Set-Constrained Delivery Broadcast: Definition, Abstraction Power, and Computability Limits

Damien Imbs, Achour Mostefaoui, Matthieu Perrin, Michel Raynal

Set-Constrained Delivery Broadcast: Definition, Abstraction Power, and Computability Limits

Damien Imbs
LIF, Université Aix-Marseille
Marseille, France
damien.imbs@lif.univ-mrs.fr

Achour Mostéfaoui
LS2N, Université de Nantes
Nantes, France
achour.mostefaoui@univ-nantes.fr

Matthieu Perrin
LS2N, Université de Nantes
Nantes, France
matthieu.perrin@univ-nantes.fr

Michel Raynal
Institut Universitaire de France, IRISA, Université de Rennes
Rennes, France
raynal@irisa.fr

ABSTRACT

This paper introduces a new communication abstraction, called *Set-Constrained Delivery Broadcast* (SCD-broadcast), whose aim is to provide its users with an appropriate abstraction level when they have to implement *objects* or *distributed tasks* in an asynchronous message-passing system prone to process crash failures. This abstraction allows each process to broadcast messages and deliver a sequence of sets of messages in such a way that, if a process delivers a set of messages including a message m and later delivers a set of messages including a message m', no process delivers first a set of messages including m' and later a set of message including m.

After having presented an algorithm implementing SCD-broadcast, the paper investigates its programming power and its computability limits. On the "power" side it presents SCD-broadcast-based algorithms, which are both simple and efficient, building objects (such as snapshot and conflict-free replicated data), and distributed tasks. On the "computability limits" side it shows that SCD-broadcast and read/write registers are computationally equivalent.

CCS Concepts

C.2.4 [**Computer-Communication Network**]: Distributed Systems -*distributed applications, network operating systems*; D.4.1 [**Operating Systems**] -*concurrency*; D.4.5 [**Operating Systems**] Reliability; -*fault-tolerance*; F.1.1 [**Computation by Abstract Devices**]: Models of Computation -*Computability theory*

General Terms:
Algorithms, Reliability, Theory

Keywords:
Abstraction, Asynchronous system, Communication abstraction, Conflict-free replicated data type, Design simplicity, Distributed task, Linearizability, Message-passing system, Process crash, Read/write atomic register, Snapshot object.

1 INTRODUCTION

Programming abstractions. Informatics is a science of abstractions, and a main difficulty consists in providing users with a "desired level of abstraction and generality – one that is broad enough to encompass interesting new situations, yet specific enough to address the crucial issues" as expressed in [18]. When considering sequential computing, functional programming and object-oriented programming are well-know examples of what means "desired level of abstraction and generality".

In the context of asynchronous distributed systems where the computing entities (processes) communicate –at the basic level– by sending and receiving messages through an underlying communication network, and where some of them can experience failures, a main issue consists in finding appropriate communication-oriented abstractions, where the meaning of the term "appropriate" is related to the problems we intend to solve. Solving a problem at the send/receive abstraction level is similar to the writing of a program in a low-level programming language. Programmers must be provided with abstractions that allow them to concentrate on the problem they solve and not on the specific features of the underlying system. This is not new. Since a long time, high level programming languages have proved the benefit of this approach. From a synchronization point of view, this approach is the one promoted in *software transactional memory* [34], whose aims is to allow programmers to focus on the synchronization needed to solve their problems and not on the way this synchronization must be implemented (see the textbooks [19, 30]).

If we consider specific coordination/cooperation problems, "matchings" between problems and specific communication abstractions are known. One of the most famous examples concerns the consensus problem whose solution rests on the *total order broadcast* abstraction[1]. Another "matching" example is the *causal message delivery* broadcast abstraction [11, 32], which allows for a very simple implementation of a causal read/write memory [2].

Aim of the paper. The aim of this paper is to introduce and investigate a high level communication abstraction which allows for simple and efficient implementations of concurrent objects and distributed tasks, in the context of asynchronous message-passing systems prone to process crash failures. The concurrent objects in which we are interested are defined by a sequential specification [20] (e.g., a queue). Differently, a task extends to the

[1]Total order broadcast is also called *atomic broadcast*. Actually, total order broadcast and consensus have been shown to be computationally equivalent [12]. A more general result is presented in [22], where is introduced a communication abstraction which "captures" the k-set agreement problem [13, 31] (consensus is 1-set agreement).

distributed context the notion of a function [10, 28]. It is defined by a mapping from a set of input vectors to a set of output vectors, whose sizes are the number of processes. An input vector I defines the input value $I[i]$ of each process p_i, and, similarly, an output vector O defines the output $O[j]$ of each process p_j. Agreement problems such as consensus and k-set agreement are distributed tasks. What makes the implementation of a task difficult is the fact that each process knows only its input, and, due to net effect of asynchrony and process failures, no process can distinguish if another process is very slow or crashed. The difficulty results in an impossibility for consensus [17], even in a system in which at most one process may crash.

Content of the paper: a broadcast abstraction. The SCD-broadcast communication abstraction proposed in the paper allows a process to broadcast messages, and to deliver sets of messages (instead of a single message) in such a way that, if a process p_i delivers a message set $mset$ containing a message m, and later delivers a message set $mset'$ containing a message m', then no process p_j can deliver first a set containing m' and later another set containing m. Let us notice that p_j is not prevented from delivering m and m' in the same set. Moreover, SCD-broadcast imposes no constraint on the order in which a process must process the messages it receives in a given message set.

After having defined SCD-broadcast, the paper presents an implementation of it in asynchronous systems where a minority of processes may crash. This assumption is actually a necessary and sufficient condition to cope with the net effect of asynchrony and process failures (see below). Assuming an upper bound Δ on message transfer delays, and zero processing time, an invocation of SCD-broadcast is upper bounded by 2Δ time units, and $O(n^2)$ protocol messages (messages generated by the implementation algorithm).

Content of the paper: implementing objects and tasks. Then, the paper addresses two fundamental issues of SCD-broadcast: its abstraction power and its computability limits. As far as its abstraction power is concerned, i.e., its ability and easiness to implement atomic (linearizable) or sequentially consistent concurrent objects [20, 27] and read/write solvable distributed tasks, the paper presents, on the one side, two algorithms implementing atomic objects (namely a snapshot object [1, 3], and a distributed increasing/decreasing counter), and, on the other side, an algorithm solving the lattice agreement task [6, 16].

The two concurrent objects (snapshot and counter) have been chosen because they are encountered in many applications, and are also good representative of the class of objects identified in [4]. The objects of this class are characterized by the fact that each pair op1 and op2 of their operations either commute (i.e., in any state, executing op1 before op2 leads to the same state as executing op2 before op1, as it is the case for a counter), or any of op1 and op2 can overwrite the other one (e.g., executing op1 before op2 leads to the same state as executing op2 alone). Our implementation of a counter can be adapted for all objects with commutative operations, and our implementation of the snapshot object illustrates how overwriting operations can be obtained directly from the SCD-broadcast abstraction. Concerning these objects, it is also shown that a slight change in the algorithms allows us to obtain implementations (with a smaller cost) in which the consistency condition is weakened from linearizability to sequential consistency [26].

In the case of read/write solvable tasks, SCD-broadcast shows how the concurrency inherent (but hidden) in a task definition can be easily mastered and solved.

Content of the paper: the computability limits of SCD-broadcast. The paper also investigates the computability power of the SCD-broadcast abstraction, namely it shows that SCD-broadcast and atomic read/write registers (or equivalently snapshot objects) have the same computability power in asynchronous systems prone to process crash failures. Everything that can be implemented with atomic read/write registers can be implemented with SCD-broadcast, and vice versa.

As read/write registers (or snapshot objects) can be implemented in asynchronous message-passing system where only a minority of processes may crash [5], it follows that the proposed algorithm implementing SCD-broadcast is resilience-optimal in these systems. From a theoretical point of view, this means that the consensus number of SCD-broadcast is 1 (the weakest possible).

Roadmap. The paper is composed of 8 sections. Section 2 defines the SCD-broadcast abstraction. Section 3 presents a resilience-optimal algorithm implementing SCD-broadcast in asynchronous message-passing systems prone to process crash failures, while Then, Sections 4-6 present SCD-broadcast-based algorithms for concurrent objects and tasks. Section 7 focuses on the computability limits of SCD-broadcast. Finally, Section 8 concludes the paper. Due to page limitations, proofs are not given in the article, but they can be found in [21].

2 THE SCD-BROADCAST COMMUNICATION ABSTRACTION

Process model. The computing model is composed of a set of n asynchronous sequential processes, denoted p_1, ..., p_n. "Asynchronous" means that each process proceeds at its own speed, which can be arbitrary and always remains unknown to the other processes.

A process may halt prematurely (crash failure), but it executes its local algorithm correctly until it crashes (if it ever does). The model parameter t denotes the maximal number of processes that may crash in a run r. A process that crashes in a run is said to be *faulty* in r. Otherwise, it is *non-faulty*.

Definition of SCD-broadcast. The set-constrained delivery broadcast abstraction (SCD-broadcast) provides the processes with an operation scd_broadcast(m) that takes a message m to broadcast as parameter, and triggers the event scd_deliver($mset$) that provides the process on which it was triggered with a non-empty set of messages $mset$. Using a classical terminology, when a process invokes the operation scd_broadcast(m), we say that it "scd-broadcasts a message m". Similarly, when the event scd_deliver($mset$) is triggered on this process, we say that it "scd-delivers the set of messages $mset$". By a slight abuse of language, when we are interested in a message m, we say that a process "scd-delivers the message m" when actually it scd-delivers the message set $mset$ containing m.

SCD-broadcast is defined by the following set of properties, where we assume –without loss of generality– that all the messages that are scd-broadcast are different.[2]

- Validity. If a process scd-delivers a set containing a message m, then m was scd-broadcast by a process.
- Integrity. A message is scd-delivered at most once by each process.
- MS-Ordering. Let p_i be a process that scd-delivers first a message set $mset_i$ and later a message set $mset'_i$. For any pair of messages $m \in mset_i$ and $m' \in mset'_i$, no process p_j scd-delivers first a message set $mset'_j$ containing m' and later a message set $mset_j$ containing m.
- Termination-1. If a non-faulty process scd-broadcasts a message m, it terminates its scd-broadcast invocation and scd-delivers a message set containing m.
- Termination-2. If a process scd-delivers a message m, every non-faulty process scd-delivers a message set containing m.

Termination-1 and Termination-2 are classical liveness properties (found for example in Uniform Reliable Broadcast [9, 29]). The other ones are safety properties. Validity and Integrity are classical communication-related properties. The first states that there is neither message creation nor message corruption, while the second states that there is no message duplication.

The MS-Ordering property is new, and characterizes SCD-broadcast. It states that the contents of the sets of messages scd-delivered at any two processes are not totally independent: the sequence of sets scd-delivered at a process p_i and the sequence of sets scd-delivered at a process p_j must be mutually consistent in the sense that a process p_i cannot scd-deliver first $m \in mset_i$ and later $m' \in mset'_i \neq mset_i$, while another process p_j scd-delivers first $m' \in mset'_j$ and later $m \in mset_j \neq mset'_j$. Let us nevertheless observe that if p_i scd-delivers first $m \in mset_i$ and later $m' \in mset'_i$, p_j may scd-deliver m and m' in the same set of messages.

Let us remark that, if the MS-Ordering property is suppressed and messages are scd-delivered one at a time, SCD-broadcast boils down to the well-known *Uniform Reliable Broadcast* abstraction [12, 29].
Differently, if all message sets contain exactly one message, SCD-broadcast is equivalent to the *Atomic Broadcast* abstraction.

An example. Let m_1, m_2, m_3, m_4, m_5, m_6, m_7 and m_8 be messages that have been scd-broadcast by different processes. The following scd-deliveries of message sets by p_1, p_2 and p_3 respect the definition of SCD-broadcast:

- at p_1: $\{m_1, m_2\}$, $\{m_3, m_4, m_5\}$, $\{m_6\}$, $\{m_7, m_8\}$.
- at p_2: $\{m_1\}$, $\{m_2, m_3\}$, $\{m_4, m_5, m_6\}$, $\{m_7\}$, $\{m_8\}$.
- at p_3: $\{m_1, m_2, m_3\}$, $\{m_4, m_5\}$, $\{m_6, m_7\}$, $\{m_8\}$.

Differently, due to the scd-deliveries of the sets including m_2 and m_3, the following scd-deliveries by p_1 and p_2 do not satisfy the MS-broadcast property:

- at p_1: $\{m_1, m_2\}$, $\{m_3, m_4, m_5\}$, ...
- at p_2: $\{m_1, m_3\}$, $\{m_2\}$, ...

[2] The unicity of messages is not restrictive in practice, as one can assume the broadcast operation tags messages with a unique identifer. This way, messages can be unique even if their applicative content is not.

A containment property. Let $mset_i^\ell$ be the ℓ-th message set scd-delivered by p_i. Hence, at some time, p_i scd-delivered the sequence of message sets $mset_i^1, \ldots, mset_i^x$. Let $MSet_i^x = mset_i^1 \cup \ldots \cup mset_i^x$. The following *Containment* property follows directly from the MS-Ordering and Termination-2 properties:
$\forall\, i, j, x, y$: $(MSet_i^x \subseteq MSet_j^y) \vee (MSet_j^y \subseteq MSet_i^x)$.

Partial order on messages created by the message sets. The MS-Ordering and Integrity properties establish a partial order on the set of all the messages, defined as follows. Let $\mapsto_i$ be the local message delivery order at process p_i defined as follows: $m \mapsto_i m'$ if p_i scd-delivers the message set containing m before the message set containing m'. As no message is scd-delivered twice, it is easy to see that $\mapsto_i$ is a partial order (locally know by p_i). The containment property implies that there is a total order (which remains unknown to the processes) on the whole set of messages, that complies with the partial order $\mapsto = \cup_{1 \leq i \leq n} \mapsto_i$. This is where SCD-broadcast can be seen as a weakening of total order broadcast.

3 IMPLEMENTATION OF SCD-BROADCAST

This section shows that the SCD-broadcast communication abstraction is not an oracle-like object (oracles allow us to extend our understanding of computing, but cannot be implemented). It describes an implementation of SCD-broadcast in an asynchronous send/receive message-passing system in which any minority of processes may crash. This system model is denoted $\mathcal{CAMP}_{n,t}[t < n/2]$ (where $\mathcal{CAMP}_{n,t}$ stands for "Crash Asynchronous Message-Passing" and $t < n/2$ is its restriction on failures). As $t < n/2$ is the weakest assumption on process failures that allows a read/write register to be built on top of an asynchronous message-passing system [5][3], and SCD-broadcast and read/write registers are computationally equivalent (as shown in Sections 4 and 7), the proposed implementation is optimal from a resilience point of view.

3.1 Underlying communication network

Send/receive asynchronous network. Each pair of processes communicate through two uni-directional channels on which they send and receive messages. Hence, the communication network is a complete network: any process p_i can directly send a message to any process p_j (including itself). A process p_i invokes the operation "send TYPE(m) to p_j" to send to p_j the message m, whose type is TYPE. The operation "receive TYPE() from p_j" allows p_i to receive from p_j a message whose type is TYPE.

Each channel is reliable (no loss, corruption, nor creation of messages), not necessarily FIFO, and asynchronous (while the transit time of each message is finite, there is no upper bound on message transit times). Let us notice that, due to process and message asynchrony, no process can know if another process crashed or is only very slow.

Uniform FIFO-broadcast abstraction. To simplify the presentation, and without loss of generality, we consider that the system is

[3] From the point of view of the maximal number of process crashes that can be tolerated, assuming failures are independent.

equipped with a FIFO-broadcast abstraction. Such an abstraction can be built on top of the previous basic system model without enriching it with additional assumptions (see e.g. [29]). It is defined by the operations fifo_broadcast() and fifo_deliver(), which satisfy the properties of Uniform Reliable Broadcast (Validity, Integrity, Termination 1, and Termination 2), plus the following message ordering property.

- FIFO-Order. For any pair of processes p_i and p_j, if p_i fifo-delivers first a message m and later a message m', both from p_j, no process fifo-delivers m' before m.

3.2 Algorithm

This section describes Algorithm 1, which implements SCD-broadcast in $\mathcal{CAMP}_{n,t}[t < n/2]$. From a terminology point of view, an *application message* is a message that has been scd-broadcast by a process, while a *protocol message* is an implementation message generated by Algorithm 1.

Local metadata quadruplets. For each application message m, each process stores a quadruplets $qdplt = \langle qdplt.msg, qdplt.sd, qdplt.f, qdplt.cl\rangle$ whose fields have the following meaning.

- $qdplt.msg$ contains an application message m,
- $qdplt.sd$ contains the id of the sender of $qdplt.msg$,
- $qdplt.sn$ contains the local date (seq. number) associated with m by its sender. Hence, $\langle qdplt.sd, qdplt.sn\rangle$ is the identity of the application message m.
- $qdplt.cl$ is an array of size n, initialized to $[+\infty, \ldots, +\infty]$. Then, $qdplt.cl[x]$ will contain the sequence number associated with m by p_x when it broadcast the message FORWARD_MSG$(msg.m, -, -, -, -)$. This last field is crucial in the scd-delivery by the process p_i of a message set containing m.

Local variables at a process p_i. Each process p_i manages the following local variables.

- $buffer_i$: buffer (init. empty) where are stored quadruplets containing messages that have been fifo-delivered but not yet scd-delivered in a message set.
- $to\_deliver_i$: set of quadruplets containing messages to be scd-delivered.
- sn_i: local logical clock (initialized to 0), which increases by step 1 and measures the local progress of p_i. Each application message scd-broadcast by p_i is identified by a pair $\langle i, sn\rangle$, where sn is the current value of sn_i.
- $clock_i[1..n]$: array of logical dates; $clock_i[j]$ is the greatest date x such that the application message m identified $\langle x, j\rangle$ has been scd-delivered by p_i.

Protocol message. The algorithm uses a single type of protocol message denoted FORWARD_MSG(m, sd, sn, f, sn_f). Such a message is made up of five fields: the first field is an associated application message m, the second and third form a pair $\langle sd, sn\rangle$ that is the identity of the application message and the fourth and fifth form a pair $\langle f, sn_f\rangle$ that describes the local progress (as captured by sn_f) of the forwarder process p_f when it forwarded this protocol message to the other processes by invoking fifo_broadcast FORWARD_MSG$(m, sd, sn_{sd}, f, sn_f)$ (line 11).

Operation scd_broadcast(). When p_i invokes the operation scd_broadcast(m), where m is an application message, it sends the protocol message FORWARD_MSG(m, i, sn_i, i, sn_i) to itself (this simplifies the writing of the algorithm), and waits until it has no more message from itself pending in $buffer_i$, which means it has scd-delivered a set containing m (lines 19 and 20).

Uniform fifo-broadcast of a message FORWARD_MSG**.** When a process p_i fifo-delivers a protocol message FORWARD_MSG$(m, sd, sn_{sd}, f, sn_f)$, it first invokes the internal operation forward(m, sd, sn_{sd}, f, sn_f). In addition to other statements, the first fifo-delivery of such a message by a process p_i entails its participation in the uniform reliable fifo-broadcast of this message (lines 5 and 11). In addition to the invocation of forward(), the fifo-delivery of FORWARD_MSG() invokes also try_deliver(), which strives to scd-deliver a message set (lines 4).

The core of the algorithm. Expressed with the relations $\mapsto_i$, $1 \leq i \leq n$, introduced in Section 2, the main issue of the algorithm is to ensure that, if there are two message m and m' and a process p_i such that $m \mapsto_i m'$, then there is no p_j such that $m' \mapsto_j m$.

To this end, a process p_i is allowed to scd-deliver a message m before a message m' only if it knows that a majority of processes p_j have fifo-delivered a protocol message FORWARD_MSG$(m, -, -, -, -)$ before a protocol message FORWARD_MSG$(m', -, -, -, -)$; p_i knows it either (i) because it fifo-delivered from p_j a message FORWARD_MSG$(m, -, -, -, -)$ but not yet a message FORWARD_MSG$(m', -, -, -, -)$, or (ii) because it fifo-delivered from p_j both FORWARD_MSG$(m, -, -, -, snm)$ and FORWARD_MSG$(m', -, -, -, snm')$ and the sending date snm is smaller than the sending date snm'. The MS-Ordering property follows then from the impossibility that a majority of processes "sees m before m'", while another majority "sees m' before m".

Internal operation forward(). This operation can be seen as an enrichment (with the fields f and sn_f) of the reliable fifo-broadcast implemented by the protocol messages FORWARD_MSG$(m, sd, sn_{sd}, -, -)$. Considering such a message FORWARD_MSG$(m, sd, sn_{sd}, f, sn_f)$, m was scd-broadcast by p_{sd} at its local time sn_{sd}, and relayed by the forwarding process p_f at its local time sn_f. If $sn_{sd} \leq clock_i[sd]$, p_i has already scd-delivered a message set containing m (see lines 18 and 20). If $sn_{sd} > clock_i[sd]$, there are two cases defined by the predicate of line 6.

- No quadruplet $qdplt$ in $buffer_i$ is such that $qdplt.msg = m$. In this case, p_i creates a quadruplet associated with m, and adds it to $buffer_i$ (lines 8-10). Then, p_i participates in the fifo-broadcast of m (line 11) and records its local progress by increasing sn_i (line 12).
- There is a quadruplet $qdplt$ in $buffer_i$ associated with m, i.e., $qdplt = \langle m, -, -, -\rangle \in buffer_i$. In this case, p_i assigns sn_f to $qdplt.cl[f]$ (line 7), thereby indicating that m was known and forwarded by p_f at its local time sn_f.

```
operation scd_broadcast(m) is
(1)  send FORWARD_MSG(m, sn_i, i, sn_i, i) to itself;
(2)  wait(∄ qdplt ∈ buffer_i : qdplt.sd = i).

when the message FORWARD_MSG(m, sd, sn_sd, f, sn_f) is fifo-delivered do   % from p_f
(3)  forward(m, sd, sn_sd, f, sn_f);
(4)  try_deliver().

procedure forward(m, sd, sn_sd, f, sn_f) is
(5)  if (sn_sd > clock_i[sd])
(6)    then if (∃ qdplt ∈ buffer_i : qdplt.sd = sd ∧ qdplt.sn = sn_sd)
(7)           then qdplt.cl[f] ← sn_f
(8)           else threshold[1..n] ← [∞, ..., ∞]; threshold[f] ← sn_f;
(9)                let qdplt ← ⟨m, sd, sn_sd, threshold[1..n]⟩;
(10)               buffer_i ← buffer_i ∪ {qdplt};
(11)               fifo_broadcast FORWARD_MSG(m, sd, sn_sd, i, sn_i);
(12)               sn_i ← sn_i + 1
(13)       end if
(14) end if.

procedure try_deliver() is
(15) let to_deliver_i ← {qdplt ∈ buffer_i : |{f : qdplt.cl[f] < ∞}| > n/2};
(16) while (∃ qdplt ∈ to_deliver_i, ∃ qdplt' ∈ buffer_i \ to_deliver_i : |{f : qdplt.cl[f] < qdplt'.cl[f]}| ≤ n/2) do
       to_deliver_i ← to_deliver_i \ {qdplt} end while;
(17) if (to_deliver_i ≠ ∅)
(18)   then for each qdplt ∈ to_deliver_i do clock_i[qdplt.sd] ← max(clock_i[qdplt.sd], qdplt.sn) end for;
(19)        buffer_i ← buffer_i \ to_deliver_i;
(20)        mset ← {m : ∃ qdplt ∈ to_deliver_i : qdplt.msg = m}; scd_deliver(mset)
(21) end if.
```

Algorithm 1: An implementation of SCD-broadcast in $\mathcal{CAMP}_{n,t}[t < n/2]$ (code for p_i)

Internal operation try_deliver(). When a process p_i executes try_deliver(), it first computes the set $to\_deliver_i$ of the quadruplets $qdplt$ containing application messages m which have been seen by a majority of processes (line 15). From p_i's point of view, a message has been seen by a process p_f if $qdplt.cl[f]$ has been set to a finite value (line 7).

As indicated in a previous paragraph, if a majority of processes received first a message FORWARD_MSG carrying m' and later another message FORWARD_MSG carrying m, it might be that some process p_j scd-delivered a set containing m' before scd-delivering a set containing m. Therefore, p_i must avoid scd-delivering a set containing m before scd-delivering a set containing m'. This is done at line 16, where p_i withdraws the quadruplet $qdplt$ corresponding to m if it can not deliver m' yet (i.e. the corresponding $qdplt'$ is not in $to\_deliver_i$) or it does not have the proof that the situation cannot happen, i.e. no majority of processes saw the message corresponding to $qdplt$ before the message corresponding to $qdplt'$ (this is captured by the predicate $|\{f : qdplt.cl[f] < qdplt'.cl[f]\}| \leq \frac{n}{2}$).

If $to\_deliver_i$ is not empty after it has been purged (lines 16-17), p_i computes a message set to scd-deliver. This set $mset$ contains all the application messages in the quadruplets of $to\_deliver_i$ (line 20). These quadruplets are withdrawn from $buffer_i$ (line 18). Moreover, before this scd-delivery, p_i needs to updates $clock_i[x]$ for all the entries such that $x = qdplt.sd$ where $qdplt \in to\_deliver_i$ (line 18). This update is needed to ensure that the future uses of the predicate of line 17 are correct.

Theorem 1. *Algorithm* 1 *implements the* SCD-broadcast *communication abstraction in $\mathcal{CAMP}_{n,t}[t < n/2]$. Moreover, each invocation of the operation* scd_broadcast() *requires $O(n^2)$ protocol messages. If there is an upper bound Δ on messages transfer delays (and local computation times are equal to zero), each* SCD-broadcast *costs at most 2Δ time units.*

4 SCD-BROADCAST IN ACTION (ITS POWER): SNAPSHOT OBJECT

4.1 Snapshot object

Definition. The snapshot object was introduced in [1, 3]. A snapshot object is an array $REG[1..m]$ of atomic read/write registers which provides the processes with two operations, denoted write$(r, -)$ and snapshot(). The invocation of write(r, v), where $1 \leq r \leq m$, by a process p_i assigns atomically v to $REG[r]$. The invocation of snapshot() returns the value of $REG[1..m]$ as if it was executed instantaneously. Hence, in any execution of a snapshot object, its operations write() and snapshot() are linearizable.

The underlying atomic registers can be Single-Reader (SR) or Multi-Reader (MR) and Single-Writer (SR) or Multi-Writer (MW). We consider only SWMR and MWMR registers. If the registers are SWMR the snapshot is called SWMR snapshot (and we have then $m = n$). Moreover, we always have $r = i$, when p_i invokes write$(r, -)$. If the registers are MWMR, the snapshot object is called MWMR.

Implementations based on read/write registers. Implementations of both SWMR and MWMR snapshot objects on top of read/write atomic registers have been proposed (e.g., [1, 3, 23, 24]). The "hardness" to build snapshot objects in read/write systems and associated lower bounds are presented in the survey [15]. The best algorithm known ([7]) to implement an SWMR snapshot requires $O(n \log n)$ read/write on the base SWMR registers for both write()

```
operation snapshot() is
(1)  done_i ← false; scd_broadcast SYNC(i); wait(done_i);
(2)  return(reg_i[1..m]).

operation write(r, v) is
(3)  done_i ← false; scd_broadcast SYNC(i); wait(done_i);
(4)  done_i ← false; scd_broadcast WRITE(r, v, ⟨tsa_i[r].date + 1, i⟩); wait(done_i).

when the message set { WRITE(r_{j_1}, v_{j_1}, ⟨date_{j_1}, j_1⟩), ..., WRITE(r_{j_x}, v_{j_x}, ⟨date_{j_x}, j_x⟩),
                       SYNC(j_{x+1}), ..., SYNC(j_y) } is scd-delivered do
(5)  for each r such that WRITE(r, −, −) ∈ scd-delivered message set do
(6)      let ⟨date, writer⟩ be the greatest timestamp in the messages WRITE(r, −, −);
(7)      if (tsa_i[r] <_ts ⟨date, writer⟩)
(8)        then let v the value in WRITE(r, −, ⟨date, writer⟩);
(9)             reg_i[r] ← v; tsa_i[r] ← ⟨date, writer⟩
(10)     end if;
(11) end for;
(12) if ∃ℓ : j_ℓ = i then done_i ← true end if.
```

Algorithm 2: Construction of an MWMR snapshot object $\mathcal{CAMP}_{n,t}$[SCD-broadcast] (code for p_i)

and snapshot(). As far as MWMR snapshot objects are concerned, there are implementations where each operation has an $O(n)$ cost[4].

As far as the construction of an SWMR (or MWMR) snapshot object in crash-prone asynchronous message-passing systems where $t < n/2$ is concerned, it is possible to stack two constructions: first an algorithm implementing SWMR (or MWMR) atomic read/write registers (e.g., [5])), and, on top of it, an algorithm implementing an SWMR (or MWMR) snapshot object. This stacking approach provides objects whose operation cost is $O(n^2 \log n)$ messages for SWMR snapshot, and $O(n^2)$ messages for MWMR snapshot. An algorithm based on the same low level communication pattern as the one used in [5], which builds an atomic SWMR snapshot object "directly" (i.e., without stacking algorithms) was recently presented in [14] (the aim of this algorithm is to perform better than the stacking approach in concurrency-free executions).

4.2 An algorithm for atomic MWMR snapshot in $\mathcal{CAMP}_{n,t}$[SCD-broadcast]

Local representation of *REG* at a process p_i. At each register p_i, $REG[1..m]$ is represented by three local variables $reg_i[1..m]$ (data part), plus $tsa_i[1..m]$ and $done_i$ (control part).

- $done_i$ is a Boolean variable.
- $reg_i[1..m]$ contains the current value of $REG[1..m]$, as known by p_i.
- $tsa_i[1..m]$ is an array of timestamps associated with the values stored in $reg_i[1..m]$. A timestamp is a pair made of a local clock value and a process identity. Its initial value is $\langle 0, - \rangle$. The fields associated with $tsa_i[r]$ are denoted $\langle tsa_i[r].date, tsa_i[r].proc \rangle$.

Timestamp-based order relation. We consider the classical lexicographical total order relation on timestamps, denoted $<_{ts}$. Let $ts1 = \langle h1, i1 \rangle$ and $ts2 = \langle h2, i2 \rangle$. We have $ts1 <_{ts} ts2 \stackrel{def}{=} (h1 < h2) \vee ((h1 = h2) \wedge (i1 < i2))$.

[4]Snapshot objects built in read/write models enriched with operations such as Compare&Swap, or LL/SC, have also been considered, e.g., [23, 25]. Here we are interested in pure read/write models.

Algorithm 2: snapshot operation. In this algorithm, a message SYNC(i) is scd-broadcast (line 1) and after reception of this message (line 12), the local value of $reg_i[1..m]$ is returned (line 2). The message SYNC(i) which is scd-broadcast is a pure synchronization message, whose aim is to entail the refreshment of the value of $reg_i[1..m]$ (lines 5-11) which occurs before the setting of $done_i$ to `true` (line 12).

Algorithm 2: write operation. (Lines 3-4) When a process p_i wants to assign a value v to $REG[r]$, it invokes REG.write(r, v). This operation starts by a re-synchronization, as in the snapshot operation, whose side effect is here to provide p_i with an up-to-date value of $tsa_i[r].date$ (line 3). Then, p_i associates the timestamp $\langle tsa_i[r].date + 1, i \rangle$ with v, and scd-broadcasts the data/control message WRITE$(r, v, \langle tsa_i[r].date + 1, i \rangle)$. In addition to informing the other processes on its write of $REG[r]$, this message WRITE() acts as a re-synchronization message, exactly as a message SYNC(i). When this synchronization terminates (i.e., when the Boolean $done_i$ is set to `true`), p_i returns from the write operation.

Algorithm 2: scd-delivery of a set of messages. When process p_i scd-delivers a message set, namely, $\{write(r_{j_1}, v_{j_1}, \langle date_{j_1}, j_1 \rangle), \ldots, write(r_{j_x}, v_{j_x}, \langle date_{j_x}, j_x \rangle), sync(j_{x+1}), \ldots, sync(j_y)\}$ it first looks if there are messages WRITE(). If it is the case, for each register $REG[r]$ for which there are messages WRITE$(r, -, -)$ (line 5), p_i computes the maximal timestamp carried by these messages (line 6), and updates accordingly its local representation of $REG[r]$ (lines 7-10). Finally, if p_i is the sender of one of these messages (WRITE() or SYNC()), $done_i$ is set to `true`, which terminates p_i's re-synchronization (line 12).

Time and Message costs. An invocation of snapshot() involves one invocation of scd_broadcast(), while an invocation of write() involves two. As scd_broadcast() costs $O(n^2)$ protocol messages and 2Δ time units, snapshot() cost the same, and write() costs the double.

THEOREM 2. *Algorithm 2 builds an* MWMR *atomic snapshot object in the model $\mathcal{CAMP}_{n,t}$*[SCD-broadcast]. *The operation* snapshot *costs one* SCD-broaddast, *the* write() *operation costs two.*

```
operation increase() is
(1)  done_i ← false; scd_broadcast PLUS(i); wait(done_i);
(2)  return().

operation decrease() is the same as increase() where PLUS(i) is replaced by MINUS(i).

operation read() is
(3)  done_i ← false; scd_broadcast SYNC(i); wait(done_i);
(4)  return(counter_i).

when the message set { PLUS(j_1), ..., MINUS(j_x), ..., SYNC(j_y), ... } is scd-delivered do
(5)  let p = number of messages PLUS() in the message set;
(6)  let m = number of messages MINUS() in the message set;
(7)  counter_i ← counter_i + p − m;
(8)  if ∃ℓ : j_ℓ = i then done_i ← true end if.
```

Algorithm 3: Construction of an atomic counter in $\mathcal{CAMP}_{n,t}$[SCD-broadcast] (code for p_i)

```
operation increase() is
(1)  lsc_i ← lsc_i + 1;
(2)  scd_broadcast PLUS(i);
(3)  return().

operation decrease() is the same as increase() where PLUS(i) is replaced by MINUS(i).

operation read() is
(4)  wait(lsc_i = 0);
(5)  return(counter_i).

when the message set { PLUS(j_1), ..., MINUS(j_x), ... } is scd-delivered do
(6)  let p = number of messages PLUS() in the message set;
(7)  let m = number of messages MINUS() in the message set;
(8)  counter_i ← counter_i + p − m;
(9)  let c = number of messages PLUS(i) and MINUS(i) in the message set;
(10) lsc_i ← lsc_i − c.
```

Algorithm 4: Construction of a seq. consistent counter in $\mathcal{CAMP}_{n,t}$[SCD-broadcast] (code for p_i)

Comparison with other algorithms. Interestingly, Algorithm 2 is more efficient (from both time and message point of views) than the stacking of a read/write snapshot algorithm running on top of a message-passing emulation of a read/write atomic memory (such a stacking would costs $O(n^2 \log n)$ messages and $O(n\Delta)$ time units, see Section 4.1).

Sequentially consistent snapshot object. When considering Algorithm 2, let us suppress line 1 and line 3 (i.e., the messages SYNC are suppressed). The resulting algorithm implements a sequentially consistent snapshot object. This results from the suppression of the real-time compliance due to the messages SYNC. The operation snapshot() is purely local, hence its cost is 0. The cost of the operation write() is one SCD-broadcast, i.e., 2Δ time units and $O(n^2)$ protocol messages. The proof of this algorithm is left to the reader.

5 SCD-BROADCAST IN ACTION (ITS POWER): COUNTER OBJECT

Definition. Let a *counter* be an object which can be manipulated by three parameterless operations denoted increase(), decrease(), and read(). Let C be a counter. From a sequential specification point of view C.increase() adds 1 to C, C.decrease() subtracts 1 from C, C.read() returns the value of C. As indicated in the Introduction, due to its commutative operations, this object is a good representative of a class of CRDT objects (*conflict-free replicated data type* as defined in [33]).

An algorithm satisfying linearizability. Algorithm 3 implements an atomic counter C. Each process manages a local copy of it denoted $counter_i$. The text of the algorithm is self-explanatory.

The operation read() is similar to the operation snapshot() of the snapshot object. Differently from the write() operation on a snapshot object (which requires a synchronization message SYNC() and a data/synchronization message WRITE()), the update operations increase() and decrease() require only one data/synchronization message PLUS() or MINUS(). This is the gain obtained from the fact that, from a process p_i point of view, the operations increase() and decrease() which appear between two consecutive of its read() invocations are commutative.

THEOREM 3. *Algorithm* 3 *implements an atomic counter.*

An algorithm satisfying sequential consistency. The previous algorithm can be easily modified to obtain a sequentially consistent counter. To this end, a technique similar to the one introduced in [8] can be used to allow the operations increase() and decrease() to have a fast implementation. "Fast" means here that these operations are purely local: they do not require the invoking process to wait in the algorithm implementing them. Differently, the operation read() issued by a process p_i cannot be fast, namely, all the previous increase() and decrease() operations issued by p_i must be applied

```
operation propose(in_i) is
(1)  done_i ← false; scd_broadcast MSG(i, in_i); wait(done_i);
(2)  return(lub(rec_i)).

when the message set { MSG(j_1, v_{j_1}), ..., MSG(j_x, v_{j_x})} is scd-delivered do
(3)  rec_i ← rec_i ∪ {v_{j_1}, ..., v_{j_x}};
(4)  if ∃ℓ : j_ℓ = i then done_i ← true end if.
```

Algorithm 5: Solving Lattice Agreement in $\mathcal{CAMP}_{n,t}$[SCD-broadcast] (code for p_i)

to its local copy of the counter for its invocation of read() terminates (this is the rule known under the name "read your writes").

Algorithm 4 is the resulting algorithm. In addition to $counter_i$, each process manages a synchronization counter lsc_i initialized to 0, which counts the number of increase() and decrease() executed by p_i and not yet locally applied to $counter_i$. Only when lsc_i is equal to 0, p_i is allowed to read $counter_i$.

The cost of an operation increase() and decrease() is 0 time units plus the $O(n^2)$ protocol messages of the underlying SCD-broadcast. The time cost of the operation read() by a process p_i depends on the value of lsc_i. It is 0 when p_i has no "pending" counter operations.

Remark As in [8], using the same technique, it is possible to design a sequentially consistent counter in which the operation read() is fast, while the operations increase() and decrease() are not.

6 SCD-BROADCAST IN ACTION (ITS POWER): LATTICE AGREEMENT TASK

Definition. Let S be a partially ordered set, and $\leq$ its partial order relation. Given $S' \subseteq S$, an upper bound of S' is an element x of S such that $\forall\, y \in S' :\ y \leq x$. The *least upper bound* of S' is an upper bound z of S' such that, for all upper bounds y of S', $z \leq y$. S is called a *semilattice* if all its finite subsets have a least upper bound. Let $\mathsf{lub}(S')$ denotes the least upper bound of S'.

Let us assume that each process p_i has an input value in_i that is an element of a semilattice S. The *lattice agreement* task was introduced in [6] and generalized in [16]. It provides each process with an operation denoted propose(), such that a process p_i invokes propose(in_i) (we say that p_i proposes in_i); this operation returns an element $z \in S$ (we say that it decides z). The task is defined by the following properties, where it is assumed that each non-faulty process invokes propose().

- Validity. If process p_i decides out_i, we have $in_i \leq out_i \leq \mathsf{lub}(\{in_1, \ldots, in_n\})$.
- Containment. If p_i decides out_i and p_j decides out_j, we have $out_i \leq out_j$ or $out_j \leq out_j$.
- Termination. If a non-faulty proposes a value, it decides a value.

Algorithm. Algorithm 5 implements the lattice agreement task. It is a very simple algorithm, which scd-broadcasts one message and waits for its local rception. The text of the algorithm is self-explanatory.

Theorem 4. *Algorithm 5 solves lattice agreement.*

Remark 1. SCD-broadcast can be built on top of read/write registers (see below Theorem 5). It follows that the combination of Algorithm 5 and Algorithm 6 provides us with a pure read/write algorithm solving the lattice agreement task. As far as we know, this is the first algorithm solving lattice agreement, based only on read/write registers.

Remark 2. Similarly to the algorithms implementing snapshot objects and counters satisfying sequential consistency (instead of linearizability), Algorithm 5 uses no message SYNC().

Let us also notice the following. Objects are specified by "witness" correct executions, which are defined by sequential specifications. According to the time notion associated with these sequences we have two consistency conditions: linearizability (the same "physical" time for all the objects) or sequential consistency (a logical time is associated with each object, independently from the other objects). Differently, as distributed tasks are defined by relations from input vectors to output vectors (i.e., without referring to specific execution patterns or a time notion), the notion of a consistency condition (such as linearizability or sequential consistency) is meaningless for tasks.

7 THE COMPUTABILITY POWER OF SCD-BROADCAST (ITS LIMITS)

This section presents an algorithm building SCD-broadcast on top of SWMR snapshot objects. (Such snapshot objects can be easily obtained from MWMR snapshot objects.) Hence, it follows from (a) this algorithm, (b) Algorithm 1, and (c) the impossibility proof to build an atomic register on top of asynchronous message-passing systems where $t \geq n/2$ process may crash [5], that SCD-broadcast cannot be implemented in $\mathcal{CAMP}_{n,t}[t \geq n/2]$, and snapshot objects and SCD-broadcast are computationally equivalent.

7.1 From snapshot to SCD-broadcast

Shared objects. The shared memory is composed of two SWMR snapshot objects. Let ϵ denote the empty sequence.

- $SENT[1..n]$: snapshot object (initialized to $[\emptyset, \ldots, \emptyset]$), such that $SENT[i]$ contains the messages scd-broadcast by p_i.
- $SETS\_SEQ[1..n]$: snapshot object (init. to $[\epsilon, \ldots, \epsilon]$), such that $SETS\_SEQ[i]$ contains the sequence of the sets of messages scd-delivered by p_i.

The notation $\oplus$ is used for the concatenation of a message set at the end of a sequence of message sets.

Local objects. Each process p_i manages the following local objects.

- $sent_i$: local copy of the snapshot object $SENT$.
- $sets\_seq_i$: local copy of the object $SETS\_SEQ$.

```
operation scd_broadcast(m) is
(1) sent_i[i] ← sent_i[i] ∪ {m}; SENT.write(sent_i[i]); progress().

(2) background task T is repeat forever progress() end repeat.

procedure progress() is
(3)   enter_mutex();
(4)   catchup();
(5)   sent_i ← SENT.snapshot();
(6)   to_deliver_i ← (∪_{1≤j≤n} sent_i[j]) \ members(sets_seq_i[i]);
(7)   if (to_deliver_i ≠ ∅)
(8)     then sets_seq_i[i] ← sets_seq_i[i] ⊕ to_deliver_i; SETS_SEQ.write(sets_seq_i[i]);
(9)     scd_deliver(to_deliver_i)
(10)  end if;
(11)  exit_mutex().

procedure catchup() is
(12)  sets_seq_i ← SETS_SEQ.snapshot();
(13)  while (∃j, set : set is the first set in sets_seq_i[j] : set ⊈ members(sets_seq_i[i])) do
(14)        to_deliver_i ← set \ members(sets_seq_i[i]);
(15)        sets_seq_i[i] ← sets_seq_i[i] ⊕ to_deliver_i; SETS_SEQ.write(sets_seq_i[i]);
(16)        scd_deliver(to_deliver_i)
(17)  end while.
```

Algorithm 6: An implementation of SCD-broadcast on top of snapshot objects (code for p_i)

- $to\_deliver_i$: auxiliary variable whose aim is to contain the next message set that p_i has to scd-deliver.

The function members(set_seq) returns the set of all the messages contained in set_seq.

Description of Algorithm 6. When a process p_i invokes scd_broadcast(m), it adds m to $sent_i[i]$ and $SENT[i]$ to inform all the processes on the scd-broadcast of m. It then invokes the internal procedure progress() from which it exits once it has a set containing m (line 1).

A background task T ensures that all messages will be scd-delivered (line 2). This task invokes repeatedly the internal procedure progress(). As, locally, both the application process and the underlying task T can invoke progress(), which accesses the local variables of p_i, those variables are protected by a local fair mutual exclusion algorithm providing the operations enter_mutex() and exit_mutex() (lines 3 and 11).

The procedure progress() first invokes the internal procedure catchup(), whose aim is to allow p_i to scd-deliver sets of messages which have been scd-broadcast and not yet locally scd-delivered.

To this end, catchup() works as follows (lines 12-17). Process p_i first obtains a snapshot of $SETS\_SEQ$, and saves it in $sets\_seq_i$ (line 12). This allows p_i to know which message sets have been scd-delivered by all the processes; p_i then enters a "while" loop to scd-deliver as many message sets as possible according to what was scd-delivered by the other processes. For each process p_j that has scd-delivered a message set set containing messages not yet scd-delivered by p_i (predicate of line 13), p_i builds a set $to\_deliver_i$ containing the messages in set that it has not yet scd-delivered (line 14), and locally scd-delivers it (line 16). This local scd-delivery needs to update accordingly both $sets\_seq_i[i]$ (local update) and $SETS\_SEQ[i]$ (global update).

When it returns from catchup(), p_i strives to scd-deliver messages not yet scd-delivered by the other processes. To this end, it first obtains a snapshot of $SENT$, which it stores in $sent_i$ (line 5). If there are messages that can be scd-delivered (computation of $to\_deliver_i$ at line 6, and predicate at line 7), p_i scd-delivers them and updates $sets\_seq_i[i]$ and $SETS\_SEQ[i]$ (lines 7-9) accordingly.

Theorem 5. *Algorithm* 6 *implements* SCD-Broadcast *in the classical wait-free read/write model* $\mathcal{CARW}_{n,t}[t < n]$.

8 CONCLUSION

What was the paper on? This paper has introduced a new communication abstraction, suited to asynchronous message-passing systems where computing entities (processes) may crash. Denoted SCD-broadcast, it allows processes to broadcast messages and deliver *sets of messages* (instead of delivering each message one after the other). More precisely, if a process p_i delivers a set of messages containing a message m, and later delivers a set of messages containing a message m', no process p_j can deliver a set of messages containing m' before a set of messages containing m. Moreover, there is no local constraint imposed on the processing order of the messages belonging to a same message set. SCD-broadcast has the following noteworthy features:

- It can be implemented in asynchronous message passing systems where any minority of processes may crash. Its costs are upper bounded by twice the network latency (from a time point of view) and $O(n^2)$ (from a message point of view).
- Its computability power is the same as the one of atomic read/write register (anything that can be implemented in asynchronous read/write systems can be implemented with SCD-broadcast).
- When interested in the implementation of a concurrent object O, a simple weakening of the SCD-broadcast-based atomic implementation of O provides us with an SCD-broadcast-based implementation satisfying sequential consistency (moreover, the sequentially consistent implementation is more efficient than the atomic one).

On programming languages for distributed computing. Differently from sequential computing for which there are plenty of high level languages (each with its idiosyncrasies), there is no specific language for distributed computing. Instead, addressing distributed settings is done by the enrichment of sequential computing languages with high level communication abstractions. When considering asynchronous systems with process crash failures, *total order broadcast* is one of them. SCD-broadcast is a candidate to be one of them, when one has to implement read/write solvable objects and distributed tasks.

ACKNOWLEDGMENTS

This work was partially supported by the Franco-German DFG-ANR Project 14-CE35-0010-02 DISCMAT (devoted to connections mathematics/distributed computing) and the French ANR project 16-CE40-0023-03 DESCARTES (devoted to layered and modular structures in distributed computing).

REFERENCES

[1] Afek Y., Attiya H., Dolev D., Gafni E., Merritt M. and Shavit N., Atomic snapshots of shared memory. *JACM*, 40(4):873-890 (1993)

[2] Ahamad M., Neiger G., Burns J.E., Hutto P.W., and Kohli P. Causal memory: definitions, implementation and programming. *Distributed Computing*, 9:37-49 (1995)

[3] Anderson J., Multi-writer composite registers. *Distributed Computing*, 7(4):175-195 (1994)

[4] Aspnes J. and Herlihy M., Wait-free data structures in the asynchronous PRAM model. *Proc. 2nd ACM Symposium on Parallel algorithms and architectures (SPAA'00)*, ACM Press, pp. 340-349 (1990)

[5] Attiya H., Bar-Noy A. and Dolev D., Sharing memory robustly in message passing systems. *JACM*, 42(1):121-132 (1995)

[6] Attiya H., Herlihy M., and Rachman O., Atomic snapshots using lattice agreement. *Distr. Comp.*, 8:121-132 (1995)

[7] Attiya H. and Rachman O., Atomic snapshots in $O(n \log n)$ operations. *SIAM JC*, 27(2):319-340 (1998)

[8] Attiya H. and Welch J.L., Sequential consistency versus linearizability. *ACM TOCS*, 12(2):91-12 (1994)

[9] Attiya H. and Welch J.L., *Distributed computing: fundamentals, simulations and advanced topics*, (2d Edition), Wiley-Interscience, 414 pages (2004)

[10] Biran O., Moran S., and Zaks S., A combinatorial characterization of the distributed tasks which are solvable in the presence of one faulty processor. *Proc. 7th ACM Symposium on Principles of Distributed Computing (PODC'88)*, ACM Press, pp. 263-275 (1988)

[11] Birman K. and Joseph T. Reliable communication in the presence of failures. *ACM TOCS*, 5(1):47–76 (1987)

[12] Chandra T. and Toueg S., Unreliable failure detectors for reliable distributed systems. *JACM*, 43(2):225-267 (1996)

[13] Chaudhuri S., More choices allow more faults: set consensus problems in totally asynchronous systems. *Information and Computation*, 105(1):132-158 (1993)

[14] Delporte-Gallet C., Fauconnier H., Rajsbaum S., and Raynal M., Implementing snapshot objects on top of crash-prone asynchronous message-passing systems. *Proc. 16th Int'l Conf. on Alg. and Arch. for Par. Proc. (ICA3PP'16)*, Springer LNCS 10048, pp. 341-355 (2016)

[15] Ellen F., How hard is it to take a snapshot? *Proc. 31th Conf. on Current Trends in Theory & Prac. of Comp. S. (SOFSEM'05)*, Springer LNCS 3381, pp. 27-35 (2005)

[16] Faleiro J.M., Rajamani S., Rajan K., Ramalingam G., and Vaswani K., Generalized lattice agreement. *Proc. 31th ACM Symposium on Principles of Distributed Computing (PODC'12)*, ACM Press, pp. 125-134 (2012)

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

[18] Fischer M.J. and Merritt M., Appraising two decades of distributed computing theory research. *Distributed Computing*, 16(2-3):239-247 (2003)

[19] Herlihy M.P. and Shavit N., The Art of Multiprocessor Programming. *Morgan Kaufmann Pub.*, 508 pages (2008)

[20] Herlihy M. P. and Wing J. M., Linearizability: a correctness condition for concurrent objects. *ACM TOPLAS*, 12(3):463-492 (1990)

[21] Imbs D., Mostéfaoui A., Perrin M., and Raynal M., Set-Constrained Delivery Broadcast: Definition, Abstraction Power, and Computability Limits. *[Research Report] LIF, UniversitÃľ Aix-Marseille; LINA-University of Nantes; IMDEA Software Institute; Institut Universitaire de France; IRISA, UniversitÃľ de Rennes.*, 2017. ãĂĹhal-01540010ãĂĽ

[22] Imbs D., Mostéfaoui A., Perrin M., and Raynal M., Which broadcast abstraction captures k-set agreement?
Proc. 31th Int'l Symposium on Distributed Computing (DISC'17), to appear in LIPICs (2017)

[23] Imbs D. and Raynal M., Help when needed, but no more: efficient read/write partial snapshot. *Journal of Parallel and Distributed Computing*, 72(1):1-12 (2012)

[24] Inoue I., Chen W., Masuzawa T. and Tokura N., Linear time snapshots using multi-writer multi-reader registers. *Proc. 8th Int'l Workshop on Distributed Algorithms (WDAG'94)*, Springer LNCS 857, pp. 130-140 (1994)

[25] Jayanti P., An optimal multiwriter snapshot algorithm. *Proc. 37th ACM Symposium on Theory of Computing (STOC'05)*, ACM Press, pp. 723-732 (2005)

[26] Lamport L., How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, C28(9):690–691 (1979)

[27] Lamport L., On interprocess communication, Part I: basic formalism. *Distributed Computing*, 1(2):77-85 (1986)

[28] Moran S. and Wolfstahl Y., Extended impossibility results for asynchronous complete networks. *Information Processing Letters*, 26(3):145-151 (1987)

[29] Raynal M., *Communication and agreement abstractions for fault-tolerant asynchronous distributed systems*. Morgan & Claypool, 251 pages (2010)

[30] Raynal M., *Concurrent programming: algorithms, principles and foundations*. Springer, 515 pages (2013)

[31] Raynal M., Set agreement. *Encyclopedia of Algorithms*, Springer, pp. 1956-1959 (2016)

[32] Raynal M., Schiper A., and Toueg S., The causal ordering abstraction and a simple way to implement it. *Information Processing Letters*, 39:343-351 (1991)

[33] Shapiro M., Preguiça N., Baquero C., and Zawirski M., Conflict-free replicated data types. *Proc. 13th Int'l Symp. on Stabilization, Safety, and Security of Distr. Systems (SSS'11)*, Springer LNCS 6976, pp. 386-400 (2011)

[34] Shavit N. and Touitou D., Software transactional memory. *Distributed Computing*, 10(2):99-116 (1997)