Compas'2022 : Parallélisme/ Architecture/ Système
MIS/UPJV - Amiens France, 5-8 juillet 2022

Improving accuracy of probabilistic-based causal broadcast

Causal broadcast is at the core of many distributed, collaborative, and database applications. It ensures that any two messages whose broadcasts are causally related are delivered respecting their broadcast order. Its implementation usually includes extra control information attached to the messages. However, it has been proved that the size of such an information grows linearly with N, the size of the system, being, therefore, not scalable in case of large-scale applications. In [11], the authors present a scalable causal broadcast algorithm using, as control information, probabilistic logical clocks, whose size does not depend on N. However, even if the broadcast provides causal delivery of messages with a high probability, some of the messages can still be delivered out of causal order. To tackle this problem, a mechanism to detect some possible delivery errors is offered. Aiming at improving such an error detection, we propose in this paper a new mechanism that, by analyzing received messages before delivering them, identifies those messages which do not have causal dependencies, not detected by the causal broadcast algorithm neither the authors' error detector.
The efficiency and overhead of our error detector were firstly evaluated theoretically and then experiments on the OMNeT++ simulator were conducted with both error detectors. Results show that our error detector outperforms the authors' one and ensures causal broadcast with a much higher probability.

1. Introduction

Distributed and parallel applications are composed of an increasing number of processes that often cooperate by acting as a single group. In order to exchange information within the group, they usually use a communication service, built on top of point-to-point send/receive networks, that offer the primitives *broadcast* and *delivery*. The first one sends a message to all members of the group and the second one delivers a received message to the application.
On the other hand, many applications also require causal broadcast ordering, i.e., any two messages, whose broadcasts are causally ordered as defined by Lamport's *happened before* relationship [8], must be delivered by all processes of the group respecting such an order. For instance, if process p_1 broadcasts a message m_1 to inform its update to a shared data and, after having delivered m_1, p_2 broadcasts m_2 with its update to the same shared data, no member of the group will deliver m_2 before m_1, i.e., processes will see the updates in the order that they causally took place. Therefore, causal broadcast, which was first introduced by Birman in the ISIS system [2], provides the same two previous primitives, respecting broadcast causality order.
Many implementations of causal broadcast exist in the literature. Several of them include control information on messages, such as logical vector clocks with one entry per process, that grows linearly with the number of processes [14][5][9] being, thus, not sustainable in large-scale systems. Others make assumptions on the network topology [6][13][3], being not suitable for dynamic systems where processes can join and leave the system.
By arguing that the probability to deliver a message out of causal order in some real systems might be quite low, Mostefaoui et al. proposed in [10] the scalable probabilistic logical clocks.
Unfortunately, even though probabilistic clocks deliver messages causally with a high probability, some of them are still delivered out of causal order, especially in systems where the above

system conditions are not always satisfied during application execution. To tackle this problem, Mostefaoui et al. present in [11] a mechanism to detect some possible delivery errors before delivering a message to the application. However, such a mechanism does not detect all out of causally ordered messages nor provides any procedure to handle those messages that it detects as out of causal order.

This paper proposes a new error detector for causal broadcast based on probabilistic clocks which, by analyzing received messages before delivering them, identifies those messages which do have causal dependencies not detected by the causal broadcast algorithm nor the error detector proposed in [11]. It also detects causal dependencies of messages tagged as not causally ordered by the latter, delivering them in causal order. Our error detector is based on hashs and presents a very high accuracy with very few mistakes.

We have theoretically analyzed our hash-based error detector. Furthermore, experiments were conducted on OMNeT++[18] simulator with our error detector as well as the one proposed in [11]. Both theoretical and experimental evaluations show that our error detector misses very few - experimentally none - out of causal ordered messages, confirming, therefore, its good accuracy. We also present performance results concerning scalability, hash computation overhead, and resilience to message load.

The paper is organized as follows. Section 2 and Section 3 respectively address some background concepts and the considered model. Our proposed procedure to retrieve a message's causal dependencies is presented and discussed in Section 4 while Section 5 describes our hash-based error detector. Performance results are presented in Section 6. Finally, Section 7 concludes the paper.

2. Background

Causal order ensures that processes deliver messages while respecting the causal relation between them, based on the happened before relation [8] introduced by Leslie Lamport :

Happened before relation : *Considering two events e_1 and e_2, e_1 causally precedes e_2, or $e_1 \to e_2$ 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 $e_1 \to e_3$ and $e_3 \to e_2$.*

If the send of m precedes the send of m', we have the formal definition of causal order : $send(m) \to send(m') \Rightarrow deliver(m) \to deliver(m')$. The delivery of a message is delayed until it is causally ordered. Therefore, a message might not be delivered at reception, and the delivery and reception of a message are considered distinct events. We define causal broadcast by applying causal order to broadcast messages and adding the condition that each message should be delivered exactly once :

Causal Broadcast *Processes deliver messages exactly once by respecting the causal relation between them. If a message m causally precedes a messages m', then all processes must deliver m before m' : broadcast(m)$\to$broadcast(m') $\Rightarrow$ deliver(m)$\to$deliver(m').*

Charron-Bost proved in [4] that logical vector clocks with one entry per process are the minimal structure required to exactly track causality. However, these clocks are not suited to large distributed systems, because assigning one entry of the vector per process does not scale. Authors of [10][16][15] propose approaches that use a vector much smaller than the number of processes in the system. Thus, they do scale, but they cannot exactly capture causality. Among them, Probabilistic clocks [10] have the best performances. Hence, this paper uses Probabilistic clocks to track causality of events probabilistically.

Probabilistic clocks track causality with a vector V of size M, with $M << N$, where N corresponds to the number of processes of the system. Each process p_i keeps a local vector V_i, whose entries are initialized to 0. Algorithm 1 describes the probablistic broadcast algorithm by process p_i.

Algorithm 1: Probabilistic broadcast at process p_i

Broadcast of message m

$\forall x \in f(i), V_i[x] = V_i[x] + 1$

$m.V = V_i$

broadcast(m)

Upon reception of message m **from** p_j

waitUntil$((\forall x \in f(j), V_i[x] \geq m.V[x] - 1) \wedge \forall k \notin f(j), V_i[k] \geq m.V[k])$

$\forall x \in f(j), V_i[x] = V_i[x] + 1$

deliver(m)

Before broadcasting a message, p_i increments those entries of its local vector clock V_i given by the function $f(p_i)$ and then attaches V_i to m. The values of such a vector result from m's causal dependencies plus the increases of the entries $V[k], k \in f(p_i)$ when p_i broadcasts m. Upon reception of a message m from p_j, p_i should wait until (1) $\forall x \in f(p_j), V_i[x] \geq m.V[x] - 1$ and (2) $\forall x \notin f(p_j), V_i[x] \geq m.V[x]$. Once the two conditions are satisfied, p_i increments the entries $k \in f(p_j)$ of its local clock V_i, and then delivers m.

Aiming at reducing the number of out of causal order deliveries, the same authors have proposed an error detector which tests the condition $\exists x \in f(p_j), V_{p_i}[x] = m.V[x] - 1$ on a message m once its delivery conditions are satisfied and before delivering it. If the condition is false, then m is delivered. Otherwise, an error handler function handles m. Note that the error detector might tag causally ordered messages as not causally ordered, i.e., the error detector might return false positives.

3. Model

We consider a set of processes $\Pi = \{p_1, p_2, \ldots, p_N\}$. Processes broadcast application messages to all processes of the system at an arbitrary rate.

Causal order of broadcasted messages is ensured by the use probabilistic clocks [10]. Each process p_i maintains a local probabilistic clock V_i of fixed size $M << N$.

Each message m is uniquely identified by the tuple (p_i, seq), where p_i is the identity of the sending process of m, and seq the sequence number that p_i assigns to m. Each message is composed of its id (p_i, seq), its attached probabilistic clock V, and the data carried by the message.

4. Handling messages tagged as not causally ordered

Whenever the error detector informs p_i that it cannot deliver a message m, broadcasted by p_j, because it might has not delivered yet all the messages that causally precede m, p_i must identify and deliver these messages before delivering m. To this end, Algorithm 2 extends Algorithm 1.

Each process p_i has the following variables :

- seq_i : sequence number of p_i's next broadcast message.
- V_i : p_i's probabilistic clock.
- Rec_i : set that contains the messages received by p_i but not delivered yet.
- $Deliv_i$: set that contains the ids (p_j, seq) of messages that p_i delivered.
- Dep_{msg_i} : set that contains the ids (p_j, seq) of causal message dependencies of p_i's next broadcast message.
- S_{dep_i} : set that contains for each message m broadcasted by p_i the tuple (seq, dep), where seq is the sequence number that p_i attributes to m and dep is m's causal dependencies.

Algorithm 2: Probabilistic broadcast by p_i with message dependency requesting

---

broadcast

1: $seq_i = seq_i + 1$
2: $S_{dep_i} = S_{dep_i} \cup \{(seq_i, Dep_{msg_i})\}$
3: $\forall e \in f(i), V_i[e] = V_i[e] + 1$
4: $m.V = V_i$
5: $m.(p, seq) = (p_i, seq_i)$
6: broadcast(<APP,m>)
7: $Dep_{msg_i} = \{(p_i, seq_i)\}$

Upon reception of <APP,m> from p_j

8: $Rec_i = Rec_i \cup \{m\}$
9: waitUntil($(\forall x \in f(p_j), V_i[x] \geq m.V[x] - 1) \wedge \forall k \notin f(p_j), V_i[k] \geq m.V[k]$)
10: **if** errorDetector(m) **then**
11:     send(<REQ,m.seq>) to p_j
12: **else**
13:     handle(p_j,m.seq)

Upon reception of <REQ,seq>) from p_j

14: send(<RSP,(seq,dep) $\in S_{dep_i}$>) to p_j

Upon reception of <RSP,seq,dep> from p_j

15: waitUntil($\forall (p_k, seq_k) \in dep, (p_k, seq_k) \in Deliv_i$)
16: handle(p_j,seq)

handle(p_j,seq_j)

17: deliver(m) : $m \in Rec_i \wedge m.(p, seq) = (p_j, seq_j)$
18: $Rec_i = Rec_i \backslash \{m\}$
19: $\forall e \in f(p_j)$, $V_i[e] = V_i[e] + 1$
20: $Dep_{msg_i} = Dep_{msg_i} \cup \{(p_j, seq_j)\}$
21: $Deliv_i = Deliv_i \cup \{(p_j, seq_j)\}$

For every message m that p_i broadcasts, it must store m's dependencies (line 2) in S_{dep_i}, in order to reply to processes whose error detector decides that m might not be causally ordered, and which will therefore request m's causal dependencies to p_i.

Upon reception of message m, p_i waits until the delivery conditions of m's probabilistic clock are satisfied (line 9), and then it executes the error detector on m (line 10) (Algorithm 2), which returns $true$ if it concludes that m might not be correctly causally ordered. Process p_i then requests m's causal dependencies by sending a request message REQ to p_j, the sender of m. The latter replies with message RSP that contains m's causal dependencies (line 14). When receiving RSP, p_i waits until it has delivered all of m's causal dependencies (line 15), then it delivers m (line 16).

5. Error Detector

Our error detector, described in Algorithm 3, is based on hashed causal dependencies. It detects out of causally ordered messages with a much higher probability than the error detector proposed in [10] (see Section 2, Algorithm **??**). Basically, a process which broadcasts a message m, computes the hash H_m of the causal dependencies of m, attaches H_m to m, and then broadcasts it. Upon reception, the destination processes compute the hash values of different sets of dependencies, aiming to find a dependency set whose hash value is equal to H_m.
Let's consider an execution from the broadcast of message m by process p_i till its delivery by p_j. First, p_i computes the hash H_m of m's causal dependencies Dep_m, attaches H_m to m and then broadcasts m. A process p_j executes the error detector on m once the delivery conditions of m are satisfied (Algorithm **??** line 4). The error detector of p_j builds dependency sets with messages that p_j has already delivered (Algorithm 3 line 1), and computes their respective hash value (Algorithm 3 line 3), in order to find a dependency set whose hash value is equal to H_m. The error detector considers that, for a set of dependencies Dep'_m with hash $H_{Dep'_m}$, if $H_{Dep'_m} = H_m$, then $Dep'_m = Dep_m$. The error detector returns false (no error, the message can be delivered) if it finds a set Dep'_m. Otherwise, it returns false. In this case, p_j must send a REQ message to p_i to request m's causal dependencies. Upon reception of the reply RSP which contains the causal dependencies Dep_m of m, p_j delivers m once it has delivered all messages $(id, seq) \in Dep_m$.
Collisions may occur when hashing dependency sets, i.e., two dependency sets may have the same hash value, which means that the error detector may find a set Dep'_m with hash $H_{Dep'_m} = H_m$, but $Dep'_m \neq Dep_m$. However, such a situation is very unlikely to happen, since a hash of x bits corresponds to a hash space of 2^x values. A parameter l bounds the number of computed hashs, because if p_j has not delivered yet a dependency of m, then the error detector would compute many hashs without finding the dependency set of m.

Algorithm 3: Hash error detector executed by p_i

1 **Input :** m : message from p_j to test
1: Comb = combinations of messages in $Deliv_i$
2: **for** $C \in Comb$ **do**
3: **if** computeHash(C)==$m.H_m$ **then**
4: return false # No error detected
5: return true # Error detected

6. Experimental results

Experiments were carried out on the OMNET++ simulator. Each process generates messages according to a Poisson-distribution with parameter δ. Messages have a propagation time following a normal distribution $N(100, 20)$. In the first experiment, we have evaluated the number of out of causal order deliveries compared to the error detector of [10] and the probabilistic clock algorithm without an error detector. Second, we measured the impact of the clock's size on the number of messages whose causal dependencies are requested. Third, we measured the impact of the number of nodes with a constant message load.

The first experiment evaluates the number of messages delivered out of causal order of the probabilist clock causal broadcast : (1) without any error detector, (2) with the error detector proposed in [10], and (3) our error detector. The experiment comprises 200 processes that broadcast a message every 2 seconds, i.e., 100 messages are broadcasted per second. The probabilistic clock has 50 entries and 100 000 messages are broadcasted during the experiment.

Table 1 gives the number of messages delivered out of causal order for each algorithm. When the algorithm uses no error detector, 507 out of 100 000 messages are delivered out of causal order Among them, the error detector proposed in [10] only detects 5 messages, i.e., 502 messages are still delivered out of causal order. On the other hand, our error detector detects all out of causal ordered messages, i.e., no message is delivered out of causal order.

Algorithm	Probabilistic	Mostéfaoui	Hash error detector
Errors	507	502	0

TABLE 1 – Messages delivered out of causal order

The second experiment evaluates the impact of the probabilistic clock size in the number of messages whose dependencies are requested. Figure 1 shows the rate of dependency requests *(number of* REQ *messages) / ((number of broadcasted messages)*(number of processes))* of our error detector with three vector clock size (20,25, and 50), when the number of messages broadcasted per second increases. Results show that the rate of dependency requests decreases when the size of the probabilistic clock increases. The reason is that the probability that a process requests the dependencies of a message m increases with the number of not detected concurrent messages to m, because those messages are taken into account when computing the dependency set of m. Increasing the size of the probabilistic clock increases the probability that concurrent messages to m are detected. Therefore, the size of the probabilistic clock should be chosen depending on the number of messages broadcasted per second in the system.

The third experiment evaluates the rate of dependency requests, i.e., *(number of* REQ *messages) / ((number of broadcasted messages)*(number of processes))*, of our hash error detector. The size of the probabilistic clock is set to 50 entries and processes increment 2 entries at each broadcast.

Table 2 shows the rate of dependency requests when varying the number of processes but keeping a constant number of 110 broadcasted messages per second. Results confirm that the rate of dependency requests does not vary much when the number of processes increases.

Request rate(10^{-4})	2.21	6.80	2.81	6.14	5.78
Number of processes	500	1000	2000	3000	5000

TABLE 2 – Request rate with a constant message load

7. Conclusion

In this paper we presented an error detector based on hashs. The proposed error detector heavily reduces the number of out of causal order delivered messages when providing causal broadcast with probabilistic clocks. Experimental results show that the presented error detector misses very few -experimentally none- not causally ordered messages and that it requests the causal dependencies of few messages.

Bibliographie

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. R. Friedman and S. Manor. Causal ordering in deterministic overlay networks. Technical report CS-2004-04, Technion - Computer Science Department, 2004.
7. 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.
8. L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7) :558–565, 1978.
9. F. Mattern. Virtual time and global states of distributed systems. In *Parallel And Distributed Algorithms*, pages 215–226, 1988.
10. A. Mostéfaoui and S. Weiss. Probabilistic causal message ordering. In *Parallel Computing Technologies - 14th International Conference, PaCT*, pages 315–326, 2017.
11. Achour Mostefaoui and Stéphane Weiss. A Probabilistic Causal Message Ordering Mechanism. Research report, LS2N, Université de Nantes, May 2017.
12. B. Nédelec, P. Molli, and A. Mostéfaoui. Causal broadcast : How to forget ? In *22nd International Conference on Principles of Distributed Systems,OPODIS*, 2018.
13. 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.
14. 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.
15. L. Ramabaja. The bloom clock. *CoRR*, abs/1905.13064, 2019.
16. F. Rojas and M. Ahamad. Plausible clocks : Constant size logical clocks for distributed systems. In *WDAG 1996*, pages 71–88, 1996.
17. Mukesh Singhal and Ajay D. Kshemkalyani. An efficient implementation of vector clocks. *Inf. Process. Lett.*, 43(1) :47–52, 1992.
18. A. Varga. The omnet++ discrete event simulation system. *Proc. ESM'2001*, 9, 2001.

A. Experimental results

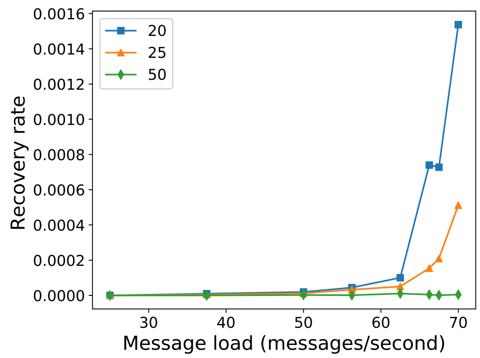

Figure 1 : Request rate following the number of messages broadcasted per seconds for different vector clock sizes