A probabilistic Dynamic Clock Set to capture message causality

Daniel Wilhelm, Luciana Arantes, Pierre Sens

A probabilistic Dynamic Clock Set to capture message causality

Daniel Wilhelm
Sorbonne Université, CNRS
Inria, LIP6
Paris, France
daniel.wilhelm@lip6.fr

Luciana Arantes
Sorbonne Université, CNRS
Inria, LIP6
Paris, France
luciana.arantes@lip6.fr

Pierre Sens
Sorbonne Université, CNRS
Inria, LIP6
Paris, France
pierre.sens@lip6.fr

***Abstract*—Several scalable constant size clocks were proposed in the literature for tracking causality of events in distributed systems with a high accuracy. Such clocks capture but do not characterize causality. Moreover, their size is fixed at initialization and cannot vary during execution. The efficiency of constant size clocks to causally order messages is negatively impacted by the message load, but such impact could be reduced by varying the size of the clock according to the message load. Therefore, this paper proposes a clock whose size can vary dynamically.**

Causal broadcast is a fundamental building block of many distributed and parallel applications, where processes collaborate to perform common tasks, such as high performance computing or providing a service to many users. To this end, processes require a broadcast primitive to share information that often requires to be causally ordered to be meaningful.

In this paper, we present the Dynamic Clock Set (DCS), which consists of a set of Probabilistic clocks. The size of the set can dynamically vary during execution. We also propose a causal broadcast algorithm using DCS clocks. Performance evaluation conducted on the OMNET++ simulator in scenarios where the message load varies shows the effectiveness of *DCS* clocks.

***Index Terms*—Logical clock, Causal Broadcast, Probabilistic clock, distributed algorithms**

I. Introduction

Distributed and parallel applications are composed of an increasing number of processes that cooperate by exchanging messages. Many application require that events, such as the sending and reception of messages or local events, are causally ordered as defined by Lamport's *happened before* relation [7].

Logical clocks have been used in the literature [7][5][8] to track the causality of events in distributed systems. Logical clocks are updated through rules, and are used to timestamp events and messages. Charon-Bost[4] proved that a vector with one entry per process in the system is necessary to characterize the causality of events. Thus, in a distributed system with N processes, causality can only be characterized with a logical clock in $\mathcal{O}(N)$. This induces scalability issues, especially in large systems, since the size of such clocks grows linearly with the number of processes.

Some works have proposed constant size clocks to circumvent these scalability issues, such as Probabilistic [9] or Plausible [15] clocks, whose size M is independent to N, the number of processes inside the system, with $M \ll N$. On the other hand, even though constant size clocks provide a high accuracy of causal message ordering, they do not characterize causality but do only capture it. Constant size clocks are adapted to systems in which ordering concurrent events only impacts performance and not correctness. [15].

Probabilistic clocks [9], proposed by Mostefaoui et al., have the best accuracy in causal message ordering among constant size clocks. A Probabilistic clock consists of a constant size clock V of size M, and a hash function f that associates one to several entries of V to each process. Authors showed that the capacity of a Probabilistic clock of size M to causally order messages decreases when the message load increases, because the number of concurrent messages to m increases. Moreover, the bigger the Probabilistic clock, the higher the resillience to a high message load. The size of a Probabilistic clock should, therefore, be chosen following the message load, i.e, it should dynamically vary with the message load, increasing (resp. decreasing) whenever the message load is above (resp. below) a given value. However, the size of Probabilistic clocks is fixed at initialization and cannot vary during execution.

Authors of [9] proposed a causal broadcast algorithm as usecase for Probabilistic clocks. Many group-based applications exchange information between processes through the *broadcast* and *deliver* primitives. The former sends a message to all members of the group, and the latter delivers broadcasted messages to the application. Some of these applications also require that broadcast messages are causally ordered, as defined by Lamport's *happened before* relationship [7]: if $broadcast(m_1)$ causally precedes $broadcast(m_2)$ then $deliver(m_1)$ causally precedes $delivery(m_2)$ at all processes of the group. Therefore, the delivery of a message m is postponed till all messages that causally precede m are delivered.

Causal broadcast, introduced by Birman et al. in [2], has been extensively investigated and innumerable applications use it, such as publish-subscriber systems, video conferencing, multimedia systems, online games, distributed replicated databases, discussion forums, social networks, distributed collaborative editing, and so on. Many implementations of causal broadcast exist in the literature.

This paper presents a logical clock, denoted Dynamic Clock Set (*DCS*), which consists of a set of Probabilistic clocks. The size of the set can dynamically vary during execution. We give the operations required to modify the size of the clock as well

as to compare two clocks. We also propose a causal broadcast algorithm using *DCS* clocks. We measured the performances of the causal broadcast algorithm using *DCS* clocks on the OMNeT++ simulator and compared it to Probabilistic clocks. Results show that *DCS* clocks have a higher accuracy than Probabilistic clocks in delivering messages in causal order, and depending on the message load, have also a lower memory overhead than probabilistic clocks.

The paper is organized as follows. Section II introduces background concepts. Section III, and Section IV respectively present the DCS clock with its operations, and a causal broadcast algorithm that uses DCS clocks. Section V presents performance results. Section VI discusses some related work. Finally, Section VII concludes the paper.

II. Background

Lamport introduced the *happened-before* relationship to order events in distributed systems[7]:

Happened-before relation: *Considering two events e_1 and e_2, e_1 causally precedes e_2 ($e_1 \rightarrow e_2$) iff: (a) e_1 and e_2 occur on the same process and e_1 precedes e_2 or (b) for a message m e_1=send(m) and e_2=deliver(m) or (c) there exists an event e_3 such that $e_1 \rightarrow e_3$ and $e_3 \rightarrow e_2$.*

Causal order is based on the *happened-before relationship*. It ensures that any two causally related messages are delivered to applications respecting that order.

Causal order: *Processes deliver messages while respecting the causal relation between them. For any message m and m', if m causally precedes m', denoted $m \rightarrow m'$, then all processes deliver m before m':*

$$send(m) \rightarrow send(m') \Rightarrow deliver(m) \rightarrow deliver(m').$$

Logical clocks

Leslie Lamport also introduced the concept of logical clocks to order a set of events E in a distributed system [7]. Logical clocks map each event to a scalar timestamp and *capture* the causality between events: $\forall a, b \in E : a \rightarrow b \Rightarrow L(a) < L(b)$ (*weak clock condition*). However, Lamport Clocks do not characterize the causal order between events: $l(a) < L(b) \not\Rightarrow a \rightarrow b$. To characterize causality, vector clocks were independently proposed by Fidge [5] and Mattern [8]. Vector clocks map each event to a scalar vector of N entries, where N is the number of nodes. Vector clocks *characterize* causality: $\forall a, b \in E : a \rightarrow b \Leftrightarrow V(a) < V(b)$ and $a||b \Leftrightarrow V(a)||V(b)$ (*strong clock condition*).

In [4], Charron-Bost proved that causality can only be characterized by timestamps with O(N) entries. Therefore, vector clocks do not scale. Moreover, they require to know beforehand the exact number of nodes involved in the application, which restrict their use in many applications.

Probabilistic clocks

Some existing logical clocks such as Probabilistic [9], Plausible [15] or Bloom [13] clocks do not grow in size with the number of processes and use vectors of constant size M, where M is much smaller than the number of processes in the system. Thus, they do scale, but do not exactly characterize causality, and aim therefore to have the best accuracy of detecting the causality of events.

The clock presented in this article is built with Probabilistic clocks, which have the best performances among constant size clocks[9]. In a system using Probabilistic clocks, each process p_i keeps a local clock V_i of size M, whose entries are initialized to 0. A hash function $f(p_i)$ returns the set of k clock entries assigned to p_i, with $1 \leq k \leq M$, i.e. one to several entries of the Probabilistic clock are respectively associated to each process. Process p_i uses the following two rules $R1$ and $R2$ to update its local Probabilistic clock:

- $R1$: Before executing an event, p_i updates its local Probabilistic clock as follows:
$$\forall x \in f(p_i), V_i[x] = V_i[x] + 1$$
- $R2$: Each message m carries with it the vector clock of its sender process at sending time. On the receipt of a message m, process p_i :
  - Updates its local probabilistic clock as follows: $\forall x, V_i[x] = max(V_i[x], m.V[x])$
  - Executes $R1$, Deliver(m)

Comparison of two Probabilistic clocks. A process compares two Probabilistic clocks V_1 and V_2 as follows: $V_1 < V_2$, iff $\forall x, 1 \leq x \leq M, V_1[x] \leq V_2[x] \wedge \exists i, V_1[i] < V_2[x]$

Note that Probabilistic clocks ensure that:

$$send(m_1) \rightarrow send(m_2) \Rightarrow m_1.V < m_2.V. \quad .$$

Authors of [9] showed that the probability that a process delivers a message m out of causal order when using Probabilistic clocks to causally order messages is equal to $(1 - (1 - \frac{1}{M})^{X*k})^k$, where M is the size of the clock attached on m, k is the number of entries associated to each process, and X the number of messages concurrent to m. Two observations can be made out of this equation. First, increasing the size of the clock attached on m decreases the probability that a process delivers m out of causal order. Second, increasing the number of concurrent messages increases the probability that m is delivered out of causal order. This is because a concurrent message might increment the same clock entries as dependencies of m. Therefore, the higher the number of concurrent messages to m that p_i delivers, the higher the probability that such deliveries increment the same clock entries as dependencies of m, and p_i will then wrongly conclude that it has delivered all dependencies of m upon receiving m, thus delivering m out of causal order.

Causal broadcast

Causal broadcast: *Each process delivers each message exactly once, and processes deliver messages while respecting the causal relation between them. Formally :*
broadcast(m)$\rightarrow$ broadcast(m') $\Rightarrow$ deliver(m)$\rightarrow$ deliver(m')

To ensure causal broadcast, a process that receives a message m should postpone the delivery of m until it has delivered all messages that causally precede m. Causal broadcast should ensure the following properties:

- *Validity*: if a process delivers a message m from a process p, p previously broadcasted m.
- *Integrity*: a process delivers a message m at most once.
- *Causal Delivery*: if a process broadcasts a message m' after it has delivered another message m, then no process in the system will deliver m after m'.
- *Termination*: A message broadcasted by a correct process is delivered by all correct processes.

Algorithm 1: Broadcast at process p_i

Broadcast of message m

1: $\forall x \in f(i), V_i[x] = V_i[x] + 1$
2: $m.V = V_i$
3: broadcast(m)

Upon reception of message m **from** p_j

3: waitUntil($(\forall x \in f(j), V_i[x] \geq m.V[x] - 1) \wedge \forall k \notin f(j), V_i[k] \geq m.V[k]$)
4: $\forall x \in f(j), V_i[x] = V_i[x] + 1$
5: deliver(m)

Algorithm 1 describes the causal broadcast algorithm using Probabilistic clocks presented in [9].

Before broadcasting a message m, p_i increments the entries $f(p_i)$ of its local vector clock V_i and then broadcasts (V_i, m). Upon reception of m, process p_j buffers m until the two following conditions are satisfied: (1) $\forall x \in f(p_i), V_j[x] \geq m.V[x] - 1$ and (2) $\forall x \notin f(p_i), V_j[x] \geq m.V[x]$. p_j then delivers m and increments the entries $k \in f(p_i)$ of V_j. The formula, $ln(2) * \frac{R}{X}$ (where R is the size of V and X is the number of messages that are, on average, concurrent to m) is given by the authors as the optimal number of clock entries that processes should increment when broadcasting a message.

Figure 1 shows the broadcast of two messages m and m'. The entries assigned to p_1, p_2, and p_3 are $f(p_1) = \{0, 1\}$, $f(p_2) = \{0, 2\}$, and $f(p_3) = \{1, 2\}$ respectively. p_i broadcasts (m, V_1) after incrementing the entries $f(p_1)$ of its local vector clock V_1. Process p_2 receives and delivers m. Then, it broadcasts (m', V_2) after incrementing the entries $f(p_2)$ of V_2. Thus, $m \rightarrow m'$. When p_3 receives m', the delivery conditions of m' are not yet satisfied since $m'.V[0] = 2 > V_3[0] = 0$. Consequently, p_3 buffers m'. Upon the reception of m, p_3 delivers m and increments the entries $f(p_1)$ of V_3. It then also delivers m' and increments the entries $f(p_2)$, since the delivery conditions of m' have been satisfied.

Probabilistic clocks capture causality but do not characterize it. Hence, processes might deliver messages out of causal order. For example, assume that in Figure 1 process p_3 delivers messages concurrent to m before receiving m', such that the delivery of those messages increment $V_3[0]$ and $V_3[2]$. p_3 will then deliver m' out of causal order, because the delivery conditions of m' will be satisfied at p_3 upon reception.

III. Dynamic Clock Set

A Dynamic Clock Set (DCS) is composed of a set of Probabilistic clocks, denoted components, which all have the same number of entries (M). Figure 2 gives a representation of a *DCS* clock. The number of components of a *DCS* clock can dynamically vary during execution.

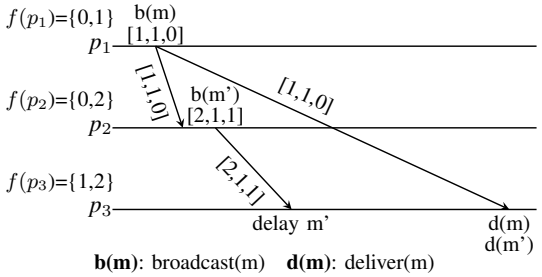


Fig. 1: Causal broadcast using probabilistic clocks

A. Component

A component C_k is uniquely identified by its index k. C_k is either *active* or *inactive*. A *DCS* clock D is composed of one or several *active* components, followed by no or several *inactive* components, as shown in Figure 2, i.e. C_0 is always *active*, followed by no or several *active* components C_i with $1 < i < |D|$, followed by no or several *inactive* components C_j with $i < j \leq |D|$. A process only attaches the *active* components of its *DCS* clock on messages.

A process increments one or several *active* components of its *DCS* clock to keep track of events. The set $S_{incr,i}$ contains the index of the components that process p_i increments to track events. p_i applies the hash function $f(p_i)$ to all components whose index is contained in $S_{incr,i}$.

B. Update of a DCS clock

Process p_i uses the following two rules $R1$ and $R2$ to update its local *DCS* clock:

- $R1$: Before executing an event, it updates its local clock: $\forall x \in f(p_i), \forall k \in S_{incr}, V_i.C_k[x] = V_i.C_k[x] + d \ (d > 0)$
- $R2$: Each message m carries with it the vector clock of its sender process at sending time. On the receipt of a message (m, V_m), process p_i:
  - Updates its local clock as follows:
    - **(1)** If $|D_i| < |D|$, p_i calls $Add()$, defined below, till $|D_i| = |D|$.
    - **(2)** $\forall k \in [1, |D|], \forall x \in [1, M], D_i.C_k[x] = max(D_i.C_k[x], D.C_k[x])$
  - Executes $R1$, Deliver(m)

C. Comparison of two DCS clocks

Each component of a *DCS* clock is an independent Probabilistic clock. Hence, the comparison operator $<$ of *DCS* clocks is based on the comparison operator of Probabilistic clocks. As a reminder, the comparison operator $<$ of two Probabilistic clocks C_1 and C_2 is defined as follows:

$C_1 < C_2$ iff $\forall x, C_1[x] \leq C_2[x] \wedge \exists k, C_1[k] < C_2[k]$

For two *DCS* clocks D_1 and D_2, we have $D_1 < D_2$ provided that :

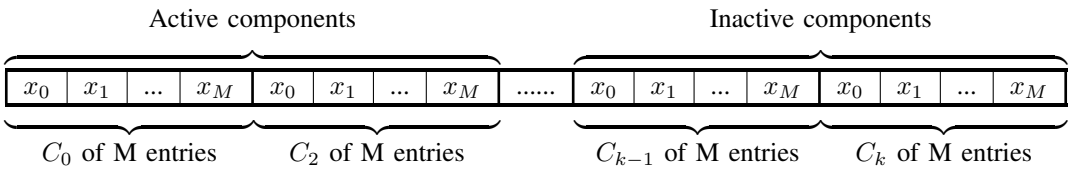


Fig. 2: Representation of a Dynamic Clock Set

(1) $|D_1| \leq |D_2|$

(2) Each component C_k of D_1 is smaller or equal to the corresponding component C_k of D_2, and at least one component C_j of D_1 is strictly smaller than the component C_j of D_2: $\forall k \in [1, |D_1|], D_1.C_k \leq D_2.C_k \wedge \exists C_j, D_1.C_j < D_2.C_j$.

Two causally related messages m_1 and m_2 with respective *DCS* clocks $m_1.D$ and $m_2.D$ verify the following condition:

$$send(m_1) \rightarrow send(m_2) \Rightarrow m_1.D < m_2.D.$$

Note that two messages m_1 and m_2 whose *DCS* clock comparison does not satisfy the above two conditions are said to be concurrent, denoted as $m_1||m_2$. Formally:

$$m_1.D \not\leq m_2.D \wedge m_2.D \not\leq m_1.D \Rightarrow m_1||m_2$$

Theorem 1. *For any two messages m and m' of respective DCS clocks $m.D$ and $m'.D$, if $m \rightarrow m'$ then we have : $send(m) \rightarrow send(m') \Rightarrow m.D < m'.D$*

Proof. Consider that process p_i of *DCS* clock D_i sends a message m of causal dependencies Dep_m. We prove that $\forall m' \in Dep_m, m'.D < m.D$, by showing that when p_i sends m, we have $\forall m' \in Dep_m, m'.D < D_i$.

A process p_j updates its DCS clock D_j when delivering a message m: p_j adds components to D_j in order to ensure that D_j has at least as many components than $m.D$. Therefore, we have $|D_j| > |m.D|$. Second p_j updates D_j : $\forall x, \forall k, D_j.C_k[x] = max(D_j.C_k[x], m.D.C_k[x])$. Therefore, we have $\forall x, \forall k, D_j.C_k[x] \geq m.D.C_k[x]$. Therefore, after p_j delivered m, we have $m.D \leq D_j$.

For all messages $m' \in Dep_m$, either p_i delivered m', or another process p_j delivered m' and broadcasted a message m" such that $m' \rightarrow m$" $\rightarrow m$ and that p_i delivered m". If p_i delivered m', then $m'.D \leq D_i$ as showed above. Otherwise, (1) a process p_j has delivered m' and therefore $m'.D \leq m$".D (2) p_i has delivered m" and therefore m".$D \leq D_i$. Therefore, $m'.D \leq D_i$. Hence, we have $\forall m' \in Dep_m, m'.D \leq D_i$.

When p_i sends m, it first updates its DCS clock by incrementing at least one entry x of at least one component C_k before appending D_i on m. Therefore, $\exists k, \exists x, \forall m' \in Dep_m, m.D.C_k[x] > m'.D.C_k[x]$.

Therefore, $\forall m' \in Dep_m, m'.D < m.D$.

□

D. Operations required for DCS clock dynamicity

DCS clocks are dynamically modifiable during execution by adding and removing components to them. The size of *DCS* clocks can particularly be adapted to the system's message load. In fact, as shown in Section II, the efficiency of Probabilistic clocks to causally order messages decreases when the message load increases. Nevertheless, increasing the size of a Probabilistic clock increases its accuracy to causally order messages. A desired accuracy of causal message ordering can therefore be reached by varying the size of *DCS* clocks according to the system's message load.

Process p_i modifies its local *DCS* clock D_i through one of the following operations:

- **Activate():** Activates the component of D_i with the lowest index among D_i's inactive components.
- **Deactivate():** Deactivates the component of D_i with the highest index among D_i's active components.
- **Add():** Creates a new component, sets its entries to 0, and adds the component at the end D_i.
- **Remove():** Removes the component of D_i with the highest index.

***Activate()**:* Process p_i calls the operation *Activate()* to activate the inactive component of D_i with the lowest index, provided that D_i has at least one inactive component. The call to *Activate()* immediately returns $false$ if D_i has no inactive component. Otherwise the inactive component of D_i with the lowest index is activated.

***Deactivate()**:* Process p_i calls the operation *Deactivate()* to deactivate the active component of D_i with the highest index. A *DCS* clock must have at least one active component. Thus, the call to *Deactivate()* immediately returns *false* if D_i has only one active component. Otherwise the active component of D_i with the highest index is deactivated.

For example, let's consider a process p_i whose DCS clock D has four components: $D = \{C_0, C_1, C_2, C_3\}$. If p_i wants to deactivate components, then it will deactive them in decreasing order, i.e, first C_3, then C_2, and finally C_1. On the other hand, if p_i wants to re-activate them, then it will first activate C_1, then C_2, and finally C_3.

A process keeps deactivated components C_d locally, because it might receive a message whose *DCS* clockcontains C_d, and it will then require the local C_d to ensure that the delivery conditions of the message's C_d are satisfied.

***Add()**:* A process decides locally to add a new component to its *DCS* clock, i.e., without communicating with other processes. When p_i calls the *Add()* operation, it first creates a new component C_k in active state, sets its entries to 0, and appends C_k to the end of D_i. Therefore, C_k will be the component of D_i with the highest index.

After adding a component to D_i, p_i also activates all components of D_i, since the inactive components of a *DCS* clock have a strictly higher index than the active components. Therefore, adding a component at the end of D_i implicates that all components of D_i must be active.

Remove(): Process p_i calls *Remove()* to remove the component of D_i with the highest index. It returns *false* if D_i has only one component, since by definition a *DCS* clock must have at least one component.

IV. Causal broadcast algorithm using *DCS* clocks

This section presents an algorithm that implements causal broadcast with *DCS* clocks.

A. Model

We consider a set $\Pi = \{p_1, p_2, \ldots, p_N\}$ of N processes. Processes communicate through message passing. Each pair of processes is connected by a reliable communication channel. Local events induce no interactions with other processes and are therefore omitted. Each application message is broadcasted to all processes of the system.

B. Definition of the algorithm

Algorithm 2 describes the *DCS* clock-based causal broadcast algorithm. Each process p_i keeps:

- D_i: its local *DCS* clock.
- S_{incr}: a set containing the indexes of the components of D_i that p_i increments when broadcasting a message.

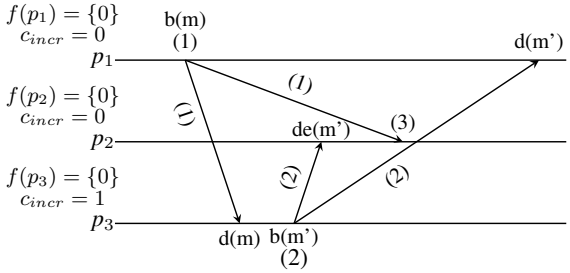
Broadcast(m): Process p_i first updates its *DCS* clock by executing the Rule $R1$ given in Section III-B: $\forall x \in f(p_i), \forall k \in S_{incr}, D_i.C_k[x] = D_i.C_k[x] + d$ $(d > 0)$. Then, p_i broadcasts m with S_{incr} and the *active* components of D_i.

Reception of a message m: Process p_i first calls *checkExpand()* which has three functions:

- First, p_i must have at least as many components than D_m, the *DCS* clock of m, to ensure that D_i satisfies all delivery conditions of D_m. Thus, *checkExpand()* calls *Add()* till $|D_i| = |D_m|$.
- Second, p_i might have deactivated components contained in D_m. p_i reactivates such components C_d if $\exists x, D_i.C_d[x] < D_m.C_d[x]$, because C_d then contains causal information not acknowedged by all processes. Note that p_i then also activates the components $C_i, i \leq d$ to satisfy the condition that active components always have lower indexes than inactive ones.
- Third, if p_i activated some components in *checkExpand()*, then it affects itself randomly to new components among the active ones of D_i, in order to ensure that all components are on average incremented by the same number of processes.

After calling *checkExpand()*, p_i waits till the following delivery conditions of D_m are satisfied locally:

- For each component $C_{k \notin S_{incr,m}}$: The broadcast of m did not increment the entries of C_k. Hence, the entries of $D_i.C_k$ should be equal or greater than those of $D_m.C_k$:
  waitUntil($\forall C_{k \notin S_{incr,m}}, D_m.C_k[x] \leq D_i.C_k[x]$)



(1): [{[1],[0]},0] (2): [{[1],[1]},1] (3): d(m), d(m')
b(m): broadcast(m) **d(m)**: deliver(m) **de(m)**:cache(m)

Fig. 3: Causal broadcast using DCS clocks

- For each component $C_{k \in S_{incr,m}}$: The broadcast of m did increment the entries $f(p_j)$ of C_k. Hence, the entries $x \in f(p_j)$ of $D_i.C_k$ should be equal or greater than those of $D_m.C_k$ minus one, and the entries $x \notin f(p_j)$ of $D_i.C_k$ should be equal or greater than those of $D_m.C_k$:
  waitUntil($\forall C_{k \notin S_{incr,m}}, \forall x \in f(p_j), D_m.C_k[x] - 1 \leq D_i.C_k[x] \wedge \forall x \notin f(p_j), D_m.C_k[x] \leq D_i.C_k[x]$)

Note that D_i might have more components than D_m. It is then sufficient to ensure that D_i satisfies the delivery conditions of the components of D_m.

Process p_i executes Rule $R1$ given in Section III-B once D_i satisfies the delivery conditions $of D_m$:

$\forall x \in f(p_i), \forall k \in S_{incr}, D_i.C_k[x] = D_i.C_k[x] + 1$

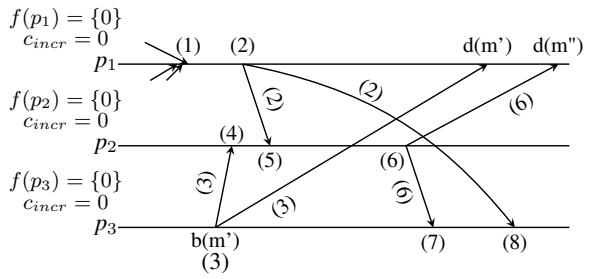
Finally, p_i delivers m.

Algorithm 2: Broadcast at process p_i

Broadcast of message m
1: $\forall x \in f(p_i), \forall k \in S_{incr,i}, D_i.C_k[x] = D_i.C_k[x] + 1$
2: broadcast(m,D_i,$S_{incr,i}$)

Upon reception of message $(m, D_m, S_{incr,m})$ **from** p_j
3: checkExpand()
4: waitUntil($\forall C_{k \notin S_{incr,m}}, D_m.C_k[x] \leq D_i.C_k[x]$)
5: waitUntil($\forall C_{k \notin S_{incr,m}}, \forall x \in f(p_j), D_m.C_k[x] - 1 \leq D_i.C_k[x] \wedge \forall x \notin f(p_j), D_m.C_k[x] \leq D_i.C_k[x]$)
6: $\forall x \in f(p_j), \forall k \in S_{incr,m}, D_i.C_k[x] = D_i.C_k[x] + 1$
7: deliver(m)

Figure 3 shows the broadcast of two messages. The system is composed of three processes p_1, p_2, and p_3. Each process maintains two components with each component having one entry. Processes p_1 and p_2 have $S_{incr} = \{0\}$, i.e. they increment component C_0 when broadcasting a message while p_3 increments component C_1 when broadcasting a message. In the scenario, p_1 first broadcasts m whose causal information is represented by (1). Upon reception of m, p_3 delivers it since its delivery conditions are satisfied. Then, p_3 broadcasts m', i.e. $m \rightarrow m'$. The causal information of m' is represented in (2). Process p_2 receives m' before m. Thus, it postpones the delivery of m since its delivery conditions are not satis-



(1): expand() (2): [{[0],[1]},1] (3): [{[1],[0]},1]
(4): d(m') (5): expand(), d(m), D_2=[{[1],[1]},0]
(6): b(m"), m"=[{[2],[1]},0] (7): delay(m")
(8): d(m),d(m"),D_3=[{[2],[1]},1]}

Fig. 4: Causal broadcast using *DCS* clocks

fied, because $p_2.D.C_0[0] < m'.D.C_0[0]$. When it eventually receives m at (3), it delivers m and then m'.

C. Expand of the local DCS clock

DCS clocks are expanded in order to reduce the number of processes that increment the same entries when broadcasting a message. Process p_i decides to increment its local *DCS* clock D_i without communicating with other processes. p_i decides to expand D_i spontaneously, when observing for example a high message load. To expand its *DCS* clock, p_i:

- First, calls *Activate()* described in Section III-D, which activates an inactive component of D_i if one is available, and which returns *false* otherwise.
- Second, calls *Add()* described in Section III-D if *Activate()* returns false, i.e. p_i adds a new component to D_i if D_i has no inactive component.
- Third, p_i affects itself randomly to new components among the active ones of D_i, in order to ensure that components are on average incremented by the same number of processes.

Figure 4 shows a scenario where processes expand their *DCS* clock. The system is composed of three processes. Initially, each process maintains a *DCS* clock of one component, namely C_0, and each process increments C_0 when broadcasting a message.

At (1), p_1 detects decides to expand its *DCS* clock D_1 (e.g., detection of high message load) by adding a new component to D_1, since D_1 has no deactivated component. Moreover, it re-assigns itself to a random component of D_1 which gives $S_{incr,1} = \{random()\%2\} = \{1\}$. At (2) an (3) p_1 and p_3 broadcasts a message m with *DCS* clock $\{\{[0],[1]\}\}$ and $\{\{[1]\}\}$ respectively. Process p_2 first receives m' at (4) and delivers it. At (5), p_2 receives m, adds a component to its *DCS* clock, since $D_2 < |m.D|$, and sets $S_{incr,2} = \{random()\%2\} = \{0\}$, then it delivers m.

At (6), p_2 broadcasts message m" with *DCS* clock $\{\{[2],[1]\},0\}$. p_3 receives m" at (7), expands its *DCS* clock by adding a new component since $|D_3| < |m".D|$ and sets $S_{incr,3} = \{random()\%2\} = \{1\}$. It postpones the delivery of m" since $m".D.C_0[0] = 1$ and $D_3.C_0[0] = 0$. At (8), p_3 receives m, and delivers both m then m'.

D. Deactivate DCS clock components

This section describes how a process can deactivate components of its *DCS* clock without loss of causal information, i.e. how a process can deactivates components of its *DCS* clock while ensuring that the deactivated components do not contain causal information that is till useful to some other process.

Processes should deactivate components whenever possible, as for example when the message load decreases, because deactivated components are not sent with messages and are only kept locally. Consequently, deactivating components reduces the causal information carried by messages.

A process p_i that wants to deactivate a component of its *DCS* clock D_i deactivates the component of D_i with the highest index among the active components, i.e. if k components of D_i are active, then p_i first deactivates C_{k-1}, then C_{k-2}, etc up to C_1. C_0 cannot be deactivated since a *DCS* clock must have at least one active component.

Component C_k of D_i provides causal information to at least one other process as long as C_k's delivery conditions are not satisfied by all processes, i.e. as long as $\exists p_j, \exists x, p_j.D_j.C_k[x] < D_i.C_k[x]$.

Moreover, three other conditions are required to ensure that C_k will not be reactivated immediately after its deactivation:

- There exist no process p_j of *DCS* clock D_j such that $\exists x, D_j.C_k[x] > D_i.C_k[x]$, because p_i would then reactivate C_k as soon as it receives a message from p_j.
- No process currently increments C_k.
- No process currently delays the delivery of a message m with $k \in m.S_{incr}$, because the delivery of m might violate the first condition above.

*1) **Deactivation round**:* A process verifies the satisfaction of the above conditions by exchanging messages with the other processes of the system through a message exchange consisting of two phases. *Phase 1* sends the component C_k to deactivate to all processes, which reply with a positive or negative acknowledgement. *Phase 2* confirms or not the deactivation of C_k to all processes.

Phase 1. Process p_i starts $Phase$ 1 by broadcasting a *Deactivate* message containing C_k and C_k's index k. The other processes reply with an *AckDeactivate* message containing a positive or negative acknowlegment of C_k, depending whether they locally satisfy the deactivation conditions of C_k or not. Moreover, they block the dynamicity operations of their *DCS* clock till the end of the *Deactivation round*, i.e. till they receive the *DecisionDeactivate* message from p_i.

Phase 2. p_i broadcasts a *DecisionDeactivate* message once it received the *AckDeactivate* message from all processes. The *DecisionDeactivate* message contains C_k's index k and a boolean that confirms or not the success of the round, i.e. the deactivation or not of C_k. p_i sets this boolean to *true* if all processes positively acknowledged the deactivation of C_k and *false* otherwise. Upon reception of the *DecisionDeactivate* message, a process unblocks the dynamicity operations of its

DCS clock, and deactivates C_k provided that the boolean is set to *true*.

Any process can start a *Deactivation round*. The process can be chosen probabilistically or, for example, in a predefined way based on the process identifier. Several processes could start a *Deactivation round* for the same component or for different components simultaneously, but this should be avoided since acknowledging the deactivation of the same components several times is useless.

2) ***Complexity analysis****:* A *Deactivation round* has a message complexity in $\mathcal{O}(N)$: N $Deactivate$ messages, N $AckDeactivate$ messages and N $DecisionDeactivate$ messages. A *Deactivatation round* has a message memory complexity in $\mathcal{O}(M)$: $Deactivate$ messages contain a component of size M and one integer, while $AckDeactivate$ and $DecisionDeactivate$ messages contain some integer and boolean values and have therefore a space complexity in $\mathcal{O}(1)$.

E. Remove DCS clock components

This section describes how a process can remove components from its *DCS* clock. Processes keep inactive components of their *DCS* clock locally, and should eventually remove them in order to free memory space.

Processes must coordinate the removal of inactive components with each other, because if one process removes a component C_k from its *DCS* clock, then all processes should remove C_k from their *DCS* clock. In fact, assume that one process p_i removes C_k from its *DCS* clock, and that another process p_j then broadcasts a message m without having removed C_k from its *DCS* clock. The *DCS* clock appended on m will then contain C_k. Upon reception of m, p_i will have lost the causal information of C_k since it deleted C_k. Thus, a process should only delete a component C_k once it is ensured that it will receive no other message containing this C_k. Before removing C_k, p_i should therefore verify that :

(1): C_k is inactive at all processes of the system.

(2): It will receive no message containing C_k after removing C_k from its *DCS* clock.

1) ***Remove round****:* A process verifies the satisfaction of the above conditions by exchanging messages with the other processes of the system through a message exchange consisting of two phases. $Phase$ 1 synchronizes processes to ensure that they all have delivered the messages containing the component C_k to be removed and that they will broadcast no new message containing C_k, while $Phase$ 2 propagates the deletion decision of C_k, which depends on the satisfaction by all processes of the two above conditions.

Phase 1. Process p_i broadcasts a *Remove* message containing a vector clock with a tuple $< p_k, seq_k >$ for each process p_k that broadcasted a message since the last *Remove* round, with seq_k corresponding to the number of messages p_k broadcasted since the last *Remove* round.

Upon reception of the *Remove* message, process p_j blocks the dynamicity operations of its *DCS* clock. Moreover, it verifies that C_k is locally inactive and that for each tuple $< p_k, seq_k >$ it delivered seq_k messages from p_k since the last *Remove* round. Finally, p_j replies with an *AckRemove* message containing k, the index of C_k, as well as a boolean set to *true* if both conditions are satisfied, and *false* otherwise.

Phase 2. p_i sends a *DecisionRemove* message once it received the *AckRemove* message from all processes. The *DecisionRemove* message contains C_k's index k and a boolean that confirms or not the success of the round, i.e. the removal or not of C_k. p_i sets this boolean to *true* if all processes positively acknowledged the removal of C_k and *false* otherwise. Upon reception of the *DecisionRemove* message, a process unblocks the dynamicity operations of its *DCS* clock, and removes C_k provided that the boolean is set to *true*.

2) ***Complexity analysis****:* A *Remove round* has a message complexity in $\mathcal{O}(N)$: N $Remove$ messages, N $AckRemove$ messages and N $DecisionRemove$ messages. A *Remove round* has a message memory complexity in $\mathcal{O}(N)$: $Remove$ messages contain a vector with up to N entries, while $RepRemove$ and $Remove$ messages contain some integer and boolean values.

Remove rounds should not be executed often because of the message memory complexity in $\mathcal{O}(N)$. Nevertheless, several components can be acknowledged at once. Moreover, a *DCS* clock is usually composed of a small number of components, and keeping them locally without sending them with messages only represents a small local memory overhead.

F. Termination proof of DCS clocks

This section gives the proof of termination of the causal broadcast algorithm using *DCS* clocks. The proof is divided in two parts. *Theorem 2* proves the termination property for static *DCS* clocks. *Theorem 3* proves that the termination property holds when adding and removing components to *DCS* clocks.

Theorem 2. *A well formed message broadcasted with an algorithm using a static DCS clock to causally order messages is eventually delivered by all processes.*

Proof. We prove it by induction. We assume that each process p_i has a *DCS* clock D_i of $l \geq 1$ components.

H_0: Messages generated on the initial state are eventually delivered by all processes.

Processes initialize the entries of components at 0. Hence, a message m generated by p_i in the initial state carries a *DCS* clock with $\forall x \in f(p_i), \forall k \in S_{incr,i}, m.C_k[x] = 1$ and for all the other component entries values equal to 0. Since all processes initialize the entries of components to 0, their *DCS* clock satisfies both delivery conditions upon reception of m. Thus, messages generated in the initial state are eventually delivered by all processes.

H_1: We assume a set of messages M that are eventually delivered by all processes. We show that any message generated by a process after it delivered the messages of M will be eventually delivered by all processes.

Let's consider a message m of *DCS* clock D_m generated by a process p_i after it has delivered all messages of M.

By hypothesis, all processes eventually deliver the messages of M and increment their *DCS* clock accordingly.

Moreover, p_i only increments the entries $x \in f(p_i)$ of the components $C_k, k \in S_{incr,i}$ when broadcasting m. Hence the entries of the *DCS* clock appended on m and the *DCS* clock of processes after they delivered m only differs by one $\forall x \in f(p_i), \forall k \in S_{incr,i}$. Therefore, processes will satisfy the delivery conditions of m once they delivered the messages of M, which they do by definition. Therefore, processes eventually deliver m.

Any message generated after a set of eventually delivered messages will eventually be delivered by all processes (*see H1*). Since all messages generated on the initial state are eventually delivered by all processes (*see H0*), we conclude that any message is eventually delivered by all processes.

□

Lemma 1. *The termination property of the causal broadcast algorithm using DCS clocks holds when processes add or activate components of their DCS clock.*

Proof. Adding a new component to a *DCS* clock is equivalent to activate an inactive component not yet contained in the *DCS* clock of any other process. Therefore, it is sufficient to show that the termination property holds when a process p_i activates an inactive component C_k of its *DCS* clock D_i. We consider that p_i broadcasts a message m after activating C_k.

p_i reaffects itself to new components when activating C_k, and stores the index of those components in $S_{incr,i}$. Any process p_j that receives m first adds and activates C_k to its *DCS* clock D_j if D_j has no such component yet. Moreover, p_j knows which components p_i incremented when broadcasting m, since m carries $S_{incr,i}$. Therefore, p_j adds and activates C_k to its *DCS* clock, and by using S_{incr} it will also increment the entries of the right components when delivering m. □

Lemma 2. *The termination property of the causal broadcast algorithm using DCS clocks holds when processes remove components of their DCS clock.*

Proof. The *Remove round* ensures that processes only remove a component C_k provided that all processes have delivered all messages whose *DCS* clock contains C_k. Moreover, the *Remove round* ensures that no new message containing C_k will be broadcasted (even though a process might add a new component of index k after C_k was removed). Therefore, after a successfull *Remove round*, C_k will not be used in any delivery of message, and it's deletion will therefore impact no message delivery. □

Lemma 3. *The termination property of the causal broadcast algorithm using DCS clocks holds when processes deactivate components of their DCS clock.*

Proof. Consider that process p_i deactivates component C_k. p_i does not increment deactivated components, i.e. C_k. Hence, C_k will contain no new causal information. Moreover, deactivated components are not sent with messages, but are kept locally by processes. Therefore, deactivating components only removes delivery conditions of a message without loosing causal information. The deactivation of components does therefore impact no message delivery. □

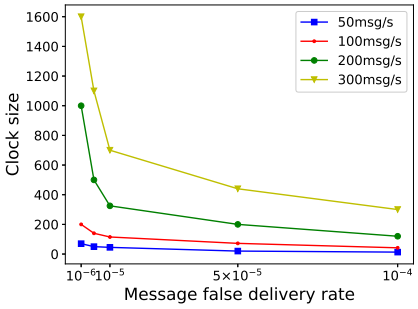


Fig. 5: Clock size following the message load to achieve a given causal ordering accuracy

Theorem 3. *A well formed message is eventually delivered by all processes when broadcasting messages with the causal broadcast algorithm using DCS clocks.*

Proof. Following *Theorem 2*, processes eventually deliver messages when using a causal broadcast algorithm using static DCS clocks to causally order messages. Following *Lemma 1*, *Lemma 2* and *Lemma 3*, the termination property holds when the dynamicity operations of *DCS* clock are considered (*Add(),Activate(), Remove(), Deactivate()*). The termination property of *DCS* clocks holds therefore also when considering *DCS* clock dynamicity.

□

V. Experimental results

Experiments were carried out on the OMNET++ simulator. Processes generate messages according to a Poisson distribution. The propagation time of messages follows a normal distribution $N(100, 20)$. An independent controller module detects out of causal order deliveries. The first experiments aim to determine the required *DCS* clock size to achieve a given accuracy of causal message ordering, depending on the system's message load. The second set of experiments compare *DCS* clocks to Probabilistic clocks for two different message load patterns. The third experiment evaluates the load balancing ability of *DCS* clocks.

A. Clock size following the message load

The size of *DCS* clocks should depend on the message load as well as on the accepted probability that a message is delivered out of causal order. The purpose of the first experiment is to determine the required size of the *DCS* clock following the message load and the accepted probability that a message is delivered out of causal order.

The system consists of 2000 processes that broadcast messages on average with the same frequency, determined such that the system has a given message load. The hash function returns two entries, i.e., processes increment two entries when broadcasting a message. Figure 5 presents the required size

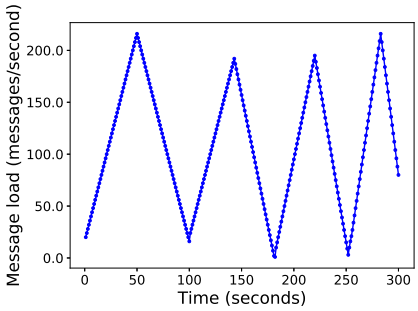


Fig. 6: Message load following intervals

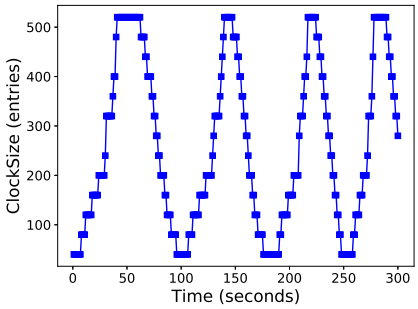


Fig. 7: DCS clock size attached on messages

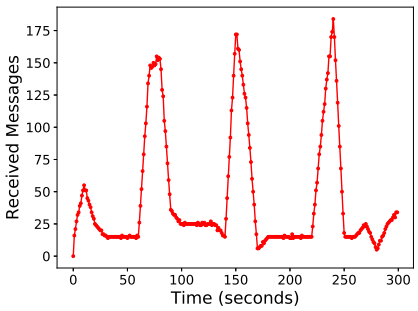


Fig. 8: Message load following random message load pattern

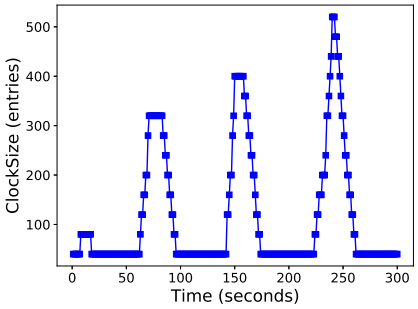


Fig. 9: DCS clock size attached on messages in the random message load pattern

of Probabilistic clocks, depending on the message load, in order for a message to be delivered out of causal order with a probability of $10^{-4}, 5.10^{-5}, 10^{-5}, 5.10^{-6}$ and 10^{-6}.

Results show that a higher causal ordering accuracy requires bigger clocks, i.e. to have a probability to deliver a message out of causal order of 10^{-5} requires a bigger clock than a probability of only 10^{-4}.

Moreover, we also observe that the clock size required to causally order messages increases fast with the message load. Indeed, it increases faster than linearly. We can intuitively explain such a behavior by analyzing the formula presented in [9] and described in Section II that gives the probability that a message is delivered out of causal order: $(1-(1-\frac{1}{M})^{X*k})^k$, where X corresponds to the number of concurrent messages, which is directly affected by message load, and M corresponds to the clock size. The formula confirms that the increase in message load (resp., clock size) has an exponential (resp., division) impact in the formula result, thus explaining why the clock size increases faster than linearly. This observation even increases the importance of adjusting dynamically the size of the clock, and not choosing a size following the highest supposed message load, since the size of the clock will be highly overestimated.

B. Behavior following different message load patterns

The second set of experiments compare *DCS* clocks to Probabilistic clocks for two different message load patterns. The first pattern is composed of intervals in which the message load goes from 10 to 200 messages broadcasted per second. The second pattern consists of randomly chosen message load targets reached every 20 seconds, with 3 peaks during the experiment. Figures 6 and Figure 8 show the message loads of the experiment using each pattern. The experiment contains 1000 processes that increment two entries when broadcasting a message.

1) Bell message load pattern: Figure 6 shows the message load corresponding to the first message load pattern. Figure 7 shows the average size of the active components of processes' *DCS* clocks.

We observe that *DCS* clocks fastly adapt to the message load. Moreover, in the experiment they have on average 260 entries. Therefore, we executed an experiment with the same message load pattern by using a Probabilistic clocks of 260 entries. Results show that, in a system with a message load as in Figure 6, *DCS* clocks are more effective to causally order messages than Probabilistic clocks: they deliver only 58 messages out of causal order while Probabilistic clocks deliver 231 messages out of causal order. The out of causal order deliveries are for both algorithms concentrated around the message load peaks.

To conclude, *DCS* clocks have better performances than Probabilistic clocks in systems with a bell message load pattern as presented in Figure 6.

2) Random message load pattern: Figure 8 shows a random message load pattern. Figure 9 shows the average size of the active components of processes' *DCS* clocks.

Results confirm that *DCS* clocks adapt well to the message

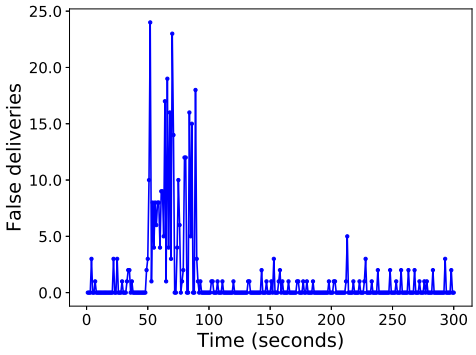


Fig. 10: Out of causal order deliveries before load balancing

load: they remain small most of the time and grow only during message load peaks. Processes maintain on average a *DCS* clock with 97 active entries. Therefore, we executed an experiment with the same message load and a Probabilistic clock of 97 entries.

The results are the following: with *DCS* clock we measured 45 out of causal order deliveries, while with a Probabilistic clock of 97 entries we measured 305 out of causal order deliveries. The out of causal order deliveries were concetrated around the message load peaks for both algorithms.

Therefore, *DCS* clocks are much more accurate than Probabilistic clocks - 6x in our experiment - in random patterns as the one presented in Figure 8.

C. Load balancing

Constant size clocks are the most accurate when their entries are incremented uniformely. Processes should therefore be affected to clock entries such that the entries are incremented uniformely. However, a process might vary the number of messages it broadcasts during execution, thus rendering an initially uniform clock incrementation non uniform.

Nevertheless, with *DCS* clocks processes can be associated to new clock entries. The last experiment measures the ability of *DCS* clocks to reaffect processes to new clock entries in order to increment clock entries more uniformely.

We consider a system composed of 1000 processes, which increment two entries when broadcasting a message (i.e. $|f| = 2$). The system has a message load of 100 messages broadcasted per second, and processes maintain a *DCS* clock of 200 entries. Processes broadcast on average the same number of messages until t=50s, i.e. all 4 components are incremented uniformely until t=50s. After that, the broadcast pattern of processes is modified from 25 to 75% of message broadcasts that increment component C_0, i.e. component C_0 is incremented 3 times more after t=50s. Figure 10 shows the number of messages that are delivered out of causal order.

We observe that the number of messages delivered out of causal order increases dramatically after t=50s. At t=70s, processes that increment component C_0 detect that component C_0 is much more incremented than other components. In order to rebalance, they assign themselves to other components with a probability of 50%. Consequently, the number of messages delivered out of causal order drops. Eventually, the remaining processes that increment component C_0 detect that C_0 is still more incremented than other components, and reassign themselves therefore to other components with a probability of 50% till the *DCS* clock is again incremented uniformely, which happens at t=90s. Therefore, the components of the *DCS* clock clock are eventually again incremented uniformely.

VI. Related work

As proved in [4], logical vector clocks [8] [5] are the smallest data structures that characterize causality of messages without assumptions on the model. They have a fixed size and contain an entry per node in the system. Therefore, they do neither scale nor tolerate the join/leave of nodes to/from the system. Some works have attempted to reduce vector clock control information included in messages. In [14], only incremental changes regarding previously transmitted values are included in messages while in [6], vector clocks are encoded in prime numbers. However, the size can still grow linearly or exponentially respectively.

Prakash proves in [12] that direct dependencies are sufficient to characterize causality. Hence, the control information of a message m is composed of the set of identities of those messages that are immediate predecessors of m. Since the causal information, denoted *causal barrier*, is not related to nodes but messages, the join and leave of nodes can be tolerated. Nevertheless, nodes maintain a matrix of size $O(N^2)$, and, in the worst case, the *causal barrier* structure included in a message has $2 * N$ size.

Depending on the system model and requirements, existing causal broadcast algorithms provide message causal order based on different approaches: (1) piggyback of control information on messages, (2) logical overlay (e.g. tree, ring, star, etc.) with FIFO links on which messages are broadcasted, and (3) logical hierarchical structures.

The first approach orders messages at reception based on control information (e.g., vector clocks, plausible clocks, probabilistic clocks, causal barriers, etc.) attached to the messages, as discussed in the paper.

In the second approach, upon the first reception of a message, a process sends it over all its outgoing links. Consequently, application messages are causally ordered at reception and do not require any control information. However, such an approach has several drawbacks: all channels must be FIFO, flooding can cause a great message overhead in overlay with cycles, and topology changes require extra cost handling. To avoid cycles, Blessing et al. [3] organize nodes into a static tree overlay that does not tolerate the dynamic inclusion of extra links since the latter would create new paths temporarily non-FIFO. Then, Nédelec et al. [11][10] extended [3] to dynamic topologies. The inclusion of new links in the topology is handled by handoff procedures which initialize new links before they are used. On the other hand, a path of already initialized links must already exist before adding a new link.

Finally, in Adly[1], nodes are grouped into clusters, and the latter are organized into a logical tree. Thereby, nodes only keep causal information about the nodes of their respective cluster. However, in the case of topology changes, the reorganization of multilevel hierarchical overlay is required.

VII. Conclusion

This paper has presented the Dynamic Clock Set (DCS), a new logical clock based on Probabilistic clocks. The main feature of *DCS* clocks over constant size ones is that their size can be dynamically adjusted during execution. This is particularly important since the optimal size of *DCS* and constant size clocks usually depend on the system's message load, which can drastically vary and whose knowledge beforehand is difficult or even impossible.

DCS clocks can for example be used to implement a causal broadcast algorithm. The paper gives such an algorithm. Moreover, the implementation of *DCS* clocks dynamicity operations is also provided.

Experimental results confirm that *DCS* clocks have a higher accuracy of causal message ordering when compared to Probabilistic clocks. Moreover, depending on the system's message load pattern, *DCS* clocks require less memory than Probabilistic clocks.

References

[1] N. Adly and M. Nagi. Maintaining causal order in large scale distributed systems using a logical hierarchy. In *IASTED Int. Conf. on Applied Informatics*, pages 214–219, 1995.

[2] K. P. Birman and T. A. Joseph. Reliable communication in the presence of failures. *ACM Trans. Comput. Syst.*, 5(1):47–76, 1987.

[3] S. Blessing, S. Clebsch, and S. Drossopoulou. Tree topologies for causal message delivery. In *AGERE workshop*, pages 1–10, 2017.

[4] B. Charron-Bost. Concerning the size of logical clocks in distributed systems. *Inf. Process. Lett.*, 39(1):11–16, 1991.

[5] C. J. Fidge. Timestamps in message-passing systems that preserve the partial ordering. In *11th Australian Computer Science Conference*, 1988.

[6] Ajay D. Kshemkalyani, Min Shen, and Bhargav Voleti. Prime clock: Encoded vector clock to characterize causality in distributed systems. *J. Parallel Distributed Comput.*, 140:37–51, 2020.

[7] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, 1978.

[8] F. Mattern. Virtual time and global states of distributed systems. In *Parallel And Distributed Algorithms*, pages 215–226, 1988.

[9] A. Mostéfaoui and S. Weiss. Probabilistic causal message ordering. In *Parallel Computing Technologies - 14th International Conference, PaCT*, pages 315–326, 2017.

[10] B. Nédelec, P. Molli, and A. Mostéfaoui. Causal broadcast: How to forget? In *22nd International Conference on Principles of Distributed Systems,OPODIS*, 2018.

[11] B. Nédelec, P. Molli, and A. Mostéfaoui. Breaking the scalability barrier of causal broadcast for large and dynamic systems. In *37th IEEE Symposium on Reliable Distributed Systems, SRDS*, pages 51–60, 2018.

[12] R. Prakash, M. Raynal, and M. Singhal. An efficient causal ordering algorithm for mobile computing environments. In *16th International Conference on Distributed Computing Systems*, pages 744–751, 1996.

[13] L. Ramabaja. The bloom clock. *CoRR*, abs/1905.13064, 2019.

[14] Mukesh Singhal and Ajay D. Kshemkalyani. An efficient implementation of vector clocks. *Inf. Process. Lett.*, 43(1):47–52, 1992.

[15] F. Torres-Rojas and M. Ahamad. Plausible clocks: Constant size logical clocks for distributed systems. In *WDAG 1996*, pages 71–88, 1996.