On combining fault tolerance and partial replication with causal consistency

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

Abstract

The purpose of this paper is to discuss the limitations imposed by introducing fault-tolerance in a partial replication system which aims to provide causal consistency.

The general outcome is that, to provide support for indefinite replica-failure, the notion of partial in partial replication becomes not-really-partial-at-all. We prove that to implement causal consistency when indefinite replica-failures are possible, it is impossible to respect the concept of genuine partial replication – not storing or propagating operations which are on objects a given replica does not replicate locally.

In our initial approach to tackle this issue client replicas need only to replicate the operations they depend on which have not yet been marked as durable by the centralised component. We discuss remaining limitations and expected improvements in future work.

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

Keywords: Causal consistency, Fault tolerance, Partial replication

ACM Reference Format:
Albert van der Linde, Diogo Serra, João Leitão, Nuno Preguiça. 2020. On combining fault tolerance and partial replication with 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, 5 pages. https://doi.org/10.1145/3380787.3393684

1 Motivation

When attempting to create a partial replication algorithm with inter-object causal consistency we encountered a clear problem – what happens to dependencies on operations respective to objects which are not locally replicated, if any given node can fail indefinitely.

Publication rights licensed to ACM. ACM acknowledges that this contribution was authored or co-authored by an employee, contractor or affiliate of a national government. As such, the Government retains a nonexclusive, royalty-free right to publish or reproduce this article, or to allow others to do so, for Government purposes only.

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

© 2020 Copyright held by the owner/author(s). Publication rights licensed to ACM.

ACM ISBN 978-1-4503-7524-5/20/04…$15.00
https://doi.org/10.1145/3380787.3393684

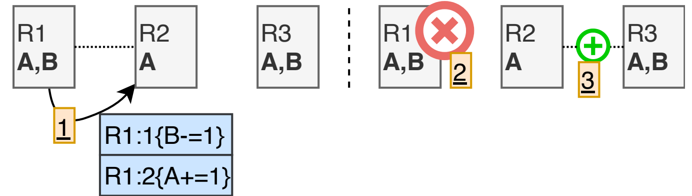

Figure 1. The problem of lost operations.

A motivating example can be found in Figure 1, where replicas share counter CRDTs A and B. [1] What happens if a replica $R2$ only has interest in replicating object A, but its direct neighbour creates operations for both objects A and B? The question is, will $R2$ have to replicate operations on B, or can it just ignore such operations?

Consider that replicas $R1$ and $R2$ are connected and exchanging operations. Replica $R1$ creates its first operation, $(R1 : 1)$, decrementing B and later creates $(R1 : 2)$ incrementing A – notice that the latter operation depends on the former.

After $R1$ propagates these operations to $R2$ (1), the connection between replicas $R1$ and $R2$ fails (2) and, afterwards, a new connection between $R2$ and $R3$ is established (3).

$R2$ and $R3$ will now attempt to synchronise state, specifically, propagating to each other any operations on A not known to the other. As $R2$ sends operation $(R1 : 2)$ to $R3$, $R3$ discovers that it depends on $(R1 : 1)$. [2] The problem is that $R1$ might have failed, and the operation $(R1 : 1)$ can be lost forever if there is no special care taken here.

There are two cases which could happen:

- $R2$ stored nothing about B, leading to operations on A to miss dependencies on B, and therefore $(R1 : 2)$ can not be executed at $R3$;
- $R2$ kept all operations on B where operations on A it has executed on depend – correctly synchronises as $R2$ can simply propagate $(R1 : 1)$ along with $(R1 : 2)$.

Notice that if replica $R1$ fails and never recovers, and replica $R2$ stored nothing about B, then replica $R2$ will never be able to fully synchronise with all other replicas due the lost dependency.

[1] The issue is not limited to counters nor CRDTs – any data-type requiring causal delivery suffers from the same problem. The reader can find a comprehensive list of CRDT specifications in [17].
[2] Using some mechanisms for dependency tracking – such as keeping direct dependencies [12, 13] or version vectors [10, 11]).

2 Impossibility result

This section proves that it is impossible to tolerate non-recoverable failures when using genuine partial replication with causal consistency. For the purpose of this paper, genuine only requires replicas not to store data, operations, or meta-data of objects a replica does not replicate directly.

2.1 System model

We consider a system comprised of R replicas. The global state of the system contains O objects, where each replica $r \in R$ replicates a subset S_r of objects ($S_r \subseteq O$).

Communication. We assume that any replica r_1 can communicate with any other replica r_2. Communication happens in two phases:

synchronisation – the initial phase of synchronising both replica's objects;

propagation – when synchronisation is complete, replicas propagate operations directly.

Failures. Any replica can fail, temporarily due to, for example, networking or indefinitely due crash (i.e., fail-stop). It will be clear later in this paper why we include indefinite failures without the possibility to recover. Additionally, assume that replicas do not behave arbitrarily – all replicas follow system specification without any deviation.

2.2 Definitions

2.2.1 Causal consistency. 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 [8]. 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 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)[1, 2, 5, 7–9, 18, 20]. Two of the most popular techniques consist in using version vectors [10, 11] and direct dependency graphs [12, 13]. 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.

2.2.2 Partial replication. An interesting topic especially when considering consistency semantics not per-object, but with respect to all objects. For example, if a system aims to only provide per-object causal consistency, then the motivating example in Figure 1 in fact poses no issue at all – individual objects can trivially be synchronised as no verification has to be done for operations over other objects. Inter-object causal consistency on partial replication is, to the best of our knowledge, still not a fully solved problem.

2.2.3 Synchronisation. Defined as two replicas being able to synchronise their local states (converge) with each other with respect to all objects they have in common. As our focus is on partial replication, any pair of replicas will have different subsets of objects in their replication sets. We consider the synchronisation phase among two replicas successful when all objects in the intersection of the replication sets of both replicas converge to a single unified value.

For example in Figure 1 this means replica that $R2$ synchronises A with both of the other replicas, while these will synchronise both objects A and B directly among them.

2.3 Synchronising state with inter-object causality

We follow the example of Figure 1 directly to prove that causal consistency cannot be provided with genuine partial replication when replicas can fail indefinitely.

Theorem 1. *A system cannot provide both causal consistency and genuine partial replication if any replica can fail indefinitely.*

Proof. Let's assume any two replicas, $r1$ and $r2$ are able to synchronise among them. Replica r_1 replicates objects S_{r_1}, r_2 replicates objects S_{r_2}. The intersection of both replica's replication sets is $I_{1,2} = S_{r_1} \cap S_{r_2}$.

For these two replicas to be synchronised, the final values of all objects in the set $I_{1,2}$ must be equal. I.e., $\forall_{o \in I_{1,2}} (r_1.o.state = r_2.o.state)$. [3]

For r_1 and r_2 to be able to synchronise correctly, two things must happen:

- every operation op applied in r_1 such that $op.object \in I_{1,2}$ must be also applied at r_2;
- every operation op applied in r_2 such that $op.object \in I_{1,2}$ must be also applied at r_1.

Any mechanism which propagates the relevant missing operations in an order respecting causality suffices [1–5, 7–12, 14, 16, 18–20].

[3] Save for concurrent reception or creation of new operations but a similar proof can be devised for such scenarios.

The third replica. Assume there is a replica r_3 with replication set S_{r_3}, such that $S_{r_2} \subsetneq (S_{r_1} \cap S_{r_3})$. I.e., replicas r_1 and r_3 replicate at least one object which replica r_2 does not have in its replication set.

The set of objects which r_2 is missing is defined by $M = I_{1,3} \setminus S_{r_2}$. The set of objects which all replicas have is defined as $I_{1,2,3}$.

Consider objects $m \in M$, and $i \in I_{1,2,3}$ in the following scenario: r_1 creates operations m_1 altering m, followed by i_1 altering i. By the definition of causal consistency, $m_1 \prec i_1$.

For replication to be genuine, r_2 may store no information on any object $o \in M$ – thus m_1 may not be stored at r_2. After r_1 and r_2 are correctly synchronised with each other, r_1 fails indefinetely. Now, r_2 and r_3 attempt synchronisation.

For r_2 and r_3 to be able to synchronise correctly, two things must happen:

- every operation op applied in r_2 such that $op.object \in I_{2,3}$ must be also applied at r_3;
- every operation op applied in r_3 such that $op.object \in I_{2,3}$ must be also applied at r_2.

As r_3 attempts to apply operation i_1, it discovers it is missing the dependency on m_1[4]. Notice that r_3 must wait for m_1 to arrive before being able to apply i_1. Thus for r_2 and r_3 to be able to finish synchronisation (and inter-object causal consistency to hold), m_1 must eventually be delivered at r_3.

As r_2 didn't hold any information on m_1, and as the only other replica (r_1) which stored the operation has failed, this means that r_3 will never observe m_1 and can thus never apply i_1, concluding the proof. □

3 An initial approach

A distributed application typically has two options: either clients communicate through a central set of servers, or they communicate directly (in a peer-to-peer fashion). Using a central set of servers negatively impacts both availability and latency [20] while using a peer-to-peer communication model is directly impacted by the impossibility result of the previous section. Here we propose a hybrid approach which uses both a central server also allowing client-to-client communication to avoid these limitations.

3.1 System model

Our system model is comprised of two types of replicas, as shown in Figure 2:

server-replica – seen as a black-box for all server-side replicas, can store all client data[5]. We also assume failures are temporary and that the server-replica eventually recovers;

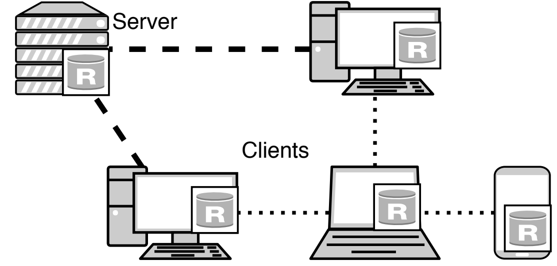

Figure 2. System model.

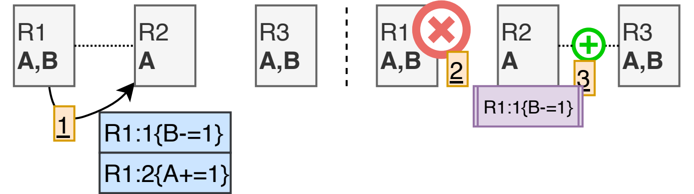

Figure 3. Keeping dependencies of applied operations.

client-replicas – reside on client (the users') devices, these can only hold a small subset of all client-data and are ephemeral in nature – last for a short time and failures might not be recovered from (e.g., application uninstall).

Network and Communication. The networking model departs from the typical client-server communication model (star graph) to a client-client enhanced network (leading to undirected graphs, for example, a tree where clients relay operations through each other).

For the purpose of this paper we assume replicas initially state which objects are to be replicated whose initial versions (for the replica) are fetched from the server. Any newly replicated object follows the same mechanism – an initial state is fetched from the server. Algorithms for fetching from direct neighbours or on no longer replicating objects are orthogonal to this work.

3.2 Initial algorithm

To ensure any replica is able to synchronise with any other replica, it must hold the dependencies for any operations it has applied. As briefly shown in Figure 3, this adds the overhead of keeping all operations our current state depends on. Besides countering the primary objective – genuine partial replication – it is impractical to store all operations forever. Additionally, if all replicas can, indirectly, communicate to all other replicas, this eventually grows towards full-replication.

[4]Dependency tracking can be done, for example, by propagating version vectors [10, 11] or direct dependencies [9].

[5]The central server replicas can use multiple geo-replication mechanisms [1, 2, 5, 9], in this paper we simplify the idea to a single centralised server for brevity and clarity.

3.3 Optimising the algorithm

Existing algorithms [4, 16] based on Lamport's notion of stability [8] can be applied to garbage collect operations which are no longer required to be kept to ensure eventual synchronisation. These algorithms give a notion of what is stable – which operations are known to be already propagated throughout the whole set of replicas – and thus clearly defines which operations are no longer needed. Constructing a dependency-collection algorithm thus follows directly, but this comes at the cost of running a stability protocol which *a*) becomes increasingly expensive as additional replicas join the system, and *b*) typically requires fixed and globally known membership (which is impractical in our model).

An alternative is to offload the notion of reliability to a specialised replica for this effect. This goes well with our system model, where clients communicate in a peer-to-peer setting but also with a server to bootstrap to the peer-to-peer network and, possibly, to store data to ensure durability.

If the server replica stores all objects, then any replica can fetch any missing dependencies from that replica. If it is known (at client replicas) which operations the server replica has observed, then garbage collection of these operations can safely be executed as there will be the possibility to fetch any dependencies from the server as a fallback when any dependencies are found to be missing. Summarily, our algorithm goes as follows:

- clients keep only a sub-set of all data objects;
- all operations are sent to all connected replicas, not taking into account any knowledge of the interest sets of the other replicas;
- operations on objects which the replica has no interest in are kept until marked as *durable*.

An operations is said to be *durable* if it is stored at the server-replica. This information is sent by the server every Δ seconds to any connected replicas and then propagated over the peer-to-peer network. Notice that this is similar to the discussed notion of stability – it comes at the cost of meta-data size which, in the worst case, is in the order of the amount of replicas in the system [6].

There are, however, some important properties which make the proposed algorithm interesting:

first, is that not every replica is required to communicate with (or even know about) every other replica,

second, is that any replica is able to correctly synchronise with any other replica,

third, Δ can be dynamically adjusted for the specific needs of the application at hand,

and finally, an interruption on the durability protocol has no impact on safety and liveness, it only brings the cost of keeping operations until the mechanism restarts.

4 Conclusion

This work follows directly from the attempt to extend a causally consistent system to the client-side [20]. The existing synchronisation model assumed full replication, which was promptly deemed impractical for client-side replicas. Ideally each client replica receives and stores only operations and data it is directly interested in.

An initial attempt, shown in Figure 1, only synchronises objects between a pair of replicas if these are within the intersections on their respective interest sets. Such a mechanism doesn't suffice – causal consistency cannot be maintained on inter-object dependencies.

Our approach is to leverage on server replicas which already provide the notion of durability. Any operation can safely be discarded as soon as it is know to be durable. The algorithm has interesting properties which are a good basis for a real client-side causally consistent replication system.

Considerations for future work

The proposed algorithm, even if academically interesting, is clearly is not enough. Neither aspects of the initially discussed notion of genuine are actually achieved – replicas receive operations on objects they are not interested in, and are forced to keep them to provide causality among objects.

The chosen implementation of the overlay – the network client establish by synchronising and propagating operations – can have a great impact here. If the overlay creates connections to other replicas in a way that promotes a large overlap on the interest set of replicas, then the amount of stored dependencies on non-replicated objects is small. In fact, if a given workload has clear boundaries on interest sets then a completely separate overlay can be created for each interest set, separating them at the network level without any alteration to the consistency mechanism.

An approach leveraging strict topology rules (such as a causal propagation trees [5]) can be leveraged to reduce the size of the necessary meta-data to track dependencies. In fact, techniques such as causal separators [15] – where specific topologies are leveraged to create disjoint sets of replicas to reduce vector sizes – can easily be applied with the help of the multiple server replicas of our centralised component.

We intend to explore these ideas in future work.

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 scholarship awarded to Albert van der Linde (SFRH/BD/117446/2016).

References

[1] 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.

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

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

[4] 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.

[5] 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.

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

[7] 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.

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

[9] 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.

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

[11] 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

[12] 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.

[13] 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.

[14] 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.

[15] Luis ET Rodrigues and Paulo Verissimo. 1995. Causal separators for large-scale multicast communication. In *Proceedings of 15th International Conference on Distributed Computing Systems*. IEEE, 83–91.

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

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

[18] 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.

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

[20] 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