

Synchrony Weakened by Message Adversaries vs Asynchrony Restricted by Failure Detectors

Michel Raynal
Institut Universitaire de France
& IRISA, Campus de Beaulieu
35042 Rennes Cedex, France
raynal@irisa.fr

Julien Stainer
IRISA, Université de Rennes
Campus de Beaulieu
35042 Rennes Cedex, France
julien.stainer@irisa.fr

ABSTRACT

A message adversary is a daemon that suppresses messages in round-based message-passing synchronous systems in which no process crashes. A property imposed on a message adversary defines a subset of messages that cannot be eliminated by the adversary. It has recently been shown that when a message adversary is constrained by a property denoted TOUR (for tournament), the corresponding synchronous system and the asynchronous crash-prone read/write system have the same computability power for task solvability.

This paper introduces new message adversary properties (denoted SOURCE and QUORUM), and shows that the synchronous round-based systems whose adversaries are constrained by these properties are characterizations of classical asynchronous crash-prone systems (1) in which processes communicate through atomic read/write registers or point-to-point message-passing, and (2) enriched with failure detectors such as Ω and Σ . Hence these properties characterize maximal adversaries, in the sense that they define strongest message adversaries equating classical asynchronous crash-prone systems. They consequently provide strong relations linking round-based synchrony weakened by message adversaries with asynchrony restricted with failure detectors. This not only enriches our understanding of the synchrony/asynchrony duality, but also allows for the establishment of a meaningful hierarchy of property-constrained message adversaries.

Categories and Subject Descriptors

D.1.3 [Programming techniques]: Concurrent Programming

F.1.1 [Computation by Abstract Devices]: Models of Computation, automata, relations among models

Keywords: Asynchronous system, Distributed computability, Failure detector, Message adversary, Message-passing model, Model equivalence, Ω , Process crash, Quorum, Read/write model, Round, Σ , Simulation, Source, Synchronous system, Task, Tournament, Wait-freedom.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

PODC'13, July 22–24, 2013, Montréal, Québec, Canada.

Copyright 2013 ACM 978-1-4503-2065-8/13/07 ...\$15.00.

1. INTRODUCTION

Message adversaries for synchronous message-passing systems. In a round-based message-passing synchronous system, processes communicate by exchanging messages at every round, and the synchrony assumption provided by the model guarantees that the messages sent at the beginning a round are received by their destination processes by the end of the corresponding round. Assuming that no process is faulty, the notion of a *message adversary* has been introduced in [22] (where it is called *mobile transmission failures*) to model messages losses and study their impact on the computability power of synchronous systems [22, 23].

Interestingly, the notion of constraining message deliveries has also been investigated in asynchronous systems, with distinct names, and in different contexts. As an example, asynchronous message patterns which allow failure detectors to be implemented despite asynchrony have been investigated in [10, 16]. The view of failure detectors as schedulers which encapsulate fairness assumptions can also be related to this approach [7, 17]. Recently, assumptions on message deliveries and message exchange patterns have been used to define new asynchronous computation models and study their computability power [6, 13, 18, 24]. The general idea, which underlies these works, consists in capturing the “weakest pattern of information exchange” that allows a family of problems to be solved despite failures.

Notation. The notation $\mathcal{SMP}_n[adv : AD]$ is used to denote a round-based synchronous system made up of n sequential processes whose communications are under the control of the adversary AD. While, in every round, each process sends a message to each other process, the power of the adversary AD consists in suppressing some of these messages (which are consequently never received).

According to their power, several classes of adversaries can be defined. $\mathcal{SMP}_n[adv : \emptyset]$ denotes a synchronous system in which the adversary has no power (it can suppress no message), while $\mathcal{SMP}_n[adv : \infty]$ denotes the synchronous system in which the adversary can suppress all messages. It is easy to see that, from a message adversary and computability point of view, $\mathcal{SMP}_n[adv : \emptyset]$ is the most powerful crash-free synchronous system model, while $\mathcal{SMP}_n[adv : \infty]$ is the weakest. More generally, the weaker the message adversary AD, the more powerful the system.

Asynchrony from synchrony. Informally, a *task* is a one-shot distributed computing problem where each process has a private input, and each process has to compute a local output such that each output may depend on the input values of the participating processes. The most famous and studied task is the consensus task.

Afek and Gafni addressed recently task solvability in synchronous message-passing systems weakened by message adversaries [1]¹. Let $\mathcal{ARW}_{n,n-1}[fd : \emptyset]$ denote the asynchronous read/write model where up to $(n - 1)$ processes may crash (“ $fd : \emptyset$ ” stands for “no failure detector”, see below; this is the classical read/write wait-free model [11]). Afek and Gafni’s main results are the following ones.

- Their first result concerns the adversary TOUR (for tournament) whose behavior is the following one. For each pair of processes p_i and p_j , and in each synchronous round, TOUR is allowed to suppress either the message sent by p_i to p_j or the message sent by p_j to p_i , but not both. The important result attached to TOUR is that $\mathcal{SMP}_n[adv : \text{TOUR}]$ and $\mathcal{ARW}_{n,n-1}[fd : \emptyset]$ have the same computability power for read/write wait-free solvable tasks.
- In addition to TOUR, two more adversaries, denoted TP and PAIRS, are described and it is shown that the three adversary-based distributed synchronous models $\mathcal{SMP}_n[adv : \text{TOUR}]$, $\mathcal{SMP}_n[adv : \text{TP}]$, and $\mathcal{SMP}_n[adv : \text{PAIRS}]$ are equivalent for task solvability. Moreover, $\mathcal{SMP}_n[adv : \text{PAIRS}]$ is used to show that, from a topology point of view, the protocol complex of PAIRS is a subdivided complex. This means that the message adversary PAIRS (and consequently also TOUR and TP) captures, in a very simple way, Herlihy and Shavit’s condition equating the read/write wait-free model with a complex subdivision [12].

Failure detectors for asynchronous crash-prone systems. Informally, a failure detector is a device that provides each process p_i with information on failures [4]. According to the quality and the type of information they provide, several classes of failure detectors can be defined (see [19] for an introductory survey).

The failure detectors denoted Ω and Σ are among the most important failure detectors. This is due to the following reasons: (1) Ω is the weakest failure detector that allows consensus to be solved in $\mathcal{ARW}_{n,n-1}[fd : \emptyset]$ [5, 15]; (2) Σ is the weakest failure detector that allows an atomic register to be implemented on top of $\mathcal{AMP}_{n,n-1}[fd : \emptyset]$ [8], where $\mathcal{AMP}_{n,n-1}[fd : \emptyset]$ denotes the classical asynchronous message-passing system where up to $(n - 1)$ processes may crash and every message is eventually received. “Weakest” means that any failure detector that allows to solve consensus (resp., implement a register) provides at least as much information on failures as the one provided by Ω (resp., Σ). Finally, the pair (Σ, Ω) is the weakest failure detector that allows consensus to be solved in $\mathcal{AMP}_{n,n-1}[fd : \emptyset]$ [5, 8].

Let FD denote a failure detector. $\mathcal{ARW}_{n,n-1}[fd : \text{FD}]$ denotes the asynchronous read/write model where up to $(n - 1)$ processes may crash, augmented with FD. Similarly, $\mathcal{AMP}_{n,n-1}[fd : \text{FD}]$ denotes $\mathcal{AMP}_{n,n-1}[fd : \emptyset]$ augmented with FD.

Content of the paper. Following Afek and Gafni’s seminal approach, the aim of the paper is to better understand and extend the message adversary approach, and capture its relations with asynchrony restricted by failure detectors. Considering the reliable synchronous round-based message-passing model ($\mathcal{SMP}_n[adv : \emptyset]$) as core model, the paper has the following contributions. Those concern (1) the crash-prone asynchronous read/write model, and (2) the crash-prone message-passing model, both enriched with failure detectors. These contributions are described in the hierarchy depicted in Figure 1. $A \simeq_M B$ means that the computing model A

can be simulated in the model B and vice-versa. $A \simeq_T B$ means that any task that can be solved in the model A , can be solved in the model B and vice-versa. An arrow from A to B means that the model A is stronger than the model B , but not vice-versa. These arrows follow from known results (e.g., $\mathcal{ARW}_{n,n-1}[fd : \Omega]$ is stronger than both $\mathcal{ARW}_{n,n-1}[fd : \emptyset]$ and $\mathcal{AMP}_{n,n-1}[fd : \Omega]$). Let us observe that, as they are failure-free, the system models $\mathcal{SMP}_n[adv : \emptyset]$, $\mathcal{ARW}_{n,0}[fd : \emptyset]$, and $\mathcal{AMP}_{n,0}[fd : \emptyset]$, are computationally equivalent (first line of the figure).

- Starting from the fact that the property TOUR (for tournament) captures the constraint on message delivery such that $\mathcal{SMP}_n[adv : \text{TOUR}] \simeq_T \mathcal{ARW}_{n,n-1}[fd : \emptyset]$, Section 3 focuses on the properties of a message adversary which allow to enrich $\mathcal{SMP}_n[adv : \text{TOUR}]$ to obtain $\mathcal{ARW}_{n,n-1}[fd : \Omega]$. To that end (1) it presents a new message delivery property, denoted SOURCE, and (2) shows that $\mathcal{SMP}_n[adv : \text{SOURCE}, \text{TOUR}]$ and $\mathcal{ARW}_{n,n-1}[fd : \Omega]$ are equivalent for task solvability. It follows that SOURCE is a minimal requirement that has to be added to $\mathcal{SMP}_n[adv : \text{TOUR}]$ in order to proceed from the model $\mathcal{SMP}_n[adv : \text{TOUR}]$ to the model $\mathcal{ARW}_{n,n-1}[fd : \Omega]$.
- Then Section 4 shows that, by weakening $\mathcal{SMP}_n[adv : \text{SOURCE}, \text{TOUR}]$ into $\mathcal{SMP}_n[adv : \text{SOURCE}]$, the resulting synchronous message-passing system is such that we have $\mathcal{SMP}_n[adv : \text{SOURCE}] \simeq_T \mathcal{AMP}_{n,n-1}[fd : \Omega]$. It follows that SOURCE captures the weakest property on message delivery that an adversary AD has to satisfy so that any task solvable in $\mathcal{AMP}_{n,n-1}[fd : \Omega]$ can be solved in $\mathcal{SMP}_n[adv : \text{AD}]$.² Said differently, when considering tasks solvability in crash-prone asynchronous systems enriched with Ω , what allows going from “message-passing” communication to “read/write” communication is characterized by the property TOUR from a message adversary point of view in a synchronous system (vertical arrow on the right of Figure 1).
- Then Section 5 focuses on a new message delivery property denoted QUORUM, and shows that a message adversary constrained by this property captures in $\mathcal{SMP}_n[adv : \emptyset]$ the same computability power (from a task point of view) as the one added by the failure detector Σ to $\mathcal{AMP}_{n,n-1}[fd : \emptyset]$. To that end, it shows that $\mathcal{SMP}_n[adv : \text{QUORUM}] \simeq_T \mathcal{AMP}_{n,n-1}[fd : \Sigma]$.
- Finally, as a consequence of the previous results, Section 6 shows that the properties SOURCE + QUORUM characterize the pair of failure detectors $\Sigma + \Omega$, i.e., $\mathcal{SMP}_n[adv : \text{SOURCE}, \text{QUORUM}] \simeq_T \mathcal{AMP}_{n,n-1}[fd : \Sigma, \Omega]$.

As indicated, the paper provides message adversary-based characterizations of failure detectors for both read/write and message-passing crash-prone asynchronous systems. The aim of these results is to enrich our understanding of both message adversaries used to weaken communication in synchronous systems and failure detectors used to enrich asynchronous crash-prone systems. They complement the results of [1] and exhibit strong intimate relations linking synchrony, message losses, and round-based model, on the one side, with asynchrony, process crashes, and failure detectors, on the other side. Interestingly, this seems to show that $\mathcal{SMP}_n[adv : \emptyset]$ (basic reliable synchronous round-based model)

²As shown in [1] with the properties TOUR, TP, and PAIRS, several properties can be equivalent (i.e., each one can be implemented in $\mathcal{SMP}_n[adv : \emptyset]$ under the control of an adversary constrained by any other one). Hence, if a property P is the “weakest”, so are the properties equivalent to P .

¹A close approach focused only on consensus is presented in [3].

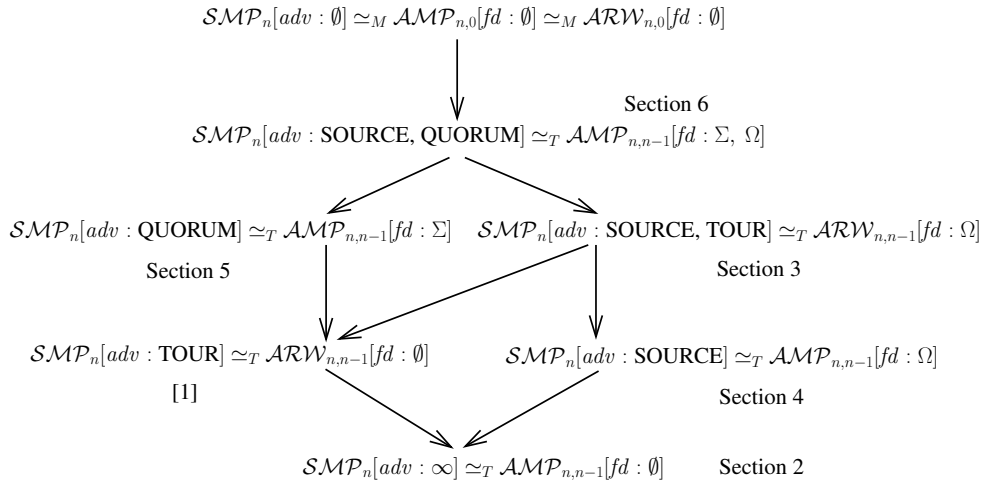


Figure 1: A message adversary hierarchy based on task equivalence and failure detectors

and the notion of a message adversary are central in the quest for a universal model of distributed computing.

Roadmap. The paper is composed of 7 sections. Section 2 presents base models, message adversaries, and failure detectors. Section 3 introduces the property SOURCE on message deliveries, and show that it characterizes the failure detector Ω in read/write systems. Section 4 shows that, taken alone, the property SOURCE characterizes the failure detector Ω in asynchronous message-passing systems. Section 5 and Section 6 introduce the property QUORUM and show that it characterizes the failure detector Σ . Finally, Section 7 concludes the paper. Due to page limitation, all missing proofs can be found in [21].

2. MODELS, MESSAGE ADVERSARIES, FAILURE DETECTORS, TASKS

2.1 Base Computation Models

The base computation models relevant to this paper have been presented in the introduction. They are (1) the reliable round-based synchronous model $SMP_n[adv : \emptyset]$ possibly weakened with a message adversary AD, and (b) the crash-prone asynchronous models $ARW_{n,n-1}[fd : \emptyset]$ (basic *read/write wait-free* model [11]) and $AMP_{n,n-1}[fd : \emptyset]$, both possibly enriched with a failure detector FD.

2.2 Message Adversary, Message Graphs, and Dynamic Graphs in Synchronous Systems

Message adversary. Given a run of a synchronous system, a message adversary suppresses messages sent by processes. A property associated with a message adversary restricts its power by specifying messages which cannot be suppressed. A message adversary is consequently defined by a set of properties which constrain its behavior.

Message graphs associated with the rounds of a synchronous system. Given a message adversary AD, and a round r of a run of a synchronous system, let \mathcal{G}^r be the directed graph (as defined in [1]), whose vertices are the process identities, and such that there is an edge from i to j iff the adversary AD does not suppress the

message sent by p_i to p_j at round r . We consider the following definition associated with each graph \mathcal{G}^r .

- $i \xrightarrow{r} j$ means that the directed edge (i, j) belongs to \mathcal{G}^r (at round r , the message from p_i to p_j is not removed by the adversary).

The property TOUR. As indicated in the introduction, the property TOUR [1] restricts the behavior of a message adversary as follows. For any r , and any pair of processes (p_i, p_j) , \mathcal{G}^r contains the directed edge (i, j) or the directed edge (j, i) or both. This means that, at every round, the adversary cannot suppress both the messages sent to each other by two processes. Hence, the graphs \mathcal{G}^r associated with the rounds r of a run in $SMP_n[adv : TOUR]$ are such that:

$$\forall r \geq 1 : \forall (i, j) : (i \xrightarrow{r} j) \vee (j \xrightarrow{r} i).$$

Strongly/weakly correct processes in a synchronous run. The aim of this section is to introduce the notion of a *strongly correct* process which captures the processes whose an infinite number of messages are received (directly or indirectly) by any other process [20]. Such a notion is defined as follows.

- $i \xrightarrow{r} j$ means that there is a directed path starting from p_i and leading to p_j in a dynamically defined sequence of message graphs starting at a round $\geq r$. More formally, $\exists k \geq 0, \exists r_1 < \dots < r_k, \exists \lambda_0, \lambda_1, \dots, \lambda_k \in \{1, \dots, n\} : (r_1 \geq r) \wedge (\lambda_0 = i \wedge \lambda_k = j) \wedge (\forall m \in \{1, \dots, k\} : \lambda_{m-1} \xrightarrow{r_m} \lambda_m)$.
- $i \xrightarrow{\infty} j \stackrel{def}{=} (\forall r > 0 : i \xrightarrow{r} j)$. Hence, $i \xrightarrow{\infty} j$ means that, whatever r , there is eventually a directed path starting at p_i at a round $\geq r$ and finishing at p_j in the dynamically defined sequence of message graphs.
- $(i \xrightarrow{\infty} j) \Leftrightarrow (i \xrightarrow{\infty} j \wedge j \xrightarrow{\infty} i)$. Assuming each process always receive its own messages, this relation is reflexive, symmetric, and transitive. Hence, it is an equivalence relation.
- Let G be the graph whose vertices are $\{1, \dots, n\}$ and directed edges are defined by the relation $\xrightarrow{\infty}$; let $SC(G)$ be the graph of its strongly connected components. If $SC(G)$ has a single

vertex X with no input edge, the processes in X are called *strongly correct* processes, while the processes in $\{1, \dots, n\} \setminus X$ are called *weakly correct*. If $SC(G)$ has several vertices with no input edge, all processes are weakly correct.

Let SC denote the (possibly empty) set of strongly correct processes in a synchronous round-based system under the control of a message adversary.

2.3 Failure Detectors in Asynchr. Systems

While a message adversary weakens a synchronous round-based system (made up of reliable processes) by suppressing messages, a failure detector enriches an asynchronous system where no message is lost but where processes may suffer crash failures. Informally, a failure detector is a device that provides each process p_i with a read-only local variable xx_i containing (possibly unreliable) information on process crashes [4]. This paper considers two failure detectors. Let τ denote any time instant; xx_i^τ denotes the value of xx_i at time τ . This time notion, which is used in the definition of a failure detector, is not accessible to the processes. The identity of a process p_i is i . Given a run, a process that crashes is said to be *faulty* in that run, otherwise it is *correct*. Let \mathcal{C} denote the sets of identities of the correct processes.

- Ω is called an *eventual leader* failure detector [5]. In the system models $\mathcal{ARW}_{n,n-1}[fd : \Omega]$ or $\mathcal{AMP}_{n,n-1}[fd : \Omega]$, each process p_i is endowed with a local variable $xx_i = leader_i$ that always contains a (possibly changing) process identity. Moreover, there is an unknown but finite time τ and a process identity $\ell \in \mathcal{C}$ such that $\forall \tau' \geq \tau : (i \in \mathcal{C}) \Rightarrow (leader_i^{\tau'} = \ell)$.
- Σ is called a *quorum* failure detector [8]. In the system model $\mathcal{AMP}_{n,n-1}[fd : \Sigma]$, each process p_i is endowed with a local variable $xx_i = qr_i$ that always contains a non-empty set of process identities and is such that (1) $\forall \tau, \tau', \forall i, j : qr_i^\tau \cap qr_j^{\tau'} \neq \emptyset$ (intersection property), and (2) $\forall i \in \mathcal{C} : \exists \tau : \forall \tau' \geq \tau : qr_i^{\tau'} \subseteq \mathcal{C}$ (liveness property).

2.4 Tasks

A task is a one-shot computation problem specified in terms of an input/output relation Δ . Each process starts with a private input value and must eventually compute a private output value, which is a function of the inputs of the participating processes. From an external observer point of view, an input vector $I[1..n]$ specifies the input value $I[i] = v_i$ of each process p_i . Similarly, an output vector $O[1..n]$ specifies a result value $O[j]$ for each process p_j .

A task is defined by a set of input vectors and a relation Δ which describes which output vectors are correct for each input vector I . More precisely, for each valid input vector I , the values computed by the processes must be such that there is an output vector $O \in \Delta(I)$ such that, for each j , $O[j]$ is the value computed by p_j ; moreover, if no value is computed by p_j , it is because p_j has crashed during the computation. (A formal introduction to tasks can be found in [12].)

THEOREM 1. *A task T can be solved in $\mathcal{SMP}_n[adv : \infty]$ iff it can be solved in $\mathcal{AMP}_{n,n-1}[fd : \emptyset]$.*

3. THE PAIR SOURCE + TOUR CAPTURES EXACTLY Ω IN $\mathcal{ARW}_{n,n-1}[fd : \emptyset]$

This section shows that the computing models $\mathcal{SMP}_n[adv : \text{SOURCE, TOUR}]$ and $\mathcal{ARW}_{n,n-1}[fd : \Omega]$ have the same computational power for tasks.

3.1 The Property SOURCE

This property is defined as follows:

$$\exists s \in \{1, \dots, n\} : \exists r_0 \geq 1 : \\ \forall r \geq r_0 : \forall i \in \{1, \dots, n\} : (s \xrightarrow{r} i).$$

This statement means that, in each run of $\mathcal{SMP}_n[adv : \text{SOURCE}]$, there are a process p_s and a round r_0 , such that, at every round $r \geq r_0$, the adversary does not suppress the message sent by p_s to the other processes.

3.2 From the Read/Write Model $\mathcal{ARW}_{n,n-1}[fd : \Omega]$ to $\mathcal{SMP}_n[adv : \text{SOURCE, TOUR}]$

This section presents a simulation of the synchronous model $\mathcal{SMP}_n[adv : \text{SOURCE, TOUR}]$ on top of $\mathcal{ARW}_{n,n-1}[fd : \Omega]$ such that, any task that can be solved in $\mathcal{ARW}_{n,n-1}[fd : \Omega]$ can be solved in $\mathcal{SMP}_n[adv : \text{SOURCE, TOUR}]$.

Global and local variables of the simulation. The simulation uses a shared 3-dimensional array MEM where $MEM[i][r][j]$ is an atomic read/write register written by p_i and read by p_j . This register contains the message sent by p_i to p_j in round r of the simulation of $\mathcal{SMP}_n[adv : \text{SOURCE, TOUR}]$; \perp is a default value used to indicate that no message has yet been written or the corresponding message has been suppressed by the adversary. The local variable r_i simulates the current round number of $\mathcal{SMP}_n[adv : \text{SOURCE, TOUR}]$, while ls_state_i represents the local simulation state. The local variable $msgs_to_send_i[1..n]$ contains the messages that p_i will send to each other process during the next simulated round ($msgs_to_send_i[j]$ contains the message for p_j). $leader_i$ is the read-only local variable containing the current local output of Ω .

The simulation is locally defined by the function `simulate()`. It takes as input parameters the current local state of the simulation and the messages received from the other processes at the current round. It modifies accordingly the local simulation state and computes the messages that will be sent to the other processes during the next round.

Simulation algorithm. The local simulation algorithm is described in Figure 2. The local simulator of process p_i first proceeds to the next round (line 5) and waits until its current leader has sent it a message (predicate $MEM[leader_i][r_i][i] \neq \perp$) or it is its own leader (lines 6-8). When this occurs, the simulator writes in $MEM[i][r_i]$ the messages sent by p_i at the current round (line 9). Then, p_i consumes messages (line 10), and uses them to modify its local simulation state and compute the message it will send during the next round (line 11).

LEMMA 1. *If a task can be solved in the synchronous model $\mathcal{SMP}_n[adv : \text{SOURCE, TOUR}]$, it can be solved in the enriched read/write model $\mathcal{ARW}_{n,n-1}[fd : \Omega]$.*

3.3 From $\mathcal{SMP}_n[adv : \text{SOURCE, TOUR}]$ to the Read/Write Model $\mathcal{ARW}_{n,n-1}[fd : \Omega]$

This section presents a simulation of $\mathcal{ARW}_{n,n-1}[fd : \Omega]$ on top of $\mathcal{SMP}_n[adv : \text{SOURCE, TOUR}]$ such that, any task that can be solved in $\mathcal{SMP}_n[adv : \text{SOURCE, TOUR}]$ can be solved in $\mathcal{ARW}_{n,n-1}[fd : \Omega]$. This simulation has the same structure as the simulation of $\mathcal{ARW}_{n,n-1}[fd : \emptyset]$ on top of $\mathcal{SMP}_n[adv : \emptyset]$ described in [1]. Basically, it adds to it the management of the local variables $missed_i$ (defined below) from which Ω is extracted.

Global and local variables of the simulation. The shared memory of $\mathcal{ARW}_{n,n-1}[fd : \Omega]$ is made up of an array of single-


```

initialization:
(1)  $r_i \leftarrow 0$ ;
(2)  $ls\_state_i \leftarrow$  initial state of the local simulated algorithm;
(3)  $msgs\_to\_send_i[1..n] \leftarrow$  initial msgs to send to each proc.;
(4)  $\forall r > 0 : MEM[i][r][1..n]$  init to  $[\perp, \dots, \perp]$ .

repeat forever
(5)  $r_i \leftarrow r_i + 1$ ;
(6) repeat  $leader\_val_i \leftarrow MEM[leader_i][r_i][i]$ 
(7)   until  $(leader\_val_i \neq \perp) \vee (leader_i = i)$ 
(8) end repeat;
(9)  $MEM[i][r_i] \leftarrow msgs\_to\_send_i$ ;
(10)  $rec\_msgs_i[1..n] \leftarrow MEM[1..n][r_i][i]$ ;
(11)  $(msgs\_to\_send_i, ls\_state_i)$ 
       $\leftarrow simulate(ls\_state_i, rec\_msgs_i)$ 
end repeat.

```

Figure 2: From the enriched read/write model $\mathcal{ARW}_{n,n-1}[fd : \Omega]$ to $\mathcal{SMP}_n[adv : \text{SOURCE, TOUR}]$

writer/multi-reader atomic registers $MEM[1..n]$ such that only p_i can write $MEM[i]$. The simulation associates a sequence number with each read or write operation of a simulated process p_i . To simplify notations, a read of $MEM[\ell]$ by p_i is denoted $read_i(\ell)$ and a write of v into $MEM[i]$ is denoted $write_i(v)$.

As in the previous simulation, the procedure $simulate()$ is used to locally simulate the behavior of p_i from its current step until its next invocation of a communication operation (i.e., a read or a write of the simulated shared memory). The simulation stops just before this invocation. It takes as input parameters the current local state of p_i (ls_state_i) and the last value read from the shared memory by p_i . This value, saved in $read_value_i$ (and initialized to \perp), is meaningless if the operation is a write. The local variable $next_op_i$ contains p_i 's next read or write operation to be simulated.

The local variable $view_i$ contains all the read/write operations issued by the processes and known by p_i . Such an operation is represented by a triple $(j, seq_nb, next_op)$. The simulation algorithm is a full information algorithm and consequently the set $view_i$ increases forever.

The local variable $informed_i$ contains the set of processes which, to p_i knowledge, know the last read/write operation it is currently simulating. Finally, the set $missed_i$ (from which Ω is built) contains pairs (k, r) whose meaning is the following: $((k, r) \in missed_i) \Rightarrow$ there is at least one process that, during round r of the simulation, has not received and delivered the message sent by (the simulator of) p_k during that round.

Simulation algorithm. The simulation algorithm is described in Figure 3. When it starts a new round, the simulator of p_i sends its control local state, i.e., the triple $(i, view_i, missed_i)$ to each other process (line 5). Then (lines 6-10), it considers all the messages it has received during the current round r , and updates accordingly rec_msg_i and $missed_i$.

Lines 11-12 locally implement Ω (see below). The local variable $informed_i$ is then updated to take into account what has been learnt from the messages just received. Let us notice (line 13) that it follows from TOUR that $(j \notin rec_from_i) \Rightarrow p_j$ has received p_i 's round r message.

Then (the simulator of) p_i executes rounds in $\mathcal{SMP}_n[adv : \text{SOURCE, TOUR}]$ until it learns that (the simulators of) all the processes know its last read/write operation (line 15). Then, it invokes $simulate(ls_state_i, read_value_i)$ (line 16). If its (simulated) shared memory operation is a read, the value $read_value_i$ is the value obtained by this read operation. Otherwise (the simulated

operation is a write of p_i), $read_value_i$ is useless. As already indicated, the invocation of $simulate(ls_state_i, read_value_i)$ simulates then the behavior of p_i until its next read/write operation.

If the operation at which the local simulation stopped is a read of $MEM[\ell]$ (line 17), the local simulator computes, and deposits in $read_value_i$, the value that will be associated with this read (line 18-22). If p_ℓ has not issued a write, $read_value_i$ is set to the default value \perp (line 19). Otherwise, $read_value_i$ is set to the last value written by p_ℓ (line 18-21). Then, whatever the next operation (read or write) of p_i , the local simulator associates a sequence number with it and adds the triple $(i, seq_nb_i, next_op_i)$ to $view_i$ (line 24-25). Moreover, as its scope is the simulation of $next_op_i$, the set $informed_i$ is reset to $\{i\}$.

As previously indicated, the current value (kept in ld_i) of the read-only variable $leader_i$, which locally implements Ω , is computed from the set $missed_i$ at lines 11-12. The simulator of p_i (1) computes, for each p_j , the set of rounds at which at least one simulator has not received the round r message sent by p_j 's simulator (these are messages suppressed by the adversary); then (2) it associates with each p_j the cardinality of the previous set; and finally, (3) it considers the process p_ℓ for which the adversary has suppressed the less messages (if there are several such processes, ties are solved by using the total order on process identities).

LEMMA 2. *If a task can be solved in $\mathcal{ARW}_{n,n-1}[fd : \Omega]$, it can be solved in $\mathcal{SMP}_n[adv : \text{SOURCE, TOUR}]$.*

3.4 The Pair of Adversaries SOURCE + TOUR Captures Exactly Ω in the Read/Write Model $\mathcal{ARW}_{n,n-1}[fd : \emptyset]$

THEOREM 2. *A task can be solved in the synchronous model $\mathcal{SMP}_n[adv : \text{SOURCE, TOUR}]$ iff it can be solved in the enriched read/write model $\mathcal{ARW}_{n,n-1}[fd : \Omega]$.*

Proof The proof follows immediately from Lemma 1 and Lemma 2.

□_{Theorem 2}

Remark. Let us remark that it is not possible to conclude from the previous theorem and the fact that Ω is the weakest failure detector to solve consensus in $\mathcal{ARW}_{n,n-1}[fd : \emptyset]$, that the property SOURCE + TOUR defines a weakest message adversary AD allowing consensus to be solved in $\mathcal{SMP}_n[adv : \text{AD}]$. It remains possible that a property AD weaker than SOURCE + TOUR allows consensus to be solved in $\mathcal{SMP}_n[adv : \text{AD}]$. Said differently nothing allows us to claim that the “granularity” on the properties which can be defined to constrain message adversaries is the same as the “granularity” on the information on failures provided by failure detectors. (The approach investigated in [3] is a step in that direction.)

4. SOURCE CAPTURES EXACTLY THE FAILURE DETECTOR Ω IN $\mathcal{AMP}_{n,n-1}[fd : \emptyset]$

4.1 From $\mathcal{AMP}_{n,n-1}[fd : \Omega]$ to $\mathcal{SMP}_n[adv : \text{SOURCE}]$

The algorithm described in Figure 5 presents a simulation (for tasks) of $\mathcal{SMP}_n[adv : \text{SOURCE}]$ on top of $\mathcal{AMP}_{n,n-1}[fd : \Omega]$. Its principles are close to the ones of the simulation of Figure 2. The algorithm ensures that the eventual leader p_ℓ satisfies the property SOURCE. Hence, there are strongly correct processes and the eventual leader is one of them. The aim of the simulation algorithm is then to eventually withdraw all the messages except the ones from the leader.

```

initialization:
(1)  $ls\_state_i \leftarrow$  initial state of the local simulated algorithm;  $read\_value_i \leftarrow \perp$ ;
(2)  $(next\_op_i, ls\_state_i) \leftarrow simulate(ls\_state_i, read\_value_i)$ ;
(3)  $seq\_nb_i \leftarrow 1$ ;  $informed_i \leftarrow \{i\}$ ;  $missed_i \leftarrow \emptyset$ ;
(4)  $view_i \leftarrow \{(i, seq\_nb_i, next\_op_i)\}$ .

round  $r = 1, 2, \dots$  do:
(5)  $send(i, view_i, missed_i)$  to each other process;
(6)  $rec\_msgs_i \leftarrow$  set of triples  $(j, view\_j, missed\_j)$  received during this round;
(7)  $view_i \leftarrow view_i \cup \left( \bigcup_{(j, view\_j, missed\_j) \in rec\_msgs_i} view\_j \right)$ ;
(8)  $missed_i \leftarrow missed_i \cup \left( \bigcup_{(j, view\_j, missed\_j) \in rec\_msgs_i} missed\_j \right)$ ;
(9)  $rec\_from_i \leftarrow \{j \in \{1, \dots, n\} : \exists (j, view\_j, missed\_j) \in rec\_msgs_i\} \cup \{i\}$ ;
(10)  $missed_i \leftarrow missed_i \cup \{(k, r) : k \in \{1, \dots, n\} \setminus rec\_from_i\}$ ;
(11)  $min\_missed_i \leftarrow \min\{|\{r' : (j, r') \in missed_i\}|, j \in \{1, \dots, n\}\}$ ;
(12)  $ld_i \leftarrow \min\{j : |\{r' : (j, r') \in missed_i\}| = min\_missed_i\}$ ;
(13)  $informed_i \leftarrow informed_i \cup (\{1, \dots, n\} \setminus rec\_from_i)$ 
(14)  $\cup \{j \in rec\_from_i : (i, seq\_nb_i, next\_op_i) \in view_j\}$ ;
(15) if ( $informed_i = \{1, \dots, n\}$ ) then
(16)  $(next\_op_i, ls\_state_i) \leftarrow simulate(ls\_state_i, read\_value_i)$ ;
(17) if ( $next\_op_i = read_i(\ell)$ ) then
(18) if ( $\nexists (\ell, -, write_\ell(-)) \in view_i$ )
(19) then  $read\_value_i \leftarrow \perp$ 
(20) else  $max\_snl_i \leftarrow \max\{sn_\ell, (\ell, sn_\ell, write_\ell(-)) \in view_i\}$ ;
(21)  $read\_value_i \leftarrow v_\ell : (\ell, max\_snl_i, write_\ell(v_\ell)) \in view_i$ 
(22) end if
(23) end if;
(24)  $seq\_nb_i \leftarrow seq\_nb_i + 1$ ;  $informed_i \leftarrow \{i\}$ ;
(25)  $view_i \leftarrow view_i \cup \{(i, seq\_nb_i, next\_op_i)\}$ 
(26) end if.

when  $leader_i$  is read: return  $(ld_i)$ .

```

Figure 3: Simulation of $ARW_{n,n-1}[fd : \Omega]$ in $SMP_n[adv : \text{SOURCE}, \text{TOUR}]$

Local variables of the simulation. As in the previous simulations, r_i is the locally simulated round number; $msgs_to_send_i[j]$ (initialized to \perp) contains the next simulated message to be sent to p_j ; $rec_msgs_i[r]$ contains the simulated messages received at round r ; $sim_rec_msgs_i[x]$ contains the message received from the process p_x currently considered as the leader by p_i ; $leader_i$ is the read-only variable provided by Ω .

```

(1)  $r_i \leftarrow 0$ ;  $sim\_rec\_msgs_i[1, \dots, n] \leftarrow [\perp, \dots, \perp]$ ;
(2)  $(msgs\_to\_send_i[1, \dots, n], ls\_state_i) \leftarrow simulate(sim\_rec\_msgs_i)$ ;
(3) for each  $r > 0$ 
(4) do  $rec\_msgs_i[r][1, \dots, n] \leftarrow [\perp, \dots, \perp]$  end for;
(5) repeat forever
(6)  $r_i \leftarrow r_i + 1$ ;
(7) for each  $j \in \{1, \dots, n\}$ 
(8) do  $send(r_i, msgs\_to\_send_i[j])$  to  $p_j$  end for;
(9) repeat  $cur\_ld_i \leftarrow leader_i$ 
(10) until ( $cur\_ld_i = i \vee rec\_msgs_i[r_i][cur\_ld_i] \neq \perp$ )
(11) end repeat;
(12)  $sim\_rec\_msgs_i[cur\_ld_i] \leftarrow rec\_msgs_i[r_i][cur\_ld_i]$ ;
(13)  $(msgs\_to\_send_i[1, \dots, n], ls\_state_i) \leftarrow simulate(sim\_rec\_msgs_i)$ ;
(14)  $sim\_rec\_msgs_i[1, \dots, n] \leftarrow [\perp, \dots, \perp]$ 
(15) end repeat.

when  $(r, m)$  received from  $p_j$ :  $rec\_msgs_i[r][j] \leftarrow m$ .

```

Figure 5: From $AMP_{n,n-1}[fd : \Omega]$ to $SMP_n[adv : \text{SOURCE}, \text{TOUR}]$

Simulation algorithm. The procedure $simulate()$ takes as input parameter the simulated messages received by p_i at the current round, and simulates the local algorithm until the next sending of messages by p_i . This procedure returns the simulated messages to be sent at the beginning of the next round.

After the initialization stage (lines 1-3), the local simulator of p_i enters a loop whose each body execution simulates a round of the synchronous system. It first sends the messages that p_i has to send at the current round (line 6). Then it waits until it has received a message from its current leader or it is its own leader (lines 7-9). When this occurs, it retrieves the message sent by its current leader (line 10) and invokes the procedure $simulate()$ with this message as input parameter, before proceeding to the simulation of the next synchronous round.

LEMMA 3. *If a task can be solved in $SMP_n[adv : \text{SOURCE}]$, it can be solved in $AMP_{n,n-1}[fd : \Omega]$.*

Proof The simulator concerned by this lemma is the one described in Figure 5. Let us first show that no correct process remains blocked forever in the loop lines of 7-9. Indeed, there is a finite time τ after which an eventual leader (say p_ℓ) is elected by Ω at each process. It then follows from the first part of the predicate of line 8 that p_ℓ cannot remain blocked at line 8, and consequently executes rounds forever. Moreover, as its messages are eventually received at each round by all correct processes, it follows that there is a time after which the second part of the predicate of line 8 is always satisfied by these processes. Consequently, none of them can remain blocked forever at line 8.

```

initialization:
(1)  $ls\_state_i \leftarrow$  initial state of the local simulated algorithm;
(2)  $msgs\_to\_rec_i \leftarrow \emptyset$ ;  $msgs\_received_i \leftarrow \emptyset$ ;
(3)  $(msgs\_to\_send_i, ls\_state_i) \leftarrow \text{simulate}(ls\_state_i, msgs\_to\_rec_i)$ ;
(4)  $view_i \leftarrow msgs\_to\_send_i$ ;  $missed_i \leftarrow \emptyset$ ;  $ld_i \leftarrow i$ .

round  $r = 1, 2, \dots$  do:
(5)  $\text{send}(i, view_i, missed_i)$  to each other process;
(6)  $rec\_msgs_i \leftarrow$  set of triples  $(j, view\_j, missed\_j)$  received during this round;
(7)  $view_i \leftarrow view_i \cup \left( \bigcup_{(j, view\_j, missed\_j) \in rec\_msgs_i} view\_j \right)$ ;
(8)  $missed_i \leftarrow missed_i \cup \left( \bigcup_{(j, view\_j, missed\_j) \in rec\_msgs_i} missed\_j \right)$ ;
(9)  $rec\_from_i \leftarrow \{j \in \{1, \dots, n\} : \exists (j, view\_j, missed\_j) \in rec\_msgs_i\} \cup \{i\}$ ;
(10)  $missed_i \leftarrow missed_i \cup \{(k, r) : k \in \{1, \dots, n\} \setminus rec\_from_i\}$ ;
(11)  $min\_missed_i \leftarrow \min\{|\{r' : (j, r') \in missed_i\}|, j \in \{1, \dots, n\}\}$ ;
(12)  $ld_i \leftarrow \min\{j : |\{r' : (j, r') \in missed_i\}| = min\_missed_i\}$ ;
(13) if  $(ld_i \in rec\_from_i)$  then
(14)   let  $view\_ld_i$  be such that  $(ld_i, view\_ld_i, missed\_ld_i) \in rec\_msgs_i$ ;
(15)    $msgs\_to\_rec_i \leftarrow msgs\_to\_rec_i \cup \{(ld_i, i, m) : (ld_i, i, m) \in view\_ld_i\}$ ;
(16)   if  $(msgs\_to\_send_i \subseteq view\_ld_i)$  then
(17)      $(msgs\_to\_send_i, ls\_state_i) \leftarrow \text{simulate}(ls\_state_i, msgs\_to\_rec_i \setminus msgs\_received_i)$ ;
(18)      $msgs\_received_i \leftarrow msgs\_to\_rec_i$ ;  $view_i \leftarrow view_i \cup msgs\_to\_send_i$ 
(19)   end if
(20) end if.

when leaderi is read: return  $(ld_i)$ .

```

Figure 4: Simulation of $\mathcal{AMP}_{n,n-1}[fd : \Omega]$ in $\mathcal{SMP}_n[adv : \text{SOURCE}]$

The previous reasoning shows also that the eventual leader elected by Ω behaves as a source, and consequently the property SOURCE is satisfied in the simulated synchronous system. $\square_{\text{Lemma 3}}$

4.2 From $\mathcal{SMP}_n[adv : \text{SOURCE}]$ to $\mathcal{AMP}_{n,n-1}[fd : \Omega]$

The simulation algorithm is described in Figure 4. It is similar to the algorithm of Figure 3 (which simulates $\mathcal{ARW}_{n,n-1}[fd : \Omega]$ on top of $\mathcal{SMP}_n[adv : \text{SOURCE}, \text{TOUR}]$).

Local variables of the simulation. The local variables ls_state_i , $view_i$, rec_from_i , and $missed_i$ have the same meaning as in Figure 3. The local variable denoted $msgs_to_rec_i$ contains messages to be consumed by the simulated process (it corresponds to $read_value_i$ in Figure 3). The variable $msgs_to_send_i$ contains the messages to be sent in the next simulation round (it corresponds to $next_op_i$ in Figure 3). The variable $msgs_received_i$ is a new variable containing the messages already received by the simulated process p_i . Finally, ld_i is the local variable containing the current local value of Ω built by the algorithm.

Simulation algorithm. As in the simulation of Figure 3, lines 1-4 are an initialization stage. Similarly to previous simulations, the procedure $\text{simulate}()$ locally simulates the process p_i . It takes messages to be consumed by p_i as input parameter and returns the next set of messages to be sent.

The simulation algorithm is a full information algorithm. During each simulation round r , the simulator of p_i first sends its control local state to each other process, and waits for the same information from them (lines 5-6). Then, according to the messages it has received during the current round, it updates $view_i$, $missed_i$, and rec_from_i (lines 7-10). As in Figure 3, it also computes the identity ld_i of its current candidate to be the eventual leader (lines 11-12).

If a simulation message has been received from the process p_{ld_i} , the simulator of p_i strives to make p_i progress. It considers the last message sent by p_{ld_i} to p_i (triple (ld_i, i, m)), and adds it to the set $msgs_to_rec_i$ (lines 14-15). Then, if the messages p_i has to send are known by its current leader p_{ld_i} (line 16), the procedure $\text{simulate}()$ is invoked to make p_i progress (line 17), and the local control variables $msgs_received_i$ and $view_i$ are updated accordingly (line 18).

LEMMA 4. *If a task can be solved in $\mathcal{AMP}_{n,n-1}[fd : \Omega]$, it can be solved in $\mathcal{SMP}_n[adv : \text{SOURCE}]$.*

Proof The simulator concerned by this lemma is the one described in Figure 4. Preliminary definition on simulators in $\mathcal{SMP}_n[adv : \text{SOURCE}]$.

Let S be the set of processes which satisfy the property SOURCE. As, by assumption there at least one source, we have $S \neq \emptyset$. Moreover, due the definition of the set \mathcal{SC} of strongly correct simulators we have $S \subseteq \mathcal{SC}$. Let S' be the set of processes which, albeit they are not necessarily source, appear as sources to all processes of \mathcal{SC} . Hence we have $S \subseteq S' \subseteq \mathcal{SC}$, and $S' \neq \emptyset$.

The variables $leader_i$ implement Ω .

According to the definition of \mathcal{SC} , there is a round r_0 after which no more message from a weakly correct simulator is received (directly or indirectly) by a strongly correct simulator. Let $r_1 = \max\{r_s, s \in S'\}$ where r_s is the first round after which no message sent by p_s to a strongly correct simulator is eliminated. As, after r_1 , each strongly correct simulator receives at every round a message from each simulator in S' , it follows that none of them adds a pair (s, r) , $r \geq r_1$, $s \in S'$ in its variable $missed_i$ at line 10. After $r_2 = \max\{r_0, r_1\}$, the only pairs (s, r) , $s \in S'$ ($r < r_1$) that are added by a strongly correct simulator in its variable $missed_i$ are those that have been added by other strongly correct simulators at line 8 or line 11 before r_2 . Since strongly correct simulators are infinitely often able to transmit (directly or not) messages to

each other, there is a round $r_3 \geq r_2$ such that any strongly correct simulator p_i has received (directly or not) during a round $r_j \geq r_2$ the information contained in the variable $missed_j$ from each other strongly correct simulator p_j . After r_3 , for any $s \in S'$, the number of pairs (s, r) in the variables $missed_i$ of all strongly correct simulators p_i is the same and does not increase anymore.

For each simulator p_i , $i \notin S'$, there is an infinite number of rounds r such that p_i 's message is not received during round r by at least one of the strongly correct simulator p_j , and accordingly, this simulator adds a pair (i, r) to its variable $missed_j$ during round r at line 10. As the strongly correct simulators communicate (directly or not) infinitely often with each other, all of them eventually add this pair to their variable $missed$ during r (at line 10) or later (at line 8). Consequently, for each such simulator p_i , $i \notin S'$, the number of pairs (i, r) in the variable $missed_j$ of every strongly simulator p_j increases forever.

It follows from the previous discussion that the minimal number of rounds missed by a simulator (as calculated at line 12, and using simulator identity to do tie-breaking) eventually becomes and remains the same at each strongly correct simulator. Let ℓ_d denote this simulator identity. As it is the identity that is eventually always returned when $leader_i$ is read by any simulated process p_i whose simulator is strongly correct, the unicity eventual property of Ω is ensured for these processes. The next paragraph shows that the set of strongly correct simulators corresponds exactly to the set of correct simulated processes. As p_{ℓ_d} is strongly correct, the elected process is a correct process, which concludes the proof of Ω .

Correct and faulty (simulated) processes in $\mathcal{AMP}_{n,n-1}[fd : \Omega]$.

It follows from the previous paragraphs that each strongly correct simulator p_i is always eventually able to transmit (directly or not) a new message m to p_{ℓ_d} , and then eventually receive (directly) a message from p_{ℓ_d} containing m . Hence the conditions of line 13 and line 16 are fulfilled an infinite number of times and, consequently, the corresponding simulator can always issue enough steps (line 17) to progress in the simulated code.

Hence, the correct simulated processes and the faulty simulated processes are the ones simulated by the strongly correct and weakly correct simulators, respectively.

Linearization of communication operations.

Let us consider a simulated process p_i that sends a message m to a simulated process p_j . This operation is disseminated to each simulator by p_i 's simulator at line 5. Then a simulator considers this simulated message m only at line 17 when the second input parameter of its invocation of `simulate()` contains the message m . (Let us observe, that this message m arrives at a simulator p_k from its current leader ℓ_d , lines 13 and 16).

Let τ_1 be the time of the first invocation of `simulate()` by a simulator such that m belongs to the second input parameter of this invocation, where $\tau_1 = \infty$ if there is no such invocation. Let τ_2 be the time at which the simulator of p_i starts the execution of `simulate()` (line 17) after it has disseminated m , where $\tau_2 = \infty$ if there is no such invocation.

The send of m is linearized at time $\min(\tau_1, \tau_2)$ (let us notice that the simulation of p_i does not progress between the sending of m by p_i and its linearization point). If $\min(\tau_1, \tau_2) = \infty$, the send of m is linearized after the receiver p_j has computed its result.

The reception of m is linearized at the time of the invocation by p_j of `simulate()` whose second input parameter contains the message m , or after p_j has computed its result if there is no such invocation. \square *Lemma 4*

4.3 The Message Adversary SOURCE Captures Exactly Ω in $\mathcal{AMP}_{n,n-1}[fd : \emptyset]$

THEOREM 3. *A task can be solved in $\mathcal{AMP}_{n,n-1}[fd : \Omega]$ iff it can be solved in $\mathcal{SMP}_n[adv : \text{SOURCE}]$.*

Proof The proof follows directly from Lemma 3 and Lemma 4. \square *Theorem 3*

5. THE ADVERSARY QUORUM CAPTURES EXACTLY Σ IN $\mathcal{AMP}_{n,n-1}[fd : \emptyset]$

This section shows that the computing models $\mathcal{SMP}_n[adv : \text{QUORUM}]$ and $\mathcal{AMP}_{n,n-1}[fd : \Sigma]$ have the same computational power for tasks.

5.1 The Property QUORUM

Let us remember that \mathcal{SC} is the set of strongly correct processes in the considered synchronous message-passing system (processes whose an infinite number of messages are received by each other process). The property QUORUM is defined as follows:

$$[\forall i, j : \forall r_i, r_j : (\{k : k \xrightarrow{r_i} i\} \cap \{k : k \xrightarrow{r_j} j\} \neq \emptyset) \wedge (\mathcal{SC} \neq \emptyset)].$$

This property is a statement of Σ suited to the context of round-based synchronous message-passing systems prone to message adversaries. Given any pair of processes p_i and p_j , its first part states that, whatever the synchronous rounds r_i and r_j executed by p_i and p_j , respectively, there is a process p_k whose messages to p_i at round r_i and to p_j at round r_j are not eliminated by the adversary (intersection property). The second part states that there is at least one process whose messages are infinitely often received by each other process (liveness property). Theorem 4 will show that this formulation of Σ is correct for the equivalence of $\mathcal{AMP}_{n,n-1}[fd : \Sigma]$ and $\mathcal{SMP}_n[adv : \text{QUORUM}]$ for task solvability.

```

(1)  $r_i \leftarrow 0$ ;  $sim\_rec\_msgs_i[1, \dots, n] \leftarrow [\perp, \dots, \perp]$ ;
(2)  $(msgs\_to\_send_i[1, \dots, n], ls\_state_i) \leftarrow simulate(sim\_rec\_msgs_i)$ ;
(3) for each  $r > 0$ 
    do  $rec\_msgs_i[r][1, \dots, n] \leftarrow [\perp, \dots, \perp]$  end for;
(4) repeat forever
(5)    $r \leftarrow r_i + 1$ ;
(6)   for each  $j \in \{1, \dots, n\}$ 
    do send( $r_i, msgs\_to\_send_i[j]$ ) to  $p_j$  end for;
(7)   repeat  $cur\_qr_i \leftarrow qr_i$ 
(8)   until  $(\forall j \in cur\_qr_i \setminus \{i\} : rec\_msgs_i[r_i][j] \neq \perp)$ 
(9)   end repeat;
(10)  for each  $j \in cur\_qr_i$ 
    do  $sim\_rec\_msgs_i[j] \leftarrow rec\_msgs_i[r_i][j]$  end for;
(11)   $(msgs\_to\_send_i[1, \dots, n], ls\_state_i) \leftarrow simulate(sim\_rec\_msgs_i)$ ;
(12)   $sim\_rec\_msgs_i[1, \dots, n] \leftarrow [\perp, \dots, \perp]$ 
(13) end repeat.

when  $(r, m)$  received from  $p_j$ :  $rec\_msgs_i[r][j] \leftarrow m$ .
```

Figure 6: From asynchronous $\mathcal{AMP}_{n,n-1}[fd : \Sigma]$ to synchronous $\mathcal{SMP}_n[adv : \text{QUORUM}]$

5.2 From Asynchronous $\mathcal{AMP}_{n,n-1}[fd : \Sigma]$ to Synchronous $\mathcal{SMP}_n[adv : \text{QUORUM}]$

The simulation from $\mathcal{AMP}_{n,n-1}[fd : \Sigma]$ to $\mathcal{SMP}_n[adv : \text{QUORUM}]$ is described in Figure 6. It has the same local variables as, and is very close to, the one of Figure 5. In addition to the


```

initialization:
(1)  $ls\_state_i \leftarrow$  initial state of the local simulated algorithm;
(2)  $msgs\_to\_rec_i \leftarrow \emptyset; msgs\_received_i \leftarrow \emptyset;$ 
(3)  $(msgs\_to\_send_i, ls\_state_i) \leftarrow \text{simulate}(ls\_state_i, msgs\_to\_rec_i);$ 
(4)  $view_i \leftarrow msgs\_to\_send_i; rec\_from_i \leftarrow \{1, \dots, n\}.$ 

round  $r = 1, 2, \dots$  do:
(5)  $\text{send}(i, view_i)$  to each other process;
(6)  $rec\_msgs_i \leftarrow$  set of pairs  $(j, view\_j)$  received during this round;
(7)  $view_i \leftarrow view_i \cup \left( \bigcup_{(j, view\_j) \in rec\_msgs_i} view\_j \right);$ 
(8)  $rec\_from_i \leftarrow \{j \in \{1, \dots, n\} : \exists (j, view\_j) \in rec\_msgs_i\} \cup \{i\};$ 
(9) if  $(msgs\_to\_send_i \in \bigcap_{(j, view\_j) \in rec\_msgs_i} view\_j)$  then
(10)  $msgs\_to\_rec_i \leftarrow msgs\_to\_rec_i \cup \{(j, i, m) : (j, view\_j) \in rec\_msgs_i \wedge (j, i, m) \in view\_j\};$ 
(11)  $(msgs\_to\_send_i, ls\_state_i) \leftarrow \text{simulate}(ls\_state_i, msgs\_to\_rec_i \setminus msgs\_received_i);$ 
(12)  $msgs\_received_i \leftarrow msgs\_to\_rec_i; view_i \leftarrow view_i \cup msgs\_to\_send_i;$ 
(13) end if.

when  $qr_i$  is read:  $\text{return}(rec\_from_i).$ 

```

Figure 7: Simulation of $\mathcal{AMP}_{n,n-1}[fd : \Sigma]$ in $\mathcal{SMP}_n[adv : \text{QUORUM}]$

local output of the failure detector Σ , which is denoted qr_i , the only modifications are the lines 7-10 which differ in both algorithms.

The simulator of p_i waits until it has received a message from each process that appears in its current quorum qr_i (lines 7-9). It then invokes the procedure `simulate()` with these messages as input (line 10).

The principle of this simulation is the following: after some time, the simulated message adversary suppresses all the messages sent by processes that do not belong to a quorum, but is prevented from suppressing the messages sent by processes belonging to quorums.

LEMMA 5. *If a task can be solved in $\mathcal{SMP}_n[adv : \text{QUORUM}]$, it can be solved in $\mathcal{AMP}_{n,n-1}[fd : \Sigma]$.*

5.3 From Synchronous $\mathcal{SMP}_n[adv : \text{QUORUM}]$ to Asynchronous $\mathcal{AMP}_{n,n-1}[fd : \Sigma]$

This simulation algorithm is described in Figure 7. It is very close to the simulation of $\mathcal{AMP}_{n,n-1}[fd : \Omega]$ on top of the model $\mathcal{SMP}_n[adv : \text{SOURCE}]$ presented in Figure 4. It has the same local variables, except the variable $missed_i$ which is now useless. The value returned when qr_i is read by a simulated process p_i is now the current value of the set rec_from_i .

The only other difference appears at lines 9-10. The simulation of the simulated process p_i (invocation of the procedure `simulate()` at lines 11) is now constrained by the predicate of line 9 which states that the messages that p_i wants to send (the messages saved in $msg_to_send_i$) must be known by at all the simulators defining the current quorum of p_i (set rec_from_i). When this is satisfied, the set of messages to be received by p_i in the next invocation of `simulate()` is redefined (line 11) to include the last simulated messages sent to p_i by processes p_j such that $j \in rec_from_i$.

LEMMA 6. *If a task can be solved in $\mathcal{AMP}_{n,n-1}[fd : \Sigma]$, it can be solved in $\mathcal{SMP}_n[adv : \text{QUORUM}]$.*

5.4 The Message Adversary QUORUM Captures Exactly Σ in $\mathcal{AMP}_{n,n-1}[fd : \emptyset]$

THEOREM 4. *A task can be solved in the synchronous model $\mathcal{SMP}_n[adv : \text{QUORUM}]$ iff it can be solved in the message-passing model $\mathcal{AMP}_{n,n-1}[fd : \Sigma]$.*

Proof Follows immediately from Lemmas 5 and 6. $\square_{\text{Theorem 4}}$

6. THE PAIR OF MESSAGE ADVERSARIES (SOURCE, QUORUM) CAPTURES EXACTLY THE PAIR (Σ, Ω) IN $\mathcal{AMP}_{n,n-1}[fd : \emptyset]$

Let us notice that the properties SOURCE and QUORUM are independent of one another in the sense that none of them can be obtained from the other. It follows that the power provided by SOURCE and the power provided by QUORUM can be added. More specifically, we have the following:

- A merge of the simulation of $\mathcal{SMP}_n[adv : \text{SOURCE}]$ in $\mathcal{AMP}_{n,n-1}[fd : \Omega]$ (Figure 5) with the simulation of the model $\mathcal{SMP}_n[adv : \text{QUORUM}]$ in $\mathcal{AMP}_{n,n-1}[fd : \Sigma]$ (Figure 6) provides a simulation of the model $\mathcal{SMP}_n[adv : \text{SOURCE, QUORUM}]$ in $\mathcal{AMP}_{n,n-1}[fd : \Sigma, \Omega]$. The difference between this simulation and the one of Figure 5 (or Figure 6) is at lines 7-10 which becomes
 - (7) **repeat** $cur_ld_i \leftarrow leader_i; cur_qr_i \leftarrow qr_i$ **until**
 - (8) $[(\forall j \in cur_qr_i \setminus \{i\} : rec_msgs_i[r_i][j] \neq \perp) \wedge (cur_ld_i = i \vee rec_msgs_i[r_i][cur_ld_i] \neq \perp)]$
 - (9) **end repeat;**
 - (10) **for each** $j \in cur_qr_i \cup cur_ld_i$ **do** $sim_rec_msgs_i[j] \leftarrow rec_msgs_i[r_i][j]$ **end for.**

The proof is the same as in Lemma 5 augmented by the fact that the eventual leader elected by Ω verifies the property SOURCE as shown in Lemma 3.

- Similarly, adding the management of $missed_i$ and the procedure to query Ω (as done at lines 8-11 of Figure 4) to the simulation of $\mathcal{AMP}_{n,n-1}[fd : \Sigma]$ in $\mathcal{SMP}_n[adv : \text{QUORUM}]$ (Figure 7) provides a simulation of the asynchronous model $\mathcal{AMP}_{n,n-1}[fd : \Sigma, \Omega]$ in synchronous $\mathcal{SMP}_n[adv : \text{SOURCE, QUORUM}]$.

The linearization points and the proof of the properties of Σ are the same as in Lemma 6, while the proof of the properties of Ω follows the one of Lemma 4. Let us finally notice that it follows directly from the properties SOURCE and QUORUM that a process verifying the SOURCE property appears eventually in all the simulated quorums.

Theorem 5 then follows:

THEOREM 5. *A task can be solved in the synchronous model $\mathcal{SMP}_n[adv : \text{SOURCE, QUORUM}]$ iff it can be solved in the enriched message-passing model $\mathcal{AMP}_{n,n-1}[fd : \Sigma, \Omega]$.*

7. CONCLUSION

Considering crash-free synchronous round-based systems, message adversaries have been designed as daemons that suppress messages. Failure detectors have been introduced to enrich crash-prone asynchronous (read/write or message-passing) systems. A previous work [1] has shown that, from a task solvability point of view, the message adversaries constrained by a property denoted TOUR (for tournament) characterizes the well-known wait-free read/write model.

Considering task solvability, this paper has introduced relations linking failures detectors and message adversaries. More precisely, it has introduced two new properties, denoted SOURCE and QUORUM, which are restrictions on message adversaries, and has shown that

- The synchronous message adversaries SOURCE+TOUR characterizes the asynchronous wait-free read/write model enriched with Ω ,
- The synchronous message adversary SOURCE characterizes the asynchronous Ω -enriched crash-prone message-passing model,
- The synchronous message adversary QUORUM characterizes the asynchronous crash-prone message-passing model enriched with Σ ,
- The synchronous message adversaries SOURCE+QUORUM characterizes the asynchronous crash-prone message-passing model enriched with the pair (Σ, Ω) .

Hence, when considering task solvability, these characterizations state properties' defining the strongest message adversaries for synchronous round-based message-passing systems equating classical asynchronous crash-prone systems. Interestingly, this allows for the establishment of a hierarchy on message adversaries (e.g., $SMP_n[adv : QUORUM]$ is stronger than $SMP_n[adv : TOUR]$ as shown on the left of Figure 1; this follows from the fact that the computability power of Σ is strictly stronger than read/write registers).

In our understanding of the foundations of distributed computing, a lot of issues remain still open. As examples, here are two interesting message adversary-related problems. Which is the weakest message adversary AD such that consensus can be solved in $SMP_n[adv : AD]$? (The only thing we know is that the synchronous model $SMP_n[adv : CONS]$ is weaker than or equivalent to synchronous model $SMP_n[adv : SOURCE, TOUR]$ and strictly stronger than $SMP_n[adv : TOUR]$). Is the addition of the constraint $|SC| \geq n - t$ to an adversary sufficient to characterize t -resilient asynchronous crash-prone read/write or message-passing systems?

Acknowledgments

This work has been partially supported by the French ANR project DISPLEXITY devoted to computability and complexity in distributed computing. Special thanks to D. Imbs and S. Rajsbaum for discussions on distributed computing models, and to the referees for their constructive comments.

8. REFERENCES

- [1] Afek Y. and Gafni E., Asynchrony from synchrony. *Proc. Int'l Conference on Distributed Computing and Networking (ICDCN'13)*, Springer LNCS 7730, pp. 225-239, 2013.
- [2] Afek Y., Gafni E., and Linial N., A king in two tournaments. *Unpublished report*, 3 pages, March 2012. http://www.cs.huji.ac.il/~nati/PAPERS/king_tournaments.pdf.
- [3] Biely M., Robinson P., and Schmid U., Agreement in directed dynamic networks. *Proc. 19th Int'l Colloquium on Structural Information and Communication Complexity (SIROCCO'12)*, Springer LNCS 7355, pp. 73-84, 2012.
- [4] Chandra T. and Toueg S., Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2):225-267, 1996.
- [5] Chandra T., Hadzilacos V. and Toueg S., The weakest failure detector for solving consensus. *Journal of the ACM*, 43(4):685-722, 1996.
- [6] Charron-Bost B. and Schiper A., The *heard-of* model: computing in distributed systems with benign faults. *Distributed Computing*, 22(1):49-71, 2009.
- [7] Cornejo A., Rajsbaum S., Raynal M., Travers C., Failure Detectors as Schedulers (Brief Announcement). *Proc. 26th ACM Symposium on Principles of Distributed Computing (PODC)*, ACM Press, pp. 308-309, 2007.
- [8] Delporte-Gallet C., Fauconnier H., and Guerraoui R., Tight failure detection bounds on atomic object implementations. *Journal of the ACM*, 57(4), Article 22, 2010.
- [9] Delporte-Gallet C., Fauconnier H., and Toueg S., The minimum information about failures for solving non-local tasks in message-passing systems. *Distributed Computing*, 24(5):255-269, 2011.
- [10] Fernández Anta A. and Raynal M., From an asynchronous intermittent rotating star to an eventual leader. *IEEE Transactions on Parallel Distributed Systems*, 21(9):1290-1303, 2010.
- [11] Herlihy M.P., Wait-free synchronization. *ACM Transactions on Programming Languages and Systems*, 13(1):124-149, 1991.
- [12] Herlihy M.P. and Shavit N., The topological structure of asynchronous computability. *Journal ACM*, 46(6):858-923, 1999.
- [13] Kuhn F., Lynch N.A., and Oshman R., Distributed computation in dynamic networks. *Proc. 42nd ACM Symposium on Theory of Computing (STOC'10)*, ACM press, pp. 513-522, 2010.
- [14] Landau H.G., On dominance relations and the structure of animal societies, III: The condition for score structure. *Bulletin of Mathematical Biophysics*, 15(2):143-148, 1953.
- [15] Lo W.-K. and Hadzilacos V., Using failure detectors to solve consensus in asynchronous shared-memory systems. *Proc. 8th Int'l Workshop on Distributed Algorithms (WDAG'94)*, Springer LNCS 857, pp. 280-295, 1994.
- [16] Mourgaya E., Mostéfaoui A., and Raynal M., Asynchronous implementation of failure detectors. *Proc. Int'l IEEE Conference on Dependable Systems and Networks (DSN 2003)*, IEEE Press, pp. 351-360, 2003.
- [17] Pike S.M., Sastry S. and Welch J.L., Failure detectors encapsulate fairness. *Distributed Computing*, 25(4): 313-333, 2012.
- [18] Rajsbaum S., Raynal M., Travers C., The iterated restricted immediate snapshot model. *Proc. 14th Annual Int'l Conference on Computing and Combinatorics (COCOON 2008)*, Springer LNCS 5092, pp. 487-497, 2008.
- [19] Raynal M., Failure detectors for asynchronous distributed systems: an introduction. *Wiley Encyclopedia of Computer Science and Engineering*, 2:1181-1191, 2009.
- [20] Raynal M. and Stainer J., Increasing the power of the iterated immediate snapshot model with failure detectors. *Proc. 19th Int'l Colloquium on Structural Information and Communication Complexity (SIROCCO'12)*, Springer LNCS 7355, pp. 231-242, 2012.
- [21] Raynal M. and Stainer J., Round-based synchrony weakened by message adversaries *vs* asynchrony enriched with failure detectors. *Tech Report 8235*, IRISA, Université de Rennes (F), 19 pages, 2012. (<http://hal.inria.fr/hal-00787978/>)
- [22] Santoro N. and Widmayer P., Time is not a healer. *Proc. 6th Annual Symposium on Theoretical Aspects of Computer Science (STACS'89)*, Springer LNCS 349, pp. 304-316, 1989.
- [23] Santoro N. and Widmayer P., Agreement in synchronous networks with ubiquitous faults. *Theoretical Computer Science*, 384(2-3): 232-249, 2007.
- [24] Schmid U., Weiss B., and Keidar I., Impossibility results and lower bounds for consensus under link failures. *SIAM Journal of Computing*, 38(5): 1912-1951, 2009.