The Intrinsic Cost of Causal Consistency

Albert van der Linde, Pedro Fouto, João Leitão, Nuno Preguiça
NOVA LINCS DI, FCT, Universidade NOVA de Lisboa

Abstract

In the last few years, causal consistency has become a popular consistency model for geo-replicated databases. The algorithms proposed to enforce causal consistency typically associate with each operation some metadata, which is used to guarantee that an operation is not executed if its execution would break causality. This may lead to the impression that causal consistency is intrinsically costly and non scalable.

In this paper, we analyze the metadata costs of enforcing causal consistency and put these costs in perspective, considering the metadata that is necessary to enforce reliability. We show that by wisely ordering the propagation of operations it is possible to enforce causal consistency without any additional metadata other than the already necessary to enforce reliability.

CCS Concepts: • **Computer systems organization** → *Peer-to-peer architectures*; **Distributed architectures**; *Reliability*.

Keywords: Causal consistency, Reliability

ACM Reference Format:
Albert van der Linde, Pedro Fouto, João Leitão, Nuno Preguiça. 2020. The Intrinsic Cost of Causal Consistency. In *7th Workshop on Principles and Practice of Consistency for Distributed Data (PaPoC '20), April 27, 2020, Heraklion, Greece.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3380787.3393674

1 Introduction

Causal consistency [18] has gained attention as a consistency model for geo-replicated databases in the last few years, as it has been shown that causal consistency is the strongest consistency model that can be enforced in an highly available way [3]. A large number of algorithms have been proposed recently in literature to enforce causal consistency [1, 7, 11, 12, 18]. In the past there were also works that have studied the problem of executing operations in causal order [15]. In fact, enforcing causal consistency is equivalent to enforcing that operations are delivered (and executed) in a causal order across all replicas. [1]

In general, to enforce causal consistency, algorithms use additional metadata – typically added to each operation. When an operation is received, this metadata is used to guarantee that the execution of the operation will not violate causal consistency, delaying the execution of the operation if necessary.

Charron-Bost in 1991 [9] proved that, to provide the ability to check for concurrency in causal consistency (i.e., characterising causality or verifying causal independence), the minimum necessary metadata attached to any operation is on the order of the amount of nodes which can apply writes to the system state. Intuitively, this is because n is the upper bound of the dimensions of a computation distributed over n nodes (the global execution lattice). Interestingly, to provide causal consistency, being able to characterise causality is not at all important, as long as operations are delivered respecting their causal order [16, 26, 29].

In a system that aims to provide causal consistency, a more basic property which needs to be enforced is reliability: guaranteeing that every operation submitted is eventually executed in every replica. To enforce reliability, algorithms also need to use some metadata.

In this paper, we analyse the cost of metadata to enforce causal consistency knowing that some metadata already needs to be used to enforce reliability. We show that it is possible to enforce causal consistency in peer-to-peer networks, where nodes can synchronise pairwise, with only the metadata which is already necessary to enforce reliability.

2 Definitions and system model

Causal consistency is a consistency model that can be described, at a high level, as enforcing all replicas to always observe a state that respects the happens-before relationships among operations [16]. Essentially, considering any two operations o_1 and o_2 such that $o_1 \prec o_2$, where $\prec$ is the partial order that encodes the happens-before relationship, causal consistency forbids any replica to observe the effects of o_2 without observing the effects of o_1.

We say that an operation o_1 happened before operation o_2, $o_1 \prec o_2$, iff o_2 was generated in some replica n while o_1 had already been executed in n. For a set of operations Ops, this defines a partial order among operations $(Ops, \prec)$.

We say that for a set of operations Ops, $O_i = (Ops, <)$ is a causal serialization of $O = (Ops, \prec)$ iff O_i is a linear

[1] To include convergence, Causal+ consistency, although not *equivalent*, only requires that operations are delivered in causal order in all replicas.

PaPoC '20, April 27, 2020, Heraklion, Greece

ACM ISBN 978-1-4503-7524-5/20/04…$15.00
https://doi.org/10.1145/3380787.3393674

extension of O, i.e., $\forall o_1, o_2 \in Ops, o_1 \prec o_2 \Rightarrow o_1 < o_2$. A system enforces causal consistency iff, across all replicas, operations are executed according to a causal serialization.

Multiple algorithms have been proposed to enforce causal consistency (or implement causal dissemination)[2, 4, 8, 13, 16, 19, 27, 29]. Two of the most popular techniques consist in using version vectors [20, 21] and direct dependency graphs [22, 23]. In the former, the dependencies of each operation are summarized in a vector that states which operations generated at each site happened before a given operation. Using direct dependencies, each operation includes information on the concurrent operations that have been executed before their generation. By leveraging on the transitivity of dependencies, it is possible to build the complete dependency graph of an operation using only its direct dependencies.

2.1 Model

For the remainder of this paper we consider a distributed system composed of n nodes, where each node maintains the full state of the database (unless mentioned otherwise). Nodes can fail by crashing, but they do not behave arbitrarily.

When discussing algorithms, we call *local* operations to the operations that were submitted in a given node (operations created at that node), and *remote* operations to the operations submitted in any of the other nodes (those received through the network).

3 Causal consistency with a central node

We start by observing that a simple way to enforce causal consistency is to use a central node for propagating operations – this approach is used, for example in CVS and subversion. A simple algorithm could work as follows (depicted in Figure 1).

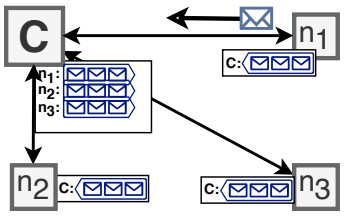

Figure 1. Central node approach.

Each node has a FIFO channel with the central node. Every operation executed in a node is propagated to the central node using the FIFO communication channel.

The central node receives the operations from each channel in order, and adds it to the outgoing queue of every other node atomically (in partial replication scenarios, it is only necessary to put the operation in the queue of nodes interested in the operation). Operations are propagated to nodes asynchronously using the FIFO channel.

This algorithm enforces causal consistency because: (i) operations submitted in a replica execute in the same order across all replicas, as they are propagated through FIFO channels and processed in the order in which they are received; and (ii) when an operation is submitted in a replica, all remote operations known locally are, at least, in the outgoing queues to all other nodes, thus guaranteeing that they will be executed before the operation that is currently being submitted.

In this case, no specific metadata is necessary to enforce causal consistency in a run without faults. Considering that nodes and channels can fail, we would need some metadata to guarantee reliability and causal consistency. Independently of the failure recovery algorithm, it seems clear that the recovery process would need to determine if a given operation had already been propagated or not, for which it would need operations to be identified with a unique identifier. Lamport clocks [17] can be used to create unique identifiers (for example, composed of the pair `replica:timestamp`) that would allow to enforce reliability while also being the base to enforce causality without any additional metadata.

Recovering from faults can be performed as follows. For faults in the channels, when creating a replacement channel, nodes start by exchanging the identifier of the last message they have received. Each node resumes sending messages in the queue for the peer starting with the message following the last received message from the peer. For faults in nodes, in a crash-recovery model, when a node recovers with its previous state, it only needs to resume the propagation of channels. This can be done by executing the previous channel recovery process for every channel it has. For recovering from a definite fault of the central node, each node can replay its log – when receiving an operation, all nodes (including the central node) discard operations they have already received.

4 From a star topology to a dissemination tree

When reasoning why using a central node is sufficient to enforce causal consistency, we can conclude that it is due to the fact that when an operation goes through the central node, all of its dependencies had already been propagated by that central node to all other communication channels (or added to the respective queues).

It is possible to extend this idea and, instead of using a central node connected to every other node for propagating operations, to use a dissemination tree connected by FIFO channels (as depicted in Figure 2) [2]. A node receives operations from each of its channels in order. When a node receives an operation, it atomically both delivers the operation locally and puts it in the outgoing queues of every other channel. Additionally, the creation of a new local operation

[2]The overhead of maintaining a tree, possibly per partition, is orthogonal to the meta-data cost of reliability.

leads to it being atomically added to the outgoing queue of every channel that node has.

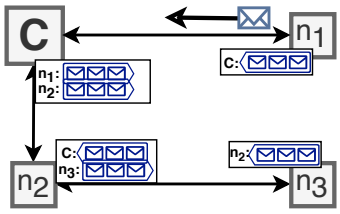

Figure 2. Dissemination tree.

This guarantees that when an operation is added to a channel's queue, all of its dependencies have already been propagated through the channel (in one or the other direction) or are queued ahead of that operation in the channel. Saturn [7] uses this approach to enforce causal consistency in partially replicated databases, by propagating operations through all channels that will reach nodes interested in the operations[3].

In this case, again, we do not need any metadata to enforce causal consistency in a run without faults – the way messages are propagated guarantees that they will be received in causal order. Recovering from faults would be more complex than in the central node scenario, but similar techniques could be used relying on Lamport clocks.

5 Peer-to-peer networks

We now consider the more general case where any pair of nodes can communicate with each other to propagate operations. Two classical approaches are used to enforce causal consistency in this setting.

In the first, proposed by Lamport, operations are tagged using a Lamport clock and an operation can only be executed after it is known that there is no operation to be received with a smaller Lamport clock. If every node communicates with every other node directly, and nodes propagate local operations in order, when a node n_1 receives an operation with clock t from n_2, it knows that it has already received all operations with clocks smaller than t from n_2. This approach, depicted in Figure 3, does not need any specific information to enforce causal consistency, but it requires every node to communicate with every other node to execute a stability process which can take long.

An alternative approach is to use vector clocks, where every operation includes a vector clock that records the exact operations an operation depends on (depicted in Figure 4). When receiving an operation, a node can locally verify if all dependencies are satisfied and, if not, it knows exactly which operations are missing. When compared with the previous approach, this trades having specific metadata to enforce causal consistency for being faster in determining when it is safe to execute an operation.

[3]Saturn actually only propagates unique identifiers of the operations through the channels, and propagates operations directly among nodes, but for this discussion this implementation aspect is not relevant.

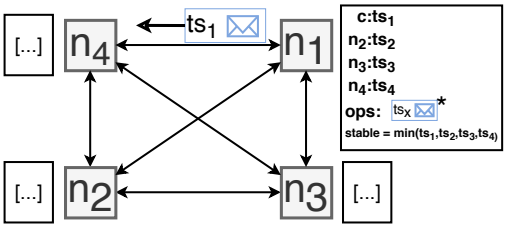

Figure 3. Lamport clocks and stability.

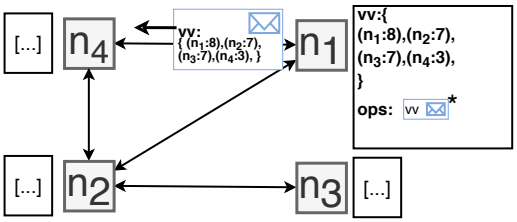

Figure 4. Version vectors.

Direct dependencies can be used as a compressed history instead of using version vectors which can become large when multiple replicas are able to create operations. Each operation is tagged with its direct dependencies – the last operations to have been locally applied which are concurrent among each other. As depicted in Figure 5, instead of maintaining a version vector, a dependency graph is stored at every node.

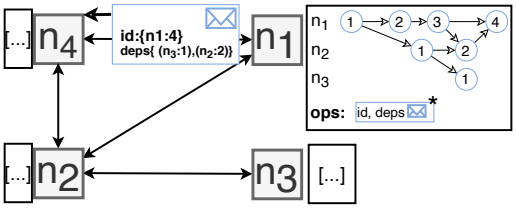

Figure 5. Direct dependencies.

Optimisations and garbage collection. Each of the previous mechanisms can be augmented with garbage collection. Building on the stability notion of Lamport clocks, many algorithm proposals exist for providing the notion of stability for both version vectors and direct dependencies [5, 6, 22, 24, 25, 28].

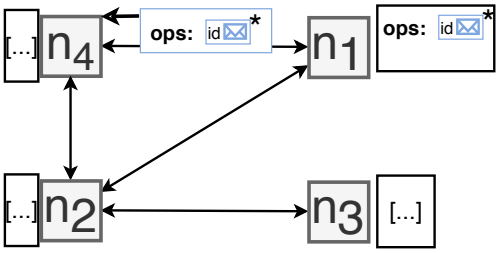

Figure 6. Initial algorithm.

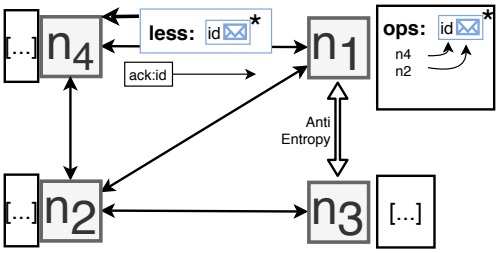

Figure 7. Optimisations.

5.1 Algorithm

We now show that it is possible to avoid executing complex stability processes or having additional metadata to enforce causal consistency. The only cost of additional message overhead is the one already required to provide reliable delivery.

In our algorithm, every node keeps an ordered list of operations it has previously executed. The key idea is that the list of operations maintained in each node respects causality, i.e., all dependencies of an operation o appear before it in the list. We start by presenting a non-optimised version of the algorithm (Figure 6) and then discuss possible ways to optimise it (Figure 7).

When an operation is created at a node, the operation is appended to the list. This maintains the list causally ordered with respect to locally created operations.

One node communicates with some other node by sending it the full ordered list of operations. When receiving a list of operations from a remote node, the node iterates through the list in order and for each operation, if it is not in the local list, it appends the operation to the local list. The local list thus remains causally ordered, as when an operation is added to the local list, all operations which appear before in the received list are already in the local list. Thus, any dependency of the operation is necessarily already satisfied.

For executing this algorithm, it is only necessary to be able to check if an operation is already in a list, for which we need to assign a unique identifier to an operation. This algorithm tolerates network faults. It also allows a node to fail and recover if it maintains its list in stable storage. We note that this information would be also necessary to enforce reliable execution of operations. Thus, causal consistency is adding a grand total of zero additional metadata.

This algorithm has another interesting property: it is possible to add new nodes to the network, at any moment and in a decentralised way, which is not the case for algorithms that need to execute a stability process. The algorithm described is not new, in fact a very similar causal ordering mechanism for exchanging messages is presented in [25]. Our adaption suffices to make the point we wish to make.

5.2 Optimizing the algorithm

Although from the theoretical point of view the previous algorithm has interesting properties, propagating all messages in every synchronisation is not acceptable in practice. It is clear that when a node propagates its full list of operations to a remote node, it actually only needs to propagate the operations that are still not known by the remote node (as already known operations are ignored when they are received – i.e., the list does not keep duplicates). Several techniques can be used to minimize the operations to send. We present simple adaptions in Figure 7 which we describe next.

First, when a node sends operations to a remote node, and the reception is acknowledged, it can locally record that information – for each remote node, it would suffice to maintain the last position of the local list that was acknowledged remotely. Thus, each node will send each operation only once to a remote node (in lieu of failures). Second, when a node receives an operation from a remote node, if it already knows the operation, it can also record that the remote node already has that operation.

Previous works [6, 25] implementing such mechanisms require every node to store information about every other node. Global knowledge about every node is impractical and a solution which permits any pair of nodes to efficiently communicate is preferred. Our algorithm only keeps such information for every connected node, but still needs to efficiently handle new connections (or recover from failures).

When two nodes synchronise they could start by propagating a summary of locally known operations – e.g., using a vector clock as in anti-entropy epidemic communication [10]. This last optimisation is typically more interesting when synchronising with a node for the first time (or after a long period without direct communication). In systems with a large number of nodes, it is possible to use mechanisms to minimize the size of vectors transmitted [14].

We note that if we want to enforce reliability in the same setting, similar techniques must be used. Thus, when enforcing reliability we can enforce causal consistency without any additional overhead by only carefully deciding the order in which operations are propagated.

6 Conclusion

In this paper we explored the different communication models that existing causality tracking solutions employ: using a centralised node, a dissemination tree, or a general peer-to-peer model. We show, for each communication model, that the metadata overhead required to enforce reliability is enough to also provide causal consistency. For the most common scenario, where every node can communicate with any other, we also present an overview of an algorithm that supports our claim.

Overall we show that, for a system that already aims to provide reliable delivery, the intrinsic cost of providing causal consistency is zero, further motivating causal consistency as a practical and adequate consistency model.

Acknowledgments

This work was partially supported by FCT (Fundação para a Ciência e a Tecnologia - Portugal), through SAMOA (Project #PTDC/CCI-INF/32662/2017) and PhD scholarships awarded to Albert van der Linde (SFRH/BD/117446/2016) and Pedro Fouto (SFRH/BD/143668/2019).

References

[1] Deepthi Devaki Akkoorath, Alejandro Z Tomsic, Manuel Bravo, Zhongmiao Li, Tyler Crain, Annette Bieniusa, Nuno Preguiça, and Marc Shapiro. 2016. Cure: Strong semantics meets high availability and low latency. *Distributed Computing Systems (ICDCS), 2016 IEEE 36th International Conf. on*, 405–414.

[2] Sérgio Almeida, João Leitão, and Luís Rodrigues. 2013. ChainReaction: a causal+ consistent datastore based on chain replication. In *Proceedings of the 8th ACM European Conference on Computer Systems*. ACM, 85–98.

[3] Hagit Attiya, Faith Ellen, and Adam Morrison. 2015. Limitations of Highly-Available Eventually-Consistent Data Stores. In *Proceedings of the 2015 ACM Symposium on Principles of Distributed Computing* (Donostia-San Sebastián, Spain) *(PODC '15)*. ACM, New York, NY, USA, 385–394. https://doi.org/10.1145/2767386.2767419

[4] Carlos Baquero and Nuno Preguiça. 2016. Why logical clocks are easy. *Commun. ACM* 59, 4 (2016), 43–47.

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

[6] 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.

[7] Manuel Bravo, Luís Rodrigues, and Peter Van Roy. 2017. Saturn: A Distributed Metadata Service for Causal Consistency. In *Proceedings of the Twelfth European Conference on Computer Systems* (Belgrade, Serbia) *(EuroSys '17)*. ACM, New York, NY, USA, 111–126. https://doi.org/10.1145/3064176.3064210

[8] Manuel Bravo, Luís Rodrigues, and Peter Van Roy. 2017. Saturn: A distributed metadata service for causal consistency. In *Proceedings of the Twelfth European Conference on Computer Systems*. ACM, 111–126.

[9] Bernadette Charron-Bost. 1991. Concerning the size of logical clocks in distributed systems. *Inform. Process. Lett.* 39, 1, 11–16.

[10] Alan Demers, Dan Greene, Carl Hauser, Wes Irish, John Larson, Scott Shenker, Howard Sturgis, Dan Swinehart, and Doug Terry. 1987. Epidemic Algorithms for Replicated Database Maintenance. In *Proceedings of the Sixth Annual ACM Symposium on Principles of Distributed Computing* (Vancouver, British Columbia, Canada) *(PODC '87)*. ACM, New York, NY, USA, 1–12. https://doi.org/10.1145/41840.41841

[11] Jiaqing Du, Sameh Elnikety, Amitabha Roy, and Willy Zwaenepoel. 2013. Orbe: Scalable Causal Consistency Using Dependency Matrices and Physical Clocks. In *Proceedings of the 4th Annual Symposium on Cloud Computing* (Santa Clara, California) *(SOCC '13)*. ACM, New York, NY, USA, Article 11, 14 pages. https://doi.org/10.1145/2523616.2523628

[12] Jiaqing Du, Călin Iorgulescu, Amitabha Roy, and Willy Zwaenepoel. 2014. GentleRain: Cheap and Scalable Causal Consistency with Physical Clocks. In *Proceedings of the ACM Symposium on Cloud Computing* (Seattle, WA, USA) *(SOCC '14)*. ACM, New York, NY, USA, Article 4, 13 pages. https://doi.org/10.1145/2670979.2670983

[13] Robert Escriva, Ayush Dubey, Bernard Wong, and Emin Gün Sirer. 2014. Kronos: The design and implementation of an event ordering service. In *Proceedings of the Ninth European Conference on Computer Systems*. ACM, 3.

[14] Richard A Golding. 1992. *WEAK-CONSISTENCY GROUP COMMUNICATION AND MEMBERSHIP (Ph.D. Dissertation)*. Technical Report. Santa Cruz, CA, USA.

[15] Rivka Ladin, Barbara Liskov, and Liuba Shrira. 1990. Lazy Replication: Exploiting the Semantics of Distributed Services. In *Proceedings of the Ninth Annual ACM Symposium on Principles of Distributed Computing* (Quebec City, Quebec, Canada) *(PODC '90)*. ACM, New York, NY, USA, 43–57. https://doi.org/10.1145/93385.93399

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

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

[18] Wyatt Lloyd, Michael J. Freedman, Michael Kaminsky, and David G. Andersen. 2011. Don'T Settle for Eventual: Scalable Causal Consistency for Wide-area Storage with COPS. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles* (Cascais, Portugal) *(SOSP '11)*. ACM, New York, NY, USA, 401–416. https://doi.org/10.1145/2043556.2043593

[19] Wyatt Lloyd, Michael J Freedman, Michael Kaminsky, and David G Andersen. 2011. Don't settle for eventual: scalable causal consistency for wide-area storage with COPS. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*. ACM, 401–416.

[20] Friedemann Mattern. 1989. Virtual Time and Global States of Distributed Systems. In *Parallel and Distributed Algorithms*. North-Holland, 215–226.

[21] D. S. Parker, G. J. Popek, G. Rudisin, A. Stoughton, B. J. Walker, E. Walton, J. M. Chow, D. Edwards, S. Kiser, and C. Kline. 1983. Detection of Mutual Inconsistency in Distributed Systems. *IEEE Trans. Softw. Eng.* 9, 3 (May 1983), 240–247. https://doi.org/10.1109/TSE.1983.236733

[22] Larry L Peterson, Nick C Buchholz, and Richard D Schlichting. 1989. Preserving and using context information in interprocess communication. *ACM Transactions on Computer Systems (TOCS)* 7, 3 (1989), 217–246.

[23] Ravi Prakash, Michel Raynal, and Mukesh Singhal. 1996. An efficient causal ordering algorithm for mobile computing environments. In *Proceedings of 16th International Conference on Distributed Computing Systems*. IEEE, 744–751.

[24] Michel Raynal, André Schiper, and Sam Toueg. 1991. The causal ordering abstraction and a simple way to implement it. *Information processing letters* 39, 6 (1991), 343–350.

[25] André Schiper, Jorge Eggli, and Alain Sandoz. 1989. A new algorithm to implement causal ordering. In *International Workshop on Distributed Algorithms*. Springer, 219–232.

[26] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. 2011. A comprehensive study of convergent and commutative replicated data types.

[27] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. 2011. Conflict-Free Replicated Data Types. In *Stabilization, Safety, and Security of Distributed Systems*, Xavier Défago, Franck Petit, and Vincent Villain (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 386–400.

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

[29] Albert van der Linde, Pedro Fouto, João Leitão, Nuno Preguiça, Santiago Castiñeira, and Annette Bieniusa. 2017. Legion: Enriching Internet Services with Peer-to-Peer Interactions. In *Proceedings of the 26th International Conference on World Wide Web* (Perth, Australia) *(WWW '17)*. International World Wide Web Conferences Steering Committee, Republic and Canton of Geneva, Switzerland, 283–292. https://doi.org/10.1145/3038912.3052673