Synchronous System and Perfect Failure Detector: solvability and efficiency issues

Bernadette Charron-Bost
École Polytechnique
91128 Palaiseau Cedex, France
charron@lix.polytechnique.fr

Rachid Guerraoui André Schiper
Dépt. de Systèmes de Communication
École Polytechnique Fédérale de Lausanne
1015 Lausanne EPFL, Switzerland
{rachid.guerraoui, andre.schiper}@epfl.ch

Abstract

We compare, in terms of solvability and efficiency, the synchronous model, noted $\mathcal{S}_S$, with the asynchronous model augmented with a perfect failure detector, noted $\mathcal{S}_P$. We first exhibit a problem that, although time-free, is solvable in $\mathcal{S}_S$ but not in $\mathcal{S}_P$. We then examine whether one of these two models allows more efficient solutions for designing fault-tolerant applications. In particular, we concentrate on the uniform consensus problem which is solvable in both models, and we design a uniform consensus algorithm for the $\mathcal{S}_S$ model that is more efficient than any algorithm solving uniform consensus in $\mathcal{S}_P$ with respect to some significant time complexity measure. From a practical viewpoint, the synchronous model thus seems better than the asynchronous model augmented with a perfect failure detector.

1 Introduction

The choice of a model is a fundamental issue in the design of a fault-tolerant distributed system. Since agreement protocols (e.g., atomic broadcast, atomic commit) are at the heart of such systems [3, 4, 14], one need to consider models that are strong enough to circumvent the impossibility result of [13]. For this concern, two main approaches have been proposed: the *timing-based approach* [11, 12, 10, 16, 9] and the *failure detector* approach [6, 5, 1, 2].

The first approach consists in providing processes with information about time: the resulting models are called *timing-based models*. For example, message delays and relative processes speeds are bounded, and these bounds are known in the "perfect" timing-based model, namely the *synchronous model*. In contrast, none of these bounds exist in the *asynchronous model*. Intermediate timing-based models between the synchronous and the asynchronous models are those in which timing information is partial or inexact.

The second approach, i.e., the failure detector approach, is based on the observation that the impossibility results in the asynchronous model stem from the inherent lack of reliable failure detection. Chandra and Toueg [6] propose to augment the asynchronous model with an external failure detection mechanism, which may make mistakes. Instead of focusing on timing features, the models of [6] are defined according to *axiomatic properties* of failure detectors. Failure detectors are classified in a hierarchy according to the correctness of their suspicions. The strongest element of this hierarchy is called the *perfect failure detector*, and is denoted by P. Roughly speaking, the failure detector P suspects a process (to have crashed) iff that process has indeed crashed.

Each approach has some advantages. On the one hand, timing-based models are more realistic than time-free models, since distributed systems do use timing information. For example, in most of real systems processes have access to almost-synchronized clocks and know approximate bounds on process step time or message delivery time. On the other hand, the abstract failure detector approach is particularly powerful for investigating the solvability of fault-tolerant problems. As an example, Chandra, Hadzilacos, and Toueg [5] determined the minimal amount of information about failures that processes require to achieve consensus. This information is simply expressed as axiomatic properties of a failure detector. In contrast, there is no analogous result for timing-based models: the minimal *amount of synchrony* required to solve consensus is still an open problem.

The motivation of our work is to explore the similarities and differences between the timing-based approach and the failure detector approach. Some relations between the system models considered in these two approaches have already been discussed in [6]. In

Appeared in Proc. Intl Conf. on Dependable System and Networks (DSN'00).

particular, it is shown that a perfect failure detector can be implemented in the synchronous model using time-outs. In the system models of [12], time-out mechanisms can also be used to implement an *eventual perfect failure detector* – one of the eight failure detectors in Chandra and Toueg's hierarchy [6]. More generally, timing assumptions can be used to implement some failure detectors, and so are translated into axiomatic properties of failure detectors. However, some features of timing-based models may be lost in this failure detector translation. Comparing the two approaches consists in the determination of the properties of the timing-based models that are not preserved by the translation.

Rather than addressing this general issue, the paper restricts the comparison to the strongest cases. More precisely, we compare the strongest timing-based model, namely the synchronous model, with the model obtained by augmenting the asynchronous model with the strongest failure detector, namely P. These two models are denoted by $\mathcal{S}_S$ and $\mathcal{S}_P$, respectively. We compare $\mathcal{S}_S$ and $\mathcal{S}_P$ assuming process crash failures and reliable links, and we address the following two questions:

1. Is the class of problems solvable in $\mathcal{S}_S$ the same as the class of problems solvable in $\mathcal{S}_P$?
2. For problems that are solvable both in $\mathcal{S}_S$ and $\mathcal{S}_P$, does one of these models allow for more efficient solutions?

We first consider question (1), about problem solvability. Since $\mathcal{S}_S$ can implement $\mathcal{S}_P$, any problem that can be solved in $\mathcal{S}_P$ can also be solved in $\mathcal{S}_S$. Conversely, there trivially exist problems that are solvable in $\mathcal{S}_S$ but not in $\mathcal{S}_P$: problems whose specifications contain timing conditions (such as the scheduled deadline of some events) might be solvable in $\mathcal{S}_S$ but cannot be solved in $\mathcal{S}_P$. This observation leads us to define a notion of "time-free" problems which captures problems whose specifications do not involve absolute and relative timing conditions. We show that even when considering this class of problems only, solvability in $\mathcal{S}_S$ does not enforce solvability in $\mathcal{S}_P$. We do so by exhibiting a time-free problem, called the *strong dependent decision* problem, that is solvable in in $\mathcal{S}_S$ but not in $\mathcal{S}_P$. The initial motivation for introducing this problem specification is theoretical. However, we show that the strong dependent decision problem is quite relevant in the context of atomic commit.

To address question (2) about efficiency, we consider the time complexity measure. For that, we introduce an adequate round-based computational model for $\mathcal{S}_P$, denoted $\mathcal{R}_{WS}$, which is comparable to the well-known synchronous round model $\mathcal{R}_S$ for $\mathcal{S}_S$ [16]. In both $\mathcal{R}_S$ and $\mathcal{R}_{WS}$, we measure the time complexity in terms of the number of rounds until the required outputs are produced. More precisely, we consider the *latency degree* introduced in [18] and two refinements of this time complexity measure. We concentrate on the fundamental uniform consensus problem that is well-known to be solvable in $\mathcal{R}_S$ (and so in $\mathcal{S}_S$) despite failures. We show that this result still holds in $\mathcal{R}_{WS}$ (and so in $\mathcal{S}_P$). We analyze various algorithms that solve uniform consensus for $\mathcal{R}_S$ and for $\mathcal{R}_{WS}$ in the presence of up to t faulty processes. When $t = 1$, we show that the $\mathcal{R}_S$ model enables to gain one round over $\mathcal{R}_{WS}$ in failure-free runs. This result is extended to any value of t ($t < n$) in a companion paper [7]. Therefore, uniform consensus can sometimes be reached in $\mathcal{R}_S$ sooner than in $\mathcal{R}_{WS}$. In conclusion, contrary to a fairly common idea the $\mathcal{S}_S$ synchronous model is strictly stronger than the asynchronous model augmented with a perfect failure detector $\mathcal{S}_P$, in terms of both solvability and efficiency.

The paper is organized as follows. Section 2 defines the $\mathcal{S}_S$ and $\mathcal{S}_P$ system models. Section 3 shows that $\mathcal{S}_S$ is strictly stronger than $\mathcal{S}_P$ in terms of problem solvability, even when considering only problems with time-free specifications. Section 4 compares the efficiency of uniform consensus algorithms in $\mathcal{S}_S$ and $\mathcal{S}_P$. Finally Section 5 discusses our results and summarizes our contribution.

2 System Models

In this section, we define the synchronous system model $\mathcal{S}_S$ (Sect. 2.4) and the system model $\mathcal{S}_P$ obtained from augmenting the asynchronous model with the perfect failure detector P (Sect. 2.6) in a unified framework. Our definitions are taken from [11, 6].

We consider distributed systems consisting of a set of n processes $\Pi = \{p_1, \cdots, p_n\}$. Processes communicate by exchanging messages. Communications are point to point. Every pair of processes is connected by a reliable channel. We assume the existence of a discrete global clock to which processes do not have access. The range of the clock's ticks is the set of natural numbers, and is denoted by $\mathcal{T}$.

2.1 Failures and Failure Patterns

Processes can fail by crashing. A failure pattern F is a function from $\mathcal{T}$ to 2^{Π}, where $F(t)$ denotes the set of processes that have crashed by time t. If $p_i \notin F(t)$, we say that p_i is *alive at time* t. Processes are assumed not to recover, i.e., $\forall t \in \mathcal{T} : F(t) \subseteq F(t+1)$. Process p_i is *faulty* if $p_i \in Faulty(F) = \cup_{t \in \mathcal{T}} F(t)$; otherwise, p_i is *correct* and $p_i \in Correct(F) = \Pi \setminus Faulty(F)$.

2.2 Algorithms

Each process p_i has a buffer, $buffer_i$, that represents the set of messages that have been sent to p_i but that are not yet received. An *algorithm* A is a collection of n deterministic automata, one for each process. The automaton which runs on p_i is denoted by A_i. A *configuration* C of A consists of:

- n process states $state_1(C), \cdots, state_n(C)$ of $A_1, \cdots, A_n$, respectively;
- n sets of messages $buffer_1(C), \cdots, buffer_n(C)$, representing the messages presently in $buffer_1, \cdots, buffer_n$.

Configuration C is an *initial configuration* if every state $state_i(C)$ is an initial state of A_i and $buffer_i(C)$ is empty. Computations proceed in *steps* of A. In each step, a unique process p_i atomically (1) receives a (possibly empty) set of messages, (2) changes its state, and (3) may send a message to a single process, depending on its state at the beginning of the step and on the set of messages received in the step. A *schedule* of A is an infinite sequence of A's steps.

System models are defined according to the way algorithms execute. In other words, a system model determines the set of runs that algorithms can produce in the model. In this way, we define the *asynchronous model*, as well as the $\mathcal{S}_S$ and $\mathcal{S}_P$ models that are both derived from the asynchronous model by restricting the set of possible runs.

2.3 The Asynchronous Model

A *run of algorithm* A in the *asynchronous model* is a tuple $<F, C_0, S, T>$, where F is a failure pattern, C_0 is an initial configuration of A, S is a schedule of A, and T is an infinite list of increasing time values of $\mathcal{T}$ (indicating when each step of S occurs), and which satisfies the following properties: (1) every correct process takes an infinite number of steps in S, (2) every process cannot take a step in S if it has crashed, and (3) every message sent to a correct process is eventually received in S.

An *asynchronous run* (or a *run*, for short) is a run of some algorithm in the asynchronous model.

2.4 The $\mathcal{S}_S$ Synchronous Model

Following [11], the synchronous model $\mathcal{S}_S$ is defined by two constants $\Phi \geq 1$ and $\Delta \geq 1$ such that any *run* $<F, C_0, S, T>$ *of algorithm* A *in* $\mathcal{S}_S$ satisfies the two following synchrony conditions:

- *Process synchrony:* for any finite subsequence S' of consecutive steps in S, if some process takes $\Phi + 1$ steps in S' then any process that is still alive at the end of S' has taken at least one step in S'.
- *Message synchrony:* for any pair of indices k, l with $l \geq k + \Delta$, if message m is sent to p_i during the k-th step and p_i takes the l-th step, then m is received by the end of the l-th step.

Notice that these two conditions are only in terms of steps: they both refer only to schedule S and not to T. In particular, they do not specify bounds on the real time required to execute a step nor to deliver a message. In contrast, [15] defines the synchronous model in terms of real time synchrony conditions, which imply that the Φ and Δ bounds of the process and message synchrony properties exist. With respect to non-real time problems, we prefer the above definition borrowed from [11], since it does not refer to some specific operational features of systems.

2.5 Models with Failure Detectors

A *failure detector history* H is a function from $\Pi \times \mathcal{T}$ to 2^{Π}, where $H(p_i, t)$ represents the set of processes that p_i suspects to have crashed at time t in the history H. A *failure detector* D is a function that maps each failure pattern F to a set of failure detection histories.

In a system equipped with a failure detector, algorithms are defined in the same way as above, except that each step taken by p_i contains an intermediate *failure detector query phase* during which p_i queries and receives a value from its failure detector module. The state and the message resulting from the send phase of a step depend additionally on the value returned by the failure detector. A run r is defined as in the asynchronous model, except that one has to specify a "compatible" (cf. [5]) failure detector history H_D of D, i.e., $r = <F, H_D, C_0, S, T>$.

2.6 The $\mathcal{S}_P$ Model with Perfect Failure Detector

Roughly speaking, the so-called *perfect failure detector* P detects process crashes without any mistakes: all failures are eventually detected, and processes are

Appeared in Proc. Intl Conf. on Dependable System and Networks (DSN'00).

never wrongly suspected to crash. The system model obtained by augmenting the asynchronous model with the perfect failure detector P is denoted by $\mathcal{S}_P$.

2.7 Problem Specifications

A *problem specification* (or a *problem*, for short) is defined as requirements on runs. Therefore, a problem may be modelled by a set Σ of runs.

We restrict our attention to specifications Σ that do not depend on failure detector histories, i.e., for any runs $r = <F, H, C_0, S, T>$ and $r' = <F, H', C_0, S, T>$, $r \in \Sigma$ implies $r' \in \Sigma$. In problem specifications, references to failure detector history are thus useless and will be omitted.

As we saw in the introduction, problems whose specifications are related to synchrony conditions lead to a trivial comparison between $\mathcal{S}_S$ and $\mathcal{S}_P$. We thereby restrict attention to problems whose specifications are insensitive to consistent schedule modifications [17]. Formally, let S_i denote the sequence of the steps taken by p_i in S. A problem Σ is *time-free* if for any runs $r = <F, C_0, S, T>$ and $r' = <F, C_0, S', T'>$ such that $S_i = S'_i$ for any process p_i, $r \in \Sigma$ implies $r' \in \Sigma$. In particular, this holds when $S = S'$ but step time lists T and T' are different. From now on, we only consider such problem specifications. The term "problems" will thus refer only to time-free problems.

For any system model $\mathcal{S}$ and any distributed algorithm A, let $Run(A, \mathcal{S}, t)$ denote the set of runs that A can produce in $\mathcal{S}$, in which at most t processes crash. Algorithm A *tolerates* t *crashes and solves problem* Σ *in* $\mathcal{S}$ if $Run(A, \mathcal{S}, t) \subseteq \Sigma$.

3 $\mathcal{S}_S$ is strictly stronger than $\mathcal{S}_P$

In the synchronous model, detecting failures perfectly is easy: a simple time-out mechanism with time-out periods that depend on the Δ and Φ bounds, one can implement a perfect failure detector. This implies that any problem that can be solved with a perfect failure detector is also solvable in a synchronous system.

A widespread argument explaining why some problems (such as consensus, atomic broadcast, ...) cannot be solved in asynchronous systems is the impossibility to determine whether a process crashed or is very slow in such a system. In other words, with respect to problem solvability, asynchronous systems seem to differ from synchronous systems basically on the impossibility of achieving perfect failure detections. This leads to think that an asynchronous system becomes as "suitable" as a synchronous system for solving problems as soon as it is equipped with a perfect failure detector. So it may appear that any problem solvable in a synchronous system is still solvable in an asynchronous system where one can detect failures perfectly.

We show below that this intuition is incorrect by exhibiting a problem which can be solved in $\mathcal{S}_S$ but not in $\mathcal{S}_P$. For this problem, we consider two processes p_i and p_j. Process p_i starts with an input value in $\{0, 1\}$. The goal is for process p_j to eventually output a decision in $\{0, 1\}$. There are three conditions imposed on the decision made by p_j:

- *Integrity:* Process p_j decides at most once.
- *Validity:* If p_i has not initially crashed, the only possible decision value for p_j is p_i's initial value.
- *Termination:* If p_j is correct, then p_j eventually decides.

This problem will be referred to as the *Strongly Dependent Decision* problem (or simply, *SDD*). Note that SDD is clearly a time-free problem.

The motivation for introducing the SDD problem is not just theoretical. This problem turns out to be quite relevant in the context of atomic commit. Indeed, solving SDD provides more efficient atomic commit algorithms, i.e., algorithms that lead to the commit decision more often, while preserving the validity property: When all processes propose to commit and there is no initially dead process, processes may safely decide to commit despite failures if the SDD problem is solvable.

In $\mathcal{S}_S$, the SDD problem has a very simple algorithm: p_i sends its initial value to p_j during its first step. Process p_j executes $\Phi + 1 + \Delta$ (possibly empty) steps. If p_j receives a message from p_i during this period, p_j decides the value sent by p_i; otherwise, it decides 0.

Since the Φ and Δ bounds do not exist in the $\mathcal{S}_P$ model, this algorithm does not work in $\mathcal{S}_P$. More generally, we show that SDD cannot be solved in $\mathcal{S}_P$.

Theorem 3.1 *There is no algorithm that solves SDD in the* $\mathcal{S}_P$ *model and tolerates one crash.*

Proof: Suppose, for contradiction, that there is such an algorithm A. Let r_0 be a run of A where p_i's initial value is 0, p_i crashes from the beginning, and p_j suspects p_i as soon as p_i crashes. By the termination requirement, p_j eventually decides in r_0. Since messages may experience arbitrary (but finite) delays in the $\mathcal{S}_P$ model, we construct a run r'_0 of A whose first step is taken by p_i, p_i crashes just after taking one step, and except its first step, r'_0 is identical to r_0 up to p_j's decision. By validity, p_j decides 0 in r'_0. From p_j's viewpoint, r'_0 is indistinguishable from r_0 until p_j decides. So p_j also decides 0 in r_0. Let r_1 and r'_1 be the runs of A that are the same as r_0 and r'_0, respectively, except p_i's initial value is 1. By the same reasoning as above, process p_j decides 1 in both r_1 and r'_1. Clearly, r_1 is indistinguishable from r_0 with respect to p_j, and so p_j decides the same value in both r_0 and r_1 — a contradiction. $\Box$

From Theorem 3.1, it follows that there exist atomic commit algorithms for synchronous systems that are more efficient (i.e., that lead to the commit decision more often) than any atomic commit algorithm for asynchronous systems equipped with a perfect failure detector.

This result points out a fundamental difference between $\mathcal{S}_S$ and $\mathcal{S}_P$: *the delay for detecting a failure is bounded in the $\mathcal{S}_S$ model whereas it is finite but unbounded in the $\mathcal{S}_P$ model.* More precisely, if p_i is supposed to send a message m to p_j while p_j is taking its k-th step, if p_j is aware of that, and if p_i crashes and fails in sending m, then p_j can detect p_i's crash when taking its $(k+\Phi+1+\Delta)$-th step in $\mathcal{S}_S$. Such a bound does not exist in $\mathcal{S}_P$. Basically, the SDD problem has been specified to capture this difference.

4 Round-Based Computational Models

To address the efficiency issue, we consider the time complexity measure. For that, we introduce two round-based computational models that can be easily emulated from $\mathcal{S}_S$ and $\mathcal{S}_P$. More precisely, we show that computations in $\mathcal{S}_S$ and $\mathcal{S}_P$ can be organized in *synchronous rounds* and in *weakly synchronous rounds*, respectively. This defines two round-based computational models $\mathcal{R}_S$ and $\mathcal{R}_{WS}$, which provide a uniform framework for describing and assessing algorithms. In both $\mathcal{R}_S$ and $\mathcal{R}_{WS}$, the time complexity is measured in terms of the number of rounds until all the required outputs are produced.

4.1 The $\mathcal{R}_S$ Model.

As in Section 2, each process has a buffer denoted $buffer_i$. An algorithm A of the $\mathcal{R}_S$ model consists for each process $p_i \in \Pi$ of the following components [16]: a set of states denoted by $states_i$, an initial state $init_i$, a message-generation function $msgs_i$ mapping $states_i \times \Pi$ to a unique (possibly *null*) message, and a state transition function $trans_i$ mapping $state_i$ and vectors (indexed by Π) of message to $states_i$. In any execution of A, each process p_i repeatedly performs the following two actions in lock-step mode:

1. Apply $msgs_i$ to the current state to generate the messages to be sent to each process. Put these messages in the appropriate buffers.
2. Apply $trans_i$ to the current state and the messages present in $buffer_i$ to obtain the new state. Remove all messages from the $buffer_i$.

The combination of these two actions is called a *round* of A.

It turns out that each round in $\mathcal{R}_S$ satisfies the *round synchrony property*: *If p_i is alive at the end of round r and does not receive a message from p_j at round r, then p_j fails before sending a message to p_i at round r.*

The $\mathcal{R}_S$ computational model can be easily emulated by $\mathcal{S}_S$. The basic idea of the emulation is the following: In each round r, every process p_i executes $n+k$ steps of the $\mathcal{S}_S$ model. The first n steps are used to send *real* messages whereas in the k last steps, p_i sends *null* messages to make sure that, before moving to round $r+1$, p_i receives all messages sent to it by other processes in round r (k is a function of n, Δ, Φ and r).

4.2 The $\mathcal{R}_{WS}$ Model.

We define now a comparable round-based computational model $\mathcal{R}_{WS}$ for $\mathcal{S}_P$. Likewise in $\mathcal{R}_S$, the code of each process p_i is entirely determined by the state set $states_i$, the message-generation function $msgs_i$, and the state transition function $trans_i$. The difference between $\mathcal{R}_S$ and $\mathcal{R}_{WS}$ lies in the fact that the state-transition function $trans_i$ is now applied to a *subset* of messages present in $buffer_i$.

The round synchrony property is no longer guaranteed, but we assume that rounds of $\mathcal{R}_{WS}$ satisfy the *weak round synchrony property*: *If p_i is alive at the end of round r and does not receive a message from p_j at round r, then p_j crashes by the end of round $r+1$.* Contrary to $\mathcal{R}_S$, it might be the case that a (faulty) process sends a message to p_i at round r but p_i does not receive this message although p_i is alive at the end of round r. Such a message is called a *pending* message.

The $\mathcal{R}_{WS}$ model can be emulated from $\mathcal{S}_P$. The reception of messages in round r is done as follows in $\mathcal{S}_P$: Process p_i keeps executing (possibly null) steps of model $\mathcal{S}_P$ until, for every process p_j, either p_i receives a message from p_j or p_i suspects p_j. In this emulation,

Appeared in Proc. Intl Conf. on Dependable System and Networks (DSN'00).

it may be possible that p_j sends a pending message at round r and does not crash at round r. However, the following lemma states that p_j crashes by the end of round $r+1$.

Lemma 4.1 *The above emulation guarantees the weak round synchrony property.*

Proof: Suppose for contradiction that some process p_j that is alive at the beginning of round $r+2$ sends a pending message m to p_i at round r. Let t' be the time at which p_j starts round $r+2$ and t'' be the time at which p_i starts round $r+1$. From the emulation and the fact that m is pending, it follows that p_i suspects p_j at t'', i.e., $p_j \in H(p_i, t'')$. Since the failure detector is perfect, this implies that p_j has crashed by time t''. Therefore, we have $t' < t''$.

On the other hand, p_j is allowed to start round $r+2$ only if it has received a message from p_i at round $r+1$ or it suspects p_i. But p_i is alive at time t'' and thus at time t'. Since the failure detector is perfect, p_j cannot suspect p_i at time t'. It follows that p_j receives a message from p_i at round $r+1$. Let t be the time at which p_j receives this message; since a message must be sent before it is received, we have $t'' < t < t'$. This is a contradiction. $\Box_{Lemma\ 4.1}$

4.3 Runs in $\mathcal{R}_S$ and $\mathcal{R}_{WS}$

Let A be any algorithm of the $\mathcal{R}_{WS}$ model (and so of the $\mathcal{R}_S$ model). A *run of A in $\mathcal{R}_S$* is an infinite sequence of A's rounds. A *partial run of A* is a finite prefix of a run of A. Definitions of Section 2 concerning problem specifications are easily adapted to $\mathcal{R}_S$ and $\mathcal{R}_{WS}$. Note that since synchronous rounds are weakly synchronous, if an algorithm solves a problem Σ in $\mathcal{R}_{WS}$, then it also solves Σ in $\mathcal{R}_S$. The number of rounds required to solve a problem in $\mathcal{R}_S$ is thus not greater than in $\mathcal{R}_{WS}$.

5 $\mathcal{R}_S$ is More Efficient than $\mathcal{R}_{WS}$ with Respect to Uniform Consensus

We now concentrate on the uniform consensus problem. In order to measure time complexity of uniform consensus algorithms, we consider the *latency degree* introduced in [18] and various refinements of it. All these time complexity measures are in terms of the number of rounds until all the correct processes decide, and so are quite significant in practice. We prove that with regard to the most discriminating time complexity measures among the ones we consider, $\mathcal{R}_S$ allows to achieve uniform consensus faster than $\mathcal{R}_{WS}$ does.

5.1 The Uniform Consensus Specification

Let V be a fixed value set that is totally ordered. Here each process starts with an input value from V and must reach an irrevocable decision on one value of V. The *uniform consensus* specification is defined as the set of all runs that satisfy the following conditions:

- *Uniform validity:* If all processes start with the same initial value $v \in V$, then v is the only possible decision value.
- *Uniform agreement:* No two processes (whether correct or faulty) decide on different values.
- *Termination:* All correct processes eventually decide.

The uniform consensus problem for crash failures has a very simple algorithm in $\mathcal{R}_S$, called *FloodSet* [16]. Processes just propagate all the values in V that they have ever seen and use a simple decision rule at the end. More precisely, each process maintains a variable W containing a subset of V. Initially, process p_i's variable W contains only p_i's initial value. For each of $t+1$ rounds, each process broadcasts W, then adds all the elements of the received sets to W. After $t+1$ rounds, any still alive process decides on the minimum value of W. The complete code of *FloodSet* is given in Figure 1.[1] From the observation that if at most t processes may crash, then among $t+1$ rounds there must be some round at which no process fails, we easily deduce that *FloodSet* solves the uniform consensus problem in $\mathcal{R}_S$ if at most t processes may crash.

Because of pending messages, *FloodSet* allows disagreement in $\mathcal{R}_{WS}$. However, we can slightly modify this algorithm by forcing any process that does not receive a message from process p_i at some round r to ignore the message that may arrive from p_i at round $r+1$. The code of the resulting algorithm called *FloodSetWS* is given in Figure 2. In a companion paper [7], we prove that *FloodSetWS* actually solves the uniform consensus problem in $\mathcal{R}_{WS}$.

Uniform consensus differs from the consensus problem in the uniform agreement condition: it prevents two processes to disagree even if one of the two processes crash some (maybe long) time after deciding. Thereby, uniform consensus, that is achievable in the context of crash failures of $\mathcal{R}_S$ and $\mathcal{R}_{WS}$, precludes undesirable runs and thus specifies the agreement problem in a more satisfactory way than consensus does. For much of system models, Charron-Bost *et al.* [8]

[1] In the code of *FloodSet*, as well as in the codes that follow, null messages are not specified in the $msgs_i$'s.

states$_\mathbf{i}$
$rounds \in N$, initially 0
$W \subseteq V$, initially the singleton set consisting of p_i's initial value
$decision \in V \cup \{unknown\}$, initially $unknown$

msgs$_\mathbf{i}$
if $rounds \leq t$ then send W to all processes

trans$_\mathbf{i}$
$rounds := rounds + 1$
let X_j be the message from p_j, for each p_j from which a message arrives
$W := W \cup \bigcup_j X_j$
if $rounds = t + 1$ then $decision := \min(W)$

Figure 1: FloodSet Algorithm

states$_\mathbf{i}$
$rounds \in N$, initially 0
$W \subseteq V$, initially the singleton set consisting of p_i's initial value
$halt \subseteq \Pi$, initially $\emptyset$
$decision \in V \cup \{unknown\}$, initially $unknown$

msgs$_\mathbf{i}$
if $rounds \leq t$ then send W to all processes

trans$_\mathbf{i}$
$rounds := rounds + 1$
let X_j be the message from p_j, for each p_j from which a message arrives
$W := W \cup \bigcup_{p_j \notin halt} X_j$
for all p_j from which no message has arrived do
$halt := halt \cup \{p_j\}$
if $rounds = t + 1$ then $decision := \min(W)$

Figure 2: FloodSetWS Algorithm

show that in fact, considering the consensus specification instead of its uniform version has no bad effect: for such systems, any algorithm that solves consensus also solves uniform consensus. However, this result holds neither in $\mathcal{R}_S$ nor in $\mathcal{R}_{WS}$. Therefore, the uniform consensus problem actually differs from consensus in the system models considered here.

5.2 Latency Degrees of Uniform Consensus Algorithms

Let A be any uniform consensus algorithm of the $\mathcal{R}_{WS}$ model (and so of the $\mathcal{R}_S$ model). For any run r of A, the *latency degree of* r introduced in [18] corresponds to the number of rounds until all the correct processes decide in run r. We denote the latency degree of r by $|r|$.

Of course, any uniform consensus algorithm generates many different runs, depending for example on initial configurations or on failure histories. Given a system $\mathcal{S}$ and an integer t such that $t < n$, Schiper [18] defines the *latency degree of* A as the minimal run latency degree over all possible runs that can be produced by A when running in $\mathcal{S}$ and in the presence of at most t process failures. Namely,

$$lat(A) = \min\{|r| : r \in Run(A, \mathcal{S}, t)\}.$$

Basically, this time complexity measure captures the algorithm abilities of taking advantage of some run parameters (e.g., initial configuration, number of failures) to achieve consensus quickly.

Because of the validity condition, any process that receives n messages with the same value v at round 1 could safely decide v at the end of round 1. Based on this remark, we can slightly modify *FloodSet* and *FloodSetWS* to obtain two new uniform consensus algorithms for the $\mathcal{R}_S$ and $\mathcal{R}_{WS}$ models, respectively. Each process operates as in *FloodSet* and *FloodSetWS*, except that it decides v at round 1 if it receives n messages with value v at round 1. Formally, $trans_i$ functions are modified by substituting the following decision rule:

if $rounds = 1$ and
a message has arrived from every process then
if $|W| = 1$ then $decision := v$, where $W = \{v\}$
else if $rounds = t + 1$ then $decision := \min(W)$

for

if $rounds = t + 1$ then $decision := \min(W)$

We denote the resulting algorithms by *C_OptFloodSet* and $C\_OptFloodSetWS$, respectively. Clearly, we have

$$lat(C\_OptFloodSet) = lat(C\_OptFloodSetWS) = 1$$

This is because *C_OptFloodSet* and $C\_OptFloodSetWS$ both take advantage of the fact that by validity, the decision value is determined from the beginning if all the initial values are the same. A sharper comparison of $\mathcal{R}_S$ and $\mathcal{R}_{WS}$ thus requires the definition of a time complexity measure that does not focus on some runs with "too specific" initial configurations. This leads discriminating runs according

to their initial configurations. Namely, for any initial configuration C we define
$lat(A, C) =$
$\min\{|r| : r \in Run(A, \mathcal{S}, t)$ and r starting from $C\}$, and

$$Lat(A) = \max\{lat(A, C) : C \in \mathcal{C}\}$$

where $\mathcal{C}$ denotes the set of A's initial configurations.

In turn, algorithms can exploit some specific failure histories to decide quickly. For example, if process p_i receives $n - t$ messages at round 1 then p_i knows the exact set of faulty processes. In this case, p_i can decide at the end of round 1 provided it notifies its decision at round 2 and this decision is then forced on other processes. According to this idea, we can modify *FloodSet* and *FloodSetWS* to design uniform consensus algorithms in which a decision is taken at the end of the first round if t processes initially crash, regardless initial configurations. We denote the resulting new versions of *FloodSet* and *FloodSetWS* by *F_OptFloodSet* and *F_OptFloodSetWS*, respectively. The formal code of *F_OptFloodSet* is given in Figure 3.

Theorem 5.1 F_OptFloodSet *and* F_OptFloodSetWS *solve the uniform consensus problem in* $\mathcal{R}_S$ *and* $\mathcal{R}_{WS}$*, respectively.*

Proof: We only show the correctness of *F_OptFloodSet*. The proof for *F_OptFloodSetWS* is similar. We use the notation $W_i(r)$ to denote the value of variable W at process p_i just after r rounds.

Termination is obvious, by the decision rules. For uniform validity, suppose that all the initial values are equal to v. Then v is the only value that ever gets sent anywhere. Each set W_i is non empty, because it contains p_i's initial value. Therefore, each $W_i(t+1)$ must be exactly equal to $\{v\}$, so the decision rules say that v is the only possible decision.

For uniform agreement, we consider two cases:

1. No process receives exactly $n-t$ messages during the first round. That is, all processes run the *FloodSet* algorithm. By uniform agreement of *FloodSet*, it follows that all processes make the same decision.

2. The set Π' of processes that receive exactly $n-t$ messages during the first round is not empty. By the round synchrony property which holds in $\mathcal{R}_S$, the set of processes from which every process in Π' receives a message at round 1 is exactly the set of correct processes. It follows that for any processes p_i and p_j in Π', we have

$$W_i(1) = W_j(1).$$

Then all the processes in Π' make the same decision, say v. At round 2, processes of Π' force decision v. Therefore, all the correct processes that have not decided at round 1, also decide v at the end of round 2.

□

---

states$_\mathbf{i}$
  $rounds \in N$, initially 0
  $W \subseteq V$, initially the singleton set consisting of p_i's initial value
  $decided \in \{false, true\}$, initially $false$
  $decision \in V \cup \{unknown\}$, initially $unknown$

msgs$_\mathbf{i}$
  if $rounds \leq t$ then
    if $decided = false$ then send W to all processes
    else send $(D, decision)$ to all processes

trans$_\mathbf{i}$
  $rounds := rounds + 1$
  let X_j be the message from p_j, for each p_j from which a message arrives
  if $rounds = 1$ and $n - t$ messages have arrived then
    $W := W \cup \bigcup_j X_j$
    $decided := true$
    $decision := \min(W)$
    $decided := true$
  else if at least one X_j equals to (D, v) then
      $decision := v$
      $decided := true$
    else $W := W \cup \bigcup_j X_j$
  if $rounds = t + 1$ and $decided = false$ then
    $decision := \min(W)$
    $decided := true$

Figure 3: *F_OptFloodSet* Algorithm

Considering the runs of *F_OptFloodSet* and *F_OptFloodSetWS* in which t processes initially crash, we have:

$$Lat(F\_OptFloodSet) = Lat(F\_OptFloodSetWS) = 1.$$

Interestingly, this contradicts a widespread idea that minimal latency degree is typically obtained with failure free runs.

The above examples yield a new refinement of the latency degree: we now discriminate runs according to the number of failures which occur. Formally, for any integer f such that $0 \leq f \leq t$ we define

$$Lat(A, f) = \max\{|r| : r \in Run(A, \mathcal{S}, f)\},$$

and

$$\Lambda(A) = \min_{0 \le f \le t} Lat(A, f).$$

By definition of $Lat(A, f)$, we have $Lat(A, f) \le Lat(A, f+1)$, and so $\Lambda(A) = Lat(A, 0)$. In other words, $\Lambda(A)$ is the maximal latency degree over all failure free runs.

5.3 $\mathcal{R}_S$ May Be More Efficient Than $\mathcal{R}_{WS}$

We now compare the $\mathcal{R}_S$ and $\mathcal{R}_{WS}$ models in terms of the Λ latency degree. First we restrict to the case $t = 1$, and we present a simple algorithm A_1 for uniform consensus in $\mathcal{R}_S$ that tolerates at most one crash. The A_1 algorithm is based on the following ideas: Process p_1 broadcast its initial value v_1 during the first round. Upon receiving v_1, process p_i decides v_1 at the end of round 1. Subsequently, p_i reports its decision at round 2. If p_2 does not receive a message from p_1 in the first round (because p_1 has crashed), it broadcasts its initial value v_2 at round 2. Since at most one failure may occur, every correct process has received v_1 or v_2, or both by the end of the second round. If it receives v_1, it decides v_1; otherwise it decides v_2. All the runs of A_1 have thus two rounds. The code of A_1 is given in Figure 4.

Theorem 5.2 *The A_1 algorithm tolerates one crash and solves the uniform consensus problem in the $\mathcal{R}_S$ model.*

Proof: Termination is obvious, by the decision rules. For uniform validity, suppose that all the initial values are equal to some $v_0 \in V$. Then v_0 is the only possible value of any w variable. So the decision rules say that v_0 is the only possible decision.

For uniform agreement, there are two cases to consider:

1. Process p_1 decides at round 1. Then p_1 decides on its initial value v_1. Before deciding, p_1 succeeded in sending v_1 to all processes. Therefore, any correct process receives v_1 and then decides v_1 at the end of the first round.

2. Process p_1 does not decide at round 1. In this case, p_1 crashes during the first round. Since $t = 1$, all the other processes are correct. We consider two subcases:

   (a) Process p_1 succeeds in sending at least one message to some (correct) process p_i before crashing. From the algorithm, we deduce that p_i decides v_1 at round 1 and broadcasts (p_1, v_1) at round 2. All correct processes then receive at least one message of the form (p_1, v_1) in the second round, and so decide v_1.

   (b) No correct process receives a message from p_1 at round 1, and thus no process decides at the end of this round. At round 2, process p_2 broadcasts its initial value v_2 and all the other messages that are sent at round 2 are null. All processes, except process p_1 (that never decides), decide v_2.

□

states$_i$
  $rounds \in N$, initially 0 ;
  $w \in V$, initially p_i's initial value ;
  $decided \in \{false, true\}$, initially $false$;
  $decision \in V \cup \{unknown\}$, initially $unknown$

msgs$_i$
  if $rounds = 1$ and $i = 1$ then send w to all
  if $rounds = 2$ then
    if $decided = true$ then send (p_1, w) to all
    else if $i = 2$ then send w to all processes

trans$_i$
  $rounds := rounds + 1$
  let x_j be the message from p_j, for each p_j from which
    a message arrives
  if $rounds = 1$ and a message has arrived from p_1 then
    $w := x_1$
    $decision := x_1$
    $decided := true$
  if $round = 2$ then
    if at least one message x_j is equal to (p_1, w_j) then
      $decision := w_j$
      $decided := true$
    else   {a message $x_2 = w_2$ arrives from p_2}
      $decision := x_2$
      $decided := true$

Figure 4: The A_1 Algorithm

In every failure free run of A_1, each process decides at the end of the first round. We thus have $Lat(A_1, 0) = 1$, and so $\Lambda(A_1) = 1$.

Basically, the uniform agreement property is no more guaranteed by A_1 in the $\mathcal{R}_{WS}$ model. To see that, assume that at round 1, p_1 succeeds in broadcasting v_1, decides, and then crashes. In addition, suppose that all the messages sent by p_1 are pending. In this scenario, p_1 decides v_1 whereas all the other processes decide v_2. Modifications such as the one used to transform *FloodSet* into *FloodSetWS* do not preclude such disagreement. In fact, a result in [7] shows that if $n \geq 3$, then there is no uniform consensus algorithm in the $\mathcal{R}_{WS}$ model such that all correct processes decide at round 1 of all failure free runs. In other words, for any uniform consensus algorithm A in $\mathcal{R}_{WS}$ that tolerates one crash we have $Lat(A, 0) \geq 2$, and so $\Lambda(A) \geq 2$. This shows that $\mathcal{R}_S$ allows to design more efficient solutions to the uniform consensus problem than $\mathcal{R}_{WS}$ does, in the presence of one crash. Thanks to the general result stated in [7], we extend this result: we show that this discrepancy between $\mathcal{R}_S$ and $\mathcal{R}_{WS}$ still exists with any number $t < n$ of possible crashes.

6 Discussion

With regard to both solvability and efficiency concerns, we have shown that the synchronous model is better than the asynchronous model equipped with a perfect failure detector. We have pointed out some significant properties of the synchronous model that are lost when translating the synchronous timing assumptions into the axiomatic properties of a perfect failure detector. This is a first step towards the general comparison between timing-based models and models with failure detectors. This wide question is indeed a fundamental issue on which the practical relevance of the failure detector approach relies. It thus seems worthy to extend these results to other classes of timing-based models and other classes of failure detectors.

References

[1] M. Aguilera, W. Chen, and S. Toueg. Failure detection and consensus in the crash-recovery model. In *Proceedings of the 12th International Symposium on Distributed Computing*, Lecture Notes on Computer Science, pages 231–245. Springer-Verlag, September 1998.

[2] 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.

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

[4] K. P. Birman and T. A. Joseph. Reliable communication in the presence of failures. *ACM Transactions on Computer Systems*, 5(1):47–76, February 1987.

[5] T. D. Chandra, V. Hadzilacos, and S. Toueg. The weakest failure detector for solving consensus. *Journal of the ACM*, 43(4):685–722, July 1996.

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

[7] B. Charron-Bost and A. Schiper. Uniform consensus is harder than consensus. Technical Report, EPFL, Dépt. de Systèmes de Communication, April 2000.

[8] B. Charron-Bost, S. Toueg, and A. Basu. Revisiting safety and liveness in the context of failures. Technical report, LIX, École Polytechnique, January 2000.

[9] F. Cristian and C. Fetzer. The timed asynchronous distributed system model. *IEEE Transactions on Parallel & Distributed Systems*, 10(6):642–657, June 1999.

[10] F. Cristian and F. Schmuck. Agreeing on processor-group membership in asynchronous distributed systems. Technical Report CSE95-428, UCSD, 1995.

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

[12] 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.

[13] 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.

[14] R. Guerraoui and A. Schiper. Consensus service: A modular approach for building agreement protocols in distributed systems. In *International Symposium on Fault-Tolerant Computing System*. IEEE, June 1996.

[15] V. Hadzilacos and S. Toueg. A modular approach to fault-tolerant broadcasts and related problems. Technical Report TR 94-1425, Cornell University, Dept. of Computer Science, May 1994.

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

[17] Gil Neiger and Sam Toueg. Simulating synchronized clocks and common knowledge in distributed systems. *Journal of the ACM*, 40(2):334–377, April 1993.

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

Appeared in Proc. Intl Conf. on Dependable System and Networks (DSN'00).