Which Broadcast Abstraction Captures k-Set Agreement?

Damien Imbs, Achour Mostefaoui, Matthieu Perrin, Michel Raynal

Which Broadcast Abstraction Captures k-Set Agreement?*

Damien Imbs[1], Achour Mostéfaoui[2], Matthieu Perrin[3], and Michel Raynal[4]

1 **LIF, Université d'Aix-Marseille & CNRS, Marseille, France**
2 **LINA, Université de Nantes, Nantes, France**
3 **IMDEA Software Institute, Pozuelo de Alarcón, Madrid, Spain**
4 **IRISA, Université de Rennes, Rennes, France, and Institut Universitaire de France, Paris, France**

Abstract

It is well-known that consensus (one-set agreement) and total order broadcast are equivalent in asynchronous systems prone to process crash failures. Considering wait-free systems, this article addresses and answers the following question: which is the communication abstraction that "captures" k-set agreement? To this end, it introduces a new broadcast communication abstraction, called k-BO-Broadcast, which restricts the disagreement on the local deliveries of the messages that have been broadcast (1-BO-Broadcast boils down to total order broadcast). Hence, in this context, $k = 1$ is not a special number, but only the first integer in an increasing integer sequence.

This establishes a new "correspondence" between distributed agreement problems and communication abstractions, which enriches our understanding of the relations linking fundamental issues of fault-tolerant distributed computing.

1998 ACM Subject Classification C.2.4 Distributed Systems – distributed applications, network operating systems, D.4.5 Reliability – fault-tolerance, F.1.1 Models of Computation – computability theory

Keywords and phrases Agreement problem, Antichain, Asynchronous system, Communication abstraction, Consensus, Message-passing system, Partially ordered set, Process crash, Read/write object, k-Set agreement, Snapshot object, Wait-free model, Total order broadcast

Digital Object Identifier 10.4230/LIPIcs.DISC.2017.27

1 Introduction

Agreement problems vs communication abstractions. Agreement objects are fundamental in the mastering and understanding of fault-tolerant crash-prone asynchronous distributed systems. The most famous of them is the *consensus* object. This object provides processes with a single operation, denoted propose(), which allows each process to propose a value and decide on (obtain) a value. The properties defining this object are the following: If a process invokes propose() and does not crash, it decides a value (termination); No two processes decide different values (agreement); The decided value was proposed by a process (validity). This object has been generalized by S. Chaudhuri in [7], under the name *k-set agreement*

* This work has been partially supported by the French ANR project 16-CE40-0023-03 DESCARTES, the French ANR project MACARON (anr-13-js02-0002), and the Franco-German DFG-ANR Project 14-CE35-0010-02 DISCMAT.

Table 1 Associating agreement objects and communication abstractions.

Concurrent object	Communication abstraction
Consensus	Total order broadcast [6]
Snapshot object [1, 2] (and R/W register)	SCD-broadcast [11]
k-set agreement object ($1 \leq k < n$)	k-BO-broadcast (this paper)

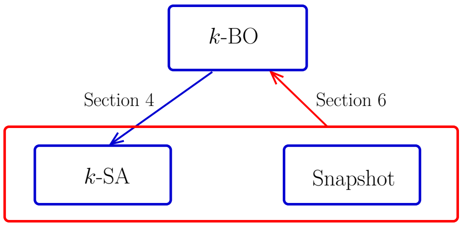


Figure 1 Global picture.

(k-SA), by weakening the agreement property: the processes are allowed to collectively decide up to k different values, i.e., k is the upper bound on the disagreement allowed on the number of different values that can be decided. The smallest value $k = 1$ corresponds to consensus.

On another side, communication abstractions allow processes to exchange data and coordinate, according to some message communication patterns. Numerous communication abstractions have been proposed. Causal message delivery [4, 19], total order broadcast, FIFO broadcast, to cite a few (see the textbooks [3, 15, 16, 17]). In a very interesting way, it appears that some high level communication abstractions "capture" exactly the essence of some agreement objects, see Table 1. The most famous –known for a long time– is the *Total Order broadcast* abstraction which, on one side, allows an easy implementation of a consensus object, and, on an other side, can be implemented from consensus objects. A more recent example is the SCD-Broadcast abstraction that we introduced in [11] (SCD stands for *Set Constrained Delivery*). This communication abstraction allows a very easy implementation of an atomic (Single Writer/Multi Reader or Multi Writer/Multi Reader) snapshot object (as defined in [1]), and can also be implemented from snapshot objects. Hence, as shown in [11], SCD-Broadcast and snapshot objects are the two sides of a same "coin": one side is concurrent object-oriented, the other side is communication-oriented, and none of them is more computationally powerful than the other in asynchronous wait-free systems (where "wait-free" means "prone to any number of process crashes").

Aim and content of the paper. As stressed in [10], Informatics is a science of abstractions. Hence, this paper continues our quest relating communication abstractions and agreement objects. It focuses on k-set agreement in asynchronous wait-free systems. More precisely, the paper introduces the k-BO-broadcast abstraction (BO stands for *Bounded Order*) and shows that it matches k-set agreement in these systems.

k-BO-broadcast is a *Reliable Broadcast* communication abstraction [3, 15, 16, 17], enriched with an additional property which restricts the disagreement on message receptions among the processes. Formally, this property is stated as a constraint on the width of a partial order whose vertices are the messages, and directed edges are defined by local message reception orders. This width is upper bounded by k. For the extreme case $k = 1$, k-BO-broadcast boils down to total order broadcast.

The correspondence linking k-BO-broadcast and k-set agreement, established in the paper, is depicted in Figure 1. The algorithm building k-SA on top of the k-BO-broadcast is surprisingly simple (which is important, as communication abstractions constitute the basic programming layer on top of which distributed applications are built). In the other direction, we show that k-BO-broadcast can be implemented in wait-free systems enriched with k-SA objects and snapshot objects. (Let us recall that snapshot objects do not require additional computability power to be built on top of wait-free read/write systems.) This direction is not as simple as the previous one. It uses an intermediary broadcast communication abstraction, named k-SCD-broadcast, which is a natural and simple generalization of the SCD-broadcast introduced in [11].

Roadmap. The paper is composed of 7 sections. Section 2 presents the basic crash-prone process model, the snapshot object, and k-set agreement. Section 3 defines the k-BO broadcast abstraction and presents a characterization of it. Then, Section 4 presents a simple algorithm implementing k-set agreement on top of the k-BO broadcast abstraction. Section 5 presents another simple algorithm implementing k-BO broadcast on top of the k-SCD-broadcast abstraction. Section 6 presents two algorithms whose combination implements k-SCD-broadcast on top of k-set agreement and snapshot objects. Finally, Section 7 concludes the paper. A global view on the way these constructions are related is presented in Figure 2 of the conclusion.

Due to page limitations, we recommend the reader to refer to the technical report [12] for the proofs of some lemmas and theorems, as well as some considerations about the scope of the results presented here.

2 Process Model, Snapshot, and k-Set Agreement

Process and failure model. The computing model is composed of a set of n asynchronous sequential processes, denoted $p_1, \ldots, p_n$. "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 its possible crash. It is assumed that up to $(n-1)$ processes may crash in a run (wait-free failure model). A process that crashes in a run is said to be *faulty*. Otherwise, it is *non-faulty*. Hence a faulty process behaves as a non-faulty process until it crashes.

Snapshot object. The snapshot object was introduced in [1, 2]. It is an array $REG[1..n]$ of single-writer/multi-reader atomic read/write registers which provides the processes with two operations, denoted write() and snapshot(). Initially, $REG[1..n] = [\bot, \ldots, \bot]$. The invocation of write(v) by a process p_i assigns v to $REG[i]$, and the invocation of snapshot() by a process p_i returns the value of the full array as if the operation had been executed instantaneously. Expressed in another way, the operations write() and snapshot() are atomic, i.e., in any execution of a snapshot object, its operations write() and snapshot() are linearizable.

If there is no restriction on the number of invocations of write() and snapshot() by each process, the snapshot object is multi-shot. Differently, a one-shot snapshot object is such that each process invokes once each operation, first write() and then snapshot(). The one-shot snapshot objects satisfy a very nice and important property, called *Containment*. Let $reg_i[1..n]$ be the vector obtained by p_i, and $view_i = \{\langle reg_i[x], i\rangle \mid reg_i[x] \neq \bot\}$. For any pair of processes p_i and p_j which respectively obtain $view_i$ and $view_j$, we have $(view_i \subseteq view_j) \vee (view_j \subseteq view_i)$.

Implementations of snapshot objects on top of read/write atomic registers have been proposed (e.g., [1, 2, 13, 14]). The "hardness" to build snapshot objects in read/write systems and associated lower bounds are presented in the survey [9].

k-Set agreement. k-Set agreement (k-SA) was introduced by S. Chaudhuri in [7] (see [18] for a survey of k-set agreement in various contexts). Her aim was to investigate the impact of the maximal number of process failures (t) on the agreement degree (k) allowed to the processes, where the smaller the value of k, the stronger the agreement degree. The maximal agreement degree corresponds to $k = 1$ (consensus).

k-SA is a one-shot agreement problem, which provides the processes with a single operation denoted $\mathsf{propose}()$. When a process p_i invokes $\mathsf{propose}(v_i)$, we say that it "proposes value v_i". This operation returns a value v. We then say that the invoking process "decides v", and "v is a decided value". k-SA is defined by the following properties.

- Validity. If a process decides a value v, v was proposed by a process.
- Agreement. At most k different values are decided by the processes.
- Termination. Every non-faulty process that invoked $\mathsf{propose}()$ decides a value.

Repeated k-set agreement. This agreement abstraction is a simple generalization of k-set agreement, which aggregates a sequence of k-set agreement instances into a single object. Hence given such an object $RKSA$, a process p_i invokes sequentially $RKSA.\mathsf{propose}(sn_i^1, v_i^1)$, then $RKSA.\mathsf{propose}(sn_i^2, v_i^2)$, ..., $RKSA.\mathsf{propose}(sn_i^x, v_i^x)$, etc, where $sn_i^1, sn_i^2, \ldots, sn_i^x, \ldots$ are increasing (not necessarily consecutive) sequence numbers, and v_i^x is the value proposed by p_i to the instance number sn_i^x. Moreover, the sequences of sequence numbers used by two processes are sub-sequences of 0, 1, 2, etc., but are not necessarily the same sub-sequence. For each sequence number sn, the invocations of $RKSA.\mathsf{propose}(sn, v_i)$ verify the three properties of k-set agreement.

3 The k-BO-Broadcast Abstraction

Communication operations. The k-Bounded Ordered broadcast (k-BO-Broadcast) abstraction provides the processes with two operations, denoted $\mathsf{kbo\_broadcast}()$ and $\mathsf{kbo\_deliver}()$. The first operation takes a message as input parameter. The second one returns a message to the process that invoked it. Using a classical terminology, when a process invokes $\mathsf{kbo\_broadcast}(m)$, we say that it "kbo-broadcasts the message m". Similarly, when it invokes $\mathsf{kbo\_deliver}()$ and obtains a message m, we say that it "kbo-delivers m"; in the operating system parlance, $\mathsf{kbo\_deliver}()$ can be seen as an *up call* (the messages kbo-delivered are deposited in a buffer, which is accessed by the application according to its own code).

The partial order $\mapsto$. An *antichain* is a subset of a partially ordered set such that any two elements in the subset are incomparable, and a *maximum antichain* is an antichain that has the maximal cardinality among all antichains. The *width* of a partially ordered set is the cardinality of a maximum antichain.

Let $\mapsto_i$ be the local message delivery order at a process p_i defined as follows: $m \mapsto_i m'$ if p_i kbo-delivers the message m before it kbo-delivers the message m'. Let $\mapsto \stackrel{def}{=} \cap_i \mapsto_i$. This relation defines a partially ordered set relation which captures the order on message kbo-deliveries on which all processes agree. In the following, we use the same notation ($\mapsto$) for the relation and the associated partially ordered graph. Let $\mathsf{width}(\mapsto)$ denote the width of the partially ordered graph $\mapsto$.

Properties on the operations. k-BO-broadcast is defined by the following set of properties, where we assume –without loss of generality– that all the messages that are kbo-broadcast are different and every non-faulty process keeps invoking the operation kbo_deliver() forever.

- KBO-Validity. Any message kbo-delivered has been kbo-broadcast by a process.
- KBO-Integrity. A message is kbo-delivered at most once by each process.
- KBO-Bounded. $\mathsf{width}(\mapsto) \leq k$.
- KBO-Termination-1. If a non-faulty process kbo-broadcasts a message m, it terminates its kbo-broadcast invocation and kbo-delivers m.
- KBO-Termination-2. If a process kbo-delivers a message m, every non-faulty process kbo-delivers m.

The reader can easily check that the Validity, Integrity, Termination-1, and Termination-2 properties define *Uniform Reliable Broadcast*.

The KBO-Bounded property, which gives its meaning to k-BO-broadcast, is new. Two processes p_i and p_j *disagree* on the kbo-deliveries of the messages m and m' if p_i kbo-delivers m before m', while p_j kbo-delivers m' before m. Hence we have neither $m \mapsto m'$ nor $m' \mapsto m$.

k-Bounded Order captures the following constraint: processes can disagree on message sets of size at most k. (Said differently, there is no message set ms such that $|ms| > k$ and for each pair of messages $m, m' \in ms$, there are two processes p_i and p_j that disagree on their kbo-delivery order.) Let us consider the following example to illustrate this constraint.

An example. Let m_1, m_2, m_3, m_4, m_5, and m_6, be messages that have been kbo-broadcast by different processes. Let us consider the following sequences of kbo-deliveries by the 3 processes p_1, p_2 and p_3.

- at p_1: m_1, m_2, m_3, m_4, m_5, m_6.
- at p_2: m_2, m_1, m_5, m_3, m_4, m_6.
- at p_3: m_2, m_3, m_1, m_5, m_4, m_6.

The set of messages $\{m_1, m_2\}$ is such that processes disagree on their kbo-delivery order. We have the same for the sets of messages $\{m_1, m_3\}$ and $\{m_4, m_5\}$. It is easy to see that, when considering the set $\{m_1, m_2, m_3, m_4\}$, the message m_4 does not create disagreement with respect to the messages in the set $\{m_1, m_2, m_3\}$.

The reader can check that there is no set of cardinality greater than $k = 2$ such that processes disagree on all the pairs of messages they contain. On the contrary, when looking at the message sets of size ≤ 2, disagreement is allowed, as shown by the sets of messages $\{m_1, m_2\}$, $\{m_1, m_3\}$, and $\{m_4, m_5\}$. In conclusion, these sequences of kbo-deliveries are compatible with 2-BO broadcast.

Let us observe that if two processes disagree on the kbo-deliveries of two messages m and m', these messages define an antichain of size 2. It follows that 1-BO-broadcast is total order broadcast (which is computationally equivalent to Consensus [6]), while $k = n$ imposes no constraint on message deliveries.

Underlying intuition: the non-deterministic k-TO-channel notion. Let us define the notion of a *non-deterministic* k-TO-*channel* as follows (TO stands for *Total Order*). There are k different broadcast channels, each ensuring total order delivery on the messages broadcast through it. The invocation of kbo_broadcast(m) by a process entails a broadcast on one and only one of these broadcast channels, but the channel is selected by an underlying daemon, and the issuing process never knows which channel has been selected for its invocation.

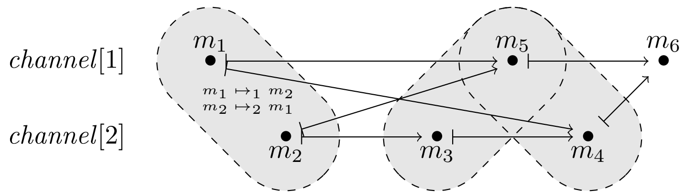
Let us consider the previous example, with $k = 2$. Hence, there are two TO-channels, *channel*[1] and *channel*[2]. As shown by the following figure, they contained the following

operation propose(nb, v) **is**
(1) kbo_broadcast$(\langle nb, v\rangle)$; wait$(\exists\ \langle nb, x\rangle \in decisions_i)$; return$(x)$.

when a message $\langle sn, x\rangle$ **is kbo-delivered do**
(2) **if** $(\langle sn, -\rangle$ never added to $decisions_i)$ **then** $decisions_i$.insert$(\langle sn, x\rangle)$ **end if**.

Algorithm 1 From k-BO-broadcast to repeated k-set agreement.

sequences of messages: $channel[1] = m_1, m_5, m_6$ and $channel[2] = m_2, m_3, m_4$. On this figure, encircled grey areas represent maximum antichains.



It is easy to check that the sequence of messages delivered at any process p_i is a merge of the sequences associated with these two channels.

The assignment of messages to channels is not necessarily unique, it depends on the behavior of the daemon. Considering $k = 3$ and a third channel $channel[3]$, let us observe that the same message kbo-deliveries at p_1, p_2, and p_3, could have been obtained by the following channel selection by the daemon: $channel[1]$ as before, $channel[2] = m3, m4$, and $channel[3] = m2$. Let us observe that, with $k = 3$ and this daemon behavior, the message kbo-delivery $m_3, m_1, m_5, m_4, m_2, m_6$ would also be correct at p_3.

A characterization. The previous non-deterministic k-TO-channel interpretation of k-BO-broadcast is captured by the following characterization theorem.

▶ **Theorem 1.** *A non-deterministic* k-TO-channel *and the* k-BO-broadcast *communication abstraction have the same computational power.*

▶ Remark. It is important to see that k-BO-broadcast and k-TO-channels are not only computability equivalent but are two statements of the very same communication abstraction (there is no way to distinguish them from a process execution point of view).

4 From k-BO-Broadcast to Repeated k-Set Agreement

Algorithm 1 implements repeated k-set agreement in a wait-free system enriched with k-BO-Broadcast. Its simplicity demonstrates the very *high abstraction level* provided by k-BO-Broadcast. All "implementation details" are hidden inside its implementation (which has to be designed only once, and not for each use of k-BO-Broadcast in different contexts). In this sense, k-BO-Broadcast is the abstraction communication which captures the essence of (repeated) k-set agreement.

When a process p_i invokes propose(nb, v), it kbo-broadcasts a message containing the pair $\langle nb, v\rangle$ and waits until a pair $\langle nb, -\rangle$ appears in its local set $decisions_i$ (line 1). Such a pair is added in $decisions_i$ the first time p_i k-BO-delivers a pair $\langle nb, x\rangle$ (line 2). Let us observe that this algorithm is purely based on the k-BO-Broadcast communication abstraction.

▶ **Lemma 2.** *If the invocation of* propose(nb, v) *returns x to a process, some process invoked* propose(nb, x).

▶ **Lemma 3.** *If a non-faulty process invokes* propose$(nb, -)$, *it eventually decides a value* x *such that* $\langle nb, x\rangle$ *is the first (and only) message* $\langle nb, -\rangle$ *it kbo-delivers.*

▶ **Lemma 4.** *The set of values returned by the invocations of* propose$(nb, -)$ *contains at most* k *different values.*

Proof. Let Π_{nb} be the set of processes returning a value from their invocations propose$(nb, -)$. For each $p_i \in \Pi_{nb}$, let $\langle nb, x_i\rangle$ denote the first message $\langle nb, -\rangle$ received by p_i. By Lemma 3, $X_{nb} = \{x_i : p_i \in \Pi_{nb}\}$ is the set of all values returned by the invocations of propose$(nb, -)$.

For any pair x_i and x_j of distinct elements of X_{nb}, we have that p_i kbo-delivered x_i before x_j, and p_j kbo-delivered x_j before x_i. Hence, $\langle nb, x_j\rangle \not\mapsto_i \langle nb, x_i\rangle$ and $\langle nb, x_i\rangle \not\mapsto_j \langle nb, x_j\rangle$, which means $\langle nb, x_i\rangle$ and $\langle nb, x_j\rangle$ are not ordered by $\mapsto$. Therefore, $\{\langle nb, x_i\rangle : p_i \in \Pi_{nb}\}$ is an antichain of $\mapsto$. It then follows from the KBO-Bounded property that $|\{x_i : p_i \in \Pi_{nb}\}| = |\{\langle nb, x_i\rangle : p_i \in \Pi_{nb}\}| \le k$. ◀

▶ **Theorem 5.** *Algorithm 1 implements repeated* k*-set agreement in any system model enriched with the communication abstraction* k-BO-broadcast*.*

5 From k-SCD-Broadcast to k-BO-Broadcast

5.1 The intermediary k-SCD-Broadcast abstraction

This communication abstraction is a simple strengthening of the SCD-Broadcast abstraction introduced in [11], where it is shown that SCD-Broadcast and snapshot objects have the same computability power (SCD stands for Set Constrained Delivery).

SCD-Broadcast: definition. SCD-broadcast consists of two operations scd_broadcast() and scd_deliver(). The first operation takes a message to broadcast as input parameter. The second one returns a non-empty set of messages to the process that invoked it. By a slight abuse of language, we say that a process "scd-delivers a message m" when it delivers a message set ms 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 and that every non-faulty process keeps invoking the operation scd_deliver() forever.

- SCD-Validity. If a process scd-delivers a set containing a message m, then m was scd-broadcast by some process.
- SCD-Integrity. A message is scd-delivered at most once by each process.
- SCD-Ordering. If a process p_i scd-delivers first a message m belonging to a set ms_i and later a message m' belonging to a set $ms'_i \neq ms_i$, then no process scd-delivers first m' in some scd-delivered set ms'_j and later m in some scd-delivered set $ms_j \neq ms'_j$.
- SCD-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.
- SCD-Termination-2. If a process scd-delivers a message set containing m, every non-faulty process scd-delivers a message set containing m.

***k*-SCD-Broadcast: definition.** This communication abstraction is SCD-Broadcast strengthened with the following additional property:

- KSCD-Bounded. No set ms kscd-delivered to a process contains more than k messages.

In the following, all properties of k-SCD-broadcast are prefixed by "KSCD".

operation $\mathsf{kbo\_broadcast}(v)$ **is** $\mathsf{kscd\_broadcast}(m)$.

when a message set ms **is kscd-delivered do for each** $m \in ms$ **do** $\mathsf{kbo\_deliver}(m)$ **end for**.

Algorithm 2 From k-SCD-broadcast to k-BO-broadcast.

An example. Like in Section 3, let m_1, m_2, m_3, m_4, m_5, and m_6 be messages that have been kbo-broadcast by different processes. Let us consider the following sequences of message sets k-scd-delivered by the 3 processes p_1, p_2 and p_3.

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

The processes do not agree on the message sets they k-scd-deliver. For example, p_1 and p_3 k-scd-deliver m_2 in the same set as m_1, whereas p_2 k-scd-deliver m_2 in the same set as m_3. However, at any time, the union of message sets previously k-scd-delivered by any process is part of the following sequence of message sets: $\{m_2\}$, $\{m_1, m_2\}$, $\{m_1, m_2, m_3\}$, $\{m_1, m_2, m_3, m_5\}$, $\{m_1, m_2, m_3, m_4, m_5\}$, $\{m_1, m_2, m_3, m_4, m_5, m_6\}$, which implies the SCD-Ordering property. Moreover, all k-scd-delivered message sets are of size at most $k = 2$.

5.2 From k-SCD-Broadcast to k-BO-Broadcast

Description of the algorithm. Algorithm 2 implements k-BO-Broadcast on top of any system model providing k-SCD-Broadcast. It is an extremely simple self-explanatory algorithm.

► **Theorem 6.** *Algorithm 2 implements* k-BO-broadcast *in any system model enriched with the communication abstraction* k-SCD-broadcast*.*

Proof. k-BO-Validity, k-BO-Integrity, k-BO-Termination-1 and k-BO-Termination-2 are direct consequences of their homonym SCD-broadcast properties.

To prove the additional k-BO-Bounded property, let us consider a message set ms containing at least $(k+1)$ messages. For each process p_i, let fms_i (resp. lms_i) denote the first (resp. last) set containing a message in ms received by p_i. Thanks to the KSCD-Ordering property, there exists a message $fm \in \cap_i\, fms_i$ and a message $lm \in \cap_i\, lms_i$. (Otherwise, we will have messages m and m' such that $m \in fms_i \wedge m \notin fms_j$ and $m' \notin fms_i \wedge m' \in fms_j$.)

Let ums_i denote the union of all the message sets k-scd-delivered by p_i starting with the set including fms_i and finishing with the set including lms_i. As, for each process p_i, ums_i contains at least the $(k+1)$ messages of ms, we have $fms_i \neq lms_i$. Therefore, we have $fm \neq lm$ and $fm \mapsto lm$. It follows that ms cannot be an antichain of $\mapsto$. Consequently, the antichains of $\mapsto$ cannot contain more than k messages, hence $\mathsf{width}(\mapsto) \leq k$. ◄

6 From Repeated k-Set Agreement and Snapshot to k-SCD-Broadcast

6.1 The K2S abstraction

Definition. The following object, denoted K2S, is used by Algorithm 4 to implement k-SCD-broadcast. “K2S” stands for k-set agreement plus two snapshots. A K2S object provides a single operation $\mathsf{k2s\_propose}(v)$ that can be invoked once by each process. Its output is a set of sets whose size and elements are constrained by both k-set agreement and the input size (number of different values proposed by processes). The output $sets_i$ of each process p_i is a

operation k2s_propose(v) **is**
(1) $val_i \leftarrow KSET$.propose(v);
(2) $SNAP1$.write(val_i); $snap1_i \leftarrow SNAP1$.snapshot();
(3) $view_i \leftarrow \{snap1_i[j] \mid snap1_i[j] \neq \bot\}$;
(4) $SNAP2$.write($view_i$); $snap2_i \leftarrow SNAP2$.snapshot();
(5) $sets_i \leftarrow \{snap2_i[j] \mid snap2_i[j] \neq \bot\}$;
(6) return($sets_i$).

Algorithm 3 An implementation of a K2S object.

non-empty set of non-empty sets, called views and denoted *view*, satisfying the following properties. Let *inputs* denote the set of different input values proposed by the processes.

- K2S-Validity. $\forall\ i$: $\forall\ view \in sets_i$: $(m \in view) \Rightarrow (m$ was k2s-proposed by a process).
- Set Size. $\forall\ i$: $1 \leq |sets_i| \leq \min(k, |inputs|)$.
- View Size. $\forall\ i$: $\forall\ view \in sets_i$: $(1 \leq |view| \leq \min(k, |inputs|))$.
- Intra-process Inclusion.$\forall\ i$: $\forall\ view1, view2 \in sets_i$: $view1 \subseteq view2 \vee view2 \subseteq view1$.
- Inter-process Inclusion. $\forall\ i, j$: $sets_i \subseteq sets_j \vee sets_j \subseteq sets_i$.
- K2S-Termination. If a non-faulty process p_i invokes k2s_propose(), it returns a set $sets_i$.

Algorithm. Algorithm 3 implements a K2S object. It uses an underlying k-set agreement object $KSET$, and two one-shot snapshot objects denoted $SNAP1$ and $SNAP2$.

- Phase 1 (line 1). When a process p_i invokes k2s_propose(v), it first proposes v to the k-set agreement object, from which it obtains a value val_i (line 1).
- Phase 2 (lines 2-3). Then p_i writes val_i in the first snapshot object $SNAP1$, reads its content, saves it in $snap1_i$, and computes the set of values ($view_i$) that, from its point of view, have been proposed to the k-set agreement object.
- Phase 3 (lines 4-6). Process p_i then writes its view $view_i$ in the second snapshot object $SNAP2$, reads its value, and computes the set of views ($sets_i$) obtained – as far as it knows – by the other processes. Process p_i finally returns this set of views $sets_i$.

▶ **Theorem 7.** *Algorithm* 3 *satisfies the properties defining a* K2S *object.*

Repeated K2S. In the following we consider a repeated K2S object, denoted KSS. A process p_i invokes KSS.k2s_propose(r, v) where v is the value it proposes to the instance number r. The instance numbers used by each process are increasing (but not necessarily consecutive). Hence, two snapshot objects are associated with every K2S instance, and line 1 of Algorithm 3 becomes $KSET$.propose(r, v).

6.2 From k-Set Agreement and Snapshot to k-SCD-Broadcast

Algorithm 4 builds the k-SCD-Broadcast abstraction on top of k-set agreement and snapshot objects.

Shared objects and local objects.

- The processes cooperate through two concurrent objects: $MEM[1..n]$, a multishot snapshot object, such that $MEM[i]$ contains the set of messages kscd-broadcast by p_i, and a repeated K2S object denoted KSS.
- A process p_i manages two local copies of MEM denoted $mem1_i$ and $mem2_i$, two auxiliary sets $to\_deliver1_i$ and $to\_deliver2_i$, and a set $delivered_i$, which contains all the messages it has locally kscd-delivered; $mem1_i[i]$ is initialized to an empty set.

operation kscd_broadcast(m) **is**
(1) MEM.write($mem1_i[i] \cup \{m\}$); $mem1_i \leftarrow MEM$.snapshot();
(2) $to\_deliver1_i \leftarrow (\cup_{1\leq j\leq n}\ mem1_i[j]) \setminus delivered_i$; wait($to\_deliver1_i \subseteq delivered_i$).

background task T **is**
(3) **repeat forever**
(4) $prop_i \leftarrow \bot$;
(5) **if** ($seq_i = \epsilon$) **then** $mem2_i \leftarrow MEM$.snapshot();
(6) $to\_deliver2_i \leftarrow (\cup_{1\leq j\leq n}\ mem2_i[j]) \setminus delivered_i$;
(7) **if** ($to\_deliver2_i \neq \emptyset$) **then** $prop_i \leftarrow$ a message $\in to\_deliver2_i$ **end if**
(8) **else** $prop_i \leftarrow$ a message of the first message set of seq_i
(9) **end if**;
(10) **if** ($prop_i \neq \bot$)
(11) **then** $r_i \leftarrow |delivered_i|$; $sets_i \leftarrow KSS$.k2s_propose($r_i, prop_i$); $new\_seq_i \leftarrow \epsilon$;
(12) **while** ($sets_i \neq \{\emptyset\}$) **do**
(13) $min\_set_i \leftarrow$ non-empty set of minimal size in $sets_i$;
(14) $new\_seq_i \leftarrow new\_seq_i \oplus min\_set_i$;
(15) **for each set** $s \in sets_i$ **do** $sets_i \leftarrow (sets_i \setminus \{s\}) \cup \{s \setminus min\_set_i\}$ **end for**
(16) **end while**;
(17) **let** aux_i = all the messages in the sets of new_seq_i;
(18) **for each set** $s \in seq_i$ **do** $s \leftarrow s \setminus aux_i$ **end for**;
(19) $seq_i \leftarrow new\_seq_i \oplus seq_i$; **let** $first_i$ = head(seq_i); **let** $rest_i$ = tail(seq_i);
(20) kscd_deliver($first_i$); $delivered_i \leftarrow delivered_i \cup first_i$; $seq_i \leftarrow rest_i$
(21) **end if**
(22) **end repeat**.

Algorithm 4 From k-set agreement and snapshot objects to k-SCD-broadcast (code for p_i).

- r_i denotes the next round number that p_i will execute; $sets_i$ is a local set whose aim is to contain the set of message sets returned by the last invocation of a K2S object.
- Each process p_i manages two sequences of messages sets, both initialized to ϵ (empty sequence), denoted seq_i and new_seq_i; head(sq) returns the first element of the sequence sq, and tail(sq) returns sq without its first element; $\oplus$ denotes sequence concatenation. The aim of the local sequence new_seq_i is to contain a sequence of message sets obtained from $sets_i$ (last invocation of a K2S object) such that no message belongs to several sets. As far as seq_i is concerned, we have the following (at line 19 of Algorithm 4). Let $seq_i = ms_1,\ ms_2,\ \cdots, ms_\ell$, where $1 \leq \ell \leq k$ and each ms_x is a message set. This sequence can be decomposed into two (possibly empty) sub-sequences $ms_1,\ ms_2,\ \cdots, ms_y$ and $ms_{y+1} \cdots, ms_\ell$ such that:
  - $ms_1,\ ms_2,\ \cdots, ms_y$ can be in turn decomposed as follows: $(ms_1 \cup ms_2 \cup \cdots \cup ms_a), (ms_{a+1} \cup ms_{a+2} \cup \cdots \cup ms_b),\ \cdots, (ms_c \cup \cdots \cup ms_y)$ where each union set (e.g., $ms_{a+1} \cup ms_{a+2} \cup \cdots \cup ms_b$) is a message set that has been kscd-delivered by some process (some union sets can contain a single message set)[1].
  - For each $x :\ y+1 \leq x \leq \ell :\ m_x$ is a message set whose messages have not yet been kscd-delivered by a process.

Operation kscd_broadcast(). When it invokes kscd_broadcast(), a process p_i first adds m to the shared memory MEM, which contains all the messages it has already kscd-broadcast (line 1). Then p_i reads atomically the whole content of MEM, which is saved in $mem1_i$ (line 1). Then, p_i computes the set of messages not yet locally kscd-delivered and waits

[1] Let us remark that it is possible that, while a process kscd-delivered the message set $ms = ms_1 \cup ms_2 \cup \cdots \cup ms_a$, another process kscd-delivered the messages in ms in several messages sets, e.g., first the message set $ms_1 \cup ms_2 \cup ms_3$ and then the message set $ms_4 \cup \cdots \cup ms_a$.

until all these messages appear in kscd-delivered message sets (line 2). Let us notice that, it follows from these statements, that a process has kscd-delivered its previous message when it issues its next kscd_broadcast().

Underlying task T. This task is the core of the algorithm. It consists of an infinite loop, which implements a sequence of asynchronous rounds (lines 11-20). Each process p_i executes a sub-sequence of non-necessarily consecutive rounds. Moreover, any two processes do not necessarily execute the same sub-sequence of rounds. The current round of a process p_i is defined by the value of $|delivered_i|$ (number of messages already locally kscd-delivered).

The progress of a process from a round r to its next round $r' > r$ depends on the size of the message set (denoted $first_i$ in the algorithm, line 20) it kscd-delivers at the end of round r ($delivered_i$ becomes then $delivered_i \cup first_i$). The message set $first_i$ depends on the values returned by the K2S object associated with the round r, as explained below.

Underlying task T: proposal computation. (Lines 4-9) Two rounds executed by a process p_i are separated by the local computation of a message value ($prop_i$) that p_i will propose to the next K2S object. This local computation is as follows (lines 5-9), where seq_i (computed at lines 18-20) is a sequence of message sets that, after some "cleaning", are candidates to be locally kscd-delivered. There are two cases.

- Case 1: $seq_i = \epsilon$. In this case (similarly to line 2) p_i computes the set of messages ($to\_deliver2_i$) it sees as kscd-broadcast but not yet locally kscd-delivered (lines 5-6). If $to\_deliver2_i \neq \emptyset$, a message of this set becomes its proposal $prop_i$ for the K2S object associated with the next round (line 7). Otherwise, we have $prop_i = \epsilon$, which, due to the predicate of line 10, entails a new execution of the loop (skipping lines 11-20).
- Case 2: $seq_i \neq \emptyset$. In this case, $prop_i$ is assigned a message of the first set of seq_i (line 8).

Underlying task T: benefiting from a K2S object to kscd-deliver a message set. (Lines 11-20) If a proposal has been previously computed (predicate of line 10), p_i executes its next round, whose number is $r_i = |delivered_i|$. The increase step of $|delivered_i|$ can vary from round to round, and can be any value $\ell \in [1..k]$ (lines 14 and 15). As already indicated, while the round numbers have a global meaning (the same global sequence of rounds is shared by all processes), each process executes a subset of this sequence (as defined by the increasing successive values of $|delivered_i|$). Despite the fact processes skip/execute different rounds, once combined with the use of K2R objects, round numbers allow processes to synchronize in a consistent way. This round synchronization property is captured by Lemmas 11-12.

From an operational point of view, a process starts a round with the invocation KSS.k2s_propose$(r_i, prop_i)$ where $r_i = |delivered_i|$, which returns a set of message sets $sets_i$ (line 11). Then ("while" loop at lines 12-16), p_i builds from the message sets belonging to $sets_i$ a sequence of message sets new_seq_i, that will be used to extract the next message set kscd-delivered by p_i (lines 17-20). The construction of new_seq_i is as follows. Iteratively, p_i takes the smallest set of $sets_i$ (min_set_i, line 13), adds it at the end of new_seq_i (line 14), and purges all the sets of $sets_i$ from the messages in min_set_i (line 15), so that no message will locally appear in two different messages sets of new_seq_i.

When new_seq_i is built, p_i first purges all the sets of the sequence seq_i from the messages in new_seq_i (lines 17-18), and adds then new_seq_i at the front of seq_i (line 19). Finally, p_i kscd-delivers the first message set of seq_i, and updates $delivered_i$ and seq_i (lines 20).

6.3 Proof of the algorithm

▶ **Lemma 8.** *A message set kscd-delivered (line 20) contains at most k messages.*

▶ **Lemma 9.** *If a process kscd-delivers a message set containing a message m, m was kscd-broadcast by a process.*

Notations.

- $msg\_set_i(r)$ = message set kscd-delivered by process p_i at round r if p_i participated in it, and $\emptyset$ otherwise.
- $seq_i(r)$ = value of seq_i at the end of the last round $r' \leq r$ in which p_i participated.
- $msgs_i(r, r')$ = set of messages contained in message sets kscd-delivered by p_i between rounds r (included) and $r' > r$ (not included), i.e. $msgs_i(r, r') = \bigcup_{r \leq r'' < r'} msg\_set_i(r'')$.
- $KSS(r)$ = K2S instance accessed by $KSS.\mathsf{k2s\_propose}(r, -)$ (line 11).
- $sets_i(r)$ = set of message sets obtained by p_i from $KSS[r]$.

▶ **Lemma 10.** *Let p_i and p_j be two processes that terminate round r, with $|msg\_set_i(r)| \leq |msg\_set_j(r)|$. Then* (i) *$msg\_set_i(r) \subseteq msg\_set_j(r)$, and* (ii) *there is a prefix $pref_i$ of $seq_i(r)$ such that $msg\_set_j(r) = msg\_set_i(r) \cup (\bigcup_{msg\_set\ \in\ pref_i} msg\_set)$.*

Proof. Let p_i and p_j be two processes that kscd-deliver the message sets $msg\_set_i(r)$ and $msg\_set_j(r)$, respectively, these sets being such that $|msg\_set_i(r)| \leq |msg\_set_j(r)|$. Let us observe that, as both p_i and p_j invoked $KSS.\mathsf{k2s\_propose}(r, -)$ (lines 11 and 20), we have $sets_i(r) \subseteq sets_j(r)$ or $sets_j(r) \subseteq sets_i(r)$ (Inter-process Inclusion).

As $|msg\_set_i(r)| \leq |msg\_set_j(r)|$, it follows from the Inter-process and Intra-process inclusion properties of $KSS(r)$, and the definition of $msg\_set_i(r) = \mathit{first}_i = min\_set_i \in sets_i(r)$, and $msg\_set_j(r) = \mathit{first}_j = min\_set_j \in sets_j(r) \subseteq sets_i(r)$, that $msg\_set_i(r) \subseteq msg\_set_j(r)$, which completes the proof of (i).

As far as (ii) is concerned, we have the following. If $msg\_set_i(r) = msg\_set_j(r)$, we have $pref_i = \epsilon$ and the lemma follows. So, let us assume $msg\_set_i(r) \subsetneq msg\_set_j(r)$. As $msg\_set_i(r)$ is the smallest message set of $sets_i(r)$ (lines 13-14 and 19-20), and $msg\_set_j(r)$ is the smallest message set of $sets_j(r)$, it follows that $sets_j(r) \subset sets_i(r)$. The property $msg\_set_j(r) = msg\_set_i(r) \cup (\bigcup_{msg\_set\ \in\ pref_i} msg\_set)$ follows then from the following observation. Let $sets_i(r) = \{s_1, s_2, ..., s_\ell\}$, where $\ell \leq k$ and $s_1 \subsetneq s_2 \subsetneq \cdots \subsetneq s_\ell$. As $sets_j(r) \subset sets_i(r)$, one s_x is $msg\_set_j(r)$. It follows that the union of the sets min_set_i computed by p_i in the while loop of round r (lines 13-15) eventually includes all the messages of $msg\_set_j(r)$, from which we conclude that there is a prefix $pref_i$ of $seq_i(r)$ (lines 12-19, namely a prefix of the sequence new_seq_i, which is defined from the sequence of the sets min_set_i), such that $msg\_set_j(r) = msg\_set_i(r) \cup (\bigcup_{msg\_set\ \in\ pref_i} msg\_set)$, which completes the proof of the lemma. ◀

Lemmas 11-12 capture the global message set delivery synchronization among the processes.

▶ **Lemma 11.** *Let p_i and p_j be two processes that terminate round $r' \geq r + |msg\_set_j(r)|$, and are such that $|msg\_set_i(r)| \leq |msg\_set_j(r)|$. Then* (i) *$msgs_i(r, r + |msg\_set_j(r)|) = msgs_j(r, r+|msg\_set_j(r)|)$, and* (ii) *$p_i$ and p_j will both participate in round $r+|msg\_set_j(r)|$.*

Proof. If $|msg\_set_i(r)| = |msg\_set_j(r)| = \alpha$, both p_i and p_j are such that $|delivered_i| = |delivered_j| = r + \alpha$ when they terminate round r. Consequently, they both proceed from round r to round $r+\alpha$, thereby skipping the rounds from $r+1$ until $r+\alpha-1$. We then have (i) $msgs_i(r, r + |msg\_set_j(r)|) = msg\_set_i(r) = msg\_set_j(r) = msgs_j(r, r + |msg\_set_j(r)|)$, (ii) both p_i and p_j will participate in round $r + |msg\_set_j(r)|$, and the lemma follows.

Hence, let us consider that $|msg\_set_i(r)| = \alpha < |msg\_set_j(r)| = \alpha + \beta$. The next round executed by p_i will be the round $r + \alpha$, while the next round executed by p_j will be the round $r + \alpha + \beta$. Moreover, to simplify and without loss of generality, let us assume that $msg\_set_i(r)$ (resp. $msg\_set_j(r)$) is the smallest (resp. second smallest) message set in the sets of message sets $sets$ output by $KSS(r)$.

According to Lemma 10, after round r, the first element of seq_i is $msg\_set_j(r) \setminus msg\_set_i(r)$. This also applies to any other process that delivered $msg\_set_i(r)$ at round r. At round $r+\alpha$, all these processes will then propose a message in $msg\_set_j(r) \setminus msg\_set_i(r)$. Because of the K2S-Validity property of $KSS(r + \alpha)$, all these processes will then deliver a subset of $msg\_set_j(r) \setminus msg\_set_i(r)$. For the same reason, until round $r + \alpha + \beta$, no process will propose a message not in $msg\_set_j(r) \setminus msg\_set_i(r)$. At round $r + \alpha + \beta$, they will then have delivered all the messages in $msg\_set_j(r) \setminus msg\_set_i(r)$, and they will participate in round $r + \alpha + \beta$, from which the lemma follows. ◀

▶ **Lemma 12.** *Let r be a round in which all the non-faulty processes participate. There is a round r' with $r < r' \leq r + k$ in which all non-faulty processes participate and such that, for any pair of non-faulty processes p_i and p_j, we have $msgs_i(r, r') = msgs_j(r, r')$.*

Proof. As initially $\forall i : |delivered_i| = 0$, KSS.k2s_propose$(0, -)$ is invoked by all non-crashed processes. We prove that there is a round $r \in [1..k]$ in which all the non-crashed processes participate, and for any pair of them p_i and p_j, we have $msgs_i(0, r) = msgs_j(0, r)$. This constitutes the base case of an induction. Then, the same reasoning can be used to show that if the non-faulty processes participate in a round r, there is a round r' with $r < r' \leq r+k$ and such that, for any pair of non-faulty processes p_i and p_j, we have $msgs_i(r, r') = msgs_j(r, r')$.

Let us consider any two p_i and p_j that terminate round 0. Moreover, without loss of generality, let us assume that, among the sets of message sets output by $KSS(0)$, $sets_i(0)$ is the greatest and $sets_j(0)$ is the smallest. It follows from the Inter-process inclusion property that $sets_j(0) \subseteq sets_i(0)$, and from line 13 plus the Intra-process inclusion property that $msg\_set_i(0) \subseteq msg\_set_j(0)$. Hence, $|msg\_set_i(0)| \leq |msg\_set_j(0)|$. Moreover, due to the View size property of $KSS(0)$ we have $|msg\_set_i(0)| \leq |msg\_set_j(0)| = r \leq k$. Applying Lemma 11, we have $msg_i(0, 0 + r) = msg_j(0, 0 + r)$, which concludes the proof. ◀

▶ **Lemma 13.** *If a process p_i kscd-delivers first a message m belonging to a set ms_i and later a message m' belonging to a set $ms'_i \neq ms_i$, then no process kscd-delivers first m' in some kscd-delivered set ms'_j and later m in some kscd-delivered set $ms_j \neq ms'_j$.*

Proof. Let us first note that, at each process, the kscd-delivery of message sets establishes a partial order on messages. Given a process p_i, let $\rightarrow_i$ be the partial order defined as follows[2]: $m \rightarrow_i m'$ if p_i kscd-delivered first a message set ms_i including m, and later kscd-delivered a message set ms'_i including m'. Hence, if m and m' were kscd-delivered in the same message set by p_i, we have $m \not\rightarrow_i m'$ and $m' \not\rightarrow_i m$.

Let us also note that, along the execution of a process p_i, the partial order $\rightarrow_i$ can only be extended, i.e. if $m \rightarrow_i m'$ at time t, we cannot have $m \not\rightarrow_i m'$ at time $t' > t$. This, along with the fact that a faulty process executes its algorithm correctly until it crashes, allows us to consider, in the context of this proof, that p_i and p_j are non-faulty.

In order to prove the lemma, we then have to show that the partial orders $\rightarrow_i$ and $\rightarrow_j$ are compatible, i.e. for any two messages m and m', $(m \rightarrow_i m') \Rightarrow (m' \not\rightarrow_j m)$ and $(m \rightarrow_j m') \Rightarrow (m' \not\rightarrow_i m)$.

[2] This definition is similar to the definition of $\mapsto_i$ given in Section 3 devoted to kBO-broadcast.

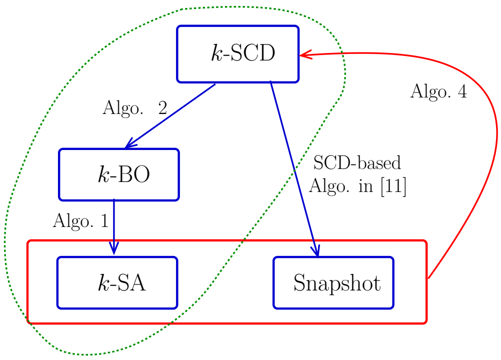


Figure 2 Detailing the global view.

According to Lemma 12, for each round r in which all processes participate, there is a round $r' > r$ in which all processes participate. Moreover, for any two non-faulty process p_i and p_j, we have $msgs_i(r, r') = msgs_j(r, r')$. For any such round r, we then have that if p_i delivered message m strictly before round r and delivered m' at round r or afterwards, we have both $(m \rightarrow_i m')$ and $(m' \rightarrow_j m)$. We will then consider the messages delivered between two such rounds r and r'.

Without loss of generality, suppose that the message set kscd-delivered by p_i at round r is smaller than, or equal to, the message set kscd-delivered by p_j at the same round, i.e. $|msg\_set_i(r)| \leq |msg\_set_j(r)|$. It follows from Lemma 11 that $msgs_i(r, |msg\_set_j(r)|) = msgs_j(r, |msg\_set_j(r)|)$. Moreover, as all the messages in $msg\_set_j(r)$ were kscd-delivered by p_j in a single set, they are all incomparable when considering $\rightarrow_j$. The partial orders $\rightarrow_i$ and $\rightarrow_j$, when restricted to the messages in $msg\_set_j(r)$, are thus compatible.

According to Lemma 11, p_i and p_j will both participate in round $r+\alpha = r+|msg\_set_j(r)|$. If $r+\alpha = r'$, the lemma follows. Otherwise, let $\beta = \max(|msg\_set_i(r+\alpha)|, |msg\_set_j(r+\alpha)|)$. The previous reasoning, again due to Lemma 11, can then be applied again to the messages in $msgs_i(r+\alpha, r+\alpha+\beta) = msgs_j(r+\alpha, r+\alpha+\beta)$, and p_i and p_j will both participate in round $r+\alpha+\beta$. This can be repeated until round r', showing that the partial orders $\rightarrow_i$ and $\rightarrow_j$ are compatible, which concludes the proof of the lemma. ◀

▶ **Lemma 14.** *No message m is kscd-delivered twice by a process p_i.*

▶ **Lemma 15.** *Let m be a message that has been deposited into MEM. Eventually, m is kscd-delivered (at least) by the non-faulty processes.*

▶ **Lemma 16.** *If a process kscd-delivers a message m, every non-faulty process kscd-delivers a message set containing m.*

▶ **Lemma 17.** *If a non-faulty process p_i kscd-broadcasts a message m, it terminates its kscd-broadcast invocation and kscd-delivers a message set containing m.*

▶ **Theorem 18.** *Algorithm* 4 *implements* KSCD-broadcast *from k-set agreement and snapshot objects.*

7 Conclusion

This paper has introduced a new communication abstraction, denoted k-BO-broadcast, which captures k-set agreement in asynchronous crash-prone wait-free systems. In the case $k = 1$ (consensus is 1-set agreement), 1-BO-broadcast boils down to Total Order broadcast.

"Capture" means here that (i) k-set agreement can be solved in any system model providing the k-BO-broadcast abstraction, and (ii) k-BO-broadcast can be implemented from k-set agreement in any system model providing snapshot objects. It follows that, when considering asynchronous crash-prone wait-free systems where basic communication is through a set of atomic read/write, or the asynchronous message-passing system enriched with the failure detector Σ [5, 8], k-BO-broadcast and k-set agreement are the two faces of the same coin: one is its communication-oriented face while the other one is its agreement-oriented face.

From a technical point of view, a complete picture of the content of the paper appears in Figure 2. It is important to notice that the two constructions inside the dotted curve are free from concurrent objects: each rests only on an underlying (appropriate) communication abstraction.

References

1 Yehuda Afek, Hagit Attiya, Danny Dolev, Eli Gafni, Michael Merritt, and Nir Shavit. Atomic snapshots of shared memory. *J. ACM*, 40(4):873–890, 1993.

2 James H. Anderson. Multi-writer composite registers. *Distributed Computing*, 7(4):175–195, 1994.

3 Hagit Attiya and Jennifer L. Welch. *Distributed computing: fundamentals, simulations and advanced topics*. Wiley-Interscience, 2004.

4 Kenneth P. Birman and Thomas A. Joseph. Reliable communication in the presence of failures. *ACM Trans. Comput. Syst.*, 5(1):47–76, 1987.

5 François Bonnet and Michel Raynal. A simple proof of the necessity of the failure detector sigma to implement an atomic register in asynchronous message-passing systems. *Inf. Process. Lett.*, 110(4):153–157, 2010.

6 Tushar Deepak Chandra and Sam Toueg. Unreliable failure detectors for reliable distributed systems. *J. ACM*, 43(2):225–267, 1996.

7 Soma Chaudhuri. More choices allow more faults: Set consensus problems in totally asynchronous systems. *Inf. Comput.*, 105(1):132–158, 1993.

8 Carole Delporte-Gallet, Hugues Fauconnier, and Rachid Guerraoui. Tight failure detection bounds on atomic object implementations. *J. ACM*, 57(4):22:1–22:32, 2010.

9 Faith Ellen. How hard is it to take a snapshot? In *SOFSEM 2005: Theory and Practice of Computer Science, 31st Conference on Current Trends in Theory and Practice of Computer Science, Liptovský Ján, Slovakia, January 22-28, 2005, Proceedings*, pages 28–37, 2005.

10 Michael J. Fischer and Michael Merritt. Appraising two decades of distributed computing theory research. *Distributed Computing*, 16(2-3):239–247, 2003.

11 Damien Imbs, Achour Mostéfaoui, Matthieu Perrin, and Michel Raynal. Set-constrained delivery broadcast: Definition, abstraction power, and computability limits. *CoRR*, abs/1706.05267, 2017. URL: `http://arxiv.org/abs/1706.05267`.

12 Damien Imbs, Achour Mostéfaoui, Matthieu Perrin, and Michel Raynal. Which broadcast abstraction captures k-set agreement? *CoRR*, abs/1705.04835, 2017. URL: `http://arxiv.org/abs/1705.04835`.

13 Damien Imbs and Michel Raynal. Help when needed, but no more: Efficient read/write partial snapshot. *J. Parallel Distrib. Comput.*, 72(1):1–12, 2012.

14 Michiko Inoue, Toshimitsu Masuzawa, Wei Chen, and Nobuki Tokura. Linear-time snapshot using multi-writer multi-reader registers. In *Distributed Algorithms: 8th International Workshop, WDAG '1994 Terschelling, The Netherlands, September 29 – October 1, 1994 Proceedings*, pages 130–140. Springer Berlin Heidelberg, 1994.

15 Nancy A. Lynch. *Distributed Algorithms*. Morgan Kaufmann, 1996.

16 Michel Raynal. *Communication and Agreement Abstractions for Fault-Tolerant Asynchronous Distributed Systems.* Synthesis Lectures on Distributed Computing Theory. Morgan & Claypool Publishers, 2010.

17 Michel Raynal. *Distributed Algorithms for Message-Passing Systems.* Springer, 2013.

18 Michel Raynal. Set agreement. In *Encyclopedia of Algorithms*, pages 1956–1959. 2016.

19 Michel Raynal, André Schiper, and Sam Toueg. The causal ordering abstraction and a simple way to implement it. *Inf. Process. Lett.*, 39(6):343–350, 1991.