first $k'$ rounds, the processes in $P$ cannot distinguish $R_{PQ}$ from $R_P$, and the processes in $Q$ cannot distinguish $R_{PQ}$ from $R_Q$. Thus, in $R_{PQ}$, the processes in P decide 0 and the processes in $Q$ decide 1; a violation of uniform agreement. □

# Chapter 4

# Tight Bounds in the Synchronous Model

In this chapter we investigate local decision bounds for agreement problems in the synchronous model. Additionally, we show that in some sense local decision and global decision tight bounds are incompatible for consensus, and we revisit the global decision lower bounds for consensus and uniform consensus (Theorems 1 and 3).

## 4.1 Consensus

In this section, we give two lower bounds for weak binary agreement (WA) in the synchronous model. Since any consensus (NC) algorithm solves weak binary agreement, the lower bounds immediately apply to consensus.

### Local decision

The following lemma says that any WA algorithm in $SM_t$ has a run in $SM_f$ (i.e., a run with at most $f$ crashes) in which every correct process decides in round $f$ or in a higher round.

**Lemma 10** $\forall t \in [1, n-1]$, $\forall f \in [0, t]$, *(*$\mathrm{SM}_t$, $\mathrm{SM}_f$, *WA, ld)* $\geq f$.

**Proof**: Suppose by contradiction that there is an WA algorithm $A$ in $SM_t$ and an integer $f$ in $[0, t]$ such that, in every run of $A$ with at most $f$ crashes, some correct process decides by round $f-1$. Notice that the contradiction is immediate for the case $f = 0$: no process can decide by round $-1$. So we consider the case $f \in [1, t]$. (Also recall that we have defined the notion of deciding in round 0, as deciding in the initialization subround of round 1.)

It follows from Lemma 6 that, there are two runs of $A$ in $SM_t$ such that their round $f-1$ configurations, $y$ and $y'$, satisfy the following: (1) at most $f-1$ processes have crashed in each configuration, (2) the configurations differ at exactly one process, say $p_i$, and (3) $val(y) = 0$ and $val(y') = 1$.

As $r(y)$ is a run with at most $f-1$ crashes, it follows from our assumption on $A$ that, in $r(y)$, there is a correct process $q_1$ that has decided $val(y) = 0$ by round $f-1$. As all correct processes in $r(y)$ are alive in $y$, it follows that, in $y$, $q_1$ is alive and has decided $val(y) = 0$.

We now show that no alive process distinct from $p_i$ has decided in $y$ (which implies that $p_i = q_1$). Suppose by contradiction that some alive process distinct from $p_i$, say $q_2$, has decided in $y$. Since $q_2$ is alive in $y$, it is correct in $r(y)$, and hence, $q_2$ decides 0 in $r(y)$. Thus $q_2$ has decided 0 in $y$. As $y$ and $y'$ differ only at $p_i$ and $p_i$ is distinct from $q_2$, so $q_2$ is alive and has decides 0 in $y'$. Thus, in $r(y')$, $q_2$ is a correct process and decides 0. However, every correct process in $r(y')$ decides $val(y') = 1$; a contradiction.

Thus, $p_i$ is the only alive process that has decided in $y$. Consider any run $r'$ that extends $y$ and in which only process $p_i$ crashes after round $f-1$. At most $f$ processes crash in $r'$. At the end of round $f-1$ in $r'$, the only alive process which has decided is $p_i$, but $p_i$ is a faulty process in $r'$. Thus, $r'$ is a run with at most $f$ crashes in which no correct process decides by round $f-1$; a contradiction. □

## C-decision

The above lemma gives a lower bound on the number of rounds required for *at least one* correct process to decide (local decision). The global decision lower bound from Theorem 1 specifies the number of rounds required for *all* correct processes to decide. It is natural to investigate the number of rounds required when we consider an intermediate time complexity metric for the runs.

For every $c$ in $[1, n]$, we say that a run $r$ of an agreement algorithm *c-decides* in round $k$, and we write $(r, d_c) = k$, if at least $c$ correct processes decide by round $k$ and less than $c$ correct processes decide before round $k$. Using this notation, in a run with $f$ crashes, the local decision is 1-decision, and the global decision is $(n-f)$-decision.

In the following lemma we state that any WA algorithm in $SM_t$ has a run in $SM_f$ (i.e., a run with at most $f$ crashes) in which *at most one* correct process decides in round $f$ or in a *lower* round. In other words, the number of rounds needed for $c$-decision, when $c \geq 2$, is $f+1$. (Following this terminology, Lemma 10 states that the number of rounds needed for a 1-decision is $f$, and Theorem 1 states that the number of rounds needed for a $(n-f)$ decision is $f+1$.)

**Lemma 11** $\forall t \in [1, n-1]$, $\forall f \in [0, t]$, $\forall c \in [2, n-f]$, *(*$\text{SM}_t$*,* $\text{SM}_f$*, WA,* $d_c$*)* $\geq f+1$.

**Proof**: It is sufficient to show that ($SM_t$, $SM_f$, WA, $d_2$) $\geq f+1$. Suppose by contradiction that there is a WA algorithm $A$ in $SM_t$ and an integer $f$ in $[0, t]$ such that, in every run of $A$ with at most $f$ crashes, there are at least two correct processes which decide by round $f$.

It follows from Lemma 6 that, there are two runs of $A$ in $SM_t$ such that their round $f$ configurations, $y$ and $y'$, satisfy the following: (1) at most $f$ processes have crashed in each configuration, (2) the configurations differ at exactly one process, say $p_i$, and (3) $val(y) = 0$ and $val(y') = 1$.

From our initial assumption about algorithm $A$, it follows that (1) in $y$, there are two alive processes which have decided 0, and (2) in $y'$, there are two alive processes which have decided 1. As $y$ and $y'$ differ at exactly one process and there are two alive processes in $y'$ that have decided 1, it follows that, in $y$, there is an alive process, say $p_j$, which has decided 1. Thus, $p_j$ is correct and decides 1 in $r(y)$. However, every correct process in $r(y)$ decides $val(y) = 0$; a contradiction. □

**Remarks.** Lemma 11 is a generalization of the lower bound of Theorem 1: in a run with at most $f$ crashes, global decision is $(n-f)$-decision.

It is important to notice the special case $f = t = n-1$. The $f+1$ round lower bound of Theorem 1 does not hold when $f = t = n-1$: in this case, we can design a consensus algorithm that globally decides in $f = t = n-1$ rounds. At first glance, Lemma 11 seems to hold for $f = t = n-1$, and that would imply that a global decision requires $f+1$ in this case. However, this observation is flawed because, when $f = t = n-1$, the allowed set of values for $c$ (in Lemma 11) is $[2, n-f] = \emptyset$.

### Incompatibility

It is easy to design a consensus algorithm that matches either the early local decision lower bound of Lemma 10 or the early global decision lower bound of Theorem 1. In the following lemma, we show that, perhaps surprisingly, no consensus algorithm can match both the early local decision and the early global decision lower bounds, even for two consecutive values of $f$.

**Lemma 12** $\forall t \in [1, n-2]$, $\forall f \in [0, t-1]$, *there is no WA algorithm in* $\text{SM}_t$ *that satisfies the following two conditions:*

*(a) in every run with at most $f$ crashes, every correct process decides by round $f+1$, and*

*(b) in every run with at most $f+1$ crashes, some correct process decides by round $f+1$.*

(Remarks: Condition (a) is for matching the global decision lower bound for $f$ crashes, and condition (b) is for matching the local decision lower bound for $f+1$ crashes. Note that, we do not consider the case $f = t$, because when $f = t$, (a) implies (b), as there is no run in $SM_t$ with $t+1$ crashes.)

**Proof**: Suppose by contradiction that there is a WA algorithm $A$ and an integer $f$ in $[0, t-1]$ such that (a) by round $f+1$ of every run with at most $f$ crashes, every correct process decides, and (b) by round $f+1$ of every run with at most $f+1$ crashes, some correct process decides.

It follows from Lemma 6 that, at the end of round $f$, there are two configurations $y_0$ and $y_1$ such that (a) at most $f$ processes have crashed in each configuration, (b) the configurations differ at exactly one process, say $p_i$, and (c) $val(y_0) = 0$ and $val(y_1) = 1$.

Consider run $r(y_0)$. Obviously, $r(y_0)$ is a run with at most $f$ crashes, and from our initial assumption about $A$, every correct process decides $val(y_0) = 0$ at the end of round $f+1$. Similarly, in run $r(y_1)$, every correct process decides $val(y_1) = 1$ at the end of round $f+1$. There are two cases to consider.

*Case 1.* (See Figure 4.1.) Process $p_i$ is alive in $y_0$ and $y_1$. Consider the extension of $y_0$ to a run $r'(y_0)$ such that $p_i$ crashes at the beginning of round $f+1$, and no process crashes thereafter. (Recall that $f \leq t-1$.) Notice that $r'(y_0)$ is a run with at most $f+1$ crashes and $p_i$ is a faulty process in $r'(y_0)$. Thus, from our initial assumption about $A$, it follows that there is a correct process $p_j$ ($\neq p_i$) in $r'(y_0)$ which decides some value $v \in \{0, 1\}$ at round $f+1$. (Notice that, since $p_j \neq p_i$, $p_j$ cannot decide before round $f+1$: as $y_0$ and $y_1$ differ only at $p_i$, if $p_j$ decides by round $f$, then $p_j$ decides identical values in $y_0$ and $y_1$.)

Similarly, consider the extension of $y_1$ to a run $r'(y_1)$ such that $p_i$ crashes at the beginning of round $f+1$, and no process crashes thereafter. Notice that, at the end of round $f+1$, $p_j$ cannot distinguish $r'(y_1)$ from $r'(y_0)$ because $p_j$ does not receive any message from $p_i$ in round $f+1$ of both runs. Therefore, as in $r'(y_0)$, $p_j$ decides $v$ at the end of round $f+1$ in $r'(y_1)$.

Consider runs $r'(y_{1-v})$ and $r(y_{1-v})$. Since $f \leq n-3$, in run $r'(y_{1-v})$ there is a correct process $p_l$ which is distinct from $p_i$ and $p_j$. Obviously, $p_l$ is also correct in $r(y_{1-v})$, and hence, $p_l$ decides $val(y_{1-v}) = 1-v$ at the end of round $f+1$ in $r(y_{1-v})$.

Now we construct a run $r''$ by extending configuration $y_{1-v}$: process $p_i$ crashes in round $f+1$ such that, in round $f+1$, $p_l$ receives a message from $p_i$ but $p_j$ does not receive any message from $p_i$. No process distinct from $p_i$ crashes in round $f+1$ or in a higher round. Obviously, $p_j$ and $p_l$ are correct in $r''$. At the end of round $f+1$ in run $r''$, $p_j$ cannot distinguish $r''$ from $r'(y_{1-v})$ because $p_j$ does not receive any message from $p_i$ in round $f+1$ in both runs. Therefore, $p_j$ decides $v$ at the end of

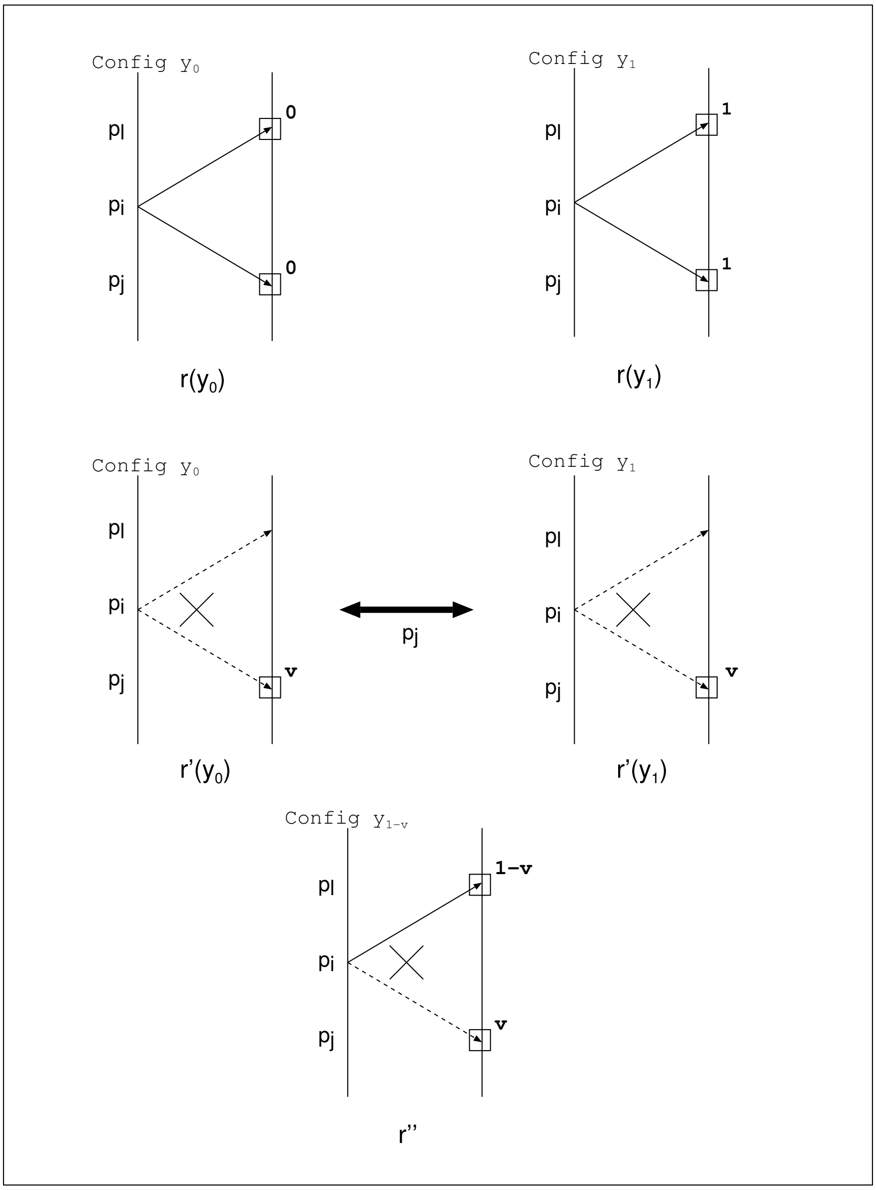


Figure 4.1: Round $f + 1$ of Case 1, Lemma 12

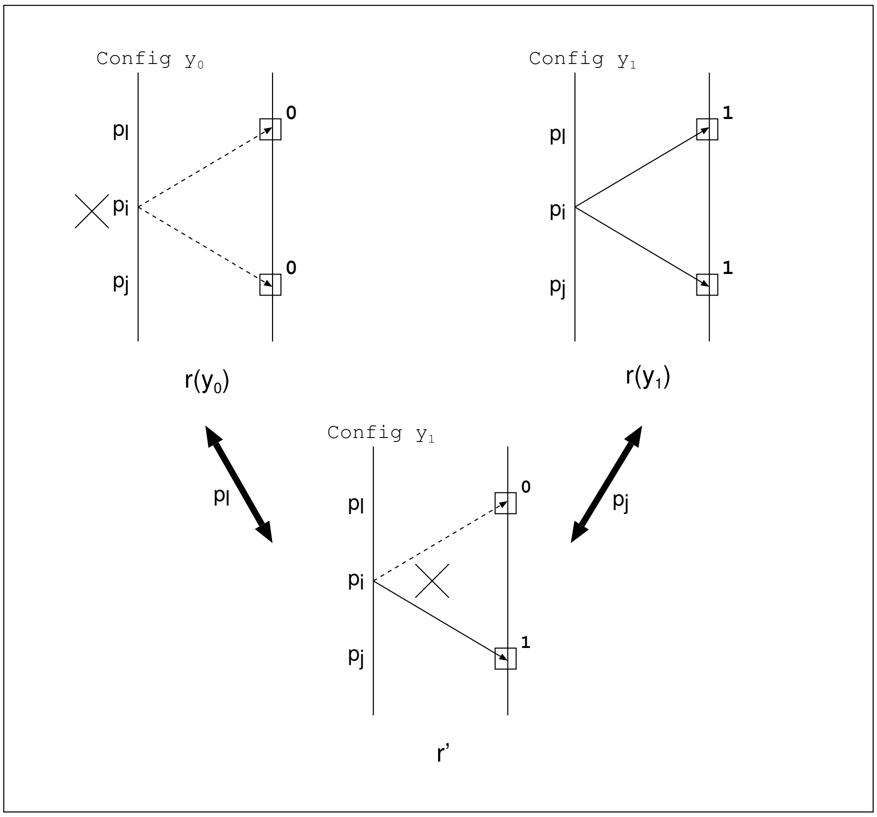


Figure 4.2: Round $f + 1$ of Case 2, Lemma 12

round $f + 1$ in $r''$. However, since $p_l$ receives a message from $p_i$ in round $f + 1$, at the end of round $f + 1$, $p_l$ cannot distinguish $r''$ from $r(y_{1-v})$, and therefore, $p_l$ decides $1-v$ at the end of round $f+1$; a contradiction with agreement property of WA.

*Case 2.* (See Figure 4.2.) Process $p_i$ has crashed in either $y_0$ or $y_1$. Without loss of generality, we can assume that $p_i$ has crashed in $y_0$, and hence, $p_i$ is alive in $y_1$. (Recall that $p_i$ has different states in the two configurations.) As $f$ processes, including $p_i$, have crashed in $y_0$, and $p_i$ has not crashed in $y_1$, so $f - 1$ processes have crashes in $y_1$. Since $f \leq n - 3$ and at most $f - 1$ processes have crashed in $y_1$, so there are at least two correct process $p_j$ and $p_l$ (both distinct from $p_i$) in $r(y_1)$. Consider the run $r'$ which extends $y_1$ such that process $p_i$ crashes in round $f + 1$ and the only alive process that *does not* receive round $f + 1$ message from $p_i$, is $p_l$, and no process crashes after round $f + 1$. Obviously $p_j$ and $p_l$ are correct in $r'$. At the end of round $f + 1$, $p_l$ cannot distinguish $r(y_0)$ from $r'$ because $p_l$ does not receive the round $f + 1$ message from $p_i$ in both runs. Thus, $p_l$ decides 0 at the end of round $f + 1$ in $r'$. At the end of round $f + 1$, $p_j$ cannot distinguish $r(y_1)$

from $r'$ because both runs extend $y_1$ and $p_j$ receives round $f+1$ message from $p_i$ in both runs. Thus, $p_j$ decides 1 at the end of round $f+1$ in $r'$; a contradiction with agreement property of WA. □

## 4.2 Uniform consensus

In this section, we give two lower bounds for weak binary uniform agreement (UA) in the synchronous model. Since any uniform consensus (UC) and non-blocking atomic commit (NBAC) algorithm also solves UA, the lower bounds immediately apply to UC and NBAC.

### Local decision

The following lemma says that any UA algorithm in $SM_t$ has a run in $SM_f$ (i.e., a run with at most $f$ crashes) in which every correct process decides in round $f+1$ or in a higher round.

**Lemma 13** *$\forall t \in [1, n-1]$, $\forall f \in [0, t-1]$, (*$\mathrm{SM}_t$*,* $\mathrm{SM}_f$*, UA, ld) $\geq f+1$.*

**Proof**: Suppose by contradiction that there is a UA algorithm $A$ in $SM_t$ and an integer $f$ in $[0, t-1]$ such that, in every run of $A$ with at most $f$ crashes, some correct process decides by round $f$.

It follows from Lemma 6 that, there are two runs of $A$ in $SM_t$ such that their round $f$ configurations, $y$ and $y'$, satisfy the following: (1) at most $f$ processes have crashed in each configuration, (2) the configurations differ at exactly one process, say $p_i$, and (3) $val(y) = 0$ and $val(y') = 1$.

From our initial assumption about algorithm $A$, it follows that there is an alive process $q_1$ in $y$ which has already decided. (Otherwise, since every correct process in $r(y)$ is an alive process in $y$, $r(y)$ is a run with at most $f$ crashes in which no correct process decides by round $f$.) Furthermore, $q_1$ has decided $val(y) = 0$ in $r(y)$ (and hence, in $y$) because $q_1$ is a correct process in $r(y)$. Similarly, in $y'$, there is an alive process $q_2$ which has decided $val(y') = 1$. There are two cases to consider.

(1) $q_1 \neq p_i$: As $y$ and $y'$ are identical at all processes different from $p_i$, so in $y'$, $q_1$ is alive and has decided 0. Thus, in $r(y')$, $q_1$ is a correct process and decides 0. However, in $r(y')$, every correct process decides $val(y') = 1$; a contradiction.

(2) $q_1 = p_i$: We distinguish two subcases:

- $q_2 = p_i$: Thus $p_i = q_1 = q_2$, and hence, $p_i$ is alive in $y$ and $y'$. Consider a run $r1$ which extends $y$ and in which $p_i$ crashes at the beginning of round $f+1$ and no process crashes thereafter. (Recall that $f \leq t-1$.) As $p_i$ has decided 0 in $y$, it follows from the uniform agreement property that, every correct process decides 0 in $r1$. Since $t < n$, there is at least one correct process, say $p_l$ in $r1$. Now consider a run $r2$ which extends $y'$ and in which $p_i$ crashes at the beginning of round $f+1$ and no process crashes thereafter. Notice that no correct process can distinguish $r1$ from $r2$: at the end of round $f$, no alive process that is distinct from $p_i$ can distinguish $y$ from $y'$, and $p_i$ crashes before sending any message in round $f+1$. Thus every correct process decides the same value in $r1$ and $r2$, in particular $p_l$ decides 0 in $r2$. However, $p_i = q_2$ decides 1 in $r2$; a contradiction with uniform agreement.

- $q_2 \neq p_i$: As $y$ and and $y'$ differ only at $p_i$, $q_2 \neq p_i$ implies that, $q_2$ has the same state in $y$ and $y'$. Thus, in $y$, $q_2$ is alive and has decided 1. In any run which extends $y$, $p_i = q_1$ has decided 0 and $q_2$ has decided 1; a contradiction with uniform agreement.

□

### C-decision

In the following lemma, we show that any UA algorithm in $SM_t$ has a run in $SM_f$ (i.e., a run with at most $f$ crashes) in which *at most one* correct process decides in round $f+1$ or in a *lower* round. In other words, the number of rounds needed for $c$-decision, when $c \geq 2$, is $f+2$. (Following this terminology, Lemma 13 states that the number of rounds needed for 1-decision is $f+1$, and Theorem 3 states that the number of rounds needed for $n-f$ decision is $f+2$.)

**Lemma 14** *$\forall t \in [3, n-1]$, $\forall f \in [0, t-3]$, $\forall c \in [2, n-f]$, ($\mathrm{SM}_t$, $\mathrm{SM}_f$, UA, $d_c$) $\geq f+2$.*

**Proof**: It is sufficient to show that ($SM_t$, $SM_f$, UA, $d_2$) $\geq f+2$. Suppose by contradiction that there is a UA algorithm $A$ in $SM_t$ and an integer $f$ in $[0, t-3]$ such that, in every run of $A$ with at most $f$ crashes, at least two correct processes decide by round $f+1$.

It follows from Lemma 6 that, there are two runs of $A$ in $SM_t$ such that their round $f$ configurations, $y$ and $y'$, satisfy the following: (1) at most $f$ processes have crashed in each configuration, (2) the configurations differ at exactly one process, say $p_i$, and (3) $val(y) = 0$ and $val(y') = 1$. Let $z$ and $z'$ denote the configurations at the end of round $f+1$ of $r(y)$ and $r(y')$, respectively.

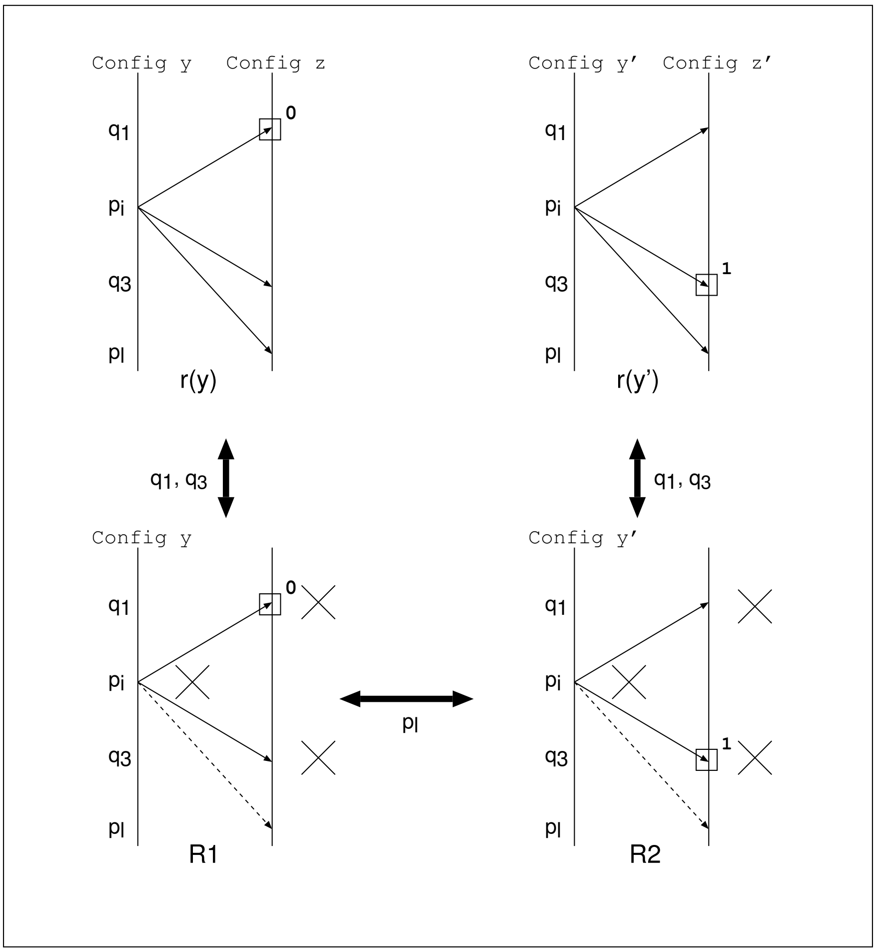


Figure 4.3: Round $f+1$ of Case 1, Lemma 14

From our initial assumption about algorithm $A$, it follows that, in $z$, there are two alive processes $q_1$ and $q_2$ which have decided $val(y) = 0$. (Otherwise, if at most one alive process has decided in $z$, then at most one correct process has decided by round $f+1$ in $r(y)$, a run with at most $f$ crashes; a contradiction.) Similarly, in $z'$, there are two alive processes $q_3$ and $q_4$ which have decided $val(y') = 1$. Since $q_1$ and $q_2$ are distinct, at least one of them is distinct from $p_i$, say $q_1$. Similarly, without loss of generality we may assume that $q_3$ is distinct from $p_i$.

Thus, we have (1) a round $f+1$ configuration $z$ and a process $q_1$ such that, at most $f$ processes have crashed in $z$, and $q_1$ is alive and has decided 0 in $z$, (2) a round $f+1$ configuration $z'$ and a process $q_3$ such that, at most $f$ processes have crashed in $z'$, and $q_3$ is alive and has decided 1 in $z'$, and (3) process $p_i$ is distinct from both $q_1$ and $q_3$. (Processes $q_1$ and $q_3$ might not be distinct.) There are two cases to consider.

*Case 1.* (See Figure 4.3.) Process $p_i$ is alive in $y$ and $y'$. Consider the following two runs of $A$:

**R1** is a run such that (1) the round $f$ configuration is $y$, (2) $p_i$ crashes in round $f+1$ such that only $q_1$ and $q_3$ receive the message from $p_i$, (3) $q_1$ and $q_3$ crash at the beginning of round $f+2$, and (4) no process distinct from $p_i$, $q_1$, and $q_3$ crashes after round $f$. Notice that $q_1$ cannot distinguish the round $f+1$ configuration of $R1$ from $z$, and therefore, decides 0 at the end of round $f+1$ in $R1$. By uniform agreement, every correct process decides 0. Since $t \leq n-1$, there is at least one correct process in $R1$, say $p_l$.

**R2** is a run such that (1) the round $f$ configuration is $y'$, (2) $p_i$ crashes in round $f+1$ such that only $q_1$ and $q_3$ receive the message from $p_i$, (3) $q_1$ and $q_3$ crash at the beginning of round $f+2$, and (4) no process distinct from $p_i$, $q_1$, and $q_3$ crashes after round $f$. Notice that $q_3$ cannot distinguish the round $f+1$ configuration of $R2$ from $z'$, and therefore, decides 1 at the end of round $f+1$ in $R2$. However, $p_l$ cannot distinguish $R1$ from $R2$: at the end of round $f+1$, the two runs are different only at $p_i$, $q_1$, and $q_3$, and none of the three processes send messages after round $f+1$ in both runs. Thus (as in $R1$) $p_l$ decides 0 in $R2$; a contradiction with uniform agreement.

*Case 2.* (See Figure 4.4.) Process $p_i$ has crashed in either $y$ or $y'$. (Process $p_i$ has not crashed in both $y$ and $y'$ because $p_i$ has different states in $y$ and $y'$.) Without loss of generality, we can assume that $p_i$ has crashed in $y$, and hence, $p_i$ is alive in $y'$. Consider the following two runs of $A$:

**R12** is a run such that (1) the round $f$ configuration is $y$ (and hence, $p_i$ has crashed before round $f+1$), (2) no process crashes in round $f+1$, (3) $q_1$ and $q_3$ crash at the beginning of round $f+2$, and (4) no process distinct from $p_i$, $q_1$ and $q_3$ crashes after round $f$. Observe that the round $f+1$ configuration of $R12$ is $z$, and hence, $q_1$ decides 0 at the end of round $f+1$ in $R12$. Due to uniform agreement, every correct process decides 0 in $R12$. Since $t \leq n-1$, there is at least one correct process in $R12$, say $p_l$.

**R21** is a run such that (1) the round $f$ configuration is $y'$, (2) $p_i$ crashes in round $f+1$ such that only $q_1$ and $q_3$ receive the message from $p_i$, (3) $q_1$ and $q_3$ crash at the beginning of round $f+2$, and (4) no process distinct from $p_i$, $q_1$ and $q_3$ crashes after round $f$. Notice that $q_3$ cannot distinguish the round $f+1$ configuration of $R21$ from $z'$ because it receives the round $f+1$ message from $p_i$ in both runs. Thus (as in $z'$) $q_3$ decides 1 at the end of round $f+1$ in $R21$. However, $p_l$ cannot distinguish $R12$ from $R21$: at the end of round $f+1$, the two runs are different only at $p_i$, $q_1$ and $q_3$, and none of them send messages after round $f+1$ in both runs. Thus (as in $R12$), $p_l$ decides 0 in $R21$; a contradiction with uniform agreement. □

**Remark.** Since in a run with at most $f$ crashes, the global decision is $(n-f)$-decision, Lemma 14 generalizes Theorem 3. However, the values of $f$ and $t$, for which

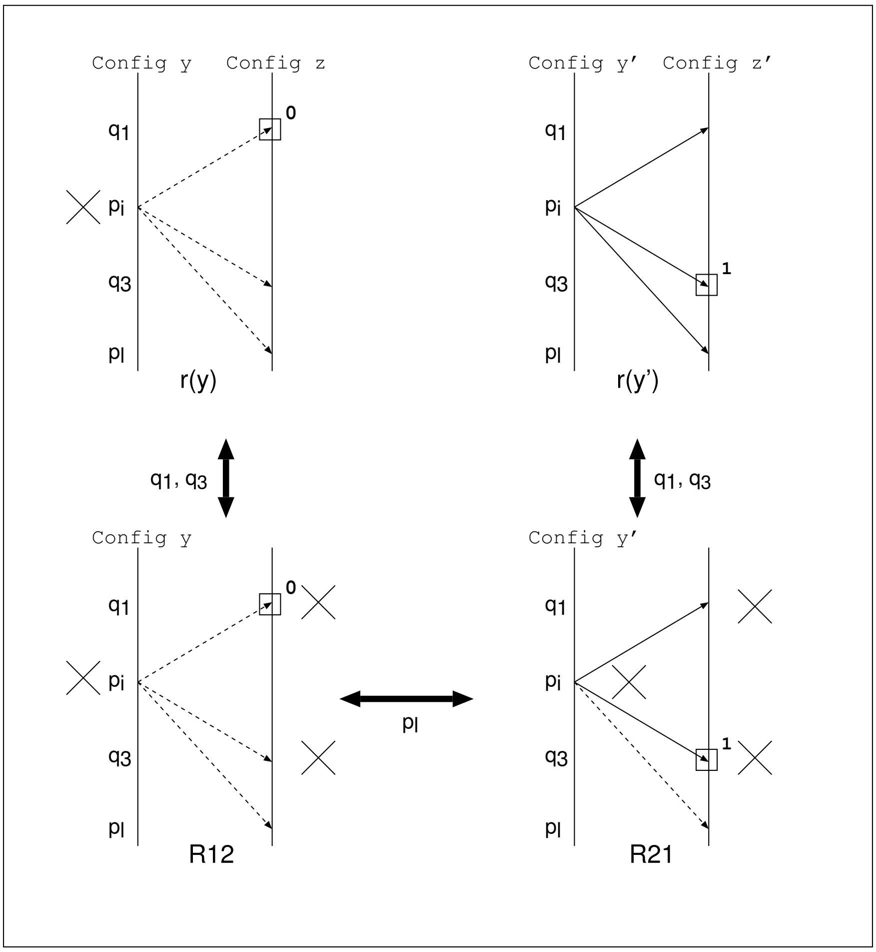


Figure 4.4: Round $f+1$ of Case 2, Lemma 14

Lemma 14 is valid are slightly different from Theorem 3, e.g., unlike Theorem 3, Lemma 14 does not consider the case where $f = t - 2$, as well as, the case where $t = 2$.

## 4.3 Non-Blocking Atomic Commit and Interactive Consistency

### Non-blocking atomic commit

Recall that the lower bounds presented in Section 4.2 hold for both UC and NBAC. In the following, we show that for NBAC, the local decision lower bound in the failure-free case ($f = 0$) can be shifted to 2. This result does not hold for UC. In

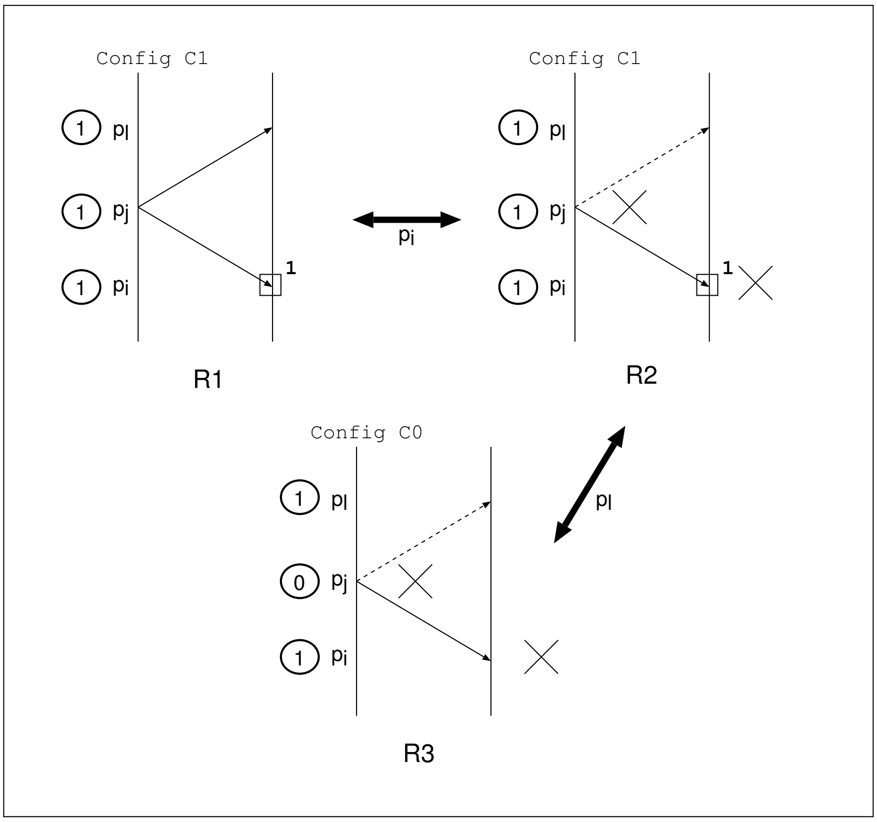


Figure 4.5: Round 1, Lemma 15

Section 4.4.4, we exhibit a UC algorithm that locally decides in 1 round in failure-free runs.

**Lemma 15** $\forall t \in [2, n-1]$, *(*$\text{SM}_t$*,* $\text{SM}_0$*, NBAC, ld)* $\geq 2$.

**Proof**: (See Figure 4.5.) Suppose by contradiction that there is a NBAC algorithm $A$ in $SM_t$ such that, in every failure-free run, some correct process decides in round 1. Let $C1$ be the initial configuration in which all processes propose 1. Consider the failure-free run $R1$ starting from $C1$; i.e., $R1 = r(C1)$. Suppose that, in $R1$, $p_i$ is the process that decides in round 1. It follows from the abort validity property of NBAC that $p_i$ decides 1 in $R1$.

Consider another run $R2$ such that every process proposes 1. Some process $p_j$ ($\neq p_i$) crashes in round 1 and only $p_i$ receives round 1 message from $p_j$. Process $p_i$ crashes at the beginning of round 2, and no process crashes thereafter. At the end of round 1, $p_i$ cannot distinguish $R1$ from $R2$. Thus, $p_i$ decides 1 in round 1 of $R2$. From uniform agreement, every correct process decides 1. There is at least one correct process in $R2$, say $p_l$, because $t \leq n-1$.

Let $C0$ be the initial configuration in which $p_j$ proposes 0 and all other processes propose 1. Consider a run $R3$ starting from $C0$ with the same failure pattern as $R2$; i.e., $p_j$ crashes in round 1 and only $p_i$ receives round 1 message from $p_j$, $p_i$ crashes at the beginning of round 2, and no process crashes thereafter. No process distinct from $p_i$ and $p_j$ can distinguish $R2$ from $R3$: at the end of round 1, only $p_i$ receives a round 1 message from $p_j$, but $p_i$ crashes before sending any message in round 2. Therefore, as in $R2$, every process distinct from $p_i$ and $p_j$, decides 1, in particular $p_l$. But from the commit validity property of NBAC, it follows that, no process should decide 1 in $R3$ because some process ($p_j$) has proposed 0; a contradiction. □

### Interactive consistency

In *RM*, or any of its submodels (e.g., in *SM* or *EM*), any interactive consistency (IC) algorithm can be easily transformed to a NBAC algorithm as follows. Let $V1$ denote the ordered $n$-tuple in which every component is 1. Suppose we have an IC algorithm with IC-propose() primitive. We implement the NBAC-propose() primitive of the NBAC specification as follows:

- When a process NBAC-proposes $v \in \{0, 1\}$, then it IC-proposes $v$.

- If a process IC-decides $V1$, then it NBAC-decides 1; if the process IC-decides a vector different from $V1$, then it NBAC-decides 0.

Uniform agreement and termination properties of NBAC follow directly from the corresponding properties of IC. Consider the remaining two properties of NBAC.

*1. Commit validity:* If some process NBAC decides 1 then it has IC-decided $V1$, and hence, from the IC validity property, every process has IC-proposed and NBAC-proposed 1, thus ensuring NBAC commit validity.

*2. Abort validity:* If some process NBAC decides 0 then it has IC-decided an $n$-tuple that is different from $V1$, and hence, from the IC validity property, either some process is faulty, or some process has IC-proposed a value different from 1. The NBAC abort validity immediately follows.

Observe that the transformation by itself does not require any additional communication. Thus, the time-complexity lower bounds of Lemmas 13, 14 and 15 on NBAC, also apply to IC.

## 4.4 A matching algorithm

One of the first early deciding agreement algorithm was presented in [LF82]. Inspired by that algorithm, [CBS04] showed NC and UC algorithms that match the global decision lower bounds. (The NC algorithm in [CBS04], also matches the global halting lower bound.) However, we knew of no UC algorithm that matches the local decision lower bounds.

In this section, we present an IC algorithm that *simultaneously* matches the local decision, global decision, and global halting lower bounds for most values of $f$ and $t$. (We do not match the lower bound in some boundary cases when the values of $f$, $t$ and $n$ are close to each other.) From our IC algorithm, we then derive matching algorithms for UC and NBAC.

For ease of presentation, in all algorithms presented in this thesis, we assume the following: at the end of round $k$, after generating the message for round $k+1$, a process $p_i$ makes $n$ copies of that message, and sets the tags (*sender*, *recp*, and *halted*) in each copy, to generate an appropriate round $k+1$ message to be sent to each process.

### 4.4.1 IC algorithm overview

Our IC algorithm $A_{ic}$ (Figure 4.6) in $SM_t$ is inspired by the NC algorithm of [CBS04]. The algorithm runs for at most $t+1$ rounds. Process $p_i$ maintains two primary variables: (1) an ordered n-tuple $est_i$, component $j$ of which contains the proposal value of $p_j$, provided $p_i$ has learnt that value (either directly from $p_j$ or relayed by some other process), and $\perp$ otherwise, and (2) a set of processes $halt_i$ that $p_i$ knows to have either crashed or halted.

In each round, the processes exchange estimate (EST) messages containing their *est* values, and compute new values for *est* and *halt*. If *halt* set at a process does not change in round $k$ then (1) if *est* does not change in round $k$ as well, then the process decides on its *est* in round $k$, otherwise, (2) the process decides on its *est* in round $k+1$. Before halting, a process sends a special decision (DEC) message to all processes, so that the processes can distinguish a halt from a crash.

Roughly speaking, if the *halt* set at a process $p_i$ does not change in some round $k$ then, at the end of round $k$, no *alive* process has seen more proposal values than $p_i$. Thus, $p_i$ can decide on $est_i$ at the end of round $k$, but $p_i$ has to ensure that all other processes also see its current *est*. So $p_i$ sends its *est* to all processes in round $k+1$ and then decides. However, if the *est* of $p_i$ does not change in round $k$, then $p_i$ has already sent that *est* to all processes in round $k$; so $p_i$ can decide at the end of round $k$.

at process $p_i$:
initialize()
**in round k** $\{$*rounds 1, ..., $t+1$*$\}$
  send round $k$ messages
  receive messages
  compute()

**procedure** initialize()
  Ordered n-tuple $est_i$ and $newest_i$: element $i$ initialized to $prop_i$ and other elements to $\bot$
  Set $halt_i \leftarrow newhalt_i \leftarrow \emptyset$; Boolean $lastRound_i \leftarrow false$; round 1 msg $\leftarrow (k, \text{EST}, est_i)$
  **for** $1 \leq l \leq t+1$ **do** Multiset $S_i^l \leftarrow \emptyset$

**procedure** compute()
  $halt_i \leftarrow newhalt_i$; $est_i \leftarrow newest_i$
  $S_i^k \leftarrow \{est_j \mid (k, \text{EST}, est_j) \text{ was received}\}$
  **if** $lastRound_i$ **then**
    **if** $dec_i = \bot$ **then** $dec_i \leftarrow est_i$ $\{$*decision*$\}$
    **halt**
  **if** received any $(k, \text{DEC}, est_j)$ **then**
    $newest_i \leftarrow est_j$; $lastRound_i \leftarrow true$
  **else**
    $newhalt_i \leftarrow \Pi \backslash \text{sender}(S_i^k)$ $\{$*processes from which $p_i$ did not receive any message*$\}$
    **for** $1 \leq j \leq n$ **do**
      **if** there is any $est' \in S_i^k$ s.t. $est'[j] \neq \bot$ **then** $newest_i[j] \leftarrow est'[j]$ **else** $newest_i[j] \leftarrow \bot$
    **if** $newhalt_i = halt_i$ **then**
      **if** $est_i = newest_i$ **then** $dec_i \leftarrow est_i$ $\{$*decision*$\}$
      $lastRound_i \leftarrow true$
  **if** $k = t+1$ **then**
    **if** $dec_i = \bot$ **then** $dec_i \leftarrow newest_i$ $\{$*decision*$\}$
    **halt**
  **if** $lastRound_i$ **then** round $k+1$ msg $\leftarrow (k+1, \text{DEC}, newest_i)$
    **else** round $k+1$ msg $\leftarrow (k+1, \text{EST}, newest_i)$

Figure 4.6: Interactive consistency algorithm $A_{ic}$

### 4.4.2 Correctness

In the following, if $p_i$ completes any round $k$, then *for any variable $var_i$, $var_i^k$ denotes the value of that variable at the end of round $k$*; $var_i^0$ denotes the value of the variable at the end the initialization subround in round 1. For $1 \leq k \leq t+1$, $faulty^k$ denotes the set of processes which have crashed by round $k$, and $faulty^0 = \emptyset$. For any pair of ordered $n$-tuples $d$ and $d'$, we say that (1) $d = d'$ if for all $j \in [1, n]$, $d[j] = d'[j]$, (2) $d \preceq d'$ if for all $j \in [1, n]$, either $d[j] = \bot$ or $d[j] = d'[j]$, and (3) $d \npreceq d'$ if $d \preceq d'$ is false.

First, we make the following simple observations which we use frequently in the correctness proofs: (**Observation O1**) for $est$ value at any process and any $j \in [1, n]$, $est[j]$ is either the proposal value of $p_j$ or $\bot$, (**Observation O2**) if $p_j$ does not decide by round $k$ and $p_j$ receives an EST message from some process $p_l$ in round

$k$, then $newest_l^{k-1} \preceq newest_j^k$. (From the loopback property of the model, it follows that $p_j$ receives its own round $k$ message, and therefore, $newest_j^{k-1} \preceq newest_j^k$.)

Every process decides on some *est* value; thus, validity immediately follows from Observation O1. Termination follows from the simple observations that no process halts without deciding and no process completes round $t+1$ without halting (lines 25 to 27). Thus we only detail the proof of uniform agreement. We start with some general lemmas about the algorithm.

**Lemma 16** *If for some $k \in [1, t]$, no process decides by round $k$, then the following holds for any process $p_i$ that completes round $k$. If $lastRound_i^k = true$, then any process $p_j$ that completes round $k$ has $newest_j^k \preceq newest_i^k$.*

**Proof**: We prove the lemma by induction on round number $k \in [1, t]$.

*Base case* $k = 1$. Suppose $lastRound_i^k = true$ and no process decides in round 1. Then $p_i$ has either executed line 17 or line 24 of round 1. Observe that $p_i$ executes line 17 only if some process sends a DEC message to $p_i$. Since *lastRound* is initialized to false and the processes send a DEC messages only when $lastRound = true$, no process sends a DEC message in round 1. Thus $p_i$ has executed line 24. So $newhalt_i^1 = halt_i^1 = \emptyset$, and hence, $newest_i^1$ contains proposal values of all processes. Thus, any process $p_j$ which completes round 1 has $newest_j^1 \preceq newest_i^1$.

*Induction Hypothesis.* If no process decides by round $k$ then the following holds for any process $p_i$ that completes round $k$. If $lastRound_i^k = true$, then any process $p_j$ that completes round $k$, has $newest_j^k \preceq newest_i^k$.

*Induction Step.* Suppose by contradiction that (1) no process decides by round $k+1$, (2) there is a process $p_i$ that completes round $k+1$ such that $lastRound_i^{k+1} = true$ and $newest_i^{k+1} = d'$, and (3) another process $p_j$ completes round $k+1$ with $newest_j^{k+1} = d$ such that $d \npreceq d'$. Process $p_i$ has either executed line 17 or line 24. If $p_i$ executed line 17, then $p_i$ has received message $(k+1,$ DEC, $d')$ from some process $p_l$. To send a DEC message in round $k+1$, $p_l$ must have set $lastRound_l$ to *true* in round $k$. Thus, from the induction hypothesis, every process that completes round $k$ has $newest^k \preceq d'$. Since $d \npreceq d'$, process $p_j$ receives a round $k+1$ message from some process with a n-tuple $d''$ such that $d'' \npreceq d'$; a contradiction because, for all processes which complete round $k$, $newest^k \preceq d'$. Hence, $p_i$ executed line 24, and $halt_i^{k+1} = newhalt_i^{k+1}$. Since $p_j$ completes round $k+1$, $p_i$ received the round $k+1$ message from $p_j$ containing $newest_j^k$, and hence, $newest_j^k \preceq newest_i^{k+1} = d'$. As $newest_j^{k+1} = d \npreceq d'$, it follows that $p_j$ received $(k+1, *, d'')$ from some process $p_m$ such that $d'' \npreceq d'$, and $p_i$ did not receive $(k+1, *, d'')$ from $p_m$ (otherwise, from Observation O2, $d'' \preceq newest_i^{k+1} = d'$). Thus $p_m \in newhalt_i^{k+1}$. However, as $p_m$ completed round $k$, $p_m \notin newhalt_i^k = halt_i^{k+1}$. Thus, $halt_i^{k+1} \neq newhalt_i^{k+1}$; a contradiction. □

**Lemma 17** *If a process $p_i$ has $halt_i^k = newhalt_i^k$ in some round $k \in [1, t]$, then $p_i$ does not complete round $k+1$ without halting.*

**Proof**: If $p_i$ has $halt_i^k = newhalt_i^k$ then $p_i$ sets $lastRound_i$ to true in round $k$, and from lines 13 and 15, $p_i$ does not complete round $k+1$ without halting. □

**Lemma 18** *If no correct process halts by some round $k-1 \in [1, t]$, and if there is a process $p_i$ such that, for every round number $k' \in [1, k]$, $halt_i^{k'} \neq newhalt_i^{k'}$, then $|faulty^k| \geq k$.*

**Proof**: Suppose that there is a round $k$ such that no correct process halts by round $k-1$ and there exists a process $p_i$ such that, for every round number $k' \in [1, k]$, $halt_i^{k'} \neq newhalt_i^{k'}$. Clearly, for all $k' \in [1, k]$, $halt_i^{k'} = newhalt_i^{k'-1} \subseteq newhalt_i^{k'}$. As $newhalt_i^{k'-1} = halt_i^{k'} \neq newhalt_i^{k'}$, $|newhalt_i^{k'-1}| + 1 \leq |newhalt_i^{k'}|$. Thus $|newhalt_i^k| \geq k$. Any process in $newhalt_i^k$ has either halted by round $k-1$ or crashed by round $k$. Since no correct process halts by round $k-1$, every process in $newhalt_i^k$ is faulty, and hence, $|faulty^k| \geq k$. □

**Lemma 19** *If no correct process halts by round $k+1 \in [1, t]$, then $|faulty^k| \geq k$.*

**Proof**: The proof is trivial for $k+1 = 1$. So we consider the case $k+1 \in [2, t]$. Suppose that no correct process halts by round $k+1$.

Consider any correct process $p_i$. Since $p_i$ does not halt by round $k+1$, it follows from Lemma 17 that for every $k' \in [1, k]$, $halt_i^{k'} \neq newhalt_i^{k'}$. Since no correct process halts by round $k-1$, applying Lemma 18, we have $|faulty^k| \geq k$. □

**Lemma 20** *If every process that decides, decides in line 23 of round $t+1$ or line 26 of round $t+1$, then $|faulty^t| \geq t$.*

**Proof**: Suppose that every process that decides, decides in line 23 of round $t+1$ or line 26 of round $t+1$. Consider any correct process $p_i$. Since $p_i$ does not decide in line 14 of round $t+1$, $lastRound_i^t = false$. Thus $newhalt_i^t \neq halt_i^t$ (from lines 22 and 24). Furthermore, as $p_i$ does not halt by round $t$, for every $k \in [1, t-1]$, $newhalt_i^k \neq halt_i^k$. Thus for every $k \in [1, t]$, $newhalt_i^k \neq halt_i^k$. Observing that no process halts by round $t-1$ and applying Lemma 18, we have $|faulty^t| \geq t$. □

**Lemma 21** *(Uniform Agreement) No two processes decide differently.*

**Proof**: If no process decides then the lemma trivially holds. Suppose some process decides. Consider the lowest round number $k$ in which some process decides. Let $p_i$ be a process that decides in round $k$, say on some $n$-tuple $d$. We divide the proof into two parts: (a) $k \leq t+1$ and $p_i$ does not decide in line 26 of round $t+1$, and (b) $k = t+1$ and $p_i$ decides in line 26 of round $t+1$.

(a) *$k \leq t+1$ and $p_i$ does not decide in line 26 of round $t+1$:* Process $p_i$ decides either in (1) line 14 or (2) line 23 of round $k$. In both cases, we show the following: no process can decide a $n$-tuple different from $d$ in round $k$, and any process that completes round $k$ without deciding, does so with $newest^k = d$. This immediately implies uniform agreement. (Note that, even if $k = t+1$, and another process $p_j$ decides in line 26 of round $k$, $p_j$ decides on $newest_j^k = d$.)

- *Process $p_i$ decides in line 14 of round $k$:* Notice that $k > 1$ because no process can decide at line 14 in round 1 (as $lastRound^0 = false$). Since $p_i$ decides in line 14, $lastRound_i^{k-1} = true$ and $p_i$ sends a DEC message in round $k$. We claim that every DEC message sent in round $k$ is ($k$, DEC, $d$). Suppose that another process $p_j$ sends a ($k$, DEC, $d1$) message. Then $lastRound_j^{k-1} = true$. Since no process decides by round $k-1$, applying Lemma 16 twice we have $d1 = newest_j^{k-1} \preceq newest_i^{k-1} = d$ and $d = newest_i^{k-1} \preceq newest_j^{k-1} = d1$, i.e., $d1 = d$. As $p_i$ completes round $k$, every process receives at least one ($k$, DEC, $d$) message, and either decides $d$ in line 14 or adopts $d$ as *newest* in line 17.

- *Process $p_i$ decides in line 23 of round $k$:* We claim that no process decides a value different from $d$ in round $k$. Clearly, $p_i$ does not receive any DEC message in round $k$ (otherwise, $p_i$ does not execute line 23). Suppose some process $p_j$ decides $d1$ in round $k$. If process $p_j$ decides in line 14, then $p_j$ sends DEC message in round $k$, and $p_i$ receives that message (as $p_j$ executes the receive subround in round $k$, so none of its round $k$ messages are lost); a contradiction. Suppose $p_j$ decides in line 23. From the predicate at line 23, if follows that $p_i$ sent ($k$, EST, $d$) in round $k$ and $p_j$ sent ($k$, EST, $d1$) in round $k$. Since $p_i$ receives round $k$ message from $p_j$ and vice versa, $d1 \preceq d$ and $d \preceq d1$, i.e., $d = d1$. If $p_j$ decides in line 26, then it decides on the *newest* value adopted in round $t+1$. We show below that every process that updates its *newest* in round $k$, updates it to $d$.

  We now show that any process that completes round $k$ without deciding at line 14 or line 23, does so with $newest = d$. Suppose by contradiction that some process $p_j$ completes round $k$ with $newest = d2 \neq d$ and without deciding in line 14 or 23. Process $p_j$ updates its variable *newest* in line 17 or line 21. Suppose $p_j$ updates its *newest* in line 17. Then $p_j$ has received a DEC message from some process $p_m$. Since $p_i$ decides at line 23, it does not receive any DEC message in round $k$. Thus $p_m \in newhalt_i^k$. Since $p_m$ completes round $k-1$, $p_m \notin newhalt_i^{k-1} = halt_i^k$ (if $k = 1$ then obviously $p_m \notin halt_i^k = \emptyset$.). Hence, the predicate at line 22 evaluates to false at $p_i$, and $p_i$ cannot decide in line 23; a contradiction. Thus, $p_j$ updates its *newest* in line 21. Since $p_i$ completes round $k$ by deciding $d$ and evaluates the condition in line 23 to true, $p_i$ sends a ($k$, EST, $d$) in round $k$. Thus $p_j$ receives ($k$, EST, $d$) from $p_i$, and hence, $d \preceq d2$. As $d2 \neq d$, it follows that $d2 \npreceq d$. Consequently, there is a process $p_m$ such that $p_j$ receives $d3 \npreceq d$ from $p_m$, but $p_i$ does not receive any message from $p_m$ in round $k$. Thus, $p_m \in newhalt_i^k$. However, $p_m$ completes round $k-1$ and hence, $p_m \notin newhalt_i^{k-1} = halt_i^k$ (if $k = 1$ then obviously $p_m \notin halt_i^k = \emptyset$). Hence, the predicate at line 22 evaluates to false at $p_i$, and $p_i$ cannot decide in line 23; a contradiction.

(b) $k = t+1$ *and* $p_i$ *decides in line 26 of round* $k = t+1$*:* From the definition of $k$, every process which decides, decides in round $t+1$. We have shown above that, if any process decides in line 14 or line 23 of round $t+1$, then every process which decides in round $t+1$, decides the same value. Therefore, we need to only consider the case where every process which decides, decides at line 26 of round $t+1$. From Lemma 20, we have $|faulty^t| \geq t$. Since at most $t$ processes may crash in a run, $|faulty^t| = t$, and hence, every process which enters round $t+1$, is a correct process. Consequently, every process which enters round $t+1$, receives the same set of messages in round $t+1$. Observe that no process sends DEC message in round $t+1$ (otherwise, that process decides in line 14 of round $t+1$ or line 23 of round $t$; a contradiction). Thus every process which enters round $t+1$, updates *newest* to the same value in line 21, and decides on identical values in line 26. □

### 4.4.3 Time-complexity

We now discuss the time complexity properties of $A_{ic}$. In the following lemma, we show that, in runs with $f \geq 1$ crashes, the algorithm achieves a local decision in $f+1$ rounds and a global decision in $f+2$ rounds. However, when $f = 0$, the local decision takes the same number of rounds as the global decision (2 rounds) — recall that, we showed in Section 4.3 that IC algorithms require 2 rounds for a local decision when $f = 0$. (In Section 4.4.4, we describe a UC algorithm that achieves local decision in 1 round when $f = 0$.)

We say that a process $p_i$ *learns* index $l \in [1, n]\backslash\{i\}$ in round $k$ if $newest_i^{k-1}[l] = \bot$ and $newest_i^k[l] \neq \bot$. (In other words, $p_i$ learns about the proposal value of $p_l$ in round $k$.) We say that $p_i$ learns index $i$ in round 0. We also say that $p_i$ learns index $l$ from $p_j$ in round $k$ if $newest_i^{k-1}[l] = \bot$ and $p_i$ receives a round $k$ message from $p_j$ containing an *est* such that $est[l] \neq \bot$. On the other hand, if $p_j$ sends an *est* such that $est[l] \neq \bot$ the we say that $p_j$ *propagates* index $l$ in round $k$. (Note that there may be more than one process from which a process learns the same index in a round.) Clearly, if $p_i$ propagates $l$ in round $k$, then $p_i$ learns $l$ in a lower round.

**Lemma 22** *In any run with at most $f$ faulty processes, the following properties hold:*
*(a) if $f \in [1, t]$, then there is a correct process that decides by round $f + 1$.*
*(b) if $f \in [0, t-2]$, then any process that halts, halts by round $f + 2$.*
*(c) Any process that halts, halts by round $t + 1$.*

**Proof**: (a) For $f = t$, the proof is trivial because every correct process decides by round $t + 1$. Consider a run in which at most $f \in [1, t-1]$ processes crash, and suppose, by contradiction that no correct process decides by round $f + 1$. Thus, no correct process halts by round $f + 1 \leq t$. It follows from Lemma 19 that $|faulty^f| \geq f$. Since at most $f$ processes fail in the run, $|faulty^f| = f$ and every process which enters round $f + 1$ is correct. Furthermore, since no correct process halts by round $f$, Lemma 19 implies that $|faulty^{f-1}| \geq f - 1$. Since $|faulty^f| = f$, at most one process crashes in round $f$ (**Observation O3**).

Let $S$ be the set of processes that enter round $f + 1$. Since every process in $S$ is correct, all of them complete round $f + 1$. We establish a contradiction by showing that some process in $S$ decides in line 23 of round $f + 1$. We demonstrate this fact indirectly by showing the following four claims for processes in $S$ in round $f + 1$: (1) every process has *lastRound* = *false* in line 13, (2) no process receives a DEC message in round $f + 1$, (3) every process evaluates the predicate at line 22 to true, and (4) some process evaluates the predicate at line 23 to true.

*Claim 1.* If $lastRound = true$ at a process in line 13 of round $f+1$ then that process halts in round $f + 1$. This immediately leads to a contradiction because we know that every process in $S$ is correct, and no correct process halts by round $f + 1$.

*Claim 2.* Suppose by contradiction that some process $p_i \in S$ receives a DEC message from some process $p_j$ in round $f + 1$. Since every process which enters round $f + 1$ is correct, $p_j$ is a correct process, and hence, $p_j$ decides in line 14 of round $f + 1$ or line 23 of round $f$; a contradiction. Thus no process in $S$ receives a DEC message in round $f + 1$.

*Claim 3.* Suppose by contradiction that some process $p_i \in S$ evaluates the predicate at line 22 to false; i.e., $halt_i^{f+1} \neq newhalt_i^{f+1}$. Since $p_i$ does not halt by round $f + 1$, from Lemma 17 we have, $halt_i^k \neq newhalt_i^k$ for every $k$ in $[1, f]$. Thus $halt_i^k \neq newhalt_i^k$ for every $k$ in $[1, f + 1]$. As no correct process halts by round $f + 1$, from Lemma 18 it follows that $|faulty^{f+1}| \geq f + 1$; a contradiction.

*Claim 4.* (See Figure 4.7.) Suppose by contradiction that every process in $S$ evalu-

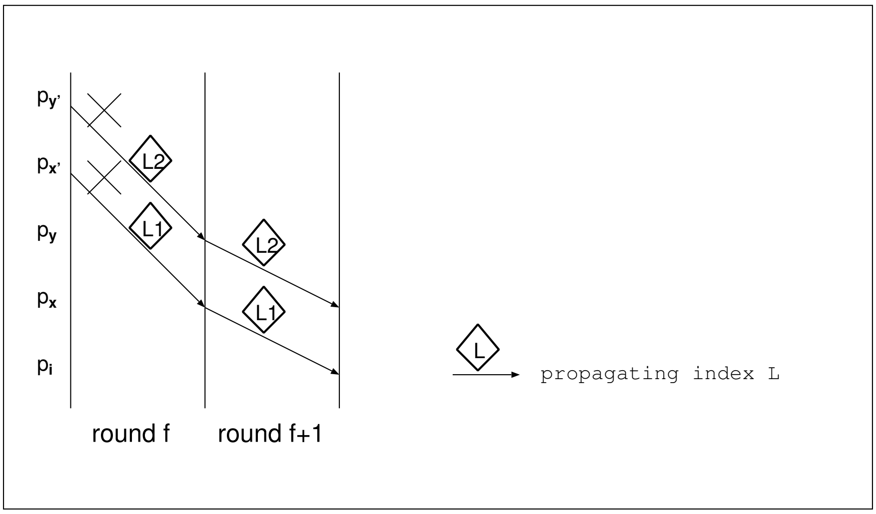


Figure 4.7: Claim 4, Lemma 22(a)

ates the predicate at line 23 to false. It follows that, in round $f+1$, every process in $S$ learns an index. (Recall that every process that enters round $f+1$ is correct and is in set $S$.)

Consider a process $p_i \in S$ which learns index $l1$ in round $f+1$ from some process $p_x$. Suppose $p_x$ learns index $l2$ in round $f+1$ from process $p_y$. Since $p_i$ learns from $p_x$, $p_i \neq p_x$. Similarly, $p_x \neq p_y$. (Note that $p_i$ and $p_y$ may not be distinct.) Since $p_x$ propagates $l1$ and learns $l2$ in the same round, we have $l1 \neq l2$.

Since $p_x$ is a correct process, $p_x$ learns $l1$ in round $f$ (otherwise, if $p_x$ learned $l1$ in a round lower than $f$, $p_x$ would have propagated $l1$ to $p_i$ by round $f$). Similarly, $p_y$ learns $l2$ in round $f$. Consider the process $p'_x$ from which $p_x$ learns $l1$ in round $f$. Process $p'_x$ must have crashed in round $f$, otherwise, on receiving the round $f$ message from $p'_x$, $p_i$ would have learned $l1$ in round $f$. Similarly, the process $p'_y$ from which $p_y$ learns $l2$ in round $f$ must have crashed in round $f$, otherwise, $p_x$ would have learned $l2$ from $p'_y$ in round $f$. We claim that $p'_x$ and $p'_y$ are distinct processes. Otherwise, if $p'_x = p'_y$, then $p'_x$ propagates both $l1$ and $l2$ in round $f$, and when $p_x$ receives a message from $p'_x$ in round $f$, $p_x$ learns both $l1$ and $l2$ in round $f$; a contradiction. (Recall that we assumed that $p_x$ learns $l2$ in round $f+1$.)

We thus have two processes $p'_x$ and $p'_y$ that crash in round $f$. However, from Observation O3, we know that at most one process crashes in round $f$; a contradiction.

(b) Consider a run where at most $f \in [0, t-2]$ processes crash, and suppose by contradiction that a process $p_i$ completes round $f+2$ without halting. Observe that, if any process $p_j$ halts at round $k \leq f+1$ then $p_j$ sends a DEC message in round $k$. Since $p_j$ completes round $k$, $p_i$ receives the DEC message from $p_j$, sets *lastRound* to *true* in round $k$, and halts in round $k+1 \leq f+2$. Thus no process halts by $f+1$. As $p_i$ does not halt by round $f+2$, so for every $k \in [1, f+1]$, we have $newhalt_i^k \neq halt_i^k$. Applying Lemma 18 we have $|faulty^{f+1}| \geq f+1$; a contradiction.

(c) Obvious from the algorithm. □

### 4.4.4 NBAC, UC and NC algorithms

In Section 4.3, we showed how to transform any IC algorithm to a NBAC algorithm, without any additional communication between processes. An equally straightforward transformation generates a UC algorithm from an IC algorithm: on UC-propose($v$), a process invokes IC-propose($v$), and if a process IC-decides an n-tuple $d$, then it UC-decides $d[l]$ where $l$ is the lowest index such that $d[l] \neq \bot$.

Our IC algorithm $A_{ic}$ does not locally decide in round 1 in a failure-free run ($f = 0$). Therefore, to match the early local decision lower bound for UC when $f = 0$, we modify the UC algorithm obtained from $A_{ic}$ by adding the following: $p_1$ UC-decides on its proposal value $v_1$ in the computation subround of round 1. To see why this modification does not violate the agreement property of UC, notice that, if $p_1$ executes the receive subround of round 1, then none of its round 1 messages are lost. Therefore, every process which completes round 1 has $newest[1] = v_1$. Subsequently, at all processes, $newest[1]$ and $est[1]$ is always $v_1$. Thus no process can UC-decide a value different from $v_1$.

Now consider NC. We showed in Section 1.2 that there is a NC algorithm that matches the local decision lower bound. As we mentioned earlier, [CBS04] gives an algorithm that matches the global decision and the global halting bounds for NC. Recall that it follows from Lemma 12 that no single NC algorithm can match both the early local decision and early global decision lower bounds.

## 4.5 Summary of results in the synchronous model

Combining our lower bound results with algorithm $A_{ic}$, the derived NBAC and UC algorithms, and the trivial NC algorithm sketched in Section 1.2, we get the following tight bounds.

**Theorem 23** *(Local decision tight bound for consensus.)*
$\forall t \in [1, n-1]$, $\forall f \in [0, t]$, *(*$\mathrm{SM}_t$, $\mathrm{SM}_f$, *NC, ld) =* $f$.

**Proof**: Follows from Lemma 10, and the NC algorithm sketched in Section 1.2. □

**Theorem 24** *(Local decision tight bound for uniform consensus.)*
$\forall t \in [1, n-1]$, $\forall f \in [0, t-1]$, $(\mathrm{SM}_t,\ \mathrm{SM}_f,\ UC,\ ld) = f + 1$.

**Proof**: Follows from Lemma 13, and the UC algorithm derived from $A_{ic}$ in Section 4.4.4. □

**Theorem 25** *(Local decision tight bounds for non-blocking atomic commit and interactive consistency.)*
*(a)* $\forall t \in [1, n-1]$, $\forall f \in [1, t-1]$, $\forall P \in \{NBAC,\ IC\}$, $(\mathrm{SM}_t,\ \mathrm{SM}_f,\ P,\ ld) = f + 1$.
*(b)* $\forall t \in [1, n-1]$, $\forall P \in \{NBAC,\ IC\}$, $(\mathrm{SM}_t,\ \mathrm{SM}_0,\ P,\ ld) = 2$.

**Proof**: Lower bound for part (a) follows from Lemma 13. Lower bound for part (b) follows from Lemma 15. The matching algorithms are the IC algorithm $A_{ic}$ presented in Section 4.4, and the NBAC algorithm derived from $A_{ic}$ in Section 4.4.4. □

**Theorem 26** *(For $c \geq 2$, c-decision tight bounds for uniform consensus, non-blocking atomic commit and interactive consistency.)*
$\forall t \in [3, n-1]$, $\forall f \in [0, t-3]$, $\forall c \in [2, n-f]$, $\forall P \in \{UC,\ NBAC,\ IC\}$,
$(\mathrm{SM}_t,\ \mathrm{SM}_f,\ P,\ d_c) = f + 2$.

**Proof**: Note that algorithm $A_{ic}$ globally decides by round $f + 2$ in runs with at most $f$ crashes, and therefore, for all $c \in [2, n-f]$, *c-decides* by round $f + 2$ in runs with at most $f$ crashes. Thus the theorem follows from Lemma 14, algorithm $A_{ic}$, and the NBAC and UC consensus algorithms derived from $A_{ic}$. □

# Chapter 5

# Tight Bounds in the Eventually Synchronous Model

## Part A — Synchronous Runs

In this chapter and in Chapter 6, we investigate bounds for uniform consensus (UC) in the eventually synchronous model ($EM_t$). We do not consider the bounds for non-blocking atomic commit (NBAC) because the problem is impossible to solve in the eventually synchronous model when processes may crash (Lemma 7). In Section 4.3 we showed that any interactive consistency (IC) algorithm can be transformed to solve non-blocking atomic commit without any additional communication. Thus, interactive consistency is also impossible to solve in the eventually synchronous model when processes may crash. Furthermore, in this model, any algorithm that solves consensus (NC) also solves uniform consensus (Lemma 8). Thus in the eventually synchronous model, we only investigate lower bounds for uniform consensus. To strengthen our lower bounds, in all subsequent lower bound proofs, we only consider binary uniform consensus, i.e., we fix $V = \{0, 1\}$.

In this chapter, we focus on *synchronous runs* of the eventually synchronous model, namely, runs that are also runs of the synchronous model ($SM_t$) (in other words, runs with $GSR = 1$). As $SM_t$ is a submodel of $EM_t$, lower bounds for local decision and global decision in $SM_t$ also hold in synchronous runs of $EM_t$, namely, the early local decision lower bound is $f + 1$ and the early global decision lower bound is $f + 2$. However, we knew of no matching algorithms for these bounds. The only exception is the failure-free case ($f = 0$): the global decision tight bound is 2 rounds in failure-free synchronous runs of $EM_t$ [KR03, Sch97, MR99].

## 5.1 Local decision lower bound

In following lemma, we state that, for most values of $f$, the early local decision lower bound in synchronous runs can be improved to be the same as the early global decision lower bound, namely, $f + 2$. In other words, every UC algorithm in $EM_t$ has a run in $SM_f$ (i.e., a synchronous run with at most $f$ crashes) in which every correct process decides in round $f + 2$ or in a higher round. (Note that, as $t$ and $n$ are integers, $t < n/2$ is equivalent to $t \leq (n-1)/2$.)

**Lemma 27** *$\forall t \in [1, (n-1)/2]$, $\forall f \in [0, t-3]$, $(\mathrm{EM}_t, \mathrm{SM}_f, UC, ld) \geq f + 2$.*

*Remarks.* We exclude the following two cases. (1) $t = 0$: in this case, processes can decide after exchanging proposal values in the very first round in synchronous runs (e.g., decide always on the proposal value of $p_1$). (2) $t \geq n/2$: in this case, from Lemma 9, we know that there is no UC algorithm in $EM_t$.

**Proof**: Suppose by contradiction that there is a UC algorithm $A$ in $EM_t$ and an integer $f$ in $[0, t-3]$ such that, in every synchronous run of $A$ with at most $f$ crashes some correct process decides by round $f + 1$. Since $SM_t$ is a submodel of $EM_t$, algorithm $A$ is a UC algorithm in $SM_t$ as well. If follows from Lemma 6 that, there are two runs of $A$ in $SM_t$ such that their round $f$ configurations, $y$ and $y'$, satisfy the following: (1) at most $f$ processes have crashed in each configuration, (2) the configurations differ at exactly one process, say $p_i$, and (3) $val(y) = 0$ and $val(y') = 1$.

Let $z$ and $z'$ denote the configurations at the end of round $f + 1$ of $r(y)$ and $r(y')$, respectively. Runs $r(y)$ and $r(y')$ are runs of $A$ in $SM_t$, and so, synchronous runs of $A$ in $EM_t$. As $r(y)$ and $r(y')$ each have at most $f$ crashes, it follows from our assumption on $A$ that, some correct process decides by round $f + 1$ in each run. Thus, there is at least one alive process in $z$, say $q_1$, that has decided 0. Similarly, there is at least one alive process in $z'$, say $q_3$, that has decided 1. There are three cases to consider.

*Case 1.* $p_i \notin \{q_1, q_3\}$. This case is exactly the same as the case in the proof of Lemma 14. We can derive a contradiction by constructing the same runs R1, R2, R12, and R21 in $SM_t$. (These are valid runs of $A$ in $EM_t$ because $SM_t$ is a submodel of $EM_t$.)

*Case 2.* (See Figure 5.1.) $p_i \in \{q_1, q_3\}$ and $p_i$ is alive in both $y$ and $y'$.

> *Remark:* To see why we cannot reuse the proof of Lemma 14 in this case, observe that, if $p_i = q_1$ then run $R1$ (in the proof of Lemma 14) is not a valid run of $A$ in $SM_t$: in $SM_t$, $p_i$ cannot decide in the computation subround of

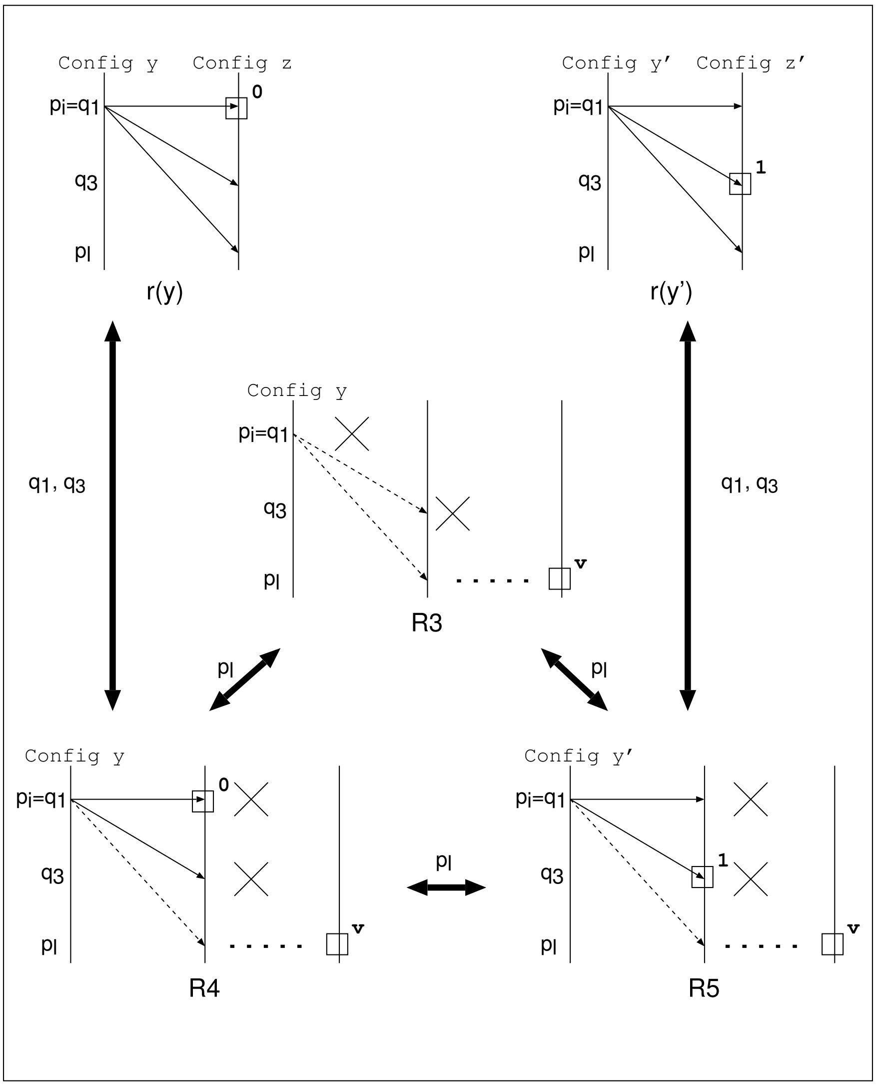


Figure 5.1: Rounds $f+1$ and $K1$ of Case 2, Lemma 27

round $f+1$ while some of its round $f+1$ messages are lost. Similarly, if $p_i = q_3$ then run $R2$ is not a run in $SM_t$. Hence, in this case, we construct some runs of $A$ in $EM_t$, that are not in $SM_t$ (i.e., non-synchronous runs), to derive a contradiction.

Without loss of generality we can assume that $p_i = q_1$. (Note that the proof holds even if $p_i = q_1 = q_3$.) Consider the following three runs (R3 is a synchronous run, whereas R4 and R5 are non-synchronous runs):

**R3** is a run such that (1) the round $f$ configuration is $y$, (2) $p_i$ crashes at the beginning of round $f+1$, (3) if $q_3 \neq p_i$ then $q_3$ crashes at the beginning of round $f+2$, (4) no process distinct from $p_i$ and $q_3$ crashes in round $f+1$ or a in higher round, and (5) no message sent in round $f+1$ or a in higher round is lost. Since $t \leq (n-1)/2 \leq n-1$, there is at least one correct process in $R3$, say $p_l$. Suppose $p_l$ decides $v \in \{0,1\}$ in some round $K1 \geq f+1$. (To see why $p_l$ cannot decide before round $f+1$ in $R3$, notice that the state of $p_l$ at the end of round $f$ is the same in the three runs $r(y)$, $r(y')$ and $R3$, because $p_l \neq p_i$. If $p_l$ decides $v$ before round $f+1$ in $R3$ then it also decides $v$ in $r(y)$ and $r(y')$. However, $val(y) \neq val(y')$.)

**R4** is a run such that (1) the round $f$ configuration is $y$, (2) $p_i$ and $q_3$ crash at the beginning of round $f+2$, and only $p_i$ and $q_3$ receive the round $f+1$ message from $p_i$ (all other round $f+1$ messages from $p_i$ are lost — as $R4$ is a non-synchronous run, messages may be lost in round $f+1$ even if the sender does not crash in that round), (3) no process distinct from $p_i$ and $q_3$ crashes in round $f+1$ or in a higher round, and (4) the only messages lost in round $f+1$ are the messages from $p_i$ to $\Pi\backslash\{p_i, q_3\}$, and no message is lost in a higher round. Notice that $p_i$ cannot distinguish the configuration at the end of round $f+1$ in $R4$ from $z$, and thus, $p_i$ decides 0 at the end of round $f+1$ in $R4$ (because $q_1 = p_i$ decides 0 in $z$). However, $p_l$ cannot distinguish round $K1$ configuration of $R4$ from that of $R3$ because (a) at the end of round $f$, the two runs are different only at $p_i$, (b) all round $f+1$ messages sent by $p_i$ to $\Pi\backslash\{p_i, q_3\}$ are lost, and (c) $p_i$ and $q_3$ do not send messages after round $f+1$. Thus (as in $R3$) $p_l$ decides $v$ in round $K1$.

**R5** extends $y'$ in the same way as R4 extends $y$. Namely, R5 is a run such that (1) the round $f$ configuration is $y'$, (2) $p_i$ and $q_3$ crash at the beginning of round $f+2$, and only $p_i$ and $q_3$ receive the round $f+1$ message from $p_i$ (all other round $f+1$ messages from $p_i$ are lost), (3) no process distinct from $p_i$ and $q_3$ crashes in round $f+1$ or in a higher round, and (4) the only messages lost in round $f+1$ are the messages from $p_i$ to $\Pi\backslash\{p_i, q_3\}$, and no message is lost in a higher round. Notice that $q_3$ cannot distinguish the configuration at the end of round $f+1$ in $R5$ from $z'$ (because in both configurations, $q_3$ has received the round $f+1$ message from $p_i$), and thus, $q_3$ decides 1 at the end of round $f+1$ in $R5$. However, $p_l$ cannot

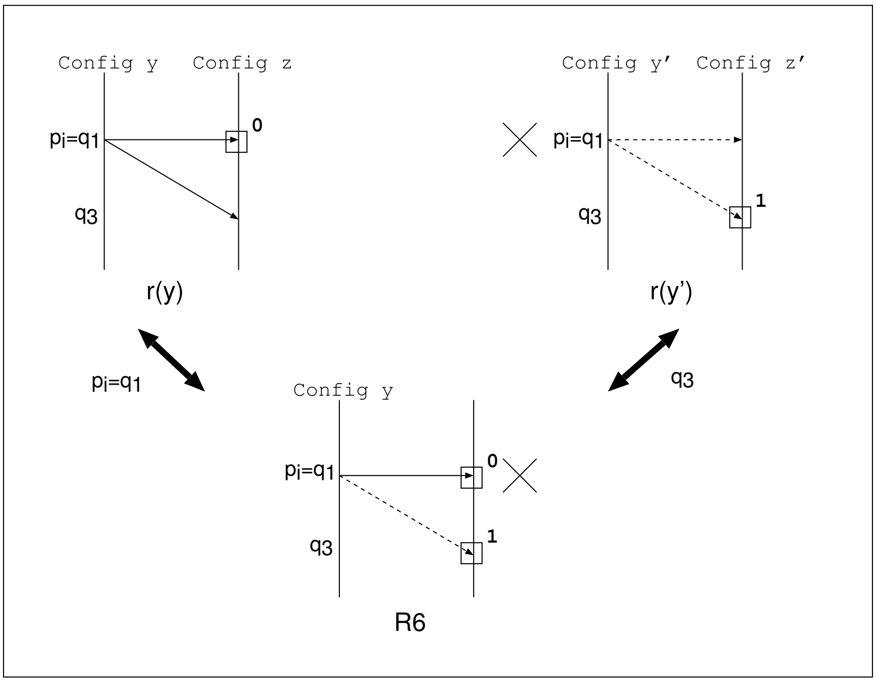


Figure 5.2: Round $f+1$ of Case 3, Lemma 27

distinguish round $K1$ configuration of $R5$ from that of $R3$ because, (a) at the end of round $f$ the two runs are different only at $p_i$, (b) all round $f+1$ messages sent by $p_i$ to $\Pi \backslash \{p_i, q_3\}$ are lost, and (c) $p_i$ and $q_3$ do not send messages after round $f+1$. Thus (as in $R3$) $p_l$ decides $v$ in round $K1$.

Clearly, either $R4$ or $R5$ violates uniform agreement: $p_l$ decides $v$ in both runs, however, $p_i$ decides 0 in $R4$ and $q_3$ decides 1 in $R5$.

*Case 3.* (See Figure 5.2.) $p_i \in \{q_1, q_3\}$ and $p_i$ has crashed in either $y$ or $y'$. (Process $p_i$ has not crashed in both $y$ and $y'$ because $p_i$ has different states in $y$ and $y'$.) Notice that the case $p_i = q_1 = q_3$ is not possible because, in that case, $p_i$ is alive in both $z$ and $z'$, and hence in $y$ and $y'$. We show the contradiction for the case when $p_i = q_1 \neq q_3$. (The contradiction for $p_i = q_3 \neq q_1$ is symmetric.)

Since, $p_i = q_1$, $p_i$ is alive in $z$, and hence, alive in $y$. Thus $p_i$ has crashed in $y'$. Consider the following non-synchronous run:

**R6** is a run such that (1) the round $f$ configuration is $y$, (2) in round $f+1$, only $p_i$ receives the round $f+1$ message from itself (all other messages sent by $p_i$ in round $f+1$ are lost), (3) $p_i$ crashes at the beginning of round $f+2$, (4) no process distinct from $p_i$ crashes in round $f+1$ or in a higher round, and (5) the only messages lost in round $f+1$ are the messages from $p_i$ to $\Pi \backslash \{p_i\}$, and no message is lost in a higher round. Process $p_i$ cannot distinguish the configuration at the end of round $f+1$ in $R6$ from $z$, and therefore, decides 0 (because $q_1 = p_i$ decides 0 in $z$). However, $q_3$ does not receive the round $f+1$ message from $p_i$ in $R6$, and hence, $q_3$ cannot distinguish the configuration at the end of round $f+1$ in $R6$ from $z'$. (Observe that, in $z'$, $q_3$ does not receive the round $f+1$ message from $p_i$ because $p_i$ has crashed in $y'$.) Consequently, $q_3$ decides 1 in $R6$; a contradiction with uniform agreement. □

**Remark.** A closer look at the proof of lemma 27 reveals that the non-synchronous runs we construct (R4, R5, and R6) require only a small amount of non-synchrony in the model. The three runs are valid in a weakened synchronous model where the following holds: even if some message from process $p_i$ is lost in round $f+1$, then $p_i$ might complete round $f+1$. (Recall that, in the synchronous model, if some messages from $p_i$ is lost in round $f+1$, then $p_i$ has necessarily crashed before the receive subround of round $f+1$.) It is easy to see that such runs are also valid in the synchronous send-omission model [Had83, PR04] as well as in an asynchronous round based model enriched with a *Perfect* failure detector [CT96]. Thus the $f+2$ early local decision lower bound in synchronous runs also extends to these two models.

## 5.2 Global decision lower bound

In this section we show that the early global decision lower bound for synchronous runs of any UC algorithm in $EM_t$ is $f+2$. We first show the bound for $f = t$ (worst-case), and then derive the bound for the remaining values of $f$.

The following lemma says that every UC algorithm in $EM_t$ has a run in $SM_t$ in which some correct process decides in round $t+2$ or in a higher round.

**Lemma 28** *$\forall t \in [1, (n-1)/2]$, ($\mathrm{EM}_t$, $\mathrm{SM}_t$, UC, gd) $\geq t+2$.*

**Proof**: Suppose by contradiction that there is a UC algorithm $A$ in $EM_t$ such that, in every synchronous run of $A$ with $t$ crashes every correct process decides by round $t+1$. Clearly, algorithm $A$ solves UC in $SM_t$. It follows from Lemma 6 that, there are two runs of $A$ in $SM_t$ such that their round $t-1$ configurations, $y$ and $y'$, satisfy the following: (1) at most $t-1$ processes have crashed in each configuration, (2) the configurations differ at exactly one process, say $p_i$, and (3) $val(y) = 0$ and $val(y') = 1$.

Let $z$ and $z'$ denote the configurations at the end of round $t$ of $r(y)$ and $r(y')$, respectively. Clearly, $r(z) = r(y)$, and therefore, $val(z) = val(y) = 0$, and similarly, $val(z') = 1$. For notational convenience, let us rename (a) $p_i$ as $q_1$, (b) for all

$j \in [1, i-1]$, $p_j$ as $q_{j+1}$, and (c) for all $j \in [i+1, n]$, $p_j$ as $q_j$. Thus $y$ and $y'$ differ only at $q_1$.

There are two cases to consider: (1) $q_1$ is alive in both $y$ and $y'$ and (2) $q_1$ has crashed in either $y$ or $y'$. (Note that $q_1$ has not crashed in both $y$ and $y'$ because $q_1$ has different states in the two configurations.)

*Case 1.* Process $q_1$ is alive in both $y$ and $y'$. Consider a series of round $t$ configurations $z_j$ ($j \in [0, n]$): (1) $z_0 = z$, and (2) for all $j \in [1, n]$, $z_j$ is identical to $z_0$ except that $q_1$ crashes in round $t$ such that, in round $t$, the messages from $q_1$ to processes in $\{q_1, ..., q_j\}$ are lost (and no other message is lost). Note that $val(z_0) = val(z) = 0$. We claim the following:

**Claim 28.1** For all $j \in [1, n]$, if $val(z_{j-1}) = 0$ then $val(z_j) = 0$.

Claim 28.1, immediately implies that $val(z_n) = 0$. (We give the proof of Claim 28.1 later.) Similarly, we construct another series of round $t$ configurations $z'_j$ ($j \in [0, n]$): (1) $z'_0 = z'$, and (2) for $j \in [1, n]$, $z'_j$ is identical to $z'_0$ except that $q_1$ crashes in round $t$ such that, in round $t$, the messages from $q_1$ to processes in $\{q_1, ..., q_j\}$ are lost (and no other message is lost). Note that $val(z'_0) = val(z') = 1$. We claim the following:

**Claim 28.2** For all $j \in [1, n]$, if $val(z'_{j-1}) = 1$ then $val(z'_j) = 1$.

Claim 28.2, immediately implies that $val(z'_n) = 1$. Now we observe that configurations $z_n$ and $z'_n$ are extensions of $y$ and $y'$, respectively, and in both $z_n$ and $z'_n$, no process receives a round $t$ message from $q_1$. Since $y$ and $y'$ differ only at $q_1$, no alive process can distinguish $z_n$ from $z'_n$. (As we assume $n \geq 3$ and $t < n/2$, we have $n - t > n/2 > 1$, and hence, there is a process which is alive in both $z_n$ and $z'_n$.) Thus, $val(z_n) = val(z'_n)$; a contradiction.

*Case 2.* Process $q_1$ has crashed in either $y$ or $y'$ Without loss of generality we can assume that $q_1$ has crashed in $y'$. As in Case 1, we can construct a series of round $t$ configurations $z_j$, for $j \in [0, n]$ (we can do so because $q_1$ is alive in $y$), and then show that $val(z_n) = 0$. Recall that $val(z') = 1$.

We now observe that in $z_n$, no process receives any round $t$ message from $q_1$. Furthermore, since $q_1$ has crashed in $y'$, it follows that, in $z'$, no process receives any round $t$ message from $q_1$. As $z_n$ and $z'$ are extensions of $y$ and $y'$, respectively, and $y$ and $y'$ differ only at $q_1$, so no correct process can distinguish $z_n$ from $z'$. Thus, $val(z_n) = val(z')$; a contradiction.

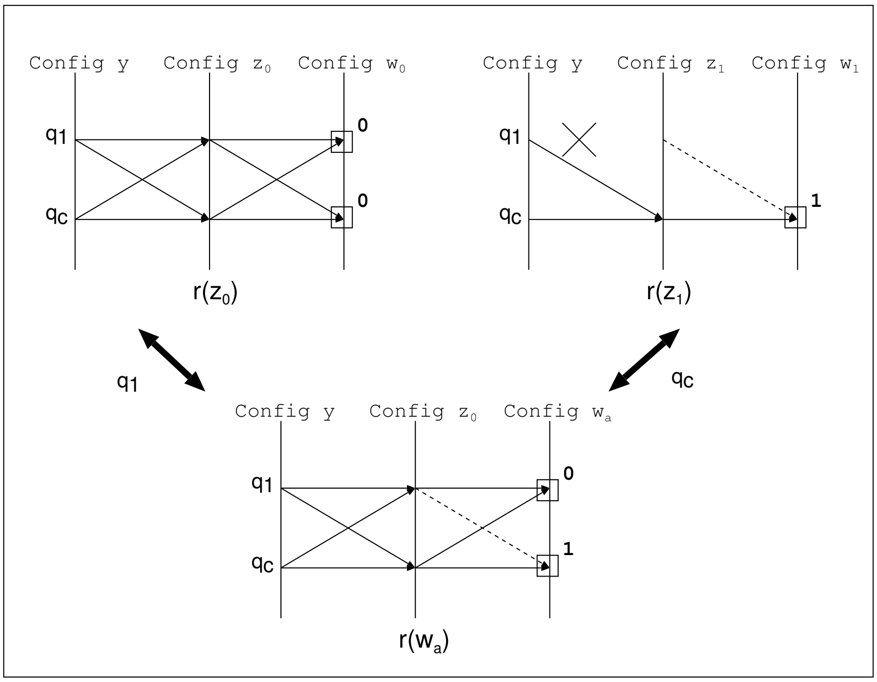


Figure 5.3: Rounds $t$ and $t+1$ of Case 1, Claim 28.1

We now give a proof of Claim 28.1. We omit the proof of Claim 28.2, which is similar. (We note that, in this proof, all runs constructed above are synchronous, but to prove Claim 28.1, we use some non-synchronous runs.)

**Proof of Claim 28.1** Suppose by contradiction that for some $j \in [1, n]$, $val(z_{j-1}) = 0$, and $val(z_j) = 1$. Configurations $z_{j-1}$ and $z_j$ differ only in the state of process $q_j$. There are two cases to consider: (1) $j = 1$ and (2) $j > 1$.

*Case 1.* $j = 1$. (See Figure 5.3.) Thus we have, $val(z_0) = 0$ and $val(z_1) = 1$. The round $t$ configurations $z_0$ and $z_1$ are identical at all processes except at $q_1$: $q_1$ is alive in $z_0$ but has crashed in $z_1$. As no process has crashed in round $t$ of $z_0$, and at most $t-1$ processes have crashed in the first $t-1$ rounds of $z_0$ (i.e., in configuration $y$), so a total of at most $t-1$ processes have crashed in $z_0$.

Let $w_0$ and $w_1$ be round $t+1$ configurations of $r(z_0)$ and $r(z_1)$, respectively. Recall that, it follows from our assumptions that $n - t > 1$, and hence, there is a process $q_c$ that is alive in $z_1$, and hence, correct in $r(z_1)$. As every process that is alive in $z_1$ is also alive in $z_0$, it follows that $q_c$ is correct in $r(z_0)$. As $q_1$ has crashed in $z_1$ but $q_c$ is alive in $z_1$, it follows that $q_c \neq q_1$.

As $r(z_0)$ and $r(z_1)$ are runs of algorithm $A$ in $SM_t$, they are synchronous runs of $A$ in $EM_t$. Therefore, from our assumption on $A$, correct processes decide by round $t+1$ in $r(z_0)$ and $r(z_1)$, i.e., correct processes have decided in configurations $w_0$ and $w_1$. Thus, $q_c$ decides $val(z_0) = 0$ in $w_0$, and $val(z_1) = 1$ in $w_1$. On the other hand, $q_1$ decides $val(z_0) = 0$ in $w_0$, and $q_1$ has crashed in $w_1$.

Now consider a round $t+1$ configuration $w_a$ that is a non-synchronous extension of $z_0$ in which (a) all round $t+1$ messages from $q_1$ to other processes are lost, (b) $q_1$ does not crash in round $t+1$ and receives the same messages that it receives in round $t+1$ of $w_0$. We note that (1) $q_1$ cannot distinguish round $w_a$ from $w_0$ because $q_1$ receives the same set of messages in round $t+1$ of both configurations, and (2) $q_c$ cannot distinguish $w_a$ from $w_1$ because $q_c$ does not receive round $t+1$ message from $q_1$ in both runs. Thus, in $w_a$, $q_1$ decides 0 and $q_c$ decides 1. Any run which extends $w_a$ violates uniform agreement.

*Case 2.* $j > 1$. (See Figure 5.4.) Thus we have, $val(z_{j-1}) = 0$ and $val(z_j) = 1$. The round $t$ configurations $z_{j-1}$ and $z_j$ are identical at all processes except $q_j$: $q_j$ receives round $t$ message from $q_1$ in $z_{j-1}$, and does not receive such a message in $z_j$. (Note that, $q_j$ is distinct from $q_1$ because $j > 1$.)

In the following, we construct five runs; the first two are synchronous and the remaining three are non-synchronous. Consider the first two synchronous runs $s^0$ and $s^1$ in which $q_j$ decides different values.

$s^0$: This run is the same as $r(z_{j-1})$. All correct processes decide $val(z_{j-1}) = 0$ by round $t+1$.

$s^1$: This run is the same as $r(z_j)$. All correct processes decide $val(z_j) = 1$ by round $t+1$.

We now construct three non-synchronous runs $a^2$, $a^0$, and $a^1$. In the constructions, we maintain the additional property that, in each round of each run, every process that completes that round, receives at least $n-t$ messages of that round. It is important to notice that, in the following three runs, we do not crash more than $t$ processes in each run: after round $t-1$, only $q_j$ crashes in each run.

$a^2$: This non-synchronous run is an extension of configuration $y$. The next two rounds are constructed as follows:

- round $t$: No process crashes in this round. Unlike $s^0$, $q_1$ does not crash in round $t$ of this configuration. But, every process distinct from $q_1$, receives the same set of messages as in round $t$ of $s^0$. (In other words, messages from $q_1$ to processes in $\{q_2, ..., q_{j-1}\}$ are lost.) Process $q_1$ receives messages from all processes that complete round $t$.

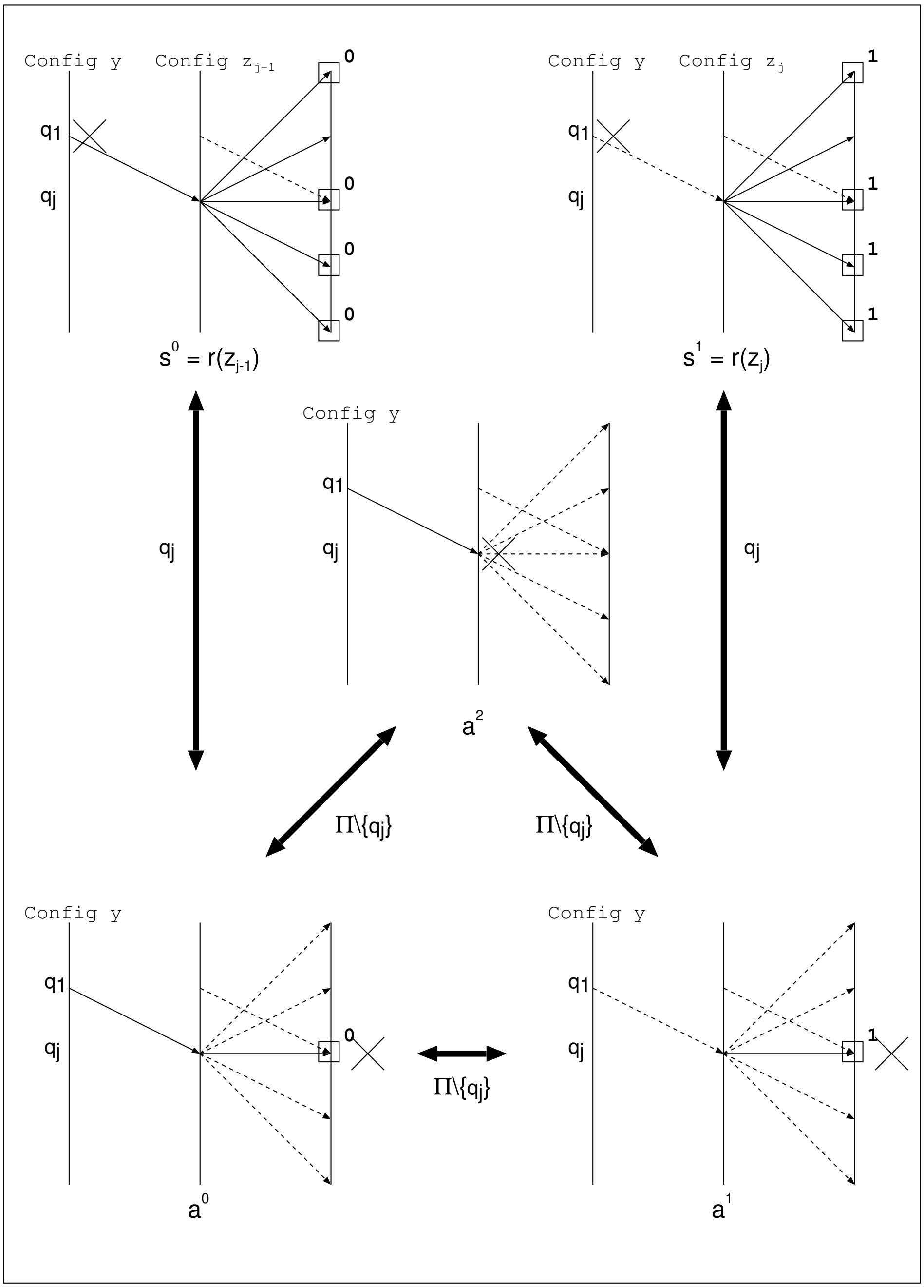


Figure 5.4: Rounds $t$ and $t + 1$ of Case 2, Claim 28.1

- rounds higher than $t$: Process $q_j$ crashes at the beginning of round $t+1$. No other process crashes and no message is lost in round $t+1$ or in a higher round. From the termination property, there is a round $k' \geq t+1$ such that $a^2$ reaches a global decision at $k'$.

Observations: (1) At the end of round $t$, only $q_1$ can distinguish $a^2$ from $s^0$. (2) At most $t-1$ processes have crashed in the first $t$ rounds of $a^2$: to see why, notice that no process crashes in round $t$ of $a^2$, the round $t-1$ configuration of $a^2$ is $y$, and at most $t-1$ processes have crashed in $y$.

$a^0$: This non-synchronous run is constructed as follows:

- first $t$ rounds: The first $t$ rounds $a^0$ are identical to those of $a^2$.
- rounds higher than $t$: Unlike in $a^2$, $q_j$ does not crash in round $t+1$. However, in round $t+1$, the messages from $q_j$ to all other processes are lost, and the message from $q_1$ to $q_j$ is lost. Also in round $t+1$, $q_j$ receives messages from all processes that complete round $t$, except $q_1$. Process $q_j$ crashes at the beginning of round $t+2$. In round $t+2$ and in higher rounds, no other process crashes and no message is lost.

We claim that $q_j$ cannot distinguish $a^0$ from $s^0$ at the end of round $t+1$. Notice that the first $t-1$ rounds of $s^0$, $a^2$, and $a^0$ are identical (round $t-1$ configurations of all three runs are $y$). At the end of round $t$, $a^2$ and $a^0$ are identical, and only $q_1$ can distinguish $a^2$ from $s^0$. Thus, at the end of round $t$, only $q_1$ can distinguish $a^0$ from $s^0$. Now consider the messages received by $q_j$ in round $t+1$ of $a^0$ and $s^0$. Process $q_j$ does not receive any message from $q_1$ in round $t+1$ ($q_1$ crashes before sending such a message in $s^0$, and the message is lost in $a^0$). As no process distinct from $q_1$ can distinguish $a^0$ from $s^0$ at the end of round $t$, so $q_j$ receives identical sets of messages in round $t+1$ of both $a^0$ and $s^0$. Thus $q_j$ cannot distinguish $a^0$ from $s^0$ at the end of round $t+1$, and hence, decides 0 at the end of round $t+1$.

$a^1$: This non-synchronous run is constructed as follows:

- first $t-1$ rounds: The first $t-1$ rounds of $a^1$ are identical to those of $s^1$ (i.e., configuration $y$).
- round $t$: No process crashes in this round. Unlike $s^1$, $q_1$ does not crash in round $t$ of $a^1$, but every process distinct from $q_1$, receives the same set of messages as in round $t$ of $s^1$. (In other words, messages from $q_1$ to processes in $\{q_2, ..., q_j\}$ are lost.) Process $q_1$ receives messages from all processes that complete round $t$.
- rounds higher than $t$: In round $t+1$, the messages from $q_j$ to all other processes are lost, and the message from $q_1$ to $q_j$ is lost. Also in round $t+1$, $q_j$ receives messages from all processes that complete round $t$, except

$q_1$. Process $q_j$ crashes at the beginning of round $t+2$. In round $t+2$ and in higher rounds, no other process crashes and no message is lost.

We make the following two claims.

- Process $q_j$ cannot distinguish $a^1$ from $s^1$ at the end of round $t+1$. Notice that, at the end of round $t$, only process $q_1$ can distinguish $a^1$ from $s^1$, and in round $t+1$, $q_j$ does not receive a message from $q_1$. Thus, in round $t+1$, $q_j$ receives identical sets of messages in $a^1$ and $s^1$. Thus $q_j$ cannot distinguish $a^1$ from $s^1$ at the end of round $t+1$, and hence, decides 1 at the end of round $t+1$.

- At the end of round $k'$, the processes distinct from $q_j$ cannot distinguish $a^2$, $a^0$, and $a^1$. (Round $k'$ is defined in the description of run $a^2$.) To see why, observe that the first $t-1$ rounds of the three runs are identical (i.e., configuration $y$). At the end of round $t$, the runs $a^2$, $a^0$ and $a^1$ differ only at process $q_j$: round $t$ message from $q_1$ to $q_j$ is received in $a^2$ and $a^0$, but lost in $a^1$. After round $t$, no process distinct from $q_j$ receives a message from $q_j$ because (1) $q_j$ crashes at the beginning of round $t+1$ in $a^2$, and (2) in $a^0$ and $a^1$, all round $t+1$ messages send from $q_j$ to other processes are lost, and $q_j$ crashes at the beginning of round $t+2$. Thus, the processes that are distinct from $q_j$ cannot distinguish the three runs at the end of round $k'$. As $a^2$ globally decides in round $k'$, so $a^1$ and $a^0$ also globally decide by round $k'$. We also note that the three runs have the same set of correct processes — round $t-1$ configuration of the three runs are identical, and only $q_j$ crashes after round $t-1$ in all three runs. Thus, there is a process which is correct in all three runs and decides the same value in these runs.

  Clearly, either $a^0$ or $a^1$ violates uniform agreement because $q_j$ decides 0 in $a^0$ and 1 in $a^1$; a contradiction.

□

We now state our lower bound on the number of rounds required for a global decision in synchronous runs of the eventually synchronous model.

**Lemma 29** $\forall t \in [1,(n-1)/2]$, $\forall f \in [0,t]$, *(*$\mathrm{EM}_t$*,* $\mathrm{SM}_f$*, UC, gd)* $\geq f+2$.

**Proof**: Suppose by contradiction that there is a UC algorithm $A$ in $EM_t$ and an integer $f$ in $[0,t]$ such that, in every run of $A$ in $SM_f$, all correct processes decide by round $f+1$. There are three cases to consider:

Case 1. $t \in [1,(n-1)/2]$, $f \in [1,t]$. As $EM_f$ is a submodel of $EM_t$, algorithm $A$ also solves UC in $EM_f$. As $f \in [1,(n-1)/2]$, we can replace $t$ by $f$ in Lemma 28.

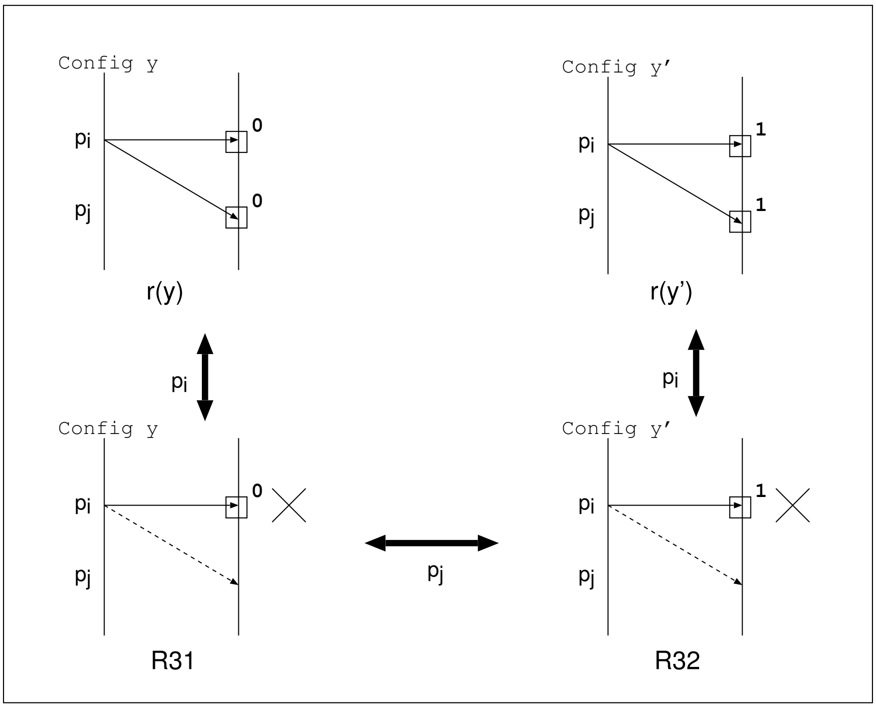


Figure 5.5: Round 1 of Case 3, Lemma 29

It follows that, there is a run of $A$ in $SM_f$ in which some correct process decides in round $f+2$ or in a higher round; a contradiction.

Case 2. $t \in [2, (n-1)/2]$, $f = 0$: As $SM_t$ is a submodel of $EM_t$, algorithm $A$ also solves UC in $SM_t$. Thus, from Theorem 3, there is run of $A$ in $SM_0$, in which some correct process decides in round 2 or in a higher round; a contradiction.

Case 3. $t = 1$, $f = 0$: Thus algorithm $A$ is a UC algorithm in $EM_1$ such that, in every failure-free synchronous run of $A$ all correct processes decides by round 1. Clearly, algorithm $A$ solves UC in $SM_1$. It follows from Lemma 6 that, there are two runs of $A$ in $SM_1$ such that their round 0 configurations (i.e., initial configurations), $y$ and $y'$, satisfy the following: (1) no process has crashed in each configuration, (2) the configurations differ at exactly one process, say $p_i$, and (3) $val(y) = 0$ and $val(y') = 1$. Obviously, $p_i$ is alive in both initial configurations $y$ and $y'$.

Runs $r(y)$ and $r(y')$ are failure-free runs of $A$ in $SM_1$, and hence, failure-free synchronous runs of $A$ in $EM_1$. From our assumption of $A$ it follows that, by round 1, all processes decides $val(y) = 0$ and $val(y') = 1$, in $r(y)$ and $r(y')$, respectively. Consider the following two non-synchronous runs $R31$ and $R32$. (See Figure 5.5.)

**R31** is a non-synchronous run whose (1) round 0 configuration is $y$, (2) in round 1, all messages sent by $p_i$ to processes distinct from $p_i$ are lost, and no other message is lost, (3) $p_i$ crashes at the beginning of round 2, and (4) no process distinct from $p_i$ crashes, and no message is lost in round 2 or a higher round. Process $p_i$ cannot distinguish round 1 configuration of $R31$ from that of $r(y)$, and therefore, $p_i$ decides $val(y) = 0$ in round 1 of $R31$.

**R32** is a non-synchronous run whose (1) round 0 configuration is $y'$, (2) in round 1, all messages sent by $p_i$ to processes distinct from $p_i$ are lost, and no other message is lost, (3) $p_i$ crashes at the beginning of round 2, and (4) no process distinct from $p_i$ crashes, and no message is lost in round 2 or a higher round. Process $p_i$ cannot distinguish round 1 configuration of $R32$ from that of $r(y')$, and therefore, $p_i$ decides $val(y') = 1$ in round 1 of $R32$.

Observe that, correct processes cannot distinguish $R32$ from $R31$, because (1) the round 0 configurations of the two runs differ only at $p_i$, and (2) no process distinct from $p_i$, ever receives a message from $p_i$. Thus, correct processes decide the same value in both runs. (Note that, there is at least one process $p_j$ that is correct in both runs because $t \leq n - 1$.) Either, $R31$ or $R32$ violates uniform agreement, because $p_i$ decides different values in the two runs. □

## 5.3 A matching algorithm

In this section, we present a UC algorithm $A_{em1}$ in $EM_t$ that matches the lower bounds of Lemma 29, i.e., the $f + 2$ rounds early global decision lower bound in synchronous runs. UC algorithms in $EM$ that match the early global decision bound for $f = 0$ appeared in [Sch97, MR99], but tight bound when $f > 0$ was unknown. Note that, since the early local decision lower bound is also $f + 2$ (for $f \in [0, t-3]$, Lemma 27), $A_{em1}$ also matches that bound. Algorithm $A_{em1}$ assumes that $t < n/2$, as dictated by Lemma 9.

### 5.3.1 Overview

Algorithm $A_{em1}$ (Figure 5.6) is inspired by the interactive consistency algorithm presented in Section 4.4, modified for exchanging and tracking false suspicions. The algorithm progresses in *sessions*, where each session is composed of $t + 2$ rounds. A run globally decides within $f + 2$ rounds in the first "synchronous" session, provided at most $f$ processes crash in the run. In each round of a session, the processes exchange their estimate of the decision value and the associated timestamp, and roughly speaking, adopt the estimate received with the maximum timestamp. (We

```
at process p_i:
initialize()
in round k                                                        {rounds 1, 2, ...}
    send round k messages
    receive messages
    compute()

procedure initialize()
    est_i ← prop_i; ts_i ← −i; Halt_i ← ∅; STATE_i ← SYNC1; msgSet_i ← ∅; λ_i ← 1
    commitTs_i ← 0; commitEst_i ← ⊥; round 1 msg ← (1, est_i, ts_i, STATE_i, Halt_i)

procedure compute()
    if dec_i ≠ ⊥ then
        if received any (k, est′, ts′, DECIDE, ∗) then
            est_i ← est′; ts_i ← ts′; dec_i ← est_i; STATE_i ← DECIDE   {decision DEC-1}
        else
            if STATE_i ∈ {SYNC1, SYNC2} then
                Halt_i ← Halt_i ∪                                             {HALT-1}
                    {p_j | (p_i received(k, ∗, ∗, NSYNC, ∗) from p_j) or            {HALT-2}
                        (p_i received(k, ∗, ∗, ∗, Halt_j) from p_j s.t. p_i ∈ Halt_j) or {HALT-3}
                        (p_i did not receive any round k message from p_j)}     {HALT-4}
                msgSet_i ← {m | m is a round k message received from p_j ∉ Halt_i}
                ts_i ← Max{ts | (k, ∗, ts, ∗, ∗) ∈ msgSet_i}
                est_i ← any est s.t. (k, est, ts_i, ∗, ∗) ∈ msgSet_i
                if (|Halt_i| ≤ t) and (STATE = SYNC2 for every message in msgSet_i) then
                    dec_i ← est_i; STATE_i ← DECIDE                          {decision DEC-2}
                else if (|Halt_i| ≤ λ_i − 1) and (λ_i < t + 2) then
                    STATE_i ← SYNC2; commitTs_i ← k; commitEst_i ← est_i          {commit}
                else if |Halt_i| ≤ t then
                    STATE_i ← SYNC1
                else                                                      {|Halt_i| > t}
                    STATE_i ← NSYNC
    {at the end of a session, update ts_i and est_i, and reset Halt_i and STATE_i}
    if (λ_i = t + 2) and (dec_i ≠ ⊥) then
        if committed in this session then {ts_i ← commitTs_i; est_i ← commitEst_i}
        Halt_i ← ∅; STATE_i ← SYNC1
    {generate the message for the next round}
    if λ_i = t + 2 then λ_i ← 1 else λ_i ← λ_i + 1          {step λ_i varies from 1 to t + 2}
    round k + 1 msg ← (k + 1, est_i, ts_i, STATE_i, Halt_i)
```

Figure 5.6: Uniform consensus algorithm $A_{em1}$

later discuss why we introduce the timestamps. At present, let us assume that the timestamp of an estimate does not change inside a session.) In this respect, a session of $A_{em1}$ is similar to the IC algorithm presented in Section 4.4: if the model was $SM_t$, then a process $p_i$ could simply monitor the set of processes from which $p_i$ did not receive any message (set $Halt_i$), and then, $p_i$ could decide on its own estimate when $Halt_i$ did not change for a round. Intuitively, $p_i$ could do so because in $SM_t$, $Halt_i$ is a superset of the set of processes that crashed in earlier rounds, and a subset of the set of processes that crashes by the current round. Hence, if $Halt_i$ did not change for a round, then $p_i$ would have the estimate with the maximum timestamp among all alive processes.

However, in $EM_t$, even if process $p_i$ does not receive a message from some process $p_j$, $p_j$ may be alive, and could continue sending messages in subsequent rounds. Thus, even if $Halt_i$ does not change in a round, $p_i$ might not have the estimate with the maximum timestamp among all alive processes. Therefore, in $A_{em1}$, in addition to the estimate value, the processes also exchange the $Halt$ sets to detect whether a session is synchronous. Furthermore, to enforce early decision, $p_i$ maintains and exchanges the variable $\text{STATE}_i$ which says whether $p_i$ considers the session to be synchronous ($\text{STATE}_i = \text{SYNC1}$), or $p_i$ considers the session to be synchronous and there is possibility of decision in the next round ($\text{STATE}_i = \text{SYNC2}$), or whether $p_i$ considers the session to be non-synchronous ($\text{STATE}_i = \text{NSYNC}$).

Early deciding algorithms that tolerate arbitrary periods of asynchrony [Sch97, MR99, HR99, GR04] typically require the processes to receive at least $n - t$ messages in a round, whereas in our model $EM_t$ any number of messages may be lost before $GSR$. To tolerate the scenario in which a process decides upon receiving $n-t$ messages in some round but another process does not receive any message up to that round, we use the timestamping scheme from [DLS88]. In $A_{em1}$, on selecting an estimate value that may be decided in the next round (i.e., in the round where STATE is updated to SYNC2), the processes timestamp the selected estimate to the current round number. Informally, this ensures that when a process decides upon receiving $n - t$ messages with STATE $=$ SYNC2, the decided value has the highest timestamp in the system, and it is present at $n - t$ processes. Subsequently, when another process adopts an estimate with the maximum timestamp, it adopts the decided value.

### 5.3.2 Description

The algorithm progresses in sessions, where a session $sn$ consists of $t + 2$ rounds from $((sn - 1) * (t + 2)) + 1$ to $sn * (t + 2)$. We call the $k^{th}$ round in a session $sn$ (i.e., round $((sn - 1) * (t + 2)) + k$) as *step* $k$ of session $sn$. We say that a session $sn$ in run $r$ is *synchronous* if the session starts in round $GSR(r)$ or in a higher round; i.e.,$(sn-1)*(t+2))+1 \geq GSR(r)$. Every process $p_i$ maintains the following variables:

1. $ts_i$ is the timestamp, $est_i$ is the estimate of the possible decision value, and $\lambda_i$ is the current step number at $p_i$;

2. STATE$_i$ reflects $p_i$'s view on how much progress is made towards achieving a decision in the current session: (1) if STATE$_i$ is updated to NSYNC then $p_i$ considers the session to be non-synchronous, (2) if STATE$_i$ is updated to SYNC1 then $p_i$ considers the session to be synchronous but $p_i$ cannot decide in the next round, (3) if STATE$_i$ is updated to SYNC2 then $p_i$ considers the session to be synchronous with the possibility of a decision in the next round, (4) and upon decision, $p_i$ updates STATE$_i$ to DECIDE;

3. $Halt_i$ is the set of processes $p_j$ such that, in the current step or a lower step of the current session, at least one of the following occurred: $p_i$ received STATE = NSYNC from $p_j$, $p_i$ did not receive a message from $p_j$, or $p_j$ did not receive a message from $p_i$;

4. $msgSet_i$ is the set of messages received by $p_i$ in the current step from processes that are not in $Halt_i$.

5. When $p_i$ sets its STATE to SYNC2, we say that $p_i$ *commits*. When $p_i$ commits, $commitTs_i$ and $commitEst_i$ are set to the current round number and the current estimate of $p_i$, respectively.

The algorithm can be very briefly described as follows. In each step of a session, the processes evaluate if the run is synchronous or non-synchronous based on their $Halt$ set. If the session turns out to be non-synchronous, then the processes do not decide in that session. Otherwise, in each step of that session, the processes exchange timestamp and estimate, and adopt the maximum timestamp seen, and the corresponding estimate. Consider any process $p_i$. If in some step, $p_i$ observes that $Halt_i$ is sufficiently small in size, then $p_i$ commits; i.e., sets it STATE to SYNC2, and updates $commitTs_i$ and $commitEst_i$. A process decides upon receiving $n - t$ message with STATE = SYNC2. Before moving to a new session, if $p_i$ has committed in the current session, then $p_i$ adopts the $commitTs_i$ and $commitEst_i$ as its new timestamp and estimate, respectively. It also sets $Halt_i$ and STATE to $\emptyset$ and SYNC1, respectively, so as to start the synchrony evaluation of the next session afresh.

We now discuss the algorithm in more details. The variables are initialized as follows. Variables STATE$_i$ and $Halt_i$ are initialized to SYNC1 and $\emptyset$ respectively, and are reset to their initial values at the beginning of each session. Variable $est_i$ is initialized to the proposal value, and $ts_i$ is initialized to $-i$ (to ensure that no initial timestamp has two different estimates associated with it). In each round, the processes exchange STATE, $est$, $ts$, and $Halt$ variables, update their own variables depending upon the messages received, and possibly decide. In step $\lambda$ of a session, $p_i$ updates its variables as follows.

1. If $p_i$ receives a DECIDE message, then $p_i$ decides on the decision value received (and we say that $p_i$ *decides at dec-1 in step* $\lambda$.)

2. If STATE$_i$ is SYNC1 or SYNC2 then:

- $p_i$ updates $Halt_i$ to include all processes that are already in $Halt_i$ (called condition halt-1), and any process $p_j$ such that: (1) $p_i$ has received an NSYNC message from $p_j$ in step $\lambda$ (condition halt-2), (2) $p_i$ received a message from $p_j$ with $p_i \in Halt_j$ in step $\lambda$ (condition halt-3), or (3) $p_i$ has not received any message from $p_j$ in step $\lambda$ (condition halt-4).
- $p_i$ includes in $msgSet_i$ every message received in step $\lambda$ whose sender is not in $Halt_i$. Then $p_i$ updates $ts_i$ to the maximum timestamp among messages in $msgSet_i$, and updates $est_i$ to any $est$ contained in a message with the maximum timestamp. (We do not care about non-determinism here because, as we will show later, messages with same timestamp have same estimate.)
- if $Halt_i$ is of at most size $t$, and all messages in $msgSet_i$ contains STATE $=$ SYNC2, then $p_i$ decides on its estimate (and we say that $p_i$ *decides at dec-2 in step* $\lambda$).
- Depending on the size $h$ of the set $Halt_i$, $p_i$ updates STATE$_i$ as follows: if $h$ is lower than the current step number, then STATE$_i$ is set to SYNC2, else if $h$ is at most $t$ then STATE$_i$ is set to SYNC1, otherwise, STATE$_i$ is set to NSYNC. If $p_i$ sets STATE$_i$ to SYNC2, then $p_i$ also updates $commitEst_i$ to the current estimate, and $commitTs_i$ to the current round number, and we say that $p_i$ *commits in step* $\lambda$. (It is important to note that $commitTs_i$ is updated to the current round number, and not to the current step number.)

3. At the end of a session; i.e., when $\lambda = t + 2$, $Halt_i$ and STATE$_i$ are reset to their initial values, and if $p_i$ has committed in the current session, then $ts_i$ and $est_i$ are updated to $commitTs_i$ and $commitEst_i$, respectively.

Observe that, inside a session, no new timestamp is attached to an estimate: timestamp and estimate are always adopted as a pair by a process. Only at the end of a session, a process may attach a new timestamp to an estimate, provided the process has committed in that session. The heart of the algorithm is ensuring that no two different estimates are committed in the same step, and any estimate committed in a step has the highest timestamp in that step. This, in turn, ensures that no timestamp is associated with two different estimates. As a process decides at dec-2 in step $\lambda$ only when it receives $n - t$ messages from processes that commit in step $\lambda - 1$, it follows that the decision value has the highest timestamp at the end of step $\lambda - 1$, and it is present at a majority of processes. Since in each step, the processes select estimates with highest timestamp, any estimate value selected in a later step is the decision value.

### 5.3.3 Correctness

The validity property of the algorithm follows from the following three simple observations: (1) the *est* value of a process is initialized to its proposal value, (2) the *est* value of a process at the beginning of round $k \geq 2$ is the *est* value of some process at the beginning of round $k-1$, and (3) every process decides on the *est* value of some process. In the rest of the section, we prove the uniform agreement property of the algorithm. We defer the proof of termination property to the next subsection, where we prove termination along with the time-complexity property of the algorithm.

For any given session, we consider the following notations. For any variable $val_i$ at process $p_i$, we denote by $val_i[\lambda]$ ($\lambda \geq 1$) the value of the variable $val_i$ immediately before line 27 in the compute procedure of step $\lambda$; $val_i[0]$ denotes the value of $val_i$ immediately before sending messages in step 1. For ease of presentation, we abuse the notation slightly and say that the lines 29 and 30 (which are actually executed in step $t+2$) are outside step $t+2$. We say that these two lines are executed *at the end of a session.*

We assume that there is a symbol *undefined* that is distinct from any possible value of the variables in the algorithm. If $p_i$ crashes before completing step $\lambda$, then $val_i[\lambda]$ is *undefined*; if $p_i$ crashes before sending messages in step 1, then $val_i[0]$ is *undefined.* For a process $p_l$ that has not decided by step $\lambda$, let $senderMS_l[\lambda]$ be the set of processes that have sent the messages that are in $msgSet_l[\lambda]$.

We now present some lemmas that help us to prove the uniform agreement property.

**Lemma 30** *Consider any session. If a process $p_l$ completes step $\lambda$ with* $\text{STATE}_l[\lambda] \in \{\text{SYNC}1, \text{SYNC}2\}$ *then* $senderMS_l[k] = \Pi - Halt_l[k]$.

**Proof**: If $p_l$ completes round $\lambda$ with $\text{STATE}_l[\lambda] \in \{\text{SYNC1}, \text{SYNC2}\}$ then it updates *msgSet* in step $\lambda$. The lemma follows from the condition halt-4 and the way $msgSet_l$ is updated. □

**Lemma 31** *Consider any session sn. For any process $p_l$ that completes step $\lambda$,* $p_l \notin Halt_l[\lambda]$.

**Proof**: Suppose by contradiction that there is a step $\lambda$ in session *sn* in which $p_l \in Halt_l[\lambda]$. Consider the lowest such step $\lambda'$. ($\lambda'$ cannot be 0 because $Halt_l[0] = \emptyset$.) Thus, $p_l \in Halt_l[\lambda']$ and $p_l \notin Halt_l[\lambda' - 1]$, and therefore, $p_l$ updated $Halt_l$ in step $\lambda'$. Thus, one of the four conditions, halt-1 to halt-4, holds for $p_l$ in step $\lambda'$. As

$p_l \notin Halt_l[\lambda' - 1]$, so conditions halt-1 and halt-3 cannot be true. If $p_l$ sends a NSYNC message to itself in step $\lambda'$, then its STATE is NSYNC at the end of step $\lambda' - 1$, and hence, $p_l$ does not update $Halt_l$ in step $\lambda'$ (thus, condition halt-2 cannot be true). Recall that, from the loopback property of $EM_t$, no process completes a round $k$ without receiving any round $k$ message from itself (thus, condition halt-4 cannot be true); a contradiction. □

**Lemma 32** *The timestamp of a process may decrease in a round only if it decides at dec-1 in that round.*

**Proof**: From the algorithm, once a process decides, its timestamp does not change (i.e., it does not update its timestamp). Suppose by contradiction that there is a round in which the timestamp of a process decreases and the process does not decide at dec-1 in that round. Consider any process $p_l$ that does not decides at dec-1 in some round $k$, and $ts_l[k-1] > ts_l[k]$. There are two cases to consider:

1. Process $p_l$ updates $ts_l$ to the maximum timestamp in $msgSet_l$ in round $k$. Then, from Lemma 30 and Lemma 31, $msgSet_l[k]$ contains a message from itself, and hence, the maximum timestamp in $msgSet_l[k]$ is at least $ts_l[k-1]$. Thus, $ts_l[k-1] \leq ts_l[k]$, a contradiction.

2. Round $k$ is the last round of a session, say $sn$, $p_l$ commits in session $sn$, and updates $ts_l$ to $commitTs_l$ at the end of the session. Note that, in a step of a session, if a process updates its timestamp, then it updates it to a timestamp received in some message in that step. Thus, in a step of any session, timestamp at any process is not higher than the highest timestamp at the beginning of the session. The timestamps at the beginning of a session, are round numbers from lower sessions. Thus, in a step of any session, the timestamp of a process is always some round number from a lower session. Now notice that, if $p_l$ commits in session $sn$, then $commitTs_l$ is set to a round number of the current session, and therefore, higher than any timestamp in a step of session $sn$. Thus, $ts_l$ increases when it is updated to $commitTs_l$ at the end of session $sn$; a contradiction.

□

**Lemma 33** *(Elimination) Consider any step $\lambda'$ of any session $sn$. If there are two processes $p_x$ and $p_y$ such that* $\text{STATE}_x[\lambda'] \in \{\text{SYNC}\mathit{1}, \text{SYNC}\mathit{2}\}$ *and* $\text{STATE}_y[\lambda'] = \text{SYNC}\mathit{2}$ *then* $ts_y[\lambda'] \geq ts_x[\lambda']$.

**Proof**: Suppose by contradiction that,

**Assumption A1**: $\text{STATE}_x[\lambda'] \in \{\text{SYNC1},\text{SYNC2}\}$, $\text{STATE}_y[\lambda'] = \text{SYNC2}$, $ts_x[\lambda'] = d$, $ts_y[\lambda'] = c$, and $ts_y[\lambda'] < ts_x[\lambda']$; i.e., $c < d$. (Note that, as $\text{STATE}_y[\lambda'] = \text{SYNC2}$, it follows that $p_y$ commits in step $\lambda'$. From the condition for committing, we have $\lambda' < t + 2$.)

We prove Claim 33.1 to Claim 33.7 based on assumption A1. Claim 33.4 contradicts Claim 33.7, thus proving Lemma 33 by contradiction. □

We first define the following sets for $\lambda \in [1, \lambda']$:

- $D[\lambda] = \{p_i | ts_i[\lambda] \geq d\}$ (The set of processes that complete step $\lambda$ with $ts \geq d$.)
- $crashed[\lambda]$ = the set of processes that crashed before completing step $\lambda$.
- $NSYN[\lambda] = \{p_i | \text{STATE}_i[k] = \text{NSYNC}$ **or** $\text{STATE}_i[k] = \text{DECIDE}\}$.
- $Z[\lambda] = D[\lambda] \cup crashed[\lambda] \cup NSYN[\lambda]$.

Additionally, we define $D[0]$ to be the set of processes that start step 1 with $ts$ at least $d$, $crashed[0]$ to be the set of processes which crash before sending any message in step 1, $NSYN[0]$ to be the set of processes that have decided in a lower session, and $Z[0] = D[0] \cup crashed[0] \cup NSYN[0]$. We make the following observation:

**Observation A2**: $|D[0]| \geq 1$, and hence, $|Z[0]| \geq 1$. Otherwise, if every process starts step 1 with a timestamp less than $d$, then $ts_x[\lambda'] < d$ (contradicts A1).

**Claim 33.1:** (1) $\forall \lambda \in [0, \lambda' - 1], (crashed[\lambda] \cup NSYN[\lambda]) \subseteq (crashed[\lambda + 1] \cup NSYN[\lambda + 1])$.
(2) $\forall \lambda \in [1, \lambda']$, if $p_i \notin (crashed[\lambda] \cup NSYN[\lambda])$ then $p_i$ sends messages with $\text{STATE} \in \{\text{SYNC1}, \text{SYNC2}\}$ in step $\lambda$, and in all steps lower than $\lambda$, of this session.

**Proof:** (1) Suppose by contradiction that there is a process $p_i$ such that $p_i \in crashed[\lambda] \cup NSYN[\lambda]$ and $p_i \notin crashed[\lambda + 1] \cup NSYN[\lambda + 1]$. Since a crashed process does not recover, $crashed[\lambda] \subseteq crashed[\lambda + 1]$, and hence, $p_i \notin crashed[\lambda + 1] \cup NSYN[\lambda + 1]$ implies that $p_i \notin crashed[\lambda]$. Thus, $p_i \in crashed[\lambda] \cup NSYN[\lambda]$ implies that $p_i \in NSYN[\lambda]$, i.e., $p_i$ completes step $\lambda$ with $\text{STATE} = \text{NSYNC}$ or $\text{STATE} = \text{DECIDE}$. If $p_i$ completes step $\lambda$ with $\text{STATE} = \text{NSYNC}$, then it cannot changes its state back to SYNC1 or SYNC2 in this session. If $p_i$ completes step $\lambda$ with $\text{STATE} = \text{DECIDE}$, then its state does not change thereafter. Thus, $p_i \in NSYN[\lambda]$ implies $p_i \in NSYN[\lambda + 1]$; a contradiction.

(2) If $p_i \notin (crashed[\lambda] \cup NSYN[\lambda])$ then, from Claim 33.1.1, it follows that $p_i \notin (crashed[\lambda_1] \cup NSYN[\lambda_1])$ for all $\lambda_1 \in [0, \lambda]$; i.e., $p_i$ completes every step lower than or equal to $\lambda$ with STATE $\neq$ NSYNC and STATE $\neq$ DECIDE. Thus $p_i$ has sent messages with STATE $\in$ {SYNC1, SYNC2} in step $k$ and in all lower steps. □

**Claim 33.2:** $\forall \lambda \in [0, \lambda' - 1], Z[\lambda] \subseteq Z[\lambda + 1]$.

**Proof:** Suppose by contradiction that there is a process $p_i$ and some $\lambda \in [0, \lambda' - 1]$ such that $p_i \in Z[\lambda]$ and $p_i \notin Z[\lambda+1]$. Since $p_i \notin Z[\lambda+1]$, then $p_i \notin crashed[\lambda+1] \cup NSYN[\lambda+1]$. Applying Claim 33.1.1, we get $p_i \notin crashed[\lambda] \cup NSYN[\lambda]$. However, $p_i \in Z[\lambda] = D[\lambda] \cup crashed[\lambda] \cup NSYN[\lambda]$, and hence, $p_i \in D[\lambda]$. Thus in round $\lambda + 1$, $p_i$ sends a message with $ts \geq d$.

As $p_i \notin crashed[\lambda + 1] \cup NSYN[\lambda + 1]$, $p_i$ updates its $ts$ in round $\lambda + 1$. From Lemma 30 and Lemma 31, the round $\lambda + 1$ message from $p_i$ is in $msgSet_i[\lambda + 1]$, and hence, the timestamp evaluated by $p_i$ in round $\lambda + 1$ is at least $d$. Thus, $p_i \in D[\lambda + 1] \subseteq Z[\lambda + 1]$; a contradiction. □

**Claim 33.3:** $\forall \lambda \in [0, \lambda' - 1], \forall p_i \notin Z[\lambda + 1], Z[\lambda] \subseteq Halt_i[\lambda + 1]$.

**Proof:** Consider a process $p_j \in Z[\lambda]$ and a process $p_i \notin Z[\lambda + 1]$. In step $\lambda + 1$, $msgSet_i[\lambda + 1]$ either contains a message from $p_j$ or does not contain any message from $p_j$. In the second case, Lemma 30 implies that $p_j \in Halt_i[\lambda + 1]$, and we are done. Consider the case where $msgSet_i[\lambda + 1]$ contains a message $m$ from $p_j$. Then from the way $msgSet$ is updated, $m$ has STATE $\notin$ {NSYNC, DECIDE}; i.e., $p_j \notin NSYN[\lambda]$. Furthermore, $p_j$ sent a message in step $k + 1$, and so, $p_j \notin crashed[\lambda]$. Thus $p_j \notin crashed[\lambda] \cup NSYN[\lambda]$, but we have assumed $p_j \in Z[\lambda]$. So, $p_j \in D[\lambda]$, and hence, $m$ contains $ts \geq d$. Since $m \in msgSet_i[\lambda+1]$, in step $k+1$, $p_i$ evaluates $ts_i$ to a value at least $d$. Thus $p_i \in D[\lambda+1] \subseteq Z[\lambda+1]$; a contradiction. □

**Claim 33.4:** $|Z[\lambda' - 1]| \leq \lambda' - 1$.

**Proof:** Suppose by contradiction that $|Z[\lambda' - 1]| > \lambda' - 1$. From Assumption A1, it follows that $p_y \notin Z[\lambda']$. Therefore, from Claim 33.3, $Z[\lambda' - 1] \subseteq Halt_y[\lambda']$. Hence, $|Halt_y[\lambda']| > \lambda' - 1$. However, STATE$_y[\lambda']$ = SYNC2 implies that $|Halt_y[\lambda']| \leq \lambda' - 1$ (from the condition for commit); a contradiction. □

**Claim 33.5:** $p_x \in Z[\lambda']$ and $p_x \notin Z[\lambda' - 2]$.

**Proof:** As $est_x[\lambda'] = d$, so $p_x \in D[\lambda'] \subseteq Z[\lambda']$.

For the second part of the claim, suppose by contradiction that $p_x \in Z[\lambda' - 2]$. Then, from Claim 33.3, for every process $p_i \in \Pi - Z[\lambda' - 1]$, $p_x \in Halt_i[\lambda' - 1]$. Therefore, in step $\lambda'$, for any message $m$ sent by a process in $\Pi - Z[\lambda' - 1]$, we have $p_x \in m.Halt$ (where, $m.Halt$ denotes the $Halt$ field of $m$). If $p_x$ receives $m$ in step $\lambda'$, then it includes the sender of $m$ in $Halt_x$ (because of condition halt-3). Moreover, if $p_i$ does not receive $m$ in step $\lambda'$, then $p_i$ includes the sender of $m$ in $Halt_x$ (because of condition halt-4). Thus $\Pi - Z[\lambda' - 1] \subseteq Halt_x[\lambda']$. Using, Claim 33.4, $|Halt_x[\lambda']| \geq |\Pi - Z[\lambda' - 1]| \geq n - (\lambda' - 1)$. Applying $\lambda' < t + 2$ (from Assumption A1) and $t < n/2$, we have $|Halt_x[\lambda']| \geq n - t > t$. However, $|Halt_x[\lambda']| > t$ implies that $\text{STATE}_x[\lambda'] = \text{NSYNC}$; a contradiction. □

**Claim 33.6:** (1) $\forall \lambda \in [0, \lambda' - 3], Z[\lambda] \subset Z[\lambda + 1]$. ($Z[\lambda]$ is a proper subset of $Z[\lambda + 1]$.)
(2) $|Z[\lambda' - 2]| \geq \lambda' - 1$.

**Proof:** (1) From Claim 33.2, $Z[\lambda] \subseteq Z[\lambda + 1]$ ($\lambda \in [0, \lambda' - 1]$). Suppose by contradiction that there is some $g \in [0, \lambda' - 3]$ such that $Z[g] = Z[g + 1]$. We first show the following result by induction on the step number $\lambda$:

**Result 33.6.1:** $\forall \lambda \in [g + 1, \lambda' - 1]$, $D[\lambda] - (NSYN[\lambda] \cup crashed[\lambda]) \supseteq D[\lambda + 1] - (NSYN[\lambda + 1] \cup crashed[\lambda + 1])$.

*Base Case ($k = g + 1$):* $D[g + 1] - (NSYN[g + 1] \cup crashed[g + 1]) \supseteq D[g + 2] - (NSYN[g + 2] \cup crashed[g + 2])$. Suppose by contradiction that there is a process $p_i$ such that $p_i \in D[g + 2] - (NSYN[g + 2] \cup crashed[g + 2])$ (**Assumption A3**), and $p_i \notin D[g + 1] - (NSYN[g + 1] \cup crashed[g + 1])$ (**Assumption A4**).

Assumption A3 implies that $p_i \notin NSYN[g + 2] \cup crashed[g + 2]$. Applying Claim 33.1.1, we have $p_i \notin NSYN[g + 1] \cup crashed[g + 1]$, and therefore, from Assumption A4, it follows that $p_i \notin D[g+1]$. Thus $p_i$ completes step $g+1$ with $ts < d$, $\text{STATE} \neq \text{NSYNC}$, and $\text{STATE} \neq \text{DECIDE}$ (i.e., with $\text{STATE} \in \{\text{SYNC1}, \text{SYNC2}\}$). Furthermore, Assumption A3 implies that $p_i$ completes step $g + 2$ with $ts \geq d$, $\text{STATE} \neq \text{NSYNC}$, and $\text{STATE} \neq \text{DECIDE}$ (i.e., with $\text{STATE} \in \{\text{SYNC1}, \text{SYNC2}\}$). So, $msgSet_i[g + 2]$ contains a message with $ts \geq d$ from some process $p_j$, i.e., $p_j \in senderMS_i[g + 2]$ (**Observation A5**). As $p_j$ sends a message with $ts \geq d$ in step $g+2$, it follows that $p_j \in D[g + 1] \subseteq Z[g + 1]$.

As $p_i \notin NSYN[g + 1] \cup crashed[g + 1]$ and $p_i \notin D[g + 1]$, so from the definition of $Z[g+1]$ we have $p_i \notin Z[g+1]$. Claim 33.3 implies that $Z[g] \subseteq Halt_i[g+1]$. Recall that we assumed $Z[g] = Z[g + 1]$ and, from condition halt-1, $Halt_i[g + 1] \subseteq Halt_i[g + 2]$. Therefore, $Z[g + 1] \subseteq Halt_i[g + 2]$. Thus $p_j \in D[g + 1] \subseteq Z[g + 1]$ implies that $p_j \in Halt_i[g + 2]$. From Observation A5, $p_j \in senderMS_i[g + 2] \cap Halt_i[g + 2]$.

As $p_i \notin NSYN[g+2] \cup crashed[g+2]$, so $p_i$ completed step $g+2$ with STATE $=$ SYNC1 or STATE $=$ SYNC2. Then, from Lemma 30, it follows that, $senderMS_i[g+2] \cap Halt_i[g+2] = \emptyset$. However, $p_j \in senderMS_i[g+2] \cap Halt_i[g+2]$; a contradiction.

*Induction Hypothesis:* $\forall \lambda \in [g+1, \rho]$, $D[\lambda] - (NSYN[\lambda] \cup crashed[\lambda]) \supseteq D[\lambda+1] - (NSYN[\lambda+1] \cup crashed[\lambda+1])$.

*Induction Step ($\lambda = \rho + 1$):* $D[\rho+1] - (NSYN[\rho+1] \cup crashed[\rho+1]) \supseteq D[\rho+2] - (NSYN[\rho+2] \cup crashed[\rho+2])$. Suppose by contradiction that there is a process $p_i$ such that $p_i \in D[\rho+2] - (NSYN[\rho+2] \cup crashed[\rho+2])$ (**Assumption A6**) and $p_i \notin D[\rho+1] - (NSYN[\rho+1] \cup crashed[\rho+1])$ (**Assumption A7**).

Similar to the base case, applying Assumptions A6, A7, and Claim 33.1, gives us $p_i \notin NSYN[\rho+2] \cup crashed[\rho+2]$, $p_i \notin NSYN[\rho+1] \cup crashed[\rho+1]$, and $p_i \notin D[\rho+1]$. Thus $p_i \notin Z[\rho+1]$. Since $g+1 < \rho+1$, from Claim 33.2, we have $Z[g+1] \subseteq Z[\rho+1]$, and therefore, $p_i \notin Z[g+1]$.

Applying Claim 33.3 on $p_i \notin Z[g+1]$ implies that $Z[g] \subseteq Halt_i[g+1]$.

Recall that we assumed $Z[g] = Z[g+1]$, and from condition halt-1 and the observation that $g+1 < \rho+2$, $Halt_i[g+1] \subseteq Halt_i[\rho+2]$. Therefore, $Z[g+1] \subseteq Halt_i[\rho+2]$ (**Observation A8**).

From the induction hypothesis, we have $(D[g+1] - (NSYN[g+1] \cup crashed[g+1])) \supseteq (D[\rho+1] - (NSYN[\rho+1] \cup crashed[\rho+1]))$. From the definition of $Z[g+1]$, $D[g+1] - (NSYN[g+1] \cup crashed[g+1]) \subseteq D[g+1] \subseteq Z[g+1]$, and therefore, $D[\rho+1] - (NSYN[\rho+1] \cup crashed[\rho+1]) \subseteq Z[g+1]$. Applying Observation A8, we have $(D[\rho+1] - (NSYN[\rho+1] \cup crashed[\rho+1])) \subseteq Halt_i[\rho+2]$ (**Observation A9**).

As $p_i \notin Z[\rho+1]$, $p_i$ completes step $\rho+1$ with $ts < d$, STATE $\neq$ NSYNC and STATE $\neq$ DECIDE. Furthermore, Assumption A6 implies that $p_i$ completes step $\rho+2$ with $ts \geq d$, STATE $\neq$ NSYNC and STATE $\neq$ DECIDE. Therefore, $msgSet_i[\rho+2]$ contains a message with $ts \geq d$ from some process $p_j$, i.e., $p_j \in senderMS_i[\rho+2]$ (**Observation A10**). As $p_j$ sends a message with $ts \geq d$ in step $\rho+2$, $p_j \in D[\rho+1] \subseteq Z[\rho+1]$.

As the step $\rho+2$ message of $p_j$ is in $msgSet_i[\rho+2]$, so from condition halt-2 it follows that the message sent by $p_j$ had STATE $\neq$ NSYNC. Moreover, if $p_i$ receives a message with STATE $=$ DECIDE, then it decides at dec-1. Therefore the message sent by $p_j$ in step $\rho+2$ had STATE $\neq$ DECIDE. It follows that $p_j \notin NSYN[\rho+1]$. As $p_j$ sends a message in step $\rho+2$, $p_j \notin crashed[\rho+1]$. Therefore, $p_j \in D[\rho+1] - (NSYN[\rho+1] \cup crashed[\rho+1])$. From Observation A9 it follows that $p_j \in Halt_i[\rho+2]$. From Observation A10, $p_j \in senderMS_i[\rho+2] \cap Halt_i[\rho+2]$.

As $p_i \notin NSYN[\rho+2] \cup crashed[\rho+2]$ (from Assumption A6), so $p_i$ completed step $\rho+2$ with STATE $=$ SYNC1 or STATE $=$ SYNC2. Lemma 30 implies that

$senderMS_i[\rho+2] \cap Halt_i[\rho+2] = \emptyset$. However, $p_j \in senderMS_i[\rho+2] \cap Halt_i[\rho+2]$; a contradiction. (End of the proof of Result 33.6.1.)

From the above Result 33.6.1, we have $D[\lambda'-2] - (NSYN[\lambda'-2] \cup crashed[\lambda'-2]) \supseteq D[\lambda'] - (NSYN[\lambda'] \cup crashed[\lambda'])$. From Assumption A1, $p_x \in D[\lambda'] - (NSYN[\lambda'] \cup crashed[\lambda']))$. From Claim 33.5, we have $p_x \notin Z[\lambda'-2] \supseteq (D[\lambda'-2] - (NSYN[\lambda'-2] \cup crashed[\lambda'-2])$. In other words, $p_x$ is in $D[\lambda'] - (NSYN[\lambda'] \cup crashed[\lambda'])$ but not in $D[\lambda'-2] - (NSYN[\lambda'-2] \cup crashed[\lambda'-2])$; a contradiction.

(2) Part (1) of this claim implies that, for every $\lambda \in [0, \lambda'-3]$, $|Z[\lambda+1]| - |Z[\lambda]| \geq 1$. From Observation A2, $|Z[0]| \geq 1$. Therefore, $|Z[\lambda'-2]| \geq \lambda'-1$. □

**Claim 33.7:** $|Z[\lambda'-1]| > \lambda'-1$.

**Proof:** Suppose by contradiction that $|Z[\lambda'-1]| \leq \lambda'-1$. Since $Z[\lambda'-2] \subseteq Z[\lambda'-1]$ (Claim 33.2) and $|Z[\lambda'-2]| \geq \lambda'-1$ (Claim 33.6.2), we have $|Z[\lambda'-2]| = |Z[\lambda'-1]| = \lambda'-1$, and therefore, $Z[\lambda'-2] = Z[\lambda'-1]$ (**Assumption A11**).

From Claim 33.5, we know that $p_x \notin Z[\lambda'-2] = Z[\lambda'-1]$. Applying Claim 33.3, we have $Z[\lambda'-2] \subseteq Halt_x[\lambda'-1]$. As $Z[\lambda'-2] = Z[\lambda'-1]$ (from Assumption A11), it follows that $Z[\lambda'-1] \subseteq Halt_x[\lambda'-1]$.

Since $p_x \notin Z[\lambda'-1]$, $p_x$ completes step $\lambda'-1$ with $ts < d$, STATE $\neq$ NSYNC and STATE $\neq$ DECIDE. From Assumption A1, we also know that $p_x$ completes step $\lambda'$ with $ts \geq d$, STATE $\neq$ NSYNC and STATE $\neq$ DECIDE. Therefore, $msgSet_x[\lambda']$ contains a message, say from process $p_j$, with $ts \geq d$, i.e., $p_j \in senderMS_x[\lambda']$. From the definition of $D[\lambda'-1]$, $p_j \in D[\lambda'-1] \subseteq Z[\lambda'-1]$. Recall that we showed that $Z[\lambda'-1] \subseteq Halt_x[\lambda'-1]$, and from condition halt-1, it follows that $Halt_x[\lambda'-1] \subseteq Halt_x[\lambda']$. Thus $Z[\lambda'-1] \subseteq Halt_x[\lambda']$, and hence, $p_j \in Halt_x[\lambda']$.

From Assumption A1, we know that $p_x$ completed step $\lambda'$ with STATE $=$ SYNC1 or STATE $=$ SYNC2. Therefore, Lemma 30 implies that $senderMS_x[\lambda'] \cap Halt_x[\lambda'] = \emptyset$. However, $p_j \in senderMS_x[\lambda'] \cap Halt_x[\lambda']$; a contradiction. □

**Tsval.** We define the *tsval* of a message $m$ to be the ordered pair ($m.ts$, $m.est$), where $m.ts$ is the timestamp and $m.est$ is the estimate of message $m$. We say that a process sends tsval $tv$ in a step $\lambda$, if the process sends a message containing the same timestamp and estimate as $tv$, in step $\lambda$. For a process $p_i$ and step $\lambda \geq 0$ of some session, we define $tsval_i[\lambda]$ to be the ordered pair ($ts_i[\lambda]$, $est_i[\lambda]$). An *initial tsval* is $tsval[0]$ of some process in the first session.

**Consistent tsvals and messages.** We say that two tsvals are *consistent* if either (1) the two tsvals have different timestamp, or (2) the two tsvals have the same timestamp and the same estimate. We say that two tsvals are *inconsistent* if they have the same timestamp but different estimates. Clearly, any two tsvals are either consistent or inconsistent. Two messages are consistent (inconsistent) if their tsvals are consistent (resp. inconsistent).

**Lemma 34** *Consider any session sn. For any process $p_i$ that completes some step $\lambda \geq 1$ with $tsval_i[\lambda] = tv$, there is a process $p_j$ that sent tsval tv in step 1.*

**Proof**: Observe that, if a process completes a step $\lambda$, it either retains its own tsval at the end of step $\lambda - 1$, or adopts one from some message $m$ of step $\lambda$. Message $m$, in turn, contains tsval of some process at the end of step $\lambda - 1$, or if $\lambda = 1$, $m$ is a step 1 message. The lemma follows from a trivial backward induction. (Note that, if a process commits in a session, then the process generates a new *tsval* at the end of the session by updating *ts* and *est* to *commitTs* and *commitEst*, respectively.) □

**Lemma 35** *Consider any session sn. If any two step 1 messages of session sn are consistent, then any two messages of session sn are consistent.*

**Proof**: If a process $p_i$ sends a message with some tsval $tv$ in a step $\lambda > 1$, then $tsval_i[\lambda - 1] = tv$. It follows from Lemma 34 that some process has sent a step 1 message with tsval $tv$. The lemma follows immediately. □

**Lemma 36** *Any two messages sent with negative timestamps are consistent.*

**Proof**: Suppose by contradiction that there is some $ts' < 0$ such that two inconsistent messages are sent with timestamp $ts'$, say with tsvals $tv1$ and $tv2$.

Observe that, apart from the initial tsvals of the processes, a process generates a new tsval only if it commits. (If a process commits in a session, then it generates a new tsval at the end of that session.) Moreover, if a process commits in a session, the new tsval generated by the process at the end of the session, has a timestamp equal to a round number of that session. Therefore, any tsval that is generated by a process upon committing has a positive timestamp.

As both $tv1$ and $tv2$ have timestamp $ts' < 0$, it follows that the two tsvals are initial tsvals. Since every process has a distinct initial timestamp, no two initial tsvals are inconsistent; a contradiction. □

**Lemma 37** *If two messages have the same timestamp, then they have the same estimate.*

**Proof**: The statement of the lemma is equivalent to saying that, any two messages are consistent. Suppose by contradiction that there are two inconsistent messages. Consider the lowest timestamp $ts'$ such that two inconsistent messages were sent with timestamp $ts'$. Let $tv1$ and $tv2$ be the two different tsvals which were sent with timestamp $ts'$. From Lemma 36, we have $ts' > 0$.

Consider the lowest round $k1$ in which some process sent $tv1$, say process $p_a$. Notice that round $k1$ is necessarily step 1 of some session: otherwise, $tv1$ was either adopted by $p_a$ from a round $k1-1$ message, or $p_a$ did not update its tsval in round $k1-1$ (in the second case $p_a$ sent $tv1$ in round $k1-1$); both cases contradict the definition of $k1$.

If $k1$ is the step 1 of the first session, then $ts' < 0$ because $p_a$ initializes its timestamp to $-a$; a contradiction. Thus $k1$ is step 1 of some session $sn1+1 \geq 2$, and therefore, $p_a$ generates $tv1$ at the end of session $sn1$. It follows that $p_a$ commits in a step of session $sn1$, and that step corresponds to round $ts'$.

Using a similar argument, we know that there is another process $p_b$ that generates $tv2$ at the end of some session $sn2$, and therefore, $p_b$ commits in a step of session $sn2$, and that step corresponds to round $ts'$. As round $ts'$ corresponds to a step of both session $sn1$ and session $sn2$, we have $sn1 = sn2$.

Let $ts'$ be step $\lambda'$ of session $sn1$. Then, $p_a$ commits in step $\lambda'$ with estimate $tv1.est$. So $\text{STATE}_a[\lambda'] = \text{SYNC}2$ and $est_a[\lambda'] = tv1.est$. Similarly, $\text{STATE}_b[\lambda'] = \text{SYNC}2$ and $est_b[\lambda'] = tv2.est$. (Also note that, from the condition for committing, $\lambda' < t+2$.)

Applying Lemma 33, with $p_x = p_a$ and $p_y = p_b$, gives $ts_b[\lambda'] \geq ts_a[\lambda']$. Applying Lemma 33 again, with $p_x = p_b$ and $p_y = p_a$, gives $ts_a[\lambda'] \geq ts_b[\lambda']$. Thus $ts_a[\lambda'] = ts_b[\lambda']$.

Observe that every message sent in step 1 of session $sn1$ has a timestamp that is equal to a round number from a session lower than $sn1$ (or has a timestamp that is less than 0 if $sn1$ is the first session). Thus every timestamp sent in step 1 is lower than $ts'$ (because $ts'$ is a round number of session $sn1$). As $ts'$ is the lowest timestamp such that two inconsistent tsvals were sent with that timestamp, any two tsvals sent in step 1 of $sn1$ are consistent. Thus, from Lemma 35, in session $sn1$, any two messages are consistent. In step $\lambda'+1$ of $sn'$, (1) process $p_a$ sends a message with tsval $(ts_a[\lambda'], tv1.est)$, (2) process $p_b$ sends a message with tsval $(ts_b[\lambda'], tv2.est)$, (3) $ts_a[\lambda'] = ts_b[\lambda']$, and (4) any two messages in session $sn1$ are consistent. It follows that $tv1.est = tv2.est$; a contradiction. □

**Lemma 38** *Any pair of* SYNC*2 message sent in the same round have the same timestamp and the same estimate.*

**Proof**: Consider two processes $p_a$ and $p_b$ that send SYNC2 messages in some round $k'$, say step $\lambda'$ of session $sn'$. As every step 1 message is a SYNC1 message, $\lambda' >$

1. Applying Lemma 33, with $p_x = p_a$ and $p_y = p_b$, gives $ts_b[\lambda' - 1] \geq ts_a[\lambda' - 1]$. Applying Lemma 33, with $p_x = p_b$ and $p_y = p_a$, gives $ts_a[\lambda' - 1] \geq ts_b[\lambda' - 1]$. Thus $ts_a[\lambda' - 1] = ts_b[\lambda' - 1]$. Thus the two SYNC2 messages have the same timestamp. Applying Lemma 37, it follows that the two SYNC2 messages have the same estimate. □

Consider any session $sn'$. For any step $\lambda > 0$ of $sn'$, let $hts[\lambda] = \mathbf{Max}\{ts_i[\lambda] | \text{STATE}_i[\lambda] \in \{\text{SYNC1}, \text{SYNC2}\}\}$. $hts[0]$ is the maximum $ts$ at the beginning of session $sn'$ over all processes that enter session $sn'$.

**Lemma 39** *Consider any session $sn$. For $\lambda \in [0, t+1]$, $hts[\lambda] \geq hts[\lambda + 1]$.*

**Proof**: Suppose by contradiction that there is a process $p_i$ that completes step $\lambda+1$ of session $sn$ with $\text{STATE}_i[\lambda+1] \in \{\text{SYNC1}, \text{SYNC2}\}$, and with timestamp higher than $hts[\lambda]$. Then, $p_i$ has updated its timestamp in step $\lambda + 1$, and so, $msgSet_i[\lambda + 1]$ contains a message $m$ with a timestamp higher than $hts[\lambda]$. From conditions halt-2, $m$ has STATE $\neq$ NSYNC. Also, $m$ has STATE $\neq$ DECIDE, otherwise, $p_i$ decides at dec-1 on receiving $m$. Thus, at the end of step $\lambda$, the sender of $m$ has STATE $\in$ {SYNC1, SYNC2} and a timestamp higher than $hts[\lambda]$; a contradiction. □

**Lemma 40** *(Uniform agreement) No two processes decide differently.*

**Proof**: If no process ever decides then the lemma is trivially true. Suppose some process decides in a run. Consider the lowest round in which some process decides, say $k'$. Let round $k'$ be step $\lambda'$ of session $sn'$. Let $p_c$ be a process that decides in round $k'$, say on value $x$. Note that, if $p_c$ decides at dec-1 in round $k'$, then some process has sent a DECIDE message in round $k'$, and therefore, some process has decided in a round lower than $k'$; a contradiction. Therefore, $p_c$ decides at dec-2 in round $k'$.

Consider the set $msgSet_c[\lambda']$ and $senderMS_c[\lambda']$. From the condition for deciding at dec-2 it follows that, $msgSet_c[\lambda']$ has at least $n - t$ messages, and all those messages have STATE = SYNC2. From Lemma 30 and Lemma 31, we know that $p_c \in senderMS_c[\lambda']$. From Lemma 38, we know that all messages in $msgSet_c[\lambda']$ have the same estimate and the same timestamp. Since $p_c$ decides on one of the estimates in $msgSet_c[\lambda']$ (namely, the one sent by itself), all messages in $msgSet_c[\lambda']$ have estimate $x$. Let $ts'$ be the timestamp of the messages in $msgSet_c[\lambda']$. It follows from Lemma 33 that every process that completes step $\lambda' - 1$ either has $ts \leq ts'$ or STATE = NSYNC (we call this **Observation B1**). (No process has STATE = DECIDE at the end of step $\lambda' - 1$ because no process decides before step $\lambda'$.) Let us denote $senderMS_c[\lambda']$ by $S_c$. Note that, as $msgSet_c[\lambda']$ has at least $n - t$ messages, so $S_c$ contains at least $n - t$ processes.

We prove uniform agreement through two claims. We show that (1) no process can send a SYNC2 message with an estimate value different from $x$ in steps higher than $\lambda' - 1$ in session $sn'$ and (2) no process can send a SYNC2 message with an estimate value different from $x$ in any step of a session higher than $sn'$. The two claims imply uniform agreement (i.e., no process decides a value different from $x$) because: (1) no process decides before step $\lambda'$ of session $sn'$, (2) if a process decides $y$ at dec-2 in some round then it sends a SYNC2 message with estimate $y$ in that round, and (3) if a process decides $y$ at dec-1 in some round, then some process has decided $y$ at dec-2 of a lower round. We now show the two claims.

**Claim 40.1.** No process can send a SYNC2 message with an estimate value different from $x$ in steps higher than $\lambda' - 1$ of session $sn'$. From Lemma 38, any message sent in step $\lambda'$ has the same estimate value as that of the messages contained in $msgSet_c[\lambda']$, i.e., an estimate value $x$. Therefore, we consider SYNC2 messages in steps $\lambda$ such that $\lambda \geq \lambda' + 1$.

Consider any process $p_j$ that sends a SYNC2 message in a step $\lambda \geq \lambda' + 1$. By definition of $hts[\lambda-1]$, $hts[\lambda-1] \geq ts_j[\lambda-1]$. From Lemma 39, $hts[\lambda'-1] \geq hts[\lambda-1]$. Moreover, from observation B1, $hts[\lambda' - 1] = ts'$. Therefore, $ts_j[\lambda - 1] \leq ts'$.

As process $p_j$ sends a SYNC2 message in step $\lambda$, so there is at least $n - t$ messages in $msgSet_j[\lambda - 1]$, and therefore, $msgSet_j[\lambda - 1]$ contains at least one message from some process $p_l$ in $S_c$ (because $S_c$ contains at least $n - t$ processes).

As $p_l \in S_c$, it follows from the definition of $ts'$, $ts_l[\lambda' - 1] = ts'$. From Lemma 32, $ts_l[\lambda - 2] \geq ts_l[\lambda' - 1]$ (because $\lambda - 2 \geq \lambda' - 1$ and if $p_l$ decides by round $\lambda - 2$ then $p_j$ decides in round $\lambda - 1$ upon receiving the message from $p_l$), and therefore, $ts_l[\lambda-2] \geq ts'$. It follows that the step $\lambda-1$ message from $p_l$ has timestamp at least $ts'$. Since $msgSet_j[\lambda - 1]$ contains the step $\lambda - 1$ message from $p_l$, the timestamp evaluated by $p_j$ in step $\lambda - 1$ is at least $ts'$; i.e., $ts_j[\lambda - 1] \geq ts'$. As we have already shown $ts_j[\lambda-1] \leq ts'$, it follows that, $ts_j[\lambda-1] = ts'$. Thus, the step $\lambda$ message from $p_j$ contains timestamp $ts'$, and from Lemma 37, that message contains estimate $x$.

**Observation B2.** (Recall that round $k'$ is the step $\lambda'$ of session $sn'$.) From Claim 40.1, at the end of session $sn'$, every process that updates its timestamp to $commitTs$ either (1) has $commitTs \geq k' - 1$ and $commitEst = x$ (and therefore, goes to the next session with timestamp at least $k' - 1$ and estimate $x$), or (2) has $commitTs < k' - 1$ (and therefore, goes to the next session with timestamp less than $k' - 1$). Furthermore, at the end of session $sn'$, every process in $S_c$ is either crashed, has decided, or updates its $ts$ to $commitTs \geq k' - 1$ (because the processes in $S_c$ update their $commitTs$ to $k' - 1$ in step $\lambda' - 1$ of $sn'$). Thus any process in $S_c$ that enters session $sn' + 1$ has timestamp at least $k' - 1$ and estimate $x$.

**Claim 40.2.** No process can send a SYNC2 message with an estimate value different from $x$ in any step of a session higher than $sn'$. We prove this part by induction on session numbers.

*Base Case.* Session number $sn' + 1$. Note that, in step 1 of any session, no process sends a message with STATE $=$ SYNC2. Consider any process $p_j$ that sends a SYNC1 or SYNC2 message in step 2 of $sn' + 1$. Process $p_j$ must have completed step 1 with STATE $\in \{$SYNC1, SYNC2$\}$. Thus, $msgSet_j[1]$ contains at least $n - t$ messages, and hence, contains at least one message from a process in $S_c$. The message in $msgSet_j[1]$ from a process in $S_c$ contains a timestamp at least $k' - 1$ (from Observation B2). Thus the maximum timestamp in $msgSet_j[1]$ is at least $k' - 1$, and therefore, (again from Observation B2) the estimate contained in the message with maximum timestamp (in $msgSet_j[1]$) is $x$. Thus, $est_j[1] = x$ and $ts_j[1] \geq k' - 1$, and hence, every message sent in step 2 with STATE $\in \{$SYNC1, SYNC2$\}$ has $est = x$. It follows that, no SYNC1 or SYNC2 message, with an estimate different from $x$, can be sent in a step higher than 1 of session $sn' + 1$.

*Induction Hypothesis.* For every session $sn$ such that $sn' + 1 \leq sn \leq sn''$, every SYNC2 message sent in session $sn$ contains estimate $x$.

> **Observation B3.** At the end of session $sn''$, any tsval that has a timestamp at least $k' - 1$, has been generated by a process that has committed in session $sn \geq sn'$ (because $k' - 1$ is a round of session $sn'$). From observation B2 we know that, at the end of session $sn'$, every process with a timestamp at least $k' - 1$ has estimate $x$. For every $sn$ such that $sn' + 1 \leq sn \leq sn''$, any process that commits in a step $\lambda$ of session $sn$, sends a SYNC2 message $m$ in step $\lambda + 1$. From the induction hypothesis, $m$ contains estimate $x$. Thus any process that commits in session $sn$, commits with estimate $x$. Thus, at the end of session $sn''$, any process that has a timestamp at least $k' - 1$, has estimate $x$. Furthermore, from Lemma 32 and observation B2, at the end of session $sn''$, every process in $S_c$ is either crashed, has decided, or has timestamp at least $k' - 1$.

*Induction step.* Every SYNC2 message sent in session $sn'' + 1$ contains estimate $x$. Consider session $sn'' + 1$. Note that, in step 1 of any session, no process sends a message with STATE $=$ SYNC2. Consider any process $p_j$ that sends a SYNC1 or SYNC2 message in step 2 of $sn'' + 1$. Process $p_j$ must have completed step 1 with STATE $\in \{$SYNC1, SYNC2$\}$. Thus, $msgSet_j[1]$ contains at least $n - t$ messages, and hence, contains at least one message from a process in $S_c$. The message in $msgSet_j[1]$ from a process in $S_c$ contains a timestamp at least $k' - 1$ (from Observation B3). Thus the maximum timestamp in $msgSet_j[1]$ is at least $k' - 1$, and therefore, (again from Observation B3) the estimate contained in the message with maximum timestamp in $msgSet_j[1]$ is $x$. Thus, $est_j[1] = x$ and $ts_j[1] \geq k' - 1$, and hence, every message sent in step 2 with STATE $\in \{$SYNC1, SYNC2$\}$ has $est = x$. It follows that, no SYNC1 or SYNC2 message, with an estimate different from $x$, can be sent in a step higher than 1 of session $sn'' + 1$. □

### 5.3.4 Time-complexity

We now discuss the termination and the time-complexity properties of the algorithm. Fix any run $r$, and consider the lowest synchronous session $sn$, i.e., the first session that starts at round $GSR(r)$ or at a higher round. Let $f \in [0, t]$ be the number of processes that crash in run $r$.

**Lemma 41** *Consider any process $p_i$ that completes step $\lambda \in [1, t+2]$ of session $sn$. Every process in $Halt_i[\lambda]$ has crashed by step $\lambda$.*

**Proof**: For any step $l \in [0, \lambda]$ in $sn$, let $H[l]$ be the union of all $Halt_j[l]$ such that $Halt_j[l] \neq$ *undefined*. The following claim immediately implies the lemma: *every process in $H[l]$ has crashed by step $l$.*

We prove the claim by induction on the step number $l$. For $l = 0$, the claim is trivially true, because the processes update $Halt$ to $\emptyset$ at the beginning of every session, and so, $H[0] = \emptyset$ (base case). Suppose that the claim is true for all $l \in [0, l' - 1]$: every process in $H[l]$ has crashed by step $l$ (induction hypothesis). Consider the set $H[l']$ (induction step). If $H[l'] - H[l' - 1] = \emptyset$ then the induction step is trivial. Suppose by contradiction that there is a process $p_j \in H[l'] - H[l' - 1]$ such that $p_j$ has not crashed by step $l'$. Thus there is a process $p_a$ such that $p_j \notin Halt_a[l' - 1]$ and $p_j \in Halt_a[l']$. Thus $p_a$ updates $Halt_a$ in step $l'$. Note that, $p_a$ has not decided by step $l' - 1$ or at dec-1 in step $l'$, otherwise, it does not update $Halt_a$ in step $l'$.

Consider step $l'$ at $p_a$. As $p_j \in Halt_a[l' - 1]$, condition halt-1 is false. As $p_j$ has not crashed by step $l'$, and $sn'$ is a synchronous session, so $p_a$ must have received the step $l'$ message $m$ from $p_j$. Thus condition halt-4 is false. Since, $p_j \in Halt_a[l']$, $m$ contains either (a) STATE $=$ NSYNC (condition halt-2) or (b) set $Halt_j$ such that $p_a \in Halt_j$ (condition halt-3). We show both cases to be impossible and thus prove the induction step by contradiction.

From our induction hypothesis, for every step $l < l'$, every process in $Halt_j[l]$ has crashed by step $l$. Since no more than $t$ processes can crash in a run, in rounds lower than $l'$, $|Halt_j|$ is never higher than $t$. Thus $p_j$ can not update its STATE to NSYNC in rounds lower than $l'$. Thus the round $l'$ message from $p_j$ does nor contain STATE $=$ NSYNC.

If the round $l'$ message from $p_j$ contains $Halt_j$ such that $p_a \in Halt_j$ then $p_a \in Halt_j[l' - 1] \subseteq H[l' - 1]$. However, from our induction hypothesis, every process in

$H[l'-1]$ crashes before completing round $l'-1$, which implies that $p_a$ crashes before completing round $l'-1$; a contradiction. □

**Lemma 42** *Every correct process in run $r$ decides by step $f+2$ of session $sn$.*

**Proof**: Suppose by contradiction that there is a correct process $p_i$ that does not decide by step $f+2$ of session $sn$. If some correct process decides before step $f+2$, then by step $f+2$, every process receives a DECIDE message, and decides. Therefore, from our assumption, no correct process decides before step $f+2$ in $r$.

Since $f$ processes crash in $r$, from Lemma 41, $|Halt|$ at a process is never more than $f$. As $p_i$ does not decide in round $f+2$ and $|Halt_i[f+2]| \leq f$, at least one of the following is true: (1) $\text{STATE}_i[f+1] = \text{NSYNC}$, or (2) some process $p_j$ sent a message in round $f+2$ with STATE = SYNC1. Case 1 implies that $|Halt_i| > t$ in round $f+1$ or in a lower round; a contradiction. Case 2 implies that $p_j$ updates $\text{STATE}_j$ to SYNC1 in step $f+1$, and hence, $f+1 \leq |Halt_j[f+1]| \leq t$; a contradiction. □

**Lemma 43** *(Time-complexity) For every $f \in [0,t]$, every correct process decides by round $f+2$ in a run of* $\text{SM}_f$.

**Proof**: Consider any run $r$ of the algorithm in $SM_f$ (synchronous run with at most $f$ crashes. Thus the lowest synchronous session is the first session. From Lemma 42, every correct process decides by step $f+2$ of the first session, i.e., by round $f+2$, in run $r$. □

**Lemma 44** *(Termination) Every correct process eventually decides.*

**Proof**: From Lemma 42, every correct process decides in the lowest synchronous session of a run. □

## 5.4 Summary of the results in the eventually synchronous model

Combining local decision lower bound of Lemma 27, the global decision lower bound of Lemma 29, and the time-complexity of algorithm $A_{em1}$, we get the following tight bounds in the eventually synchronous model.

**Theorem 45** *(Local decision bound for uniform consensus.)*
$\forall t \in [1,(n-1)/2]$, $\forall f \in [0,t-3]$, *(*$\text{EM}_t$, $\text{SM}_f$, *UC, ld)* $= f+2$.

**Proof**: Follows from Lemma 27, and the algorithm $A_{em1}$. □

**Theorem 46** *(Global decision bound for uniform consensus.)*
$\forall t \in [2, (n-1)/2]$, $\forall f \in [0, t]$, $(\mathrm{EM}_t, \mathrm{SM}_f, UC, gd) = f + 2$.

**Proof**: Follows from Lemma 29, and the algorithm $A_{em1}$. □

# Chapter 6

# Tight Bounds in the Eventually Synchronous Model
## (Part B) — Recovering from Asynchrony

In Chapter 5, we investigated how fast we can reach a decision in the eventually synchronous model ($EM$), when a run is synchronous from the beginning (i.e., $GSR = 1$) and there are $f$ failures in the run. In this chapter, we study a complementary question: how fast we can reach an agreement once the run becomes synchronous and no new failures occurs. In Chapter 7 (conclusion), we briefly discuss the general bound on the number of rounds required to decide once the run becomes synchronous and there are $f$ failures.

For any run in the eventually synchronous model, we define $GFR(r)$ (Global Failure stabilization Round) as the unknown round number such that (1) $GFR(r) \geq GSR(r)$, and (2) every process that enters round $GFR(r)$ is correct (in other words, every faulty process crashes in a round lower than $GFR(r)$ or crashes at the beginning of round $GFR(r)$). Note that there is always such a round in every run because faulty processes execute only a finite number of rounds. In this chapter, considering uniform consensus (UC) algorithms in the eventually synchronous model, we investigate bounds on the number of rounds required for global decision from round $GFR$.

## 6.1 The lower bound

In this section we give a lower bound on the number of rounds required for a global decision in $EM$. Actually, to strengthen our lower bound, we consider a model which satisfies all the properties of $EM$ as well as the following property: in every round $k$, each process that completes round $k$, has received at least $n - t$ round $k$ messages. (Note that we consider this additional property for showing the lower bound only. In particular, none of our algorithms rely on this property.) As we are concerned with proving a lower bound, without loss of generality we assume that the UC algorithms are (1) full-information, and (2) binary, i.e., we fix $V = \{0, 1\}$.

We say that round $k$ configuration is *failure-free* if all processes complete round $k$ in that configuration or the configuration is an initial configuration. Given a failure-free round $k$ configuration $C$ of a UC algorithm $A$, we define $r_j(C)$ ($j \in [1, n]$) to be a run such that (1) $C$ is the round $k$ configuration of $r_j(C)$, (2) $GFR(r_j(C))$ is $k+1$, and (3) $p_j$ does not enter round $k+1$ (i.e., $p_j$ crashes at the beginning of round $k+1$). Note that the run $r_j(C)$ is unambiguously defined by these three conditions because, (1) as $A$ is a full-information algorithm, $C$ completely defines the run until round $k$, and (2) the message exchange pattern is completely defined from round $k+1$. We denote by $val_j(C)$ the decision value of correct processes in $r_j(C)$. We say that a configuration $C$ is *uniFvalent* (uni-failure-valent) if all $val_j(C)$ have the same value; i.e., for any pair of $i, j$ such that $i, j \in [1, n]$, $val_i(C) = val_j(C)$. We denote this common value by $val^F(C)$. A uniFvalent configuration is 1-Fvalent if $val^F(C) = 1$; 0-Fvalent, otherwise. A configuration that is not uniFvalent is called *biFvalent*. In other words, in a biFvalent configuration, there are two processes $p_i$ and $p_j$, such that $val_i(C) \neq val_j(C)$. (Note that our notion of biFvalency is different from traditional notion of bivalency, introduced in [FLP85], and used in [AT99] to prove the $t+1$ lower bound on consensus. Roughly speaking, if a configuration $C$ is bivalent, there are two runs starting from $C$ with different decision values, whereas, if a configuration $C$ is biFValent, there are two processes, crashing each of which in C, leads to different decision values.) We now show the following lemma that we use to prove our lower bound.

**Lemma 47** *Let $t \in [1, n-1]$. Let $A$ be any UC algorithm in* $\mathrm{EM}_t$*. For every $k \geq 0$, there is a failure-free biFvalent round $k$ configuration.*

**Proof**: We prove the lemma by induction on round number $k$.

*Base Case:* There is a failure-free biFvalent initial configuration. By definition, all initial configurations are failure-free. Suppose by contradiction that all initial configurations are uniFvalent. Let $C_0$ be the initial configuration in which all processes propose 0. For $j \in [1, n]$, let $C_j$ be an initial configuration in which all processes $p_l$, where $l \in [1, j]$, propose 1 and the rest of the processes propose 0. Notice that, from UC validity, $val^F(C_0) = 0$ and $val^F(C_n) = 1$. We claim that, for $j \in [1, n]$, $val^F(C_{j-1}) = val^F(C_j)$. To see why, notice that $C_{j-1}$ and $C_j$ differ only the proposal value of $p_j$, and hence, no process can distinguish $r_j(C_{j-1})$ from $r_j(C_j)$. So $val_j(C_{j-1}) = val_j(C_j)$, and since $C_{j-1}$ and $C_j$ are uniFvalent, $val^F(C_{j-1}) = val_j(C_{j-1}) = val_j(C_j) = val^F(C_j)$. Our claim immediately implies that, if $val^F(C_0) = 0$ then $val^F(C_n) = 0$; a contradiction.

*Induction Hypothesis:* There is a failure-free biFvalent round $k$ configuration.

*Induction Step:* There is a failure-free biFvalent round $k+1$ configuration. Suppose by contradiction that all failure-free round $k+1$ configurations are uniFvalent. From induction hypothesis, there is a failure-free biFvalent round $k$ configuration $C$. Thus there are $i, j \in [1, n]$, such that $val_i(C) = 0$ and $val_j(C) = 1$. (In the rest of the proof, note that in round $k+1$ of each configuration we construct, each process receives at least $n-1 \geq n-t$ messages.)

Consider the failure-free round $k+1$ configuration $C^0$ that extends $C$ by one round, such that, in round $k+1$, all messages sent by $p_i$ are lost and no other message is lost. Consider the runs $r_i(C)$ and $r_i(C^0)$. The round $k+1$ configuration of $r_i(C)$ differ from $C^0$ only in the state of process $p_i$. Since $p_i$ crashes at the beginning of of round $k+2$ in $r_i(C^0)$, no correct process can distinguish $r_i(C)$ from $r_i(C^0)$. Thus, $val_i(C^0) = val_i(C) = 0$. $C^0$ being a failure-free round $k+1$ configuration, is uniFvalent, and hence, $val^F(C^0) = val_i(C^0) = 0$.

We now consider a series of round $k+1$ configurations which extend $C$ by one round. Configuration $C^l$ ($l \in [1, n]$) extends $C$ by one round in which (1) no process crashes, and (2) all messages sent by $p_i$ in round $k+1$ to processes in $\Pi \backslash \{p_1, ..., p_l\}$ are lost, and no other message is lost.

Consider configurations $C^{l-1}$ and $C^l$. The two configurations differ only at $p_l$: $p_l$ does not receive round $k+1$ message from $p_i$ in $C^{l-1}$, but receives that message in $C^l$. Thus no correct process can distinguish run $r_l(C^{l-1})$ from $r_l(C^l)$. Thus $val_l(C^{l-1}) = val_l(C^l)$. $C^{l-1}$ and $C^l$ being failure-free round $k+1$ configurations, are uniFvalent, and hence, $val^F(C^{l-1}) = val^F(C^l)$. A simple induction over $l$, along with our previous observation that $val^F(C^0) = 0$, gives us $val^F(C^n) = 0$. Observe that configuration $C^n$ extends $C$ by one round such that no process crashes and no message is lost in round $k+1$.

If we replace $p_i$ with $p_j$ in the above construction, we immediately get that $val^F(C^n) = val_j(C) = 1$; a contradiction. (The intermediate configurations will be different from the above paragraph, but the final configuration will still be $C^n$: a configuration that extends $C$ by one round such that no process crashes and no message is lost in round $k+1$.) □

**Lemma 48** *Let $t \in [1, n-2]$. For any $G \geq 1$, every UC algorithm in* $\mathrm{EM}_t$ *has a run $r$ in which* $\mathrm{GFR}(r) = G$ *and some process decides in round* $\mathrm{GFR}(r) + 1$ *or in a higher round.*

*Remark.* This lemma can also be shown using a simple modification of the proof of [KR03]. However, a straightforward modification of the proof of [KR03] would require $t \geq 2$, whereas our proof holds for $t \geq 1$.

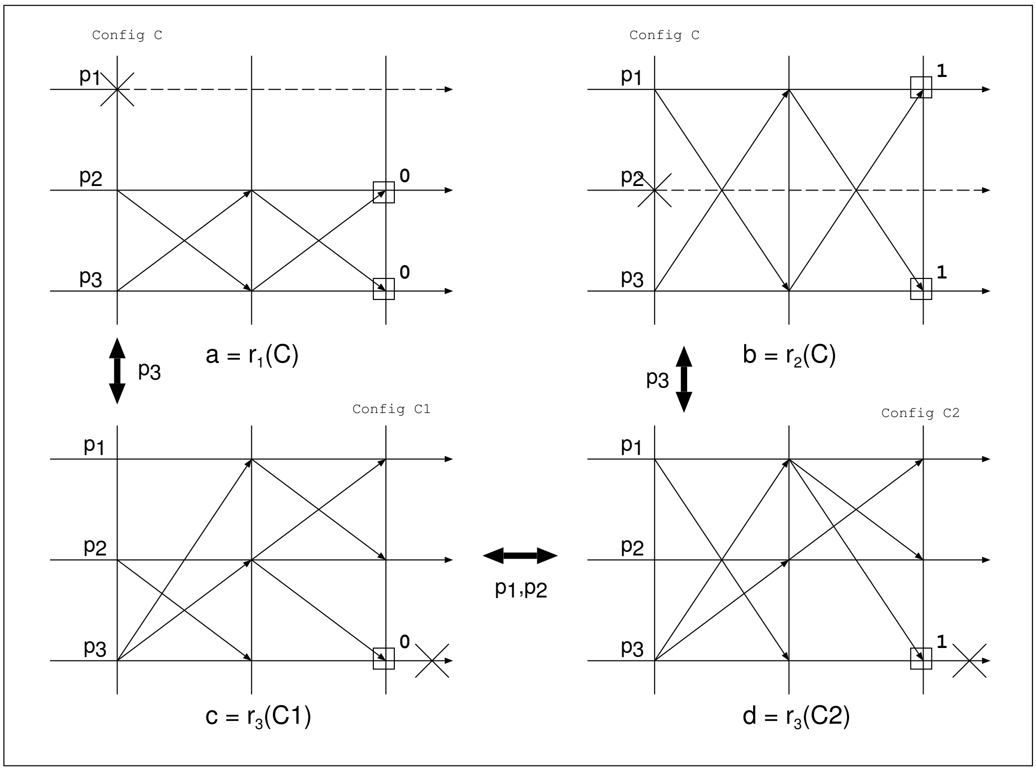


Figure 6.1: Rounds $G$ and $G+1$, Lemma 49

**Proof**: Suppose by contradiction that there exists a UC algorithm $B$, and some round number $G$, such that for every run $r$ of $B$ in which $GFR(r) = G$, all correct processes decide by round $G$.

Recall from Section 2.4, for any round $k$ configuration $C$, $r(C)$ denotes a run that is an extension of $C$ such that, every process that is alive is $C$ is correct in $r(C)$, and in every round higher than $k$, no message is lost (i.e., correct processes receive messages from all correct processes).

Consider a failure-free biFvalent round $G-1$ configuration $C$. (From Lemma 47, such a configuration exists.) Thus there are $i, j \in [1, n]$ such that $val_i(C) = 0$ and $val_j(C) = 1$. Observe that from our assumption on algorithm $B$, by the end of round $G$, every process distinct from $p_i$ decides 0 in $r_i(C)$, and every process distinct from $p_j$ decides 1 in $r_j(C)$. Also, from our assumption on $B$, every process decides by the end of round $G$ in $r(C)$. Let $x \in \{0, 1\}$ be the decision value of processes in $r(C)$. We show a contradiction assuming $x = 1$. (The case $x = 0$ is symmetric.)

Consider a run $r(C')$, where $C'$ is a failure-free round $G$ configuration that extends $C$ by one round, such that, in round $G$, $p_i$ receives its own message, all other messages sent by $p_i$ are lost, and no other message is lost. (Note that $GFR(r(C')) = G + 1$.)

Let $p_c$ be a process distinct from $p_i$. At the end of round $G$, $p_i$ cannot distinguish $r(C')$ from $r(C)$, and $p_c$ cannot distinguish $r(C')$ from $r_i(C)$. Thus, at the end of round $G$ in $r(C')$, $p_i$ decides $x = 1$ and $p_c$ decides 0. Run $r(C')$ violates uniform agreement; a contradiction. □

**Lemma 49** *Let $n = 3$ and $t = 1$. For any $G \geq 1$, every UC algorithm in* $\mathrm{EM}_t$ *has a run $r$ in which* $\mathrm{GFR}(r) = G$ *and some process decides in round* $\mathrm{GFR}(r) + 2$ *or in a higher round.*

**Proof**: Suppose by contradiction that there exists a UC algorithm $A$, and some round number $G$, such that for every run $r$ of $A$ in which $GFR(r) = G$, all correct processes decide by round $G + 1$.

Consider a failure-free biFvalent round $G - 1$ configuration $C$. (From Lemma 47, such a configuration exists.) Thus there are $i, j \in [1, 3]$ such that $val_i(C) = 0$ and $val_j(C) = 1$. For convenience of presentation and without loss of generality, we assume that $i = 1$ and $j = 2$.

We consider four runs that extend $C$. (In each run, note that processes receive at least $n - t = 2$ messages in every round — including one from itself.) Rounds $G$ and $G + 1$ of these runs are depicted in Figure 6.1. We now describe them in words.

- Run $a$ is $r_1(C)$. Thus $GFR(a) = G$, and from our assumption on $A$, correct processes decide $val_1(C) = 0$ in round $G + 1$.
- Run $b$ is $r_2(C)$. Thus $GFR(b) = G$, and from our assumption on $A$, correct processes decide $val_2(C) = 1$ in round $G + 1$.
- Run $c$ is $r_3(C1)$, where round $G+1$ configuration $C1$ is constructed as follows. In round $G$, the messages from $p_1$ to $\{p_2, p_3\}$ are lost, and the message from $p_2$ to $p_1$ is lost. In round $G+1$, the messages from $p_1$ to $p_3$, and $p_3$ to $\{p_1, p_2\}$ are lost. Process $p_3$ cannot distinguish round $G + 1$ configuration of run $c$ (i.e., configuration $C1$) from round $G + 1$ configuration of run $a$. To see why, notice that $p_3$ does not receive any message from $p_1$ in round $G$ and $G + 1$ of both runs. Furthermore, $p_2$ distinguishes $a$ from $c$ only at the end of round $G + 1$, and hence, sends identical messages to $p_3$ in rounds $G$ and $G + 1$ of both runs. Therefore, as in run $a$, $p_3$ decides 0 in round $G + 1$ in run $c$. Due to the uniform agreement property, $p_1$ and $p_2$ eventually decide 0 in run $c$.
- Run $d$ is $r_3(C2)$, where round $G+1$ configuration $C2$ is constructed as follows. In round $G$, the message from $p_1$ to $p_2$ is lost, and the messages from $p_2$ to $\{p_1, p_3\}$ are lost. In round $G + 1$, the message from $p_2$ to $p_3$, and from $p_3$ to $\{p_1, p_2\}$ are lost. Notice that $p_3$ cannot distinguish round $G + 1$ configuration of $d$ (i.e., configuration $C2$) from round $G+1$ configuration of run $b$. Therefore $p_3$ decides 1 at the end of round $G+1$ of run $d$. Due to the uniform agreement property, $p_1$ and $p_2$ eventually decide 1 in run $d$.

Now consider runs $c$ and $d$. At the end of round $G$, the two runs differ only at process $p_3$ (because it receives different sets of messages). Process $p_1$ receives the same set of messages in round $G+1$ of runs $c$ and $d$, and that does not include a message from $p_3$. Therefore, the state of $p_1$ is the same at the end of round $G+1$ in both runs. Similarly, we can show that the state of $p_2$ is the same at the end of round $G+1$ in both runs. Since process $p_3$ does not send any message after round $G+1$ (recall that $c$ is $r_3(C1)$ and $d$ is $r_3(C2)$), $p_1$ and $p_2$ can never distinguish run $c$ from run $d$. Therefore, $p_1$ (and $p_2$) must decide the same value in $c$ and $d$: a contradiction. □

**Lemma 50** *Let $t \in [1, n-2]$ and $t \geq n/3$. For any $G \geq 1$, every UC algorithm in* $\text{EM}_t$ *has a run $r$ in which* $\text{GFR}(r) = G$ *and some process decides at round* $\text{GFR}(r)+2$ *or at a higher round.*

**Proof**: We prove this lemma by simulating the proof of Lemma 49 over a system where $t \geq n/3$. (Recall that, we always assume $n \geq 3$.) Divide the set of processes $\Pi$ into 3 sets of processes, $P_1$, $P_2$, and $P_3$, each of size less than or equal to $\lceil \frac{n}{3} \rceil$. (This is always possible because $3(\lceil \frac{n}{3} \rceil) \geq n$.) Since $t \geq n/3$ and $t$ is an integer, it follows that $t \geq \lceil \frac{n}{3} \rceil$. Therefore, the sets $P_1$, $P_2$, and $P_3$ are each of size less than or equal to $t$, and hence, in a given run all the processes in any one of the sets may crash.

We now construct runs corresponding to the runs in Lemma 49. The relationship between a run $r'$ constructed in this simulation to the corresponding run $r$ in Lemma 49 is as follows: (1) if $p_i$ proposes $x$ (0 or 1) in $r$, then every process in $P_i$ proposes $x$ in $r'$, (2) if $p_i$ crashes without sending any message in some round $k$ of $r$, then every process in $P_i$ crashes without sending any message in round $k$ of $r'$, (3) if $p_i$ crashes in some round $k$ of $r$, then every process in $P_i$ crashes in round $k$ of $r'$, (4) if $p_i$ does not crash in $r$ then no process in $P_i$ crashes in $r'$, and (5) for $j \in [1, 3]$, if $p_i$ receives a messages from $p_j$ in some round $k$ of $r$, then every process in $P_i$ receives a message from every process in $P_j$ in round $k$ of $r'$. (Note that in $r$, if $p_i$ does not crash at round $k$, then it receives a message from itself, and therefore, at round $k$ of $r'$, each process in $P_i$ receives a message from every process in $P_i$.)

Following the above rules, we construct the configuration $C'$ corresponding to $C$ and the four runs $a'$, $b'$, $c'$, and $d'$, corresponding to runs $a$, $b$, $c$, and $d$, respectively, to reach a contradiction. □

We now state our lower bound on the number of rounds required to globally decide after *GFR*.

**Theorem 51** *Let $t \in [1, n-2]$. For any $G \geq 1$,*
*(a) every UC algorithm in* $\text{EM}_t$ *has a run $r$ in which* $\text{GFR}(r) = G$ *and some process decides at round* $\text{GFR}(r)+1$ *or at a higher round.*

*(b) if $t \geq n/3$, then every UC algorithm in* $\mathrm{EM}_t$ *has a run $r$ in which* $\mathrm{GFR}(r) = G$ *and some process decides at round* $\mathrm{GFR}(r) + 2$ *or at a higher round.*

**Proof**: Immediate from Lemma 48 and Lemma 50. □

**Communication closed rounds and reliable channels.** There is an obvious way in which we can strengthen $EM$. We remove the restriction of communication closed rounds and we add reliable channels: a process may receive messages from any round, and messages from correct processes to correct processes are eventually received. We now argue why Theorem 51 holds despite this modification. Our discussion is informal.

A *delayed* message is a message that is not received in the round in which it is sent. We claim that in the above proofs we can ignore all delayed messages. Recall that, we assumed that algorithms are full-information. So on receiving a message from a process $p_i$ in round $k$, another process $p_j$ can simulate reception of all delayed messages sent by $p_i$ to $p_j$ in lower rounds. This simulation in $EM$ satisfies the requirements of the modification even if all delayed messages are lost because, starting from round $GSR \leq GFR$ every correct process receives messages from all correct processes. Clearly, the simulation does not provide any additional information to processes because the algorithms we consider are already full-information. Thus, for the proof of Theorem 51, we can ignore the delayed messages even in the modified model.

## 6.2 A matching algorithm when $t < n/2$

We now present an algorithm $A_{em2}$ that solves UC in $EM_t$ when $t < n/2$. Algorithm $A_{em2}$ matches the lower bound of Theorem 51(b). Recall that, from Lemma 9, there is no UC algorithm in $ES_t$ when $t \geq n/2$.

### 6.2.1 Algorithm description

Algorithm $A_{em2}$ is presented in Figure 6.2. In every round, each process $p_i$ sends its four primary variables to all processes: (1) the message type $msgType_i$ initialized to PREPARE, (2) an estimate $est_i$ of the decision value, initialized to the proposal value (that is read from $prop_i$), (3) the timestamp $ts_i$ of the estimate value, initialized to 0, and (4) the leader $ld_i$ of the current round, initialized to $p_n$. In the computation subround, processes update their primary variables depending on the messages received in that round, and possibly decide. First we briefly explain the purpose of these variable at process $p_i$.

```
at process p_i
initialize()
in round k                                                        {rounds 1, 2, ...}
    send round k messages
    receive messages
    compute()
procedure initialize()
    est_i ← prop_i; ld_i ← p_n; ts_i ← 0; msgType_i ← PREPARE; nextLD_i ← p_n; maxTS_i ← 0
    round 1 message ← (1, msgType_i, est_i, ts_i, ld_i)
procedure compute()
    if dec_i = ⊥ then
        nextLD_i ← p_j where j = Max{w|p_i received a round k message from p_w}
        maxTS_i ← Max{ts|p_i received a message (k, ∗, ∗, ts, ∗) }
        if received (k, DECIDE, est′, ts′, ∗) then
            est_i ← est′; ts_i ← ts′; dec_i ← est_i; msgType_i ← DECIDE             {decision}
        else if received (k, COMMIT, ∗, ∗, ∗) from a majority including itself (p_i) and ld_i then
            dec_i ← est_i; msgType_i ← DECIDE                                     {decision}
        else if (received(k, ∗, ∗, ∗, ld_i) from a majority of processes)         {COMMIT-1}
                and (received (k, ∗, ∗, maxTS_i, ld_i) from ld_i)                 {COMMIT-2}
                and (ld_i = nextLD_i) then                                        {COMMIT-3}
            msgType_i ← COMMIT; est_i ← est received from ld_i; ts_i ← k
        else
            est_i ← any est s.t. received (k, ∗, est, maxTS_i, ∗); ts_i ← maxTS_i; msgType_i ← PREPARE
        ld_i ← nextLD_i
    round k + 1 message ← (k + 1, msgType_i, est_i, ts_i, ld_i)
```

Figure 6.2: Uniform consensus algorithm $A_{em2}$

Roughly speaking, the message type indicates the level of progress a process has made towards reaching a decision. In the computation subround of round $k$, if $p_i$ sees a possibility of decision in the next round then it sends a round $k+1$ message with type COMMIT. We then say that $p_i$ *commits in round* $k$. If the process decides or has already decided then it sends a message with type DECIDE in the next round. Otherwise, the message type is PREPARE.

In the computation subround of a round $k$, $p_i$ adopts one of the estimate values received in that round. Process $p_i$ also adopts the timestamp received along with the estimate, except when $p_i$ commits in round $k$, in which case $p_i$ updates its timestamp to $k$. Thus the timestamp associated with an estimate value $x$ simply indicates a round number in which some processes has committed while adopting estimate $x$.

The *leader of* $p_i$ *at round* $k \geq 2$ is simply the process $p_j$ with the highest id such that, $p_i$ received the round $k-1$ message from $p_j$. Process $p_n$ is the leader at all processes in round 1. Note that different processes may have different leaders in the same round. Now we describe the computation subround in more details.

Once a process $p_i$ decides, it sends a DECIDE message with the decision value in every round. Otherwise, in round $k$, $p_i$ updates its primary variables in the procedure compute(), as follows. From the set of messages received, $p_i$ first computes its leader for the next round ($nextLD_i$) and the highest timestamp received ($maxTS_i$). Then it executes the following four conditional statements. (A statement is executed only if the conditions in all the previous statements are false.)

- If $p_i$ receives a DECIDE message then it decides on the received estimate (by writing that estimate in $dec_i$).
- If $p_i$ receives COMMIT messages from a majority of processes, including itself and its current leader, then $p_i$ decides on its own estimate.
- Let $ld_i$ be the leader of $p_i$ at round $k$. Consider the following three conditions on the messages received by $p_i$. (1) *commit-1*: received messages from a majority of processes, that say that $ld_i$ is their leader at round $k$, (2) *commit-2*: received a message from $ld_i$ that has the highest timestamp ($maxTS_i$) and has $ld_i$ as the leader, and (3) *commit-3*: $ld_i = nextLD_i$. If all three conditions are satisfied, then $p_i$ sets its message type (for the message to be send in round $k+1$) to COMMIT, adopts the estimate received from $ld_i$, say $x$, and sets its timestamp to the current round number $k$. We say that $p_i$ *commits in round $k$ with estimate $x$.*
- Else, $p_i$ adopts the estimate and the timestamp of the message with the highest timestamp $maxTS_i$, and sets its message type to PREPARE.

Finally, $p_i$ updates its $ld_i$ to $nextLD_i$ and composes the message for the next round.

### 6.2.2 Correctness of $A_{em2}$

**Lemma 52** *Until a process decides, its timestamp is non-decreasing with increasing rounds.*

**Proof**: If a process $p_i$ does not decide in round $k$, then it adopts either $k$ or the maximum timestamp received in round $k$, as its new timestamp. From the loopback property of $EM_t$, we know that $p_i$ receives its own message in round $k$, and hence, the new timestamp of $p_i$ is not lower that its current timestamp. □

**Lemma 53** *In every run $r$, all correct processes decide by round* GFR$(r)+2$.

**Proof**: We prove the lemma by contradiction. Assume that some correct process $p_j$ does not decide by round $GFR(r)+2$ in run $r$. If any correct process $p_i$ decides by round $GFR(r) + 1$, then it sends a DECIDE message in round $GFR(r) + 2$, and all correct processes receive that message and decide in round $GFR(r) + 2$; contradicting our assumption. Therefore, our assumption implies that, no correct processes decides by round $GFR(r) + 1$.

Let $p_l$ be the correct process with the highest id in $r$. Since correct processes receive messages from all correct processes in round $GFR(r)$ and in all higher rounds, it follows that $p_l$ is the leader of all correct processes in round $GFR(r) + 1$ and in all higher rounds.

Consider round $GFR(r)$. We claim that, at the end of round $GFR(r)$, no process has a higher timestamp than $p_l$. Suppose by contradiction that some other process $p_j$ completes round $GFR(r)$ with a higher timestamp than $p_l$; say the timestamp of $p_j$ is $k'$. There are three cases depending on when $p_j$ adopted timestamp $k'$: (1) $p_j$ adopted timestamp $k'$ before round $GFR(r)$, (2) $p_j$ adopted timestamp $k'$ on receiving a message from some process $p_m$ in round $GFR(r)$ with timestamp $k'$, or (3) $p_j$ committed in round $GFR(r)$ and adopted $k' = GFR(r)$ as its timestamp. In the first two cases, since only correct processes enter round $GFR(r)$, and correct processes receive messages from all correct processes in round $GFR(r)$, $p_l$ receives a message with timestamp $k'$ (from $p_j$ in the first case, and from $p_m$ in the second case) and adopts a timestamp not smaller than $k'$; a contradiction.

Consider the third case. We show that $p_l$ commits in round $k' = GFR(r)$. In round $GFR(r)$, correct processes receive message from all correct processes, i.e., all correct processes receive the same set of messages. Therefore, every correct process evaluates $nextLD$ to $p_l$, and evaluates $maxTS$ to the same timestamp, say $ts'$. Since $p_j$ commits in round $GFR(r)$, so from condition commit-3, the leader of $p_j$ in round $GFR(r)$ is same as its $nextLD$; i.e., $p_l$. From condition commit-2 it follows that $p_j$ received a message $(GFR(r), *, *, ts', p_l)$ from $p_l$. Thus $p_l$ is its own leader in round $GFR(r)$. Thus at $p_l$, condition commit-3 holds. As all correct processes receive the same set of messages in round $GFR(r)$, and $p_j$ and $p_l$ have the same leader in round $GFR(r)$, commit-1 and commit-2 hold also at $p_l$. Thus, $p_l$ commits in round $GFR(r)$, and hence, updates its timestamp to $GFR(r) = k'$; a contradiction with our assumption that $k'$ is higher than the timestamp of $p_l$ at the end of round $GFR(r)$.

Thus no process has higher timestamp than $p_l$ at the end of round $GFR(r)$. Let $ts''$ be the timestamp of $p_l$ at the end of round $GFR(r)$. Consider round $GFR(r) + 1$. Clearly, $p_l$ sends $(GFR(r)+1, *, *, ts'', p_l)$. Every process on receiving this message evaluates $maxTS$ to $ts''$. At every correct process, $p_l$ is the leader, and $nextLD$ is evaluated to $p_l$. Thus, all three conditions required to commit holds at every correct process. As no correct process decides by round $GFR(r) + 1$, every correct process commits in round $GFR(r) + 1$. Thus in the next round, every correct process sends the message $(GFR + 2,$ COMMIT, $*, *, p_l)$. In round $GFR(r) + 2$, every correct process receives COMMIT messages from all correct processes, and hence, decides; a contradiction. □

**Lemma 54** *For any round $k$, no two processes commit with different estimates in round $k$, and no two processes commit with different $newLD$ in round $k$.*

**Proof**: Consider two processes $p_i$ and $p_j$ that commit in round $k$ with estimate $est_i$ and $est_j$, and $newLD$ value $newld_i$ and $newld_j$, respectively. Also, in round $k$, let $ld'_i$ be the leader of $p_i$ and $ld'_j$ be the leader of and $p_j$. Thus from commit-1, each of them has received a majority of messages in round $k$, that contain $ld'_i$ and $ld'_j$ as leaders, respectively. As two majorities intersect, $ld'_i = ld'_j$. Furthermore, from commit-3, $newld_i = ld'_i$ and $newld_j = ld'_j$. So, $newld_i = ld'_i = ld'_j = newld_j$.

From the algorithm, $p_i$ commits with the estimate sent by $ld'_i$, and $p_j$ commits with the estimate sent by $ld'_j$. As $ld'_i = ld'_j$, $p_i$ and $p_j$ commit with same estimate. □

**Lemma 55** *For any round $k$, all round $k$ messages with* msgType = COMMIT *have identical estimate values and identical ld values.*

**Proof**: Immediate from Lemma 54. □

**Lemma 56** *If some process sends a message with timestamp $ts > 0$ and estimate $x$ then some process commits in round $ts$ with estimate $x$.*

**Proof**: If a process $p_i$ sends a message with timestamp $ts$ then $p_i$ sets its timestamp to $ts$ in some round. Consider the lowest round $k$ in which some process sets its timestamp to $ts$, and let process $p_j$ be one such process. From the definition of $k$, $p_j$ cannot receive timestamp $ts$ from another process in round $k$. Thus $p_j$ commits with timestamp $ts$ in round $k$, and from the algorithm, $k = ts$.

Also, from the algorithm, if a process adopts a timestamp from a received message, it also adopts the associated estimate. Thus no two values are associated with the same timestamp. It follows that if $p_i$ sends a message with timestamp $ts$ and estimate $x$ and some process $p_j$ commits in round $ts$, then $p_j$ commit with estimate $x$. □

**Lemma 57 (Uniform Agreement)** *No two processes decide differently.*

**Proof**: If no process ever decides then the lemma trivially holds. Suppose some process decides. Let $k$ be the lowest round in which some process decides; say $p_i$ decides in round $k$. Process $p_i$ can decide either (1) by receiving a DECIDE message, or (2) by receiving a majority of COMMIT messages, that include messages from itself and its leader. In case 1, some process has sent a DECIDE message in round $k$, and hence, that process has decided in a round lower than $k$, which contradicts the definition of round $k$. We now consider case 2.

Suppose $p_i$ decides $x$ in round $k$. As $p_i$ received a majority of COMMIT messages in round $k$, and one of the COMMIT messages contains the decision value (namely, the COMMIT message from itself), from Lemma 55, it follows that the estimate in the COMMIT messages is $x$, and all COMMIT messages have the same leader, say $p_l$. Thus $p_i$ receives ($k$, COMMIT, $x$, $k-1$, $p_l$) from a majority of processes, and hence, a majority of processes commit in round $k-1$ with estimate $x$ — let us denote this majority of processes by $S_x$.

We claim that if any process commits or decides in round $k' \geq k-1$, then it commits with estimate $x$ or decides $x$. The claim immediately implies agreement. We prove the claim by induction on round number $k'$.

*Base Case.* $k' = k-1$. As processes in $S_x$ commit $x$ in round $k-1$, so from Lemma 54, no process commits with an estimate different from $x$ in round $k-1$. By definition of $k$, no process decides in round $k-1$.

*Induction Hypothesis.* If any process commits or decides in any round $k1$ such that $k-1 \leq k1 \leq k'$, then it commits with estimate $x$ or decides $x$.

*Induction Step.* If any process commits or decides in round $k'+1$, then it commits with estimate $x$ or decides $x$. There are two cases:

1. Some process commits in round $k'+1$. Suppose by contradiction that some process $p_j$ commits with estimate $z \neq x$ in round $k'+1$. Then $p_j$ has not received any DECIDE message in round $k'+1$. Also note that, from condition commit-2, $p_j$ commits on the estimate of the round $k'+1$ message $m'$ received from its leader, and this message has the highest timestamp among all messages received by $p_j$ in round $k'+1$. Let this highest timestamp be $tsMax$. Therefore, some process has sent round $k'+1$ message with timestamp $tsMax$ and estimate $z$. From Lemma 56, some process commits in round $tsMax$ with estimate $z$.

   As the highest timestamp that can be received in round $k'+1$ is $k'$, so $tsMax \leq k'$. Since $p_j$ commits in round $k'+1$, it has received round $k'+1$ messages from a majority of processes, and hence, received round $k'+1$ message from at least one process in $S_x$, say $p_a$. Recall that, every process in $S_x$ commits in round $k-1$ with estimate $x$. Thus $p_a$ has timestamp $k-1$ at the end of round $k-1$. As $p_j$ has not received any DECIDE message in round $k'+1$, $p_a$ has not decided by round $k'$. From Lemma 52, the round $k'+1$ message of $p_a$ contains timestamp at least $k-1$. Thus $tsMax \geq k-1$.

   Thus we have $k-1 \leq tsMax \leq k'$. By induction hypothesis, every process that commits in round $tsMax$ commits $x \neq z$; a contradiction.

2. If some process $p_b$ decides a value $y$ in round $k' + 1$, then in that round, either some process sends a DECIDE message with decision value $y$ or $p_b$ sends a COMMIT message with estimate $y$. From induction hypothesis, $y = x$ in both cases.

□

**Lemma 58** *Algorithm $A_{em2}$ solves UC.*

**Proof**: Termination follows from Lemma 53, validity is obvious, and uniform agreement is proved in lemma 57. □

**Theorem 59** *There is a UC algorithm in* $\mathrm{EM}_t$ *with* $t < n/2$ *such that in every run* $r$*, correct processes decide by round* $\mathrm{GFR}(r) + 2$.

**Proof**: Immediately from Lemma 53 and Lemma 58. □

## 6.3 A matching algorithm when $t < n/3$

We now present an algorithm $A_{em3}$ that solves UC in $ES_t$ when $t < n/3$. The algorithm matches the lower bound of Theorem 51(a), and is inspired by an algorithm from [MR01]. Algorithm $A_{em3}$ is presented in Figure 6.3. The algorithm is based on the following simple observation. Suppose $t < n/3$, and $S$ is a multiset of $n$ elements where some element $v$ appears $n - t$ times. Then in any multiset containing $n - t$ elements from $S$, $v$ appears at least $n - 2t$ times and all other elements appear less than $n - 2t$ times.

We assume every proposal value has a tag which contains the id of the process that proposed the value. The proposal values can be ordered based on this tag. In every round, each process $p_i$ sends its three primary variables to all processes: (1) the message type $msgType_i$ initialized to PREPARE, (2) an estimate $est_i$ of the decision value, initialized to the proposal value (that is read from $prop_i$), and (3) the timestamp $ts_i$ of the estimate value, initialized to 0. In the computation subround, $p_i$ decides if it receives a DECIDE message. If $p_i$ receives less than $n - t$ messages in round $k$ then it does not update its variables in that round. If $p_i$ receives at least $n - t$ messages then it updates its timestamp to the current round number $k$ and updates other variables as follows. First it arranges all messages received in the round in ascending order of their sender ids, selects the first $n - t$ messages, and puts them in set $msgSet_i$. If every message in $msgSet_i$ has the same estimate, say

```
at process p_i
initialize()
in round k                                                        {rounds 1, 2, ...}
    send round k messages
    receive messages
    compute()

procedure initialize()
    est_i ← prop_i; ts_i ← 0; msgType_i ← PREPARE; maxTS_i ← 0; msgSet_i ← ∅
    round 1 message ← (1, msgType_i, est_i, ts_i)

procedure compute()
    if dec_i = ⊥ then
        if received (k, DECIDE, est', ts') then
            est_i ← est'; ts_i ← ts'; dec_i ← est_i; msgType_i ← DECIDE        {decision}
        else if received at least n − t messages in round k then
            ts_i ← k
            msgSet_i ← set of n − t round k messages received by p_i with lowest sender ids
            maxTS_i ← Max{ts| (k, ∗, ∗, ts) ∈ msgSet_i}
            if every message in msgSet_i has identical est (say est') and has ts = k − 1 then
                dec_i ← est'; msgType_i ← DECIDE                               {decision}
            else if there are at least n − 2t messages in msgSet_i with identical est (say est'') then
                est_i ← est''
            else
                est_i ← Max {est| (k, ∗, est, maxTS_i) ∈ msgSet_i}
    round k + 1 message ← (k + 1, msgType_i, est_i, ts_i)
```

Figure 6.3: Uniform consensus algorithm $A_{em3}$

$est'$, and every message in $msgSet_i$ has timestamp $k-1$, then $p_i$ decides $est'$. If at least $n-2t$ messages in $msgSet_i$ have the same estimate, say $est''$, then $p_i$ adopts $est''$. Otherwise, among the estimates received with maximum timestamp, $p_i$ adopts the maximum one. We now sketch the correctness of $A_{em3}$.

**Lemma 60** *In every run $r$, all correct processes decide by round* $\mathrm{GFR}(r)+1$.

**Proof**: We prove the lemma by contradiction. Assume that some correct process $p_j$ does not decide by round $GFR(r)+1$ in run $r$. If any correct process $p_i$ decides by round $GFR(r)$, then it sends a DECIDE message in round $GFR(r)+1$, and all correct processes receive that message and decide in round $GFR(r)+1$; contradicting our assumption. Therefore, our assumption implies that, no correct processes decides by round $GFR(r)$.

Consider round $GFR(r)$. Recall that only correct processes enter the round, and all correct processes receive messages from all correct processes. It follows that every correct process receives at least $n-t$ messages, and receives the same set of messages. Since no correct process decides in that round, correct processes update their timestamp to $GFR(r)$, and compute identical $msgSet$. Then, either every correct process receives some estimate at least $n-2t$ times and adopts that estimate, or adopts the maximum estimate with maximum timestamp. In either case, since processes have identical *msgSet*, they update their estimates to the same value. Thus in round $GFR(r) + 1$, processes receive identical estimate from all correct processes with timestamp $GFR(r)$, and decide; a contradiction. □

**Lemma 61 (Uniform Agreement)** *No two processes decide differently.*

**Proof**: If no process ever decides then the lemma trivially holds. Suppose some process decides. Let $k$ be the lowest round in which some process decides; say $p_i$ decides in round $k$. Process $p_i$ can decide either (1) by receiving a DECIDE message, or (2) by receiving PREPARE messages from $n - t$ processes with identical estimate values and with timestamp $k-1$. In case 1, some process has sent a DECIDE message in round $k$, and hence, that process has decided in a round lower than $k$, which contradicts the definition of round $k$. We now consider case 2.

Suppose $p_i$ decides $x$ in round $k$. Then in round $k - 1$, at least $n - t$ processes update their timestamp to $k - 1$ and their estimate to $x$. Let this set of at least $n - t$ processes be $S_x$.

We claim that if any process updates its estimate or decides in round $k' \geq k-1$, then it updates it estimate to $x$ or decides $x$. This claim immediately implies agreement. We prove the claim by induction on round number $k'$.

*Base Case.* $k' = k - 1$. From the definition of round $k$, no process decides in round $k - 1$. Suppose some process $p_j$ updates its estimate in round $k$. Then $p_j$ has received at least $n - t$ messages. As $t < n/3$, at least $n - 2t$ of those messages are from processes in $S_x$, and hence, contain estimate $x$, and less than $n - 2t$ messages are from processes not in $S_x$. Thus $p_j$ updates its estimate to $x$.

*Induction Hypothesis.* If any process updates its estimate or decides in any round $k1$ such that $k - 1 \leq k1 \leq k'$, then it updates it estimate to $x$ or decides $x$.

*Induction Step.* If any process updates its estimate or decides in round $k' + 1$, then it updates it estimate to $x$ or decides $x$. Suppose a process decides $y$ in round $k' + 1$. Then either (1) some process has decided $y$ in a lower round and sent a DECIDE message in round $k' + 1$, or (2) at least $n - t$ processes has updated their estimate to $y$ in round $k'$. In the first case, from the induction hypothesis and our assumption that no process decides before round $k$, it follows that $y = x$. Consider the later case. Again from the induction hypothesis it follows that, by the end of round $k'$, all processes in $S_x$ has either decided $x$, retained their estimate $x$, or has crashed. As there are at least $n-t$ processes in $S_x$ and two sets of size $n-t$ intersect, we have $y = x$.

Now suppose some process $p_j$ updates its estimate in round $k'+1$. Then $p_j$ has received at least $n-t$ messages in round $k'+1$. As $t < n/3$, at least $n-2t$ of those messages are from processes in $S_x$, and hence from the induction hypothesis, contain estimate or decision value $x$. Also, less than $n-2t$ messages are from processes not in $S_x$, and so, less than $n-2t$ messages can contain a value different from $x$. Thus $p_j$ updates its estimate to $x$. □

**Lemma 62** *Algorithm $A_{em3}$ solves UC.*

**Proof**: Termination follows from Lemma 60, validity is obvious, and uniform agreement is proved in lemma 61. □

**Theorem 63** *There is a UC algorithm in* $\mathrm{EM}_t$ *with* $t < n/3$ *such that, in every run* $r$*, correct processes decide by round* $\mathrm{GFR}(r)+1$.

**Proof**: Immediate from Lemma 60 and Lemma 62. □

# Chapter 7

# Conclusion

This thesis investigates how fast we can achieve agreement. We focused on a more fine-grained time-complexity metric (local decision) than what was considered in the literature, and we looked into optimizing algorithms for subsets of runs that are considered to be common in practice.

The time-complexity of a local decision is a natural measure in many agreement-based distributed systems. As pointed out in the introduction, in a replication or a transaction system, it may be sufficient for a client to receive the decision value from any process executing the agreement algorithm. Besides, studying local decision metric helps uncover fundamental differences between problems and between models that were not apparent with other metrics. For example, in the synchronous model, uniform consensus and non-blocking atomic commit have the same tight bound in terms of global decision, but have different bounds when we consider local decision. Similarly, considering the local decision metric allows us to infer that early deciding uniform consensus algorithms are faster in the synchronous model than in synchronous runs of the eventually synchronous model.

Early deciding and early halting agreement algorithms have been extensively studied in the synchronous model since their introduction in [DRS90]. These algorithms optimize the subset of runs where there are less crashes than the maximum number of crashes tolerated by the algorithm. We introduced a natural extension of this optimization in the eventually synchronous model, namely, optimizing the subset of runs that are synchronous from the very beginning. Although we show that the synchronous runs of uniform consensus algorithms designed for the eventually synchronous model are inherently slower than runs of algorithms directly designed for the synchronous model, this difference is at most one round. (We would like to however recall that there is a significant resilience-price to be paid: in the synchronous model, uniform consensus can be solved if any number of processes may fail, whereas, in the eventually synchronous model, we need a majority of correct processes.)

We now outline few open issues and future directions for investigation.

### Number of rounds required for a global decision after *GSR*

Consider the lower bound on the number of rounds required for a global decision from round *GSR*. (This bound is different from the one considered in Chapter 6 because we now consider *GSR* instead of *GFR*.) Recall that, before *GSR*, any message sent by a process to other processes may be lost. Therefore, it is straightforward to extend the proof of Lemma 28 and Lemma 29 to show the following:

- For every UC algorithm $A$ in $EM_t$, for every $f \leq t$, and any $G \geq 1$, there is a run $r$ of $A$ with at most $f$ crashes in which, $GSR(r) = G$, and some correct process decides in round $G+f+1$ or in a higher round (i.e., at least $f+2$ rounds are required for a global decision once the system becomes synchronous).

We claim that algorithm $A_{em3}$ from Chapter 6 matches this lower bound when $t < n/3$. Consider any run $r$ of $A_{em3}$ with at most $f$ crashes. Note that processes receive $n-f \geq n-t$ messages in every round starting from round $GSR(r)$. Consider the $f+1$ rounds, $GSR(r)$ to $GSR(r)+f$. As there are at most $f$ crashes in $r$, it follows that, among these $f+1$ rounds, there is at least one round in which no process crashes, say round $k$. Thus, every process that enters round $k$, completes the round. It follows that every process that completes round $k$, has (1) either decided, or (2) has identical *msgSet* at the end of the round, and hence, updates its estimate to the same value, and also updates its timestamp to $k$. Thus, in round $k+1$, the processes either (1) receive a DECIDE message, or (2) all PREPARE messages have the same estimate and timestamp $k$. The correct processes decide by round $k+1$ in both cases.

Thus, $GSR + f + 1$ is a tight bound when $t < n/3$. Determining the tight bound when $t \geq n/3$ remains an open problem.

### Eventually synchronous model without rounds

This thesis considered round based models *EM* and *SM*. A natural extension would be to investigate a tight bound on the time required to reach a decision in models that do not impose any such round structure, but still, provide some timing guarantees.

Consider the following model $EM'$. Each process $p_i$ has a local clock which provides the real time. In every run $r$, there is an unknown time $GFT(r)$ (Global Failure stabilization Time of run $r$) such that (1) all faulty processes crash before $GFT(r)$, and (2) any message sent at time $GFT(r)$ or later is delivered within time $\Delta$ of being sent. ($\Delta$ is a known constant.) Also, the local processing time is negligible.

It is easy to simulate $EM$ over $EM'$: to simulate round $k$ of $EM$, the processes send round $k$ messages at time $(k-1)\Delta$, and upon receiving a round $k$ message

$m$, the reception of $m$ is simulated in $EM$ only if $m$ is received before time $k\Delta$. Thus $A_{em2}$ immediately translates to an algorithm in $EM'$ that decides within $4\Delta$ of $GFT$ — in the simulation, a round that has the same properties as round $GFR$ in $EM$ starts by time $GFT + \Delta$, and then, the algorithm $A_{em2}$ globally decides within three rounds, where each round is of duration $\Delta$.

We obtain a more interesting model, if we relax the requirement on the local clock, e.g., consider a model where local clocks do not provide real time, but after $GFT$, rates of local clocks are same as the rate of real time. In a recent paper [DGL05], we show that it is possible to achieve a global decision within a large but constant multiple of $\Delta$ (from time $GFT$). Determining a tight bound on the time required for achieving a global decision in this model remains an open problem. In general, designing efficient algorithms that can tolerate arbitrary periods of asynchrony, but decide quickly once some weak synchrony guarantees hold, seems to be a challenging research topic.

# Bibliography


[AT99] M. K. Aguilera and S. Toueg. A simple bivalency proof that $t$-resilient consensus requires $t + 1$ rounds. *Information Processing Letters*, 71(3-4):155–158, August 1999.

[AW98] H. Attiya and J. Welch. *Distributed Computing. Fundamentals, Simulations, and Advanced Topics.* McGraw-Hill, 1998.

[BDFG03a] R. Boichat, P. Dutta, S. Frølund, and R. Guerraoui. Deconstructing Paxos. *Distributed Computing Column of ACM SIGACT News*, 34(1):47–67, March 2003.

[BDFG03b] R. Boichat, P. Dutta, S. Frølund, and R. Guerraoui. Reconstructing Paxos. *Distributed Computing Column of ACM SIGACT News*, 34(2):42–57, June 2003.

[CBF04] B. Charron-Bost and F. Le Fessant. Validity conditions in agreement problems and time complexity. In *Proceedings of 30th Conference on Current Trends in Theory and Practice of Computer Science (SOFSEM 2004)*, number 2932 in Lecture Notes in Computer Science, pages 196–207. Springer-Verlag, January 2004.

[CBS04] B. Charron-Bost and A. Schiper. Uniform consensus harder than consensus. *Journal of Algorithms*, 51(1):15–37, April 2004.

[CT96] T. D. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2):225–267, March 1996.

[DDS87] D. Dolev, C. Dwork, and L. Stockmeyer. On the minimal synchrony needed for distributed consensus. *Journal of the ACM*, 34(1):77–97, January 1987.

[DFGP02] P. Dutta, S. Frolund, R. Guerraoui, and B. Pochon. An efficient universal construction for message-passing systems. In *Proceedings of the 16th International Symposium on Distributed Computing (DISC-16)*, number 2508 in Lecture Notes in Computer Science, pages 133–147. Springer-Verlag, October 2002.

[DG02a] P. Dutta and R. Guerraoui. Fast indulgent consensus with zero degradation. In *Proceedings of the Fourth European Dependable Computing Conference (EDCC-4)*, number 2485 in Lecture Notes in Computer Science, pages 191–208. Springer-Verlag, October 2002.

[DG02b] P. Dutta and R. Guerraoui. The inherent price of indulgence. In *Proceedings of the 21st ACM Symposium on Principles of Distributed Computing (PODC-21)*, pages 88–97, July 2002. To appear in Distributed Computing.

[DGFHR03] C. Delporte-Gallet, H. Fauconnier, J-M. Hélary, and M. Raynal. Early stopping in global data computation. *IEEE Transactions on Parallel and Distributed Systems*, 14(9):909–921, September 2003.

[DGK04] P. Dutta, R. Guerraoui, and I. Keidar. The overhead of consensus failure recovery. IC Technical Report 200456, École Polytechnique Fédérale de Lausanne, June 2004.

[DGL05] P. Dutta, R. Guerraoui, and L. Lamport. How fast can eventual synchrony lead to consensus? In *The International Conference on Dependable Systems and Networks (DSN), to appear*, 2005.

[DGLC04] P. Dutta, R. Guerraoui, R. Levy, and A. Chakraborty. How fast can a distributed atomic read be? In *Proceedings of the 23rd ACM Symposium on Principles of Distributed Computing (PODC-23)*, pages 236–245, July 2004.

[DGP03] P. Dutta, R. Guerraoui, and B. Pochon. Tight bounds on early local decisions. In *Proceedings of the 17th International Symposium on Distributed Computing (DISC-17)*, number 2848 in Lecture Notes in Computer Science, pages 264–278. Springer-Verlag, October 2003.

[DGP04] P. Dutta, R. Guerraoui, and B. Pochon. Fast non-blocking atomic commit: An inherent tradeoff. *Information Processing Letters (IPL)*, 91(4):195–200, August 2004.

[DGV04] P. Dutta, R. Guerraoui, and M. Vukolić. The complexity of asynchronous Byzantine consensus. IC Technical Report 200499, École Polytechnique Fédérale de Lausanne, November 2004.

[DLM82] R. A. DeMillo, N. A. Lynch, and M. Merritt. Cryptographic protocols. In *Proceedings of the 14th ACM Symposium on Theory of Computing (STOC)*, pages 383–400, May 1982.

[DLS88] C. Dwork, N. A. Lynch, and L. Stockmeyer. Consensus in the presence of partial synchrony. *Journal of the ACM*, 35(2):288–323, April 1988.

[DM90] C. Dwork and Y. Moses. Knowledge and common knowledge in a byzantine environment: Crash failures. *Information and Computation*, 88(2):156–186, October 1990.

[DRS90] D. Dolev, R. Reischuk, and H. R. Strong. Early stopping in byzantine agreement. *Journal of the ACM*, 37(4):720–741, October 1990.

[DS83] D. Dolev and H. R. Strong. Authenticated algorithms for byzantine agreement. *SIAM Journal on Computing*, 12(4):656–666, November 1983.

[FL82] M. J. Fischer and N. A. Lynch. A lower bound for the time to assure interactive consistency. *Information Processing Letters*, 14(4):183–186, June 1982.

[FLP85] M. J. Fischer, N. 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] E. Gafni. Round-by-round fault detectors: Unifying synchrony and asynchrony. In *Proceedings of the 17th ACM Symposium on Principles of Distributed Computing (PODC-17)*, pages 143–152, July 1998.

[GR04] R. Guerraoui and M. Raynal. The information structure of indulgent consensus. *IEEE Transactions on Computers*, 53(4):453–466, April 2004.

[Gra78] J. Gray. Notes on database operating systems. In *Operating systems; an advanced course*, number 60 in Lecture Notes in Computer Science, pages 393–481. Springer-Verlag, 1978.

[Gue95] R. Guerraoui. Revisiting the relationship between non blocking atomic commitment and consensus problems. In *Proceedings of the 9th International Workshop on Distributed Algorithms (WDAG-9)*, number 791 in Lecture Notes in Computer Science, pages 87–100. Springer-Verlag, September 1995.

[Gue00] R. Guerraoui. Indulgent algorithms. In *Proceedings of the 19th ACM Symposium on Principles of Distributed Computing (PODC-19)*, pages 289–298, July 2000.

[Had83] V. Hadzilacos. Byzantine agreement under restricted types of failures (not telling the truth is different from telling lies). Technical Report 19-83, Aiken Computation Laboratory, Harvard University, 1983.

[Had87] V. Hadzilacos. On the relationship between the atomic commitment and consensus problems. In *Proceedings 9th International Workshop on Fault-Tolerant Computing*, number 448 in Lecture Notes in Computer Science, pages 201–208. Springer-Verlag, 1987.

[HR99] M. Hurfin and M. Raynal. A simple and fast asynchronous consensus protocol based on a weak failure detector. *Distributed Computing*, 12(4):209–223, 1999.

[HT93] V. Hadzilacos and S. Toueg. Fault-tolerant broadcasts and related problems. In S. Mullender, editor, *Distributed Systems*, ACM Press Books, chapter 5, pages 97–146. Addison-Wesley, second edition, 1993.

[KR03] I. Keidar and S. Rajsbaum. A simple proof of the uniform consensus synchronous lower bound. *Information Processing Letters*, 85(1):47–52, January 2003. A preliminary version appeared in Distributed Computing Column of ACM SIGACT News 32(2):45-63, 2001.

[Lam78] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.

[Lam89] L. Lamport. The part-time parliament. Technical Report 49, Systems Research Center, Digital Equipment Corp, Palo Alto, September 1989. A revised version of the paper also appeared in ACM Transaction on Computer Systems, 16(2):133-169, May 1998.

[LF82] L. Lamport and M. Fischer. Byzantine generals and transaction commit protocols. Technical Report 62, SRI International, April 1982.

[LSP82] L. Lamport, R. Shostak, and M. Pease. The byzantine generals problem. *ACM Transactions on Programming Languages and Systems*, 4(3):382–401, July 1982.

[Lyn96] N. A. Lynch. *Distributed Algorithms*. Morgan Kaufmann, 1996.

[Mer85] M. Merritt. Notes on the dolev-strong lower bound for Byzantine agreement. *Unpublished manuscript*, 1985.

[MR99] A. Mostefaoui and M. Raynal. Solving consensus using Chandra-Toueg's unreliable failure detectors: A general quorum-based approach. In *Proceedings of the 13th International Symposium on Distributed Computing*, number 1693 in Lecture Notes in Computer Science, pages 49–63. Springer-Verlag, September 1999.

[MR01] A. Mostefaoui and M. Raynal. Leader-based consensus. *Parallel Processing Letters*, 11(1):95–107, March 2001.

[MR02] Y. Moses and S. Rajsbaum. A layered analysis of consensus. *SIAM Journal on Computing*, 31(4):989–1021, 2002.

[MT88] Y. Moses and M. R. Tuttle. Programming simultaneous actions using common knowledge. *Algorithmica*, 3:121–169, 1988.

[PR04] P. R. Parvédy and M. Raynal. Optimal early stopping uniform consensus in synchronous systems with process omission failures. In *Proceedings of the sixteenth annual ACM symposium on Parallelism in algorithms and architectures (SPAA)*, pages 302–310, May 2004.

[PSL80] M. Pease, R. Shostak, and L. Lamport. Reaching agreement in the presence of faults. *Journal of the ACM*, 27(2):228–234, April 1980.

[Ray02] M. Raynal. Consensus in synchronous systems: A concise guided tour. In *9th Pacific Rim International Symposium on Dependable Computing (PRDC 2002)*, pages 221–228. IEEE Computer Society, December 2002.

[Sch90] F. B. Schneider. Implementing fault-tolerant services using the state machine approach: a tutorial. *ACM Computing Surveys*, 22(4):299–319, December 1990.

[Sch97] A. Schiper. Early consensus in an asynchronous system with a weak failure detector. *Distributed Computing*, 10(3):149–157, 1997.

[Ske81] D. Skeen. Nonblocking commit protocols. In *ACM SIGMOD International Conference on Management of Data*, pages 133–142, April 1981.

# Curriculum Vitae

I was born in Bokaro, India. After completing my higher secondary education in Calcutta, I joined the Indian Institute of Technology, Kanpur (IIT Kanpur), where I obtained a Bachelor of Technology in Electrical Engineering in 1999. In the same year, I joined the Graduate School in Computer Science at École Polytechnique Fédérale de Lausanne (EPFL). In 2000, I started my Ph.D. thesis under the supervision of Prof. Rachid Guerraoui in the School of Computer and Communication Sciences.