Distributed Programming for Dummies
A Shifting Transformation Technique

Carole Delporte-Gallet, Hugues Fauconnier
LIAFA Institute
Université Denis Diderot
F-75251 Paris 5

Rachid Guerraoui, Bastian Pochon
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.

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 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* [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] (Byzantine failures are considered in [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. 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 both cases, we end up with multiplicative factor overheads.[2] 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.

[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] 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.

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, proofs are omitted, and can be found in [5].

2. Model

Processes. We consider a finite set Ω 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 $\mathcal{S}$ 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 send() allows a process to send a message to the processes in Ω. The primitive 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 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 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 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 $F : \Omega \times \mathcal{R} \rightarrow 2^{\Omega} \times 2^{\Omega} \times \{0,1\}$. 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 implicitely 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 (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., $f = |faulty(F)|$. 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 $M_s \succeq M_w$ and $M_w \not\succeq M_s$. Weaker and strictly weaker relations are defined accordingly. From the equations above, it is clear that $PSR(n,t) \succeq Crash(n,t) \succeq Omission(n,t) \succeq General(n,t)$.

Protocols. The processes execute a *protocol* $\Pi = \langle \Pi_1, \ldots, \Pi_n \rangle$. 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 current state, an external input value, and the round number.[3] 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. 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 Ω 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, $ST : \Omega \times \mathcal{R} \cup \{0\} \rightarrow \mathcal{S}$ 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 ($ST(i,0) = s_i$), and $ME : \Omega \times \mathcal{R} \rightarrow \mathcal{M}'$ 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 $R(\Pi, M, \Gamma_{\mathcal{I}})$ the set of all runs produced by protocol Π with failure patterns in model M and input pattern in $\Gamma_{\mathcal{I}}$. A *problem* (or problem specification) Σ is defined as a predicate on runs.

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

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 $\mathcal{T}$ 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 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 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.[4] 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.[5] 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 $\mathcal{I}$.

Definition 2 *An algorithm* $\mathcal{T}$ *is called a* 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* $R_w = \langle I_w, F_w, ST_w, ME_w \rangle$ *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* $R_s \in R(\Pi_s, M_s, \Gamma_{\mathcal{I}})$,
- *(ii)* $correct(F_w) \subseteq correct(F_s)$,
- *(iii)* $I_w = I_s$,
- *(iv)* $(\forall x \in \mathcal{R})(\forall p_i \in \Omega)(\forall r \in ST_w(i,x).states)$ $(ST_w(i,x).ss[r] = ST_s(i,r))$,
- *(v)* $(\forall r \in \mathcal{R})(\forall p_i \in correct(F_s))(\exists c \leq f(r))$ $(r \in ST_w(i,c).states)$,
- *(vi)* $(\forall r, r' \in \mathcal{R}, r \neq r')(\forall p_i \in \Omega)$ $(ST_w(i,r).states \cap ST_w(i,r').states = \emptyset)$,
- *(vii)* $(\forall x \in \mathcal{R})(\forall p_i \in \Omega)(\forall r \in ST_w(i,x).states)$ $(\forall r' < r)(r' \in \cup_{k=0}^{x} ST_w(i,k).states)$.

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. Π_s. Property (v) forces the simulation to accomplish progress. Property (vi) states that each state of Π_s

[3] 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.

[4] 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 appearence only, and depends on how we define the notion of simulation.

[5] Contrary to the doubling technique of [13] 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.

is simulated at most once. Property (vii) forces a process to simulate states sequentially w.r.t. Π_s. Apart from Property (iii), our definition encompasses the notion of simulation of [13].[6]

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.

Definition 3 *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 $(\forall R \in R(\Pi_w, M_w, \Gamma_{\mathcal{I}}))(Sim(R)$ satisfies $\Sigma)$.*

The next proposition follows from definitions 2 and 3.

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

4. Shifting Transformation

We present our algorithm to transform any protocol Π written in PSR into a protocol Π' in a weaker model M 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 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 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 Π_s be any protocol in model M_s, $\mathcal{T}$ any transformation 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 5 *A transformation $\mathcal{T}$ 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 δ such that all processes that decide, decide in at most δ rounds. Typically δ 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., [15] (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 gives the transformed protocol $\mathcal{T}(\Pi_s)$ for process p_i, in terms of Π_s and the input pattern I. For the sake of simplicity, this protocol is given here 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 14, $rcvd_i(simulatedRound)$ the value of the variable $rcvd$ updated in lines 21 or 23, and $simst_i(simulatedRound)$ the value of the variable $simst$ updated in lines 26 or 28. The following proposition defines the simulation function Sim in our transformation.

Proposition 6 *The simulation Sim for a run of $\mathcal{T}(\Pi_s)$, $R = \langle I, F, ST, ME \rangle$, is defined by $R' = \langle I', F', ST', ME' \rangle$ as follows. Let p_i be a process in $correct(F)$ (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', 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 $p_j \in failure_i(r)$ then $F'(j,r) = (\Omega, \Omega, 1)$, otherwise $F'(j,r) = (\emptyset, \emptyset, 0)$.*
- *(iii)* *if $r = 0$ then $ST'(j,0) = s_j$, otherwise $ST'(j,r) = simst_i(r)[j]$.*
- *(iv)* *if $p_j \in failure_i(r)$ then $ME'(j,r) = \perp$, otherwise $ME'(j,r) = rcvd_i(r)[j]$.*

[6] 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 defi ned with $f(r) = 2r$, and $c = 2r$ in (v). This implies that $ST_w(i, 2x).states = \{x\}$ and $ST_w(i, 2x+1).states = \emptyset$.

```
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)                         {simst(r)[j] is the state of protocol Π_s for process p_j at the end of round r}
4:  states := {0}                                             {set of rounds of protocol Π_s which are simulated by Π_w in the current round}
5:  ss[0] := s_i                                              {set of states of protocol Π_s which are 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:     failure := failure ∪ {p_j | decision[j] = ⊥}           {ensure failure pattern has only atomic failures}
15:     if p_i ∈ failure then halt                            {is process p_i faulty?}
16:     for each p_j ∈ Ω do                                   {adjust decision vector with previous failure pattern}
17:       if p_j ∈ failure then
18:         decision[j] := ⊥
19:     for each p_j ∈ Ω do                                   {generate messages}
20:       if p_j ∉ failure then
21:         rcvd[j] := O_j(simst(simulatedRound − 1)[j], decision[j], simulatedRound)
22:       else
23:         rcvd[j] := ⊥
24:     for each p_j ∈ Ω do                                   {perform state transitions}
25:       if p_j ∉ failure then
26:         simst(simulatedRound)[j] := T_j(simst(simulatedRound − 1)[j], rcvd, simulatedRound)
27:       else
28:         simst(simulatedRound)[j] := simst(simulatedRound − 1)[j]
29:     ss[simulatedRound] := simst(simulatedRound)[i]
30:     simulatedRound := simulatedRound + 1                  {increment simulated round counter}
31:   if r − simulatedRound ≥ δ then halt                     {is process p_i faulty?}
```

Figure 1. Transformation algorithm (code for process p_i)

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. In the transformation of Fig. 1, 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 here 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 $R' = \langle I', F', ST', ME' \rangle$ 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. 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' $(r \leq r' \leq r + \delta - 2)$, 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 2 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). We denote by $rcvd[r]$ the messages of instance r received in phase r (we assume that any process sends in any phase of the underlying IC algorithm, the value it proposes to this instance).

Detailed Description. Consider any process p_i simulating state machine $\Pi_i = \langle s_i, T_i, O_i \rangle$, and any phase r of the simulated run $R = \langle I, F, ST, ME \rangle$ of Π. At the beginning of any phase r, es is the extended set of states and gathers all the possible states for p_i at the end of phase $r-2$. We describe the message generation and the simulation.

Message generation. At the beginning of phase r, p_i receives an input value $input = I(i,r)$, and computes a new extended set of states es' and the corresponding extended message em, which is a set of tuples. A tuple in em is of the form $\langle num(st), rec, num(st'), m \rangle$, and contains (i) the identifier $num(st)$ 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 $num(st')$ of the state st' of p_i at the beginning of phase r, such that $st' = T_i(st, rec, r-1)$, (iv) the message sent in round r, i.e., $m = O_i(st', I(i,r), r)$. 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 $st' = T_i(st, rec, r)$ (whenever p_i includes a new state st' in es', it associates a unique identifier $num(st')$ with st'), and the corresponding message $m = O_i(st', input, r)$. p_i sends em and the extended messages of other running IC instances in phase r.

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 $simulatedRound - 1$). Each process p_i maintains (1) the simulated state of machine Π_i at the end of round $simulatedRound - 1$ (denoted by $ss[simulatedRound-1]$), and (2) the identifier associated with the state currently simulated at each process p_j, at the end of round $simulatedRound - 1$, denoted by $sim[j]$.

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 $decision_i(simulatedRound)$ denotes the decided vector of messages at line 24, $failure_i(simulatedRound)$ the value of the variable $failure$ updated in line 25, and $trueRcvd_i(simulatedRound)$ the value of the variable $trueRcvd$ updated in line 31. Process p_i uses the decided vector $decision_i(simulatedRound)$ to update the simulated state of machine Π_i, i.e., p_i adds $simulatedRound$ in $states$ and computes $ss[simulatedRound]$. More precisely,

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

2. p_i updates $ss[simulatedRound]$ with the state $T_i(ss[simulatedRound-1], trueRcvd, simulatedRound)$.

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

Proposition 8 *The simulation Sim for a run of $\mathcal{T}(\Pi_s)$, $R = \langle I, F, ST, ME \rangle$ is defined by $R' = \langle I', F', ST', ME' \rangle$ as follows. Let p_i be a process in $correct(F)$ (p_i is such that, for all r, $F(i,r) = (\emptyset, \emptyset, 0)$, 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.*

- *(i)* $I' = I$.
- *(ii)* *if $p_j \in failure_i(r)$ then $F'(j,r) = (\Omega, \Omega, 1)$, otherwise $F'(j,r) = (\emptyset, \emptyset, 0)$.*
- *(iii)* *$ST'(i,0) = s_i$ and $ST'(i,r) = ss_i[r]$. For any process p_j (including p_i) not in $failure_i(r)$, $ST'(j,r)$ 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 $failure_i(r)$, $ST'(j,r) = ST'(j,r-1)$.*
- *(iv)* *if $p_j \in failure_i(r)$ then $ME'(j,r) = \perp$, otherwise $ME'(j,r) = trueRcvd(r)[j]$.*

Proposition 9 *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.* $Omission(n,t)$, $General(n,t)$*).*

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. 2 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 δ_f and τ_f the number of phases needed for all processes to respectively decide and terminate IC, in any run with at most f failures. We define the *phase complexity overhead* as the number of additional phases executed by the transformed protocol Π_w in M_w, w.r.t. the original protocol Π_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 at most 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 $nlog_2|\mathcal{I}|$-bit message per process, per phase, and per IC instance, hence any process sends a $n\tau_f log_2|\mathcal{I}|$-bit message in any phase, in any run with at most f failures. In the extended transformation, any process maintains at least $2^{n\tau_f}$ states for a round simulation. A state (tuple) is coded using $\sigma = 2log_2|\mathcal{S}| + (n+1)log_2|\mathcal{M}|$ bits, hence any process sends a $\tau_f n\sigma 2^{n\tau_f}$-bit message in any phase, in any run with f failures.

Proposition 10 ***(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* δ *phases.*

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

Our transformations require δ phases to simulate a round in PSR from any ω-$M(n,t)$. 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 ω-$M(n,t)$ requires δ 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 ω-$M(n,t)$, we need δ phases to solve this problem, otherwise we could solve binary consensus in less that δ phases (contradicting [6]). Assume some algorithm A solve problem P in less than δ phases. Consider algorithm A' defined as follows. If a process proposes 1, change it to ω otherwise (the proposed value is 0) let it unchanged, and execute algorithm A with the new proposed value. If the decision in A is ω 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 ω. By P-agreement, the difference between two decisions does not exceed $\omega - 1$, hence all decision values are 0 or ω. A' is a binary consensus algorithm and runs in less that δ, contradicting [6]. By definition, this problem is solved in $\omega - 1$ rounds in $(\omega-1)$-$M(n,t)$. In ω-$M(n,t)$, we have just shown that δ phases are required. For any bounded number of rounds r during which we observe a protocol Π, 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.

```
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   {is there 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 failure pattern has only all-or-nothing 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                          {is process p_i faulty?}
44:     halt                                                  {p_i does not perform any step}
```

Figure 2. Extended transformation algorithm (code for process p_i)

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 into the Byzantine model [11]. (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.) We need however to modify the definition of a simulation (Definition 2), that required some properties to hold over the complete set of processes Ω, which is not possible in the Byzantine model. These properties should rather refer to the set of correct processes in the original run (this is the same restriction as in [13]. The transformation algorithm in Fig. 1 has to be slightly modified: uniform agreement of IC is turned into non-uniform agreement (i.e., no two *correct* processes decide on two different vectors). Indeed, uniform agreement cannot be ensured in the Byzantine model. 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

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

[2] T. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM (JACM)*, 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 (DSN)*, pages 523–532, 2000.

[4] B. Charron-Bost and A. Schiper. Uniform consensus harder than consensus. Technical Report DSC/2000/028, EPFL, May 2000.

[5] C. Delporte-Gallet, H. Fauconnier, R. Guerraoui, and B. Pochon. Distributed programming for dummies: A shifting transformation technique. Technical Report IC/2003/49, EPFL, July 2003. Available at `http://ic2.epfl.ch/publications/documents/IC_TECH_REPORT_200349.pdf`.

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

[7] M. J. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM (JACM)*, 32(2):374–382, 1985.

[8] 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.

[9] I. Keidar and S. Rajsbaum. A simple proof of the uniform consensus lower bound. *Information Processing Letters (IPL)*, 85(1):47–52, 2002.

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

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

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

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

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

[15] M. Raynal. Consensus in synchronous systems: a concise guided tour. Technical Report 1467, IRISA, 2002.

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