Designing a causally consistent protocol for geo-distributed partial replication

Tyler Crain
Inria Paris-Rocquencourt &
Sorbonne Universités, UPMC Univ Paris 06, LIP6
tyler.crain@lip6.fr

Marc Shapiro
Inria Paris-Rocquencourt &
Sorbonne Universités, UPMC Univ Paris 06, LIP6
marc.shapiro@acm.org

Abstract

Modern internet applications require scalability to millions of clients, response times in the tens of milliseconds, and availability in the presence of partitions, hardware faults and even disasters. To obtain these requirements, applications are usually geo-replicated across several data centres (DCs) spread throughout the world, providing clients with fast access to nearby DCs and fault-tolerance in case of a DC outage. Using multiple replicas also has disadvantages, not only does this incur extra storage, bandwidth and hardware costs, but programming these systems becomes more difficult.

To address the additional hardware costs, data is often *partially replicated*, meaning that only certain DCs will keep a copy of certain data, for example in a key-value store it may only store values corresponding to a portion of the keys. Additionally, to address the issue of programming these systems, consistency protocols are run on top ensuring different guarantees for the data, but as shown by the CAP theorem, strong consistency, availability, and partition tolerance cannot be ensured at the same time. For many applications availability is paramount, thus strong consistency is exchanged for weaker consistencies allowing concurrent writes like *causal consistency*. Unfortunately these protocols are not designed with partial replication in mind and either end up not supporting it or do so in an inefficient manner. In this work we will look at why this happens and propose a protocol designed to support partial replication under causal consistency more efficiently.

The research leading to these results has received funding from the European Union Seventh Framework Programme (FP7/2007-2013) under grant agreement no. 609551.

PaPoC'15, April 21, 2015, Bordeaux, France.

http://dx.doi.org/10.1145/2745947.2745953

1. Partial replication

Partial replication is becoming essential in geo-replicated systems to avoid spending uncecessary resources on storage and networking hardware. Implementing partial replication is more difficult than deciding how many replicas to have because protocols for data consistency must hide the organisation of replicas so that the programmer sees the data as a single continuous store. Furthermore ensuring consistency with partial replication does not always easily scale, as it often requires additional communication between nodes not involved in the operations. For example, in [8], Saeida shows that a scalable implementation of partial replication, namely one that ensures genuine partial replication [9] is not compatible with the snapshot-isolation consistency criterion. Differently, this work focuses on *causal consistency* which allows concurrent writes and uses meta-data propagation instead of synchronisation to ensure consistency, but even in this case, implementing partial replication in a scalable way is not a straightforward.

1.1 Causal consistency and partal replication

While protocols ensuring causal consistency are generally efficient when compared to strongly consistent ones, they often do not support partial replication by default or if they do, limit scalability by requiring coordination with nodes that do not replicate the values updated during propagation. Within the standard structure of these protocols, updates are performed locally, then propagated to all other replicas where they are applied respecting their *causal order*, which is given by session order and reads-from order or can be defined explicitly. Given the asynchrony of the system, propagated updates might arrive out of causal order at external replicas, thus before they are applied a *dependency check* must be performed to to ensure the correctness. This check is based on ordering meta-data that is propagated along with the updates or through separate messages [2].

The best known structure of this meta-data are vector-clocks where each totally-ordered participant is given a vector entry and each of their updates are assigned a unique increasing scalar value. Since these geo-replicated systems can have a large number of participants, they often use slightly different representations of the vector-clocks. For example, certain protocols use vectors with one entry per DC [13], or one entry per partition of keys [3], or use vectors that can be trimmed based on update stability or the organisation of the partitions of keys across DCs [3]. Other than vector clocks other approaches exist including real time clocks [5], or to track reads of memory locations [6] or operations [7] up to the the last update performed by this client.

Interestingly, all of these mechanisms create (over) approximations of the dependencies of each operation, i.e. from this meta-data you cannot tell exactly what the causal dependencies for this operation are, but for correctness they cover at minimum all the dependencies. For example when using a single vector entry per server, all operations from separate clients connected to this server will be totally ordered even if they access disjoint sets of data.

Such systems use these approximations primarily because precise tracking of dependencies would not scale as the size of the meta-data would grow up to the order of the number of objects or users in the systems (depending on how dependencies are tracked). The issue with over approximating dependencies is that the dependency check might have wait on more dependencies than necessary, allowing the client to read stale versions of the data. Fortunately though this does not block the progress of clients as updates are replicated outside of the critical path.

1.2 Partial replication and approximate dependencies

This approximate tracking of dependencies also creates an unintended effect of making partial replication more costly.

To see why this happens, consider the example shown in figure 1 where a protocol is used that tracks dependencies using a version vector with an entry per server. Assume this protocol supports partial replication by only sending updates and meta-data to the servers that replicate the concerned object. In this example there are two DCs DC_A and DC_B each with 2 servers: A_1 and A_2 at DC_A and B_1 and B_2 at DC_B. Server A_1 replicates objects x_1 and x_2, server B_1 replicates objects x_2 and x_3 while server A_2 replicates objects y_1 and y_2 and server B_2 replicates objects y_2 and y_3. The system starts in an initial state where no updates have performed. Consider then that a client performs an update $u1$ on object x_1 at server A_1, resulting in the client having a dependency vector of $[1, 0, 0, 0]$, with the 1 in the first entry of the vector representing the update $u1$ at A_1. Since x_1 is not replicated elsewhere the update stays locally at A_1. Following this, the client performs an update $u2$ on object $y_2$at server A_2, returning a dependency vector of $[1, 1, 0, 0]$. The update $u2$ is then propagated asynchronously to B_2, where upon arrival a dependency check is performed. Since the dependency vector inclues a dependency from A_1, before applying the update, B_2 must check with B_1 that it has received any updates covered by this dependency in case they were on a key replicated by B_1. But since B_1 has not heard from A_1 it does not know if the update was delayed in the network, or if the update involved an object it does not replicate. Thus B_1 must send a request to A_1 checking that is has received the necessary update. A_1 will then reply that it is safe because $u1$ did not modify an object replicated by B_1, which will then be forwarded to to B_2 at which time $u2$ can be safely applied. Notice that if dependencies were tracked precisely, this additional round of dependency checks would not be necessary as the dependency included with $u2$ would let the server know that it only depended on keys not replicated at DC_B.

While this is a simple example that one could imagine easily fixing, different workloads and topologies can create complex graphs of dependencies that are not so easily avoided. Furthermore, current protocols designed for full replication do not take any additional measures specifically to minimize this cost, instead they suggest to send the meta-data to every DC as if it was fully replicated either in a separate channel [2] or simply without the update payload. In effect using no specific design patterns to take advantage of partial replication.

2. An initial approach

The goal of this work then is to develop a protocol supporting partial replication and providing performance equal to fully replicated protocols in a full replication setting, while minimising dependency meta-data and checks in a partial replication setting. We will now give a short description of the main mechanisms used to design this algorithm. It should be noted that these mechanisms are common to many protocols supporting causal consistency, except here they are combined in a way with the goal of supporting partial replication.

- **Update identification** Vector clocks are the most common way to support causality. To avoid linear growth of vector clocks in the number of (client) replicas, we apply a similar technique as in the work of Zawirski et al.[13]. Each entry in a vector represents a DC, or more precisely a cluster within a DC. Modified versions of protocols such as ClockSI [4] or a DC- local service handing out logical timestamps, such as a version counter, can be used to induce a total order to the updates issued at this DC, which can then be represented in the DC's vector entry.

  For causality tracking, each update is associated with its unique timestamp given by its home DC, plus a vector clock describing its dependencies. To provide session guarantees such as causally consistent reads when interacting with clients, the client keeps a vector reflecting its

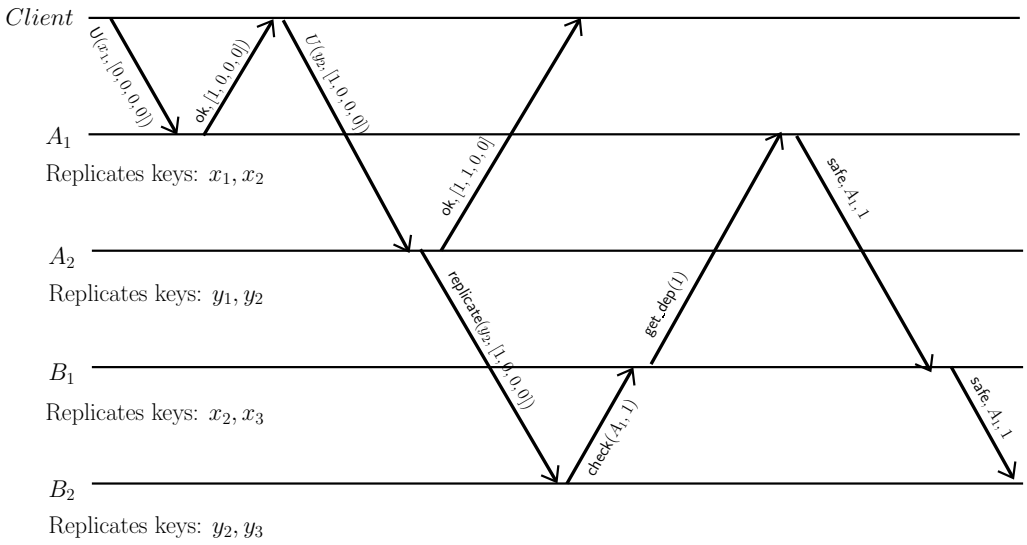

Figure 1: An example showing the possible dependency checks needed to ensure causality in a system with partial replication using version vectors with one entry per server for tracking dependencies.

previously observed values and writes. The system then ensures that clients may only read values containing all dependencies given this vector.

Given that in partial replication a DC might not replicate all objects, certain reads will have to be forwarded to other DCs where the object being read is replicated. The receiving DC then uses the client's dependency vector to generate a consistent version of the object that is then forwarded to be cached at the DC the client is connected to.

- **Disjoint safe-time metadata** In general, most protocols ensure causal consistency by not making updates from external DCs visible locally to clients until all updates causally preceding it have been received. When objects are replicated at all DCs this is fairly straightforward as all dependent updates are expected to be received. This is not always the case in partial replication since only the replicated dependent operations should be received, which could result in additional messages or dependency checks (see figure 1 for an example of why these additional checks would be needed), something which we are trying to avoid in order to have an efficient implementation.

  To avoid these additional dependency checks and metadata, the key insight in this work is to perform the dependency calculation at the origin DC and not the receiving DCs. Updates are still sent directly to their sibling replicas at other DCs, but they are not made visible to readers at the receiving DC until the origin DC confirms that its dependencies have been received i.e. the origin DC tracks which of its updates are safe to make visible at the receiver. At the origin DC, updates issued up to a time t are considered safe to apply at a receiving DC when all of the origin DC's servers have sent all their updates on the replicated objects of the receiver items up to time t. To keep track of this, a server at the origin DC communicates with each local server, keeping track of the time of the latest updates sent to external DCs, and once it has heard from each local server that time t is safe, this information is then propagated to the external DCs as a single message. Doing this avoids unnecessary cross-DC dependency checks and meta-data propagation, saving computation and network bandwidth. The negative consequence of this is that the observable data at the receiving DC might be slightly more stale than in the full replication case because the receiving DC has to wait until the sending DC has let it know that this data is safe. Such a delay can be seen as a consequence of tracking dependencies approximately as seen in the example in figure 1

- **Local writes to non-replicated keys** Given that causal consistency allows for concurrent writes, in order to ensue low latency and high availability a DC will accept writes for all objects, including those that it does not replicate. Using the vector clocks and metadata as described above this can be done without any additional synchronisation by just assigning unique timestamps to these updates that are reflected in the vector of the local DC. These updates can then be safely logged and made durable even in the case of network partition.

- **Atomic writes and snapshot reads** Beyond simple key-value operations, the protocol provides a weak form of transactions which allows to group reads and updates together and supporting CRDT objects [10]. Atomic writes

can be performed at the local DC using a 2-phase commit mechanism without contacting the remote replicas in order to allow for low latency and high availability. The updates are then propagated to the other DCs using the total ordered dependency metadata described previously ensuring their atomicity. (Note that atomic updates can include keys not replicated at the origin DC.) Causally consistent snapshot reads can be performed at a local DC by reading values according to a consistent vector clock, where reads of data items not replicated at the local DC are performed at another DC using the same vector clock.

Using these mechanisms allow partial or full replication with causal consistency while limiting the amount of unnecessary inter-DC meta-data traffic. All DCs are able to accept writes to any key, and causally consistent values can be read as long as one replica is available. Additionally the way the keys are partitions within a DC is transparent to external DCs, allowing this to be maintained locally.

An implementation of this protocol [12] is being developed within Antidote [11], the research platform for the SyncFree FP7 project, which is built on top of Riak-core [1] designed for testing scalable geo-replicated protocols.

Finally, it is important to note that while this protocol helps mitigate some of the costs of implementing partial replication in previous protocols, it does not completely solve the problem. It still uses imprecise representation of dependencies, which can still lead to false dependencies (that are checked locally within the sending DC) and can result in reading stale values that might otherwise be safe to read. Further studies are still needed and novel approaches using different mechanisms to see if these costs can be avoided entirely.

References

[1] Basho. Riak-core. `https://github.com/basho/riak_core`, 2015.

[2] N. Belaramani, M. Dahlin, L. Gao, A. Nayate, A. Venkataramani, P. Yalagandula, and J. Zheng. PRACTI replication. In *Networked Sys. Design and Implem. (NSDI)*, pages 59–72, San Jose, CA, USA, May 2006. Usenix, Usenix. URL `https://www.usenix.org/legacy/event/nsdi06/tech/belaramani.html`.

[3] J. Du, S. Elnikety, A. Roy, and W. Zwaenepoel. Orbe: Scalable causal consistency using dependency matrices and physical clocks. In *Symp. on Cloud Computing*, pages 11:1–11:14, Santa Clara, CA, USA, Oct. 2013. Assoc. for Computing Machinery. . URL `http://doi.acm.org/10.1145/2523616.2523628`.

[4] J. Du, S. Elnikety, and W. Zwaenepoel. Clock-SI: Snapshot isolation for partitioned data stores using loosely synchronized clocks. In *Symp. on Reliable Dist. Sys. (SRDS)*, pages 173–184, Braga, Portugal, Oct. 2013. IEEE Comp. Society. . URL `http://doi.ieeecomputersociety.org/10.1109/SRDS.2013.26`.

[5] J. Du, C. Iorgulescu, A. Roy, and W. Zwaenepoel. Closing the performance gap between causal consistency and eventual consistency,. In *W. on the Principles and Practice of Eventual Consistency (PaPEC)*, Amsterdam, the Netherlands, 2014. URL `http://eventos.fct.unl.pt/papec/pages/program`.

[6] W. Lloyd, M. J. Freedman, M. Kaminsky, and D. G. Andersen. Don't settle for eventual: scalable causal consistency for wide-area storage with COPS. In *Symp. on Op. Sys. Principles (SOSP)*, pages 401–416, Cascais, Portugal, Oct. 2011. Assoc. for Computing Machinery. .

[7] W. Lloyd, M. J. Freedman, M. Kaminsky, and D. G. Andersen. Stronger semantics for low-latency geo-replicated storage. In *Networked Sys. Design and Implem. (NSDI)*, pages 313–328, Lombard, IL, USA, Apr. 2013. URL `https://www.usenix.org/system/files/conference/nsdi13/nsdi13-final149.pdf`.

[8] M. Saeida Ardekani, P. Sutra, M. Shapiro, and N. Preguia. On the scalability of snapshot isolation. In F. Wolf, B. Mohr, and D. an Mey, editors, *Euro-Par 2013 Parallel Processing*, volume 8097 of *Lecture Notes in Computer Science*, pages 369–381. Springer Berlin Heidelberg, 2013. ISBN 978-3-642-40046-9. . URL `http://dx.doi.org/10.1007/978-3-642-40047-6_39`.

[9] N. Schiper, P. Sutra, and F. Pedone. P-Store: Genuine partial replication in wide area networks. In *Symp. on Reliable Dist. Sys. (SRDS)*, pages 214–224, New Dehli, India, Oct. 2010. IEEE Comp. Society. URL `http://doi.ieeecomputersociety.org/10.1109/SRDS.2010.32`.

[10] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski. Conflict-free replicated data types. In X. Défago, F. Petit, and V. Villain, editors, *Int. Symp. on Stabilization, Safety, and Security of Distributed Systems (SSS)*, volume 6976 of *Lecture Notes in Comp. Sc.*, pages 386–400, Grenoble, France, Oct. 2011. Springer-Verlag. . URL `http://www.springerlink.com/content/3rg3912287330370/`.

[11] SyncFree. Antidote reference platform. `https://github.com/SyncFree/antidote`, 2015.

[12] SyncFree. Antidote reference platform - partial replication branch. `https://github.com/SyncFree/antidote/tree/partial_replication`, 2015.

[13] M. Zawirski, A. Bieniusa, V. Balegas, S. Duarte, C. Baquero, M. Shapiro, and N. Preguiça. Swiftcloud: Fault-tolerant geo-replication integrated all the way to the client machine. *arXiv preprint arXiv:1310.3107*, 2013.