A Probabilistic Causal Message Ordering Mechanism

Achour Mostefaoui, Stéphane Weiss

A Probabilistic Causal Message Ordering Mechanism

Achour Mostefaoui Stéphane Weiss

Abstract

Causal broadcast is a classical communication primitive that has been studied for more then three decades and several implementations have been proposed. The implementation of such a primitive has a non negligible cost either in terms of extra information messages have to carry or in time delays needed for the delivery of messages. It has been proved that messages need to carry a control information the size of which is linear with the size of the system. This problem has gained more interest due to new application domains such that collaborative applications are widely used and are becoming massive and social semantic web and linked-data the implementation of which needs causal ordering of messages.This paper proposes a probabilistic but efficient causal broadcast mechanism for large systems with changing membership that uses few integer timestamps.

1 Introduction

Nowadays, we are facing an increasing number of collaborative applications. The nature of these applications is diverse as they appear as web 2.0 applications such as blogs, wikis or even social networks, as well as applications for mobile devices such as foursquare, yelp, latitude. Moreover, semantic web (web 3.0) and now social semantic web and linked-data (web 4.0) such as DBpedia are gaining more and more interest. The common point to these applications is that they gather the outcome of numerous users in order to provide a service for their users. The more users participate, the more content is created, attracting more users. This virtuous circle tends to create very large scale systems. However, while the content is created by users for users, for many applications, the underlying architecture remains centralized, leading to scalability issues as well as privacy and censorship threats. Stating this observation, several work envision decentralized architectures. The idea behind this concept, is to put the users as nodes of the network, allowing direct communication between users.

Currently, the development of such applications is restricted by several scientific problems. Among them, the problem of data replication has been investigated for many years, and have provided several approaches such as appropriate replicated data structures [10, 13] and programming languages [1]. Hence replicated data can be enriched, updated and queried. However, the implementation of these operations has an underlying requirement: causally ordered communication (causal order for short) [14, 15]. Informally a causal communication primitive imposes some restrictions on the delivery order of sent messages. It can be seen as an extension of the FIFO channel property to a whole communication network. Hence, a causal broadcast communication service imposes that a message is delivered to a some process only if all the messages that have been delivered in its past have been already delivered.

Unfortunately, causal communication has a cost that can be high either in time (message exchanges) or in space (the size of control information carried by messages). This cost becomes unacceptable when we consider a very large scale network of nodes with churn. Moreover, if the set of participating users changes over time, one needs to offer a join/leave decentralized procedure that is theoretically impossible to implement in asynchronous systems [7].

Interestingly, in a real setting, depending on the system we consider, the probability to deliver in a non causal order two messages the sending of which are causally related may be quite low. For example,

if the time between the generation of two messages on each peer is bigger than the transit time of a message, most of messages will be received in the causal order without any explicit control or synchronization. This observation is in favor of a probabilistic mechanism.

In this paper, we propose a probabilistic causal broadcast that provides a causal communication with high probability at a low cost for very large systems while allowing continuous joins and leaves. Of course it may happen, in few situations, that causal ordering is not respected. The proposed solution is then evaluated from a theoretical point of view and by simulation.

2 Related Work

The first causal broadcast mechanism was introduced in the ISIS system [2]. The simplest way to implement causal communication consists in piggybacking on each messsage a process want to send the whole set of messages it has delivered prior to this sending. Of course, this is very costly and there is a need to some kind of garbage collector. Otherwise, prior work mainly use either a logical structure (central node, tree, ring, etc.) or are based on the use of timestamps. A timestamp is an integer value that counts events (possibly not all events). A vector clock is a vector of such counters. The first solution based on vector clocks for a broadcast primitive has been proposed is [12] (a solution based on a matrix of counters has been proposed in [11] for point-to-point communication). Vector clocks introduced simultaneously by [6, 9] have been proved to be the smallest data structure that can capture exactly causality [4]. Moreover, vector clocks require to know the exact number of sites involved in the application. As an example, the churn (intempestive join and leave of processes) and the high (and unknown) number of processes make the use of vector clocks unrealistic.

Torres-Rojas and Ahamad presented an approach based on *plausible clocks* [16]. Its aim is to trade the quality of the detected causality (number of false positives and false negative) among events (messages sending events) against timestamp size. When using a vector clock of size 1, a plausible clock boils down to Lamport's clocks [8] then as the timestamp size increases, a more and more accurate causal relation is encoded. Finally, when considering a timestamp size equal to the total number of processes, plausible clocks meet vector clocks. In a vector clock, the entry j of the vector managed by a given process p_i counts the number of messages broadcast by process p_j, to the knowledge of p_i. Indeed due to asynchronism, the different processes do not have the same view of the state of the system at a given time instant. The approach of Torres-Rojas and Ahamad consists in associating several processes to the same entry of each vector clock.

The approach presented in this paper is an extension to the one of Torres-Rojas, namely each entry is associated to several processes and moreover, to each process are associated several entries of the vector clock. To summarize, let us consider the triplet (a, b, c). Where a is the size of the system (number of processes), b the size of the vector and c the number of entries associated with each process. A Lamport clock is $(n, 1, 1)$ where n is the total number of users in the system, a vector clock is $(n, n, 1)$, a plausible clock is $(n, r, 1)$ and the proposed approach is (n, r, k) (r and k being two constants $n \geq r \geq k$).

3 System Model

When we consider the application level, the different users, nodes, processses or whatever we call them share common information by mean of replication to be able to tolerate chrashes and unexpected leaves (each process manages a local copy of part of the whole set of data). The differents processes interact by mean of operations (insert / delete / update a piece of data, make a query, etc.). The frequency and the distribution of operations through time and space depend on the application. At the underlying level, an operation will entail a change in the local state of a process and possibly the sending of messages to inform the other processes as the system is message-passing (no shared memory).

At the abstraction level considered in this paper, a distributed computation is a large set Π of n processes/users (N and Π are not necessarily known to the different processes of the system). We note p_i or p_j any processes in Π. We assume that processes gen-

erate messages at arbitrary rates. Messages are sent to all processes using a reliable broadcast mechanism (broadcast sending primitive). Any process $p_i \in \Pi$ generates three kinds of events. An event e could be a local event, a send event or a delivery event. A local event induces no interaction with other processes and thus will be omitted in the rest of the paper.

Events produced by a distributed computation are ordered by Lamport's *happened-before* relation [8].

Definition 1 (Happened-before Relation [8]) *We say that event e_1 happened before event e_2 denoted by $e_1 \rightarrow e_2$ if:*

- *e_1 occurred before e_2 on the same process, or*
- *e_1 is the send event of some message m and e_2 is the delivery event of the same message by some process, or*
- *there exists an event e_3 such that $e_1 \rightarrow e_3$ and $e_3 \rightarrow e_2$ (transitive closure).*

Let us note $send(m)$ the sent event of a message m and $del(m)$ the associated delivery event. Note that $del(m) \neq rec(m)$ the receive event. The receive event corresponds to the arrival of a message to the underlying communication level of some process. The application level of this same process is not aware of the arrival of such message and thus has no access to its content. The delivery of a message corresponds to the arrival of the message at the application level leading to the use of its content. Moreover, when considering two messages m_1 and m_2, we say that $m_1 \rightarrow m_2$ if $send(m_1) \rightarrow send(m_2)$.

A distributed computation *respects causal order* if for any pair of messages (m_1, m_2) the following holds.

- $send(m_1) \rightarrow send(m_2) \Rightarrow del(m_1) \rightarrow del(m_2)$

To ensure the aforementioned property an arrived message m, event $rec(m)$, at a destination process p_i can be possibly delayed until all the messages sent or delivered by the sending process p_j before the sending of m have been already delivered by the receiving process p_i.

4 Probabilistic Causal Broadcast

Several works in the literature propose to reduce the communication cost to increase scalability by proposing probabilistic solutions. One example is the probabilistic broadcast [5]. We can define a probabilistic broadcast as following:

Definition 2 (Probabilistic Broadcast) *A probabilistic broadcast is a mechanism that ensures, with a high probability, the deliverance of sent messages to all participants.*

Unlike traditional broadcast that ensures that each message is delivered exactly once to all recipients, a probabilistic broadcast offers only a very high probability that all recipients will receive the message. In addition, this message can be received several times, requiring a mechanism to discard duplicated messages. On the contrary, probabilistic broadcast have a greater scalability.

In this paper, we introduce two definitions: the probabilistic causal ordering mechanism and the probabilistic causal broadcast.

Existing mechanisms such as vector clocks are often used to provide a perfect causal ordering mechanism. We believe that their cost is not compatible with large systems, and propose to use a probabilistic causal ordering mechanism. We define it as follows:

Definition 3 (Probabilistic Causal Ordering) *A probabilistic causal ordering mechanism is a mechanism that ensures a causal delivery with high probability.*

4.1 A Probabilistic Causal Ordering Mechanism

In this section, we start by describing the data structures used by our causal ordering mechanism. Then, we describe the algorithms that allow this probabilistic causal delivery.

4.1.1 Data structure

The main idea of the proposed solution is to associate with each process p_i a vector of integer values V_i of

size $R < N$ (N being the total number of processes in the system). This vector acts as a logical clock that allows to timestamp a subset of the events generated by this process. This timestamping allows to test whether an event occured before another one. Mainly, the proposed protocol associates a logical date with each send event and this date is attached to the message and is called its timestamp. When a message is received, the receiving process compares its local logical clock with the timestamp carried by the received message. This allows it to know whether there exist messages sent causally before it and that have not yet been delivered. As soon as a message can be delivered it is given to the upper layer application and the local clock is updated to take into account that this message has been delivered.

We denote $V_i[j]$ with $0 \leq j < R$ the j-th entry of the vector clock of the process p_i. A vector clock assigns exactly one entry to each process. With a plausible clock, each process is assigned only one entry, but one entry is assigned to several processes. Finally, our approachproposes to assign several entries to each process, each entry being assigned to several processes. We denote $f(p_i)$ the set of the entries assigned to p_i.

4.1.2 Probabilistic causal ordering delivery mechanism

The proposed probabilistic causal ordering delivery mechanism is an adaptation of the classical and well-known causal delivery mechanism [2].

When a process p_i wants to broadcast a message m, it executes Algorithm 1 given below.

First, process p_i increments all its assigned set of entries $f(p_i)$ in its local vector. Then, a copy of this local vector is attached to the message to be sent. $m.V$ denotes the vector timestamp attached to message m. Finally, m is broadcast to all processes.

Input: m: message to broadcast
$\forall x \in f(p_i), V_i[x] = V_i[x] + 1$;
$m.V = V_i$;
Broadcast(m);

Algorithm 1: Broadcasting a message m by p_i

When a message m broadcast by a process p_j is received by a process p_i, this process executes Algorithm 2 given below.

Input: m: message received by p_i from p_j
waitUntil($(\forall x \in f(p_j), V_i[x] \geq$
$m.V[x] - 1) \wedge \forall k \notin f(p_j), V_i[k] \geq m.V[k]$);
$\forall x \in f(p_j), V_i[x] = V_i[x] + 1$;
deliver(m);

Algorithm 2: Upon reception of message m by p_i

A message m received by a process p_i from a process p_j is queued until it is considered as causally ready, namely all the messages m' sent causally before it ($m' \rightarrow m$) have been already delivered by process p_i to the application level. Note that:

On the one side, the $f(p_j)$ entries are at least as high as the local vector of process p_j before it generates that message: $\forall x \in f(p_j), V_i[x] \geq m.V[x] - 1$. This means that all the messages sent by process p_j but message m are already known at process p_i.

On the other side, the other entries of the vector are at least as high as the local vector of process p_j before it generates that message: $\forall k \notin f(p_j), V_i[k] \geq m.V[k]$. This means that at least all the messages that have been delivered to process p_j before it broadcast message m are known at process p_i before p_i delivers m to its application level.

When the delivery test of a message m holds, process p_i increments the entries of its local vector that belong to the set $f(p_j)$ of entries of its local vector before delivering the message to the application. Hence p_i has recorded the information "m has been delivered to the p_i" in its local vector.

Figure 4.1.2 shows how the proposed mechanism works. Assume that three processes p_i, p_j and p_k are in the same initial state, meaning they have generated no messages yet, and all values in their vector are set to 0. Also, we consider here that each process has a vector of $R = 4$ entries and each process is assigned $K = 2$ entries. We assume that $f(p_i) = \{0, 1\}$, $f(p_j) = \{1, 2\}$ and $f(p_k) = \{3, 4\}$.

On that state, p_i generates a first message called m. If we assume that $f(p_i) = \{0, 1\}$, then after applying the algorithm 1 at p_i, its vector becomes $[1, 1, 0, 0]$. Notice that it is this vector ($[1, 1, 0, 0]$) that is attached to m. This message is sent to all other processes, here we represent only p_j and p_k. We assume that p_j receives m first: Algorithm 2 is applied and,

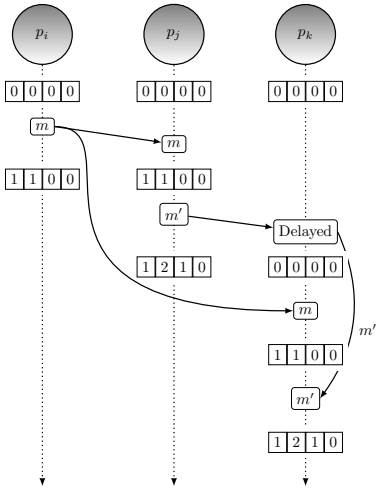

Figure 1: Example of message delivery

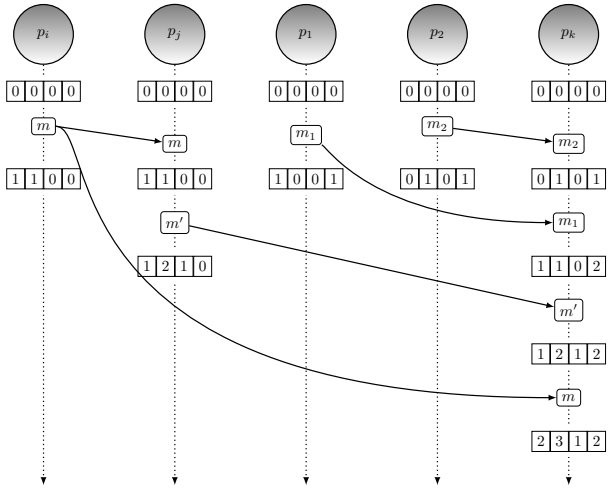

Figure 2: Example of possible delivery error

p_j's vector is updated to $[1, 1, 0, 0]$. Then p_j generates a new message m'. Assuming that $f(p_j) = \{1, 2\}$, the generation of m' leads to update p_j's vector to the value $[1, 2, 1, 0]$ which is piggybacked with the message m'. As m' is generated after m, m' is causally dependent of m ($m \rightarrow m'$). Although message m' is broadcast and will eventually reach p_i, we only represent its reception by process p_k for the sake of simplicity.

When p_k receives m', the vector p_k is $[0, 0, 0, 0]$ while the vector attached to m' is $[1, 2, 1, 0]$. The delivery of m' is delayed because its delivery condition (see Algorithm 2) is not satisfied ($V_k[1] < (m.V[1]-1)$ and $V_k[0] < m.V[0]$).

The reception of m turns p_k's vector into $[1, 1, 0, 0]$ which fulfills the condition for delivering m'.

This protocol cannot be perfect as it uses control information the size of which is smaller than a vector clock of size N that has been proved to be minimal for ensuring causal delivery of messages. Indeed, it is possible that if process p_k receives some set of messages before receiving m', the vector of p_k could have been updated in such a way that p_k believes that m' is causally ready. This scenario is illustrated by Figure 4.1.2. In addition to the previous processes, we now consider p_1 and p_2 whose assigned entries are respectively $f(p_1) = \{0, 3\}$ and $f(p_2) = \{1, 3\}$. Processes p_1 and p_2 generate m_1 and m_2, which are received by p_k before m'.

The reception of m_2 and m_1 updates the vector of p_k to the value $[1, 1, 0, 2]$. When m' arrives, p_k evaluates to true the delivery condition of Algorithm 2 and delivers m' while m has not been received yet. This error comes from the fact that all entries of p_i are matched by the combination of the entries of p_1 and p_2: $f(p_i) \subseteq (f(p_1) \cup f(p_2))$. It is interesting to notice that, as each set of entries is assigned to at most one process, the error occurs only if we have at least two concurrent messages, here m_1 and m_2. If only one concurrent message is received, the protocol delivers the message in causal order.

In addition, it is easy to see that the proposed protocol never delays the delivery of an arrived message m' at some process p_i if all the messages m that have been sent causally before m' (i.e. $m \rightarrow m'$) have been already delivered by process p_i. Indeed, between two consecutive sendings of messages, on the same process, the increment is one for some entries and when a message is delivered the corresponding entries are at least incremented by 1 then if the first one is delivered, the second cannot be blocked as the corresponding entries are augmented by at least 1. The same happens for messages sent by different processes as in this case the sender of the second message has

necessarily delivered the first message to establish the causal dependency between the two messages.

4.1.3 Generation and distribution of the sets of keys

The example given above gives the intuition why the generation of the sets of keys (entries in the vector clock) is the core of the approach, and how it heavily affects the accuracy of the resulting protocol. In this section, we present our approach to assign sets of keys to processes.

Perfect distribution of keys We assume the existence of a mapping function f used to assign to each process a set of K entries of the vector such that, the function f returns exactly K distinct values between 0 and $R-1$ (R being the total number of entries, i.e. the size of the vector clock). Moreover, the values are equally distributed among processes. In other words, for all subsets of s values assigned to x processes, there exists no other subset of s values assigned to more than $x+1$ processes, or less than $x-1$ processes, for $1 \leq s \leq K$.

Input: set_{id}, K, R
Output: $\{a_1, \ldots, a_k\}$
$k \leftarrow K-1$; $r \leftarrow R-1$; $t \leftarrow C_k^r$; $sav \leftarrow t$;
for $i = 0 \rightarrow K-1$ **do**
  while $set_{id} \geq t$ **do**
    $a_i \leftarrow a_i + 1$; $r \leftarrow r-1$; $sav \leftarrow t$; $t \leftarrow C_k^r$;
  end
  $t \leftarrow sav$;
  if $i < K-1$ **then**
    $a_{i+1} \leftarrow a_i$;
  end
  $k \leftarrow k-1$;
end

Algorithm 3: Generating the set of entries from set_{id} for $K > 1$

Distributed key assignment algorithms Finding the best solution is a difficult problem. Moreover, such an algorithm would not support dynamicity. Indeed, the addition or the removal of one process would lead to re-assign all new entries to all or part of the processes. Therefore, to select the K values, we propose to use a random algorithm.

Each process generates randomly a value called set_{id} chosen between 1 and C_K^R. Then, the corresponding set of values is obtained using the function f defined below:

This algorithm ensures that no peer has twice the same entry. Moreover, if we can ensure that all peers have different identities, all the generated sets are distinct, and the intersection between all the sets is at most $K-1$.

4.2 Detecting Delivery Errors

Most of the applications that require causal ordering of messages assume that no message is wrongly delivered. The proposed approach that aims to scale, may deliver a message before a message that causally precedes it. This would lead to inconsistencies, however, we make the assumption that a recovery procedure does exist (e.g., anti-antropy). This procedure may be costly, and we must determine when it is required.

A simple solution could be to run it at an arbitrarily chosen period of time. However, this period impacts the system performance: on the one hand, if it is underestimated, we will waste time by running unnecessary costly recovery procedure. On the other hand, the over estimation of this period will cause heavy changes on the application and will hurt the usability of the system and the recovery from this inconsistent global state.

We propose a mechanism (Algorithm 4) that allows to detect possible delivery errors if run when receiving a message, prior to the deliver function

Input: m: message received from p_j
Output: $error$: *true* if an error has occurred
$error \leftarrow$ *true*;
if $(\exists x \in f(p_j), V_i[x] = m.V[x] - 1)$ **then**
$error \leftarrow$ *false*;
endif

Algorithm 4: Before delivering m from p_j to the application

Before delivering a message m from a process p_j, we check if all its entries are already higher than their corresponding value in the message, meaning that concurrent messages have covered all its entries. In that case, the receiving process p_i wrongly believes that it has already received this message, and may have delivered other messages m' such that $m \rightarrow m'$.

If the procedure returns *false*, we are sure that all is fine: no such message m' does exist. However, if it returns true then either there may be a causal order violation or not. In other words, this does not mean that m will not be delivered before some message m'' such that $m'' \rightarrow m$.

Interestingly, the detection gives an alert within the propagation time of the message from p_j. As a result, errors are detected quickly after they occur. Unfortunately, this mechanism greatly over-estimate the number of errors (causal order violations). In the following section, we propose two mechanisms to limit the number of false detections.

4.2.1 Improving the protocol

As explained above, we can have an alert within the propagation time of the late message. As a consequence, we propose the following protocol where we assume that each process keeps a list L of the last received messages.

Input: m: message received from process p_j, L: list of previously delivered messages
Output: *error*: true if an error may have occurred
$error \leftarrow false$;
if $(\forall x \in f(p_j), V_i[x] > m.V_i[x]) \wedge (\exists m_i \in L. \forall x \in f(p_j), m_i.V[x] \geq m.V[x])$ **then** $error = true$;
endif

Algorithm 5: Before delivering m from a process p_j to the application

Algorithm 5 is similar Algorithm 4 but it checks in the list of previously delivered messages. The main idea is to check if a message previously delivered depends on the same values for the keys $f(p_j)$ set by the message m. If the size of L is infinite, this mechanism may over-estimate the number of error, but will never under-estimate it. The list L should contain all messages received for a time $O(T_{propagation})$ where $T_{propagation}$ depends on that actual communication network and keeps the value of the estimate transmission delay.

It is interesting to notice that such a list may already be available. Indeed, broadcast mechanisms based on gossip protocols, which duplicate a message to ensure that it reaches all peers or based on an unreliable communication protocol like UDP, keep a short list of the moste recently received messages to circumvent message loss and multiple deliveries of a same message.

5 Theoretical analysis

5.1 Correctness

Lemma 1 *A well-formed message is eventually delivered.*

Proof 1 *We prove the liveness by induction.*

H0: Any message generated on the initial state will be eventually delivered.

In the initial state, all entries of the local vector are 0. Therefore, the vector of a message m_0^i generated by p_i on the initial state contains zeros for all entries except the all $f(p_i)$ entries which are 1.

$$\forall x \notin f(p_i), m_0^i.V[x] = 0 \wedge \forall x \in f(p_i), m_0^i.V[x] = 1$$

A vector cannot have values below 0, hence we can prove that any message generated on the initial state can be delivered on any process p_j.

$$\forall x, j\, V_j[x] \geq 0 \Rightarrow (\forall i, j\, (\forall x \notin f(p_i))\, V_j[x] \geq m_0^i.V[x] \wedge \forall x \in f(p_i)\, V_j[x] \geq (m_0^i.V[x] - 1))$$

H1: We assume a set S of messages that are eventually delivered on all processes. A message m generated on any process p_i after the delivery of S will be eventually delivered on all processes.

The delivery of all messages in S implies that the local vector p_i is defined by $\forall x, V_i[x] = \sum_{\{j|x \in f(j)\}} n_j$ where n_j is the number of messages generated by p_j in the set S. Notice that the order of delivery have no impact on the obtained vector.

We make the assumption that S is eventually delivered on all processes, hence it is obvious that eventually all processes will have a local vector which entries are greater or equal to V_i before the generation of m. A direct consequence of this observation is that m will be eventually delivered:

$$\forall x, j\, (V_j[x] \geq V_i[x]) \Rightarrow \forall i, j, (\forall x \notin f(p_i))\, V_j[x] \geq m^i.V[x] \wedge \forall x \in f(p_i)\, (V_j[x] \geq (m^i.V[x] - 1))$$

We prove that any message generated after a set of eventually delivered messages is also eventually delivered (see H1). As any message generated on the initial state is eventually delivered (see H0), we conclude that any message is eventually delivered.

Corollary 1 *A message causally ready is never delayed.*

5.2 Complexity Analysis

In this section, we evaluate the complexity of our algorithms. The first algorithm is in charge of the creation of a message to be sent. The K entries assigned to a process that wants to send a message are incremented. Then, the R entries of the vector are attached to the message to be sent. Thus, Algorithm 1 has a complexity of $O(R)$. Similarly, the Algorithm 2 that evaluates the delivery condition of an arrived message has to go through the R entries of the vector leading to a complexity of $O(R)$.

The third algorithm assigns K entries to a site from its unique identifier. The first loop in the algorithm is done at most K times. The embedded loop will be done at most R times independently from the first loop, meaning that the R entries in the loop are split over the K entries of the for loop. The algorithm needs to compute $R+1$ combinations denoted as C_k^r each of them having a complexity of $O(K)$. As a consequence, the overall complexity of the third algorithm is $O(RK)$.

5.3 Error Analysis

In this section, we evaluate the error rate of our probabilistic causal ordering mechanism depending on the estimation of the system load and the different parameters of our approach (N, R and K).

Let us first say that the higher R the better is the resulting protocol (better in the sense less messages that violate causal ordering). At the extreme, if $R = 1$ we have a linear clock similar to Lamport's clock. At the other extreme, $R = N$ we get the perfect and optimal (no causal order violation and delivery at the earliest) solution to causal ordering. Concerning K, it is easy to see that the situation is less clear. Indeed, if K is too small; at the extreme $K = 1$ we get the plausible clocks of Torres and Ahamad, and we are sure that if two process are assigned the same entry they will interfere at each message sending. At the other extreme $K = R$ the protocol boils down to the use of a Lamport's clock merging all processes within one single entry. Hence the intuition of the proposed approach is that there is some value of K to determine that lies between 1 and R that is optimal. The aim of this theoretical analysis and the simulations presented in the coming sections is to determine the best value for K and to evaluate the impact of K on the error rate.

First, we need to compute the probability that a message m is delivered before a message m' that precedes it causally ($m' \rightarrow m$). As explained in Section 4.1.2 if such messages m and m' are received by some process p_i with m received first, then m can be delivered before m' with a probability that we note P_{nc} the probability that a message m bypasses a message m' sent causally before it. Depending on the system we consider, this probability may be quite low. Therefore, we have to consider this probability to dimension precisely the size of the vector and the number of entries each process chooses. A necessary condition, though not sufficient, to wrongly delivered a message is that this message is received after a preceding message, and the entries of the delayed messages have been all matched by concurrent messages. Let us note P_{error} the probability that all the K entries of the missing message are covered by a set of concurrent messages (see the example given in Section 4.1.2). Consequently, the probability P of wrongly delivering a message is bounded by the probability that a delayed message has its entry matched concurrently $P \leq P_{nc} * P_{error}$.

The probability that a message is replaced by a set combination of previous messages is computed following the same scheme as the false positive error of a bloom filter [3].

The probability that one entry is incremented is $1/R$. So the probability that it is not incremented is: $1 - 1/R$ and that it is not set by X messages is $(1 - 1/R)^{k*X}$. Then the probability that one entry is incremented by X messages is $1 - (1 - 1/R)^{k*X}$. Finally the probability of an error delivery is $(1-(1-$

$1/R)^{k*X})^k$.

We need to find the value K that minimizes the probability of an error. We can easily show that $(1-(1-1/R)^{k*X})^K$ is minimal when $K_{min} = ln(2) * \frac{R}{X}$.

5.4 Experiments

In this section, we detail the model used to run our simulation. In the first part, we show that the estimation of the optimal value of K is sound. Then, in a second part, the experiments are run using this optimal value. We, therefore, show the behavior of the mechanism based on the size of the vector. In a third part, we show the accuracy of the estimation of the probability of an error occurrence.

Methodology: In order to evaluate our proposal, we have developed a simple event-based simulator. Each process generates messages according to a Poisson distribution of parameter λ. Each message has its own propagation time d described as a random value which follows a Gaussian distribution $N(\mu, \sigma^2)$ law. Each process receives a message whose propagation time is according to a $N(d, \sigma_m^2)$.

In the average, each node generates a message each second The message propagation time d follows a normal distribution law $N(100, 20)$ and the skew between a message reception on all nodes follows also a normal distribution law $N(d, 20)$.

5.4.1 Detecting delivery errors

One of the challenges to evaluate the proposed approach is to measure the error rate. When a message is said to be "causally ready" by our mechanism, we need to verify that it is really causally ready, therefore, in our simulator we also need to implement a perfect causal broadcast. This additional mechanism should have the lowest cost possible as it limits the simulator scalability. To detect an error, we must know all the messages the sending of which happened-before a given message. A simple solution would be to attached a set of messages to each sent message. Obviously, this would limit drastically the scalability of the simulator. Therefore, we use a mechanism based on vector clocks. Unfortunately, a vector clock cannot capture wrongly delivered messages.

Indeed, when a non-causally ready message arrives, the causal ordering mechanism delays it, while in our case, it may be delivered to the application.

To deal with this case, we update the local vector clock by taking the maximum of the local vectors and the wrongly delivered messages. Therefore, missing messages will be dropped by the perfect causal delivery mechanism. Detecting precisely if the missing messages are causally ready is costly, instead we propose two metrics. The first one, ϵ_{min}, simply assumes that all missing messages are delivered in a causal order, while the second one ϵ_{max} assume that all of them are delivered in a non-causal order. Finally, we have two bounds on the error rate: the lower bound ϵ_{min} and the upper bound ϵ_{max}.

5.4.2 Choosing the optimal number of keys

The first step in validating our approach is to verify that we choose the best value for the parameter K. Therefore, we ran several experiments by changing only the value of K and then compare the value that minimizes it with the theoretical optimal value.

Figure 3 shows the error rate for respectively 500, 1000, 1500 and 2000 peers. In this experiment, the average number of messages received by a process is constant (200 messages/second) and, the number of exchanged messages is more than a hundred million.

As our simulation considers an average message propagation time of 100ms, the average number of messages that are received concurrently is 20. We use 100-entry vectors, hence, the optimal number of keys is theoretically $ln(2) * 100/20 \approx 3.5$ and the experimental results show that the value for K that minimizes the error rate for this configuration is 4.

5.4.3 Impact of the different parameters

In this section, we assume these system parameters:

- Number of nodes: $N = 1000$
- Each node generates a message every 5 seconds in the average ($\lambda = 5000$)
- The vector has 100 entries ($R = 100$)
- Each node has 4 entries ($K = 4$)

As a real system may behave differently from the estimation, we are now interested in the impact on the error rate when we vary only one parameter.

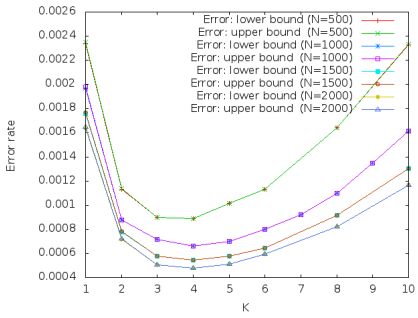

Figure 3: Number of errors for different value of K (Theoretical best value is 3.5)

Impact of λ on the error rate The error rate depends on the number of processes that send messages. Therefore, to determine the optimal size of the vector, and the number of keys, we need to estimate the number of "active" processes during an average message propagation time. Figure 4 shows the variation of the error rate according to different values of the average delay between two messages generated by each process.

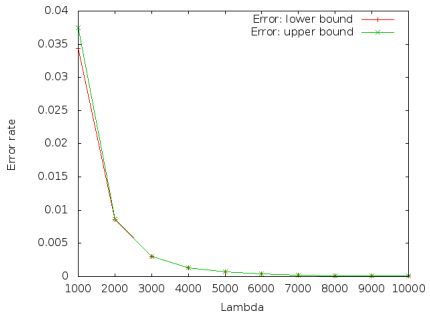

Figure 4: Error rate for different values of λ

The Figure shows that the error rate is stable around the estimate value ($\lambda = 5000$), but it increases quickly when λ is lower than 3000.

Impact of N on the error rate Figure 5 shows the impact of the number of nodes. The error rate increases quickly as soon as the number of nodes is higher than the estimation ($N = 1000$). As a result, 1000 should be considered as the maximum number of nodes in this case (λ, R, K, N).

Figure 6 shows the impact of the number of nodes

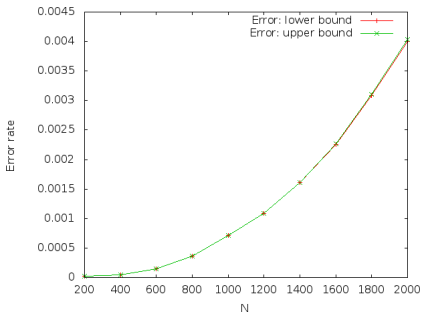

Figure 5: Error rate for different values of N

when the number of received messages is constant. In this experiment, the estimated number of nodes is 1000. We can see that the error rate remains constant when the number of nodes increases. Also it increases when the number of nodes decreases. This can be explained by the constant message rate, when the number of nodes decreases, the message rate of each node increases.

The previous experiments show that indeed, it is not λ, N by themselves that direct impact the error rate but the "concurrency". We mean by concurrency the mean number of messages that are broadcast during the transit time of some message (latency of the network).

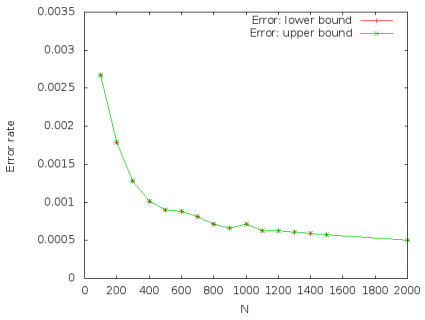

Figure 6: Error rate for different values of N, but with a constant number of received messages

6 Conclusion

In this paper we presented a new approach that allows to heavily reduce the cost of causal broadcast communication primitive. This reduction of the cost leads to a small rate of errors. The errors being the cases when a message is delivered while there are causally related messages that need to be delivered that are not yet delivered.

We have shown that the approach is theoretically sound. The main parameter K optimizes the protocol and may vary from 1 and R the two extreme already existing cases. The second contribution of the paper is an alert mechanism that allows to check the bad cases. In case there is no alert, we are sure there is no error.

References

[1] P. Alvaro, N. Conway, J.M. Hellerstein, and W.R. Marczak. Consistency analysis in bloom: a CALM and collected approach. In *Proc. of CIDR-11,*, pages 249–260, 2011.

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

[3] Burton H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, 13(7):422–426, 1970.

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

[5] P.T. Eugster, R. Guerraoui, S.B. Handurukande, P. Kouznetsov, and A.M. Kermarrec. Lightweight probabilistic broadcast. In *DSN*, pages 443–452, 2001.

[6] C. Fidge. Logical time in distributed computing systems. *Computer*, 24(8):28–33, August 1991.

[7] M.J. Fischer, N.A. Lynch, and M. Paterson. Impossibility of distributed consensus with one faulty process. *J. ACM*, 32(2):374–382, 1985.

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

[9] Friedemann Mattern. Virtual time and global states of distributed systems. In *Proc. of the International Workshop on Parallel and Distributed Algorithms*, pages 215–226, 1989.

[10] G. Oster, P. Urso, P. Molli, and A. Imine. Data consistency for p2p collaborative editing. In *CSCW*, pages 259–268, 2006.

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

[12] A. Schiper, J. Eggli, and A. Sandoz. A new algorithm to implement causal ordering. In *Distributed Algorithms*, volume 392 of *Lecture Notes in Computer Science*, pages 219–232. 1989.

[13] Marc Shapiro, Nuno M. Preguiça, Carlos Baquero, and Marek Zawirski. Conflict-free replicated data types. In *SSS*, pages 386–400, 2011.

[14] C. Sun, X. Jia, Y. Zhang, Y. Yang, and D. Chen. Achieving convergence, causality preservation, and intention preservation in real-time cooperative editing systems. *ACM Trans. Comput.-Hum. Interact.*, 5(1):63–108, 1998.

[15] D.B. Terry, A.J. Demers, K. Petersen, M. Spreitzer, M. Theimer, and B.B. Welch. Session guarantees for weakly consistent replicated data. In *Proc. of PDIS-94), Austin, Texas, September 28-30*, pages 140–149, 1994.

[16] Francisco J. Torres-Rojas and Mustaque Ahamad. Plausible clocks: constant size logical clocks for distributed systems. *Distrib. Comput.*, 12(4):179–195, 1999.