No Broadcast Abstraction Characterizes k-Set-Agreement in Message-Passing Systems (Extended Version)

Sylvain Gay, Achour Mostefaoui, Matthieu Perrin

▶ To cite this version:

Sylvain Gay, Achour Mostefaoui, Matthieu Perrin. No Broadcast Abstraction Characterizes k-Set-Agreement in Message-Passing Systems (Extended Version). 2024. hal-04571653

HAL Id: hal-04571653
https://hal.science/hal-04571653

Preprint submitted on 8 May 2024

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire HAL, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

No Broadcast Abstraction Characterizes k-Set-Agreement in Message-Passing Systems (Extended Version)

Sylvain Gay*
École Normale Supérieure
sylvain.gay@ens.psl.eu

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

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

Abstract

This paper explores the relationship between broadcast abstractions and the k-set agreement (k-SA) problem in crash-prone asynchronous distributed systems. It specifically investigates whether any broadcast abstraction is computationally equivalent to k-SA in message-passing systems.

A key contribution of the paper is the introduction of two new symmetry properties: *compositionality* and *content-neutrality*, inspired by the principle of network neutrality. Such clarity in definition is essential for this paper's scope, as it aims not to characterize the computing power of a specific broadcast abstraction, but rather to demonstrate the nonexistence of a broadcast abstraction with certain characteristics. Hence, delineating the realm of "meaningful" broadcast abstractions becomes essential. The paper's main contribution is the proof that no broadcast abstraction, which is both content-neutral and compositional, is computationally equivalent to k-set agreement when $1 < k < n$, in the crash-prone asynchronous message-passing model. To the best of our knowledge, this result represents the first instance of showing that a coordination problem cannot be expressed by an equivalent broadcast abstraction. It does not establish the absence of an implementation, but rather the absence of a specification that possesses certain properties.

Key-words: Agreement problem, Asynchronous system, Broadcast abstraction, Communication abstraction, Compositionality, Message-passing system, Network neutrality, Process crash, k-Set agreement, Wait-free model, Total order broadcast.

*This author was at LS2N, Nantes Université when this research was conducted.

1 Introduction

1.1 From Send/Receive to Communication Abstractions

This paper considers distributed systems consisting of a set of asynchronous processes prone to crash failures. These processes communicate by sending and receiving messages across an asynchronous network and must cooperate to achieve a common goal. What makes distributed computing challenging is that the dynamics of the underlying network on which the distributed application operates are beyond the programmer's direct control. This necessitates treating the environment as a "hidden input" [23] and to "manage uncertainty" at runtime. To facilitate the design of advanced algorithms in this unpredictable setting, it is usual to define appropriate communication abstractions, that allow modularity and help mitigate uncertainty by restricting communication patterns that may occur at a higher abstraction level.

In crash-prone asynchronous distributed systems, a significant source of uncertainty stems from the divergent perceptions of the event set (i.e., message emissions and receptions) among different processes. *Broadcast abstractions*, which enable processes to transmit a message to all participants within the same operation, alleviate this issue by ensuring consistent and reliable communication across different nodes, thereby simplifying the complexity of managing individual send/receive operations and enhance fault tolerance by reducing the impact of node failures. Hence, message broadcasts (at least by correct processes) constitute a set of global events for which all correct processes eventually agree they took place, thereby underlining their significance in the architecture of reliable distributed computing systems.

Another source of uncertainty arises from the disparate order in which different participants may receive messages, leading to varied perceptions of the global state of the system. Several communication abstractions have been defined by enforcing properties on the message delivery order. FIFO and Causal Ordering are examples of such properties at the heart of FIFO-broadcast and Causal-broadcast [3, 24]. These abstractions facilitate the construction of distributed objects, like causal memory in asynchronous message-passing systems [2].

A remark on vocabulary Throughout this paper, to avoid confusion, we distinguish between the terms "send" and "receive", which denote low-level point-to-point communication primitives applied to individual messages, and "broadcast" and "deliver", which describe the higher-level operations of broadcast abstractions (one-to-all). Consequently, in the context of this paper, the terms "receive" and "deliver" are not used interchangeably or as quasi-synonyms.

1.2 Capturing Coordination Problems with Broadcasts

This paper follows the quest of identifying broadcast abstractions that characterize the major fundamental problems in distributed computing. Specifically, we aim to determine broadcast abstractions that are computationally equivalent to particular synchronization problems in a crash-prone asynchronous message-passing system. This equivalence means that the broadcast abstraction can resolve the synchronization problem regardless the

number of crash failures, and vice versa.

A well-known such characterization is the equivalence between *Total Order Broadcast* and the *consensus* problem. Consensus is a fundamental problem of distributed computing, that allows each process to propose a value, and ensures all correct processes decide on a common value. The defining properties of this problem are as follows: if a process invokes *propose*(v) and does not crash, it will decide on a value (termination); no two processes will decide on different values (agreement); and the decided value must have been proposed by a process (validity). One of the primary practical applications of consensus is to maintain consistency across replicated machines in a message-passing system. However, State Machine Replication (SMR) [26] typically builds on an intermediate communication abstraction, the well-known and powerful Total Order Broadcast abstraction [21]. This abstraction ensures that the order of message delivery is consistent across all processes.

The consensus problem is famously unsolvable in an asynchronous distributed system, even under the assumption that at most one process may crash [11]. The same holds for Total Order Broadcast. Indeed, both abstractions are computationally equivalent [7]. In a sense, Total Order Broadcast precisely "characterizes" the essence of the consensus problem. In a similar vein, *Mutual Broadcast* was recently proposed as a broadcast abstraction equivalent to read/write atomic registers [9]. Moreover, *Pair Broadcast* characterizes the computational power of both test-and-set and consensus between two processes [10]. Such capturing broadcast abstractions are instrumental for understanding the fundamentals of distributed computing problems by reducing their complexity into a logical property about the order in which different processes perceive events occurring in the system.

1.3 On the k-set Agreement Side

Specifically, this paper delves into characterizing the k-set agreement problem (k-SA), a generalization of consensus introduced by S. Chaudhuri in [8]. In k-SA, the agreement property is weakened as follows: processes are allowed to collectively decide up to k different values. Here, k represents the maximum disagreement in the number of different values that can be decided. The smallest value $k = 1$ corresponds to consensus. As k increases, the problem becomes less constrained and may become easier to solve. However, it still embodies numerous complexities and challenges of distributed systems. It remains insoluble in a crash-prone asynchronous system when $k < t$, where t is the maximum number of processes in the system that may crash [5, 14, 25].

The exploration of a broadcast abstraction that characterizes k-SA was initiated in a work dedicated to the shared-memory model, which proposed *k-Bounded Order Broadcast* (k-BO Broadcast in short) [15]. The k-BO Broadcast abstraction limits the disagreement on the message reception order among processes. Specifically, its ordering property asserts that every set of $k+1$ messages contains two messages delivered in the same order by all processes. In the special case where $k = 1$, it boils down to Total Order Broadcast.

In crash-prone asynchronous systems where processes additionally have access to a shared memory composed of atomic read/write registers, k-BO Broadcast is computationally equivalent to k-set agreement. However, this equivalence in shared memory does not inherently translate to message-passing systems. Indeed, although k-BO broadcast

can be used to solve k-set agreement on its own, it remains unproven whether it can be implemented using solely k-set agreement objects and send/receive operations. While consensus is strong enough to emulate atomic registers, k-set agreement, for $k > 1$, is unable to emulate shared memory. Indeed, it has been proved that on one hand, k-SA and a problem called the k-simultaneous-agreement are equivalent in shared memory systems [1], and on the other hand, the k-simultaneous-agreement problem is harder than k-SA in message-passing systems where a shared memory emulation is not possible [6]. A corollary of this paper is that the implementation of k-BO broadcast on top of k-SA is not feasible in message-passing systems.

Problem Statement This paper investigates the following question: *Does there exist a broadcast abstraction computationally equivalent to k-SA in crash-prone asynchronous message-passing systems?*

1.4 Contributions

Symmetric broadcast abstractions. A simplistic approach to the discussed question might propose the following ordering property: "At most k distinct messages can be delivered as the first messages by the processes." Indeed, on the one hand, a k-SA object can select the set of messages eligible for initial delivery; and on the other hand, k-SA can be trivially solved by broadcasting all proposed values and deciding on the first delivered ones, hence establishing equivalence. However, such a solution is "unsatisfactory", as an instance of this broadcast abstraction would only be effective for solving k-SA once, before the ordering property becomes meaningless. Hence, an iterative resolution of k-SA would necessitate a difference instance of the broadcast for each k-SA object to implement. This requirement contrasts with the traditional understanding of how processes interact with the communcation layer in a message-passing system, where a broadcast abstraction serves as a system-wide service, shared among multiple algorithms for solving higher-level tasks. Each algorithm employs only a subset of the system's messages.

Hence, before delving into our main problem statement, another important question needs to be clarified: *What constitutes a satisfactory solution?* A major contribution of this article is the introduction of two symmetry properties drawing inspiration from the principle of network neutrality: *compositionality* and *content-neutrality*. Compositionality ensures that a broadcast abstraction does not discriminate based on the application using it. This property is essential for constructing higher-level systems in a modular way, as a composition of independent components that share the same underlying broadcast abstraction. Content-neutrality ensures that the behavior of a broadcast abstraction does not depend on the content of the messages.

An Inexistence Result. Having defined what constitutes an appropriate broadcast abstraction, we are now equipped to address our problem statement, to which we provide a negative answer: we demonstrate that no broadcast abstraction, which is both content-neutral and compositional, is computationally equivalent to k-SA for $1 < k < n$.

To the best of our knowledge, this research presents the first instance where a coordination problem has been proven to lack an equivalent broadcast abstraction. This proof introduces an additional layer of abstraction compared to standard impossibility proofs.

Classical impossibility proofs typically demonstrate that a given specification cannot be implemented within a certain model. In contrast, our approach, which involves dealing with a specification that remains an unknown variable, presents new challenges. These challenges require more precise definitions of the computing model and the scheduler, along with a more careful analysis of arguments related to the expected behavior of the broadcast abstraction.

Paper Organization. The remainder of this paper is organized as follows. Section 2 delineates the crash-prone asynchronous message-passing distributed computing model pertinent to our results. Subsequently, Section 3 defines permissible broadcast abstractions, introducing the novel symmetry properties. Section 4 then establishes that no content-neutral and compositional broadcast abstraction is computationally equivalent to k-set agreement for $1 < k < n$. Finally, Section 5 concludes the paper.

2 Computing Model

The computing model is the classical asynchronous crash-prone message-passing model.

Process Model. The computing model consists of a set Π of n sequential processes denoted $p_1, \ldots, p_n$. Each process operates asynchronously, meaning it progresses at its own speed, which is arbitrary, unknown to other processes, and may vary through time. A process may halt prematurely (crash failure) but executes its local algorithm correctly until it possibly crashes. We do not assume any bound on the number of processes that may crash, hence $t = n - 1$. A process that crashes in a run is said to be *faulty*. Conversely, a process is called *correct* or *non-faulty* if it does not crash.

Communication Model. Communication between each pair of processes occurs through two uni-directional channels, one for each direction. Consequently, the network is complete: any process p_s can directly send a message to any process p_r (including itself). Each channel is reliable (free from loss, corruption, or message creation), not necessarily FIFO (First-In/First-Out), and asynchronous (messages have finite but unbounded transit times).

A process p_s invokes the operation "`send` m `to` p_r" to send a message whose content is m to p_r. The event "`receive` m `from` p_s" occurs at p_r upon receiving a message whose content is m from p_s. Although messages may share content, each sent message is unique. By a slight abuse of language, we say that "a process p_i sends (resp. receives) a message m" when p_i sends or receive a message whose content is m. The communication channels are governed by the following properties:

SR-Validity. If a process p_r receives a message m from p_s, then p_s has indeed sent m to p_r.

SR-No-Duplication. No process receives the same message more than once.

SR-Termination. If a process p_s sends a message m to a correct process p_r, then p_r will eventually receive m from p_s.

It is important to note that, due to asynchrony in processes and message delivery, no process can ascertain whether another process has crashed or is merely slow.

Notation The acronym $\mathcal{CAMP}_n[\emptyset]$ denotes the described Crash-prone Asynchronous Message-Passing model without additional computational power. $\mathcal{CAMP}_n[H]$ represents $\mathcal{CAMP}_n[\emptyset]$ enhanced with the additional computational power denoted by H. For instance, $\mathcal{CAMP}_n[k\text{-SA}]$ denotes the model $\mathcal{CAMP}_n[\emptyset]$ in which processes have access to as many instances of the k-set agreement object as needed. Similarly, if B represents a broadcast abstraction, then $\mathcal{CAMP}_n[B]$ refers to the $\mathcal{CAMP}_n[\emptyset]$ model in which processes can broadcast and deliver messages via the abstraction B.

Execution An execution α is a sequence of steps, each represented as a pair $\langle p_i : a \rangle$, where $p_i \in \Pi$ represents a process, and a is an action occurring at p_i. These actions can be local computations, the invocation of primitives (such as message emissions), the triggering of local events (including message receptions), as well as invocations and responses of high-level operations as specified in the enriching hypothesis H. Examples of such high-level operations include proposing or deciding on a value in a k-SA object.

We define an execution α as being admitted by the model $\mathcal{CAMP}_n[H]$ if it satisfies several criteria: it must adhere to the three properties of the communication channels, namely SR-Validity, SR-No-Duplication, and SR-Termination; it must conform to all properties specified by H and the high-level abstractions it provides; and it must be *well-formed* with respect to the algorithm it executes, as delineated by the following definition.

Definition 1 (Well-Formed Executions)**.** *Consider $\mathcal{A}$, an algorithm that implements a high-level abstraction A within the $\mathcal{CAMP}_n[H]$ model. An execution is deemed* Well-Formed *with respect to $\mathcal{A}$ if it fulfills the following conditions:*

- *Only processes labeled from p_1 to p_n take actions in α;*
- *A process only invokes an operation of A after having returned from its previous invocations;*
- *The actions undertaken by any process between the invocation of an operation on A and its corresponding response (if one exists), excluding local events (such as message receptions and deliveries), must align with the actions specified by $\mathcal{A}$.*

3 Defining Admissible Broadcast Abstractions

3.1 Interface of broadcast abstractions

A broadcast abstraction denoted as B, enables processes to broadcast messages that are guaranteed to be delivered at least to all correct processes. Consequently, all broadcast abstractions share the same interface, comprising a single operation named `broadcast` and an event called `deliver`.

A process p_i invokes the operation "B.`broadcast`(m)" to utilize B for broadcasting a message whose content is m. This is referred to as p_i B-broadcasting a message whose

content is m. Subsequently, the event "B.`deliver` m `from` p_i" might be triggered at some processes p_j, leading us to say that p_j B-delivers a message m from p_i. Analogous to the send/receive interface, it is assumed that each broadcast message is unique, regardless of having identical content. However, for the seek of conciseness, we amalgamate a message and its content whenever the distinction is immatereal. The set of all messages that can be broadcast during an execution is denoted by $\mathbb{M}$. The following properties must be verified by all broadcast abstractions.

BC-Validity. If a process p_i B-delivers a message m from p_j, then it is guaranteed that p_j has previously B-broadcast m.

BC-No-Duplication. A process will not B-deliver the same message more than once.

BC-Local-Termination. If a correct process invokes B.`broadcast`(m), it will eventually return from this invocation.

BC-Global-CS-Termination. If a correct process B-broadcasts a message m, then all correct processes will eventually B-deliver m.

The first two properties mentioned are classical safety properties and share the same definitions as their send/receive counterparts. The third property is a classical liveness property. It is important to note that the BC-Global-CS-Termination property only applies to correct processes. (The abbreviation "CS", standing for *correct sender*, emphasizes that this property is contingent on the sender's correctness.) Consequently, if a process p_i crashes during its execution of `broadcast`(m), it is permissible for some processes to deliver m while others do not, unless otherwise specified. This specification choice is intentionally made to allow for flexible definitions of liveness properties in broadcast abstractions.

In particular, the most basic broadcast abstraction that can be defined, only verifies the four properties defined above. In the $\mathcal{CAMP}_n[\emptyset]$ model, its implementation involves simply sending messages to all participants. For this reason, it is commonly referred to as Send-To-All Broadcast.

Remark on Expressiveness Set-Constrained-Delivery Broadcast (SCD Broadcast) [16] and its extension k-SCD Broadcast [15] are two examples of broadcast abstractions whose specification slightly deviate from the propose interface. Indeed, these abstractions deliver messages not individually, but within unordered sets of messages, hence the designation. While it is easy to generalize the definitions and the proofs to accommodate this particularity, doing so would compromise readability. For the sake of maintaining clarity, we have chosen not to pursue this generalization

A local ordering property When considered together, the BC-Validity and BC-Global-CS-Termination properties ensure that a step $\langle p_i : B.\texttt{broadcast}(m)\rangle$ executed by a correct process p_i is always followed by a step $\langle p_i : B.\texttt{deliver}\ m\ \texttt{from}\ p_i\rangle$. In a similar vein, the BC-Local-Termination property guarantees that the B-broadcasting step is consistently succeeded by $\langle p_i : \textbf{return from}\ B.\texttt{broadcast}(m)\rangle$. However, there is no inherent order between the delivery of its own message m by p_i, and p_i returning

from its B.`broadcast` invocation. Once again, this specification choice is deliberately made to accommodate flexible definitions of broadcast abstractions. For instance, certain abstractions may require that B.`broadcast` returns immediately, or they may wait until the broadcast message has been delivered, while others may delegate the decision to the implementation. Nevertheless, it is occasionally beneficial to reason based on a fixed total order among the three events. Adopting the terminology suggested in [9], we augment all broadcast abstractions with a trait B.`sync-broadcast`(m), defined as: B.`broadcast`(m); `wait`$(m$ has been B-delivered locally). For every message m B-broadcast by each correct process p_i, the following three steps occur sequentially: $\langle p_i$: B.`sync-broadcast`$(m)\rangle$, followed by $\langle p_i$: B.`deliver` m `from` $p_i\rangle$, and then $\langle p_i$: **return from** B.`sync-broadcast`$(m)\rangle$.

3.2 Symmetry Properties of Broadcast Abstractions

Broadcast abstractions can be characterized by additional predicates on the set of executions they admit. Typically, these predicates fall into two categories. On one hand, liveness predicates ensure message delivery in scenarios not covered by Send-To-All Broadcast. Examples of this include the definitions of Reliable Broadcast and Uniform Reliable Broadcast [13]. On the other hand, safety predicates concern the relative order in which processes deliver messages. Examples in this category are FIFO Broadcast, Causal Broadcast [3, 24], Mutual Broadcast [9], Pair Broadcast [10], k-Bounded Order Broadcast [15], and Total Order Broadcast [21].

As highlighted in the Introduction, not all predicates are equally appropriate for the design of a broadcast abstraction. In this section, we introduce two novel symmetry properties inspired by the broader principle of "network neutrality". Network neutrality advocates, among other tenets, that network services should not discriminate based on the content, sender, or usage of the messages they transmit. While concerns regarding network neutrality often arise in discussions about non-functional aspects of message routing, they hold significant relevance for the functional design of broadcast abstractions. Within this framework, we interpret network neutrality to include two essential symmetry properties: *Compositionality* and *Content Neutrality*. These properties assert that the broadcast abstraction should impartially treat all messages, irrespective of their usage or content.

Compositionality. Building upon earlier concepts, one might propose characterizing iterated k-SA using an iterated version of the broadcast described in the Introduction. This approach, denoted by *k-Stepped Broadcast*, would be characterized by the following ordering property: "for each a, define S_a as the set containing the ath message broadcast by each process; then there are at most k messages $m \in S_a$ such that some process delivers m before any other message in S_a". Now, the ordering of messages within each S_a set could determine the set of values decided on a sequence of k-SA objects, and conversely, thereby establishing equivalence.

However, since the ordering property only governs specific sets of messages, it imposes an overly precise communication pattern (lock-step pattern), severely limiting its utility for constructing modular higher-level systems. Indeed, a broadcast abstraction typically serves as a system-wide abstraction, manifesting as a single service that is shared among

multiple algorithms for solving higher-level tasks. Consider, for instance, a system that integrates two applications built upon the same service that provides this broadcast abstraction: the iterated k-SA algorithm described above and a messaging service utilizing only the Reliable Broadcast capabilities of k-Stepped Broadcast. Each application employs only a distinct subset of the system's messages, and the messages used by the messaging service interfere with the communication pattern followed by the k-SA algorithm. Unless a shared global counter is used to track the number a of broadcast messages, the applications cannot fully benefit from the offered ordering property. This limitation hinders their independent design and composition.

Compositionality is the property required for the implementation of composable algorithms or applications on top of a broadcast abstraction. Each higher-level construction uses only a subset of the messages broadcast at the lower level. Compositionality ensures that each of these message sets maintains the same ordering properties as those of the entire message set. This is achieved by requiring that the restriction of an admissible execution to any subset of its messages remains an admissible execution.

Definition 2 (Compositionality). *A broadcast abstraction B is* compositional *if, for all executions α admissible by B, and for any set of messages M, the restriction of α onto the messages of M is also admissible by B.*

To exemplify the Compositionality property, let us demonstrate that k-BO Broadcast is compositional. Indeed, its ordering property is defined by a predicate P that must be satisfied by any set S of messages. Specifically, $P(S)$ stipulates that if S contains at least $k+1$ messages, then at least two of these messages must be delivered in the same order by all processes. Consider an execution α admissible by k-BO Broadcast, with its set of sent messages denoted as M_α. For any subset $M \subseteq M_\alpha$ of these messages, every subset S of M is also a subset of M_α, ensuring $P(S)$ is satisfied, which is the condition imposed by compositionality. This logical framework can be applied to all broadcast abstractions defined by a predicate on the relative order of emission and delivery events, independent of the context of the complete execution. Notably, this encompasses all broadcast abstractions mentioned in the Introduction and, to the best of our knowledge, all broadcast abstractions currently described in the literature.

Conversely, the limitations of compositionality can be highlighted by revisiting our initial counter-example involving k-Stepped Broadcast. Consider an execution α where two processes, p_0 and p_1, engage in the 1-Stepped-broadcasting of two messages each: m_i and m'_i. In α, p_0 delivers the messages $[m_0, m'_0, m_1, m'_1]$ in this order. Simultaneously, p_1 delivers the sequence $[m_0, m_1, m'_0, m'_1]$. Although both processes deliver m_0 before m_1 and m'_0 before m'_1, conforming to the 1-stepped predicate, the execution's restriction to the subset $\{m'_0, m_1\}$ fails to maintain this order. This issue arises because the definition relies on the sequence number a of the broadcast messages, which is only contextually relevant within the full scope of the execution and varies when subsets of messages are considered.

Content Neutrality. The second property asserts that the defining predicates of a broadcast abstraction should be applicable based solely on the occurrence of broadcast and delivery events during an execution, independent of the message's content. Hence, if some messages get substituted by other within an execution, it should not hinder the

admissibility of the execution. Content neutrality then stipulates that an admissible execution must remain admissible even when some of its messages are replaced.

Definition 3 (Content-Neutrality)**.** *A broadcast abstraction B is* content-neutral *if, for all executions α admissible by B, and all injective functions r on the set of messages, the execution obtained by replacing all messages m by $r(m)$ in α, is also admissible by B.*

It is important to note that while all broadcast abstractions mentioned in the Introduction adhere to the Content-Neutrality property, this is not necessarily true for all broadcast abstractions found in the literature. For instance, Generic Broadcast [20] supposes that the messages it transmits encapsulate a *command*, i.e., an operation invocation on a replicated data structure implemented using the broadcast. In the vein of Generalized Paxos [18], processes only need to agree on a common delivery order for pairs of non-commuting commands, as executing commuting commands in different orders does not compromise the consistency of the implemented data structure. However, *specifying* such a broadcast necessitates differentiating between messages, which violates content neutrality.

Returning to the present paper, it would be straightforward to propose a broadcast abstraction equivalent to k-set agreement that is not content-neutral. For example, one could enforce an ordering property that only applies to messages of a special type $\text{SA}(ksa, v)$, where ksa uniquely identifies a k-SA object and v is a value proposed to ksa. This would require that, for each ksa, at most k distinct messages of the form $\text{SA}(ksa, _)$ are delivered first by any process. However, such an approach would not be conducive to understanding the essence of k-set agreement. In the following section, we focus exclusively on content-neutral broadcast abstractions.

4 On Capturing k-Set Agreement

In this section, we establish that no broadcast abstraction, which is both content-neutral and compositional, can be equivalent to k-set agreement in the model $\mathcal{CAMP}_n[\emptyset]$ when $1 < k < n$. It is evident that for $k = 1$, boils down to consensus, which is characterized by Total Order broadcast; conversely, for $k = n$, n-set agreement can be trivially solved without any communication, rendering it equivalent to Send-To-All Broadcast.

We begin by recalling the definition of k-set agreement in Section 4.1. The ensuing proof is structured as a *reductio ad absurdum.* We hypothesize the existence of a broadcast abstraction B satisfying the aforementioned conditions. Two deterministic reduction algorithms are then considered: $\mathcal{A}$, which implements k-set agreement in the model $\mathcal{CAMP}_n[B]$, and $\mathcal{B}$, which implements B in the model $\mathcal{CAMP}_n[k\text{-SA}]$. For any $N \in \mathbb{N}$, Section 4.2 constructs an execution $\alpha_{k,N,B,\mathcal{B}}$ (as defined in Definition 4 and illustrated on Figure 1) of $\mathcal{B}$, wherein each process B-delivers N of its own messages before any messages from other processes. Subsequently, in Section 4.3, we demonstrate that sufficiently large values of N inhibit $\mathcal{A}$ from effectively resolving k-set agreement, thereby leading to a contradiction.

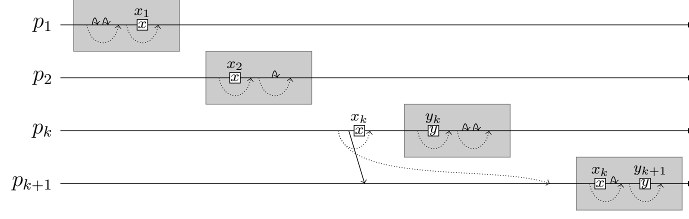

Figure 1: Illustration of the adversarial execution $\alpha_{k,N,B,\mathcal{B}}$ for $k = 3$ and $N = 2$, extending up to Line 7 of Algorithm 1. Within the $\mathcal{CAMP}_{k+1}[k\text{-SA}]$ model, plain arrows signify sent and received messages, while white squares denote propositions on k-SA objects, with their respective decided values indicated above. In the context of the $\mathcal{CAMP}_{k+1}[B]$ model, simulated by Algorithm $\mathcal{B}$, dotted arrows represent B-broadcast and B-delivered messages. Notably, the final N messages of each process, enclosed in grey boxes, are incompatible with an implementation of k-set agreement.

4.1 Definition of k-Set Agreement

k-Set agreement, first introduced by S. Chaudhuri in [8] (refer to [22] for a comprehensive survey of k-set agreement in various contexts), was conceptualized to analyze the relationship between the maximum number of allowable process failures (t) and the feasible degree of agreement (k) among processes. Here, a lower k value signifies a higher degree of agreement, with the ultimate agreement being $k = 1$, which corresponds to consensus.

The k-Set agreement problem (abbreviated as k-SA) is a one-shot agreement problem that equips processes with a singular operation, denoted `propose`(). When a process p_i invokes $ksa.\texttt{propose}(v_i)$ on a k-SA object ksa, it is said to "propose the value v_i to ksa". This operation yields a return value v, at which point the invoking process is described as "deciding v on ksa", and "v becomes a decided value". In other words, the steps $\langle p_i : \textbf{return } w \textbf{ from } ksa.\texttt{propose}(v)\rangle$ and $\langle p_i : ksa.\texttt{decide}(w)\rangle$ are interpreted as synonymous. It is a standard assumption that each process is limited to a single invocation of `propose`() on any given k-SA object, ensuring the problem's one-shot nature.

k-Set agreement is defined by the following properties.

k-SA-Validity. If a process decides a value v, then v was proposed by some process.

k-SA-Agreement. No more than k distinct values are decided upon by the processes.

k-SA-Termination. Every non-faulty process that invokes `propose`() eventually decides.

4.2 Definition of the adversarial scheduler

For brevity in this subsection, we pose $k > 1$ and $N > 0$. Additionally, we postulate the existence of an algorithm $\mathcal{B}$ that implements a certain broadcast abstraction B within the model $\mathcal{CAMP}_{k+1}[k\text{-SA}]$. The argument is then generalized to the case where $n > k + 1$ in the proof of the main theorem. This is achieved by observing that processes p_j, for $j > k + 1$, may fail at the beginning of the execution. The adversarial execution $\alpha_{k,N,B,\mathcal{B}}$ is constructed by an adversarial scheduler that follows the procedure outlined in

```
1  Procedure adversarial_scheduler(k, N, B, 𝓑) is:
2      α ← ε; sent ← ∅; decided ← [[⊥, ...], ..., [⊥, ...]];
3      for i from 1 to k + 1 do
4          step ← ⊥; local_del ← 0;
5          while local_del < N do
6              if step = ⊥ ∨ step = ⟨p_i : return from B.sync-broadcast(SYNCH)⟩
                then
7                  step ← ⟨p_i : B.sync-broadcast(SYNCH)⟩;
8              else step ← p_i's next local step in C(α), according to 𝓑;
9              α ← α ⊕ step;
10             if step = ⟨p_i : send m to p_i⟩ for some m then
11                 α ← α ⊕ ⟨p_i : receive m from p_i⟩
12             else if step = ⟨p_i : send m to p_j⟩ for some m and j ≠ i then
13                 sent.add(⟨m, i, j⟩)
14             else if step = ⟨p_i : B.deliver m from p_i⟩ for some m then
15                 local_del ← local_del + 1;
16             else if step = ⟨p_i : ksa.propose(v)⟩ for some k-SA object ksa and v
                then
17                 if i = k + 1 ∧ ∀j ≤ k : decided[ksa][j] ≠ ⊥ then
18                     decided[ksa][i] ← decided[ksa][k];
19                 else decided[ksa][i] ← v;
20                 α ← α ⊕ ⟨p_i : ksa.decide(decided[ksa][i])⟩;
21                 if i = k ∧ ∀j ≤ k : decided[ksa][j] ≠ ⊥ then
22                     foreach m : ⟨m, k, k + 1⟩ ∈ sent do
23                         α ← α ⊕ ⟨p_{k+1} : receive m from p_k⟩;
24                         sent.remove (⟨m, k, k + 1⟩);
25                     local_del ← −1;

26     foreach ⟨m, i, j⟩ ∈ sent do α ← α ⊕ ⟨p_j : receive m from p_i⟩;
27     return α;
```

Algorithm 1: Adversarial scheduler used by Definition 4

Algorithm 1. As validated by lemmas 1-8, $\alpha_{k,N,B,\mathcal{B}}$ constitutes an execution admitted by the model $\mathcal{CAMP}_{k+1}[k\text{-SA}]$.

The algorithm begins with a sequential execution of all processes, ranging from p_1 to p_{k+1}. During this phase, each process p_i repetitively calls B.`sync-broadcast`(SYNCH) until it has B-delivered N of its own messages. This part of the execution remains indistinguishable to p_i from an execution $\gamma_{k,N,B,\mathcal{B},i}$, where other processes p_j would have crashed before the local delivery of their own N messages. To achieve this, processes decide on their own value on k-SA objects whenever possible, and the transmission of their messages to other processes is deferred by the scheduler until the end of this phase. However, a complication arises when all processes propose a value on the same k-SA object ksa. In such scenarios, p_{k+1} is compelled to decide on the value proposed by p_k to maintain the k-SA-Agreement property. This decision renders p_{k+1}'s execution

distinguishable from a scenario where p_k had initially crashed, allowing p_{k+1} to await p_k's message. As a result, all messages sent by p_k to p_{k+1} are received by p_{k+1} (lines 22-24), and the messages that p_k B-broadcast before this juncture are excluded from its count of N messages.

Subsequently, in a later phase of the algorithm, all processes receive all messages that were sent to them in the initial stage but have yet to be received, as delineated in Line 26. Algorithm 1 concludes by returning the execution halted at this juncture. Notably, at this point of termination, not all messages that have been B-broadcast are necessarily B-delivered by every process. However, this does not pose a problem for our analysis: the counterexample required for the proof in the following section involves a safety property that is already violated in the execution prefix returned by the algorithm. The scheduler maintains the following main variables:

- α, which is initially an empty sequence ε, is the execution currently being constructed.
- i, which stores the identifier of the process currently under execution.
- *sent*, initially set to $\emptyset$, is a set of triplets. A triplet $\langle m, i, j\rangle$ is included in *sent* when a message m has been sent by process p_i to process p_j, but has not yet been received by p_j.
- $decided[ksa][j]$ is a two-dimensional associative array. The keys *ksa* correspond to k-SA objects used in $\mathcal{B}$, and j represents process identifiers. The values are either potential values that can be proposed to k-SA objects in $\mathcal{B}$, or a special value $\perp$ that cannot be proposed. For each *ksa* and j, $decided[ksa][j]$ is initially set to $\perp$. It is later updated to value w when the process p_j decides on w for *ksa*.
- *local_del* tracks the number of messages that process p_i B-delivers from itself, while avoiding communication with other processes. Under normal conditions, *local_del* cycles through values from 0 to $N-1$ for each process p_i. However, if communication between processes p_k and p_{k+1} is inevitable during the execution of a B.`sync-broadcast`(m) operation by p_k, *local_del* is assigned a value of -1. This assignment signifies that *local_del* will be reset to 0 once p_k completes its B-broadcast operation. Consequently, this setup enables p_k to B-deliver N of its own messages (excluding m) without engaging in communication.
- *step* identifies the subsequent step to be executed by Process p_i, represented either by the pair $\langle p_i, action\rangle$ or by the special value $\perp$ if the step is yet to be determined. In this context, there are two primary scenarios to consider. Firstly, if p_i has initiated the B.`sync-broadcast` operation but has not yet completed this invocation, then the deterministic algorithm B is responsible for defining the subsequent step that p_i must execute (Line 8). This step is crucial to fulfilling the BC-LOCAL-TERMINATION property of B within the configuration $C(\alpha)$, which delineates its local state after the execution α. In the second scenario, if the aforementioned condition does not hold, p_i proceeds to B-broadcast a new message, specifically SYNCH.

Definition 4 now outlines the adversarial executions $\alpha_{k,N,B,\mathcal{B}}$, $\beta_{k,N,B,\mathcal{B}}$, and $\gamma_{k,N,B,\mathcal{B},i}$. Our subsequent objective is to demonstrate that $\alpha_{k,N,B,\mathcal{B}}$ qualifies as an admissible execution of the $\mathcal{CAMP}_{k+1}[k\text{-SA}]$ model. It is required to verify that $\alpha_{k,N,B,\mathcal{B}}$ is well-formed (as per Lemma 6), upholds the three defining properties of the k-set agreement: k-SA-Validity (Lemma 1), k-SA-Agreement (Lemma 2), and k-SA-Termination (Lemma 3). and ensures compliance with the three properties of send/receive communication: SR-Validity (Lemma 4), SR-No-Duplication (Lemma 5), and SR-Termination (Lemma 8).

Definition 4 (Adversarial execution)**.** *The following executions are defined:*

- *$\alpha_{k,N,B,\mathcal{B}}$ is the execution produced by the procedure* `adversarial_scheduler`$(k, N, B, \mathcal{B})$*, as delineated in Algorithm 1.*
- *$\beta_{k,N,B,\mathcal{B}}$ constitutes a subset of $\alpha_{k,N,B,\mathcal{B}}$, encompassing only those steps that involve events associated with B. This includes the invocations of, or the responses from, the B.*`broadcast` *operation, as well as any B-delivery event.*
- *For each $i \in 1, ..., k+1$, $\gamma_{k,N,B,\mathcal{B},i}$ is derived from $\alpha_{k,N,B,\mathcal{B}}$ by limiting it to, on the one hand, the steps of process p_i occurring strictly before Line 26; and on the other hand, the steps performed by p_k that are succeeded by a reset of local_del on Line 25.*

  In these executions, all processes $p_j \notin \{p_i, p_k\}$ are assumed to have crashed initially. Furthermore, p_k is treated as having crashed before executing its first step in $\alpha_{k,N,B,\mathcal{B}}$ that is absent in $\gamma_{k,N,B,\mathcal{B},i}$, should such a step be present.

Lemma 1 (k-SA-Validity)**.** *In the executions $\alpha_{k,N,B,\mathcal{B}}$ and $\gamma_{k,N,B,\mathcal{B},i}$, if a process decides on a value w on a k-SA object ksa, then the value w was proposed by some process on ksa.*

Proof. Assume that $\alpha_{k,N,B,\mathcal{B}}$ includes a step $\langle p_j : ksa.\texttt{decide}(w)\rangle$. This step originates from Line 20, following p_j's invocation of $ksa.\texttt{propose}(v)$. Consequently, $w = decided[ksa][j]$, that was set either on Line 18 or Line 19.

- If $decided[ksa][j]$ was assigned on Line 19, then $w = v$. The step $\langle p_j : ksa.\texttt{propose}(v)\rangle$ would have been included in $\alpha_{k,N,B,\mathcal{B}}$ at Line 9.
- Otherwise, $w = decided[ksa][k]$, and per Line 17, $i = k + 1$. In this case, $decided[ksa][k] \neq \perp$ was previously set by p_k in $\alpha_{k,N,B,\mathcal{B}}$ on Line 19, following the inclusion of the step $\langle p_k : ksa.\texttt{propose}(w)\rangle$ in $\alpha_{k,N,B,\mathcal{B}}$.

This sequence of events establishes the property for $\alpha_{k,N,B,\mathcal{B}}$. Consider now the case of $\gamma_{k,N,B,\mathcal{B},i}$ containing a step $\langle p_j : ksa.\texttt{decide}(w)\rangle$, following the same case disjunction as before. In the case of Line 19, the property holds because $\gamma_{k,N,B,\mathcal{B},i}$ and $\alpha_{k,N,B,\mathcal{B}}$ both encompass identical `propose` and `decide` steps executed by p_j. In the second case, the fulfillment of the condition at Line 21 for p_k leads to the subsequent reset of *local_del* on Line 25. Therefore, in both cases, the step $\langle p_k : ksa.\texttt{propose}(w)\rangle$ is also included in $\gamma_{k,N,B,\mathcal{B},i}$. □

Lemma 2 (k-SA-Agreement)**.** *In both $\alpha_{k,N,B,\mathcal{B}}$ and $\gamma_{k,N,B,\mathcal{B},i}$ executions, no more than k distinct values are decided on any given k-SA object.*

Proof. By the definition of $\gamma_{k,N,B,\mathcal{B},i}$, at most two processes, specifically p_i and p_k, are capable of deciding a value in $\gamma_{k,N,B,\mathcal{B},i}$, satisfying the condition as $2 \leq k$.

Assume that in $\alpha_{k,N,B,\mathcal{B}}$, $k+1$ distinct values are decided. Given that processes execute sequentially, processes p_1 through p_k would have already recorded their value in $decided[ksa][.]$ before p_{k+1} proposing its value. Consequently, the condition at Line 17 would be met, leading to p_{k+1} deciding the same value as p_k, thus resulting in a contradiction. □

Lemma 3 (k-SA-Termination)**.** *In the executions $\alpha_{k,N,B,\mathcal{B}}$ and $\gamma_{k,N,B,\mathcal{B},i}$, if a process proposes a value on a k-SA object ksa, then this process will also decide a value on ksa.*

Proof. Suppose that $\alpha_{k,N,B,\mathcal{B}}$ includes a step $\langle p_j : ksa.\texttt{propose}(v)\rangle$. This step was introduced on Line 9. Subsequently, the condition at Line 16 is satisfied, leading to the inclusion of a step $\langle p_j : ksa.\texttt{decide}(w)\rangle$ in $\alpha_{k,N,B,\mathcal{B}}$ at Line 20. This confirms the lemma for $\alpha_{k,N,B,\mathcal{B}}$.

Now, assume $\gamma_{k,N,B,\mathcal{B},i}$ contains a step $\langle p_j : ksa.\texttt{propose}(v)\rangle$. Here, j can only be either i or k.

- If $j = i$, then $\gamma_{k,N,B,\mathcal{B},i}$ includes the same step $\langle p_j : ksa.\texttt{decide}(w)\rangle$ as found in $\alpha_{k,N,B,\mathcal{B}}$.
- If $j = k$, it is important to note that the steps $\langle p_j : ksa.\texttt{propose}(v)\rangle$ (at Line 9) and $\langle p_j : ksa.\texttt{decide}(w)\rangle$ (at Line 20) cannot be isolated by a reset of *local_del* on Line 25. Therefore, if the proposal step exists in $\gamma_{k,N,B,\mathcal{B},i}$, the decision step must also be present.

In both scenarios, the lemma's condition is satisfied in $\gamma_{k,N,B,\mathcal{B},i}$, thus completing the proof. □

Lemma 4 (SR-Validity)**.** *In the executions $\alpha_{k,N,B,\mathcal{B}}$ and $\gamma_{k,N,B,\mathcal{B},i}$, if a process p_r receives a message m from process p_s, then process p_s has indeed sent m to p_r.*

Proof. Assume that $\alpha_{k,N,B,\mathcal{B}}$ includes a step $\langle p_r : \texttt{receive}\ m\ \texttt{from}\ p_s\rangle$. This step is either introduced on Line 11 following a step $\langle p_s : \texttt{send}\ m\ \texttt{to}\ p_r\rangle$ where $r = s$, or on Line 23 or Line 26 when $\langle m, s, r\rangle \in sent$. The triplet $\langle m, s, r\rangle$ is added to *sent* only on Line 13, implying that $\langle p_s : \texttt{send}\ m\ \texttt{to}\ p_r\rangle$ was previously included in $\alpha_{k,N,B,\mathcal{B}}$ on Line 9. This confirms the lemma for $\alpha_{k,N,B,\mathcal{B}}$.

Now, consider a reception step in $\gamma_{k,N,B,\mathcal{B},i}$. Given the previous argument, $\alpha_{k,N,B,\mathcal{B}}$ must contain a corresponding emission step. Since reception steps from Line 26 are not part of $\gamma_{k,N,B,\mathcal{B},i}$, there are two possible scenarios:

- If the reception step is added to $\gamma_{k,N,B,\mathcal{B},i}$ on Line 11, then the preceding emission step is also included in $\gamma_{k,N,B,\mathcal{B},i}$.
- If the reception step is added to $\gamma_{k,N,B,\mathcal{B},i}$ on Line 23, the sender is p_k, and *local_del* was reset on Line 25 subsequently. Therefore, the emission step is also present in $\gamma_{k,N,B,\mathcal{B},i}$.

Both cases confirm the lemma's condition on $\gamma_{k,N,B,\mathcal{B},i}$, thus completing the proof. □

Lemma 5 (SR-No-Duplication)**.** *In both $\alpha_{k,N,B,\mathcal{B}}$ and $\gamma_{k,N,B,\mathcal{B},i}$ executions, each message is received at most once.*

Proof. The property for $\alpha_{k,N,B,\mathcal{B}}$ is substantiated by the message reception mechanics: a message can only be received on Line 11, in which case it is not added to *sent* so it is not received again later, on Line 23 followed by its removal from *sent*, or singularly on Line 26 due to *sent*'s set semantics. Since $\gamma_{k,N,B,\mathcal{B},i}$ comprises only a subset of $\alpha_{k,N,B,\mathcal{B}}$'s reception events, the lemma is valid for $\gamma_{k,N,B,\mathcal{B},i}$ as well. □

Lemma 6 (Well-Formed Executions)**.** *$\alpha_{k,N,B,\mathcal{B}}$ and $\gamma_{k,N,B,\mathcal{B},i}$ are well-formed executions of $\mathcal{CAMP}_{k+1}[H]$ with respect to $\mathcal{B}$.*

Proof. To validate the property for $\alpha_{k,N,B,\mathcal{B}}$, we observe that the participation of only processes p_1 to p_{k+1} stems for (1) loop bounds defined on Line 3, and (2) the SR-Validity property and the correctness of $\mathcal{B}$ for the receiving processes on Line 26. A process initiates the operation B.`broadcast` either at the start of its execution on Line 4, or immediately after returning from its previous invocation, as indicated on Lines 6 and 7. This ensures adherence to the required pattern of alternating invocations and responses. Furthermore, the sequence of steps a process follows between its invocations and responses is consistent with $\mathcal{B}$, as defined on Line 8.

As for $\gamma_{k,N,B,\mathcal{B},i}$, the property comes from the fact that for all processes p_j, the sequence of steps taken by p_j in $\gamma_{k,N,B,\mathcal{B},i}$ is a prefix of the sequence of steps taken by p_j in $\alpha_{k,N,B,\mathcal{B}}$. □

Lemma 7 (Termination of Algorithm 1)**.** *The execution $\alpha_{k,N,B,\mathcal{B}}$ is finite.*

Proof. Assume for contradiction that $\alpha_{k,N,B,\mathcal{B}}$ contains an infinite number of steps. Given that Algorithm 1 includes no recursion and only one while loop, there exists some $i \in \{1, ..., k+1\}$ engaged in an infinite loop starting at Line 5 with $local_del < N$ remaining true indefinitely.

By Lemmas 1-5, $\gamma_{k,N,B,\mathcal{B},i}$ satisfies all the conditions required for an admissible execution, except SR-Termination. Let us establish that $\gamma_{k,N,B,\mathcal{B},i}$ also verifies SR-Termination:

- For $i < k$, $\gamma_{k,N,B,\mathcal{B},i}$ contains only messages sent by p_i, as the ith iteration does not terminate. Process p_i receives its own messages on Line 11, and others are not required to receive them as they have crashed.

- For $i = k$, similar to the previous case, $\gamma_{k,N,B,\mathcal{B},i}$ includes only messages by p_i by definition of $\gamma_{k,N,B,\mathcal{B},i}$. Message reception follows the same logic as above.

- For $i = k + 1$, note that p_k is considered faulty in $\gamma_{k,N,B,\mathcal{B},i}$ due to (1) taking a finite number of steps in $\alpha_{k,N,B,\mathcal{B}}$ since p_i is executed after p_k's last step, and (2) the condition $local_del < N$ only becoming false post Line 15 which is preceeded by a step $\langle p_k : B.\texttt{deliver}\ m\ \texttt{from}\ p_k \rangle$ that belongs to $\alpha_{k,N,B,\mathcal{B}}$ but not $\gamma_{k,N,B,\mathcal{B},i}$. Therefore, suffices to show that p_i receives all messages directed to it. Only p_k and p_i send messages in $\gamma_{k,N,B,\mathcal{B},i}$. Process p_i receives its own messages on Line 11, and all messages sent by p_k to p_i in $\gamma_{k,N,B,\mathcal{B},i}$ are sent prior to the reset of *local_del*, hence they are received by p_i on Line 23.

Therefore, $\gamma_{k,N,B,\mathcal{B},i}$ is an execution admitted by the model $\mathcal{CAMP}_{k+1}[k\text{-SA}]$, in which p_i takes an infinite number of steps. By correctness of $\mathcal{B}$ and the BC-Global-CS-Termination property of B, all B-broadcast messages by p_i in $\gamma_{k,N,B,\mathcal{B},i}$ must eventually be B-delivered by p_i in $\gamma_{k,N,B,\mathcal{B},i}$. Moreover, since $\gamma_{k,N,B,\mathcal{B},i}$ and $\alpha_{k,N,B,\mathcal{B}}$ contain the same steps of p_i, all messages B-broadcast by p_i in $\alpha_{k,N,B,\mathcal{B}}$ are eventually B-delivered by p_i in $\alpha_{k,N,B,\mathcal{B}}$. Since p_i immediately B-broadcasts a new message after returning from its previous B.`sync-broadcast` invocation (Lines 6-7), p_i B-delivers an infinite number of messages from itself, and repeatedly increments *local_del* on Line 15. As *local_del* is bounded by N, it must be reset on Line 25 infinitely, following proposals to k-SA objects.

Let K be the set of k-SA objects such that p_i executes Line 25 after proposing a value to them. Given the one-time proposal limit per k-SA object, K is infinite. Based on Line 21, $i = k$, and $\textit{decided}[\textit{ksa}][1] \neq \bot$ for all $\textit{ksa} \in K$. However, $\textit{decided}[\textit{ksa}][1]$ is set during the first iteration for an infinite number of distinct k-SA objects. This indicates that the first iteration does not terminate. This is a contradiction because (1) $k > 1$ so $k \neq 1$ and (2) the kth iteration of the loop started because p_k takes (an infinite number of) steps in $\alpha_{k,N,B,\mathcal{B}}$. This contradiction implies that $\alpha_{k,N,B,\mathcal{B}}$ must be finite, completing the proof. □

Lemma 8 (SR-Termination)**.** *In $\alpha_{k,N,B,\mathcal{B}}$[1], if a process p_s sends a message m to a correct process p_r, then p_r will eventually receive m from p_s.*

Proof. Consider a message m sent by p_s to p_r in $\alpha_{k,N,B,\mathcal{B}}$. A step $\langle p_s : \texttt{send}\ m\ \texttt{to}\ p_r\rangle$ is recorded in $\alpha_{k,N,B,\mathcal{B}}$ at Line 9. If $s = r$, then a step $\langle p_r : \texttt{receive}\ m\ \texttt{from}\ p_s\rangle$ is subsequently appended to $\alpha_{k,N,B,\mathcal{B}}$ at Line 11. In contrast, if $s \neq r$, $\langle m, s, r\rangle$ is added to *sent* at Line 13. As established in Lemma 7, $\alpha_{k,N,B,\mathcal{B}}$ is finite. If $\langle m, s, r\rangle$ remains in *sent* at the conclusion of the execution, then a step $\langle p_r : \texttt{receive}\ m\ \texttt{from}\ p_s\rangle$ is appended to $\alpha_{k,N,B,\mathcal{B}}$ at Line 26. Conversely, if $\langle m, s, r\rangle$ is not present in *sent*, it implies that it was removed at Line 24 subsequent to appending a step $\langle p_r : \texttt{receive}\ m\ \texttt{from}\ p_s\rangle$ to $\alpha_{k,N,B,\mathcal{B}}$ at Line 23. Therefore, in every case, p_r receives m from p_s. □

4.3 N-Solo Executions and the Contradiction

Definition 5 (N-solo executions)**.** *Let β be an execution of the model $\mathcal{CAMP}_n[B]$, and let $N \in \mathbb{N}$. We say that β is N-solo if, for each process p_i, there exist N messages $m_{i,1}, ..., m_{i,N}$ B-broadcast by p_i such that, in β, for all pairs of distinct processes p_i and p_j, p_i B-delivers all its own messages $m_{i,_}$ before B-delivering any of p_j's messages $m_{j,_}$.*

Lemma 9. *For all $k > 1$, and for every content-neutral and compositional broadcast abstraction B, if there exists an algorithm $\mathcal{A}$ that solves k-SA in the model $\mathcal{CAMP}_{k+1}[B]$, then there exists an integer $N > 0$ such that B does not allow any N-solo execution.*

Proof. Assume B is a broadcast abstraction and $\mathcal{A}$ is an algorithm solving k-SA in the model $\mathcal{CAMP}_{k+1}[B]$. It's noteworthy that $\mathcal{A}$ can be transformed into an alternative algorithm, $\mathcal{A}'$, which also solves k-SA in the same model but without relying on the point-to-point primitives `send` and `receive`. This transformation is feasible because the `send` and `receive` primitives can be trivially emulated using B. Moreover, the correctness of $\mathcal{A}'$

[1]Unlike previous lemmas, this property is not proven for $\gamma_{k,N,B,\mathcal{B},i}$ in the general case.

results from the compositionality of B. Specifically, the executions of $\mathcal{A}'$, when projected onto the set of messages shared with $\mathcal{A}$ (excluding those utilized solely for simulating `send/receive` in $\mathcal{A}'$), are admitted by $\mathcal{CAMP}_{k+1}[B]$, thereby yielding identical results in $\mathcal{A}$ and $\mathcal{A}'$.

Consider an execution α_i where a process $p_i \in \Pi$ proposes i to a k-SA object using $\mathcal{A}'$, while all other processes crash before taking any step. Due to the k-SA-Termination property of the k-SA object, p_i eventually decides on a value. The k-SA-Validity property ensures this value is i. Denote by $m_{i,1}$, ..., m_{i,N_i} the sequence of messages p_i B-delivers in α_i prior to its decision.

Let $N = \max\{1, N_1, ..., N_{k+1}\}$, and suppose B admits an N-solo execution β. Construct γ as the sub-execution of β containing, for each p_i, exactly N_i of the N messages B-broadcast by p_i, amongst those verifying the defining property of N-solo executions. Due to the BC-Compositionality property of B, γ is an execution admitted by B, where each process p_i B-delivers its N_i messages before any message from other processes. Now, define δ from γ by replacing each process p_i's N_i messages with the messages $m_{i,1}$, ..., m_{i,N_i} from α_i. The BC-Content-Neutrality of B ensures that δ is admitted by B. For each process p_i, α_i is indistinguishable from δ, as both executions involve identical B-broadcast and B-delivery steps for p_i. Hence, when $\mathcal{A}'$ is executed on δ, each p_i decides on its own value i, leading to $k+1$ distinct decisions. This contradicts the k-SA-Agreement property of k-SA. Therefore, such β cannot exist, implying B does not allow any N-solo execution. □

Lemma 10. *For all $k > 1$ and $N > 0$, if there exists an algorithm $\mathcal{B}$ that implements some broadcast abstraction B in the model $\mathcal{CAMP}_{k+1}[k\text{-}SA]$, then B admits an N-solo execution.*

Proof. Assume $k > 1$ and $N > 0$, and suppose an algorithm $\mathcal{B}$ implements a broadcast abstraction B in $\mathcal{CAMP}_{k+1}[k\text{-SA}]$. According to Lemmas 1-8, $\alpha_{k,N,B,\mathcal{B}}$ constitutes an admissible $\mathcal{CAMP}_{k+1}[k\text{-SA}]$ execution, thus by $\mathcal{B}$'s correctness, $\beta_{k,N,B,\mathcal{B}}$ is admitted by B. We aim to demonstrate that $\beta_{k,N,B,\mathcal{B}}$ is N-solo. For each $i \in \{1, ..., k+1\}$, the loop starting on Line 5 halts by Lemma 7, but only after *local_del* has been incremented at least N times on Line 15, without having been reset on Line 25. Each of these incrementations corresponds to the B-delivery, by p_i, of its own message $m_{i,local_del}$. We now prove that these $(k+1) \times N$ messages satisfy the criteria in Definition 5.

Consider two distinct processes p_i and p_j, assuming without loss of generality that $i < j$. Due to the sequential nature of the loop on Line 3, p_i B-delivers all its own messages before p_j even begins its B-broadcasts. Consequently, by the BC-Validity property of B, p_i completes delivering its messages before any of p_j's. Lemmas 1-6 confirm that $\gamma_{k,N,B,\mathcal{B},j}$ upholds all safety properties of send/receive and k-SA objects, and is well-formed, indicating $\gamma_{k,N,B,\mathcal{B},j}$ is the prefix of an execution of $\mathcal{CAMP}_{k+1}[k\text{-SA}]$. In $\gamma_{k,N,B,\mathcal{B},j}$, p_i does not B-broadcast its messages $m_{i,1}, ..., m_{i,N}$, hence p_j does not B-deliver these messages, as ensured by $\mathcal{B}$'s correctness and BC-Validity of B. Since $\alpha_{k,N,B,\mathcal{B}}$ and $\gamma_{k,N,B,\mathcal{B},j}$ share identical p_j steps before Line 26, in $\alpha_{k,N,B,\mathcal{B}}$, p_j B-delivers all its own messages before Line 26, without B-delivering any of the messages of p_i. Consequently, $\beta_{k,N,B,\mathcal{B}}$, which includes only B-related steps from $\alpha_{k,N,B,\mathcal{B}}$, is an N-solo execution admitted by B. □

Theorem 1. *For all n, k such that $1 < k < n$, there is no content-neutral and compositional broadcast abstraction equivalent to k-SA in the model $\mathcal{CAMP}_n[\emptyset]$.*

Proof. Assume the existence of a content-neutral and compositional broadcast abstraction B that is equivalent to k-SA in $\mathcal{CAMP}_n[\emptyset]$. Let $\mathcal{A}$ be an algorithm implementing k-SA in $\mathcal{CAMP}_n[B]$, and $\mathcal{B}$ be an algorithm implementing B in $\mathcal{CAMP}_n[k\text{-SA}]$. Remark that the model $\mathcal{CAMP}_n[\emptyset]$ is functionally identical to the model $\mathcal{CAMP}_{k+1}[\emptyset]$ when $n - k - 1$ processes crash at the start of execution. Hence, the two algorithms would still be correct in the model $\mathcal{CAMP}_{k+1}[\emptyset]$. By Lemma 9, there exists an integer $N > 0$ such that B does not admit any N-solo execution. Conversely, by Lemma 10, B admits an N-solo execution. This contradiction implies the non-existence of such a broadcast abstraction B. □

5 Conclusion

This paper investigates the computational equivalence of any broadcast abstraction to k-set agreement (k-SA) in message-passing systems. Following the introduction of two new symmetry properties defining admissible broadcast abstractions—*compositionality*, *content-neutrality*—we demonstrate that no broadcast abstraction, which is both content-neutral and compositional, is computationally equivalent to k-set agreement when $1 < k < n$. This paper highlights a crucial distinction in the application of k-set agreement in shared memory versus message-passing systems: for $1 < k < n$, k-SA is equivalent to a broadcast abstraction in shared memory (specifically, k-BO broadcast), but no such equivalence exists in message-passing systems.

As Lamport famously observed in [17], "The concept of time (...) is derived from the more fundamental concept of the order in which events occur." Therefore, at the abstraction level of message broadcasting in the system, each broadcast abstraction inherently provides a definition of time. On one end of the spectrum, broadcast abstractions that can be implemented solely through send and receive operations, such as Causal broadcast, offer processes a relativistic notion of time, defined by the "happened before" relation—a partial order. Conversely, at the other extreme where processes can utilize consensus, the set of broadcast events in Total Order broadcast forms an absolute timeline, known to all processes. Under this interpretation, k-SA represents a symmetric predicate on time —hence an elegant synchronization problem— when utilized within a shared-memory model. However, its inapplicability in message-passing systems questions the usefulness of k-SA in these contexts.

References

[1] Yehuda Afek, Eli Gafni, Sergio Rajsbaum, Michel Raynal, and Corentin Travers. The k-simultaneous consensus problem. *Distributed Comput.*, 22(3):185–195, 2010.

[2] Mustaque Ahamad, Gil Neiger, James E Burns, Prince Kohli, and Phillip W Hutto. Causal memory: Definitions, implementation, and programming. *Distributed Computing*, 9:37–49, 1995.

[3] Kenneth P Birman and Thomas A Joseph. Reliable communication in the presence of failures. *ACM Transactions on Computer Systems (TOCS)*, 5(1):47–76, 1987.

[4] François Bonnet and Michel Raynal. On the road to the weakest failure detector for k-set agreement in message-passing systems. *Theoretical Computer Science*, 412(33):4273–4284, 2011.

[5] Elizabeth Borowsky and Eli Gafni. Generalized FLP impossibility result for t-resilient asynchronous computations. In *Proceedings of the Twenty-Fifth Annual ACM Symposium on Theory of Computing, May 16-18, 1993, San Diego, CA, USA*, pages 91–100. ACM, 1993.

[6] Zohir Bouzid and Corentin Travers. Parallel consensus is harder than set agreement in message passing. In *IEEE 33rd International Conference on Distributed Computing Systems, ICDCS 2013, 8-11 July, 2013, Philadelphia, Pennsylvania, USA*, pages 611–620, 2013.

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

[8] Soma Chaudhuri. More choices allow more faults: Set consensus problems in totally asynchronous systems. *Information and Computation*, 105(1):132–158, 1993.

[9] Mathilde Déprés, Achour Mostéfaoui, Matthieu Perrin, and Michel Raynal. Brief announcement: The mbroadcast abstraction. In Rotem Oshman, Alexandre Nolin, Magnús M. Halldórsson, and Alkida Balliu, editors, *Proceedings of the 2023 ACM Symposium on Principles of Distributed Computing, PODC 2023, Orlando, FL, USA, June 19-23, 2023*, pages 282–285. ACM, 2023.

[10] Mathilde Déprés, Achour Mostéfaoui, Matthieu Perrin, and Michel Raynal. Send/receive patterns versus read/write patterns in crash-prone asynchronous distributed systems. In Rotem Oshman, editor, *37th International Symposium on Distributed Computing, DISC 2023, October 10-12, 2023, L'Aquila, Italy*, volume 281 of *LIPIcs*, pages 16:1–16:24. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2023.

[11] Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2), 1985.

[12] Eli Gafni and Petr Kuznetsov. The weakest failure detector for solving k-set agreement. In *Proceedings of the 28th ACM symposium on Principles of distributed computing*, pages 83–91, 2009.

[13] Vassos Hadzilacos and Sam Toueg. A modular approach to fault-tolerant broadcasts and related problems. Technical Report Tech Report 94-1425, Cornell University, 1994. Extended version of "Fault-Tolerant Broadcasts and Related Problems" in *Distributed systems, 2nd Edition*, Addison-Wesley/ACM, pp. 97-145 (1993.

[14] Maurice Herlihy and Nir Shavit. The asynchronous computability theorem for t-resilient tasks. In *Proceedings of the Twenty-Fifth Annual ACM Symposium on Theory of Computing, May 16-18, 1993, San Diego, CA, USA*, pages 111–120. ACM, 1993.

[15] Damien Imbs, Achour Mostéfaoui, Matthieu Perrin, and Michel Raynal. Which broadcast abstraction captures k-set agreement? In *31st International Symposium on Distributed Computing, DISC 2017, October 16-20, 2017, Vienna, Austria*, volume 91 of *LIPIcs*, pages 27:1–27:16, 2017.

[16] Damien Imbs, Achour Mostéfaoui, Matthieu Perrin, and Michel Raynal. Set-constrained delivery broadcast: A communication abstraction for read/write implementable distributed objects. *Theoretical Computer Science*, 886:49–68, 2021.

[17] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications*, 1978.

[18] Leslie Lamport. Generalized consensus and paxos. 2005.

[19] Achour Mostefaoui, Michel Raynal, and Julien Stainer. Chasing the weakest failure detector for k-set agreement in message-passing systems. In *2012 IEEE 11th International Symposium on Network Computing and Applications*, pages 44–51. IEEE, 2012.

[20] Fernando Pedone and André Schiper. Generic broadcast. In *Distributed Computing: 13th International Symposium, DISC'99 Bratislava, Slovak Republic September 27–29, 1999 Proceedings 13*, pages 94–106. Springer, 1999.

[21] David Powell. Group communication (introduction to the special section). *Commun. ACM*, 39(4):50–53, 1996.

[22] Michel Raynal. Set agreement. In *Encyclopedia of Algorithms*, pages 1956–1959. Springer, 2016.

[23] Michel Raynal. *Fault-tolerant message-passing distributed systems: an algorithmic approach*. Springer, 2018.

[24] Michel Raynal, André Schiper, and Sam Toueg. The causal ordering abstraction and a simple way to implement it. *Information processing letters*, 39(6):343–350, 1991.

[25] Michael E. Saks and Fotios Zaharoglou. Wait-free k-set agreement is impossible: the topology of public knowledge. In *Proceedings of the Twenty-Fifth Annual ACM Symposium on Theory of Computing, May 16-18, 1993, San Diego, CA, USA*, pages 101–110. ACM, 1993.

[26] Fred B. Schneider. The state machine approach: A tutorial. In *Proc. of Asilomar Workshop on Fault-Tolerant Distributed Computing*, volume 448 of *LNCS*, pages 18–41. Springer, 1986.