The Power and Limit of Adding Synchronization Messages for Synchronous Agreement*

Jiannong CAO† Michel RAYNAL⋆ Xianbing WANG‡ Weigang WU†

† Dpt of Computing, Hong Kong Polytechnic University, Kowloon, Hong Kong
⋆ IRISA, Université de Rennes 1, Campus de Beaulieu, 35042 Rennes Cedex, France
‡Singapore-Massachusetts Institute of Technology Alliance, Singapore 117576

Abstract

This paper investigates the use of additional synchronization messages in round-based message-passing synchronous systems. It first presents a synchronous computation model allowing a process to send such messages. The difference with respect to the traditional round-based synchronous model lies in the sending phase, where a process can first send a data message to each other process, and then, without a break, a synchronization message (their sendings can be pipelined). This model is suited to the class of local area networks where communication channels are reliable. (It is not for networks where unreliable communication requires message retransmission.)

To illustrate the model, the paper presents a uniform consensus algorithm suited to this model. This algorithm, based on the rotating coordinator paradigm, allows the processes to decide in at most $f+1$ rounds where f is the actual number of processes that crash in the corresponding run. (This improves the $f+2$ lower bound of the traditional synchronous model.) In addition to its efficiency, the algorithm enjoys another first class property, namely, design simplicity. The paper focuses also on lower bound results, and shows that any uniform consensus algorithm designed for the proposed model, requires at least $f+1$ rounds in the worst case. The proposed algorithm is consequently optimal. In that sense the paper has to be seen as an investigation of both the power and the limit of adding synchronization messages to synchronous systems built on top of local networks with reliable communication.

Keywords: *Distributed system, Early-stopping, Lower bound, Message-passing, Round-based computation, Two-step, Uniform consensus, Synchronous system.*

1 Introduction

The consensus problem Distributed computing is characterized by individual processors (also called nodes or processes) cooperating by exchanging data through a communication medium, in order to realize a common goal. Fault-tolerant distributed computing additionally considers that the processes can suffer failures.

So, the possibility for the processes to agree on a common decision (or a common value, or the execution of the same action, etc.) is key to fault-tolerant distributed computing, as if no agreement is ever needed, the application is actually a set of independent applications. This type of agreement is exactly what has been abstracted under the name *consensus* problem. Each process proposes a value, and is supposed to decide a value (termination property), such that there is a single decided value (agreement property)[1], and that value is a proposed value (validity property). From both a theoretical side and a practical side, this problem has received a considerable interest in the literature [3, 14].

On distributed computing models Finding right abstractions and realistic computing models is a main challenge of computer science. This has been pretty successful in sequential computing where the Turing machine model has become the standard model, and where programming languages have been designed to provide the programmers with convenient high level abstractions that allows them to concentrate on the problem they have to solve without being bothered by low level implementation details.

Two main models have been proposed for fault-tolerant distributed computing, namely, the *synchronous* model, and the *asynchronous* model [3, 11, 14, 16]. The synchronous model is characterized by time bounds on process speed and

*This work is supported by the PROCORE French/Hong Kong Joint Research Scheme under grant F-HK16/05T.

[1] This paper considers *uniform agreement*: no two processes can decide differently (be them correct or faulty). See Section 3.1.

Proceedings of the 2006 International Conference on Parallel Processing (ICPP'06)
0-7695-2636-5/06 $20.00 © 2006 IEEE

message delay, and these bounds are known by the processes (that can consequently use them in the algorithm they execute). The asynchronous model is characterized by the absence of time bounds on process speed and message delays. That is why synchronous systems are sometimes called *timely* systems, while asynchronous systems are called *time-free* systems.

The main point that makes these systems essentially different lies in the use of timeout values. Upper bounds for timeout values can be computed and safely used in synchronous systems. Differently, there is no upper bound for timeout values in an asynchronous system. This has a fundamental consequence, namely consensus can be solved in synchronous systems prone to failures while it cannot in asynchronous systems [11].

In order not to be bothered by the management of timeout vales, the behavior of synchronous systems is usually abstracted in the so-called *round-based* computation model. The processes progress by executing a sequence of rounds. In each round, each process first sends messages to the other processes, then receives messages from the other processes, and finally executes a local computation.

So, a natural way to determine the time cost of a synchronous algorithm is to count the number of rounds it requires. As far as the consensus problem is concerned we have the following lower bounds in the synchronous process crash failure model. Let t ($t < n$) be the maximum number of processes that are allowed to crash[2], and f ($0 \leq f \leq t$), be the actual number of processes that crash during a run. We have the following (where a round is a communication step):

- When considering only t: Any t-resilient consensus algorithm requires $t + 1$ rounds [2, 10].
- When considering t and f: Any t-resilient consensus algorithm requires $\min(t+1, f+2)$ rounds [7, 8, 13].

Content of the paper This paper proposes a new synchronous computing model that allows solving uniform consensus in $f + 1$ rounds. The proposed consensus protocol is coordinator-based and actually solves uniform consensus in one round, when the first coordinator does not crash during that round. Moreover, it is shown that $f + 1$ is a time lower bound for solving consensus in the new model.

There is no miracle. To obtain a uniform consensus algorithm that allows the processes to decide and stop in $f+1$ rounds in the worst case, the notion of round (communication step) used in the new model is slightly different from its classic definition. More precisely, the proposed model allows a process to do "more things" during a round. More specifically, it allows a process to send to other processes an additional "synchronization message", *after* it has sent them a "data message". This message carries no data, and is sent in the send phase of a round, which means that its content does not depend on the messages received during the current round. Differently, the content of a data message can depend on the data messages received during the previous rounds.

Moreover, when a process crashes while sending a data message to the other processes, it is possible that only an arbitrary subset of the destination processes actually receive the message (this is the usual assumption in the crash prone synchronous model [14]). Differently, a sending order is associated with synchronization messages. This means that if a process p sends such a message to q and r (in that order) and crashes during the sending, it is possible that both q and r receive the message, or only q receives the message, or none of them receives the message (it is not possible for the "second" process r to be the only one that receives the message). It is nevertheless important to observe that, despite this "additional synchronization power", the new model and the traditional synchronous model have the same power from a computational point of view: we show that any of them can simulate the other model (of course, simulating the new synchronous model on top of the traditional one has some cost in terms of additional rounds).

This synchronous computing model is envisaged for (and seems suited to) the class of synchronous systems built on top of local area networks with reliable communication. It is not designed for systems built on top of networks with unreliable communication, as then message retransmission would make unrealistic an efficient implementation of the two-sending step.

Another interesting feature of the proposed model lies in the fact it allows designing algorithms that are simple and have a simple proof. This is important as design simplicity is a first class property. As we will see in Section 4, from an algorithm design point of view, this model allows establishing a bridge from synchronous agreement protocols to asynchronous agreement protocols. More precisely, the sending of synchronization messages during a round, does correspond to the second communication step used in each round of asynchronous consensus protocols. This bridge relating synchronous and asynchronous agreement is new (as far consensus is concerned), and helps understand the basic machinery of rotating coordinator-based distributed consensus algorithms.

Related work As far as distributed computing models are concerned, several models lying between fully synchronous and fully asynchronous systems have been proposed (e.g.,

[2]This means that the algorithm is guaranteed to work correctly when there are no more than t process crashes. It can work correctly in some scenarios where there are more than t crashes, but there is no guarantee for these runs.

[9]). Models where the sending order is relevant are in [8] (to state a lower bound result for simultaneous Byzantine agreement in synchronous systems).

To prove lower bound results related to uniform consensus in synchronous systems, [13] considers runs where a process that crashes during a round does not send messages to an orderly subset of processes. This allows to know which processes have received a given message during a round, and to use this additional knowledge to obtain proofs simpler than previous proofs of the same results.

Synchronization messages are very popular in fault-free distributed computing. Maybe, one of the most known example is the distributed snapshot algorithm by Chandy and Lamport [6]. In this algorithm, when a process takes a local snapshot, it atomically sends a special message (called *marker*) on each of its outgoing channels. This synchronization message has two meanings: it informs the destination process that it has to take its local snapshot (if not already done), and it cleanly separates the data messages that have been sent on the same channel before it from the ones that are sent after it (this allows a process to safely determine the set of messages that were in transit on the corresponding incoming channel with respect to the global snapshot that is computed). A marker message defines a "synchronization point" that allows the destination process to learn consistent global information.

Another approach (and the only we are aware of) to circumvent the $f + 2$ lower bound associated with the traditional synchronous model has been recently proposed in [1] where is introduced the notion of *fast failure detector*. The idea is to enrich a traditional synchronous system with a device that informs each process about which processes have crashed. Each process p is provided with a read-only local variable $suspect(p)$ that is (1) safe: it contains only ids of processes that have crashed, and (2) live: if a process q crashes at time t, its id is added to $suspect(p)$ at the latest at time $t+d$. Let D be the duration of a round. The failure detectors are assumed to be fast in the sense that $d << D$. The authors present a consensus algorithm, based on a fast failure detector, that decides in $D+fd$ (they also show that this is optimal in the fast failure detector synchronous model).

While the fast failure detector-based model extends the classical synchronous model in allowing processes to obtain "early" information on failures, the proposed model extends the synchronous model in allowing a process to send additional control messages. These two approaches can be seen as complementary. Both enrich the classical round-based synchronous model with an additional mechanism (and both can be implemented with appropriate hardware or "urgent" messages).

Roadmap The paper is made up of 5 sections. Section 2 presents the computation model. Then Section 3 presents a consensus algorithm based on the round coordinator paradigm that requires $f+1$ rounds in the worst case, and a single round if the first coordinator does not crash during that round (so, when there is no crash, both our protocol and the fast failure detector-based protocol decide in a single round). Section 4 discusses the algorithm. Section 5 shows that $f + 1$ rounds is a lower bound in the proposed model, and consequently the proposed algorithm is optimal in this computation model.

2 Distributed Synchronous Model

2.1 The Extended Synchronous Model

The system consists of a set Π of n sequential processes, $\Pi = \{p_1, \ldots, p_n\}$, that communicate and synchronize by sending and receiving messages through channels. Every ordered pair of processes p_i and p_j is connected by a directed channel denoted (p_i, p_j). The underlying communication system is assumed to be failure-free: there is no creation, alteration, loss or duplication of message. The only failure a process can experience is a *crash*: it stops prematurely. Moreover, once crashed, a process remains crashed forever. A process is *correct* in a run if it does not crash during that run; otherwise it is *faulty*. As already indicated, t ($< n$) denotes the maximum number of processes that are allowed to crash, and f ($0 \leq f \leq t$) the number of processes that crash in a given run (so, there are $n - f$ correct processes in the corresponding run).

The system is *synchronous*. This means that the processes execute in lockstep: a run is partitioned in a *sequence of rounds*. There is a global variable r, provided by the model, that takes the successive integer values $1, 2, \ldots$, etc. (r can be seen as a global logical clock that progresses automatically, i.e., independently of the processes). A process p_i can only read it. A round r is made up of three consecutive phases:

- A send phase divided into two steps executed sequentially:
  - A *data sending step*: During that step, each process p_i (that has not crashed) can send a data message to a set of processes $Dest1(r, i) = \{p_{j_1}, \cdots, p_{j_k}\}$. (It is possible that different destination processes be sent different messages by p_i. Moreover, $Dest1(r, i)$ is not statically defined: it can be defined according to the sending process p_i and the round number.) If p_i crashes during that sending step, an arbitrary subset of the messages it was assumed to send are actually received by their destination processes.

– A *control data sending step*: During that step, p_i can send a control message to other processes, say $Dest2(r,i) =< p_{k_1}, \cdots, p_{k_\ell} >$. This is an ordered sequence of processes. If p_i crashes during that step, the control message is sent to an arbitrary prefix of the sequence $< p_{k_1}, \cdots, p_{k_\ell} >$.

It is important to notice that no local computation is allowed to take place between the two sending steps. The second sending step is executed just after the first one, without break.

- A receive phase in which each process receives messages. The fundamental property of the synchronous model lies in the fact that a message sent by a process p_i to a process p_j at round r, is received by p_j at the same round r. [3]

- A computation phase during which each process processes the messages it has received during that round and executes local computation. (This is the only place where a process executes local computation.)

2.2 Short Discussion

Computability power It is easy to see that if we suppress the second sending step (no control message is ever sent) we obtain the traditional synchronous model [3, 14, 16].

Sending each control message in separate consecutive rounds provides a (non-efficient) simulation in the other direction. (Using additional separate rounds allows ensuring that the control messages are sent in the prescribed order.) It follows that the proposed model and the basic synchronous model have the same computability power.

Cost of a round In a message-passing distributed system, it is usually considered that message transfer delays take time while local computation do not. (In some papers, the processing time associated with messages is "integrated" in their transfer delays.)

Let D be the time duration of a round in the traditional synchronous model. This means that D is an upper bound on message transfer delay + local processing time. The extended model adds the second sending step to each round. This sending step has to be done *after* the first sending step, and its control messages have to be sent in a *specified order*. This may require an additional time δ. It is important to notice that a process cannot receive and process messages between these two communication steps; this means that there is no waiting period before the second sending step. So, the the duration of a round in the extended model is $D + \delta$ with $\delta << D$ (as, differently from D, δ has not to capture message transfer delay[4]).

Let us consider an algorithm that requires $f+2$ rounds in the classic synchronous model, and an algorithm that solves the same problem in $f + 1$ rounds in the extended model. (Let us notice that $f + 2$ and $f + 1$ are the lower bounds for solving consensus in the classic model and the extended model, respectively.) Although failures are possible they are rare in practice, which means that 0 and 1 are the most common values for f.

The algorithm in the extended model performs better than the algorithm in the classic model when $(f + 1)(D + \delta) < (f + 2)D$, i.e., when $f + 1 < \frac{D}{\delta}$, which is always satisfied for realistic values of f. This means that the extended model is practically relevant for systems built on top local area networks whose communication is reliable. (As noticed in the introduction, this advantage disappears -and the model is no longer interesting- when, due to message losses, retransmission is required to obtain upper layer reliable channels.)

3 A Consensus protocol

This section presents a uniform consensus protocol suited to the proposed synchronous model. As a noteworthy side effect, as we will see in Section 4, this protocol, establishes a bridge between synchronous and asynchronous consensus.

3.1 The Consensus Problem

The consensus problem has been sketched in the introduction: every process p_i *proposes* a value v_i and all correct processes have to *decide* on some value v, in relation to the set of proposed values. More precisely, the consensus problem is defined by the following three properties:

- Termination: Every correct process eventually decides.

- Validity: If a process decides v, then v was proposed by some process.

- Agreement: No two correct processes decide different values.

Let us observe that the agreement property is only on correct processes: it allows a faulty process to decide differently from the correct processes. Such a property can be too weak for some applications that require a single decision

[3] This means that a channel never contain more than two messages, namely, a data message and a one-bit control message.

[4] If p_i sends a data message and a control message to p_j during a round r, these two messages are pipelined in the channel (p_i, p_j), and consequently $\delta << D$.

```
Function Consensus (v_i):
(1)  est_i ← v_i;
(2)  when r = 1,2,... ,t+1 do
(3)  begin_round
(4)     case (i = r) then for_each p_j ∈ {p_{i+1},... ,p_n}  do send DATA (est_i) to p_j end_do;
(5)                       for p_j from p_n until p_{i+1}      do send COMMIT () to p_j end_do;
(6)                       return (est_i)
(7)          (i > r) then if (DATA (v) received from p_r)     then est_i ← v     end_if;
(8)                       if (COMMIT () received from p_r)    then return (est_i) end_if
(9)          (i < r) then % cannot happen %
(10)    end_case
(11) end_round
```

Figure 1. Uniform consensus in at most $f+1$ rounds (code for process p_i)

whatever the deciding process be faulty or correct. *Uniform agreement* is a strengthened form of agreement that prevents such scenarios. More precisely, *uniform consensus* is defined by the previous Termination and Validity requirements plus the following Agreement property:

- Uniform Agreement: No two (correct or not) processes decide different values.

All the lower bounds that have been previously cited concern uniform consensus. Moreover, as the paper considers only the uniform consensus problem, the word "uniform" is sometimes omitted.

3.2 The Protocol

A consensus protocol suited to the extended model is described in Figure 1. Each process p_i invokes the function *Consensus* (v_i) where v_i is the value it proposes. It terminates either when it crashes or when it invokes the statement return (v) that provides it with the decided value v (line 6 or 8). The local variable est_i (initialized to v_i, line 1) contains p_i's current estimate of the decision value.

The protocol is surprisingly simple. It is based on the rotating coordinator principle[5] and uses the synchronization power provided by the second sending step executed without a break just after the first one. The first coordinator is p_1, then p_2, etc. There are at most $t+1$ processes that will play the coordinator role. For a process p_i, there are two cases, according to the fact that p_i is (or is not) the coordinator of the current round r.

- p_i is the coordinator ($i = r$): in that case, p_i tries to impose its current estimate value est_i as the decision value. To attain this goal, it sends est_i (data message) to all the processes that have a higher identity (line 4). It is important to notice that, if p_i succeeds in executing line 4 without crashing, all the non-crashed processes will have their estimates equal to est_i by the end of the round (that property will be used to prove uniform agreement).

  Then, p_i sends a synchronization message (commit) in the following order: first to p_n, then to p_{n-1}, etc., until p_{i+1} (line 5). This synchr message is to inform the processes that all of them know the estimate value of the current coordinator. Finally, (if it has not crashed before) p_i decides its current estimate value (line 6).

- p_i is not the coordinator ($i \neq r$): in that case we necessarily have $i > r$. (This is due to the following reason. As the successive values of the round number are the consecutive positive integers (line 2), $i < r$ would mean that p_i has already been the coordinator of the round $r' = i < r$. But then, when it has executed that round, p_i has either decided at line 6 or crashed.)

  So, when p_i executes a round $r < i$, it waits for the value v that the coordinator of r tries to impose. If it receives it, it adopts it (line 7). If additionally it receives the synchronization message, it knows that all the estimates values are equal to v. It then decides it (line 8).

It is easy to see that if the first coordinator (p_1) does not crash, the decision is obtained in one round, whatever the number of faulty processes. If it crashes while p_2 does not, the decision is obtained in at most two rounds, etc. This shows that the extended synchronous model allows designing algorithms that are both simple and efficient (each property "simplicity *vs* efficiency" not being obtained at the detriment of the other).

3.3 Proof and Complexity

This section proves that the protocol described in Figure 1 solves the (uniform) consensus problem in at most $f+1$

[5] Several coordinator-based consensus algorithms have been designed for asynchronous systems equipped with unreliable failure detectors (e.g., [5, 12, 15]). All the consensus algorithms for synchronous systems that we are aware of are based on the flooding strategy [3, 14, 16]: at every round, each process sends to all the other processes the new values it has received during the previous round.

rounds (of the extended model). Let $est_i[r]$ denote the value of est_i at the end of the round r.

Lemma 1 [Validity] *A decided value is a proposed value.*

Due to page limitation, the proof is given in [4].

Lemma 2 [Agreement] *No two different values are decided.*

Proof We first claim that there is a round ($1 \leq r \leq t+1$) such that the corresponding coordinator process executes entirely line 4 (*Claim* C1). Let r be the first of these rounds. From the fact that p_r executes entirely line 4, it follows that $est_j[r] = v, \forall j \geq r$, where v is the estimate value of p_r at the beginning of round r (A). We also claim (*Claim* C2) that no process p_j has decided before r, and all the processes p_j such that $1 \leq j < r$ have crashed before r. It follows from C2 that no process decides before r, and the processes that decide are a subset of $\{p_r, p_{r+1}, \ldots, p_n\}$.

Let p_i be a process that decides (so, $r \leq i \leq n$) at round $r_i \geq r$. If $r_i = r$, p_i decides v (the current estimate of the coordinator of r) as, during r, it has received (at line 7) and committed (at line 8) that value. (If p_i is p_r, it decides "quicker" at line 6.) If $r_i > r$, due to the property A (from the end of r, no estimate value present in the system is different from v), p_i cannot decide a value different from v.

It follows that a single value can be decided, namely, the current estimate value v of the first coordinator that successfully executes entirely line 4[6].

Proof of the claim C1. If p_1 does not crash before having executed entirely line 4, the claim follows. If p_1 crashes while p_2 does not, the claim follows too. So, let us assume that none of the processes $p_1, \ldots, p_{t+1}$ executes entirely line 4. This is impossible as there at most t faulty processes. (Let us notice that it is possible that the first process p_r that executes entirely line 4 crashes just after that line.) *End of the proof of the claim* C1.

Proof of the claim C2. If a process p_j decides during a round $r' < r$, either it is the coordinator of r' ($r' = j$), or it received a data message and a commit message from the coordinator $p_{r'}$ during that round. In both cases, this means that $p_{r'}$ has entirely executed line 4. But this contradicts the fact that $r > r'$ is the first round during which a coordinator executes entirely line 4.

We now show that each process p_j (with $j < r$) has crashed. This follows from the fact that, if p_j was not crashed by the end of the round j, it would have coordinated that round and decided. As p_j has not decided (as shown by the previous observation), it has necessarily crashed. *End of the proof of the claim* C2. $\square_{Lemma\ 2}$

[6] Some authors say that the value v is then *locked* [5, 12]: even if it is not known by some or all the processes, no value different from v can be decided. Line 4 is consequently a *value locking* mechanism.

Lemma 3 [Termination] *Let f be the number of processes that crash during a given run. During that run, every correct process decides, and no process decides after the round $f+1$.*

Proof If p_1 does not crash, it executes entirely the lines 4-6 during the first round. Consequently, each non-crashed process p_i receives both the data and the commit message. It follows that each non-crashed process p_i decides at line 8 (p_1 decides quicker at line 6). (If p_1 crashes, while p_2 does not, etc.)

So, let us assume that the first f processes are faulty, each having crashed before executing the line 6 of the round it coordinates. Due to the definition of f, and the fact that $p_1, \ldots, p_f$ have crashed, the process p_{f+1} is correct. We consider two cases.

- The first case is when p_{f+1} decides before the round $f+1$. In that case, it decided during a round $r < f+1$. Before deciding it received a data and a commit message during round r. Moreover, as p_r sent the commit message starting from p_n, then p_{n-1}, etc., until p_{r+1} (line 5), we can conclude that all the processes p_k such that $f+1 \leq k \leq n$ have received both messages. It follows that all these processes p_k that have not crashed decide during the round $r < f+1$. As the processes p_j such that $j < f+1$ are faulty, the termination requirement does not apply to them.
- The second case is when p_{f+1} does not decide before the round $f+1$. In that case, as the round number progresses, we eventually have $r = f+1$. Then, as p_{f+1} is correct, during that round, it executes the lines 4-6. Consequently, all the non-crashed processes p_j (that by assumption are such that $j \geq f+1$) receive the data and the commit message sent by p_i and decide accordingly.

$\square_{Lemma\ 3}$

Theorem 1 *The protocol solves the (uniform) consensus problem, and no process decides after the round $f+1$ (where f is the number of actual failures).*

Proof Follows from the Lemmas 1, 2 and 3. $\square_{Theorem\ 1}$

Theorem 2 *Let $|v|$ be the bit size of a proposed value ($|v| \geq 2$). The bit complexity varies between $(n-1)(|v|+1)$ (best), and $(f+1)(n-1-\frac{f}{2})|v| + (f+1)(n-f)$ (worst).*

Proof Let us first observe that $|v|+1$ is the total size of a data message ($|v|$ bits) plus a commit message (one bit)[7].

- Best case. This case is when there is no crash. In that case a single round, coordinated by p_1, is executed, and no other process sends messages. That process sends a data message and a control message to each other process. Hence, the bit complexity $(n-1)(|v|+1)$.

- Worst case. In that case, f processes are faulty, and those are the first f coordinators. We look for an upper bound on the messages transmitted, by considering the following worst case scenario.

  The first coordinator sends $(n-1)$ data messages, and sends commit messages from p_n until $_{f+1}$ and then crashes. Then, the second coordinator sends $(n-2)$ data messages, and sends commit messages from p_n until $_{f+1}$ and then crashes. Etc. until p_{f+1} that sends $n-(f+1)$ data messages plus $n-(f+1)$ commit messages. Summing up for the data messages, we obtain:

$$\begin{aligned} & (n-1)+\ (n-2)\ +\cdots+(n-(f+1)) \\ = & (n-1)+(n-1)-1+\cdots+(n-1)-f) \\ = & (n-1)(f+1)-\textstyle\sum_{x=1}^{f} x \\ = & (n-1)(f+1)-\frac{f(f+1)}{2} \\ = & (f+1)(n-1-\frac{f}{2}). \end{aligned}$$

  So, we have $(f+1)(n-1-\frac{f}{2})|v|$ bits for the data messages. Summing up for the commit messages, we obtain: $(f+1)(n-f)$ bits. The total number of messages is upper bounded by $(f+1)(n-1-\frac{f}{2})+(f+1)(n-f)$ messages, i.e., $(f+1)(2n-1-3f/2)$.

$\square_{Theorem\ 2}$

4 Discussion

It is interesting to compare the proposed rotating coordinator-based synchronous algorithm and the algorithm proposed in [15] (let us call it MR99) designed for asynchronous systems equipped with a failure detector of the class $\diamond\mathcal{S}$ (this class of failure detectors is the weakest that allows solving consensus in asynchronous systems prone to process failures). Differently from synchronous systems, each message in an asynchronous system has to carry its round number in order the receivers do not confuse messages sent at different rounds.

In MR99, each (asynchronous) round is coordinated by a process, and is made up of two consecutive communication steps. More precisely, during a MR99 round we have:

- During the first communication step, the current coordinator p_c sends its current estimate v to all the processes. For each process p_i, that communication step ends when it receives the current coordinator's estimate v or suspects p_c. When it terminates that step, p_i sets a local variable aux_i to v if it received it, or $\perp$ if it suspects p_c. That variable represents its local knowledge of the estimate of the current round coordinator (p_c).

- After it has determined the value of aux_i, every process p_i executes a second communication step during which it sends its local knowledge aux_i to all the processes. After that sending, p_i waits until it has received "enough" corresponding messages from the other processes. "Enough" means here "from $n-t$ processes", in order to get as many messages as possible while preventing deadlock.

  Then, p_i decides v if it has received that value from a majority of processes. The "majority of correct processes" requirement is needed to guarantee the consensus agreement property (this is a necessary requirement in asynchronous message-passing systems equipped with a $\diamond\mathcal{S}$ failure detector [5]). If p_i has not received enough messages displaying v, it adopts v as its new estimate and proceeds to the next round. If all the aux_j values it has received are such that $aux_j=\perp$, p_i proceeds to the next round without modifying its estimate.

So, in MR99, the second communication step of a round involves all the processes and is strongly synchronized with respect to the first as the value sent by a process during that step depends on the value it has (or not) received during the previous step. If p_i receives a majority of $aux_j=v\neq\perp$, it knows that, by the end of the current round, v is eventually known by all the processes, and is consequently *locked*. In the algorithm presented in Figure 1, this information is provided by the commit message. Due to the additional synchrony assumption of the extended model, (1) that message can safely be sent by a single process, namely the coordinator p_c of the current round itself, and (2) this sending can be issued just after p_c has broadcast its estimate v to all the processes (i.e., without additional synchronization requiring message exchange).

This shows that the proposed synchronous algorithm and MR99 can be seen as two implementations in different settings of the very same basic principle. From an algorithm design point of view, this displays a deep unity in the principles that underlie the design of a family of simple and efficient consensus algorithms suited to (a)synchronous distributed computing models.

[7] When a process receives a message whose size is one bit (this is a pure signal carrying no value), it knows the message is a commit message. If the size is at least two bits, it knows that the message is a data message.

5 Lower Bound in the Extended Model

The previous protocol requires $f+1$ rounds of the extended model, which means that, when $f+1 < \frac{D}{\delta}$ (see the discussion in Section 2.2), it is more efficient than an $(f+2)$-round algorithm executed in the classic synchronous model. This section shows that $f+1$ is actually a lower bound in the extended model. It follows that the proposed protocol is optimal for that distributed computing model.

The proof is as follows. First we extend a result from Aguilera and Toueg [2] to show that t-resilient uniform consensus requires at least $t+1$ rounds in the extended model (Theorem 3). Then we use this result to show that $f+1$ is a lower bound in the extended model (Theorem 4). This section assumes $t < n-1$ (this is because, some proofs requires at least two correct processes in order to compare their view of the computation).

5.1 Lower Bound for t-Resilient Consensus

Aguilera and Toueg have shown that any t-resilient consensus algorithm requires at least $t+1$ rounds in the classic synchronous model [2]. Their proof is simpler than previous proofs devoted to the same result. It is based on a bivalency argument (initially introduced to prove the impossibility of asynchronous consensus despite one crash) [11]. Our proof is nearly the same as Aguilera and Toueg's proof[8]. Due to page limitation, the proofs can be found in [4].

Theorem 3 [4] *Let us consider the extended synchronous model where $t < n-1$, and at most one process crashes in each round. There is no algorithm that solves consensus in t rounds in this model*[9].

5.2 Lower Bound for Early-Stopping

Theorem 4 *Let $t < n-1$. Uniform consensus requires at least $f+1$ rounds in the extended synchronous model.*

Proof *Claim*: Let A be a uniform consensus algorithm that tolerates up to t crashes in the extended synchronous model ($t < n-1$). Then, $\forall f:\ 0 \leq f \leq t$, there is a run of A in which at least one process does not decide by the end of f. *End of the claim.*

That claim is an immediate consequence of Theorem 3 obtained by replacing t by the number f of actual crashes. The theorem follows directly from it. $\square_{Theorem\ 4}$

[8] The fact that the proof of Aguilera and Toueg can be easily extended to a new model, shows that (1) the *valency* notion introduced in [11] is a central concept to understand and master distributed agreement problems, and (2) the proof of [2] captures the essence of the impossibility to solve consensus in less than $t+1$ rounds. (In that sense, this section can be considered as an exercise whose aim is to generalize a previous result.)

[9] As there is no algorithm that solves consensus, there is no algorithm that solves uniform consensus either.

Theorem 5 *Let $t < n-1$. The uniform consensus algorithm described in Figure 1 is round optimal for the extended failure model.*

Proof Immediate consequence of Theorem 4 (that states "the best that can be done in the worst case is $f+1$"), and Theorem 1 (that states "the algorithm requires at most $f+1$ rounds"). $\square_{Theorem\ 5}$

This theorem states the power and fixes the limit of the additional synchronization provided by the extended synchronous model defined in Section 2.

References

[1] Aguilera M.K., Le Lann G. and Toueg S., On the Impact of Fast failure Detectors on Real-Time Fault-Tolerant Systems. *Proc. 16th Symposium on Distributed Computing (DISC'02)*, Springer-Verlag LNCS #2508, pp. 354-369, 2002.

[2] Aguilera M.K. and Toueg S., A Simple Bivalency Proof that t-Resilient Consensus Requires $t+1$ Rounds. *IPL*, 71:155-158, 1999.

[3] Attiya H. and Welch J. Distributed Computing: Fundamentals, Simulations and Advanced Topics. *McGraw-Hill*, 451 pages, 1988.

[4] Cao J., Raynal M. and Zhang W., The Power and Limit of Synchronization for Synchronous Agreement. *Tech Report #1710*, IRISA, Campus de Beaulieu, University of Rennes, France, 2005. http://www.irisa.fr/bibli/publi/pi/2005/1710/1710.html

[5] Chandra T.K. and Toueg S., Unreliable Failure Detectors for Reliable Distributed Systems. *J. ACM*, 43(2):225-267, 1996.

[6] Chandy K.M. and Lamport L., Distributed Snapshots: Determining Global States of Distributed Systems. *ACM Transactions on Computer Systems*, 3(1):63-75, 1985.

[7] Charron-Bost B. and Schiper A., Uniform Consensus is Harder than Consensus. *Journal of Algorithms*, 51(1):15-37, 2004.

[8] Dolev D., Reishuk R. and Strong H.R., Early Stopping in Byzantine Agreement. *Journal of the ACM*, 37(4):720-741, 1990.

[9] Dwork C., Lynch N. and Stockmeyer L., Consensus in the Presence of Partial Synchrony. *Journal of the ACM*, 35(2):288-323, 1988.

[10] Fischer M.J. and Lynch N., A Lower Bound for the Time to Assure Interactive Consistency. *IPL*, 71:183-186, 1982.

[11] Fischer M.J., Lynch N.A. and Paterson M.S., Impossibility of Distributed Consensus with One Faulty Process. *Journal of the ACM*, 32(2):374-382, 1985.

[12] Guerraoui R. and Raynal M., The Information Structure of Indulgent Consensus. *IEEE Transactions on Computers*, 53(4), 53(4):453-466, April 2004.

[13] Keidar I. and Rajsbaum S., A Simple Proof of the Uniform Consensus Synchronous Lower Bound. *IPL*, 85:47-52, 2003.

[14] Lynch N.A., Distributed Algorithms. *Morgan Kaufmann Pub.*, San Fransisco (CA), 872 pages, 1996.

[15] Mostéfaoui A. and Raynal M., Solving Consensus Using Chandra-Toueg's Unreliable Failure Detectors: a General Quorum-Based Approach. *Proc. 13th Symp. on Distributed Computing (DISC'99)*, Springer Verlag LNCS #1693, pp. 49-63, 1999.

[16] Raynal M., Consensus in Synchronous Systems: a Concise Guided Tour. *Proc. 9th IEEE Pacific Rim Int'l Symposium on Dependable Computing (PRDC'02)*, IEEE Comp. Press, pp. 221-228, 2002.