Distributed Programming for Dummies
A Shifting Transformation Technique*

Carole Delporte-Gallet¹, Hugues Fauconnier¹,
Rachid Guerraoui², Bastian Pochon²

¹ LIAFA Institute
Université Denis Diderot
F-75251 Paris 5

² Distributed Programming Laboratory
EPFL
CH-1015 Lausanne

Abstract

The perfectly synchronized round model provides the powerful abstraction of crash-stop failures with atomic message delivery. This abstraction makes distributed programming very easy. We present an implementation of this abstraction in a distributed system with general message omissions. Protocols devised using our abstraction (i.e., in the perfectly synchronized round model) are automatically transformed into protocols for the omission model. The transformation is achieved using a round shifting technique with a constant time complexity overhead. This transformation is in a precise sense optimal. Furthermore, and rather surprisingly, no automatic transformation from a weaker model, say 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 better time complexity performance.

Keywords: Abstraction, simulation, distributed systems, fault-tolerance, synchronous systems, complexity.

1 Introduction

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

Unfortunately for the programmers, and fortunately for our research community, the assumption that all processes have the same view of the global distributed system state does rarely 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, typically because of a buffer overflow at a router, or because of the crash of some computer hosting processes involved in a distributed computation.

It is of course legitimate to figure out whether we could provide the programmer with the simplistic 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

*Technical Report ID: IC/2003/49, School of Computer and Communication Sciences, EPFL.
of a computer scientist is usually about providing programming abstractions that hide low level details, so why not try to provide those that make the job of the programmer really easy.

The very fact that no such abstraction has 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 model (PSR) abstraction 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 [14].

This observation highlights two issues. The first has to do with feasibility. To implement the PSR abstraction over a given model, one needs to make some synchrony assumptions on the model [7], 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 such evidence 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 the general-omission model (or simply omission model) [12]. In this model, processes proceed in a round-by-round manner, but messages can be lost. The PSR abstraction can indeed be implemented in such a model, but the inherent cost of implementing it in this model was unclear.

**Background.** The lack of any evidence about the cost of implementing PSR might seem surprising given the amount of work that was devoted, either (1) to devising optimal agreement algorithms over various models, including the omission model, or (2) to implementing weaker forms of PSR.

(1) In particular, we do know that, in terms of time complexity, the tight lower bound on implementing interactive consistency in an omission model where \( t \) processes can fail is \( t + 1 \) [6]. That is, \( t + 1 \) rounds of the 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 [16], 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 [5, 10] 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. There is however a tight lower bound of 2 rounds in failure-free runs [4, 9]. Thus, \( 2K + f \) rounds would anyway be needed to implement \( K \) rounds of PSR with \( f \) actual failures.

(2) Implementing a synchronous round-based model with crash failures [8] (crash-stop model) over various weaker models, such as the omission model, has been the subject of several investigations, e.g. [1, 13]. These can be viewed as implementing an abstraction that is weaker than PSR.\(^1\) The idea underlying the implementation proposed in [13], for the models we consider, is that of doubling rounds.\(^2\) 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.

---

\(^1\)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).

\(^2\)Byzantine failures are considered in [13].
In both cases, we end up with multiplicative factor overheads.\textsuperscript{3} 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., $K + C$? At first glance, this would be counter-intuitive because it would mean devising a more efficient implementation than [13] for an abstraction that is strictly stronger.

**Contributions.** This paper presents a time-efficient shifting technique to implement the PSR abstraction over an omission model: $K$ rounds of PSR require $K + (f + 1)$ rounds of the omission model when $f$ failures occur. That is, $C = f + 1$. This is clearly optimal because PSR solves interactive consistency in one round, and this costs at least $f + 2$ in the omission model [4, 9]. In other words, any shifting transformation technique from the PSR model to the omission model has to pay the $f + 1$ cost.

Furthermore, and maybe even more interestingly, we show the surprising (shifting optimality) result that, had we tried to devise a shifting technique to implement a weaker abstraction than PSR (say a crash-stop synchronous model) over a stronger model than general omissions (say send omissions [8]), we would not have gained anything in terms of time complexity. In particular, this means that our technique is also optimal even to implement the crash-stop model abstraction.

We precisely define the notion of shifting transformation and then describe our own 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 omission model 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. To conclude, the paper discusses the applicability of our technique to other models, e.g., with Byzantine failures [11]. For space limitation, we omit some proofs. These are given in an optional appendix.

## 2 Model

**Processes.** We consider a finite set $\Omega$ of $n$ processes $\{p_1, \ldots, p_n\}$, that communicate by message-passing. We assume that processes are fully connected. A process is characterized by its local state and we denote by $S$ 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 $\mathcal{M}' = \mathcal{M} \cup \{\bot\}$ the set of messages that can be received. $\bot$ is a special value that indicates that no message has been received. The primitive send() allows a process to send a message to the processes in $\Omega$. The primitive receive() allows a process to receive a message sent to it that 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 receiveInput(). We denote by $I$ 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 4, 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 the first step, the process receives an external input value. In the second step, the process

\textsuperscript{3}Even if we try to implement the weaker crash-stop abstraction along the lines of [13]. In fact, if we implement PSR directly on the crash-stop model (used as an intermediate model), and use the transformation of [13], 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.
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.

Failure Patterns. A failure pattern is a function $F : \Omega \times \mathcal{R} \to 2^\Omega \times 2^\Omega \times \{0,1\}$. For any given process $p_i$ and round number $r$, $F(i,r)$ returns the set of processes to which $p_i$ fails to send its message, the set of processes from which $p_i$ fails to receive their message, and whether $p_i$ crashes (1) or not (0), in round $r$. We assume that processes do not recover after crashing: a process that crashes in a round does not send nor receive any message in any subsequent round. Thus, for any process $p_i$ and round $r$, any failure pattern $F$ considered implicitly satisfies the condition $(\forall P, Q \subseteq \Omega)(F(i,r) = (P, Q, 1) \Rightarrow F(i, r + 1) = (\Omega, \Omega, 1))$.

A process $p_i$ is correct up to round $r$,\footnote{A process is always correct up to round 0} under any failure pattern $F$, if $p_i$ does not fail in sending nor receiving messages, and does not crash, up to and including round $r$ under $F$, i.e., $F(i, r') = (\emptyset, \emptyset, 0)$, for $1 \leq r' \leq r$. A process $p_i$ is correct if it always sends and receives correctly, i.e., $F(i, r) = (\emptyset, \emptyset, 0)$, for all $r \in \mathcal{R}$. A process that is not correct is faulty. Let correct($F$) be the set of all correct processes under failure pattern $F$, and faulty($F$) = $\Omega$ – correct($F$) be the set of all faulty processes in failure pattern $F$. For any failure pattern $F$, we denote by $f$ the effective number of faulty processes in $F$, i.e., $f = |\text{faulty}(F)|$. In this paper, we consider the following types of failures:

- Atomic failure: A process $p_i$ that crashes in a round $r$ is correct up to round $r - 1$. In round $r$, in which $p_i$ crashes, $p_i$ can either crash before sending a message to all or after sending a message to all. More precisely, the corresponding failure pattern is such that $F(i, r) = (\Omega, \Omega, 1)$ or $(\exists Q \subseteq \Omega)(F(i, r) = (\emptyset, Q, 1))$.

- Crash failure: A process $p_i$ that crashes in a round $r$ is correct up to round $r - 1$. In round $r$, in which $p_i$ crashes, $p_i$ 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. More precisely, the corresponding failure pattern is such that $(\exists P \subseteq \Omega)(F(i, r) = (P, \Omega, 1))$ or $(\exists Q \subseteq \Omega)(F(i, r) = (\emptyset, Q, 1))$.

- 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 precisely, $(\exists P \subseteq \Omega, P \neq \emptyset)(F(i, r) = (P, \emptyset, 0))$.

- General-omission failure: A process $p_i$ that commits a general-omission in a round $r$ fails to send or receive a message to or from a subset of processes in the system. More precisely, $(\exists P, Q \subseteq \Omega, P \cup Q \neq \emptyset)(F(i, r) = (P, Q, 0))$.

Models. A model $M$ is defined as a set of failure patterns. We define four distinct models:

- Model PSR$((n, t))$ (Perfectly synchronized round) is defined by all failure patterns over $n$ processes where at most $t < n$ processes are subject to atomic failures only, and the remaining processes are correct.

- Model Crash$((n, t))$ is defined by all failure patterns 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 failure patterns over $n$ processes where at most $t < n$ processes are subject to either crash failures or send-omission failures, and the remaining processes are correct.

• Model $General(n, t)$ is defined by all failure patterns over $n$ processes where at most $t < n$ processes are subject to either crash failures or general-omission 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 $M_S \supsetneq M_W$ and $M_W \not\subseteq M_S$. Weaker and strictly weaker relations are defined accordingly. From the equations above, it is clear that $PSR(n, t) \succeq Crash(n, t) \succeq Omission(n, t) \succeq General(n, t)$.

Protocols. Processes execute protocols $\langle \Pi_1, \ldots, \Pi_n \rangle$. Each process $p_i$ executes $\Pi_i$. A protocol $\Pi_i$ is a state machine, 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 $O_i : S \times I \times R \rightarrow M$ generates the message to be sent by process $p_i$ during round $r$, given its current state, an external input value, and the round number.\(^5\) The state transition function $T_i : S \times (M')^n \times R \rightarrow S$ 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. 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.

Runs and Problem Specifications. In the following, $state_i$ denotes a variable maintaining the current state of process $p_i$ (initialized to $s_i$). In round $r$, for any input pattern $I$ and any failure pattern $F$, where $F(i, r) = (P, Q, x)$ for some subsets $P, Q$ of $\Omega$ and $x \in \{0, 1\}$, $p_i$ sends to $p_j$ the message $O_i(state_i^{r-1}, I(i, r), r)$ if $j \notin P$, or $\perp$ if $j \in P$. $p_i$ receives from $p_j$ the message sent by $p_j$ if $j \notin Q$, or $\perp$ if $j \in Q$, and $p_i$ changes its state according to $T_i$ if $x = 0$, or keeps the same state in the case $x = 1$.

A run corresponds to an execution of a protocol, and is defined as a tuple $\langle I, F, ST, ME \rangle$, where $I$ is the input pattern observed in the run, $F$ is the failure pattern observed in the run, $ST : \Omega \times R \cup \{0\} \rightarrow S$ 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 $r$ ($ST(i, 0) = s_i$), and $ME : \Omega \times R \rightarrow M'$ is a function such that, for any process $p_i$ and round $r$, $ME(i, r)$ is the message sent by $p_i$ in round $r$, or $\perp$ if $p_i$ fails to send any message in round $r$. We denote by $R(\Pi, M, \Gamma_\perp)$ the set of all runs produced by protocol $\Pi$ with failure patterns in model $M$ and input pattern in $\Gamma_\perp$. A problem (or problem specification) $\Sigma$ is defined as a predicate on runs.

Definition 1 A protocol $\Pi$ solves a problem $\Sigma$ in model $M$ with input pattern in $\Gamma_\perp$ if and only if $\forall R \in R(\Pi, M, \Gamma_\perp)(R$ satisfies $\Sigma)$.

\(^5\)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.
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 in 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 $T$ takes any protocol $\Pi_s$ designed to run in the strong model $M_s$ and converts it into a protocol $\Pi_w = T(\Pi_s)$ that runs correctly in the weak model $M_w$. For example, $M_s$ is PSR and $M_w$ is Crash. To avoid ambiguities, we call a round in the weak model $M_w$, a phase.

The transformation of a protocol $\Pi_s$ in $M_s$ to a protocol $\Pi_w$ in $M_w$ is defined through a simulation function, $Sim$, which simulates a run of $\Pi_s$ by a run of $\Pi_w$. For any process $p_i$ executing a protocol $\Pi_w$ in $M_w$ simulating $\Pi_s$, the local state $s_i$ of $p_i$ contain variables $s.states_i$ and $s.ss_i$, which maintain the simulated states of protocol $\Pi_s$. More precisely, $s.states$ is a set of rounds 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 $r$ ($s.states_i^0 = \emptyset$, $s.ss_i^0[0] = s_1$). We now give the formal definitions of our transformation notions, over an arbitrary set of input values $I$.

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

(i) $R_s = \langle I_s, F_s, ST_s, ME_s \rangle$ and $R_s \in R(\Pi_s, M_s, \Gamma_T)$

(ii) $correct(F_w) \subseteq correct(F_s)$

(iii) $I_w = I_s$

(iv) $\exists r \subseteq ST_w(i, x).states(ST_w(i, x).ss[r] = ST_s(i, r))$

(v) $\forall y < x \exists z \in ST_w(i, x).states(ST_w(z, x).ss[r] = ST_s(z, r))$

(vi) $\forall r, r' \in \mathbb{R}, r \neq r' \exists p_i \in \mathbb{P}(ST_w(i, r).state \cap ST_w(i, r').state = \emptyset)$

(vii) $\forall x \in R(\forall p_i \in \mathbb{P}(\forall r \in ST_w(i, x).state \forall r' < r \exists z \in ST_w(i, z).state)$

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). Property (iii) states that the input pattern is preserved by the simulation. Property (iv) states that any simulated state is correct w.r.t. $\Pi_s$. Property (v) forces the simulation to accomplish progress. Property (vi) states that each state of $\Pi_s$ is simulated at most once. Property (vii) forces a process to simulate states sequentially w.r.t. $\Pi_s$. Apart from Property (iii), our definition encompasses the notion of simulation of $[13]$.

In the following definition, we recall the notion of effectively solving $[13]$ a problem, to indicate that the resolution is obtained through a simulation function.

---

Footnotes:

5 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 [13]. The contradiction is in appearance only, and depends on how we define the notion of simulation.

6 Contrary to the doubling technique of [13] where each state of the run in $\Pi_s$ simulates at most one state of a run in $\Pi_w$, we do not restrict our transformation to simulate only one state of a run of $\Pi_s$ in a state of the run of $\Pi_w$.

8 Indeed, the notion of input pattern does not appear in [13]. In the transformation of [13] 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 $ST_w(i, 2x + 1).states = \emptyset$. 

6
**Definition 3** For any given function $Sim$, $\Pi_w$ effectively solves problem $\Sigma$ in model $M_w$ with input pattern in $\Gamma_T$ if and only if $\forall R \in R(\Pi_w, M_w, \Gamma_T)(Sim(R)$ satisfies $\Sigma)$

The next proposition follows from definitions 2 and 3.

**Proposition 4** Let $\Pi_s$ be any protocol that solves specification $\Sigma$ in model $M_s$. For any given function $Sim$, if $T$ is a transformation from $M_s$ to $M_w$, then protocol $T(\Pi_s)$ effectively solves $\Sigma$ in model $M_w$.

## 4 Shifting Transformation

We present our algorithm to transform any protocol $\Pi$ written in $PSR$ into a protocol $\Pi'$ in a weaker model $M$ such that $\Pi'$ simulates $\Pi$, through a simulation function $Sim$ that we give. For any two distinct processes $p_i$ and $p_j$ simulating protocol $\Pi$, we do not necessarily assume that $\Pi_i = \Pi_j$. However, we will assume that $p_i$ knows the protocol $\Pi_j = (s_j, T_j, O_j)$ executed by $p_j$. We relax this assumption afterwards. Our transformation works on a round basis: it transforms a single round in $PSR$ into several phases in $M$. 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.

Let $\Pi_s$ be any protocol in model $M_s$, $T$ any transformation from $M_s$ to $M_w$, and $\Pi_w = T(\Pi_s)$ the transformed protocol. Roughly speaking, a shifting transformation is such that any process simulates round $r$ of $\Pi_s$ after a bounded number of phases counting from phase $r$. More precisely:

**Definition 5** A transformation $T$ from model $M_s$ to model $M_w$ is a shifting transformation if and only if there exists a constant $C \in \mathbb{N}$, such that, for all $r \in \mathcal{R}$, $f(r) = r + C$. We call $C$ the shift of the transformation.

**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 (IC) problem [14]. In the IC problem, each process $p_i$ is supposed to propose an initial value ($p_i$'s input value in the round) and eventually decide on a vector of values, such that (termination) every correct process eventually decides, (uniform agreement) no two decided vectors are different, and (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. We assume that, for any model $M$ among those we consider, there exists a constant $\delta$ such that all processes that decide, decide in at most $\delta$ rounds. Typically $\delta$ is $t + 1$ in our $Crash(n, t)$ model [6]. An IC algorithm with these requirements can be derived relatively easily by extending an existing uniform consensus algorithm in model $M$ (e.g., those in [15]).\footnote{Our transformation requires $t < n/2$ when considering the General($n, t$) model [13].}

In every phase, the processes start a new instance of IC, and propose to this instance the input values received in that phase. The decision vector corresponds at the same time to a round of the failure pattern, and of the input pattern.
Figure 1: Transformation algorithm (code for process $p_i$)

Figure 1 gives the transformed protocol $\mathcal{T}(\Pi_s)$ for process $p_i$, in terms of $\Pi_s$ and the input pattern $I$. 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 $\text{decision}_i(\text{simulatedRound})$ the decision vector for the instance of IC in line 13, $\text{failure}_i(\text{simulatedRound})$ the value of the variable $\text{failure}$ updated in line 14, $\text{rcvd}_i(\text{simulatedRound})$ the value of the variable $\text{rcvd}$ updated in lines 21 or 23, and $\text{simst}_i(\text{simulatedRound})$ the value of the variable $\text{simst}$ updated in lines 26 or 28. The following proposition defines the simulation function $\text{Sim}$ in our transformation.

**Proposition 6** The simulation $\text{Sim}$ for a run of $\mathcal{T}(\Pi_s)$, $R = (I, F, ST, ME)$, is defined by $R^r = (F, F^r, ST^r, ME^r)$ as follows. Let $p_i$ be a process in $\text{correct}(F)$ ($p_i$ is such that, for all $r$, $F(i, r) = (\emptyset, \emptyset, 0)$ hence $p_i$ does not halt in lines 15 or 31). We consider the simulation of round $r$ of $R^r$, for any process $p_j$. 

---

10For the sake of simplicity, this protocol is given here in an operational manner (i.e., pseudo-code).
(i) \( I'(j, r) \) is the value decision \( i_r[j] \) of the \( r \)-th instance of IC.
(ii) if \( p_j \in failure_i(r) \) then \( F'(j, r) = (\Omega, \Omega, 1) \), otherwise \( F'(j, r) = (\emptyset, \emptyset, 0) \).
(iii) if \( r = 0 \) then \( S^T(j, 0) = s_j \), otherwise \( S^T(j, r) = \text{sim} \text{an}_{i_{r[j]}} \).
(iv) if \( p_j \in failure_i(r) \) then \( ME'(j, r) = \bot \), otherwise \( ME'(j, r) = \text{rcvd}_{i_{r[j]}} \).

**Proposition 7** The algorithm of Fig. 1 is a shifting transformation from \( PSR(n, t) \) to \( Crash(n, t) \) (resp. \( Omission(n, t) \), \( General(n, t) \) (where \( t < n/2 \)) where the shift is the number of rounds needed to solve Interactive Consistency in \( Crash(n, t) \) (resp. \( Omission(n, t) \), \( General(n, t) \)).

**Transformation Extension.** We discuss here an extension of our transformation, where processes do not actually need to know each other’s protocol. (We give the full extended transformation algorithm in optional Appendix A.) In the transformation of Figure 1, each process only sends its input value in every phase, because the protocol itself can be locally simulated by every other process. Here processes do not know the protocol of 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 \( \Pi_i \) (i.e., the output of function \( O_i \)). However, as with our previous transformation, the simulation of a round does not wait for all previous simulations to terminate, but starts before the decision of some simulations are known. Process \( p_i \) cannot know in which precise state of its 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 \( O_i \)). The idea of the extended transformation is to anticipate, for each process \( p_i \), all simulated states in the simulated run that are coherent with terminated simulations of previous rounds, and thus to generate all possible messages for a given round. When all preceding simulations terminate, a process determines which was the state correctly anticipated.

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., \( |I| < \infty \)). 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. Following the structure of the algorithm in optional Appendix A, the algorithm in Fig. 1 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 may be very high.

5 Performance

We analyze the performance of our transformation technique, and prove its optimality by introducing several intermediate models, in which processes can omit messages for a bounded number of rounds before crashing.

**Complexity.** We denote by \( \delta_f \) and \( \tau_f \) the number of phases needed for all processes to respectively decide and to terminate IC, in any run with \( f \) failures. We define the *phase complexity overhead* as the number of additional phases executed by the transformed protocol \( \Pi_w \) in \( M_w \), w.r.t. the original protocol \( \Pi_s \) in \( M_s \). In our transformations, we overlap the simulation of rounds with a one-phase interval. The only phase complexity overhead is the number of phases needed before obtaining the result of the simulation of the first round, which corresponds to \( \delta_f - 1 \). In the \( Crash(n, t) \) model, the tight lower bound for reaching a global decision is \( \delta_f = f + 2 \) phases [4, 9], in every run where there are \( f \) failures. Hence the phase complexity overhead in failure-free runs (\( f = 0 \)) is exactly
one phase. In terms of messages, our transformation generates at most a \( n \log_2 |X| \)-bit message per process, per round, and per IC instance. Hence each process sends a \( nTf \log_2 |Z| \)-bit message to each other process in every phase, in any run with \( f \) failures.\(^{11}\)

**Proposition 8 (Shifting Optimality)** Any automatic shifting transformation from a model \( M_s \) weaker than PSR to any model \( M_w \) weaker than \( M_s \) introduces a shift of at least \( \delta \) phases.

**Proof (Sketch):** We introduce the notion of \( \omega \)-model, in which a faulty process may exhibit a faulty behavior during \( \omega \) phases, and then crashes. More precisely, for any process \( p_i \) and any round \( r \) in any \( \omega \)-model, we have \( F(i, r) \neq (0, 0, 0) \Rightarrow F(i, r + \omega) = (\Omega, \Omega, 1) \). For any model \( M \in \{ Crash, Omission, General \} \), we denote by \( \omega-M(n, t) \) the model defined by all failure patterns in \( M(n, t) \) which satisfy the previous equation. Note that \( Omission \) is equivalent to \( \infty-Omission \), and \( General \) to \( \infty-General \).

Our transformations require \( \delta \) phases to simulate a round in PSR from any \( \omega-M(n, t) \). We consider a problem \( P \), such that there exists an algorithm in \((\omega - 1)-M(n, t)\) which solves \( P \) in \((\omega - 1) \) rounds. We show that any algorithm solving \( P \) in \( \omega-M(n, t) \) requires \( \delta \) phases, and \( P \) can be solved in PSR in one round.

The problem we consider consists in determining the phase number in which a process \( p_i \) fails for the first time. More precisely, each process is supposed to propose a value from 0 to \( \omega - 1 \), corresponding to the phase number in which \( p_i \) appears faulty for the first time, and eventually decide on a phase number, such that \((P-termination)\) any correct process eventually decides, \((P-validity)\) any decided phase number was proposed, and \((P-agreement)\) the difference between any two decisions does not exceed \( \omega - 1 \) phases.

In \( \omega-M(n, t) \), we need \( \delta \) phases to solve this problem. Otherwise, we could solve binary consensus in less that \( \delta \) phases (contradicting [6]). Assume some algorithm \( A \) solve problem \( P \) in less than \( \delta \) phases. Consider algorithm \( A' \) defined as follows. If a process proposes 1, change it to \( \omega \) otherwise (the proposed value is 0) let it unchanged, and execute algorithm \( A \) with the new proposed value. If the decision in \( A \) is \( \omega \) decide 1 otherwise decide 0. We argue that algorithm \( A' \) implements a binary consensus algorithm. Validity and termination follow from \( P \)-validity and \( P \)-termination. Consider agreement. By \( P \)-validity, any decision value is either 0 or \( \omega \). By \( P \)-agreement, the difference between two decisions does not exceed \( \omega - 1 \), hence all decision values are 0 or \( \omega \). \( A' \) is a binary consensus algorithm and runs in less that \( \delta \), contradicting [6]. By definition, this problem is solved in \( \omega - 1 \) rounds in \((\omega - 1)-M(n, t)\). In \( \omega-M(n, t) \), we have just shown that \( \delta \) phases are required. For any bounded number of rounds \( r \) during which we observe a protocol \( \Pi \), any \( M(n, t) \) can be seen as a \( r-M(n, t) \) model. We have shown that there is a problem such that there is no loss of efficiency, in term of time complexity, to simulate PSR rather than any \( r-M(n, t) \) (\( 1 \leq r < r' \)) model. Hence the shifting optimality.

Determining the exact overhead in terms of message size complexity is an open issue.

6 Concluding Remarks

In this paper, we concentrated on three different models, and presented a shifting transformation technique to translate protocols from the perfectly synchronous model into each of these weaker models. With minor modifications, it is possible to use a shifting technique to translate protocols

\(^{11}\)In the extended transformation, a process maintains at least \( 2^{\omega r} \) possible states in a phase. Each state can be coded using \( \sigma = 2\log_2 |S| + (n + 1)\log_2 |M| \) bits, and thus, each process sends a \( \tau \sigma n \sigma 2^{\omega r} \)-bit message to each other process in each phase.
into the Byzantine model [11]. We need however to modify the definition of a simulation (Definition 2), that required some properties to hold over the complete set of processes \( \Pi \), which is not possible in the Byzantine model. These properties should rather refer to the set of correct processes in the original run.\(^{13}\) The transformation algorithm in Fig. 1 has to be slightly modified: uniform agreement of IC is turned into (non-uniform) agreement.\(^{14}\) Indeed, uniform agreement cannot be ensured in the Byzantine model. The correctness of the modified transformation w.r.t. the modified definition is proven similarly to the original transformation (optional Appendix B). Note that generally speaking, in the Byzantine model, processes need to know each other's protocol, possibly to validate received messages w.r.t. the simulated protocol (e.g., in [13]).

References


\(^{12}\)Roughly speaking, in the Byzantine model, a faulty (or byzantine) process may arbitrarily deviate from its protocol, by sending and relaying spurious messages (not necessarily according to its protocol).

\(^{13}\)This is the same restriction as in [13].

\(^{14}\)Non-uniform agreement states that no two vectors decided by correct processes are different.
Optional Appendices

A Extended Algorithm

Consider any process $p_i$ simulating a run $R' = (I', F', ST', ME')$ of PSR. Roughly speaking, the idea of the extended transformation is to maintain, for process $p_i$, all simulated states of $ST'$ that are coherent with terminated simulations of previous rounds. Hereafter, these states are called the extended set of states and denoted by $es$. The set $es$ gathers the possible values of $ST'$ for $p_i$, before the termination of the corresponding simulations.

More precisely, consider any two processes $p_i$ and $p_j$, executing protocol $\Pi$ in $PSR$, and denote by $m_j$ the message $p_j$ sends to $p_i$ in round $r$. In any phase involved in the simulation of round $r$, i.e., one of phases $r$ to $r'$ ($r' \leq r + \delta - 1$), $p_i$ does not know which value will figure in the decision vector decision of the $r$-th instance, at position $j$: (1) as long as $p_i$ has not received $m_j$, decision[$j$] can potentially take any value of $M'$ (including $\bot$), and (2) if $p_i$ receives $m_j$, decision[$j$] can be either $m_j$ or $\bot$. To be able to start the $r + 1$-th instance in the next phase, $p_i$ generates the new extended set of states $es$. To generate this set of states, process $p_i$ computes $T_i$ on every state of the current set of states $es$ and every combination of messages, where $\bot$ values are successively substituted by any value of $M$, and any value successively substituted with $\bot$.

For example, consider the case of the Crash(3,2) model with $I = M = \{0, 1\}$. After phase 1, process $p_i$ gathers the received values in the vector $[10 \bot]$. The possible combinations of the messages received are $[10 \bot], [100], [101], [1 \bot], [1 \bot], [1 \bot], [10 \bot], [100], [101], [1 \bot]$, and $[\bot \bot]$. Process $p_i$ generates the extended set of states $es_i$ by applying function $T_i$ on each combination of messages.

To each state in the extended set of states corresponds a message of $\Pi_i$ to be sent in round $r$ by $p_i$. These messages are gathered in a set, hereafter called the extended message and denoted by $em_i$. Each process sends this message in the next phase.

Figure 2 presents our extended transformation algorithm. For the sake of clarity, we ignore possible optimizations in this algorithm.\(^{15}\) We denote by $rcvd[r]$ the message sent in phase $r$ for the $r$-th IC instance. Furthermore, we assume that each process sends, in each phase of the underlying IC algorithm, the value it proposes to this instance (i.e., if $p_i$ receives a message $m$ from $p_j$ for the $r$-th IC instance in any phase, $m$ contains the value proposed by $p_j$ for this instance).

Detailed Description. Consider a process $p_i$ simulating protocol $\Pi_i = \langle s_i, T_i, O_i \rangle$, and any phase $r$ of the simulated run $R = (I, F, ST, ME)$ of $\Pi$. For any process $p_i$, at the beginning of any phase $r$, $es$ is the extended set of states and it gathers all the possible states for $p_i$ at the end of phase $r - 2$. Each time a new state $st$ is added in $es$, a number $num(st)$ is associated with $st$. The extended message $em$ is a set of tuples. For any process $p_i$, each tuple $(num(st), rec, num(st'), m)$ contains (1) the identifier $num(st)$ of a possible state $st$ of $p_i$ at the beginning of round $r - 1$, (2) a possible combination $rec$ of messages received by $p_i$ in phase $r - 1$, (3) the identifier $num(st')$ of the state $st'$ of $p_i$ at the beginning of phase $r$, such that $st' = T_i(st, rec, r - 1)$, (4) the message

\(^{15}\)E.g., each process can reduce the number of possible states as it receives more values from other processes.
sent in round $r$, i.e., $m = O_i(st', I(i, r), r)$. In the following, the variable $\text{simulatedRound}$ denotes the next round to simulate. We describe below the message generation and then the simulation.

**Message generation.** At the beginning of a phase $r$, any process $p_i$ receives an input value $\text{input} = I(i, r)$. Process $p_i$ then computes, at the same time, the extended set of states $es$ and the extended message $em$, in the following way.

In the following, we denote the extended message sent in phase $r$ by $\text{rcvd}[r]$. Process $p_i$ computes all possible combinations of received messages in phase $r - 1$ from the extended messages received in phase $r - 1$, $\text{rcvd}[r - 1]$. For each state $st$ of $es$, and for any such combination of message $\text{rec}$, $p_i$ computes a possible next state, $st' = T_i(st, \text{rec}, r)$. Whenever $p_i$ includes a new state $st$ in $es$, it associates a unique identifier $num(st')$ with $st'$. Process $p_i$ then computes the extended message $em = \langle num(st), \text{rec}, num(st'), m \rangle$, with $m = O_i(st', \text{input}, r)$. The extended message is a part of the message sent in the phase. The other parts correspond to the messages of all pending IC instances. For any process, the extended messages received in phase $r$ allow to compute the extended set of states and the extended message in phase $r + 1$. For the simulation, we need to keep the extended message received in phase $\text{simulatedRound}$ until phase $r$ (at most $\delta$ extended messages).

**Simulation.** Each process $p_i$ maintains (1) the simulated state of protocol $\Pi_i$ at the end of round $r$, denoted by $ss[r]$, and (2) the number associated with the state currently simulated at each process $p_j$, at the end of round $\text{simulatedRound} - 1$, denoted by $\text{sim}[j]$. In the following, we consider that the simulation has been performed up to, and including, round $\text{simulatedRound} - 1$.

In a phase $r$, some process $p_i$ terminates the execution of the $\text{simulatedRound}$-th IC instance ($r \leq \text{simulatedRound} + \delta - 1$). If the condition of the while loop at line 22 (“$\text{simulatedRound}$-th instance of IC has decided”) is true in a phase $x$ at process $p_i$, then $\text{decision} \langle \text{simulatedRound} \rangle$ denotes the decided vector of messages at line 24, $\text{failure}_i(\text{simulatedRound})$ the value of the variable $\text{failure}$ updated in line 25, and $\text{trueRcd}_i(\text{simulatedRound})$ the value of the variable $\text{trueRcd}$ updated in line 31. Process $p_i$ uses the decided vector $\text{decision}_i(\text{simulatedRound})$ to update the state of the simulated protocol $\Pi_i$, i.e., $p_i$ adds $\text{simulatedRound}$ in $\text{states}$ and computes $ss[\text{simulatedRound}]$. More precisely, $p_i$ proceeds as follows:

1. $p_i$ computes the messages $\text{trueRcd}$ received in the round $\text{simulatedRound}:
   - if $p_j \in \text{failure}$ or $\text{decision}[j] = \perp$, then the message received from $p_j$ is $\perp$,
   - otherwise the message is chosen in the extended message of $p_j$ of phase $r$. Precisely, $p_i$ searches for the tuple $\langle \text{sim}[j], M, s, s \rangle$, where $M$ is the set of messages received in the round $\text{simulatedRound} - 1$ (i.e., the previous value of $\text{trueRcd}$). Let $\langle \text{sim}[j], M, s, m \rangle$ be this tuple. $\text{sim}[j]$ is updated with $s$ and $\text{trueRcd}[j]$ with $m$.

2. $p_i$ updates $ss[r]$ with $T_i(ss[\text{simulatedRound} - 1], \text{trueRcd}, \text{simulatedRound})$.

If any value in the vector $\text{decision}$ is $\perp$, then the corresponding process is added to $\text{failure}$. If any process adds itself to $\text{failure}$, it stops. The following propositions assert the correctness of the extension of our transformation.

**Proposition 9** The simulation Sim for a run of $T(\Pi_s)$, $R = \langle I, F, ST, ME \rangle$ is well defined by $R' = \langle I', F', ST', ME' \rangle$ as follows. Let $p_i$ be a process in $\text{correct}(F)$ ($p_i$ is such that, for all $r$, $F(i, r) = (0, 0, 0)$ hence $p_i$ does not halt in lines 27 or 43). We consider the simulation of round $r$ of $R'$, for any process $p_j$. 

13
1: failure := \emptyset \quad \{ \text{failure corresponds to one round of the failure pattern} \}
2: simulatedRound := 1 \quad \{ \text{simulatedRound is the current simulated round number} \}
3: states := \{0\}; ss[0] := s_i \quad \{ \text{set of states of protocol \Pi which are simulated by \Pi_\omega in the current round} \}
4: \forall j \in [1, n]((\text{sim}[j] := 0) \quad \{ \text{all processes start in the 0-th state} \}
5: number := 1 \quad \{ \text{by convention num(ss[0]) = 0} \}

6: \textbf{for} phase r (r = 1, 2, \ldots) \textbf{do}
7: \textit{input} := receiveInput() \quad \{ \text{receive input value corresponding to } I(i, r) \}
8: \textbf{if} r = 1 \textbf{then}
9: \quad es := \{s_i\}; \quad em := \{\langle -, -, 0, O_i(s_i, \text{input}, r)\rangle\}
10: \textbf{else}
11: \quad es' := \emptyset; \quad em := \emptyset
12: \textbf{for} any possible vector rec of n messages of \text{rcvd}[r - 1] \textbf{do}
13: \quad \textbf{for} any possible state st of es \textbf{do}
14: \quad \quad st' := T_i(st, rec, r); \quad \text{num}(st') := \text{number}; \quad \text{number} := \text{number} + 1
15: \quad \quad es' := es' \cup \{st'\}
16: \quad \quad em := em \cup \{\langle \text{num}(st), \text{rcvd}, \text{num}(st'), O_i(st', \text{input}, r)\rangle\}
17: \quad es := es'
18: \text{start IC instance number } r, \text{ and propose(em)} \quad \{ \text{em is one part of the message sent in the phase} \}
19: \text{execute one round of all pending IC instances} \quad \{ \text{messages of IC instances are the other parts} \}
20: \text{rcvd}[r] := \text{parts of messages of phase } r \text{ corresponding to extended messages}

21: states := \emptyset
22: \textbf{while} simulatedRound-th instance of IC has decided \textbf{do} \quad \{ \text{is there any IC instance decided?} \}
23: \quad states := states \cup \{\text{simulatedRound}\} \quad \{ \text{instance simulatedRound has decided} \}
24: \quad decision := decision vector of instance simulatedRound \quad \{ \text{reconstruct patterns} \}
25: \quad failure := failure \cup \{p_j | \text{decision}[j] = \perp\} \quad \{ \text{ensure failure pattern has only all-or-nothing failures} \}
26: \textbf{if} p_i \in failure \textbf{then}
27: \quad \text{halt} \quad \{ \text{process } p_i \text{ is faulty} \}
28: \textbf{for each } p_j \in \Omega \textbf{ do} \quad \{ \text{adjust decision vector with previous failure pattern} \}
29: \quad \textbf{if} p_j \in failure \textbf{ then}
30: \quad \quad decision[j] := \perp
31: \quad \textbf{for each } p_j \in \Omega \textbf{ do} \quad \{ \text{compose the messages of the round} \}
32: \quad \quad \text{if } p_j \in failure \textbf{ then}
33: \quad \quad \quad tmpRcvd[j] := \perp
34: \quad \quad \text{else}
35: \quad \quad \quad \textbf{if} \text{simulatedRound} = 1 \textbf{ then}
36: \quad \quad \quad \quad tmpRcvd[j] := m \text{ such that } \langle *, *, 0, m \rangle \in \text{decision}[j]
37: \quad \quad \quad \text{else}
38: \quad \quad \quad \quad \text{let } k \text{ and } m \text{ such that } \langle \text{sim}[j], \text{trueRcvd}, k, m \rangle \in \text{decision}[j]
39: \quad \quad \quad \quad \text{tmpRcvd[j]} := m; \text{sim}[j] := k
40: \quad \text{trueRcvd} := \text{tmpRcvd}
41: \quad ss[\text{simulatedRound}] := T_i(ss[\text{simulatedRound} - 1], \text{trueRcvd}, \text{simulatedRound})
42: \quad \text{simulatedRound} := \text{simulatedRound} + 1
43: \textbf{if} r - \text{simulatedRound} \geq \delta \textbf{ then} \quad \{ \text{process } p_i \text{ is faulty} \}
44: \text{halt} \quad \{ \text{process } p_i \text{ does not perform any step} \}

Figure 2: Extended transformation algorithm (code for process \(p_i\)}
1. $I' = I$,

2. if $p_j \in \text{failure}_i(r)$ then $F'(j,r) = (\Omega, \Omega, 1)$, otherwise $F'(j,r) = (0,0,0)$,

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

4. if $p_j \in \text{failure}_i(r)$ then $ME'(j,r) = \bot$, otherwise $ME'(j,r) = trueRcvd(r)[j]$.

**Proposition 10** The algorithm of Fig. 2 is a shifting transformation from $PSR(n,t)$ to $Crash(n,t)$ (resp. $Omission(n,t)$, $General(n,t)$ (where $t < n/2$)) where the shift is number of rounds needed to solve Interactive Consistency in $Crash(n,t)$ (resp. $Omission(n,t)$, $General(n,t)$ ).

### B Proof of Proposition 7

To prove Proposition 7, we first show that the construction of function $Sim$ in Figure 1 is consistent with Proposition 6. We proceed through a series of lemmas.

**Lemma 11** For any failure pattern $F$ and any process $p_i$ in $\text{correct}(F)$, $p_i$ never halts, and decides in all IC instances.

**Proof:** Process $p_i$ is correct and never crashes. By the termination property of IC and the definition of $\delta$, $p_i$ always decides in any instance of IC within $\delta$ phases. Thus $p_i$ never halts in line 31. Moreover, by the validity property of IC, in any decision vector, decision[$i$] $\neq \bot$. Thus $p_i$ never halts in line 15. $\square$

**Lemma 12** For any failure pattern $F$, any process $p_i$ in $\text{correct}(F)$, 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:

- $\text{decision}_i(r) = \text{decision}_j(r)$
- $\text{failure}_i(r) = \text{failure}_j(r)$
- $\text{rcvd}_i(r) = \text{rcvd}_j(r)$
- $\text{simst}_i(r) = \text{simst}_j(r)$

**Proof:** By Lemma 11, $p_i$ 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 $\text{failure}_i$. By the agreement property of IC, $p_i$ and $p_j$ add the same set. By induction, $\text{failure}_i(r) = \text{failure}_j(r)$. As $\text{decision}(r)$ and $\text{failure}(r)$ are the same for all processes for which they are defined, by induction hypothesis, we have $\text{rcvd}_i(r) = \text{rcvd}_j(r)$ and $\text{simst}_i(r) = \text{simst}_j(r)$. $\square$

We can define the simulation through the value of the variables of any correct process. Consider any failure pattern $F$ and let $p_k$ be a correct process in $F$ (we know there exists at least one as $t < n$). We define the simulation through the variables of $p_k$. 15
Lemma 13 If it exists a round $r$ and a process $p_i$ such that $i \in \text{failure}_k(r)$ then, for any $r' \geq r$, $\text{simst}_k(r)[i] = \text{simst}_k(r')[i]$.

Proof: Directly from the algorithm.

Lemma 14 For any process $p_i$, any round $r$ such that $\text{decision}_i(r)$ and $\text{decision}_i(r+1)$ occur, $\text{failure}_i(r) \subseteq \text{failure}_i(r+1)$.

Proof: From the algorithm, $\text{failure}_i$ always increases.

Lemma 15 $\text{correct}(F) \subseteq \text{correct}(F')$.

Proof: By Lemma 11, all correct processes decide in all instances of IC. By definition of $\delta$, they decide in at most $\delta$ phases. Thus they never halt in line 31. By the validity property of IC, the decision value is not $\bot$ for any correct process in any decision vector. Thus no correct process ever halts in line 15. Therefore all correct processes in $F$ are correct in $F'$.

Lemma 16 For any process $p_i$ and any protocol $\Pi_s$ to simulate, let $\langle s_i, T_i, O_i \rangle$ be the protocol for $p_i$ in $M_s$. For any round $r$, we have

1. $s_i = ST'(i,0)$
2. If $F'(i,r) \neq (\ast, \ast, 1)$ then $ME'(i,r) = O_i(ST'(i,r-1),I'(i,r),r)$, otherwise $ME'(i,r) = \bot$
3. If $F'(i,r) \neq (\ast, \ast, 1)$ then $ST'(i,r) = T_i(ST'(i,r-1),[ME'(1,r), \ldots, ME'(n,r)],r)$, otherwise $ST'(i,r) = ST'(i,r-1)$

Proof: (1) is immediate, from the initialization of the variable $\text{simst}_k[i](0)$. We prove (2) and (3) by induction. For the case $r = 1$, and if $F'(i,1) = (\ast, \ast, 1)$, then by definition of $F'$, $p_i \in \text{failure}_k(1)$. By the algorithm $\text{rcvd}_k(1)[j] = \bot$, and by property (iv) of the definition of the simulation, $ME'(i,1) = \text{rcvd}_k(1)[i]$. By Lemma 13 and (1) $ST'(i,1) = ST'(i,0)$. If $F'(i,1) \neq (\ast, \ast, 1)$, then by definition of $F'$, $p_i \notin \text{failure}_k(1)$. By the property (iv) of the definition of the simulation, $ME'(i,1) = \text{rcvd}_k(1)[i]$. By line 21 of the algorithm, $\text{rcvd}_k(1)[i] = O_i(\text{simst}_k(0)[i], \text{decision}_k[i], 1)$, and so $ME'(i,1) = O_i(ST'(i,0), I'(i,1), 1)$. By line 26 of the algorithm, $\text{simst}_k(1)[i] = T_i(\text{simst}_k(0)[i], \text{rcvd}_k, 1)$, and $ST'(i,1) = T_i(ST'(i,0), [ME'(1,1), \ldots, ME'(n,1)], 1)$.

Assume the properties (2) and (3) are true up to round $r-1$. If $F'(i,r) = (\ast, \ast, 1)$ then by definition of $F'$, $p_i \notin \text{failure}_k(r)$. By the property (iv) of the definition of the simulation, $ME'(i,r) = \bot$. By Lemma 13 and (1) $ST'(i,r) = ST'(i,r-1)$. If $F'(i,r) \neq (\ast, \ast, 1)$, then by definition of $F'$, $p_i \notin \text{failure}_k(r)$. By the property (iv) of the definition of the simulation, $ME'(i,r) = \text{rcvd}_k(r)[i]$. By line 21 of the algorithm, $\text{rcvd}_k(r)[i] = O_i(\text{simst}_k(r-1)[i], \text{decision}_k[i], r)$, and so $ME'(i,r) = O_i(ST'(i,r-1), I'(i,r), r)$. By line 26 of the algorithm, $\text{simst}_k(r)[i] = T_i(\text{simst}_k(r-1)[i], \text{rcvd}_k, r)$, and so $ST'(i,r) = T_i(ST'(i,r-1), [ME'(1,r), \ldots, ME'(n,r)], r)$.

By Lemmas 12 and 16, the function $\text{Sim}$ is well defined, and consistent with Proposition 6.

Lemma 17 $R'$ is a run of $\Pi_s$
Proof: Lemmas 12 and 14 show that $F'$ is in $PSR$ if $F$ is in $M$, where $M \in \{\text{Crash, Omission, General}\}$. Lemma 16 shows that the functions $ST'$ and $ME'$ consistently define a run of $\Pi_S$, with input value $I'$ and failure pattern $F'$.

\begin{lemma}
Let $x$ be any phase, $p_i$ any process and $r$ any round. If $r \in ST_w(i, x).states$ then $ST_w(i, x).ss[r] = ST_h(i, r)$.
\end{lemma}

Proof: Each time a round $r$ is added to $states_i$ (line 12), $ss_i[r] = simst_i(r)[i]$ (line 29). By Lemma 12 and the definition of the simulation, $simst_i(r)[i] = ST_h(i, r)$.

\begin{lemma}
For any round $r$ and any process $p_i$ in $correct(F_h)$, if it exists $c$ such that $r \in ST_i(i, c).states$ then $ST_i(i, c).ss[r] = ST_h(i, r)$.
\end{lemma}

Proof: If such a $c$ exists, then $p_i$ has decided in the $r$-th instance of IC, and by the algorithm in Figure 1, in each previous instance. By Lemma 15, we have $ST_i(i, c).ss[r] = ST_h(i, r)$.

\begin{lemma}
Let $p_i$ be any process in $correct(F_h)$ and $r$ any round. There exists a unique $c \leq r + C$ such that $r \in ST_i(i, c).states$ and $ST_i(i, c).ss[r] = ST_h(i, r)$.
\end{lemma}

Proof: By Lemma 11, $p_i$ decides in all instance of IC. In particular, for the $r$-th instance, $p_i$ decides in phase $c$. By assumption, $C$ is the number of phases for an instance of IC to decide, thus $c \leq C + r$. We have $r \in ST_w(i, c).states$, and by Lemma 19, $ST_w(i, c).ss[r] = ST_h(i, r)$.

(End of proof of Proposition 7) We show that our function $Sim$, as defined by Proposition 6, satisfies the seven properties of Definition 2, with $f(r) = r + C$, where $C$ is the number of phases to solve IC in $M_w$.

Lemma 17 shows that $sim(R)$ is a run of $\Pi_S$, which implies Property (i). Lemma 15 shows that $correct(F) \subseteq correct(F')$, which implies Property (ii). By the definition of the simulation, we have $I = I'$, which implies Property (iii). Property (iv) follows from Lemma 18. By Lemma 11, 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 $r + C$. Lemma 20 shows that for any round $r$ and any process $p_i$ in $correct(F_h)$, there exists $c \leq r + C$ such that $r \in ST_w(i, c).states$ and $ST_w(i, c).ss[r] = ST_h(i, r)$ (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).