The Perfectly-Synchronized Round-based Model of Distributed Computing

Carole Delporte-Gallet [a], Hugues Fauconnier [a], Rachid Guerraoui [b], Bastian Pochon [b,*]

[a] *LIAFA Institute, Université Denis Diderot, F-75251 Paris 5*

[b] *School of Computer and Communication Sciences, EPFL, CH-1015 Lausanne*

Abstract

The perfectly-synchronized round-based model provides the powerful abstraction of crash-stop failures with atomic and synchronous message delivery. This abstraction makes distributed programming very easy. We describe a technique to automatically transform protocols devised in the perfectly-synchronized round-based model into protocols for the crash, send omission, general omission or Byzantine models.

Our transformation is achieved using a round shifting technique with a constant time complexity overhead. The overhead depends on the target model: crashes, send omissions, general omissions or Byzantine failures. Rather surprisingly, we show that no other automatic non-uniform transformation from a weaker model, say from the traditional crash-stop model (with no atomic message delivery), onto an even stronger model than the general-omission one, say the send-omission model, can provide a better time complexity performance in a failure-free execution.

Key words: Abstraction, simulation, distributed systems, fault-tolerance, synchronous system models, complexity.

1 Introduction

1.1 Motivations

Distributed programming would be easy if one could assume a *perfectly-synchronized round-based model* where the processes would share, after every

* Corresponding author. Present address: EPFL IC IIF LPD, Station 14, CH-1015 Lausanne, Switzerland. Email address: Bastian.Pochon@epfl.ch

round, the same view of the distributed system state. Basically, computation would proceed in a round-by-round way, with the guarantee that, in every round, a message sent by a correct process is received by all processes, and a message sent by a faulty process is either received by all or by none of the processes. All processes that reach the end of a round would have the same view of the system state.

Unfortunately for the programmers, and fortunately for the distributed computing research community, the assumption that all processes have the same view of the system state does not hold in practice. In particular, the illusion of a perfectly-synchronized world breaks because messages sent over a network might be subject to partial delivery or message loss, typically because of a buffer overflow at a router, or due to a crash failure, resulting from the crash of some computer hosting processes involved in the distributed computation.

It is of course legitimate to figure out whether we could provide the programmer with the simple view of a perfectly synchronized world, and translate, behind the scenes, distributed protocols devised in such an ideal model into more realistic and weaker models. After all, the job of a computer scientist is usually about providing programming abstractions that hide low level details, so why not try to provide those that really facilitate the job of the programmer of distributed applications.

The very fact that the abstraction of a perfectly-sychronized round-based model has not already been made available to programmers through popular programming middleware, even after several decades of research in distributed computing, might indicate that its implementation might turn out to be significantly involved. Indeed, a closer look at the semantics of the *perfectly-synchronized round-based (PSR) model* reveals that what needs to be implemented is actually a succession of instances of an agreement algorithm, more precisely an algorithm solving the *Interactive Consistency (IC)* problem [18]. Indeed, this is the key to provide processes with the same view of the system at the end of every round. Roughly speaking, in the IC problem, each process is supposed to propose a value and eventually decide on a vector of values, such that the following properties are satisfied: termination (i.e., *every correct process eventually decides on a vector*), validity (i.e., *the j^{th} component of any decided vector by a correct process is the value proposed by process p_j if p_j is correct*), and agreement (i.e., *no two correct processes decide on different vectors*).

The relationship between Interactive Consistency and the perfectly-sychronized round-based model highlights two issues. The first has to do with feasibility. On the one hand, to implement the PSR abstraction over a given model, one needs to make some synchrony assumptions on the model (e.g., the Interactive Consistency problem is not solvable in an eventually synchronous model [10]), and the coverage of these assumptions might simply not be sufficient for certain distributed environments. The second issue has to do with performance. Even when the PSR abstraction can be implemented, the cost of its implementation might be too high. That is, devising a distributed protocol over PSR, and relying on the implementation of PSR to automatically generate a distributed protocol in a weaker model might have a significant overhead with respect to devising the protocol directly in the latter model.

The lack of any evidence concerning the exact overhead of implementing the PSR absraction was the motivation of this work. More precisely, the motivation was to figure out whether we can come up with an efficient implementation, in terms of time complexity, of the PSR abstraction over synchronous round-based models with various types of failures, ranging from simple crash failures [12] to more general Byzantine failures [18,15], including send-omissions [12] and general-omissions [16].

1.2 Background

The PSR abstraction is known to have implementations in all the models mentioned above, but the inherent cost of these implementations in either of these models was unclear. The lack of any result on the cost of implementing PSR might seem surprising given the amount of work that was devoted to devising optimal agreement algorithms over various models, including the omission model and the Byzantine model.

(1) In particular, we do know that, in terms of round complexity, there is a tight lower bound on implementing interactive consistency in a synchronous round-based model where t processes can crash is $t+1$ [7]. The result is derived for the model with crashes, and thus also holds for send-omissions, general-omissions, and Byzantine failures. The result says that $t+1$ rounds of, say, the general-omission model are needed for all correct processes to reach a decision about the new global state of the distributed system (i.e., the decision vector). If, pretty much like in state machine replication [13,21], we implement PSR as a sequence of instances of interactive consistency, then the $t+1$ cost would add up. In other words, $K(t+1)$ rounds would be needed to implement K rounds of PSR.

One might wonder whether algorithms that are *early deciding* [14,5] would decrease this cost. Indeed, these algorithms need fewer rounds for processes to decide when only f failures occur, out of the total number t of failures that are tolerated. These algorithms however do not guarantee a *simultaneous* decision from all the processes [4], even from the correct processes only. In such a case, it would then be necessary to delay the simulation of the next PSR round until each process reaches the next multiple of $t+1$ rounds. In other words, $K(t+1)$ rounds would again be needed to implement K rounds of PSR.

(2) Implementing a synchronous round-based model with crash failures [12] (*crash-stop* model) over various weaker models, such as the omission model, has been the subject of several investigations, e.g. [1,17]. These can be viewed as implementing an abstraction that is weaker than PSR. (PSR prevents a message from being received by some but not all the processes, whereas the crash-stop model does not, in case the sender crashes.) The idea underlying the implementation proposed in [17], for instance for the omission model, is that of *doubling* rounds. Roughly speaking, any round of the crash-stop model is simulated with two rounds of the omission model. Hence, $2K$ rounds of the omission model are needed to simulate K rounds of the crash-stop model.

In the first case where we use a sequence of interactive consistency instances, or in the second case where we mask failures by doubling rounds, we end up with multiplicative factor overheads, and even if we try to implement the weaker crash-stop abstraction along the lines of [17]. In fact, if we implement PSR directly on the crash-stop model (used as an intermediate model), and use the transformation of [17], we end up with a cost of $K(f+1)$ rounds of the omission model for K rounds of the PSR model with f actual failures. Is this multiplicative factor inherent to implementing PSR over an omission model? Or could we devise a *shifting* implementation with an additive factor, i.e., $K+S$ with S a constant? At first glance, this would be counter-intuitive because it would mean devising a more efficient implementation than [17] for an abstraction that is strictly stronger.

1.3 Contributions

This paper presents a time-efficient shifting technique to implement the PSR abstraction over the synchronous round-based message-passing models with crash failures, send-omissions, general-omissions and Byzantine failures (for $t < n/3$): K rounds of PSR require at most $K+t$ rounds of the model with more severe failures, when t failures are tolerated. That is, with an additive factor $S = t$. This is clearly optimal because PSR solves interactive consistency in one round, and this costs at least $t+1$ rounds in either model (with crash failures, send or general omissions, or Byzantine failures) [7]. In other words, any shifting transformation technique from the PSR model to the omission model has to pay the cost of t additional rounds.

This paper gives both a uniform and a non-uniform shifting transformation. Intuitively speaking, a uniform transformation ensures that any process, be it correct or faulty, simulates a correct behavior according to the original algorithm (i.e., the algorithm to be transformed), or nothing at all, whereas a non-uniform transformation does not ensure faulty processes simulate a correct behavior with respect to the original algorithm. For both transformations, we make the details clear about the underlying Interactive Consistency algorithms that are used, respectively uniform and non-uniform. Our shifting transformations do not necessarily require that all processes decide simultaneously within each underlying Interactive Consistency instance, hence the use of early-deciding algorithms is possible. By considering an early-deciding non-uniform Interactive Consistency algorithm, we show that our shifting transformation works in "real-time" in a failure-free execution, i.e., the transformed algorithm executes as fast as the original algorithm. In this precise case, it is clear that the transformation is optimal, since K round of PSR are transformed into K rounds of the target model.

We precisely define the general notion of transformation and then describe our novel shifting transformation technique. Beforehand, we introduce the necessary machinery to formulate the definitions of simulation and transformation. The key idea of our technique is that a round in the weak model (crash, send or general omission, Byzantine failure), is involved in the simulation of more than one round of PSR. This is also the source of some tricky algorithmic issues that we had to address.

2 Model

2.1 Processes

We consider a finite set Ω of n processes $\{p_0, \ldots, p_{n-1}\}$, that communicate by point-to-point message-passing. We assume that processes are fully connected. A process is characterized by its *local state* and we denote by $\mathcal{S}$ the set of possible states of any process. Processes interact in a synchronous, round-based computational way. Let $\mathcal{R} = \mathbb{N}^*$ be the set of round numbers (strictly positive, integer numbers). We denote by $\mathcal{M}$ the set of messages that can be sent, and by $\mathcal{M}' = \mathcal{M} \cup \{\perp\}$ the set of messages that can be received. $\perp$ is a special value that indicates that no message has been received. The primitive $\mathsf{send}()$ allows a process to send a message to the processes in Ω. The primitive $\mathsf{receive}()$ allows a process to receive a message sent to it that it has not yet received. We assume that each process receives an input value from the external world, at the beginning of every round, using the primitive $\mathsf{receiveInput}()$. We denote by $\mathcal{I}$ the set of input values that can be received, for all processes. An *input pattern* is a function $I : \Omega \times \mathcal{R} \rightarrow \mathcal{I}$. For any given process p_i and round number r, $I(i, r)$ represents the input value that p_i receives at the beginning of round r. For any given set of input values

$\mathcal{I}$, we denote by $\Gamma_{\mathcal{I}}$ the set composed of all input patterns over $\mathcal{I}$. For the sake of simplicity, we assume that input values do not depend on the state of processes. In Section 5, we discuss an extension where this assumption is relaxed.

Roughly speaking, in each synchronous round r, every process goes through four, non-atomic steps (in particular, the processes do not have any atomic broadcast primitive for executing the second step). In the first step, the process receives an external input value. In the second step, the process sends the (same) message to all processes (including itself). In the third step, the process receives all messages sent to it. The fourth step is a local computation to determine the next local state of the process.

Throughout the paper, if a variable v appears in the local state of all processes, we denote by v_i the variable at process p_i, and by v_i^r the value of v after p_i has executed round r, but before p_i has started executing round $r+1$. For convenience of notation, v_i^0 denotes the value of v at process p_i after initialization, before p_i takes any step.

2.2 Protocols

The processes execute a *protocol* $\Pi = \langle \Pi_0, \ldots, \Pi_{n-1} \rangle$. Each process p_i executes a state machine Π_i, defined as a triple $\langle s_i, T_i, O_i \rangle$, respectively an initial state, a state transition function and a message output function. We assume that, at any process p_i, the corresponding state machine is initialized to s_i. The message output function $O_i : \mathcal{S} \times \mathcal{I} \times \mathcal{R} \rightarrow \mathcal{M}$ generates the message to be sent by process p_i during round r, given its state at the beginning of round r, an external input value, and the round number. Note that, throughout this paper, we assume for presentation simplicity that processes always have a value to send, and we reserve the symbol $\perp$ for the very case where a message is not received, as the result of a failure. The state transition function T_i : $\mathcal{S} \times (\mathcal{M}')^n \times \mathcal{R} \rightarrow \mathcal{S}$ outputs the new state of process p_i, given the current state of p_i, the messages received during the round from all processes (possibly $\perp$ if a message is not received) and the current round number.

We introduce three functions for describing whether the execution of any protocol by any process is correct or deviate from the one intended. In the following functions, N denotes the interval of integer values $[1, n]$, corresponding to process identifiers in Ω.

- $ST : N \times \mathcal{R} \cup \{0\} \rightarrow \mathcal{S}$ is a function such that, for any process p_i and round r, $ST(i, r)$ is the state of process p_i at the end of round r. (Slightly abusing the notation, we define $ST(i, 0) = s_i$ for any process p_i.)
- $MS : N \times N \times \mathcal{R} \rightarrow \mathcal{M}'$ is a function such that, for any processes p_i, p_j

and round r, $MS(i,j,r)$ is the message sent by p_i to p_j in round r, or $\perp$ if p_i fails to send a message to p_j in round r.

- $MR : N \times N \times \mathcal{R} \rightarrow \mathcal{M}'$ is a function such that, for any processes p_i, p_j and round r, $MR(i,j,r)$ is the message received by p_i from p_j in round r, or $\perp$ if p_i fails to receive a message from p_j in round r. In the following, $MR(i,r)$ denotes the vector of all the messages received by p_i in round r, i.e., $MR(i,r) = [MR(i,1,r), \ldots, MR(i,n,r)]$.

2.3 Correctness

When we make no assumption whatsoever about the behavior of any process p_i, we consider that p_i behaves correctly, i.e., p_i follows the state machine Π_i assigned to it. Here we define the correct behavior of any process p_i more formally.

Any process p_i is *correct up to round* r, $r \geq 1$, if for any r', $1 \leq r' \leq r$, and any input pattern I:

- p_i does not fail in sending its message:

$$(\forall p_j \in \Omega)(MS(i,j,r') = O_i(ST(i,r'-1), I(i,r'), r')),$$

- p_i does not fail in receiving any message:

$$(\forall p_j \in \Omega)(MR(i,j,r') = MS(j,i,r')),$$

- p_i makes a correct state transition (p_i does not crash):

$$ST(i,r') = T_i(ST(i,r'-1), MR(i,r'), r').$$

By definition, any process is correct up to round 0.

2.4 Failures

If any process p_i does not follow the state machine Π_i assigned to p_i in any round r, i.e., p_i is correct up to round $r-1$ and is not correct up to round r, p_i is *faulty in round* r and may fail by either of the following types of failure. (For the sake of clarity, we indicate here the complete behavior of p_i in round r, not only the faulty part.)

Atomic failure. A process p_i that commits an atomic failure in round r can either crash before sending its message to all or after sending its message to all in round r. Processes do not recover after an atomic failure: a process that crashes due to an atomic failure in round r does not send nor receive any message in any subsequent round $r' > r$. More formally,

- p_i either crashes before sending any message:
  - p_i does not send any message to any process:

$$(\forall p_j \in \Omega)(MS(i,j,r) = \perp),$$

  - p_i does not receive any message from any process:

$$(\forall p_j \in \Omega)(MR(i,j,r) = \perp),$$

- or p_i crashes after sending a message to all and before receiving any message:
  - p_i sends a message to all processes:

$$(\forall p_j \in \Omega)(MS(i,j,r) = O_i(ST(i,r-1), I(i,r), r)),$$

  - p_i does not receive any message from any process:

$$(\forall p_j \in \Omega)(MR(i,j,r) = \perp).$$

In either cases, p_i does not perform any step after crashing:

- p_i does not send nor receive any message:

$$(\forall r' > r)(\forall p_j \in \Omega)(MS(i,j,r') = \perp \wedge MR(i,j,r') = \perp),$$

- p_i does not perform any state transition:

$$(\forall r' \geq r)(ST(i,r') = ST(i,r-1)).$$

Crash failure. A process p_i that commits a crash failure in a round r — or simply that crashes in round r — can either (i) send a message to a subset of the processes, crash, not receive any message, or (ii) send a message to all, receive a subset of the messages sent to it, and crash. Processes do not recover after crashing: a process that crashes in round r does not send nor receive any message in any subsequent round $r' > r$. More formally,

- p_i either sends its message to a subset of the processes, crashes, and does not receive any message:
  - p_i sends its message to p_j or nothing at all:

$$(\forall p_j \in \Omega)(MS(i,j,r) = O_i(ST(i,r-1), I(i,r), r) \vee MS(i,j,r) = \perp),$$

  - p_i does not receive any message:

$$(\forall p_j \in \Omega)(MR(i,j,r) = \perp),$$

- or p_i sends its message to all processes, receives the message from a subset of the processes, and crashes:
  - p_i sends its message to all processes:

$$(\forall p_j \in \Omega)(MS(i,j,r) = O_i(ST(i,r-1), I(i,r), r)),$$

  - p_i receives a message from a subset of the processes:

$$(\forall p_j \in \Omega)(MR(i,j,r) = MS(j,i,r) \vee MR(i,j,r) = \bot).$$

In either cases, p_i does not perform any step after crashing:

- p_i does not send nor receive any message:

$$(\forall r' > r)(\forall p_j \in \Omega)(MS(i,j,r') = \bot \wedge MR(i,j,r') = \bot),$$

- p_i does not perform any state transition:

$$(\forall r' \geq r)(ST(i,r') = ST(i,r-1)).$$

Send-omission failure. A process p_i that commits a send-omission in a round r fails to send its message in that round to a subset of processes in the system. More formally:

$$\begin{aligned}&(\forall p_j \in \Omega)(MS(i,j,r) = O_i(ST(i,r-1), I(i,r), r) \vee MS(i,j,r) = \bot) \wedge \\ &(\exists p_j \in \Omega)(MS(i,j,r) = \bot).\end{aligned}$$

Receive-omission failure. A process p_i that commits a receive-omission in a round r fails to receive a message from a subset of processes in the system. More formally:

$$\begin{aligned}&(\forall p_j \in \Omega)(MR(i,j,r) = MS(j,i,r) \vee MR(i,j,r) = \bot) \wedge \\ &(\exists p_j \in \Omega)(MR(i,j,r) = \bot \wedge MS(j,i,r) \neq \bot).\end{aligned}$$

General-omission failure. A process p_i that commits a general omission in a round r if p_i commits either a send- and/or a receive-omission failure in round r.

Byzantine failure. A process p_i that commits a Byzantine failure in round r may arbitrarily deviate from its protocol, there is no message authentication mechanism: p_i sends any message, alters any message that p_i has received, or relays spurious messages that appear to be from other processes. More formally, a process p_i that commits a Byzantine failure in round r performs at least one of the following items in round r and behaves correctly for the rest of round r:

- p_i fails to send correctly to at least one process:

$$(\exists p_j \in \Omega)(MS(i,j,r) \neq O_i(ST(i,r-1), I(i,r), r))$$

- p_i fails to receive correctly from at least one process:

$$(\exists p_j \in \Omega)(MR(i,j,r) \neq MS(j,i,r))$$

- p_i makes an incorrect state transition:

$$(ST(i,r) \neq T_i(ST(i,r-1), MR(i,r), r))$$

2.5 Runs

A *run* corresponds to an execution of a protocol, and is defined as a tuple $\langle I, ST, MS, MR\rangle$, where I is the input pattern observed in the run, ST is the state function, MS represents the messages sent, and MR the messages received.

Any process p_i is *correct in run* R if p_i is correct up to round r, for any $r \geq 0$. A process that is not correct in run R is *faulty in* R. Let $\mathit{correct}(R,r)$, $r \geq 1$, denote the set of processes correct up to round r in run R (all the processes are correct up to round 0). The set of correct processes in run R is $\mathit{correct}(R) = \cup_{r\geq 0}\mathit{correct}(R,r)$, whereas the set of faulty processes in run R is $\mathit{faulty}(R) = \Omega \backslash \mathit{correct}(R)$.

2.6 System Models and Problem Specifications

A *system model*, or model, is the particular set of all runs that can occur under some conditions (for any protocol). Hence a system model may be defined as a set of conditions that its runs must satisfy. We denote by $R(\Pi, M, \Gamma_{\mathcal{I}})$ the set of all runs produced by protocol Π in system model M and input pattern in $\Gamma_{\mathcal{I}}$. A problem specification, or problem, Σ is defined as a predicate on runs.

Definition 1 *A protocol* Π solves *a problem* Σ *in system model* M *with input pattern in* $\Gamma_{\mathcal{I}}$ *if and only if* $(\forall R \in R(\Pi, M, \Gamma_{\mathcal{I}}))(R$ *satisfies* $\Sigma)$.

A model M is defined as a particular set of runs. In particular, we define six distinct models:

- Model $PSR(n,t)$ (*Perfectly-synchronized round*) is defined by all runs over n processes where at most $t < n$ processes are subject to atomic failures, and the remaining processes are correct.
- Model $Crash(n,t)$ is defined by all runs over n processes where at most $t < n$ processes are subject to crash failures only, and the remaining processes are correct.
- Model $Omission(n,t)$ is defined by all runs over n processes where at most $t < n$ processes are subject to crash failures or send-omission failures in some round, and the remaining processes are correct.
- Models $General(n,t)$ and $General\text{-}MAJ(n,t)$ are defined by all runs over n processes where at most t, where $t < n$ processes, respectively $t < n/2$ processes, are subject to crash failures, send- and/or receive-omission failures in some round, and the remaining processes are correct.
- Model $Byzantine(n,t)$ is defined by all runs over n processes where at most $t < n/3$ processes are subject to Byzantine failures, and the remaining processes are correct.

We say that a model M_s is *stronger* than a model M_w, and we write $M_s \succeq M_w$, if and only if $M_s \subseteq M_w$. We say that a model M_s is *strictly stronger* than M_w, and we write $M_s \succ M_w$, if and only if $M_s \succeq M_w$ and $M_w \not\succeq M_s$. Weaker and strictly weaker relations are defined accordingly. From the equations above, it is clear that $PSR(n,t) \succeq Crash(n,t) \succeq Omission(n,t) \succeq General(n,t)$.

For any run R, in any model, we denote by f the effective number of faulty processes in R, i.e., $f = |faulty(R)|$.

3 Simulation and Transformation

The notions of simulation and transformation, although intuitive, require a precise definition. In particular, some problems in a given model cannot be transformed into another model, simply because they cannot be solved in the second model.

Consider two models M_s and M_w, such that $M_s \succeq M_w$. A transformation $\mathcal{T}$ takes any protocol Π_s designed to run in the strong model M_s and converts it into a protocol $\Pi_w = \mathcal{T}(\Pi_s)$ that runs correctly in the weak model M_w. For example, M_s could be *PSR* and M_w could be *Crash*. To avoid ambiguities, we call a round in the weak model M_w, a *phase*.

The transformation of a protocol Π_s in M_s to a protocol Π_w in M_w is defined through a *simulation function*, *Sim*, which simulates a run of Π_s by a run of Π_w. In [3], the authors present a problem, called the *Strong Dependent Decision* (SDD) problem, which is solvable in a synchronous model, and show that this problem does not admit any solution in an asynchronous model augmented with a *Perfect failure detector* [2] when one process can crash. This seems to contradict the fact that algorithms designed for the former model can be run in the latter [17]. The contradiction is in apparence only, and depends on how we define the notion of simulation.

For any process p_i executing a protocol Π_w in M_w simulating Π_s, the local state s of p_i contain variables $s.states_i$ and $s.ss_i$, which maintain the simulated states of protocol Π_s. Indeed, in contrast with the doubling technique of [17] where each state of the run in Π_w simulates *at most* one state of a run in Π_s, we do not restrict our transformation to simulate only one state of a run of Π_s in a state of the run of Π_w. More precisely, $s.states$ is a set of round numbers, such that, at the end of any phase x, for any round r in $s.states_i$, $s.ss_i[r]$ gives the r-th simulated state, i.e., the simulated state at the end of round r ($s.states_i^0 = \{0\}, s.ss_i^0[0] = s_i$). We now give the formal definitions of our transformation notions, over an arbitrary set of input values $\mathcal{I}$.

We first define the notion of non-uniform transformation, and then use this definition to define the notion of uniform transformation.

Definition 2 *An algorithm $\mathcal{T}$ is called a* non-uniform transformation from model M_s to model M_w, *with input pattern in $\Gamma_{\mathcal{I}}$, if there is a corresponding simulation function Sim and a function $f : \mathcal{R} \rightarrow \mathcal{R}$, with the following property: for any protocol Π_s and any run R_w of $\Pi_w = \mathcal{T}(\Pi_s)$ running in M_w with input pattern I_w, Sim maps run $R_w = \langle I_w, ST_w, MS_w, MR_w\rangle$ onto a corresponding simulated run $R_s = Sim(R_w)$ such that*

(i) $R_s = \langle I_s, ST_s, MS_s, MR_s\rangle$ *and* $R_s \in R(\Pi_s, M_s, \Gamma_{\mathcal{I}})$,

(ii) $correct(R_w) \subseteq correct(R_s)$,

(iii) $(\forall r \in \mathcal{R})(\forall p_i \in correct(R_w)(I'(i,r) = I(i,r))$,

(iv) $(\forall x \in \mathcal{R})(\forall p_i \in correct(R_w))(\forall r \in ST_w(i,x).states)$
$(ST_w(i,x).ss[r] = ST_s(i,r))$,

(v) $(\forall r \in \mathcal{R})(\forall p_i \in correct(R_s))(\exists c \leq f(r))(r \in ST_w(i,c).states)$,

(vi) $(\forall r, r' \in \mathcal{R}, r \neq r')(\forall p_i \in correct(R_w))$
$(x \in ST_w(i,r).states \cap ST_w(i,r').states \Rightarrow$
$ST_w(i,r).ss[x] = ST_w(i,r').ss[x])$,

(vii) $(\forall x \in \mathcal{R})(\forall p_i \in correct(R_w))(\forall r \in ST_w(i,x).states)(\forall r' < r)$
$(r' \in \cup_{k=0}^{x} ST_w(i,k).states)$.

Definition 3 *An algorithm $\mathcal{T}$ is called a* uniform transformation from model

M_s to model M_w, *with input pattern in* $\Gamma_{\mathcal{I}}$*, if* $\mathcal{T}$ *is a non-uniform transformation from* M_s *to* M_w *with simulation function Sim and function* f *satisfying the properties of a non-uniform transformation and such that, for any protocol* Π_s *and any run* R_w *of* $\Pi_w = \mathcal{T}(\Pi_s)$ *running in* M_w *with input pattern* I_w*, Sim maps run* $R_w = \langle I_w, ST_w, MS_w, MR_w\rangle$ *onto a corresponding simulated run* $R_s = Sim(R_w)$*, the additional following properties are also satisfied:*

(iii') $I_w = I_s$,

(iv') $(\forall x \in \mathcal{R})(\forall p_i \in \Omega)(\forall r \in ST_w(i,x).states)(ST_w(i,x).ss[r] = ST_s(i,r))$,

(vi') $(\forall r, r' \in \mathcal{R}, r \neq r')(\forall p_i \in \Omega)$
$(x \in ST_w(i,r).states \cap ST_w(i,r').states \Rightarrow$
$ST_w(i,r).ss[x] = ST_w(i,r').ss[x])$,

(vii') $(\forall x \in \mathcal{R})(\forall p_i \in \Omega)(\forall r \in ST_w(i,x).states)(\forall r' < r)$
$(r' \in \cup_{k=0}^{x} ST_w(i,k).states)$.

The difference between the definitions of a non-uniform and a uniform transformation concerns properties (iii), (iv), (vi) and (vii) in the non-uniform case, denoted respectively (iii'), (iv'), (vi') and (vii') in the uniform case. For a non-uniform transformation, the corresponding properties must be satisfied by the processes that are correct in *the underlying run* R_w of M_w, whereas for a uniform transformation, the corresponding properties must be satisfied by the processes that are correct in the *simulated run* R_s of M_s. These processes include those that are correct in R_w by Property (ii).

Property (i) states that the simulated run should be one of the runs of the simulated protocol. Property (ii) forces a correct process to be correct in the simulated run (though a faulty process may appear correct in the simulated run). Properties (iii) and (iii') state that the input pattern is preserved by the simulation. Properties (iv) and (iv') state that any simulated state is correct w.r.t. Π_s. Property (v) forces the simulation to accomplish progress. Properties (vi) and (vi') state that each state of Π_s is simulated in at most one manner. Properties (vii) and (vii') force a process to simulate states sequentially w.r.t. Π_s.

With a non-uniform transformation, any process p_i that is faulty in any run R_w of M_w is not required, according to Definition 2, to make any progress nor to guarantee any property on the states that p_i simulates. In particular, if the underlying model M_w permits it, e.g. M_w is *Byzantine*, p_i may behave arbitrarily and may simulate states that are not consistent in M_s with the simulation of the correct processes in R_w. Nevertheless Property (i) ensures the correctness of the simulation w.r.t. M_s at all times. Roughly speaking, all the processes correct in R_w maintain a simulated state for any process faulty in R_w, that is consistent in M_s. In Section 5 we will see that we indeed define the simulation function *Sim* through the processes that are correct in R_w for a non-uniform transformation.

Apart from Property (iii), our definition encompasses the notion of simulation of [17], although the notion of input pattern does not appear in [17]. In the transformation of [17] from *Crash* to *Omission*, each round is transformed in two phases, which can be defined with $f(r) = 2r$, and $c = 2r$ in (v). This implies that $ST_w(i, 2x).states = \{x\}$ and $ST_w(i, 2x+1).states = \emptyset$.

In the following definition, we recall the notion of *effectively solving* [17] a problem, to indicate that the resolution is obtained through a simulation function.

Definition 4 *For any given function Sim,* Π_w effectively solves *problem* Σ *in model* M_w *with input pattern in* $\Gamma_{\mathcal{I}}$ *if and only if, for any run* $R \in R(\Pi_w, M_w, \Gamma_{\mathcal{I}})$, $Sim(R)$ *satisfies* Σ.

The next proposition follows from definitions 2, 3 and 4.

Proposition 5 *Let* Π_s *be any protocol that solves specification* Σ *in model* M_s. *If* $\mathcal{T}$ *is a transformation from* M_s *to* M_w *and Sim the corresponding simulation function, then protocol* $\mathcal{T}(\Pi_s)$ *effectively solves* Σ *in model* M_w.

4 Interactive Consistency Algorithms

We consider in this section the *Interactive Consistency* (IC) problem [18] that is solved to simulate a single round in our transformation. Roughly speaking, in the IC problem, each process p_i is supposed to propose an initial value and eventually decide on a vector of values.

We use two specifications of Interactice Consistency: a uniform specification and a non-uniform specification. Non-uniform Interactive Consistency is the original problem as defined in [18].

In uniform IC, each process p_i is supposed to propose a value v_i and eventually decide on a vector of values V_i such that

Termination: every correct process eventually decides,
Uniform agreement: no two decided vectors differ,
Validity: for any decision vector V, the j^{th} component of V is either the value proposed by p_j or $\perp$, and is $\perp$ only if p_j fails.

In non-uniform IC, each correct process p_i is supposed propose a value v_i and eventually decide on a vector of values V_i such that

Termination: (similar as for uniform IC) every correct process eventually decides,
Agreement: no two vectors decided by *correct* processes differ,
Validity: for any vector V decided by a *correct* process, the j^{th} component of V is either the value proposed by p_j or $\perp$, and is $\perp$ only if p_j fails.

To be self-contained, this paper presents several IC algorithms:

- for non-uniform IC, the paper presents, in Fig. 1, an early stopping Interactive Consistency algorithm in *General*, *Omission* and *Crash*, for all three models with $t < n$. In this algorithm, and for any run R of *General*, *Omission* or *Crash*, all the correct processes decide by the end of round $f+1$ in run R in which at most $f \leq t$ processes fail (this is a tight lower bound due to [14]), and halt by the end of round $min(f+2, t+1)$ in run R (this is a tight lower bound due to [5]). In any failure-free run ($f = 0$), all the processes are correct and decide by the end of round 1 and halt by the end of round 2.
- for uniform IC, the paper presents two algorithms, respectively for *Omission* and *Crash* ($t < n$) in Fig. 2 and for *General-MAJ* ($t < n/2$) in Fig. 3. In both algorithms, and for any run R of *Omission* or *Crash*, resp. *General-MAJ*, all the correct processes decide and halt at the end of round $t+1$ in run R (this is a lower bound due to [7]).

In *Byzantine*, we refer the reader to the following algorithms from the literature solving non-uniform IC: [18] (with exponentially-growing message size), or [9] (with polynomially-growing message size). It is not possible to solve uniform IC in *General* ($t < n$) [17,19] and in *Byzantine* [18].

4.1 Non-uniform Interactive Consistency

We give in Fig. 1 an algorithm solving non-uniform IC in *General*, *Omission* and *Crash*, for $t < n$.

The algorithm is derived from the *Terminating Reliable Broadcast* algorithm of [20]. We briefly describe how the algorithm works. In this description, we focus on the value of process p_i, the same mechanism extends to the value of the other processes.

In round 1, p_i sends its value to all the processes. In later rounds, every process relays p_i's value by sending its complete vector of values to all the processes. In parallel, each process p_j maintains a set *quiet* with the processes from which p_j does not receive a message in some round. When receiving a vector of values, p_i copies from this vector the values that p_i does not have in its own vector of values.

At process p_i:

$quiet_i := \emptyset$
$V_i := [\top, \ldots, v_i, \ldots, \top]$

for r from 1 to $t+1$ **do** $\quad$ *{For each round r}*
$\quad$ send V_i to all processes $\quad$ *{Send phase}*
$\quad$ **if** $(\forall k : V_i[k] \neq \top)$ **then** halt

$\quad$ $\forall p_j \in \Omega \backslash quiet_i$: **if** receive V_j **then** $\quad$ *{Receive phase}*
$\quad\quad\quad$ $\forall k$: **if** $(V_i[k] = \top)$ **then** $V_i[k] := V_j[k]$
$\quad\quad$ **else**
$\quad\quad\quad$ $quiet_i := quiet_i \cup \{p_j\}$

$\quad$ **if** $|quiet_i| < r$ **then**
$\quad\quad$ $\forall k$: **if** $V_i[k] = \top$ **then** $V_i[k] := \bot$
$\quad$ **if** $(\forall k : V_i[k] \neq \top)$ **then** decide(V_i)

$\forall k$: **if** $V_i[k] = \top$ **then** $V_i[k] := \bot$ $\quad$ *{If not decided in round $t+1$}*
decide(V_i)
halt

Fig. 1. Non-uniform Interactive Consistency in *General* $(t < n)$

At the end of any round r, if p_i did not receive a message from strictly less than r processes, then p_i fills the missing values in its vector of value with $\bot$, meaning that the process at this position is faulty. Process p_i decides on its vector of values at the end of any round when its vector is filled for each position, either with a value or with $\bot$. After sending its vector in the next round, p_i may halt. If after $t+1$ rounds, p_i still has not decided, p_i fills the missing values in its vector of values with $\bot$ and decides on its vector before halting.

4.2 Uniform Interactive Consistency in Omission and Crash

The algorithm in Figure 2 solves IC in *Omission* and *Crash*. This algorithm is given in [11]. In both models we assume $t < n$. In the algorithm, all the processes that decide, decide after $t+1$ rounds. We briefly explain how the algorithm works.

In the algorithm in Figure 2, any process p_i sends its value v_i to all the processes in round 1, v_i being embedded in the vector V_i. The value v_i is then relayed to all processes by another process in each round, until round $t+1$. When any process relays a estimate value, it sends its vector of estimate values to all the processes. For any process p_i, the value of p_i is relayed successively

At process p_i:
1: $V_i := [\bot, \ldots, v_i, \ldots, \bot]$
2: **for** r from 1 to $t+1$ **do**
3: send V_i to all processes
4: $\forall p_j \in \Omega$: **if** receive V_j **do**
5: $V_i[(j-r+1)\ mod\ n] := V_j[(j-r+1)\ mod\ n]$
6: decide(V_i)

Fig. 2. Interactive Consistency in *Omission* ($t < n$)

by the processes $p_i, \ldots, p_{(i+t)\ mod\ n}$, respectively in round 1 to $t+1$.

In the algorithm, the relaying mechanism is hidden in the reception phase. More precisely, when any process p_k receives a vector of estimate values in round r from p_j, p_k only copies into its own vector of estimate values, the component that p_k is supposed to relay in that particular round r. Process p_k does not make use of any other estimate value from p_j's vector.

In any particular round r, any process p_k relays the value of a different process (i.e., the value of process $(k-r+1)\ mod\ n$); whereas the value of any process p_i is relayed by a different process (i.e., process $(i+r-1)\ mod\ n$).

We now give an intuition of the correctness of the algorithm. The intuition is particularily simple: as $t+1$ processes are involved in relaying the value of any process p_i, at least one of them is correct, say p_j, and thus never commits omissions. When p_j relays the estimate value of p_i's value to all the processes, as p_j is correct, all processes receive p_j's estimate value. From this round on, all processes maintain the same estimate value for p_i's value in their vector of estimate values.

4.3 Uniform Interactive Consistency in General-MAJ

The algorithm in Figure 3 solves IC in *General-MAJ*. The algorithm is inspired from the uniform consensus algorithm of [6]. In *General-MAJ*, we assume $t < n/2$, as [17,19] that this is necessary for the problem to be solvable. In the algorithm, all the processes that decide, decide after $t+1$ rounds. We briefly explain how the algorithm works.

Primarily, the processes exchange vectors of estimate values, and update their own vector with the vectors received in each round. In the absence of omission, this procedure ensures that each process decides on the same vector of estimate values at the end of round $t+1$. To tolerate general omission, we do not allow some faulty processes (those with insufficent information) to decide, at the end of round $t+1$.

```
At process p_i:
halt_i := ∅
suspect_i := ∅
V_i := [⊥, ..., v_i, ..., ⊥]
for r from 1 to t + 1 do
    send [V_i, halt_i] to all processes
    ∀p_j ∈ Ω\halt_i: if receive [V_j, halt_j] then
                        if p_i ∈ halt_j then
                            suspect_i := suspect_i ∪ {p_j}
                     else
                            halt_i := halt_i ∪ {p_j}
    for all p_j ∈ Ω\halt_i and for all k do
        if V_i[k] = ⊥ then V_i[k] := V_j[k]
if |halt_i ∪ suspect_i| ≤ t then
    decide(V_i)
```

Fig. 3. Interactive Consistency in *General* $(t < n/2)$

Similary as in the algorithm for *Omission*, any process p_i maintains a set $halt_i$ with the identity of processes from which p_i does not receive a message in this round or in a previous round. Moreover, any process p_i maintains in addition a set $suspect_i$ with the identity of any process p_x which includes p_i's identity in its set $halt_x$. p_i maintains the vector of estimate values V_i, and decides on this vector at the end of round $t + 1$.

5 Shifting Transformation

We present our algorithm to transform any protocol Π written in *PSR* into a protocol Π' in a weaker model M_w such that Π' simulates Π, through a simulation function *Sim* that we give.

For any two distinct processes p_i and p_j simulating protocol Π, we do not necessarily assume that $\Pi_i = \Pi_j$. However, we will assume for the time being that p_i knows the state machine $\Pi_j = \langle s_j, T_j, O_j \rangle$ executed by p_j. We relax this assumption in Section 6.

Our transformation works on a round basis: it transforms a single round in *PSR* into several phases in M_w. The key to its efficiency is that a phase is involved in the simulation of more than one round simultaneously. We start by giving a general definition of the notion of shifting transformation, before giving our own.

Our algorithm implements at the same time a non-uniform and a uniform transformation. Roughly speaking, the transformation algorithm relies on an underlying Interactive Consistency algorithm, which alone determines if the overall transformation algorithm is uniform or non-uniform. More precisely, using a non-uniform, resp. uniform, IC algorithm as the underlying IC algorithm in the transformation leads to a transformation that is also non-uniform, resp. uniform. In the rest of this section, we thus present a single transformation algorithm that uses an underlying IC algorithm that may either be uniform or non-uniform, depending on the transformation required.

From this observation, it is thus clear that the uniform transformation may work only in *Crash*, *Omission* and *General-MAJ* ($t < n/2$), and not in *General* ($t < n$) or *Byzantine*, since it is not possible to implement uniform IC in both models [17,19]. The non-uniform transformation works for all models, i.e., transformation may work only in *Crash*, *Omission* and *General-MAJ* ($t < n/2$), *General* ($t < n$) and *Byzantine*.

Let Π_s be any protocol in model M_s, $\mathcal{T}$ any transformation (uniform or not) from M_s to M_w, and $\Pi_w = \mathcal{T}(\Pi_s)$ the transformed protocol. Roughly speaking, a shifting transformation is such that any process simulates *round* r of Π_s after a bounded number of phases counting from *phase* r. More precisely:

Definition 6 *A non-uniform (resp. uniform) transformation $\mathcal{T}$ from model M_s to model M_w is a non-uniform (resp. uniform)* shifting transformation *if and only if there exists a constant $S \in \mathbb{N}$, such that, for all $r \in \mathcal{R}$, $f(r) = r+S$. We call S the* shift *of the transformation.*

5.1 Algorithm

In our transformation, all processes collaborate to reconstruct the failure and input patterns of a run in *PSR*. They accomplish both reconstructions in parallel, one round after another. When processes terminate the reconstruction of the patterns for a round, they locally execute one step of the simulated protocol. If a process realizes that it is faulty in the simulated failure pattern, this process simulates a crash in *PSR*. To simulate one round in *PSR*, processes solve exactly one instance of the Interactive Consistency problem. In the instance of IC, each process p_i proposes its own input value for the round. The decision vector corresponds at the same time to a round of the failure pattern, and of the input pattern.

Figure 4 gives the transformed protocol $\mathcal{T}(\Pi_s)$ for process p_i, in terms of Π_s and the input pattern I. The underlying IC algorithm may either be uniform or non-uniform, for the transformation to be uniform or non-uniform. When using the early-stopping non-uniform IC algorithm in Fig. 1 for a transformation into *Crash*, *Omission*, or *General*, with $t < n$ for all three models, each round of *PSR* is transformed into $f + 1$ phases of the weaker model, in any run R

```
1: failure := ∅                                  {failure corresponds to one round of the failure pattern}
2: simulatedRound := 1                           {simulatedRound is the current simulated round number}
3: (∀j ∈ [1, n])(simst(0)[j] := s_j)             {state of protocol Π_s for p_j at the end of round 0}
4: states := {0}     {set of rounds of protocol Π_s simulated by Π_w in the current round}
5: ss[0] := s_i      {set of states of protocol Π_s simulated by Π_w in the current round}
6: for phase r (r = 1, 2, ...) do
7:    input := receiveInput()                    {receive input value corresponding to I(i, r)}
8:    start IC instance number r, and propose(input)
9:    execute one round of all pending IC instances
10:   states := ∅                                {has any IC instance decided?}
11:   while simulatedRound-th instance of IC has decided do
12:      states := states ∪ {simulatedRound}     {instance simulatedRound has decided}
13:      decision := decision vector of instance simulatedRound      {reconstruct patterns}
14:      for each p_j ∈ Ω do                     {check input against byzantine failures}
15:         if decision[j] ≠ ⊥ and decision[j] ∉ I then
16:            decision[j] := ⊥
17:      failure := failure ∪ { p_j | decision[j] = ⊥ }             {ensure atomic failures only}
18:      if p_i ∈ failure then halt                                  {is process p_i faulty?}
19:      for each p_j ∈ Ω do           {adjust decision vector with previous failure pattern}
20:         if p_j ∈ failure then
21:            decision[j] := ⊥
22:      for each p_j ∈ Ω do                                         {generate messages}
23:         if p_j ∉ failure then
24:            rcvd[j] := O_j(simst(simulatedRound − 1)[j], decision[j], simulatedRound)
25:         else
26:            rcvd[j] := ⊥
27:      for each p_j ∈ Ω do                                         {perform state transitions}
28:         if p_j ∉ failure then
29:            simst(simulatedRound)[j] := T_j(simst(simulatedRound − 1)[j], rcvd, simulatedRound)
30:         else
31:            simst(simulatedRound)[j] := simst(simulatedRound − 1)[j]
32:      ss[simulatedRound] := simst(simulatedRound)[i]
33:      simulatedRound := simulatedRound + 1    {increment simulated round counter}
34:   if r − simulatedRound ≥ δ then halt                            {is process p_i faulty?}
```

Fig. 4. Transformation algorithm (code for process p_i)

with at most $f \leq t$ failures. In any failure-free ($f = 0$) run, we observe that the transformation of one round of *PSR* requires a single phase of the weaker mode. In this sense, the simulation outputs the results in real time.

When using a uniform IC algorithm, e.g., the algorithm in Fig. 2 when the weaker model is *Crash* or *Omission* with $t < n$ or the algorithm in Fig. 3 when the weaker model is *General-MAJ* with $t < n/2$, each round of *PSR* is transformed into $t + 1$ phases of the weaker model, in any run R.

For the sake of simplicity, the transformation algorithm is given in an operational manner (i.e., pseudo-code). During any phase, many IC instances might be running together. If the condition of the **while loop** at line 11 ("*simulatedRound-th instance of IC has decided*") is true in a phase x of process p_i, then we denote by $decision_i(simulatedRound)$ the decision vector for the instance of IC in line 13, $failure_i(simulatedRound)$ the value of the variable *failure* updated in line 17, $rcvd_i(simulatedRound)$ the value of the variable *rcvd* updated in lines 24 or 26, and $simst_i(simulatedRound)$ the value of the variable *simst* updated in lines 29 or 31. The following proposition defines the simulation function *Sim* in our transformation.

The next proposition gives the construction of the simulation function *Sim* for both the uniform and non-uniform transformation.

Proposition 7 *The simulation Sim for a run of $\mathcal{T}(\Pi_s)$, $R = \langle I, ST, MS, MR \rangle$, is defined by $R' = \langle I', ST', MS', MR' \rangle$ as follows. Let p_i be a process in $correct(R)$. We consider the simulation of round r of R', for any process p_j.*

(i) $I'(j, r)$ *is the value* $decision_i(r)[j]$ *of the r-th instance of IC.*

(ii) *if* $r = 0$ *then* $ST'(j, 0) = s_j$*, otherwise* $ST'(j, r) = simst_i(r)[j]$.

(iii) *if* $p_j \in failure_i(r)$ *then* $MS'(j, k, r) = \perp$*, otherwise* $MS'(j, k, r) = rcvd_i(r)[j]$ *for any process* $p_k \in \Omega$.

(iv) *if* $p_j \in failure_i(r)$ *then* $MR'(j, r) = [\perp, \ldots, \perp]$*, otherwise* $MR'(j, r) = rcvd_i(r)$.

The next propositions assert the correctness of the non-uniform, resp. uniform, transformation.

Proposition 8 *The algorithm of Fig. 4 (used in conjonction with an underlying non-uniform IC algorithm) is a non-uniform shifting transformation from $PSR(n, t)$ to $Crash(n, t)$, $Omission(n, t)$, $General(n, t)$ $(t < n)$, or $Byzantine(n, t)$ $(t < n/3)$ where the shift S is the number of rounds needed to solve non-uniform Interactive Consistency in $Crash(n, t)$, $Omission(n, t)$, $General(n, t)$ $(t < n)$, or $Byzantine(n, t)$ $(t < n/3)$.*

Proposition 9 *The algorithm of Fig. 4 (used in conjonction with an underlying uniform IC algorithm) is a uniform shifting transformation from $PSR(n, t)$ to $Crash(n, t)$, $Omission(n, t)$ $(t < n)$, or $General\text{-}MAJ(n, t)$ $(t < n/2)$ where the shift S is the number of rounds needed to solve uniform Interactive Consistency in $Crash(n, t)$, $Omission(n, t)$ $(t < n)$, $General\text{-}MAJ(n, t)$ $(t < n/2)$.*

In this section we prove Proposition 9, but Proposition 8 would be proved in the exact same manner.

To prove Proposition 9, we first show that the construction of function *Sim* in Fig. 4 is consistent with Proposition 7. We proceed through a series of lemmas.

Lemma 10 *For any run R and any process p_i in $correct(R)$, p_i never halts, and decides in all IC instances.*

Proof: A process p_i that is correct in R exists since $t < n$ for *Crash*(n, t), *Omission*(n, t) or *General-MAJ*(n, t). Thus, p_i does not halt in lines 18 or 34, as $p_i \in$ *correct*(R))Process p_i is correct and never crashes. By the termination property of IC, p_i always decides in any instance of IC. Thus p_i never halts in line 34. Moreover, by the validity property of IC, in any *decision* vector, *decision*$[i] \neq \perp$. Thus p_i never halts in line 18. □

Lemma 11 *For any run R, any process p_i in correct(R), any process p_j, and any round r such that p_j decides in the r-th instance of IC, we have the following properties:*

- $decision_i(r) = decision_j(r)$
- $failure_i(r) = failure_j(r)$
- $rcvd_i(r) = rcvd_j(r)$
- $simst_i(r) = simst_j(r)$

Proof: By Lemma 10, p_i decides in all IC instances. By the agreement property of IC, the decision is the same for p_i and p_j. In the algorithm, p_j decides in the r-th instance of IC if and only if p_j has decided in all previous instances. We show the three last items by induction on r. Initially, because of initialization, the properties are true for $r = 0$. Assume the properties are true up to round $r - 1$. When p_i decides in the r-th instance, it adds a set of processes to $failure_i$. By the agreement property of IC, p_i and p_j add the same set. By induction, $failure_i(r) = failure_j(r)$. As *decision*$(r)$ and *failure*(r) are the same for all processes for which they are defined, by induction hypothesis, we have $rcvd_i(r) = rcvd_j(r)$ and $simst_i(r) = simst_j(r)$. □

We can define the simulation through the value of the variables of any correct process. Consider any run R and let p_k be a correct process in R (we know there exists at least one as $t < n$). We define the simulation through the variables of p_k.

Lemma 12 *If there exists a round r and a process p_i such that $i \in failure_k(r)$ then, for any $r' \geq r$, $simst_k(r')[i] = simst_k(r)[i]$.*

Proof: Directly from the transformation algorithm. □

Lemma 13 *For any process p_i, any round r such that $decision_i(r)$ and $decision_i(r+1)$ occur, $failure_i(r) \subseteq failure_i(r+1)$.*

Proof: From the transformation algorithm, $failure_i$ always increases. □

Lemma 14 *correct$(R) \subseteq$ correct(R').*

Proof: By Lemma 10, all correct processes decide in all instances of IC. By termination of IC, no correct processes ever halt in line 34. By the validity property of IC, the decision value is not $\perp$ for any correct process in any decision vector. Thus no correct process ever halts in line 18. Therefore all correct processes in R are correct in R'. $\square$

Lemma 15 *For any process p_i and any protocol Π_s to simulate, let $\langle s_i, T_i, O_i\rangle$ be the state machine for p_i. For any round r and any $p_j \in \Omega$, we have:*

(1) $ST'(i, 0) = s_i$.
(2) *if* $p_i \in correct(R', r)$ *then* $MS'(i, j, r) = O_i(ST'(i, r-1), I'(i, r), r)$, *otherwise* $MS'(i, j, r) = \perp$.
(3) *if* $p_i \in correct(R', r)$ *then* $MR'(i, j, r) = MS'(j, i, r)$, *otherwise* $MR'(i, j, r) = \perp$.
(4) *if* $p_i \in correct(R', r)$ *then* $ST'(i, r) = T_i(ST'(i, r-1), [MS'(1, i, r), \ldots, MS'(n, i, r)], r)$, *otherwise* $ST'(i, r) = ST'(i, r-1)$.

Proof: (1) is immediate, from the initialization of the variable $simst_k[i](0)$. We prove (2), (3) and (4) by induction. For the case $r = 1$, and $p_i \notin correct(R', 1)$, then $p_i \in failure_k(1)$. By the algorithm $rcvd_k(1)[i] = \perp$, and by properties (iii) and (iv) of the definition of the simulation, $MS'(i, j, 1) = rcvd_k(1)[i]$ and $MR'(i, j, 1) = rcvd_k(1)[j]$. By Lemma 12 and (1), we have $ST'(i, 1) = ST'(i, 0)$. If $p_i \in correct(R', 1)$, then $p_i \notin failure_k(1)$. By the properties (iii) and (iv) of the definition of the simulation, $MS'(i, j, 1) = rcvd_k(1)[i]$ and $MR'(i, j, 1) = rcvd_k(1)[j]$. By line 24 of the algorithm, $rcvd_k(1)[i] = O_i(simst_k(0)[i], decision_k[i], 1)$, and so $MS'(i, j, 1) = O_i(ST'(i, 0), I'(i, 1), 1)$. By line 24 of the algorithm, $rcvd_k(1)[j] = MS'(j, k, 1)$, and so $MR'(i, j, 1) = MS'(j, i, 1)$. By line 29 of the algorithm, $simst_k(1)[i] = T_i(simst_k(0)[i], rcvd_k, 1)$, and $ST'(i, 1) = T_i(ST'(i, 0), [MS'(1, i, 1), \ldots, MS'(n, i, 1)], 1)$.

Assume the properties (2) and (4) are true up to round $r-1$. If $p_i \notin correct(R', r)$, then $p_i \in failure_k(r)$. By the properties (iii) and (iv) of the definition of the simulation, $MS'(i, j, r) = \perp$, and $MR'(i, r) = [\perp, \ldots, \perp]$. By Lemma 12 and (1), we have $ST'(i, r) = ST'(i, r-1)$. If $p_i \in correct(R', r)$, then by definition and by the transformation algorithm, $p_i \notin failure_k(r)$. By the properties (iii) and (iv) of the definition of the simulation, $MS'(i, j, r) = rcvd_k(r)[i]$ and $MR'(i, j, r) = rcvd_k(r)[j]$. By line 24 of the algorithm, $rcvd_k(r)[i] = O_i(simst_k(r-1)[i], decision_k[i], r)$, and so $MS'(i, j, r) = O_i(ST'(i, r-1), I'(i, r), r)$. By line 29 of the algorithm, $simst_k(r)[i] = T_i(simst_k(r-1)[i], rcvd_k, r)$, and so $ST'(i, r) = T_i(ST'(i, r-1), [MS'(1, i, r), \ldots, MS'(n, i, r)], r)$. $\square$

By Lemmas 11 and 15, the function *Sim* is well defined, and consistent with Proposition 7.

Lemma 16 *R' is a run of Π_s.*

Proof: Lemmas 11 and 13 show that R' is in *PSR* if R is in M, where $M \in \{$*Crash*, *Omission*, *General-MAJ*$)$. Lemma 15 shows that the functions ST', MS' and MR' consistently define a run of Π_s, with input value I'. □

Lemma 17 *Let x be any phase, p_i any process and r any round. If $r \in ST(i,x).states$ then $ST(i,x).ss[r] = ST'(i,r)$.*

Proof: Each time a round r is added to $states_i$ (line 12), $ss_i[r] = simst_i(r)[i]$ (line 32). By Lemma 11 and Proposition 7, $simst_i(r)[i] = ST'(i,r)$. □

Lemma 18 *For any round r and any process p_i in correct(R'), if it exists c such that $r \in ST(i,c).states$ then $ST(i,c).ss[r] = ST'(i,r)$.*

Proof: If such a c exists, then p_i has decided in the r-th instance of IC, and by the algorithm in Figure 4, in each previous instance. By Lemma 17, we have $ST(i,c).ss[r] = ST'(i,r)$. □

Lemma 19 *Let p_i be any process in correct(R') and r any round. There exists a unique $c \leq r+S$ such that $r \in ST(i,c).states$ and $ST(i,c).ss[r] = ST'(i,r)$.*

Proof: By Lemma 10, p_i decides in all instance of IC. In particular, for the r-th instance, p_i decides in phase c. We pose S as the number of phases for any instance of IC to decide, and thus $c \leq S+r$. We have $r \in ST(i,c).states$, and by Lemma 18, $ST(i,c).ss[r] = ST'(i,r)$. □

(End of proof of Proposition 9) We show that our function *Sim*, as defined by Proposition 7, satisfies the seven properties of Definition 3, with $f(r) = r+S$, where S is the number of phases to solve IC in M_w. Hence, $S = t+1$ for *Crash*, *Omission* and *General-MAJ*.

Lemma 16 shows that *Sim*(R) is a run of Π_s, which implies Property (i). Lemma 14 shows that *correct*$(R) \subseteq$ *correct*(R'), which implies Property (ii). By the definition of the simulation, we have $I = I'$, which implies Property (iii). Property (iv) follows from Lemma 17. By Lemma 10, any correct process p_i in R decides in any instance of IC. In particular, in the r-th instance of IC, it decides at most at phase $r+S$. Lemma 19 shows that for any round r and any process p_i in *correct*(R'), there exists $c \leq r+S$ such that $r \in ST(i,c).states$ and $ST(i,c).ss[r] = ST'(i,r)$ (Property (v)) and that this c is unique (Property (vi)). If a process adds any round r in its set *states*, by the algorithm, it has decided the r-th instance of IC, and all previous instances of IC as well. This implies Property (vii).

6 Transformation Extension

We give in this section an extension of the uniform transformation of Fig. 4. In the transformation of Fig. 4, the processes only need to send their input value in a phase, because the protocol itself can be locally simulated by other processes. We assume now that the processes do not know the state machine simulated by any other process. As a result, any process p_i needs to send, in addition to the message of the previous transformation, the content of the message it would normally send in the simulated protocol, i.e., the output of function O_i. Nevertheless, as with our previous transformation, we would like to start the simulation of a round before the decision of all previous simulations are known. Thus p_i cannot know in which *precise* state of the protocol it should be at the time it has to generate a message (remember that the current state is a parameter of the message output function O_i).

More precisely, consider any process p_i simulating a run $R' = \langle I', ST', MS', MR' \rangle$ of PSR. The idea of the extended transformation is to maintain, for p_i, all simulated states of ST' that are coherent with previous (terminated) simulations, and which only depend on the outcome of on-going (not yet terminated) simulations. Hereafter, these states are called the *extended set of states* and denoted by es. For any two processes p_i and p_j simulating the execution of protocol Π in PSR, we denote by m_j the message p_j sends to p_i in round r. Before the end of round r simulation, i.e., in any phase r' such that $r \leq r' \leq r + \delta - 2$ where δ is the number of phases for the r-th IC instance to decide, p_i does not know the decision value corresponding to p_j's proposal: (1) as long as p_i has not received m_j, the decided value can be any value in $\mathcal{M}'$ (including $\perp$), and (2) if p_i receives m_j, the decided value can either be m_j or $\perp$. To be able to start the next instance in the next phase, p_i generates a new extended set of states. To generate this set of states, p_i computes T_i on every state in the current set of states, with every possible combination of messages received in phase r (i.e., $\perp$ values are successively substituted by any value of $\mathcal{M}$, and any received value successively substituted with $\perp$). To each state in the extended set of states corresponds a message of Π_i to be sent in round r by p_i. These messages are gathered in a set, hereafter called the *extended message* and denoted by em.

For example, consider the case of the $Crash(3,2)$ model with $\mathcal{I} = \mathcal{M} = \{0,1\}$. After phase 1, process p_1 gathers the received values in the vector $[1\,0\,\perp]$. The possible combinations of messages are $[1\,0\,\perp]$, $[1\,0\,0]$, $[1\,0\,1]$, $[1\perp\perp]$, $[1\perp 0]$, $[1\perp 1]$, $[\perp 0 \perp]$, $[\perp 0\,0]$, $[\perp 0\,1]$, $[\perp\perp 0]$, and $[\perp\perp 1]$. Process p_1 generates the extended set of states by applying function T_1 on each combination of messages.

Figure 5 presents our extended transformation algorithm. For the sake of clarity, we ignore possible optimizations in this algorithm (e.g., any process can reduce the number of possible states as it receives more values from other processes). In Fig. 5, we denote by $\mathit{rcvd}[r]$ the messages of instance r received in phase r. We assume without loss of generality that any process sends in any phase of the underlying IC algorithm, the value it proposes to this instance.

Let p_i be any process simulating state machine $\Pi_i = \langle s_i, T_i, O_i \rangle$. We consider the transformation algorithm at the beginning of phase r.

6.1 Message generation

At the beginning of phase r, p_i receives an input value $\mathit{input} = I(i, r)$, and computes a new extended set of states es' and the corresponding extended message em. A tuple in em is of the form $\langle \mathit{num}(\mathit{st}), \mathit{rec}, \mathit{num}(\mathit{st}'), m \rangle$, and contains (i) the identifier $\mathit{num}(\mathit{st})$ of a possible state st of p_i at the beginning of round $r-1$, (ii) a combination rec of messages received by p_i in phase $r-1$, (iii) the identifier $\mathit{num}(\mathit{st}')$ of the state st' of p_i at the beginning of round r, such that $\mathit{st}' = T_i(\mathit{st}, \mathit{rec}, r-1)$, (iv) the message sent in round r, i.e., $m = O_i(\mathit{st}', I(i, r), r)$. For each state st in the current extended set of states es, and for any combination rec of messages (according to the extended messages of phase $r-1$), p_i computes the next state $\mathit{st}' = T_i(\mathit{st}, \mathit{rec}, r)$ (whenever p_i includes a new state st' in es', it associates a unique identifier $\mathit{num}(\mathit{st}')$ with st'), and the corresponding message $m = O_i(\mathit{st}', \mathit{input}, r)$. p_i sends em and the extended messages of other running IC instances in phase r.

6.2 Simulation

In the following, the variable *simulatedRound* denotes the next round to be simulated (we consider that the simulation has been performed up to round $\mathit{simulatedRound}-1$). Each process p_i maintains (1) the simulated state of machine Π_i at the end of round $\mathit{simulatedRound}-1$ (denoted by $\mathit{ss}[\mathit{simulatedRound}-1]$), and (2) the identifier associated with the state currently simulated at each process p_j, at the end of round $\mathit{simulatedRound}-1$, denoted by $\mathit{sim}[j]$.

If the condition of the **while loop** at line 22 ("*simulatedRound-th instance of IC has decided*") is true in a phase x at process p_i, then $\mathit{decision}_i(\mathit{simulatedRound})$ denotes the decided vector of messages at line 24, $\mathit{failure}_i(\mathit{simulatedRound})$ the value of the variable *failure* updated in line 25, and $\mathit{trueRcvd}_i(\mathit{simulatedRound})$ the value of the variable *trueRcvd* updated in line 39. Process p_i uses the decided vector $\mathit{decision}_i(\mathit{simulatedRound})$ to update the simulated state of machine Π_i, i.e., p_i adds *simulatedRound* in *states* and computes $\mathit{ss}[\mathit{simulatedRound}]$. More precisely,

(1) p_i computes the messages *trueRcvd*: *(1a)* if $p_j \in \mathit{failure}$ or $\mathit{decision}[j] = \bot$, then $\mathit{trueRcvd}[j] = \bot$, otherwise *(1b)* p_i searches for the tuple $\langle \mathit{sim}[j], M, *, * \rangle$ in the extended message of p_j (generated at phase *simulatedRound*), where M is the set of messages received in round $\mathit{simulatedRound} - 1$ (i.e., the previous value of *trueRcvd*). Let $\langle \mathit{sim}[j], M, s, m \rangle$ be this tuple. $\mathit{sim}[j]$ is updated with s and $\mathit{trueRcvd}[j]$ with m.
(2) p_i updates $\mathit{ss}[\mathit{simulatedRound}]$ with the state $T_i(\mathit{ss}[\mathit{simulatedRound} - 1],$-*trueRcvd*, *simulatedRound*).

If any value in the vector *decision* is $\bot$, then the corresponding process is added to *failure*. If any process adds itself to *failure*, it stops. In the algorithm, at line 43, δ denotes the maximum number of phases for the underlying IC algorithm to decide in the system model in which the transformation algorithm is running. Indeed, a process that does not decide in an IC instance in δ phases is faulty, and thus stops taking part to the simulation.

The following propositions assert the correctness of the extension of our transformation. We introduce here the extension for the uniform transformation, in *Crash* and *Omission* ($t < n$), and *General-MAJ* ($t < n/2$).

Proposition 20 *The simulation Sim for a run of* $\mathcal{T}(\Pi_s)$, $R = \langle I, ST, MS, MR \rangle$ *is defined by the run* $R' = \langle I', ST', MS', MR' \rangle$ *as follows. Let* p_i *be a process in* $\mathit{correct}(R)$. *We consider the simulation of round* r *of* R', *for any process* p_j.

(i) $I' = I$.

(ii) *if* $p_j \in \mathit{failure}_i(r)$ *then* $p_j \notin \mathit{correct}(R', r)$ *otherwise* $p_j \in \mathit{correct}(R', r)$.

(iii) $ST'(i, 0) = s_i$ *and* $ST'(i, r) = ss_i[r]$. *For any process* p_j *(including* p_i*) not in* $\mathit{failure}_i(r)$, $ST'(j, r)$ *is the state of* p_j *at the end of round* r, *such that* $ST'(j, 0) = s_j$ *and* $ST'(j, x) = T_j(ST'(j, x-1), \mathit{trueRcvd}_i(x), x)$, *for each* x *from* 1 *to* r. *Otherwise, for any* p_j *in* $\mathit{failure}_i(r)$, $ST'(j, r) = ST'(j, r-1)$.

(iv) *if* $p_j \in \mathit{failure}_i(r)$ *then* $MS'(j, k, r) = \perp$, *otherwise* $MS'(j, k, r) = \mathit{trueRcvd}_i(r)[j]$ *for any process* $p_k \in \Omega$.

(v) *if* $p_j \in \mathit{failure}_i(r)$ *then* $MR'(j, r) = [\perp, \ldots, \perp]$, *otherwise* $MR'(j, r) = \mathit{trueRcvd}_i(r)$.

Proposition 21 *The algorithm of Fig. 5 is a uniform shifting transformation (with an underlying uniform IC algorithm) from PSR*(n, t) *to Crash*(n, t), *Omission*(n, t) $(t < n)$, *General-MAJ*(n, t) $(t < n/2)$ *where the shift* S *is number of rounds needed to solve uniform Interactive Consistency in Crash*(n, t), *Omission*(n, t) $(t < n)$, *or General-MAJ*(n, t) $(t < n/2)$.

The same idea can be applied when input values can depend on the state of the processes, and there are finitely many possible input values (i.e., $|\mathcal{I}| < \infty$). Using the technique described above, a process anticipates on the different input values that it can receive, to start the next simulations. When the preceding simulations terminate, the input value that had correctly anticipated the state of the process is determined, and only the messages and states following from this input value are kept. The algorithm in Fig. 5 can easily be adapted to the case where input values depend on the state of processes. Note that in both of the above cases, the number of messages generated is very high.

To prove Proposition 21, we first show that the construction of function *Sim* in Fig. 5 is consistent with Proposition 20. We proceed through a series of lemmas.

Lemma 22 *For any run* R *and any process* p_i *in* $\mathit{correct}(R)$, p_i *never halts, and decides in all IC instances.*

```
1:  failure := ∅                                  {failure corresponds to one round of the failure pattern}
2:  simulatedRound := 1                           {simulatedRound is the current simulated round number}
3:  states := {0};  ss[0] := s_i                  {set of states of protocol Π_s which are simulated by Π_w in
    the current round}
4:  (∀j ∈ [1,n])(sim[j] := 0)                     {all processes start in the 0-th state}
5:  number := 1                                   {by convention num(ss[0]) = 0}

6:  for phase r (r = 1,2,...) do
7:    input := receiveInput()                     {receive input value corresponding to I(i,r)}
8:    if r = 1 then
9:      es := {s_i};  em := { ⟨−, −, 0, O_i(s_i, input, r)⟩ }
10:   else
11:     es' := ∅;  em := ∅
12:     for any possible combination rec of n messages of rcvd[r − 1] do
13:       for any possible state st of es do
14:         st' := T_i(st, rec, r);  num(st') := number;  number := number + 1
15:         es' := es' ∪ {st'}
16:         em := em ∪ { ⟨num(st), rec, num(st'), O_i(st', input, r)⟩ }
17:     es := es'

18:   start instance r, and propose(em)
19:   execute one phase of all other running instances
20:   rcvd[r] := extended messages of instance r

21:   states := ∅
22:   while simulatedRound-th instance of IC has decided do   {has any IC instance decided?}
23:     states := states ∪ {simulatedRound}       {instance simulatedRound has decided}
24:     decision := decision vector of instance simulatedRound   {reconstruct patterns}
25:     failure := failure ∪ { p_j | decision[j] = ⊥}   {ensure atomic failures}
26:     if p_i ∈ failure then                     {is process p_i faulty?}
27:       halt                                    {p_i does not perform any step}
28:     for each p_j ∈ Ω do                       {adjust decision vector with previous failure pattern}
29:       if p_j ∈ failure then
30:         decision[j] := ⊥
31:     for each p_j ∈ Ω do                       {compose the messages of the round}
32:       if p_j ∈ failure then
33:         tmpRcvd[j] := ⊥
34:       else
35:         if simulatedRound = 1 then
36:           tmpRcvd[j] := m such that ⟨∗, ∗, 0, m⟩ ∈ decision[j]
37:         else
38:           let k and m such that ⟨sim[j], trueRcvd, k, m⟩ ∈ decision[j]
39:           tmpRcvd[j] := m; sim[j] := k
40:     trueRcvd := tmpRcvd
41:     ss[simulatedRound] := T_i(ss[simulatedRound − 1], trueRcvd, simulatedRound)
42:     simulatedRound := simulatedRound + 1

43:   if r − simulatedRound ≥ δ then halt         {is process p_i faulty?}
```

Fig. 5. Extended transformation algorithm (code for process p_i)

Proof: A process p_i that is correct in R exists since $t < n$ for $\mathit{Crash}(n,t)$, $\mathit{Omission}(n,t)$ or $\mathit{General\text{-}MAJ}(n,t)$. Thus, p_i does not halt in lines 27 or 43.) Process p_i is correct and never crashes. By the termination property of IC, p_i always decides in any instance of IC within a bounded number of phases. Thus p_i never halts in line 43. Moreover, by the validity property of IC, in any *decision* vector, $\mathit{decision}[i] \neq \perp$. Thus p_i never halts in line 27. □

Lemma 23 *For any run R, any process p_i in correct(R), any process p_j, and any round r such that p_j decides in the r-th instance of IC, we have the following properties:*

- $decision_i(r) = decision_j(r)$
- $failure_i(r) = failure_j(r)$
- $trueRcvd_i(r) = trueRcvd_j(r)$
- $sim_i(r)[j] = num(ST'(j, r))$

Proof: By Lemma 22, p_i decides in all IC instances. In the algorithm, p_j decides in the r-th instance of IC if and only if p_j has decided in all previous instances. By the agreement property of IC, the decision is the same for p_i and p_j. We show the three last items by induction on r. It is easy to see that these properties hold for $r = 1$: for the first IC instance, every process proposes its initial value; when all processes terminates the first IC instance, they decide upon the same vector. This implies $decision_i(1) = decision_j(1)$, $failure_i(1) = failure_j(1)$, $trueRcvd_i(1) = trueRcvd_j(1)$, and $sim_i(1)[j] = num(ST'(j, 1))$, for any $p_i \neq p_j$. Assume the properties hold up to round $r - 1$. When p_i decides in the r-th instance, it adds a set of processes to $failure_i$. By the agreement property of IC, p_i and p_j add the same set of processes. By induction, $failure_i(r) = failure_j(r)$. As $decision(r)$ and $failure(r)$ are the same for all processes for which they are defined, by induction hypothesis, we have $trueRcvd_i(r) = trueRcvd_j(r)$, and $sim_i(r)[j] = num(ST'(j, r))$. □

We may once again define the simulation through the value of the variables of any correct process. Consider any run R. In the following, p_k denotes a correct process in R (we know there exists at least one as $t < n$). We define some parts of the simulation through the variables of p_k. Note that, in contrast with the proof of Proposition 9, we cannot prove Proposition 21 only through the variables of p_k, because in the algorithm of Fig. 5, for instance, p_k does not keep the states in which the other processes are.

Lemma 24 *For any process p_i, any round r such that $decision_i(r)$ and $decision_i(r+1)$ occur, $failure_i(r) \subseteq failure_i(r + 1)$.*

Proof: From the algorithm, $failure_i$ always increases. □

Lemma 25 *$correct(R) \subseteq correct(R')$.*

Proof: By Lemma 22, all correct processes decide in all instances of IC. Thus they never halt in line 43. By the validity property of the underlying IC algorithm, the decision value is not $\perp$ for any correct process in any decision vector. Thus no correct process ever halts in line 27. Therefore all correct processes in R are correct in R'. □

Lemma 26 *For any process p_i and any protocol Π_s to simulate, let $\langle s_i, T_i, O_i \rangle$ be the state machine for p_i. For any round r and any $p_j \in \Omega$, we have:*

(1) $ST(i, 0) = s_i$.
(2) *if* $p_i \in \mathit{correct}(R', r)$ *then* $MS'(i, j, r) = O_i(ST'(i, r-1), I'(i, r), r)$, *otherwise* $MS'(i, j, r) = \perp$.
(3) *if* $p_i \in \mathit{correct}(R', r)$ *then* $MR'(i, j, r) = MS'(j, i, r)$, *otherwise* $MR'(i, j, r) = \perp$.
(4) *if* $p_i \in \mathit{correct}(R', r)$ *then* $ST'(i, r) = T_i(ST'(i, r-1), [MS'(1, i, r), \ldots, MS'(n, i, r)], r)$, *otherwise* $ST'(i, r) = ST'(i, r-1)$.

Proof: (1) is immediate, by the definition of ST' in Proposition 20. We prove (2) and (3) by induction. For the case $r = 1$ and $p_i \notin \mathit{correct}(R', 1)$, then $p_i \in \mathit{failure}_k(1)$. By the algorithm $\mathit{trueRcvd}_k(1)[i] = \perp$, and by properties (iii) and (iv) of the definition of the simulation, $MS'(i, j, 1) = \mathit{trueRcvd}_k(1)[i]$ and $MR'(i, j, 1) = \mathit{trueRcvd}_k(1)[j]$. By the algorithm and (1), $ST'(i, 1) = ST'(i, 0)$. If $p_i \in \mathit{correct}(R', 1)$, then $p_i \notin \mathit{failure}_k(1)$. By the properties (iii) and (iv) of the definition of the simulation, $MS'(i, j, 1) = \mathit{trueRcvd}_k(1)[i]$ and $MR'(i, j, 1) = \mathit{trueRcvd}_k(1)[j]$. By line 39 of the algorithm, $\mathit{trueRcvd}_k(1)[i] = O_i(ST'(i, 0), \mathit{decision}_k[i], 1)$, and so $MS'(i, j, 1) = O_i(ST'(i, 0), I'(i, 1), 1)$. By line 39 of the algorithm, $\mathit{trueRcvd}_k(1)[j] = MS'(j, k, 1)$, and so $MR'(i, j, 1) = MS'(j, i, 1)$. By line 41 of the algorithm, $ss_k(1)[k] = T_k(ss_k(0)[k], \mathit{trueRcvd}_k(1), 1)$, and $ST'(k, 1) = T_k(ST'(k, 0), [MS'(1, k, 1), \ldots, MS'(n, k, 1)], 1)$.

Assume the properties (2) and (3) are true up to round $r-1$. If $p_i \notin \mathit{correct}(R', r)$, then $p_i \in \mathit{failure}_k(r)$. By the properties (iii) and (iv) of the definition of the simulation, $MS'(i, j, r) = \perp$, and $MR'(i, r) = [\perp, \ldots, \perp]$. By the algorithm and (1) $ST'(i, r) = ST'(i, r-1)$. If $p_i \in \mathit{correct}(R', r)$, then $p_i \notin \mathit{failure}_k(r)$. By the properties (iii) and (iv) of the definition of the simulation, $MS'(i, j, r) = \mathit{trueRcvd}_k(r)[i]$ and $MR'(i, j, r) = \mathit{trueRcvd}_k(r)[j]$. By line 39 of the algorithm, and by the induction hypothesis, $\mathit{trueRcvd}_k(r)[i] = O_i(ST'(i, r-1), \mathit{decision}_k[i], r)$, and so $MS'(i, j, r) = O_i(ST'(i, r-1), I'(i, r), r)$. By line 39 of the algorithm, $\mathit{trueRcvd}_k(r)[j] = MS'(j, k, r)$, and so $MR'(i, j, r) = MS'(j, i, r)$. By line 41 of the algorithm, and by the induction hypothesis, $ss_k(r)[k] = T_k(ST'(k, r-1), \mathit{trueRcvd}_k, r)$, and so $ST'(k, r) = T_k(ST'(k, r-1), [MS'(1, k, r), \ldots, MS'(n, k, r)], r)$. □

By Lemmas 23 and 26, the function *Sim* is well defined, and consistent with Proposition 20.

Lemma 27 *R' is a run of Π_s.*

Proof: Lemmas 23 and 24 show that R' is in *PSR* if R is in M, where $M \in \{\mathit{Crash}, \mathit{Omission}, \mathit{General}\}$. Lemma 26 shows that the functions ST', MS' and MR' consistently define a run of Π_s, with input value I'. □

Lemma 28 *Let x be any phase, p_i any process and r any round. If $r \in ST(i,x).states$ then $ST(i,x).ss[r] = ST'(i,r)$.*

Proof: Each time a round r is added to $states_i$ (line 23), $ss_i[r] = T_i(ST'(i, r-1), trueRcvd_i(r), r)$ (line 29). □

Lemma 29 *For any round r and any process p_i in $correct(R')$, if it exists c such that $r \in ST(i,c).states$ then $ST(i,c).ss[r] = ST'(i,r)$.*

Proof: If such a c exists, then p_i has decided in the r-th instance of IC, and by the algorithm in Figure 5, in each previous instance. By Lemma 25, we have $ST(i,c).ss[r] = ST'(i,r)$. □

Lemma 30 *Let p_i be any process in $correct(R')$ and r be any round. There exists a unique $c \leq r + S$ such that $r \in ST(i,c).states$ and $ST(i,c).ss[r] = ST'(i,r)$.*

Proof: By Lemma 10, p_i decides in all instance of IC. In particular, for the r-th instance, p_i decides in phase c. We pose S as the number of phases for an instance of IC to decide, thus $c \leq S + r$. We have $r \in ST(i,c).states$, and by Lemma 18, $ST(i,c).ss[r] = ST'(i,r)$. □

(End of proof of Proposition 21) We show that our function *Sim*, as defined by Proposition 20, satisfies the seven properties of Definition 3, with $f(r) = r + S$, where S is the number of phases to solve IC in M_w.

Lemma 27 shows that $sim(R)$ is a run of Π_s, which implies Property (i). Lemma 25 shows that $correct(R) \subseteq correct(R')$, which implies Property (ii). By the definition of the simulation, we have $I = I'$, which implies Property (iii). Property (iv) follows from Lemma 28. By Lemma 22, any correct process p_i of F decides in any instance of IC. In particular, in the r-th instance of IC, it decides at most at phase $r + S$. Lemma 30 shows that for any round r and any process p_i in $correct(R')$, there exists $c \leq r + S$ such that $r \in ST(i,c).states$ and $ST(i,c).ss[r] = ST'(i,r)$ (Property (v)) and that this c is unique (Property (vi)). If a process adds any round r in its set *states*, by the algorithm, it has decided the r-th instance of IC, and all previous instances of IC as well. This implies Property (vii).

7 Complexity

We analyze the performance of our transformation technique in terms of message and phase complexities. For the rest of this section, we need to make a distinction between the number of phases before any process decides in a non-uniform, resp. uniform, IC algorithm, and the number of phases before any process halts in the same algorithm.

From the IC algorithms presented in Section 4, we have the following results:

- For non-uniform IC, and for any of the models *Crash*, *Omission*, and *General*, the number of phases for all processes to decide is $\delta_{\text{non-uniform}} = f + 1$ in any run R with at most f failures, whereas the number of phases for all processes to halt is $\tau_{\text{non-uniform}} = min(f+2, t+1)$ in any run R with at most f failures.
- For uniform IC, and for any of the models *Crash*, *Omission*, and *General-MAJ*, the number of phases for all processes to decide and halt is $\delta_{\text{uniform}} = \tau_{\text{uniform}} = t + 1$, in any run R.

7.1 Message Complexity

In terms of messages, the first transformation generates at most a $nlog_2|\mathcal{I}|$-bit message per process, per phase, and per IC instance. As there are $\tau_{\text{non-uniform}}$ non-uniform, resp. τ_{uniform} uniform, IC instances running in parallel, any process sends a $n\tau log_2|\mathcal{I}|$-bit message in any phase, in any run, where τ is either $\tau_{\text{non-uniform}}$ or τ_{uniform}, whether we consider the non-uniform or the uniform transformation.

In the extended, uniform transformation, any process maintains at least $2^{n\tau_{\text{uniform}}}$ states for a round simulation. A state (tuple) is coded using $\sigma = 2log_2|\mathcal{S}| + (n+1)log_2|\mathcal{M}|$ bits. As there are τ_{uniform} instances of the uniform IC algorithm running in parallel, any process sends a $\tau_{\text{uniform}}n\sigma 2^{n\tau_{\text{uniform}}}$-bit message in any phase, in any run.

Determining the exact overhead in terms of message size complexity is an open issue, as is the tight lower bound on the message overhead for a automatic shifting transformation technique.

7.2 Phase Complexity

The *phase complexity overhead* is defined as the number of additional phases executed by the transformed protocol Π_w in M_w, compared with the original protocol Π_s in M_s.

In any of our shifting transformation algorithms, the simulation of consecutive rounds is overlapped, such that the simulation of two consecutive rounds start with an interval of a single phase. Thus, for any of our shifting transformation algorithm, the only phase complexity overhead is the number of phases for obtaining the outcome of the simulation for the first round, corresponding to $\delta_{\text{non-uniform}} - 1$ or $\delta_{\text{uniform}} - 1$, depending on the transformation considered.

We observe that for the non-uniform transformation with $\delta_{\text{non-uniform}} = f + 1$, the phase complexity overhead is just $\delta_{\text{non-uniform}} - 1 = f$. In any failure-free ($f = 0$) run R, there is thus no phase complexity overhead for the non-uniform transformation. This transformation provides the outcome of the simulation in real-time, as the run executes. Had we try to simulate a weaker model than *PSR* by using our non-uniform shifting transformation, we would not have gained any improvement in the phase complexity in failure-free runs.

8 Concluding Remarks

In this paper, we have concentrated on models *Crash*, *Omission*, *General* and *General-MAJ*, and *Byzantine* and have presented different shifting transformation techniques to translate protocols from the perfectly synchronous model *PSR* into each of these weaker models. We first presented a simple shifting transformation algorithm (in which each process knows that state machine executed by any other process) that allows for both a uniform and a non-uniform transformation, and we have then extended the uniform transformation to a more sophisticated transformation algorithm (in which any process does not know the state machine executed by any other process).

We show in the paper that the complexity of the transformation in terms of rounds is optimal, in two precise senses. First the round overhead to simulate a single IC instance is optimal since we need just a single *PSR*. Second, the non-uniform transformation provides real-time outputs of the simulation, and thus had we try to simulate a weaker model than *PSR* by using our shifting transformation, we would not have gained any improvement in the round complexity.

We leave open the question of the message complexity. We have characterized the message complexity obtained in our solution, but the optimal message complexity for a shifting transformation is open, as is the question of finding the corresponding shifting transformation.

References

[1] R. A. Bazzi and G. Neiger. Simplifying fault-tolerance: providing the abstraction of crash failures. *Journal of the ACM*, 48(3):499–554, 2001.

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

[3] B. Charron-Bost, R. Guerraoui, and A. Schiper. Synchronous system and perfect failure detector: Solvability and efficiency issues. In *Proceedings of the IEEE International Conference on Dependable Systems and Networks*, pages 523–532, 2000.

[4] D. Dolev, R. Reischuk, and H. R. Strong. 'Eventual' is earlier than 'Immediate'. In *Proceedings of the* 23^{rd} *IEEE Symposium on Foundations of Computer Science (FOCS'82)*, pages 196–203, 1982.

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

[6] P. Dutta. A short note on uniform consensus in message omission model. Technical Report EPFL/IC, EPFL, 2002.

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

[8] 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, 1985.

[9] J. Garay and Y. Moses. Fully polynomial byzantine agreement for $n > 3t$ processors in $t + 1$ rounds. *SIAM Journal of Computing (SICOMP)*, 27(1), 1998.

[10] R. Guerraoui. On the hardness of failure-sensitive agreement problems. *Information Processing Letters*, 79(2):99–104, 2001.

[11] R. Guerraoui, P. Kouznetsov, and B. Pochon. A note on set agreement with omission failures. *Electronic Notes in Theoretical Computing Science*, 81, 2003.

[12] V. Hadzilacos. Byzantine agreement under restricted types of failures (not telling the truth is different from telling lies). Technical Report 18-83, Department of Computer Science, Harvard University, 1983.

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

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

[15] L. Lamport, R. Shostak, and L. Pease. The Byzantine Generals problem. *ACM Transactions on Programming Languages and Systems*, 4(3):382–401, 1982.

[16] Y. Moses and M. R. Tuttle. Programming simultaneous actions using common knowledge. *Algorithmica*, 3(1):121–169, 1988.

[17] G. Neiger and S. Toueg. Automatically increasing the fault-tolerance of distributed algorithms. *Journal of Algorithms*, 11(3):374–419, 1990.

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

[19] M. Raynal. Consensus in synchronous systems: a concise guided tour. In *Proceedings of the 2002 Pacific Rim International Symposium on Dependable Computing (PRDC'02)*, 2002.

[20] M. C. Roşu. Early-stopping terminating reliable broadcast protocol for general-omission failures. In *Proceedings of the* 15^{th} *ACM Symposium on Principles of Distributed Computing (PODC'96)*, page 209, New York, NY, USA, 1996. ACM Press.

[21] F. B. Schneider. Replication management using the state machine approach. In S. Mullender, editor, *Distributed Systems*. Addison-Wesley, 1993.

A Correctness of the Interactive Consistency Algorithms

A.1 Non-Uniform Interactive Consistency in General

We give in Fig. 1 an algorithm solving non-uniform IC in *General*, *Omission* and *Crash*, for $t < n$. We prove in this section that the algorithm in Figure 1 satisfies the specification of non-uniform IC, through a series of lemmas (Lemmas 31 to 33).

Lemma 31 *In the algorithm in Figure 1, all the correct processes decide on a vector of values by round* $f+1$*, and halt by round* $min(f+2, t+1)$.

Proof: If $f = 0$, all the processes are correct. Any correct process p_i sends its vector V_i of values in round 1 with its value v_i (line 4), receives the vector from every other process at the end of round 1 (line 7), decides on its vector of values at the end of round 1 (line 12) and halts in round 2 (line 5).

Now assume $1 \leq f \leq t$. We distinguish two cases:

(1) Some correct process p_i decides on its vector V_i in round f. In this case, process p_i sends its vector V_i to all the processes in round $f+1$ (line 4), thus every correct process decides at the end of round $f+1$ (line 12) and halts by round $min(f+2, t+1)$ (line 5).

(2) No correct process decides on its vector in round f. Thus no correct process halts by round $f+1$. In all rounds $1 \leq k \leq f+1$, every correct process sends a message to all processes, in particular to all correct processes (line 4). Consider any correct process p_i at the end of round $f+1$. As all the correct processes send a message to all the processes in round $1, \ldots, f+1$, the set $quiet_i$ for p_i at the end of round $f+1$ contain faulty processes only. Thus, $quiet_i \leq f < f+1$ at the end of round $f+1$. By the algorithm, p_i fills the missing values in V_i with $\perp$ (line 11), then decides on its vector V_i at the end of round $f+1$ (line 12), and halts in round $min(f+2, t+1)$ (line 5).

□

Lemma 32 *In the algorithm in Figure 1, for any decision vector V, the j^{th} component of V is either the value proposed by p_j or $\perp$. For any vector V decided by any correct process, the j^{th} component of V is $\perp$ only if p_j fails.*

Proof: If any process p_i is correct, p_i sends its vector with its value v_i to all the processes in round 1 (line 4). Every correct process receives p_i's vector and copies the value v_i into its own vector (line 7). Thereafter, the value does not change anymore. □

Lemma 33 *In the algorithm in Figure 1, if any correct process decides on a vector of values V, then every correct process eventually decides on the same vector of values V.*

Proof: We proceed through a series of claims.

Claim 1 *Any process decides on at most one vector.*

Proof: From the algorithm in Fig. 1, it is clear that any process may decide at most on a single vector. □

Claim 2 *If some correct process decides on a vector V such that $V[k] = v_k \neq \perp$ for some k, then some correct process copies v_k (line 7 in Fig. 1) in its own vector before round $t+1$.*

Proof: Suppose a process decides on a vector V at the end of round $t+1$ such that $V[k] = v_k \neq \perp$ for some k. This means that there exist $t+1$ processes $q_1, \ldots, q_{t+1}$, such that q_i sends a vector Q_i in round i such that $Q_i[k] = v_k$. One of them, say q_k, is necessarily correct. If $k=1$, q_1 has $v_1 = v_k$ as its initial value, at the beginning of round 1. For $2 \leq k \leq t+1$, q_k copies v_k in round $k-1$. In both cases, q_k copies v_k into its own vector before round $t+1$. □

Claim 3 *If any correct process copies v_k into its vector V_i for some k, then* *no correct process ever sets $V_j[k] := \bot$.*

Proof: Let p_i be the first correct process that sets $V_i[k] := v_k$ into its vector V_i of values, for some k. By Claim 2, p_i copies v_k in some round l, $l < t+1$. This implies the existence of l processes $q_1, \ldots, q_l$, such that q_x sends a vector Q_x in round x such that $Q_x[k] = v_k$. We distinguish two subcases, namely rounds 1 to l, and rounds $l+1$ to $t+1$:

(1) Suppose now by contradiction that some processes set $V[k] := \bot$ in their vector of values, and denote p_j the first process that sets $V_j[k] := \bot$ in some round $m \leq l$. In any round i, $1 \leq i \leq m$, p_j does not receive the vector of values Q_i from q_i. Thus, at the end of round m, the set $|quiet_j|$ includes processes $q_1, \ldots, q_m$, and thus $|quiet_j| \geq m$ at the end of round m. As p_j is the first process that sets $V_j[k] := \bot$, p_j necessarily does it because $|quiet_j| < m$ at the end of round m — a contradiction.

(2) As p_i copies v_k in round l, p_i sends its vector of values V_i such that $V_i[k] = v_k$ in round $l+1$. From subcase 1 above, no process sends a vector of values V such that $V[k] = \bot$ in round $l+1$. Thus, all the correct processes receive V_i in round $l+1$ and set $V[k] = v_k$. Thereafter, the value $V[k]$ at any correct process may not change anymore. Thus no correct process p_j ever sets $V_j[k] := \bot$.

□

By Lemma 31 and Claim 1, every correct process eventually decides on at most one vector of values. By Lemma 32, for any decided vector V and for any k, $V[k]$ is either $\bot$ or p_k's value. For any two vectors of values V_i and V_j respectively decided by two correct processes p_i and p_j, and for any k, $V_i[k] = V_j[k]$ then directly follows from Claims 2 and 3. □

A.2 Uniform Interactive Consistency in Omission and Crash

The algorithm in Figure 2 solves IC in *Omission* and *Crash*. In both models we assume $t < n$. In the algorithm, all the processes that decide, decide after $t+1$ rounds. We show in this section that the algorithm in Figure 2 satisfies the specification of Uniform IC, through a series of lemmas (Lemma 34 to 36).

Lemma 34 *In the algorithm in Figure 2, every correct process eventually decides.*

Proof: It is clear that the algorithm runs for exactly $t+1$ rounds at each process, and no process may ever block while executing the algorithm. Thus every process that does not crash, including any correct process, decides at the end of round $t+1$. $\square$

Lemma 35 *In the algorithm in Figure 2, for any decision vector V, the j^{th} component of V is either the value proposed by p_j or $\perp$, and is $\perp$ only if p_j fails.*

Proof: We observe that the processes only decide at line 6, and each process p_i decides on its vector V_i of estimate values. Throughout the algorithm, the coordinate $V_x[j]$ at any process p_x, corresponding to the estimate value proposed by any process p_j, is assigned with either $\perp$ or p_j's initial value (at line 1), or p_k's component from another vector of estimate values (at line 5). Hence, any $V_x[j]$ component at any process p_x may only contain p_j's initial value or $\perp$.

If p_j is a correct process and thus never crashes, p_j sends its initial vector of estimate values (which contains its initial value v_j) to all the other processes in round 1, and thereafter, every process p_i that reaches the end of round 1, receives p_j's message at line 5 and assigns $V_i[j]$ with v_j. (In the algorithm, the expression $(j-r+1)$ is evaluated to j at any process p_i.) In any subsequent round r, the j^{th} component of any vector V_i for any process p_i does not change. $\square$

Lemma 36 *In the algorithm in Figure 2, no two processes decide on different vectors of estimate values.*

Proof: We show that all the processes that decide a vector of estimate values at line 6, decide on the same component for any process p_i.

Suppose by contradiction that there exist two processes p_a and p_b that respectively decide on the vectors of estimate values V_a and V_b, such that $V_a[i] \neq V_b[i]$. Furthermore, assume without loss of generality that $V_a[i] = v_i$ and $V_b[i] = \perp$.

According to the algorithm in Figure 2, and in particular to line 5, all processes copy p_i's estimate value from p_i's vector in round 1, from p_{i+1}'s vector in round 2, ..., from $p_{(i+t)\ mod\ n}$'s vector in round $t+1$. Among processes p_i to $p_{(i+t)\ mod\ n}$, one is necessarily correct, say $p_{(i+x)\ mod\ n}$ $(0 \leq x \leq t)$, as there are at most t processes that may ever commit a send omission in any execution.

In round $x+1$ $(1 \leq x+1 \leq t+1)$, every process p_j, and in particular p_a and p_b, receives $p_{(i+x)\ mod\ n}$'s vector of estimate values, and copies p_i's estimate value from $V_{(i+x)\ mod\ n}$ to V_j. Thereafter, all the processes have the same estimate value for p_i's value. There are two cases, whether $V_{(i+x)\ mod\ n}[i]$ is v_i or is $\perp$:

- $V_{(i+x)\ mod\ n}[i] = v_i$. Every process, in particular p_b, receives $p_{(i+x)\ mod\ n}$'s message. Thus p_b assigns $V_b[i]$ with v_i. Thereafter, p_b only receives v_i in the i^{th} component of any vector of estimate values. A contradiction with the fact that p_b decides on V_b and $V_b[i] = \bot$.

- $V_{(i+x) \ mod \ n}[i] = \bot$. Every process, in particular p_a, receives $p_{(i+x) \ mod \ n}$'s message. Thus p_a assigns $V_a[i]$ with $\bot$. Thereafter, p_a only receives $\bot$ in the i^{th} component of any vector of estimate values. A contradiction with the fact that p_a decides on V_a and $V_a[i] = v_i$.

□

A.3 Uniform Interactive Consistency in General-MAJ

The algorithm in Figure 3 solves IC in *General-MAJ*. We show that the algorithm satisfies termination, validity and uniform agreement.

Lemma 37 *In the algorithm in Figure 3, for any decision vector V, the j^{th} component of V is either the value proposed by p_j or $\bot$, and is $\bot$ only if p_j fails.*

Proof: We observe that the processes only decide at line 14, and each process p_i decides on its vector V_i of estimate values. Throughout the algorithm, the coordinate $V_x[j]$ at any process p_x, corresponding to the estimate value of any process p_j, is assigned with p_j's initial value (at line 3), or p_k's component from another vector of estimate values (at line 12). Hence, any $V_x[j]$ component may only contain p_j's initial value, or $\bot$.

If p_j is a correct process and thus never crashes, p_j sends its initial vector of estimate values (which contains its initial value v_j) to all the other processes in round 1, and thereafter, every process p_i that reaches the end of round 1, receives p_j's message at line 6 and assigns $V_i[j]$ with v_j. In any subsequent round r, the j^{th} component of any vector V_i for any process p_i does not change. □

Lemma 38 *In the algorithm in Figure 3, every correct process eventually decides.*

Proof: Suppose by contradiction that there is a correct process p_i that does not decide. Since p_i is correct, p_i completes round $t+1$. Furthermore, since p_i does not decide, $|halt_i^{t+1} \cup suspect_i^{t+1}| \geq t+1$. Consider any process p_j in the set $halt_i^{t+1} \cup suspect_i^{t+1}$. There is at least one round $k \leq t+1$ in which either p_i does not receive a message from p_j, or p_j does not receive a message from p_i. Since p_i is correct, (1) if p_i does not receive a message from p_j in round k, then either p_j crashes in round r, or commits a send-omission in round k, or (2) if p_j does not receive a message from p_i in round k, then p_j commits a receive omission in round k. In both cases, p_j is a faulty process. Therefore, $|halt_i^{t+1} \cup suspect_i^{t+1}| \geq t+1$ implies there are more than t faulty processes. A contradiction. $\square$

Lemma 39 *In the algorithm in Figure 3, no two processes decide on different vectors of estimate values.*

Proof: Suppose by contradiction that there are two distinct processes p_a and p_b which decide on distinct vectors, respectively V_a and V_b. Without loss of generality, consider that there exists k, such that $V_a[k] = c \neq V_b[k] = d$. Hence, at the end of round $t+1$, we have $V_a[k] = c$ and $V_b[k] = d$, and $|halt_a \cup suspect_a| \leq t$ and $|halt_b \cup suspect_b| \leq t$.

For any process p_i, the k^{th} component of V_i, $V_i[k]$, is only assigned with the value $V_j[k]$ from another process p_j, and is assigned with p_k's initial value by p_k at the initialization of the algorithm. Hence, $V_i[k]$ may only contain v_k or $\perp$. Assume without loss of generality that $V_a[k] = v_k$ and that $V_b[k] = \perp$. For every run of the algorithm in Figure 3, let $C_0 = \{p_k\}$ and C_x $(1 \leq x \leq t+1)$ be the set of every process p_l such that $V_l[k] = v_k$ at the end of round $x' \leq x$. From the definition of C_x, we immediately observe that:

(1) O1: For $0 \leq x \leq t+1$, $C_x \subseteq C_{x+1}$. This follows from the fact that as soon as any process p_l assigns v_k to $V_l[k]$, p_l keeps $V_i[k]$ unchanged.
(2) O2: For $0 \leq x \leq t+1, \forall p_l \in C_x$, if p_l sends its vector of estimate values in round $x' > x$, then $V_l[k] = v_k$.

In the following, we prove five lemmas (Lemma 40 to Lemma 44) based on these assumptions. Lemma 44 contradicts Lemma 41. $\square$

Lemma 40 *Consider any process p_l after completing round k $(1 \leq k \leq t+1)$. Let $senderMsg_l^k$ be the processes from which p_l receives a message in round k and that do not already belong to $halt_l^{k-1}$. Then $senderMsg_l^k = \Omega \backslash halt_l^k$.*

Proof: Consider any process p_l that reaches the end of round k, for $1 \leq k \leq t+1$, and consider any other process $p_m \in \Omega$ distinct of p_l. There are three exhaustive and mutually exclusive cases regarding the message from p_m to p_l in round k: (1) if p_l does not receive any message from p_m, then p_l inserts p_m in $halt_l$ (line 10); (2) if p_l receives a message from p_m in round k and $p_m \notin halt_l^{k-1}$, then $p_m \in senderMsg_l^k$ and $p_m \notin halt_l^k$; (3) if p_l receives a message from p_m in round k and $p_m \in halt_l^{k-1}$, then $p_m \notin senderMsg_l^k$ and $p_m \in halt_l^k$ (line 6). Thus any process p_m is either in $senderMsg_l^k$ or in $halt_l^k$. $\square$

Lemma 41 $|C_t| \leq t$.

Proof: Suppose by contradiction that $|C_t| > t$. Consider the message sent by any process $p_m \in C_t$ to p_b. From the observation O2 here above, it follows that either p_b receives from p_m a message with $V_m[k] = v_k$ in round $t+1$, or p_b does not receive any message from p_m in round $t+1$ due to some failure.

Now consider the messages received by p_b in round $t+1$. The set $senderMsg_b^{t+1}$ does not contain the message from p_m, otherwise p_b sets $V_b[k] = v_k$. Therefore, from Lemma 40, $p_m \in halt_b^{t+1}$. As a result, $C_t \subseteq halt_b^{t+1}$ and $|halt_b \cup suspect_b| \geq |halt_b| \geq |C_t| > t$: a contradiction. $\Box$

Lemma 42 *$p_a \in C_{t+1}$ and $p_a \notin C_{t-1}$.*

Proof: Suppose by contradiction that $p_a \in C_{t-1}$. Consider any process $p_m \in \Omega \backslash C_t$ which sends a message to p_a in round $t+1$. From the definition of $\Omega \backslash C_t$, $V_m[k] = \perp$. Therefore, p_m does not receive any message from p_a in round t. Hence, from Lemma 40, $p_a \in halt_m^t$. Therefore, from each process in $\Omega \backslash C_t$, p_a either receives a message $[V', Halt']$ in round $t+1$, such that $p_a \in Halt'$, or p_a does not receive a message due to some failure. Thus, every process in $\Omega \backslash C_t$ is either in $suspect_a^{t+1}$ or is in $halt_a^{t+1}$. Consequently, $\Omega \backslash C_t \subseteq halt_a^{t+1} \cup suspect_a^{t+1}$. From Lemma 41, it follows that $|\Omega \backslash C_t| \geq n - t > t$ (recall that $t < \frac{n}{2}$ in $Omission(n,t)$). So $|halt_a^{t+1} \cup suspect_a^{t+1}| \geq |\Omega \backslash C_t| > t$: a contradiction. $\Box$

Lemma 43 *For all k such that $0 \leq k \leq t-1$, $C_k \subset C_{k+1}$.*

Proof: Consider any $0 \leq k \leq t-1$. From the observation O2 here above, $C_k \subseteq C_{k+1}$. So, either $C_k \subset C_{k+1}$, or $C_k = C_{k+1}$. Suppose by contradiction $C_k = C_{k+1}$.

For any process $p_m \in \Omega \backslash C_{k+1}$, p_m does not receive any message from any process in C_k; otherwise, p_m sets $V_m[k] = v_k$ and $p_m \in C_{k+1}$. Therefore, from Lemma 40, $C_k \subseteq halt_m^{k+1}$. Since $C_k = C_{k+1}$, so for every process $p_m \in \Omega \backslash C_{k+1}$, $C_{k+1} \subseteq halt_m^{k+1}$. Thus, in every round higher than $k+1$, the processes in $\Omega \backslash C_{k+1}$ ignore all the messages from any process in C_{k+1} while updating their vector of estimate values. For any process $p_m \in \Omega \backslash C_{k+1}$, $V_m[k] = \perp$. Thus, after $k+1$ rounds, the set C never changes (no process in $\Omega \backslash C$ ever assigns $V[k]$ with v_k), i.e., $C_{k+1} = C_{k+2} = \ldots = C_{t+1}$. A contradiction with Lemma 42. $\Box$

Lemma 44 $|C_t| \geq t+1$.

Proof: Lemma 43 implies that for every $0 \leq k \leq t-1$, $|C_{k+1}| - |C_k| \geq 1$. We know that $C_0 = \{p_k\}$ and thus $|C_0| \geq 1$. Therefore, $|Ct| \geq t+1$. (A contradiction with Lemma 41.) $\Box$