On the Cost of Fault-Tolerant Consensus When There Are No Faults – A Tutorial*

Idit Keidar† Sergio Rajsbaum‡

May 24, 2001

Abstract

We consider the consensus problem in asynchronous models enriched with unreliable failure detectors or partial synchrony, where processes can crash or links may fail by losing messages. We study the number of communication steps performed by deterministic consensus algorithms for these models in failure-free executions. We show a tight lower bound of two communication steps. In the process of showing this bound, we give a simple unified proof of a number of different impossibility and lower bound results. Thus, we shed light on the relationship among different lower bounds, and at the same time, illustrate a general technique for obtaining simple and elegant lower bound and impossibility proofs. We illustrate the matching upper bound by describing previously published algorithms that achieve the lower bound.

1 Introduction

Consensus is a fundamental problem in distributed computing theory and practice alike. A consensus service allows a collection of processes to agree upon a common value. More specifically, each process has an input, and each correct process must decide on an output, such that all correct processes decide on the same output, and furthermore, this output is the input of one of the processes. Consensus is an important building block for fault-tolerant distributed systems [Lam96]: to achieve fault-tolerance data is replicated, and consensus can be used to guarantee replica consistency using the *state machine* approach [Lam78, Sch90].

Consensus is not solvable in pure asynchronous models where even one process can crash [FLP85][1]. However, real systems are not completely asynchronous. Some partially synchronous models [DLS88, CF99] better model real systems. We consider a particularly realistic kind of model [DLS88] which allows the system to be asynchronous for an unbounded but finite period of time, as long as it eventually becomes synchronous, and less than a majority of the processes can crash. Consensus is solvable in this model. Similarly, consensus is solvable in an asynchronous model enriched with certain oracle failure detectors [CT96], including unreliable failure detectors that can provide arbitrary output for an arbitrary period of time, but eventually provide some useful semantics. In this

*Preliminary version appeared in SIGACT News 32(2), Distributed Computing column, pages 45–63, June 2001.

†MIT Lab for Computer Science. 545 Technology Square, NE43-367, Cambridge, MA 02139, U.S.A. E-mail: idish@theory.lcs.mit.edu.

‡Compaq Cambridge Research Laboratory, One Cambridge Center, Cambridge, MA 02142-1612, U.S.A. E-mail: Sergio.Rajsbaum@compaq.com, rajsbaum@math.unam.mx. On leave from Instituto de Matemáticas, UNAM.

[1]Consensus can be solved in the asynchronous model by randomized algorithms, and in some shared memory models. In this paper, we consider only message-passing, deterministic algorithms.

paper, we study consensus algorithms in the partial synchrony model and in asynchronous models with unreliable failure detectors. We call these models *non-synchronous*.

What can we say about the running time of consensus algorithms for non-synchronous models? Unfortunately, even in the absence of failures, consensus algorithms in these models are bound to have unbounded running times. This is because, by [FLP85], in an asynchronous system, a consensus algorithm can have an infinite run in which no failures occur, and non-synchronous systems can be asynchronous for an unbounded period of time. In practice however, there are often extensive periods during which communication is timely. That is, many executions are actually synchronous. In such executions, failure detectors based on time-outs can be accurate. In this paper, we study the running time of consensus algorithms for non-synchronous models under such benign circumstances. We focus on executions in which, from the very beginning, the network is synchronous or the failure detector is accurate. We will call such executions *well-behaved* if they are also failure-free[2]. Since well-behaved executions are common in practice, algorithm performance under such circumstances is significant. Note, however, that an algorithm cannot know a priori that an execution is going to be well-behaved, and thus cannot rely upon it.

There are several algorithms for non-synchronous models that decide in two communication steps in well-behaved executions (e.g., [Sch97, HR99, MR99, KD96]). Distributed systems folklore suggests that every fault tolerant algorithm in non-synchronous models must take at least two communication steps before all the processes decide, even in well-behaved executions. In this paper we formalize this folklore theorem. Specifically, we show that any consensus algorithm for non-synchronous models where at least two processes can crash will have at least one well-behaved execution in which it takes two communication steps for all the processes to decide.

We start by illustrating the upper bound. We present a simple and intuitive algorithm due to Mostefaoui and Raynal [MR99], which solves consensus in the asynchronous crash failure model with an unreliable failure detector. The failure detector can be implemented in the partial synchrony model in a way that preserves the well-behaved nature of executions. Therefore, the upper bound also applies to the partial synchrony model.

Before deriving the lower bound for non-synchronous models, let us first recall what is known about the running time of consensus algorithms in the synchronous crash failure model: in this model, there are consensus algorithms that, in failure-free executions, decide within one communication step. More generally, early stopping algorithms have all the processes decide within $f+1$ communication steps in every execution involving f crash failures [LF82]. This is optimal: it has been shown that every consensus algorithm for the synchronous crash failure model will have executions with f failures that take $f+1$ communication steps before all the processes decide [LF82, DM90, Lyn96, MR98, AT99]. (We re-prove this lower bound in Section 4).

Why then do consensus algorithms for non-synchronous models require two communication steps even in synchronous executions? We show that the need for an additional communication step (over what is needed in the synchronous model) stems from the fact that, in the non-synchronous models, a correct process can be mistaken for a faulty one. This requires consensus algorithms for these models to avoid disagreement with processes that seem faulty. In contrast, consensus in the synchronous model requires only that correct processes agree upon the same value, and allows for disagreement with faulty ones. The *uniform consensus* problem strengthens consensus to require that every two processes (correct or faulty) that decide must decide on the same value. Interestingly, uniform consensus requires two communication steps in the absence of failures even in the synchronous model, as long as two or more processes can crash, as we prove in Section 4. The

[2]Our notion of well-behaved behaved executions is similar to the notion of *stable* periods, e.g., in the timed-asynchronous model [CF99] and in self-stabilizing systems.

two communication step lower bound for consensus in non-synchronous models stems from the fact that any algorithm that solves consensus in these models, also solves uniform consensus [Gue95], as we prove in Section 5.

In order to prove more general lower bound results, we study a weaker version of consensus, called *weak consensus*. Recall that consensus requires that the decision (output value) be the input of one of the processes; this is the *validity* property of consensus. To define weak consensus, we replace validity with the following *weak validity* property: there exist two failure-free executions in which processes decide on two different values. In [FLP85], it was observed that an even weaker version of validity is sufficient to prove consensus impossibility in an asynchronous model: [FLP85] requires only that there exist two executions, not necessarily failure-free ones, in which processes decide on two different values. Interestingly, this property is too weak to imply the two step lower bound for synchronous uniform consensus in failure-free executions, as we show in Section 4.5.

In addition to crash failures, we consider message omission failures where links can fail by losing messages, including messages sent between correct processes. Consensus is not solvable in the message omission model, even in a synchronous system, unless some restrictions are placed on possible message loss patterns. In particular, consensus is not solvable in this model even with a single *mobile failure*, where the environment (the adversary) can choose a single process in each communication step of the algorithm, and lose some of the messages sent in this step by this process [SW89]. We re-prove this impossibility result in Section 4 using the same technique that we use for showing lower bounds in the synchronous crash failure model. The impossibility result relies on the adversary's ability to choose *any* process from which to lose messages in every step. In Section 3, we discuss certain restrictions that can be placed on the adversary's ability to choose message loss patterns so as to make consensus solvable, although by more complicated algorithms than those that tolerate only crash failures (e.g., [Lam98]).

As in non-synchronous models, in a synchronous message omission model where any number of messages from a correct process may be lost, any algorithm for consensus also solves uniform consensus. Thus, if at least two processes can fail, then the two communication step bound holds. In the omission model, we say that a process fails if, from some point on, all of its messages are lost. A two step bound for the synchronous omission model was previously observed by Lamport [Lam00].

The primary goal of this paper is to provide intuition for the lower and upper bounds. We especially strive to make the lower bound proofs as simple and easy-to-follow as possible: we extract the simplest arguments from known techniques (based on [MR98]), and use these simple techniques to derive new proofs (the uniform consensus lower bound). Moreover, we give a unified proof of three different results, thus illustrating a general technique for obtaining simple and elegant lower bound and impossibility proofs.

The rest of this paper is organized as follows: In Section 2, we define the models and problems, as well as a formal notion of well-behaved executions for non-synchronous models, which we use to state and prove the bounds. In Section 3, we illustrate the upper bound by describing known algorithms that decide within two communication steps in well-behaved executions. In Section 4, we prove the lower bounds for consensus and uniform consensus in the synchronous crash failure model, as well as an impossibility result for consensus in the message omission model with a single mobile failure. Then, in Section 5, we derive the lower bounds for consensus in non-synchronous models. In Section 6 we discusses optimistic alternatives to consensus as a way to circumvent the lower bound. Finally, Section 7 concludes the paper.

2 Definitions

2.1 Models

The universe consists of n processes, $p_1, \ldots, p_n$. Processes communicate by message-passing. We consider deterministic algorithms. An execution of an algorithm is an alternating sequence of global states and actions. Global states reflect the local states of all the processes, and of the communication channels (with messages in transit). Actions are steps taken by distinct processes, and by the environment. In an action, a process can send messages to any number of other processes and change its local state; the environment decides what subset of the messages in transit to deliver to their destinations. Given a deterministic algorithm, there is a single execution of the algorithm for every assignment of initial values to processes and sequence of environment actions.

In this paper we consider several distributed computing models. Each one is defined in terms of the type of failures that can occur, and the timing assumptions. We proceed to describe these two aspects of a model.

2.1.1 Failure types

We consider two types of failures: crash failures and message omission. We do not consider Byzantine failures.

Crash failures

In a *crash failure* model, processes fail by stopping to execute any actions from some point on in an execution. A parameter t bounds the number of processes that can crash. A process that crashes in an execution is *faulty* in that execution, a process that does not crash is *correct*. A process that fails before state x in an execution is *failed* at x. In a model with only crash failures, communication is reliable: if a process p_i sends a message to a correct process p_j in a given step, and p_i does not fail in this step, then the message eventually reaches p_j. If a process does fail in a given step, then any subset of the messages it sends in this step can be lost.

Omission failures

In a *message omission* model, any subset of the messages sent by correct processes can be lost in any communication step. A process is *faulty* in this model if there is a point in the execution after which all the messages from this process are lost. Since the fact that a process is faulty is defined only by examining an infinite execution, in every state of an execution, no process is failed. The *mobile failure model* [SW89] is a restriction of the message omission model, where in each step messages from at most one process can be lost. We show that consensus is unsolvable even in this very restricted omission model.

2.1.2 Timing assumptions

Our focus in this paper is on models of partial synchrony, and on asynchronous models with unreliable failure detectors. In order to derive lower bounds for these models, we consider the more restricted synchronous model. We now describe these three timing models.

Synchronous

In the *synchronous* model, processes execute in synchronous steps[3]. In each step, every processes can send messages to any number of other processes. Unless a failure occurs, each message reaches its destination in the following step.

Partially synchronous

There are several *partially synchronous* models in the literature. We consider one of the models defined in [DLS88], where there is no a priori bound on message latency. However, there is a time, called *global stabilization time (GST)* after which there is a bound on message latency, but this bound is not known to the processes. For simplicity, we assume that local computation time is zero and that processes can measure time accurately[4].

In addition, we assume that there exist a pre-defined constant δ that is known to the processes and reflects the typical message latency in the system. A *well-behaved* execution in the partial synchrony model is an execution in which there are no failures and every message reaches its destination within at most δ time. Thus, a well-behaved execution is essentially synchronous.

Asynchronous with failure detectors

In the *asynchronous model with unreliable failure detectors* [CT96], there is no bound on message latency, and every process p is equipped with a failure detector oracle. At every step of the algorithm's execution, p can query its failure detector for some information. We focus here on failure detectors that can provide arbitrary outputs during an unbounded period of time, but eventually have to provide some meaningful semantics.

Chandra and Toueg [CT96] define several classes of failure detectors whose output is a list of *suspected* processes. We say that process p *suspects* process q at a certain point in the execution, if q is included in p's failure detector's output list at this point. To illustrate the upper bound, we focus on a failure detector of class $\diamond\mathcal{S}$, which has been shown to be the weakest failure detector for solving consensus [CHT92]. A failure detector of this class guarantees two properties:

1. *strong completeness*: there is a time after which every correct process permanently suspects every crashed process; and
2. *eventually weak accuracy*: there is a correct process p such that there is a time after which p is not suspected by any correct process.

A *well-behaved* execution in this model is an execution in which there are no failures and no suspicions, that is, all the processes have empty suspicion lists.

Another example of a failure detector class is the *leader election* service $\diamond\Omega$ [CHT92]. The output of a failure detector of class $\diamond\Omega$ is the identifier of one process, presumed to be the leader. Initially, a failure detector of this class can be unreliable: it can name a faulty process as the leader, and can name different leaders at different processes. However, eventually, it must elect a single correct leader, that is, give all the processes the same output, which must be a correct process. In [CHT92], it is shown that $\diamond\Omega$ is equivalent to $\diamond\mathcal{S}$. In a *well-behaved* execution, a $\diamond\Omega$ failure detector announces the same correct leader at all the processes from the beginning of the execution.

[3] Communication steps are sometimes called rounds.

[4] Consensus can be solved even if these assumptions are weakened, as in the *timed asynchronous* model [CF99], where clock skew is possible but bounded, and different processing times are possible but their ratio is bounded.

2.2 Problem definitions

We now define several variants of the consensus problem. In all of these problems, every process p_i participating in the algorithm gets an input value v_i, and the algorithm's output is a decision value dec_i. For simplicity, we consider *binary* consensus, where the input values are in $\{0, 1\}$.

An algorithm solves *consensus* if it satisfies the following properties:

- *Agreement:* If some correct process p_i decides dec_i and another correct process p_j decides dec_j, then $dec_i = dec_j$. That is, no two correct processes decide upon different values.
- *Validity:* If a correct process p_i decides dec_i, then there is a process p_j so that $v_j = dec_i$, that is, the decision value is the input value of some process.
- *Termination:* Every correct process eventually decides on some value[5].

The agreement property requires correct processes to agree on the same value, but does not require agreement with faulty processes. Uniform consensus strengthens consensus by also requiring agreement with faulty processes, in case they decide before they fail. An algorithm solves *uniform consensus* if it satisfies validity and termination as defined above, along with the following property:

- *Uniform agreement:* If some process p_i decides dec_i and another process p_j decides dec_j, then $dec_i = dec_j$. That is, no two processes decide on different values.

For the sake of the lower bound, we look at a weaker version of consensus, called weak consensus. An algorithm solves *weak consensus* (*weak uniform consensus*) if it satisfies agreement (uniform agreement) and termination, along with the following property:

- *Weak validity:* For every value $v \in \{0, 1\}$ there is a failure-free execution in which some process decides v.

Weak validity may not be a useful property for a consensus algorithm to satisfy, but it is useful for proving lower bounds, as it makes the results more general and thus applicable to more problems. For example, one instance of weak uniform consensus is *non-blocking atomic commit* [Ske81, BHG87]. Atomic commit [Gra78] is used in distributed database systems to have multiple database sites agree whether to commit a transaction or to abort it. The decision 1 represents a commit decision, and 0 – abort. Likewise, the input value 1 represents a vote in favor of committing the transaction, the input value 0 represents a vote for aborting it. A commit decision cannot be reached unless all the processes vote in favor of committing the transaction. The non-blocking atomic commit problem [Ske81, BHG87] is a special case of weak uniform consensus. An algorithm solves non-blocking atomic commit if it satisfies uniform agreement and termination, along with the following two properties:

- *Commit-Validity:* The decision value can only be 1 if all processes voted 1.
- *Non-Triviality:* If there are no failures and all processes vote 1, then the decision will be 1.

[5]In this paper we are interested in when the processes can decide. In some circumstances the time needed to *halt* can be larger.

3 The Upper Bound

There exist various consensus algorithms for non-synchronous models that have all the processes decide within two communication steps in well-behaved executions, e.g., [Sch97, HR99, MR99, KD96]. We illustrate this, in Section 3.1, with a simple and elegant consensus algorithm due to Mostefaoui and Raynal [MR99]. The algorithm works with $\Diamond\mathcal{S}$ in the crash failure model, and in well-behaved executions, all processes decide within two communication steps.

Omission failures are harder to overcome; as we show in Section 4, consensus is not solvable even in a synchronous model with a single mobile omission failure. However consensus can be solved if certain restrictions are placed on message loss patterns. We discuss this in Section 3.2.

A failure detector of class $\Diamond\mathcal{S}$ can be implemented in the partial synchrony model using time-outs by increasing the time-out value every time a false suspicion occurs (see [CT96]). With this implementation, if the time-out value is initially chosen to be δ, the maximum latency in well-behaved executions of the partial synchrony model, then in any well-behaved execution of the partial synchrony model, the failure detector generates no suspicions. Therefore, a well-behaved execution of the partial synchrony model can simulate a well-behaved execution of the failure detector model. Thus, the upper bound also applies to the partial synchrony model.

3.1 Consensus with $\Diamond\mathcal{S}$ and crash failures

Several $\Diamond\mathcal{S}$-based algorithms that decide within two communication steps have been suggested [Sch97, HR99, MR99]. These algorithms tolerate crash failures as long as a majority of the processes are correct, that is, for $t > n/2$. They use the rotating coordinator approach [CM84, DLS88]. With this approach, the algorithm iterates through a sequence of rounds, at each round, a different coordinator leads the consensus algorithm in order to try and reach a decision. In round i, the coordinator is process p_c where $c = (i \bmod n) + 1$. The iteration through multiple rounds is needed in order to account for possible coordinator failures. Once there is a round in which the coordinator is correct and also not suspected by any process, all the processes decide.

For illustration, we present the algorithm of [MR99] in Figure 1. In this algorithm, each process p_i keeps track of the round number in a variable r_i, initially zero, and of the estimated consensus value in a variable est_i, initially the process' input value. A round of this algorithm is conducted as follows: the coordinator sends its own estimate to the other processes. Every other process waits until it either receives the estimate from the coordinator, or suspects the current coordinator. Next, all the processes send messages to each other: if a process received the estimate from the coordinator, it sends this estimate; otherwise, it sends a null value. Finally, each process waits to receive messages from a majority of the processes and then acts as follows:

1. A process that receives only null values proceeds directly to the next round.

2. A process that receives the same non-null value (the coordinator's estimate) from a majority, broadcasts a DECIDE message with this value to all, and decides on this value. That is, the process returns this value.

3. A process that receives both null and non-null values sets its estimate est_i to the received non-null value, and proceeds to the next round.

In addition, once a process receives a DECIDE message, it forwards the DECIDE message to all the processes (to account for the case that the coordinator who sent it failed after sending it), and then decides on the value in the DECIDE message.

```
Function Consensus(v_i)
cobegin
 (1) task T1: r_i ← 0; est_i ← v_i; % v_i ≠ ⊥ %
 (2)           while true do
 (3)               c ← (r_i mod n) + 1; est_from_c_i ← ⊥; r_i ← r_i + 1; % round r_i %
 (4)               case (i = c) then est_from_c_i ← est_i
 (5)                    (i ≠ c) then wait until ( (EST(r_i, v) is received from p_c) ∨ (c ∈ suspected_i) );
 (6)                                 if (EST(r_i, v) has been received) then est_from_c_i ← v endif
 (7)               endcase;   % est_from_c_i = est_c or ⊥ %
 (8)               ∀j do send EST(r_i, est_from_c_i) to p_j enddo;
 (9)               wait until (EST(r_i, est_from_c) has been received from ⌈(n + 1)/2⌉ processes);
 (10)              let rec_i = {est_from_c such that EST(r_i, est_from_c) has been received at line 5 or 9};
                              % est_from_c = ⊥ or v with v = est_c %
                              % rec_i = {⊥} or {v} or {v, ⊥} %
 (11)              case (rec_i = {⊥})    then skip
 (12)                   (rec_i = {v})    then ∀j ≠ i do send DECIDE(v) to p_j enddo; return(v)
 (13)                   (rec_i = {v, ⊥}) then est_i ← v
 (14)              endcase
 (15)          enddo

 (16) task T2: upon reception of DECIDE(v): ∀j ≠ i do send DECIDE(v) to p_j enddo; return(v)
coend
```

Figure 1: A $\diamond\mathcal{S}$-based consensus algorithm.

We now informally argue that the algorithm is correct. For a rigorous proof, see [MR99]. Validity is ensured because the first coordinator suggests its own initial value, and every other coordinator either suggests its own value or a value that was previously sent by another coordinator. Thus, by induction, only processes' initial values are possible decision values.

To see that agreement is ensured, notice that if some process decides v in round r, then a majority of processes must have sent v in round r. Since every process waits to hear from a majority before proceeding to the next round, every process receives at least one message with v in round r. Thus, every process that proceeds to round $r+1$ sets its own estimate to v beforehand. Clearly, from round $r+1$ onward, v is the only possible decision value.

A decision is made once there is a round in which the coordinator does not fail and is also not suspected by any process. The properties of the $\diamond\mathcal{S}$ failure detector, together with the fact that the algorithm continuously iterates over all possible coordinators until some process decides, ensure that eventually there is such a round. In addition, once some process decides, every process that decides broadcasts a DECIDE message to all the other processes. This guarantees that if a correct process stops participating in the iterations because it has already decided, all correct processes get its DECIDE message and also decide.

Now, let us consider a well-behaved execution. If the first coordinator does not fail and is also not suspected by any process, then every correct process receives the coordinator's estimate and sends it to the other processes, and no process sends a null value. Thus, all the processes decide after receiving messages from a majority in the first round, that is, after two communication steps.

3.2 Consensus with messages omissions

Omission failures are harder to overcome. As we will see in the next section, consensus is not always solvable in the message omission model. However, as we now discuss, it is possible to overcome certain restricted message omission patterns.

Consider, for example, the case that all the communication links among correct processes are *fair* [ACT99], in the sense that if a message is sent on a link an infinite number of times it eventually reaches its destination. In this case, a reliable transport protocol, similar to TCP, can be implemented atop the fair-lossy links by repeatedly retransmitting each message until it is known that the message reached all processes [ACT99]. Thus, one can simulate a crash failure model with fair-lossy links, and any algorithm for solving consensus with crash failures, e.g., the one in the previous section, works in this model as well, as long as there is a majority of correct processes.

Next, consider the case that communication among correct processes is eventually reliable, but before that, there are network partitions that last a substantial amount of time. In this case, continuously sending messages infinitely many times as described above is not practical; practical transport protocols like TCP "give up" and terminate the session during long-lasting partitions. Practical algorithms for this model therefore need to be able to reach a decision once communication is re-established, *without* retransmitting the entire message history. The first and best-known such algorithm is Paxos [Lam98]; other similar algorithms appear in [KD96, KD98, DFKM96]. These algorithms are quite subtle, and we do not describe them in detail here.

Lamport's Paxos algorithm [Lam98, DPLL97] solves consensus in the asynchronous message omission model, with the following restriction on message loss patterns: there is a time after which there is a process p so that p's messages to correct processes are not lost, and moreover, there is a majority of correct processes that can communicate reliably with p. Paxos uses a leader election service, which is, in essence, a failure detector of class $\Diamond\Omega$. The E3PC algorithm [KD98] solves atomic commit with similar assumptions. The consensus algorithm of [DFKM96] uses a $\Diamond\mathcal{S}$ failure detector. The Corel algorithm [KD96] uses a group communication service, which is equivalent to an eventually perfect failure detector [CKV], which is stronger than $\Diamond\mathcal{S}$ or $\Diamond\Omega$.

Let us examine the running time of Paxos in well-behaved executions, that is, in executions in which there are no failures, and the leader election service elects a single leader at all the processes from the beginning of the execution. In such executions, Paxos, as presented in [Lam98], requires five communication steps before all processes decide. The first two steps are needed for recovery from past failures, and they do not involve sending actual consensus input values. When a sequence of Paxos consensus algorithms are ran, these two steps have to occur only at the beginning and when the leader changes. In a given instance of consensus that is invoked after these two steps, only three communication steps are executed. These three steps work in a manner very similar to a round of the $\Diamond\mathcal{S}$-based consensus algorithm presented above.

In the first of these steps, the leader sends its own initial value to all the processes. In the second, all the processes that receive this value record it and send acknowledgments to the leader. Once a majority of the processes has recorded a value, the value is *locked* and no other decision value is possible. Upon receiving acknowledgments from a majority, the leader decides. In the third step, the leader sends the decision value to all the processes and they also decide.

A simple variation on Paxos can merge these two steps, by having all the processes send acknowledgments to each other. In this case, every process that receives acknowledgments from a majority can decide directly. (This approach was taken in [KD96]). However, this requires strengthening the restriction on message loss patterns, to require that a majority of processes all communicate reliably with each other, not just with the leader.

4 Synchronous Lower Bounds

In this section, we consider a synchronous system where at most t processes can fail, $t < n$. We denote by f the actual number of failures in an execution. We give a unified proof for three different results: First, we show consensus impossibility for the mobile failure model (in this model, $t = 1$). Next, we consider the crash failure model. We show lower bounds of $f+1$ and $f+2$, (for $f \leq t-2$) communication steps for consensus and uniform consensus, respectively. Most of these results are known; in Section 4.1 we discuss related work.

We prove the three results jointly: In Section 4.2 we present the technique, notation, and definitions. In particular, we define notions of *similarity* of a pair of states, and *similarity connectivity* of a set of states. In Section 4.3 we show that a uniform consensus algorithm cannot reach a decision in one communication step from some states in a similarity connected set. By showing that the set of initial states for consensus (uniform consensus) is similarity connected, we obtain the lower bound for uniform consensus for the failure-free case, that is, when $f = 0$. Then, in Section 4.4, we prove that states of longer executions are also connected, from which all three results follow – the impossibility result for consensus in the mobile failure model, and the general bounds for consensus and uniform consensus in the crash failure model.

For the sake of proving the lower bounds and the impossibility result, it suffices to assume the weak validity property, that is, we prove the bounds for weak consensus and weak uniform consensus. Thus, the result for weak uniform consensus also applies to non-blocking atomic commit (as discussed in Section 2.2). In Section 4.5, we show that the result for uniform consensus does not hold with a weaker validity property as defined in [FLP85].

4.1 Related work

The impossibility of consensus in the mobile failure model was first shown by Santoro and Widmayer [SW89]. It was reproved by Moses and Rajsbaum [MR98] using the layering technique described here.

It has been previously shown that $f+1$ steps are necessary and sufficient for consensus [LF82, Lyn96]. Charron-Bost and Schiper [CBS00] show a tight bound of $f+2$ steps for uniform consensus when $f \leq t-2$, and $f+1$ steps for $f = t-1$ or $f = t$. However, their lower bound proof does not cover the case of $f = 0$.

Dwork and Skeen [DS83] give a related result for the non-blocking atomic commit problem, which is an instance of weak uniform consensus. They consider only the failure-free case, that is, when $f = 0$. Their complexity analysis uses a different measure than the one we use here: they model the sending of k messages in a single communication step as costing $O(k)$ time. Clearly, the time bounds obtained this way are higher. However, the bound of two communication steps in failure-free executions implicitly appears in their paper.

Lamport [Lam00] shows a two communication steps bound for uniform consensus in the message omission model where processes do not crash, assuming the adversary is restricted so that consensus is solvable.

Dolev et al. [DRS90] show that a consensus algorithm that can tolerate up to $t < n-1$ crash failures will have executions with f failures involving a sequence of $min(f+2, t+1)$ communication steps before all the processes *halt*. In contrast, we analyze here the time until all processes decide, which may be earlier than the time until all processes halt.

The traditional proofs of the $f+1$ lower bound for consensus (e.g., [LF82, DM90, Lyn96]) and of the $f+2$ lower bound (for $f > 0$) for uniform consensus [CBS00] are based on an involved backward induction argument. Recently, simpler proofs of the $f+1$ lower bound for consensus based on forward induction were presented by Moses and Rajsbaum [MR98] and by Aguilera and Toueg [AT99]. Bar-Joseph and Ben-Or [BJBO98] used a similar technique to show a lower bound for randomized synchronous consensus. Moses and Rajsbaum [MR98] used the same technique to also show lower bounds and impossibility results for different asynchronous message passing and shared memory models. Similar forward arguments that apply to higher dimensional notions of connectivity and to problems other than consensus appear in [HRT98, HRT01].

The proofs of [MR98, AT99, BJBO98, SW89] use a notion of *bivalency*; a state of an algorithm is *bivalent* if it can lead to two different decision values in different executions. The proofs show, by forward induction, how to construct executions in which the algorithm remains in a bivalent state for a sequence of f steps, thus precluding decision. This notion of bivalency, however, is not strong enough to prove the $f+2$ lower bound for uniform consensus. We show, in Section 4.5, that merely having an initial bivalent state does not preclude decision in one communication step in failure-free executions. Therefore, instead of basing the proof on the existence of a bivalent state as in [MR98, AT99, BJBO98], we base our proof on the existence of two states that lead to different decision values in failure-free executions. Using this new notion, we give, for the first time, a proof by forward induction for the uniform consensus lower bound, which is also the first proof covering the case $f = 0$. We further show how to use the same forward induction argument, based on the *layering* approach of Moses and Rajsbaum [MR98], to prove lower bounds for both consensus and uniform consensus, as well as the impossibility of consensus in the mobile failure model.

4.2 The technique: layering, connectivity, coloring

Proving lower bound and impossibility results can be thought of as showing that, for any given algorithm, an adversary always has a strategy that can guarantee a desired bad behavior. In the case of the synchronous model, an adversary can choose which processes fail in each step, and what subsets of their messages are delivered. We say that such a decision is an *action of the environment*. Assume a (deterministic) consensus algorithm is given. For any initial state of the system, the environment actions completely define an execution: the environment defines the messages that a process receives in each step, and the algorithm of the process defines its new local state and the messages it sends next. Thus, we can describe an execution by specifying an initial state and a sequence of environment actions.

Given an execution R (consisting of one or more states), let us denote by $R \cdot \epsilon$ the final state of the execution that results from extending R by having the environment perform the action ϵ. Thus, every execution state can be represented in the form: $x \cdot \epsilon_1 \cdot \epsilon_2 \cdots$ where x is an initial state of the algorithm and ϵ_i is an environment action, for every integer $i \geq 1$.

Layering

To obtain an impossibility result, it is often sufficient to consider only a subset of all possible executions of the model. We consider the set of executions generated from a set of initial states by a *layering* [MR98], where a layering is a set of environment actions that can be performed at a state of the system. If the layering is chosen appropriately, these executions can have structural properties that will simplify their analysis[6].

For the sake of proving the results in this paper, we use layerings in which at most one process fails in every step. We use $[k]$ to denote the set $\{p_1, \ldots, p_k\}$, with $[0]$ denoting the empty set. The layering consists of nondeterministically choosing a process $p_i \in \{p_1, \ldots, p_n\}$ and a set $[k]$ and performing the action $(i, [k])$, meaning that any messages sent from p_i to members of $[k]$ will be lost, and only these messages will be lost. The action $(i, [0])$, for any i, models a step in which no

[6] In asynchronous models this is more significant: various layering structures can be defined to show that consensus is impossible even in executions with very little asynchrony[MR98].

failure occurs. More precisely, the layering for the mobile failure model is:

$$\mathsf{L}_m = \{(i,[k]) : 1 \le i \le n \text{ and } 0 \le k \le n\}.$$

In the crash failure model, the action $(i,[k])$ for $k > 0$ causes process p_i to permanently crash. Since we assume that at most t processes can crash, $t < n$, care must be taken not to crash more than t processes in an execution. Thus, the action (layer) $(j,[k])$ for $k > 0$ is applicable to a state x only if fewer than t processes are failed at x, and process p_j is not failed at x. We denote the layering consisting of all actions of these types by L_c. Notice that the number of processes that fail in an execution of this system is at most t: once t processes are failed at a state, all the subsequent layers have $k = 0$.

Given a layering L, we denote: $\mathsf{L}(x) = \{x \cdot \ell | \ell \in \mathsf{L}\}$. We generalize this definition for a set of states X, as follows: $\mathsf{L}(X) = \cup_{x \in X} \mathsf{L}(x)$. We define L^k to be the application of L k times. Formally, L^k is defined recursively as follows:

$$\begin{aligned} \mathsf{L}^0(X) &= X \\ \mathsf{L}^k(X) &= \mathsf{L}(\mathsf{L}^{k-1}(X)). \end{aligned}$$

The layering we consider for the mobile failure model has the property that in every state, there are no failed processes. In other words, for any state x, the system contains an execution extending x in which no process fails.

Connectivity and coloring

Lower bound and impossibility proofs for distributed algorithms are typically based on a notion of *similarity* that captures the case that different executions look the same to some processes. In this paper, we define similarity as follows:

Definition 4.1 *States x and y are* similar, *denoted by $x \sim y$, if there is a process p_j that is non-failed in these states, such that (a) the states x and y are identical except in the local state of p_j, and (b) there exists $p_i \ne p_j$ that is non-failed in both x and y. (In the mobile failure model, (b) always holds). A set X of states is* similarity connected *if for every $x, y \in X$ there is a sequence of states $x = x_0, x_1, \cdots, x_n = y$ so that $x_i \sim x_{i+1}$ for all $0 \le i < n$.*

The significance of similarity is that if two states are similar, there is at least one correct process p_i that cannot distinguish between them. In particular, process p_i will have to act the same way in both states, so, for example, p_i cannot decide 1 in one of these states and 0 in another.

To obtain the lower bounds, we will first show that the set of initial states for a consensus algorithm is similarity connected. We will then construct additional layers – corresponding to end states of longer executions – that are also similarity connected. This connectivity is a characteristic of the model, not of a particular algorithm or problem. To prove lower bounds for consensus, we observe that in a similarity connected set there must be states in which no decision has been made. For uniform consensus, we observe that in a similarity connected set there must be states from which no decision can be been made within one step. To derive these results, we view a similarity connected set as a connected graph, and use a *coloring function* to assign future decision values to nodes in the graph. Specifically, we use the following new coloring function.

Given a state x, consider the execution extending x in which no failures occur after state x, that is, the layer $(j,[0])$ is repeatedly executed in all steps after x. Note that if an algorithm solves weak consensus, then all correct processes must decide upon the same value in this execution. We denote this decision value by $val(x)$; this is our coloring function.

4.3 The uniform consensus failure-free lower bound

We denote by Con_0 the set of initial states in which no process is failed. Such an initial state is determined by the inputs of the processes: for each combination of 0's and 1's there is an initial state in Con_0. The following well-known lemma shows that Con_0 is similarity connected.

Lemma 4.1 Con_0 *is similarity connected.*

Proof: Given a state z we denote by z_j the local state of process p_j in the state z. Let y, y' be two states in Con_0. For every $0 \leq m \leq n$, define x^m by setting $x_j^m = y_j$ for all $j > m$ and $x_j^m = y'_j$ for all $j \leq m$. We get: $x^0 = y$ and $x^n = y'$. Note that x^{m-1} and x^m differ exactly in the local state of process p_m. Since all the processes are non-failed in every state in Con_0, these states are similar, that is, $x^{m-1} \sim x^m$. ■

We now use the coloring function, val, to show that a uniform consensus algorithm cannot reach decision in one communication step from some states in a similarity connected set.

Lemma 4.2 *Let X be a similarity connected set of states of a uniform consensus algorithm for the crash failure model. Assume that there are states $x, x' \in X$ so that $val(x) \neq val(x')$. Assume further that in every state in X there are at least three correct processes, of which two can fail. Then, there is a state $y \in X$, and a failure-free extension of y in which it takes two additional communication steps for all processes to decide.*

Proof: Since X is similarity connected, there are two similar states $y, y' \in X$ so that $val(y) \neq val(y')$. Assume, without loss of generality, that $val(y) = 1$. The states y and y' are identical except in the local state of one process p_j and there are at least two other non-failed processes in y and y'. Let p_m be the non-failed process, other than p_j, with the highest identifier in y and y' (that is, for all $k > m, k \neq j$, p_k is failed at y, y').

Assume by way of contradiction that every correct process decides after one step in every failure-free extension of y and y'.

Consider the state $y \cdot (j, [0])$ (where nobody fails in the next step). In this state, all processes must decide 1. Since two more processes can fail at states in X, we can add the layers $(j, [m-1]) \cdot (m, [n])$ to x, that is, we can extend x by crashing two more processes, p_j and p_m. Consider the state $y \cdot (j, [m-1])$. Process p_m has exactly the same local state in $y \cdot (j, [m-1])$ as in $y \cdot (j, [0])$. Thus, p_m decides 1 in $y \cdot (j, [m-1])$. By the uniform agreement property, in every extension of $y \cdot (j, [m-1])$ all the correct processes have to decide 1.

By a symmetric argument for y', p_m decides 0 in $y' \cdot (j, [m-1])$, and in every extension of $y' \cdot (j, [m-1])$ all the correct processes have to decide 0. In particular, in any extension of $y \cdot (j, [m-1]) \cdot (m, [n])$ all the correct processes have to decide 1, and in any extension of $y' \cdot (j, [m-1]) \cdot (m, [n])$ all the correct processes have to decide 0.

But no correct process other than p_m hears from p_j, so in $y \cdot (j, [m-1]) \cdot (m, [n])$ and $y' \cdot (j, [m-1]) \cdot (m, [n])$ all the correct processes have the same state. A contradiction. ■

By plugging in Con_0 for X in the above lemma, we get the following theorem:

Theorem 4.3 *Let $2 \leq t < n$. Every t-resilient weak uniform consensus algorithm must perform two communication steps in some failure-free execution before all the processes decide.*

Proof: By Lemma 4.1, Con_0 is similarity connected. By the weak validity property, there are two states $x, x' \in \mathsf{Con}_0$ so that $val(x) \neq val(x')$. Since $2 \leq t < n$, in every state in Con_0 there are at least three correct processes, of which two can fail. Therefore, by Lemma 4.2, there is a failure-free execution in which it takes two communication steps for all processes to decide. ■

We have seen that the lower bound for the case $f = 0$ follows from the fact that Con_0 is similarity connected. In order to prove the general case, we will prove below that $\mathsf{L}_c^k(\mathsf{Con}_0)$ is similarity connected for all $k \leq t$.

4.4 The general bounds: A connectivity-based unified proof

The following lemma applies to algorithms for solving *any* problem in the synchronous crash and mobile failure models, not just consensus or uniform consensus. The lemma shows that in these models, if we begin from a set of similarity connected states where at least one process can fail, and apply all possible layers from L_m or L_c to this state, we get another similarity connected set. When beginning in a set of states, X, in the crash failure model, we can add such layers as long as the number of failures does not exceed t; in the mobile failure model, we can add such layers indefinitely.

Lemma 4.4 *Let X be a similarity connected set of states, and let L be either L_m or L_c. Assume that for every state $x \in X$, the number of processes that are failed in x is smaller than t. (In the mobile failure model, this assumption always holds since no process is failed). Then, $\mathsf{L}(X)$ is also similarity connected.*

Proof: We first show that for every $x \in X$, $\mathsf{L}(x)$ is similarity connected. Recall that for every pair of processes p_i, p_j that are non-failed at x, the states $x \cdot (i, [0])$ and $x \cdot (j, [0])$ are the same, since there are no failures in either extension of x. In addition, for every $k > 0$, the states x_{k-1} and x_k are either similar (because they differ only in the state of p_k), or are identical (in case p_k is faulty or in case p_i did not send a message to p_k in this step).

It is left to show that if $x \sim x'$, then there are states $y \in \mathsf{L}(x)$, $y' \in \mathsf{L}(x')$, so that $y \sim y'$, and the lemma will follow. To see this, recall that the states x and x' are identical except in the local state of one process p_i, and there exists another process that is non-failed in both states. Since the number of processes that are failed in x and x' is smaller than t, it is possible for p_i to crash. Consider the states $y = x \cdot (i, [n])$ and $y' = x' \cdot (i, [n])$. All the non-failed processes other than p_i have the same local states in y and y', and since $t < n$, there is at least one such process. Therefore, $y \sim y'$. ■

In order to apply the topological result above to consensus, we replace X with Con_0. The following lemma will allow us to base impossibility results on $\mathsf{L}^k(\mathsf{Con}_0)$:

Lemma 4.5

1. *$\mathsf{L}_m^k(\mathsf{Con}_0)$ is similarity connected for all k; $\mathsf{L}_c^k(\mathsf{Con}_0)$ is similarity connected for all $k \leq t$.*
2. *For every $k \geq 0$, there exist $y, y' \in \mathsf{L}^k(\mathsf{Con}_0)$ so that $val(y) \neq val(y')$, where L is L_m or L_c.*

Proof:

1. The proof is by induction on k. *Base:* By Lemma 4.1, Con_0 is similarity connected. Moreover, in every state in Con_0, no processes are failed. *Inductive step:* Assume that $\mathsf{L}^{k-1}(\mathsf{Con}_0)$ is similarity connected (where L is either L_m or L_c). By construction of L, in every state in $\mathsf{L}^{k-1}(\mathsf{Con}_0)$, at most $k-1 < t$ processes are failed. By Lemma 4.4, $\mathsf{L}^k(\mathsf{Con}_0)$ is also similarity connected.

2. By weak validity, there exist two states $x, x' \in \mathsf{Con}_0$ so that $val(x) \neq val(x')$. Let y, y' be the states obtained by applying the layer $(i, [0])$ k times to x, x', respectively. Since y, y' are failure-free extensions of x, x', we get that $val(y) \neq val(y')$, and they are both in $\mathsf{L}^k(\mathsf{Con}_0)$. ■

We now use the coloring function, val, to show that in a similarity connected set, there are states in which no process decides.

Lemma 4.6 *Let X be a similarity connected set of states of a consensus algorithm for the crash or mobile failure model. Assume that there are states $x, x' \in X$ so that $val(x) \neq val(x')$, then there is some state in X in which some correct process has not decided yet.*

Proof: Since X is similarity connected, there are two similar states $y, y' \in X$ so that $val(y) \neq val(y')$. The states y and y' are identical except in the local state of one process p_j and there exists another correct process $p_i \neq p_j$. The local state of p_i is the same at y and y', thus, if p_i has already decided upon some value v at y, p_i must have decided upon the same value at y'. But if p_j does not fail, by $val(y) \neq val(y')$, we get that p_i cannot have decided upon the same value at y and y'. A contradiction. ■

We are now ready to prove the impossibility and general lower bound results.

Theorem 4.7 *There is no 1-resilient weak consensus algorithm in the mobile failures model.*

Proof: By Lemma 4.5, there exist $y, y' \in \mathsf{L}_m^k(\mathsf{Con}_0)$ so that $val(y) \neq val(y')$, and $\mathsf{L}_m^k(\mathsf{Con}_0)$ is similarity connected, so by Lemma 4.6, there is a correct process that has not decided in y and y'. Thus, for each k, there is an execution with k steps in which some correct process does not decide.

By König's Lemma [Kön36], using the fact that the layering is finite, there exists an infinite execution in which some correct process does not decide, violating the termination requirement. ■

Theorem 4.8 *Consider an algorithm for weak consensus in the synchronous crash model with up to t failures, where $t < n$. For every $0 \leq f \leq t$, there exists an execution of the algorithm with f failures, in which at the end of step f, some correct process does not decide. That is, decision requires $f + 1$ steps.*

Proof: Fix f, $0 \leq f \leq t$. By Lemma 4.5, there exist $y, y' \in \mathsf{L}_c^f(\mathsf{Con}_0)$ so that $val(y) \neq val(y')$ and $\mathsf{L}_c^f(\mathsf{Con}_0)$ is similarity connected, so by Lemma 4.6, there is a correct process that has not decided in y and y'. Thus, there is an execution with f steps and at most f failures, in which some correct process does not decide. ■

We showed that $f+1$ steps is a lower bound for consensus in the crash failure model in executions with f failures. As mentioned above, this bound is tight [LF82]. For uniform consensus, in case $f \leq t - 2$ (so $t > 1$), an extra step is needed:

Theorem 4.9 *Consider an algorithm for weak uniform consensus in the synchronous crash model with up to t failures, where $t < n$. For every $0 \leq f \leq t - 2$, there exists an execution of the algorithm with f failures, in which at the end of step $f + 1$, some correct process does not decide. That is, decision requires $f + 2$ steps.*

Proof: Fix f, $0 \leq f \leq t-2$. By Lemma 4.5, there exist $y, y' \in \mathsf{L}_c^f(\mathsf{Con}_0)$ so that $val(y) \neq val(y')$, and $\mathsf{L}_c^f(\mathsf{Con}_0)$ is similarity connected. Since $f \leq t-2$, and $t < n$, in every state of $\mathsf{L}_c^f(\mathsf{Con}_0)$ there are at least three correct processes two of which can fail. Therefore, by Lemma 4.2, there is a state $z \in \mathsf{L}_c^f(\mathsf{Con}_0)$, and a failure-free extension of z in which it takes two additional communication steps for all processes to decide. That is, there is an execution with at most f failures in which it takes $f+2$ communication steps for all processes to decide. ■

4.5 On the weak validity property

As mentioned in Section 2.2, the weak validity property we use is stronger than the weak validity property defined in [FLP85]. Our weak validity property requires that there be two failure-free executions that lead to two different decision values, while the property in [FLP85] requires merely that there exist two executions (not necessarily failure-free) that lead to two different decision values. Is the stronger property really needed? It turns out that the weaker property, as defined in [FLP85], is sufficient for the $f+1$ lower bound for consensus and also for the impossibility of consensus in the mobile failure model. The proof given in [MR98] for these two results actually works with the weaker weak validity property. However, the weaker weak validity property of [FLP85] does not suffice for the $f+2$ lower bound for weak uniform consensus, nor for the $f+2$ lower bound on *halting* in consensus due to [DRS90], as we now show.

In Figure 2, we give a counter-example algorithm that satisfies uniform agreement, termination, and weak validity as defined in [FLP85]. It decides and halts after one communication step in every failure-free execution. The counter-example algorithm uses a uniform consensus algorithm that satisfies (strong) validity. (We know that an algorithm for uniform consensus in this model exists [CBS00]). The uniform consensus algorithm is invoked by calling *Uniform_Consensus(v)*.

Function Counter_Example(v_i)
Step 0: Send a message to all the processes (including p_i).
Step 1: **let** S_1 be the set of processes from which Step 0 messages have been received.
  if $|S_1| = n$ **then return**(1) **else** *send* a message to all the processes **endif**
Step 2: **let** S_2 be the set of processes from which Step 1 messages have been received.
  if $|S_2| < |S_1|$ **then** *init* $\leftarrow 1$ **else** *init* $\leftarrow 0$ **endif**
  return *Uniform_Consensus(init).*

Figure 2: A counter-example algorithm for weaker weak validity.

It is easy to see that the counter-example algorithm decides and halts after one step in all failure-free executions. We now argue that the algorithm solves weak uniform consensus with the weakened validity property.

To show uniform agreement, we first observe that if some process p_i decides 1 in Step 1, then all the processes that run uniform consensus have 1 as their input value for uniform consensus. This is because i does not send a Step 1 message, and thus, for any process that does execute step 2, $|S_2| < |S_1|$ because $p_i \in S_1$ and $p_i \notin S_2$. By the (strong) validity property of uniform consensus, the only possible decision value in this case is 1. If no process decides in Step 1, then the uniform consensus algorithm ensures uniform agreement.

Termination is implied by termination of the uniform consensus algorithm, since all the correct processes either decide in Step 1 or run the uniform consensus algorithm.

The algorithm satisfies the weakened weak validity property of [FLP85], that is, both 0 and 1 are possible decision values: If there is at least one failure, but all the failures are initial failures, that is, every process that fails does not send any messages at all, the decision is 0. If there are no failures at all, the decision is 1.

5 Deriving Lower Bounds for Non-Synchronous Models

We now use the results of the previous section to derive lower bounds for consensus in the partial synchrony model and in the asynchronous model with unreliable failure detectors. We do this in two steps: first, in Section 5.1 we derive a lower bound for uniform consensus in these models; next, in Section 5.2 we show that any algorithm that solves consensus in these models also solves uniform consensus. The lower bound for consensus follows.

5.1 From synchronous to asynchronous bounds

In the previous section, we considered a synchronous model where up to t processes may fail. In that model, we have shown (Corollary 4.3) that any uniform consensus algorithm for the synchronous model where at least two processes can fail must perform two communication steps in some failure-free execution before all the processes can decide. This lower bound also applies to well-behaved executions in the partial synchrony model and in the asynchronous model with failure detectors as we show in the two lemmas below.

Lemma 5.1 *For every algorithm that solves uniform consensus in the partial synchrony model where at least two processes can crash, there is a well-behaved execution in which the algorithm performs at least two communication steps before all the processes decide.*

Proof: Assume the proposition is false, then there exists a uniform consensus algorithm that tolerates two crashes and decides after at most one communication step in every well-behaved execution, that is, in every failure-free synchronous execution. Clearly, this algorithm solves uniform consensus in the synchronous model, and decides within one communication step in every failure-free execution. A contradiction to Corollary 4.3. ∎

Lemma 5.2 *For every algorithm that solves uniform consensus in the asynchronous model enriched with failure detector $\diamond\mathcal{S}$ or $\diamond\Omega$, where at least two processes can crash, there is a well-behaved execution in which the algorithm performs at least two communication steps before all the processes decide.*

Proof: As above, it suffices to show that we can implement $\diamond\mathcal{S}$ and $\diamond\Omega$ failure detectors in the synchronous model, so that their execution is well-behaved in failure-free executions of the synchronous model.

We construct a $\diamond\mathcal{S}$ failure detector using the following algorithm: The algorithm is conducted in synchronous communication steps. In every step, every process sends a message to every other process, and waits δ time until the next step. Initially, every process p has an empty suspect list. At every subsequent communication step, p suspects processes from which it did not get messages in this step. Clearly, in the synchronous model, this failure detector suspects a process if and only if the process has crashed; this is the *perfect* failure detector defined in [CT96]. In particular, it is also a failure detector of class $\diamond\mathcal{S}$. Moreover, in failure-free executions, this failure detector does not suspect any process. Thus, a failure-free execution of the synchronous model simulates a well-behaved execution of the asynchronous model with a $\diamond\mathcal{S}$ failure detector.

To implement a $\Diamond\Omega$ failure detector, we modify the above algorithm to have each process output the process with the lowest identifier among the processes it does not suspect. Since the failure detector suspects a process if and only if the process has crashed, the failure detector at each process outputs the first non-crashed process (in lexicographical order). Thus, after the last failure occurs, all the processes elect the same correct leader. In failure-free executions, the leader is always process 1, from the beginning of the execution. Thus, this is a well-behaved execution of $\Diamond\Omega$. ■

Note that the above lower bound for uniform consensus holds for any failure detector that can be implemented in a partial synchrony model in a manner that preserves the well-behaved nature of executions.

5.2 From consensus to uniform consensus

Guerraoui [Gue95] proves that any algorithm that solves consensus in an asynchronous model enriched with certain classes of failure detectors, including $\Diamond\mathcal{S}$, also solves uniform consensus. Below, we adapt Guerraoui's proof to the models we consider in this paper, namely, the partial synchrony model, and the asynchronous model enriched with any failure detector that can give arbitrary output for an arbitrary prefix of the execution.

Lemma 5.3 *In a partial synchrony model or in an asynchronous model with a failure detector that can give arbitrary output for an arbitrary prefix of the execution, any algorithm that solves consensus also solves uniform consensus.*

Proof: Assume that there is an algorithm that solves consensus but not uniform consensus in one of the aforementioned models. Then, the algorithm has an execution e in which two processes p_i and p_j decide on different values, and at least one of them fails after deciding. Assume, without loss of generality, that p_i decides before p_j. Let t_i be the point in the execution e at which p_i decides, and t_j, the point at which p_j decides. Note that until point t_i, neither p_i nor p_j fails, and until t_j, p_j does not fail.

We now construct an execution e' in which both p_i and p_j are correct: until point t_i, e is identical to e'. Between t_i and t_j, p_i may send in e' messages that it does not send in e (because it may be crashed during part of this time in e and not in e'). We delay the receipt of such messages by any process until after point t_j in the execution[7]. In addition, we have the failure detector oracles at all the processes give the same outputs in the prefix of e' until t_j as in the prefix of e until t_j. After time t_j, we have the failure detector satisfy the required semantics. Since the failure detector can give arbitrary output for an arbitrary prefix of the execution, we can delay this prefix until time t_j.

Observe that the prefix of e' until point t_i is indistinguishable for process p_i from the prefix of e until point t_i, because e and e' are identical until that point. Therefore, p_i decides in e' at time t_i the same value that it does in e. Observe also that for every process other than p_i, e' is indistinguishable from e until time t_j. Therefore, p_j decides in e' at time t_j the same value that it does in e. Thus p_i and p_j decide upon different values in e', although both are correct. A contradiction. ■

We conclude with the following theorem:

[7] Recall that in asynchronous or partial synchrony models, the environment can delay messages arbitrarily long for an arbitrary prefix of the execution, that is, the global stabilization time (GST) can occur after t_j.

Theorem 5.4 *Consider a model of partial synchrony or an asynchronous model enriched with a $\diamond\mathcal{S}$ or $\diamond\Omega$ failure detector, where at least two processes can crash. For every algorithm that solves consensus in these models, there is a well-behaved execution in which the algorithm performs at least two communication steps before all the processes decide.*

Note that, more generally, Theorem 5.4 holds for an asynchronous model enriched with any failure detector that satisfies the following two conditions:

1. the failure detector can be implemented in a partial synchrony model in a manner that preserves the well-behaved nature of executions; this is required in Lemma 5.2; and

2. the failure detector can provide arbitrary output for an arbitrary prefix of any execution, as required in Lemma 5.3.

6 On an Optimistic Note

We have seen that in practical models where messages may be occasionally lost or delayed, fault-tolerant consensus algorithms require two communication steps even in well-behaved executions with synchronous communication and no failures or false failure suspicions. This is in contrast to algorithms for consensus in the synchronous crash failure model, which can decide in one communication step in failure-free executions. The additional communication step is required in non-synchronous models due to the fact that, in these models, correct processes may be mistaken for faulty ones. We observe that the additional communication step is needed in order to avoid disagreement with processes that are incorrectly suspected to have failed. This observation can give insight to the possible gains of using an *optimistic* approach in lieu of consensus, as we explain below.

The agreement property of consensus is conservative: it requires that two correct processes never decide on different values. Thus, when used for state machine replication [Lam78, Sch90], consensus never introduces inconsistencies among correct replica. In other words, consensus guarantees *serializability*, that is, in every execution, all the replica exhibit the same sequence of states. In contrast, an optimistic approach allows replica to temporarily diverge, as long as conflicts can later be detected and reconciled. Optimistic approaches can be used if the application can live with temporary conflicts. These approaches are generally cost-effective if conflicts are rare.

Group communication systems [ACM96, CKV], such as Isis [BSS91], take an optimistic approach to state machine replication: they implement algorithms for totally ordered multicast that operate in a single communication step. For example, the algorithms of Isis [BSS91] and Amoeba [KT91] use leader election as follows: The leader sends its own proposed value to all the processes, and immediately decides upon this value. All the recipients of the leader's message immediately decide upon the suggested value. As long as the leader is not suspected by any of the processes, consistency is preserved. When the leader is suspected, a new leader is elected, and the new leader attempts to learn the decision value from the processes it can communicate with. As long as there are no false suspicions, consistency is ensured. However, if the leader is correct but suspected to have crashed, the other processes can decide upon a different value than the leader. Isis resolves such inconsistencies by forcing the suspected leader to fail and re-incarnate itself as a new process, whereby it adopts the state of the other replica.

In general, conflicts are inevitable when a single-communication step algorithm is used in non-synchronous models, as implied by the lower bound proven in this paper. On the other hand, conflicts can be detected immediately when processes that at some point suspected each other re-establish the communication between them. Thus, conflicts can be reconciled quickly. The cost-effectiveness of optimism depends on the rate of conflicts. Recall the observation that the role of the extra communication step in practical models is to avoid potential conflicts with processes that are incorrectly suspected to have failed. This suggests that the frequency of conflicts with the optimistic approach depends on the frequency of false suspicions. Thus, the rate of false suspicions in a given system can be a guideline for deciding when to use optimism, and when a conservative approach (like consensus) would be preferable.

7 Conclusions

Network environments can exhibit complex combinations of timing behaviors and failure patterns, so in the worst case, no useful work can be successfully completed. In practice however, usually communication is timely and failures are rare. Therefore, to be useful in practice, algorithms should be able to tolerate some asynchrony and failures, but should also perform well under the more common benign circumstances.

This paper was motivated by the above observation. We considered consensus algorithms that can tolerate bounded asynchrony and failures, and we studied such algorithms' performance under benign circumstances. We defined the notion of well-behaved executions, capturing such benign circumstances. We studied the number of communication steps performed by consensus algorithms in well-behaved executions. We showed a tight lower bound of two communication steps for systems in which at least two processes can crash. In contrast, algorithms that do not tolerate any asynchrony can decide in one communication step in all failure-free executions [LF82, Lyn96].

We have shown that the additional communication step is required due to the fact that, in non-synchronous models, correct processes may be mistaken for faulty ones. That is, the additional communication step is needed in order to avoid disagreement with processes that are incorrectly suspected to have failed. This suggests that when using an optimistic approach instead of consensus, one can use a single communication step algorithm with which inconsistencies will arise only in case of false suspicions, as explained in Section 6.

An important goal of this paper was to provide intuition for the lower and upper bounds. We illustrated the upper bound by discussing known algorithms that achieve the lower bound. We aimed to give formal lower bound proofs that would be as simple and easy-to-follow as possible.

In order to derive the two communication step lower bound for non-synchronous models, we first looked at the synchronous crash failure model. We showed that in this model, uniform consensus algorithms need to make $f+2$ communication steps in some executions with f failures before all processes can decide, if the number of processes that can crash is at least $f+2$. Although this bound was previously shown by Charron-Bost and Schiper [CBS00], their lower bound proof did not cover the case that $f=0$ in which we are most interested here. In this paper, we gave, for the first time, a proof by forward induction for this lower bound, which is also the first proof covering the case $f=0$. Furthermore, we gave a unified proof showing this lower bound along with two known results: a lower bounds for consensus, and an impossibility results for consensus in the mobile failure model. Thus, we illustrated the fact that different lower bounds stem from similar principles, and also illustrated a general technique for obtaining simple and elegant lower bound and impossibility proofs [MR98].

References

[ACM96] ACM. *Communications of the ACM 39(4), special issue on Group Communications Systems*, April 1996.

[ACT99] M. K. Aguilera, W. Chen, and S. Toueg. Using the Heartbeat failure detector for quiescent reliable communication and consensus in partitionable networks. *Theoretical Computer Science*, 220(1):3–30, June 1999.

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

[BHG87] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems.* Addison-Wesley, Reading, MA, 1987.

[BJBO98] Z. Bar-Joseph and M. Ben-Or. A tight lower bound for randomized synchronous consensus. In *17th ACM Symposium on Principles of Distributed Computing (PODC)*, pages 193–199, 1998.

[BSS91] K. Birman, A. Schiper, and P. Stephenson. Lightweight causal and atomic group multicast. *ACM Transactions on Computer Systems*, 9(3):272–314, 1991.

[CBS00] B. Charron-Bost and A. Schiper. Uniform consensus is harder than consensus (extended abstract). Technical Report DSC/2000/028, Swiss Federal Institute of Technology, Lausanne, Switzerland, May 2000.

[CF99] F. Cristian and C. Fetzer. The timed asynchronous distributed system model. *IEEE Transactions on Parallel and Distributed Systems*, pages 642–657, June 1999.

[CHT92] T. D. Chandra, V. Hadzilacos, and S. Toueg. The weakest failure detector for solving consensus. In *ACM Symposium on Principles of Distributed Computing (PODC)*, pages 147–158, 1992.

[CKV] G. V. Chockler, I. Keidar, and R. Vitenberg. Group Communication Specifications: A Comprehensive Study. *ACM Computing Surveys.* To appear. Previous version: MIT Technical Report MIT-LCS-TR-790, September 1999.

[CM84] J. Chang and N. Maxemchunk. Reliable broadcast protocols. *ACM Transactions on Computer Systems*, 2(3), 1984.

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

[DFKM96] D. Dolev, R. Friedman, I. Keidar, and D. Malki. Failure Detectors in Omission Failure Environments. TR 96-13, Institute of Computer Science, Hebrew University, Jerusalem, Israel, September 1996. Also Technical Report 96-1608, Department of Computer Science, Cornell University.

[DLS88] C. Dwork, N. 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. *Inform. Comput.*, 88(2):156–186, October 1990.

[DPLL97] R. De Prisco, B. Lampson, and N. Lynch. Revisiting the Paxos algorithm. In Marios Mavronicolas and Philippas Tsigas, editors, *11th International Workshop on Distributed Algorithms (WDAG)*, pages 111–125, Saarbrucken, Germany, September 1997. Springer Verlag. LNCS 1320.

[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] C. Dwork and D. Skeen. The inherent cost of nonblocking atomic commitment. In *ACM Symposium on Principles of Distributed Computing (PODC)*, pages 1–11, 1983.

[FLP85] M. Fischer, N. Lynch, and M. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32:374–382, April 1985.

[Gra78] J. N. Gray. Notes on database operating systems. In *Operating Systems: An Advanced Course, Lecture Notes in Computer Science*, volume 60, pages 393–481. Springer Verlag, Berlin, 1978.

[Gue95] R. Guerraoui. Revisiting the relationship between non-blocking atomic commitment and consensus. In Jean-Michel Hélary and Michel Raynal, editors, *9th International Workshop on Distributed Algorithms (WDAG)*, pages 87–100. Springer Verlag, September 1995. LNCS 972.

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

[HRT98] M. Herlihy, S. Rajsbaum, and M. R. Tuttle. Unifying synchronous and asynchronous message-passing models. In *17th ACM Symposium on Principles of Distributed Computing (PODC)*, pages 133–142. ACM, June 1998.

[HRT01] M. Herlihy, S. Rajsbaum, and M. R. Tuttle. A new synchronous lower bound for set agreement. Submitted for publication, April 2001.

[KD96] I. Keidar and D. Dolev. Efficient message ordering in dynamic networks. In *15th ACM Symposium on Principles of Distributed Computing (PODC)*, pages 68–76, May 1996.

[KD98] I. Keidar and D. Dolev. Increasing the resilience of distributed and replicated database systems. *Journal of Computer and System Sciences special issue with selected papers from ACM SIGACT-SIGMOD Symposium on Principles of Database Systems (PODS) 1995*, 57(3):309–324, December 1998.

[Kön36] D. König. *Theorie der endlichen und unendlichen graphen.* Liepzig, 1936. Reprinted by Chelsea, 1950.

[KT91] M. F. Kaashoek and A. S. Tanenbaum. Group communication in the Amoeba distributed operating system. In *11th International Conference on Distributed Computing Systems (ICDCS)*, pages 882–891, May 1991.

[Lam96] B. Lampson. How to build a highly available system using consensus. In Babaoğlu and Marzullo, editors, *Distributed Algorithms*, LNCS 1151. Springer-Verlag, 1996.

[Lam98] L. Lamport. The part-time parliament. *ACM Transactions on Computer Systems*, 16(2):133–169, May 1998. Also Research Report 49, Digital Equipment Corporation Systems Research Center, Palo Alto, CA, September 1989.

[Lam00] L. Lamport. Lower bounds on consensus. Unpublished manuscript, March 2000.

[Lam78] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 78.

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

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

[MR98] Y. Moses and S. Rajsbaum. The unified structure of consensus: a layered analysis approach. In *17th ACM Symposium on Principles of Distributed Computing (PODC)*, pages 123–132. ACM, June 1998. submitted for journal publication.

[MR99] A. Mostefaoui and M. Raynal. Solving consensus using Chandra-Toueg's unreliable failure detectors: a general approach. In *13th International Symposium on DIStributed Computing (DISC)*, Bratislava, Slovak Republic, 1999.

[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 Symposium on Management of Data*, pages 133–142, 1981.

[SW89] N. Santoro and P. Widmayer. Time is not a healer. In *6th Annual Symp. Theor. Aspects of Computer Science*, volume 349 of *LNCS*, pages 304–313, Paderborn, Germany, February 1989. Springer Verlag.