Which Broadcast Abstraction Captures 
\(k\)-Set Agreement?∗

Damien Imbs¹, Achour Mostéfaoui², Matthieu Perrin³, and Michel Raynal⁴

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 \texttt{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 \texttt{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.
Which Broadcast Abstraction Captures $k$-Set Agreement?

Table 1

<table>
<thead>
<tr>
<th>Concurrent object</th>
<th>Communication abstraction</th>
</tr>
</thead>
<tbody>
<tr>
<td>Consensus</td>
<td>Total order broadcast [6]</td>
</tr>
<tr>
<td>Snapshot object [1, 2] (and R/W register)</td>
<td>SCD-broadcast [11]</td>
</tr>
<tr>
<td>$k$-set agreement object ($1 \leq k &lt; n$)</td>
<td>$k$-BO-broadcast (this paper)</td>
</tr>
</tbody>
</table>

Figure 1

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 $view_i = \{(reg_{i}[x], i) \mid reg_{i}[x] \neq \bot\}$. For any pair of processes $p_i$ and $p_j$ which respectively obtain $view_i$ and $view_j$, we have $(view_i \subseteq view_j) \lor (view_j \subseteq view_i)$. 

2 Process Model, Snapshot, and $k$-Set Agreement
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 $\text{propose}(\cdot)$. When a process $p_i$ invokes $\text{propose}(v_i)$, 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 $\text{propose}(\cdot)$ 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 $\text{RKSA}.\text{propose}(sn_i^1, v_i^1)$, then $\text{RKSA}.\text{propose}(sn_i^2, v_i^2)$, ..., $\text{RKSA}.\text{propose}(sn_i^x, v_i^x)$, 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 $\text{RKSA}.\text{propose}(sn, v_i)$ 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 $\text{kbo\_broadcast}(\cdot)$ and $\text{kbo\_deliver}(\cdot)$. 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 $\text{kbo\_broadcast}(m)$, we say that it “kbo-broadcasts the message $m$”. Similarly, when it invokes $\text{kbo\_deliver}(\cdot)$ and obtains a message $m$, we say that it “kbo-delivers $m$”; in the operating system parlance, $\text{kbo\_deliver}(\cdot)$ 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 $\rightarrow$.** 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 $\rightarrow_i$ be the local message delivery order at a process $p_i$ defined as follows: $m \rightarrow_i m'$ if $p_i$ kbo-delivers the message $m$ before it kbo-delivers the message $m'$. Let $\rightarrow \overset{def}{=} \bigcap_i \rightarrow_i$. 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 $(\rightarrow)$ for the relation and the associated partially ordered graph. Let $\text{width}(\rightarrow)$ denote the width of the partially ordered graph $\rightarrow$. 
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 \texttt{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. width(\(\rightarrow\)) \(\leq k\).
- 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 \(\leq 2\), 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 \texttt{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, \texttt{channel}[1] and \texttt{channel}[2]. As shown by the following figure, they contained the following
operation propose\((nb,v)\) is
\begin{enumerate}
\item kbo\_broadcast\((nb,v)\); wait(\exists \langle nb,x \rangle \in \text{decisions}_i); return\(x\).
\end{enumerate}
when a message \(\langle sn,x \rangle\) is kbo-delivered do
\begin{enumerate}
\item if (\langle sn,−\rangle never added to \text{decisions}_i) then \text{decisions}_i.insert(\langle sn,x \rangle) end if.
\end{enumerate}

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

sequences of messages: \text{channel}[1] = m_1, m_5 and \text{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 \text{channel}[3], let us observe that the same message kbo-delivers at \(p_1, p_2,\) and \(p_3\), could have been obtained by the following channel selection by the daemon: \text{channel}[1] as before, \text{channel}[2] = m_3, m_4, and \text{channel}[3] = m_2. 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.

\textbf{Theorem 1.} A non-deterministic k-TO-channel and the k-BO-broadcast communication abstraction have the same computational power.

\textbf{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 \text{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 \text{decisions}_i (line 1). Such a pair is added in \text{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.

\textbf{Lemma 2.} If the invocation of \text{propose}(nb,v) returns \(x\) to a process, some process invoked \text{propose}(nb,x).
Lemma 3. If a non-faulty process invokes \texttt{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 \texttt{propose}(nb, −) contains at most \(k\) different values.

Proof. Let \(\Pi_{nb}\) be the set of processes returning a value from their invocations \texttt{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 \texttt{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, \(\langle nb, x_j \rangle \not\rightarrow_i\langle nb, x_i \rangle\) and \(\langle nb, x_i \rangle \not\rightarrow_j\langle nb, x_j \rangle\), which means \(\langle nb, x_i \rangle\) and \(\langle nb, x_j \rangle\) are not ordered by \(\rightarrow\). Therefore, \(\{\langle nb, x_i \rangle : p_i \in \Pi_{nb}\}\) is an antichain of \(\rightarrow\). It then follows from the KBO-Bounded property that \(|\{x_i : p_i \in \Pi_{nb}\}| = |\{\langle nb, x_i \rangle : p_i \in \Pi_{nb}\}| \leq k\).

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 \texttt{scd\_broadcast}() and \texttt{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 \texttt{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”.

DISC 2017
Which Broadcast Abstraction Captures $k$-Set Agreement?

```
operation kbo_broadcast(v) is kscd_broadcast(m).
when a message set ms is kscd-delivered do for each m ∈ ms do 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 $m \in fms_i \land m \notin fms_j$, and $m' \notin fms_i \land m' \in fms_j$.)

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 $fm \rightarrow lm$. It follows that $ms$ cannot be an antichain of $\rightarrow$. Consequently, the antichains of $\rightarrow$ cannot contain more than $k$ messages, hence width($\rightarrow$) $\leq k$.

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 $\text{k2s\_propose}(v)$ 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
Algorithm 3 An implementation of a K2S object.

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 \( \text{k2s}_\text{propose}(v) \), it first proposes \( v \) to the k-set agreement object, from which it obtains a value \( \text{val}_i \) (line 1).
- Phase 2 (lines 2-3). Then \( p_i \) writes \( \text{val}_i \) in the first snapshot object \( SNAP1 \), reads its content, saves it in \( \text{snap1} \), and computes the set of values \( \{ \text{view}_j \} \) 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 \( \text{view}_i \) in the second snapshot object \( SNAP2 \), reads its value, and computes the set of views \( \{ \text{sets}_i \} \) obtained – as far as it knows – by the other processes. Process \( p_i \) finally returns this set of views \( \text{sets}_i \).

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 \( \text{KSS}_i \cdot \text{k2s}_\text{propose}(r, v) \) 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 \( \text{KSET}_i \cdot \text{propose}(r, v) \).

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 \( \text{mem1}_i \) and \( \text{mem2}_i \), two auxiliary sets \( \text{to}_\text{deliver1}_i \) and \( \text{to}_\text{deliver2}_i \), and a set \( \text{delivered}_i \), which contains all the messages it has locally kscd-delivered; \( \text{mem1}_i[i] \) is initialized to an empty set.
Which Broadcast Abstraction Captures \(k\)-Set Agreement?

\[
\text{operation } \text{kscd\_broadcast}(m) \text{ is }
\begin{align*}
&1. \quad \text{MEM.write}(\text{mem}1, i \cup \{m\}); \text{mem}1 \leftarrow \text{MEM.snapshot}(); \\
&2. \quad \text{to\_deliver}1 \leftarrow (U_{1 \leq j \leq n} \text{mem}1[j]) \setminus \text{delivered}_i; \text{wait(to\_deliver}1, \subseteq \text{delivered}_i).
\end{align*}
\]

background task \(T\) is

\[
\begin{align*}
&3. \quad \text{repeat forever} \\
&4. \quad \text{prop}_i \leftarrow \bot; \\
&5. \quad \text{if } (\text{seq}_i = \epsilon) \text{ then } \text{mem}2_i \leftarrow \text{MEM.snapshot}(); \\
&6. \quad \text{to\_deliver}2_i \leftarrow (U_{1 \leq j \leq n} \text{mem}2[j]) \setminus \text{delivered}_i; \\
&7. \quad \text{if } (\text{to\_deliver}2_i \neq \emptyset) \text{ then } \text{prop}_i \leftarrow \text{a message in to\_deliver}2_i \text{ end if} \\
&8. \quad \text{else } \text{prop}_i \leftarrow \text{a message of the first message set of seq}_i \\
&9. \quad \text{end if}; \\
&10. \quad \text{if } (\text{prop}_i \neq \bot) \\
&11. \quad \text{then } r_i \leftarrow |\text{delivered}_i|; \text{seq}_i \leftarrow \text{KSS.k2s\_propose}(r_i, \text{prop}_i); \text{new\_seq}_i \leftarrow \epsilon; \\
&12. \quad \text{while } (\text{seq}_i \neq \emptyset) \text{ do} \\
&13. \quad \text{min\_set}_i \leftarrow \text{non-empty set of minimal size in seq}_i; \\
&14. \quad \text{new\_seq}_i \leftarrow \text{new\_seq}_i \oplus \text{min\_set}_i; \\
&15. \quad \text{for each set } s \in \text{seq}_i \text{ do seq}_i \leftarrow (\text{seq}_i \setminus \{s\}) \cup \{s \setminus \text{min\_set}_i\} \text{ end for}; \\
&16. \quad \text{end while}; \\
&17. \quad \text{let aux}_i = \text{all the messages in the sets of new\_seq}_i; \\
&18. \quad \text{for each set } s \in \text{seq}_i \text{ do } \text{do seq}_i \leftarrow \text{aux}_i \text{ end for}; \\
&19. \quad \text{seq}_i \leftarrow \text{new\_seq}_i \oplus \text{seq}_i; \text{let first}_i = \text{head(seq}_i); \text{let rest}_i = \text{tail(seq}_i); \\
&20. \quad \text{kscd\_deliver(first}_i); \text{delivered}_i \leftarrow \text{delivered}_i \cup \text{first}_i; \text{seq}_i \leftarrow \text{rest}_i \\
&21. \quad \text{end if} \\
&22. \quad \text{end repeat}.
\]

**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; \(\text{seq}_i\) 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 message sets, both initialized to \(\epsilon\) (empty sequence), denoted \(\text{seq}_i\) and \(\text{new\_seq}_i\); \(\text{head(seq}_i)\) returns the first element of the sequence \(\text{seq}_i\), and \(\text{tail(seq}_i)\) returns \(\text{seq}_i\) without its first element; \(\oplus\) denotes sequence concatenation.

- The aim of the local sequence \(\text{new\_seq}_i\) is to contain a sequence of message sets obtained from \(\text{seq}_i\) (last invocation of a K2S object) such that no message belongs to several sets.

- As far as \(\text{seq}_i\) is concerned, we have the following (at line 19 of Algorithm 4). Let \(\text{seq}_i = m_{s_1}, m_{s_2}, \ldots, m_{s_t}\), where \(1 \leq t \leq k\) and each \(m_{s_t}\) is a message set. This sequence can be decomposed into two (possibly empty) sub-sequences \(m_{s_1}, m_{s_2}, \ldots, m_{s_y}\) and \(m_{s_{y+1}}, \ldots, m_{s_t}\) such that:

  - \(m_{s_1}, m_{s_2}, \ldots, m_{s_y}\) can be in turn decomposed as follows:

    \[
    (m_{s_1} \cup m_{s_2} \cup \cdots \cup m_{s_a}), (m_{s_{a+1}} \cup m_{s_{a+2}} \cup \cdots \cup m_{s_b}), \ldots, (m_{s_{c-1}} \cup \cdots \cup m_{s_t})
    \]

    where each union set (e.g., \(m_{s_{a+1}} \cup m_{s_{a+2}} \cup \cdots \cup m_{s_b}\)) is a message set that has been kscd-delivered by some process (some union sets can contain a single message set)\(^1\).

  - For each \(x\) : \(y + 1 \leq x \leq t\) : \(m_{s_x}\) 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 \(\text{MEM}\), which contains all the messages it has already kscd-broadcast (line 1). Then \(p_i\) reads atomically the whole content of \(\text{MEM}\), which is saved in \(\text{mem}1\) (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 \(m = m_{s_1} \cup m_{s_2} \cup \cdots \cup m_{s_{y-1}}\), another process kscd-delivered the messages in \(m\) in several messages sets, e.g., first the message set \(m_{s_1} \cup m_{s_2} \cup m_{s_3}\) and then the message set \(m_{s_4} \cup \cdots \cup m_{s_t}\).
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 \(|\text{delivered}_i|\) (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 \( \text{first}_i \) in the algorithm, line 20) it kscd-delivers at the end of round \( r \) (\( \text{delivered}_i \) becomes then \( \text{delivered}_i \cup \text{first}_i \)). The message set \( \text{first}_i \) 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 (\( \text{prop}_i \)) that \( p_i \) will propose to the next K2S object. This local computation is as follows (lines 5-9), where \( \text{seq}_i \) (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:** \( \text{seq}_i = \epsilon \). In this case (similarly to line 2) \( p_i \) computes the set of messages (\( \text{to\_deliver}_2 \)) it sees as kscd-broadcast but not yet locally kscd-delivered (lines 5-6). If \( \text{to\_deliver}_2 \neq \emptyset \), a message of this set becomes its proposal \( \text{prop}_i \) for the K2S object associated with the next round (line 7). Otherwise, we have \( \text{prop}_i = \epsilon \), which, due to the predicate of line 10, entails a new execution of the loop (skipping lines 11-20).

- **Case 2:** \( \text{seq}_i \neq \emptyset \). In this case, \( \text{prop}_i \) is assigned a message of the first set of \( \text{seq}_i \) (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 \( r_i = |\text{delivered}_i| \). The increase step of \(|\text{delivered}_i|\) 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 \(|\text{delivered}_i|\)). 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 \( \text{KSS.k2s\_propose}(r_i, \text{prop}_i) \) where \( r_i = |\text{delivered}_i| \), which returns a set of message sets \( \text{sets}_i \) (line 11). Then (“while” loop at lines 12-16), \( p_i \) builds from the message sets belonging to \( \text{sets}_i \) a sequence of message sets \( \text{new\_seq} \), that will be used to extract the next message set kscd-delivered by \( p_i \) (lines 17-20). The construction of \( \text{new\_seq} \) is as follows. Iteratively, \( p_i \) takes the smallest set of \( \text{sets}_i \) (\( \text{min\_set}_i \)) (line 13), adds it at the end of \( \text{new\_seq} \) (line 14), and purges all the sets of \( \text{sets}_i \) from the messages in \( \text{min\_set}_i \) (line 15), so that no message will locally appear in two different messages sets of \( \text{new\_seq} \).

When \( \text{new\_seq} \) is built, \( p_i \) first purges all the sets of the sequence \( \text{seq}_i \) from the messages in \( \text{new\_seq} \) (lines 17-18), and adds then \( \text{new\_seq} \) at the front of \( \text{seq}_i \) (line 19). Finally, \( p_i \) kscd-delivers the first message set of \( \text{seq}_i \), and updates \( \text{delivered}_i \) and \( \text{seq}_i \) (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.
- \(\text{msg\_set}_i(r)\) = message set kscd-delivered by process \(p_i\) at round \(r\) if \(p_i\) participated in it, and \(\emptyset\) otherwise.
- \(\text{seq}_i(r)\) = value of \(\text{seq}_i\) at the end of the last round \(r' \leq r\) in which \(p_i\) participated.
- \(\text{msgs}_i(r,r')\) = set of messages contained in message sets kscd-delivered by \(p_i\) between rounds \(r\) (included) and \(r' > r\) (not included), i.e. \(\text{msgs}_i(r,r') = \bigcup_{r'' \leq r'} \text{msg\_set}_i(r'')\).
- \(\text{KSS}(r)\) = K2S instance accessed by KSS.k2s_propose\((r, -)\) (line 11).
- \(\text{sets}_i(r)\) = set of message sets obtained by \(p_i\) from \(\text{KSS}(r)\).

Lemma 10. Let \(p_i\) and \(p_j\) be two processes that terminate round \(r\), with \(|\text{msg\_set}_i(r)| \leq |\text{msg\_set}_j(r)|\). Then (i) \(\text{msg\_set}_i(r) \subseteq \text{msg\_set}_j(r)\), and (ii) there is a prefix \(\text{pref}_i\) of \(\text{seq}_i\) such that \(\text{msg\_set}_j(r) = \text{msg\_set}_i(r) \cup (\bigcup_{\text{msg\_set} \in \text{pref}_i} \text{msg\_set})\).

Proof. Let \(p_i\) and \(p_j\) be two processes which kscd-deliver the message sets \(\text{msg\_set}_i(r)\) and \(\text{msg\_set}_j(r)\), respectively, these sets being such that \(|\text{msg\_set}_i(r)| \leq |\text{msg\_set}_j(r)|\). Let us observe that, as both \(p_i\) and \(p_j\) invoked KSS.k2s_propose\((r, -)\) (lines 11 and 20), we have \(\text{sets}_i(r) \subseteq \text{sets}_j(r)\) or \(\text{sets}_j(r) \subseteq \text{sets}_i(r)\) (Inter-process Inclusion).

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

As far as (ii) is concerned, we have the following. If \(\text{msg\_set}_i(r) = \text{msg\_set}_j(r)\), we have \(\text{pref}_i = \epsilon\) and the lemma follows. So, let us assume \(\text{msg\_set}_i(r) \subset \text{msg\_set}_j(r)\). As \(\text{msg\_set}_i(r)\) is the smallest message set of \(\text{sets}_i(r)\) (lines 13-14 and 19-20), and \(\text{msg\_set}_j(r)\) is the smallest message set of \(\text{sets}_j(r)\), it follows that \(\text{sets}_j(r) \subseteq \text{sets}_i(r)\). The property \(\text{msg\_set}_i(r) = \text{msg\_set}_j(r) \cup (\bigcup_{\text{msg\_set} \in \text{pref}_i} \text{msg\_set})\) follows then from the following observation. Let \(\text{sets}_i(r) = \{s_1, s_2, ..., s_k\}\), where \(\ell \leq k\) and \(s_1 \subsetneq s_2 \subsetneq \cdots \subsetneq s_k\). As \(\text{sets}_j(r) \subseteq \text{sets}_i(r)\), one \(s_j\) is \(\text{msg\_set}_j(r)\). It follows that the union of the sets \(\text{min\_set}_i\) computed by \(p_i\) in the while loop of round \(r\) (lines 13-15) eventually includes all the messages of \(\text{msg\_set}_j(r)\), from which we conclude that there is a prefix \(\text{pref}_i\) of \(\text{seq}_i\) (lines 12-19), namely a prefix of the sequence \(\text{new\_seq}\), which is defined from the sequence of the sets \(\text{min\_set}_i\), such that \(\text{msg\_set}_j(r) = \text{msg\_set}_i(r) \cup (\bigcup_{\text{msg\_set} \in \text{pref}_i} \text{msg\_set})\), 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 \(r' \geq r + |\text{msg\_set}_j(r)|\), and are such that \(|\text{msg\_set}_i(r)| \leq |\text{msg\_set}_j(r)|\). Then (i) \(\text{msg\_set}_i(r, r + |\text{msg\_set}_j(r)|) = \text{msgs}_i(r, r + |\text{msg\_set}_j(r)|)\), and (ii) \(p_i\) and \(p_j\) will both participate in round \(r + |\text{msg\_set}_j(r)|\).

Proof. If \(|\text{msg\_set}_i(r)| = |\text{msg\_set}_j(r)| = \alpha\), both \(p_i\) and \(p_j\) are such that \(\text{delivered}_i = |\text{delivered}_j| = r + \alpha\) 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) \(\text{msgs}_i(r, r + |\text{msg\_set}_j(r)|) = \text{msg\_set}_i(r) = \text{msg\_set}_j(r) = \text{msgs}_i(r, r + |\text{msg\_set}_j(r)|)\), (ii) both \(p_i\) and \(p_j\) will participate in round \(r + |\text{msg\_set}_j(r)|\), and the lemma follows.
Hence, let us consider that $|\text{msg}_{\text{set}}(r)| = \alpha < |\text{msg}_{\text{set}}(r)| = \alpha + \beta$. 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 $\text{msg}_{\text{set}}(r)$ (resp. $\text{msg}_{\text{set}}(r)$) is the smallest (resp. second smallest) message set in the sets of message sets $\text{sets}$ output by KSS$(r)$.

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

\begin{lemma}
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 $\text{msg}_{\text{set}}(r,r') = \text{msg}_{\text{set}}(r,r')$.
\end{lemma}

\begin{proof}
As initially $\forall i : \text{delivered}_i = 0$, 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 $\text{msg}_{\text{set}}(0,r) = \text{msg}_{\text{set}}(0,r)$. 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 $\text{msg}_{\text{set}}(r,r') = \text{msg}_{\text{set}}(r,r')$.

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$(0)$, $\text{sets}_0$ is the greatest and is the smallest. It follows from the Inter-process inclusion property that $\text{sets}_2(0) \subseteq \text{sets}_0$, and from line 13 plus the Intra-process inclusion property that $\text{msg}_{\text{set}}(0) \subseteq \text{msg}_{\text{set}}(0)$. Hence, $|\text{msg}_{\text{set}}(0)| \leq |\text{msg}_{\text{set}}(0)|$. Moreover, due to the View size property of KSS$(0)$ we have $|\text{msg}_{\text{set}}(0)| \leq |\text{msg}_{\text{set}}(0)| = r \leq k$. Applying Lemma 11, we have $\text{msg}_{\text{set}}(0,0 + r) = \text{msg}_{\text{set}}(0,0 + r)$, which concludes the proof.
\end{proof}

\begin{lemma}
If a process $p_i$ kscd-delivers first a message $m$ belonging to a set $\text{ms}_i$ and later a message $m'$ belonging to a set $\text{ms}_f \neq \text{ms}_i$, then no process kscd-delivers first $m'$ in some kscd-delivered set $\text{ms}_j'$ and later $m$ in some kscd-delivered set $\text{ms}_j \neq \text{ms}_f$.
\end{lemma}

\begin{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:\footnote{This definition is similar to the definition of $\rightarrow_i$ given in Section 3 devoted to kBO-broadcast.}: $m \rightarrow_i m'$ if $p_i$ kscd-delivered first a message set $\text{ms}_i$ including $m$, and later kscd-delivered a message set $\text{ms}_f'$ including $m'$. Hence, if $m$ and $m'$ were kscd-delivered in the same message set by $p_i$, we have $m \neq m'$ and $m' \neq m$.

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 $m \neq m'$ 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 $m'$, $(m \rightarrow_i m') \Rightarrow (m' \neq m)$ and $(m \rightarrow_j m') \Rightarrow (m' \neq m)$. 


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 \( msg_i(r, r') = msg_j(r, r') \). 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 \( msg_i(r, |msg_set_j(r)|) = msg_j(r, |msg_set_j(r)|) \). 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 \( msg_i(r+\alpha, r+\alpha+\beta) = msg_j(r+\alpha, r+\alpha+\beta) \), 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.

\[ \text{Lemma 14. No message } m \text{ is } kscd \text{-delivered twice by a process } p_i. \]

\[ \text{Lemma 15. Let } m \text{ be a message that has been deposited into MEM. Eventually, } m \text{ is } kscd \text{-delivered (at least) by the non-faulty processes.} \]

\[ \text{Lemma 16. If a process } kscd \text{-delivers a message } m, \text{ every non-faulty process } kscd \text{-delivers a message set containing } m. \]

\[ \text{Lemma 17. If a non-faulty process } p_i \text{ } kscd \text{-broadcasts a message } m, \text{ it terminates its } kscd \text{-broadcast invocation and } kscd \text{-delivers a message set containing } m. \]

\[ \text{Theorem 18. Algorithm 4 implements KSCD-broadcast from } k \text{-set agreement and snapshot objects.} \]

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 $\Sigma$ [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

27:16 Which Broadcast Abstraction Captures \( k \)-Set Agreement?


