DCS : A logical clock composed of a dynamic set of probabilistic clocks

Daniel Wilhelm, Luciana Arantes, Pierre Sens

DCS : A logical clock composed of a dynamic set of probabilistic clocks

Daniel Wilhelm[1] et Luciana Arantes[1] et Pierre Sens[1]

[1]*Sorbonne Université, CNRS, LIP6, Paris, France*

Plusieurs horloges de taille constante (Probabilistic [5], Plausible [6]) ont été proposées dans la littérature. Ces horloges capturent la causalité avec une grande probabilité mais ne la caractérisent pas.

L'efficacité d'une horloge de taille constante pour ordonner correctement les messages est négativement impactée par la charge de messages du système. Néanmoins, cet impact peut être géré en variant la taille de l'horloge en fonction de la charge de messages. Cependant, la taille des horloges de taille constante existantes est fixée à l'initialization et ne peut pas changer durant l'exécution.

Ce papier présente les Dynamic Clock Set(*DCS*), qui sont constitués d'un ensemble d'horloges probabilistes. La taille de l'ensemble peut changer dynamiquement durant l'exécution et peut notamment s'adapter à la charge de messages.

Mots-clefs : Logical clock, Causal Broadcast, probabilistic clock, distributed algorithms

1 Introduction

Distributed and parallel applications are composed of an ever increasing number of processes that cooperate by exchanging messages. Many of them also require that events, i.e., sending and reception of messages as well as local events, are causally ordered as defined by Lamport's *happened before* relationship [3].

Logical clocks have been used in the literature [3][2][4] to track the causality of events in distributed systems. Events are timestamped with logical clock values, which are appended on application messages and are updated through rules. However, Charon-Bost[1] proved that, in order to precisely capture the causality of events, i.e., to characterize their causality, logical vector clocks must have one entry per process in the system. Thus, in a distributed system with N processes, causality can only be characterized with a structure in $O(N)$, inducing scalability issues, especially in large systems, since the size of vector clocks grows linearly with the number of processes.

Constant size clocks (e.g., Probabilistic [5], Plausible [6], etc.) circumvent such scalability issues by having a size that is much smaller and independent to than the number of processes. On the other hand, even though they provide a high level of ordering accuracy, they do not characterize causality. Such clocks are useful in systems where ordering concurrent events impacts only performance and not correctness [6].

The size of existing constant size clocks is fixed at initialization and does not vary during execution. However, the system's message load negatively impacts the accuracy of constant size clocks. Therefore, the clock size should ideally vary with the current message load [5], increasing (resp., decreasing) whenever the message load is above (resp., below) a given bound value. A wrong choice can result in an oversized clock or in a clock that is less effective in ordering messages.

This paper presents a new clock, denoted Dynamic Clock Set (*DCS*), which consists of a set of probabilistic clocks. The size of the set can dynamically vary during application execution. Operations required to modify the size of the clock as well as to compare them are defined. *DCS* clocks can, for example, be used to implement causal broadcast such as in the algorithm proposed in [5] that uses Probabilistic clocks.

2 Background

Causal order ensures that two causally related messages are delivered to applications respecting that order. The causal order relation between events is based on the well-known Lamport's **happened before**

relation between events in distributed systems [3] :

Happened before relationship : *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 : *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′ :*

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

A constant size clock V [6][5] has M entries, $1 \leq M \leq N$, where N corresponds to the number of processes of the system. M is usually much smaller than N. Probabilistic clocks have the best performances among constant size clocks[5]. Therefore, we use them to build the clock proposed in this article.

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 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 clock as follows : $\forall x, V_i[x] = max(V_i[x], m.V[x])$
  — Executes $R1$, Deliver(m)

Comparison of two probabilistic clocks Considering V_1 and V_2 :

$$V_1 < V_2, \text{ iff } \forall x, 1 \leq x \leq M, V_1[x] \leq V_2[x] \wedge \exists i, V_1[i] < V_2[x]$$

Note that probabilistic clocks ensures that : $send(m_1) \rightarrow send(m_2) \Rightarrow m_1.V < m_2.V$.

Using probabilistic clocks to implement a causal broadcast algorithm, the authors of [5] showed that the probability that a process delivers a message m out of causal order 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. We do two observations out of this equation. First, increasing the size of the clock attached to 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. In fact, 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.

3 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 1 gives a representation of a DCS clock. The number of components of a DCS can dynamically vary during execution. A component C_k is uniquely identified by its index k. It 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 1, i.e. C_1 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 $C_{incr,i}$ 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 $C_{incr,i}$.

The size of DCS clocks is adaptable to the message load. As shown in the Background, the efficiency of constant size clocks to causally order messages decreases when the message load increases. Nevertheless, increasing the size of a constant size clock also increases its resillience to the message load. The size of DCS clocks is dynamically adjustable during execution. DCS clocks can therefore ensure a desired accuracy of causal message ordering by varying their size according to the message load.

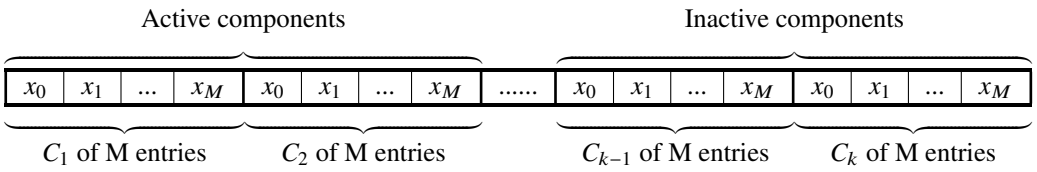

FIGURE 1: Representation of a Dynamic Clock Set

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 as follows :

$$\forall x \in f(p_i), \forall k \in C_{incr}, V_i.C_k[x] = V_i.C_k[x] + d \quad (d > 0)$$

— $R2$: 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)

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

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

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

(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 : $\forall k \in [1, |D_1|], D_1.C_k \leq D_2.C_k \wedge \exists C_j, D_1.C_j < D_2.C_j$.

We claim that 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 : $m_1.D \nleq m_2.D \wedge m_2.D \nleq m_1.D \Rightarrow m_1||m_2$.

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

□

Activate() / Deactivate() A process calls the operation $Activate()$ (resp., $Deactivate()$) to activate (resp. deactivate) an inactive (resp., active) component of its DCS clock.

We remind that *DCS* clock requires that the component of lowest index, namely C_1, to be always active.

Thus, the call to *Activate*() (resp., *Deactivate*()) immediately returns $false$ if p_i's *DCS* clock has no inactive (resp., only one active) component ; otherwise, the component of p_i's *DCS* clock with the lowest (resp., highest) index among its inactive (resp., active) components will be activated (resp., deactivated).

For example, let's consider a process p_i whose *DCS* clock D has four components : $D = \{C_1, C_2, C_3, C_4\}$. If p_i wants to deactivate one or more components, such a deactivation will be done in decreasing order, i.e, first C_4, then C_3, and finally C_2. On the other hand, if p_i wants to re-activate them, first C_2 will be activated, then C_3, and finally C_4. Furthermore, p_i locally keeps deactivated components C_d, since p_i can receive a message from another process p_j that contains C_d, i.e. C_d has not been deactivated by p_j yet. By locally keeping C_d, p_i ensures that the delivery conditions of C_d will be satisfied locally.

Add() / Remove() 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 in active state, sets its entries to 0, and appends C to the end of its *DCS* clock. Therefore, C will be the component of p_i's *DCS* clock with the highest index.

Having called the *Add*() operation, p_i should also call the *Activate*() operation for all the inactive components C_k of its *DCS* clock. Indeed, by definition, a *DCS* clock is composed of active and inactive components and the index of the latter are strictly higher than the former. Consequently, adding an active component at the end of a *DCS* clock implies that all components of that *DCS* clock must be active.

Process p_i calls *Remove*() to remove the component of its *DCS* clock with the highest index.

4 Conclusion

This paper presents the Dynamic Clock Set (*DCS*), a new logical clock built with probabilistic clocks. The size of *DCS* clocks is modifiable during execution, contrary to the size of constant size clocks. This is particularly important since the optimal size of *DCS* and constant size clocks depends on the system's message load, and knowing the system's message load beforehand is difficult or even impossible.

We implemented a causal broadcast algorithm using *DCS* clocks and compared it to the causal broadcast algorithm using probabilistic clocks proposed in [5]. Experimental results confirm that *DCS* clocks adapt well to the message load and can reduce the size of causal information as well as the number of messages delivered out of causal order. The causal broadcast algorithm and experimental results could not be included due to lack of space and can be found in [7].

Références

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

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

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

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

[5] A. Mostéfaoui and S. Weiss. Probabilistic causal message ordering. In *PaCT*, pages 315–326, 2017.

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

[7] Daniel Wilhelm, Pierre Sens, and Luciana Arantes. A probabilistic Dynamic Clock Set to capture message causality. working paper or preprint, February 2023.