The Pitfalls in Achieving Tagged Causal Delivery

Georges Younes, Paulo Sérgio Almeida and Carlos Baquero [*]
HASLab / INESC TEC
Universidade do Minho
Braga, Portugal

ABSTRACT

Causal delivery middleware respect causality when delivering messages, but do not provide information about it to the client process. When two messages, m_1 and m_2 are delivered in sequence to a given process, either they are concurrent or m_1 happens-before m_2, but the client application is not told which is the case. In some domains, like operation-based CRDTs, this information is useful, and previously we have proposed exposing it in the API of a Tagged Causal Delivery middleware. One could think this would be just a matter of taking some current middleware and exposing the vector-clocks which are internally kept. In this paper we identify some obstacles in doing so, and describe how to overcome them. The essence lies in the role of current middleware, allowing it to tag as ordered some messages that are concurrent, and we describe how it can happen in several interaction models between middleware and client code, either callback-based or with independent threads/processes. This means that vector-clocks in current middleware are not precise to describe happens-before, and cannot be simply exposed in the API.

CCS CONCEPTS

•**Theory of computation** → *Distributed algorithms;*

KEYWORDS

Causal delivery; Tagged Causal Delivery; Operation-based CRDTs; Distributed Computing

1 CAUSAL DELIVERY

Classical Causal delivery has been widely and frequently used in distributed systems [2]. Introduced by Birman in [3, 4] as an abstraction used to guarantee that messages are delivered in an order respecting happens-before causality relation. For any processes i, j, k, if an event $\mathsf{send}_i(m1) \rightarrow \mathsf{send}_j(m2)$, the causal delivery service guarantees that $\mathsf{deliver}_k(m1) \rightarrow \mathsf{deliver}_k(m2)$ in the same process k, where the happens-before ($\rightarrow$) relation is the one introduced by Lamport in [5]. Concurrent operations can be delivered in different orders in different processes, but whether two messages are concurrent or causally related, this knowledge is not provided by the service to the client application. This may be enough for some applications, that just need a combination of exactly-once delivery and session guarantees, e.g., a replicated commutative data-type, where concurrent operations commute, producing the same outcome regardless of the order in which they are applied.

2 TAGGED CAUSAL DELIVERY

For other applications, e.g., conflict-free replicated data-types [6] having non-commutative data-type operations, the semantics is defined based on the happens-before relation between mutator operations. Let us consider the case of an Add-Wins Set, where the mutator operations are $\mathsf{add}(e)$ (adding an element to the set) and $\mathsf{rmv}(e)$ (removing an element from the set). The remove operation would override an add (of the same element) that happened-before it, but if the operations are concurrent, the add wins, making the element present in the set. To implement such a replicated data-type, the happens-before relation between mutator operations is needed.

The implementor of such data-types, tends to rely on off-the-shelf causal delivery middleware, to obtain an exactly-once delivery respecting happens-before. However, the happens-before information, which is needed by the data-type, is not exposed in the causal delivery middleware API. This makes the data-type implementation explicitly add timestamping information (e.g., vector-clocks or globally-unique tags) to the state and messages in order to track the happens-before relation. This means more data-type state and more information in the message payload, which is a waste and a duplication of effort, considering that the causal-delivery middleware must be somehow tracking happens-before, but not exposing it to the client application.

To improve efficiency and spare data-type implementors from a duplication of effort, we have previously proposed [1] exposing in the causal delivery API the operation timestamps that characterize happens-before. This allowed more elegant and efficient operation-based CRDTs to be defined.

3 THE PITFALLS OF EXPOSING MIDDLEWARE TIMESTAMPS

At first glance, it might seem trivial to implement tagged causal delivery, by using any classical causal delivery middleware service and exposing to the client code the timestamps (e.g., vector-clocks) that are used internally to ensure causal delivery. However, when considering concrete implementations, some unexpected problems arose. We show how naively exposing the timestamps would lead to an incorrect characterization of causality, in either of the two typical interaction models between middleware and client code: callback-based and with independent threads/processes.

Callback-based. In an event-driven architecture with a single process, the application code runs as callbacks invoked from the middleware code when messages need to be delivered to the application logic to be processed, e.g., $\mathsf{deliver}(m, t)$ for message m

[*]The research leading to these results has received funding from the European Union's Horizon 2020 - The EU Framework Programme for Research and Innovation 2014-2020, under grant agreement No. 732505, project LightKone.

tagged by *t* timestamp. To avoid reentrancy problems, when a send is invoked inside the deliver callback, the send simply adds the message to a queue, to be handled by middleware code when the callback finishes. It can happen that the middleware has a set of messages ready to be delivered, and invokes the deliver callback for each one, before handling sends which have been enqueued. If the middleware creates timestamps for messages to be sent only upon dequeing them, then a message will be tagged as causally in the future of all messages that were delivered after the send action by client code and before dequeing occurred. This means that some messages that are actually concurrent are tagged as causally related, making timestamps reflect a larger relation than happens-before, over-ordering some events. While this does not break causal delivery, it means that these timestamps cannot be exposed as precisely characterizing happens-before.

Independent threads/processes/actors. In other architectures we have two independent processes, a client process and a middleware process. Here, in addition to the queue of messages to be sent, as above, we will typically also have a queue of messages ready to be delivered. The middleware tags and enqueues messages to the deliver queue, while the client dequeues and processes them. When doing a send, the client enqueues a message to the send queue. This message will be tagged by the middleware process as in the future of other messages not yet delivered by the client (namely, those that are still in the delivery queue but have already been tagged), when they are in fact concurrent. Note that what defines the happens-before is the total order of send and deliver events as observed by each client process; other events, e.g., when a message was enqueued or dequeued by the middleware process, are irrelevant (i.e., events which happen in the system, but invisible to the API).

4 CORRECT TAGGING OF HAPPENS-BEFORE

To correctly characterize happens-before, a message being sent must be tagged reflecting the causal knowledge according to all delivery events at the client, up to the send event (at the client).

In the single process callback-based model, this can be achieved by making the middleware update the causal timestamp (e.g., vector-clock) just before invoking each deliver callback, and either making the send function tag the message before enqueuing, or making the middleware process all messages enqueued to be sent before invoking the next deliver callback.

As for the two independent processes model, the client process will need to maintain the causal timestamp, update it at each deliver event and use it to tag each message. The middleware process keeps a causal timestamp as before and uses it for the causal delivery order. The only difference is that now messages are tagged at the client process and not at the middleware process.

5 A WELCOME SIDE-EFFECT

In classical causal delivery middleware, some concurrent messages are tagged as in happens-before relation to each other. This unnecessary over ordering, while not contradicting causal delivery, has the adverse effect of inducing extra delays on delivery, making some message wait for another when it should be possible to deliver it earlier.

In tagged causal delivery middleware, which characterizes happens-before in a precise way, as we discussed above, each message will be able to be delivered as early as semantically possible.

6 CONCLUSION

In this short paper, we address the fact that some applications require happens-before causality information, and would benefit from causal delivery middleware which exposed such information, what we have called tagged causal delivery, and present the lessons from actually implementing such a middleware. Although starting from some classic causal delivery middleware and adpating it to achieve correct tagging is not difficult, if proper care is not taken one can easily end up with an incorrect implementation. We present the reasons of why simply exposing current internal tags will not work and provide a solution for a correct implementation of tagged causal delivery. Also, we would like to note that an Erlang implementation of this tagged causal delivery middleware exists publicly at https://github.com/gyounes/trcb_base.

REFERENCES

[1] Carlos Baquero, Paulo Sérgio Almeida, and Ali Shoker. 2014. Making operation-based CRDTs operation-based. In *IFIP International Conference on Distributed Applications and Interoperable Systems*. Springer, 126–140.

[2] Kenneth Birman, Andre Schiper, and Pat Stephenson. 1991. Lightweight causal and atomic group multicast. *ACM Transactions on Computer Systems (TOCS)* 9, 3 (1991), 272–314.

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

[4] Kenneth P Birman, Robbert van Renesse, and others. 1994. *Reliable distributed computing with the Isis toolkit.* Vol. 85. IEEE Computer society press Los Alamitos.

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

[6] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. 2011. *A comprehensive study of Convergent and Commutative Replicated Data Types.* Technical Report. 50 pages. http://hal.upmc.fr/inria-00555588/