Reducing the Vulnerability Window in Distributed Transactional Protocols

Manuel Bravo[*†], Paolo Romano[†], Luís Rodrigues[†], Peter Van Roy[*]

[*]Université Catholic de Louvain, Belgium
[†]INESC-ID, Instituto Superior Técnico, Universidade de Lisboa, Portugal

Abstract

In this paper, we introduce a technique that can be used by distributed transactional protocols to reduce the vulnerability window of transactions. We propose a so far unexplored (to the best of our knowledge) usage of hybrid clocks. On one hand, it exploits loosely synchronized physical clocks to maximize the freshness of the snapshots used by transactions to read. On the other hand, logical clocks are used to reduce the extent to which the snapshot of update transactions is advanced upon their commit.

We believe that the joint usage of these two techniques can potentially reduce the abort rate in comparison to previous protocols such as Clock-SI, GMU, and SCORe.

Categories and Subject Descriptors C.2.4 [*Distributed Systems*]: Distributed Databases

Keywords hybrid clocks, concurrency control, transactional protocols, abort rate, snapshot isolation

1. Introduction

Capturing the passage of time and the cause-effect relations among events is a key problem at the core of the design of distributed systems. Unsurprisingly, this issue is also of paramount importance in the design of cloud data stores that provide some meaningful consistency guarantee, such as causal consistency [8], snapshot isolation [3] and serializable snapshot isolation [5]. A variety of clocks mechanisms have been proposed to track and reason about the order in which events happen, such as physical clocks, logical clocks, and hybrid clocks.

A key characteristic of distributed transactional protocols that impacts the performance of transactional cloud data stores is the abort rate, which is affected by the grade of concurrency. This is obviously set by the workload but it is also impacted by how the protocol handle time. For instance, a protocol that it is able to generate transactions with small *vulnerability windows* is likely to reduce the grade of concurrency; and thus, the abort rate. One could define transaction's *vulnerability window* as the time window defined by transaction's starting point and serialization point, or commit time.

[Copyright notice will appear here once 'preprint' option is removed.]

In this short paper we propose a novel technique that aims at reducing the *vulnerability window* of transactions. Our technique uses an hybrid clock implementation. The idea is to use the physical part of the hybrid clock to set the starting time of the transaction; therefore, pushing the starting time as much as possible. On the other hand, our technique proposes to use the logical part of the hybrid clock to set the transactions' serialization point. The combination of these two has the potential of reducing the *vulnerability window*; and in consequence, the abort rate.

Despite the fact that this is still work in progress, we believe this paper already discusses and flags interesting aspects of the use of clocks in distributed transactional protocols. The contributions of this paper are the following:

- A technique that proposes a novel usage of hybrid clocks in distributed transactional protocols that aims at reducing the abort rate by shortening transactions' *vulnerability windows*.
- Comparison and discussion of the implications that different types of clocks may pose in the implementation of a distributed transactional protocol in particular, and in the design of distributed systems in general. The discussion uses protocols found in the literature such as Clock-SI [4], GMU [12] and SCORe [11].

The rest of the paper is organized as follows. Section 2 gives a brief overview of the different clocks that can be used to order events in distributed systems. Section 3 describes our technique by integrating it into a protocol in order to ease readers comprehension. Section 4 compares our solution to other proposed protocols that use different clock implementations. Finally, Section 5 discusses the next steps of our research and concludes the paper.

2. Clocks

In the design of distributed systems, one could use different clocks techniques to reason about the order of events. A first type of clocks are *physical clocks*. Each participant of a distributed system can use its own physical clock to timestamp events, and reason about the ordering by comparing timestamps. Nevertheless, these clocks can never be perfectly synchronized which may increase system latencies due to the need to keep into account drifts in the clock, e.g., by introducing additional wait phases. Tightly synchronized physical clocks can be achieved by leveraging GPS protocols at the cost of expensive hardware; whereas, loosely synchronized physical clocks can be inexpensively produced by relying on distributed clock synchronization algorithms, such as NTP [10] and PTP [2].

A second type of clocks are *logical clocks*. Introduced by Lamport in 1978 [8], these clocks order events based on passage of information rather than passage of time. The community has proposed different forms of logical clocks, as scalar [8], vectors [6, 9]

and matrix [13, 16]. While scalar clocks are very efficient w.r.t. the message size, they may insert extra dependences between events. Vector and matrix clocks fix this problem at the cost of increasing the size of the messages to sometimes unbearable sizes.

Finally, the last type of clocks are a combination of the previous categories, namely *hybrid clocks*. A good example of this type of clocks is Hybrid Logical Clocks (HLC) [7]. It combines a physical clock with a scalar logical clock. Their approach can be successfully used to (i) avoid, at least in some circumstances, waiting periods due to clock drift, and (ii) precisely identify cause-effect relations avoiding the possibility of wrongly ordering events.

3. On Fully Distributed Transactional Protocols

In order to better understand and illustrate the benefits of our technique, we make use of a protocol that embodies it. We realized that some of the fully distributed transactional protocols in the literature, such as SCORe [11] and Clock-SI [4], share a common structure and mostly only diverge in the type of clocks they use. Thus, the protocol we use throughout the discussion share this common pattern and integrates our technique.

In this section, we first give an overview of the protocol and how we integrate our technique. Then, we describe the protocol in detail.

3.1 Protocol Overview

The protocol implements snapshot isolation (SI) [3]. It satisfies the following three properties: (i) each transaction reads from a consistent snapshot, (ii) conflicting update transactions commit in total order producing a new snapshot in the database, and (iii) a transaction aborts if introduces a conflict with a concurrent committed transaction. In SI, two transactions conflict if their write-sets, which is the set of updated data items, have common elements. This type of conflicts are called write-write conflicts. In consequence, SI precludes read-only transactions to abort. Since workloads are usually composed by mostly read-only transactions, SI is likely to improve performance compared to stronger consistency criteria, such as serializability where read-write conflicts abort transactions. SI is the default consistency choice of popular data engines as Oracle and Microsoft SQL Server.

In addition, the protocol can be characterized as a Genuine Partial Replication (GPR) [14] and Deferred Update Replication (DUR) [15] protocol. GPR protocols are those in which only the servers that store data needed by the transaction are involved in the coordination. This is a desirable characteristic for large-scale systems. DUR is an optimization for transactional protocols where updates are buffered in the coordinator and sent atomically in the commit step. This obviously saves some coordination and potentially reduces latency.

The protocol is composed by three phases: (i) an initial phase where transaction's *snapshot time* is set, defining the versions that transactions can read, (ii) an interactive phase where clients issue read and update requests, and (iii) a two phase commit protocol that sets transaction's *commit time*, in case all involved servers agree on committing. We define *vulnerability window* of a transaction as the window time created between transaction's *snapshot time* and transaction's *commit time*. Two transactions whose *vulnerability windows* overlap are considered concurrent by the protocol. Since a transaction is aborted if there is a concurrent conflicting transaction already committed, one goal of this type of protocols should be to shorten the *vulnerability window* as much as possible. This leads to reduce the abort rate and improve protocol's performance.

Our technique precisely focus on this observation. We propose the use of hybrid clocks to identify consistent snapshots and order committed transactions. The hybrid clock is composed by a physical clock and a scalar logical clock. The physical clock is always equal to the value of the server's physical clock and it is used to set transaction's *snapshot time*. We assume that physical clocks of different servers are loosely synchronized through a distributed clock synchronization protocol as NTP; nevertheless, the protocol correctness does not depend on how synchronized clocks are. On the other hand, the scalar logical clock will always be set to the largest time stamp the server has seen. This means that the logical clock is "infected" by the physical time. The protocol uses the logical part of the hybrid clock to propose *commit times*.

3.2 Protocol

Algorithm 1 shows the pseudocode of the protocol running in the coordinator of the transaction (lines 1-24) and on the servers (lines 25-43). Notice that any server can act as a coordinator. A transaction issued by a client would take the following steps:

1. Upon a start transaction request, the coordinator initializes the transaction and sets the snapshot time as the maximum between its physical clock and the logical clock (lines 2-5). The *snapshot time* will be used by the transaction to identify the consistent snapshot from where to read.
2. Clients interactively send operations (*read*/*update*) to the coordinator. Updates are buffered in the coordinator (line 14). Reads are sent to the partition responsible for the data item (if not buffered). Upon a read request for *key*, the server first updates its logical clock (line 26). Then, it waits for prepared conflicting transactions with smaller *prepare time* than transaction's *snapshot time* to commit (lines 27-30). Otherwise, the server may return a version that misses writes of concurrent transactions; and thus, violate SI. Finally, the server returns the largest version with a smaller or equal *commit time* than transaction's *snapshot time*.
3. Upon a commit transaction request, the coordinator starts a two phase commit protocol (2PC) to either *commit* or *abort* the transaction.
   - First, the coordinator sends a *prepare* request to the servers storing part of the transactions's write set (lines 17-18).
   - Each server first updates its logical clock (line 33). Then, it waits for already prepared conflicting concurrent transactions to either commit or abort (lines 34-35). Otherwise, SI may be violated. Next, the server runs a certification check that look for conflicting concurrent committed transactions (line 36). If none, the server increases its logical clock (line 37) and uses it as *prepare time*. The proposed *prepare time* is sent to the coordinator. Otherwise, an *abort* message is sent back to the coordinator.
   - The coordinator waits for all the partitions to reply. If all partitions agree on committing the transaction, the coordinator sets the *commit time* of the transaction to the maximum of the gathered *prepare times*. Finally, it sends committed to the client and the *commit time* to the involved servers.
   - When a server receives the *commit time*, it applies the updates to its local store using the *commit time* as a version identifier.

Our protocol has two potential waiting periods in order to ensure correctness. The first can be found in lines 27-30 of our protocol and it is a direct consequence of using physical clocks to set transactions *snapshot times*. A server waits until conflicting concurrent prepared transactions are committed or aborted if their *commit time* may be smaller than current transaction's *snapshot time*. For instance, let us assume two potentially concurrent transactions T_1 and T_2. T_1 starts before T_2, updates data items x and y, and tries

Algorithm 1: Protocol

```
// Coordinator operations
1  upon receive start_tx() from Client do
2      T.TxId←generate_txid()
3      T.SnapshotTime←max(Server.PhysicalClock, Server.MaxTS)
4      T.State←active
5      T.Client←Client
6      send T to Client

7  upon receive read(T, Key) from Client do
8      if is_buffered(T, Key) then
9          send get_buffered_value(T, Key) to Client
10     else
11         Server←get_responsible(Key)
12         send read(T, Key) to Server

13 upon receive update(T, Key, Value) from Client do
14     buffer_value(T, Key, Value)
15     send ok to Client

16 upon receive commit(T) from Client do
17     foreach Server in T.UpdatePartitions do
18         send prepare(T) to Server
19     wait until receiving PrepareTime from T.UpdatePartitions
20     T.CommitTime←max(all prepare times)
21     T.State←committed
22     foreach Server in T.UpdatePartitions do
23         send commit(T) to Server
24     send ok to Client

// Server operations
25 upon receive read(T, Key) from Coordinator do
26     Server.MaxTS←max(Server.MaxTS, T.SnapshotTime)
27     if Key is updated by T' ∧
28       T'.State = prepared ∧
29       T.SnapshotTime > T'.PrepareTime then
30         wait until T'.State = committed
31     send get(Server.Backend, Key, T.SnapshotTime) to T.Client

32 upon receive prepare(T) from Coordinator do
33     Server.MaxTS←max(Server.MaxTS, T.SnapshotTime)
34     if Key is updated by T' ∧ T'.State = prepared then
35         wait until T'.State = committed
36     if CertificationCheck(T) then
37         Server.MaxTS←Server.MaxTS + 1
38         T.PrepareTime←Server.MaxTS
39         T.State←prepared
40         send T.PrepareTime to Coordinator

41 upon receive commit(T) from Coordinator do
42     T.State←committed
43     put(Server.Backend, T.WriteSet, T.CommitTime)
```

to commit in servers P_1 and P_2. On the other hand, T_2 is a read-only transaction that reads data item x in P_1. When the read request reaches P_1, T_1 has not been committed yet; therefore, P_1 does not know whether T_1 has to be included in T_2's snapshot or not. If P_1 proposed a *prepare times* for T_1 smaller than T_2's *snapshot time*, there is a possibility that the maximum of all proposed *prepare time*, and in consequence T_1's *commit time*, is smaller than T_2's *snapshot time*. In this case, T_1 has to be included in T_2's snapshot, otherwise SI is violated. The only way to ensure correctness in this scenario, without adding extra coordination, is to wait for T_1 to finish, as our protocol does. Clock-SI [4], which uses physical clocks to set transactions *snapshot times*, also identified and solved the problem similarly.

The second waiting period is found in lines 34-35 of our protocol. The intuition behind this is that prepared transactions are not considered in the certification check (line 36) and they may pose write-write conflicts, and thus, violate SI. Therefore, we suggest to wait until there is no conflicting transaction committing before starting the certification phase. Let us discuss an example to clarify this safety property. Let us assume two transactions T_1 and T_2 whose write sets intersect in data item x stored in P_1. P_1 receives a prepare request first for T_1. Then, it receives the prepare request for T_2. Since T_1's *commit time* is unknown at this point, there is always the possibility that T_1 and T_2 are concurrent. Therefore, only one should successfully commit. If P_1 do not wait for T_1 to commit or abort before preparing T_2 both may commit, and thus, violate SI. Even when T_1 and T_2 are known to be concurrent, one should not abort T_2 immediately since T_1 may abort.

4. Comparison with Related Work

We now focus on discussing the implications and the trade-offs that our clock choice poses in comparison to other clock mechanisms proposed in the literature. We consider three protocols to compare: SCORe [11] that uses a simple scalar logical clock, GMU [12] that uses a vector clock with an entry per server in the cluster, and Clock-SI [4] that uses a single physical clock. All these protocols share a very similar protocol skeleton to the one described above. In addition, we also use Hybrid Logical Clocks (HLC) [7] in our discussion. One could easily imagine how to fit them into our protocol skeleton. Furthermore, they have already been used in transactional databases, such as CockroachDB [1].

There are two crucial points in which the type of clock used characterizes a GPR protocol: assigning the *snapshot time* when the transaction starts and proposing a *commit time* in the commit phase. We analyse them separately in the following paragraphs.

Assigning snapshot time This task (i) defines how recent the read data is, and (ii) impacts the transaction's *vulnerability window* by setting its starting point. Physical clocks are in general desirable for this task since, with logical clocks, the rate in which each server's clock advances directly depends on how often they participate in transactions. Thus, if a server that was isolated for a while happens to assign the *snapshot time* of a transaction, this is likely to (i) read quite stale data, and (ii) abort since the beginning of the transaction will be set way in the past for active servers. For instance, let us discuss a example with three servers $P_1, P_2, andP_3$ whose initial logical clocks are the same. After executing a large number of transactions in which only $P_1 and P_2$ participate, P_3's logical clock will be set way behind in the past in comparison to $P_1 and P_2$'s clocks. In this situation, we say that P_3 is isolated. In consequence, next time that P_3 sets the *snapshot time* of a transaction that updates data items in any of the other servers, the transaction is likely to abort. In the contrary, physical clocks advance automatically even for servers that are isolated by the workload. Thus, physical clocks are capable to avoid both problems. SCORe and GMU suffer from these problems. GMU tries to tackle them by advancing the *snapshot time* as a transaction reads if possible. This, however, comes at the cost of storing and shipping a vector instead of a single scalar.

On the other hand, physical clocks also have a major disadvantage: protocol's performance depends on the clock skew. This has two implications. First, a read request and a prepare request of a transaction with a *snapshot time* in the future (w.r.t. local server's clock) has to be delayed until the local clock catches up. Second, while logical clocks always assign *snapshot times* that represent an already committed transaction, physical clocks may assign a *snapshot time* that is in the future. This means that the snapshot may be unavailable and the server may need to wait for prepared conflicting transaction to commit before being able to process a read request (first waiting period of our protocol, lines 27-30). Clock-SI suffers from both problems. On the contrary, our protocol avoids the first by the use of the scalar in conjunction to the physical. Thus, instead of waiting for the physical clock to catch up, our protocol simply updates the logical one. This is possible because *snapshot times*

Protocol	Clocks	Freshness	Vulnerability Window	Unavailable Snapshot	Clock skew
SCORe	Scalar	Low	Undefined	No	No
GMU	Vector	Medium	Undefined	No	No
Clock-SI	Physical	High	Undefined	Yes	Yes
HLC	Hybrid	High	Undefined	Yes	No
Our protocol	Hybrid	High	Small	Yes	No

Table 1. Summary of GPR protocols with different clock choices and its consequences.

are set as the maximum between the physical and the logical clock. Notice that we are not first to notice this improvement of hybrid clocks over physical clocks, as the HLC paper already mentions it.

Proposing commit time This task impacts the size of transactions' *vulnerability window*. As argued before, the protocol should always try to shorten it in order to reduce the abort rate. Thus, there will be less overlapping between the transactions and less chances to find conflicts. Based on this assumption, logical clocks seems to be more suitable for this task. They only move forward when necessary while physical clocks automatically advance, potentially proposing larger *commit times*. SCORe and GMU use logical clocks for this task, while Clock-SI uses a physical clock. On the other hand, HLC would take the maximum between the physical clock and the logical clock, potentially leading to similar results than Clock-SI. Our protocol, instead, only uses the logical clock for this task.

We believe our protocol take the best clock choice in both takes, by reducing the *vulnerability window* of transactions and maximizing data freshness. Table 1 summarizes the advantages and disadvantages of different clocks techniques applied to GPR protocols. It is not simple to reason about the size of *vulnerability windows* of transactions in all the protocols but in ours. For certain workloads, where all servers participate in most of the transactions, GMU and SCORe are likely to achieve results similar to our protocol. Nevertheless, as soon as the workload starts isolating some servers, the length of the transactions started on those servers will be indefinitely long.

5. Future work

We plan to implement the proposed protocol and compare its performance and other parameters, as the abort rate, to other fully distributed transactional protocols. We are mostly interested to compare to systems with a similar protocol but using different type of clocks. This will lead us to experimentally prove or disprove whether our initial conclusions are right.

Acknowledgments

This work was partially funded by the SyncFree project in the European Seventh Framework Programme (FP7/2007-2013) under Grant Agreement n^{o} 609551 and by the Erasmus Mundus Joint Doctorate Programme under Grant Agreement 2012-0030.

References

[1] Cockroach. https://github.com/cockroachdb/cockroach.

[2] IEEE Standard for a Precision Clock Synchronization Protocol for Networked Measurement and Control Systems. *IEEE Std 1588-2002*, pages i–144, 2002.

[3] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1987. ISBN 0-201-10715-5.

[4] J. Du, S. Elnikety, and W. Zwaenepoel. Clock-si: Snapshot isolation for partitioned data stores using loosely synchronized clocks. In *Reliable Distributed Systems (SRDS), 2013 IEEE 32nd International Symposium on*, pages 173–184, Sept 2013.

[5] A. Fekete, D. Liarokapis, E. O'Neil, P. O'Neil, and D. Shasha. Making snapshot isolation serializable. *ACM Trans. Database Syst.*, 30(2): 492–528, June 2005. ISSN 0362-5915.

[6] C. J. Fidge. *Timestamps in message-passing systems that preserve the partial ordering*. Australian National University. Department of Computer Science, 1987.

[7] S. Kulkarni, M. Demirbas, D. Madappa, B. Avva, and M. Leone. Logical physical clocks. In M. Aguilera, L. Querzoni, and M. Shapiro, editors, *Principles of Distributed Systems*, volume 8878 of *Lecture Notes in Computer Science*, pages 17–32. Springer International Publishing, 2014. ISBN 978-3-319-14471-9.

[8] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, July 1978. ISSN 0001-0782.

[9] F. Mattern. Virtual time and global states of distributed systems. *Parallel and Distributed Algorithms*, 1(23):215–226, 1989.

[10] D. L. Mills. A brief history of ntp time: Memoirs of an internet timekeeper. *ACM SIGCOMM Computer Communication Review*, 33 (2):9–21, 2003.

[11] S. Peluso, P. Romano, and F. Quaglia. Score: A scalable one-copy serializable partial replication protocol. In *Proceedings of the 13th International Middleware Conference*, Middleware '12, pages 456–475, New York, NY, USA, 2012. Springer-Verlag New York, Inc. ISBN 978-3-642-35169-3.

[12] S. Peluso, P. Ruivo, P. Romano, F. Quaglia, and L. Rodrigues. When scalability meets consistency: Genuine multiversion update-serializable partial data replication. In *Distributed Computing Systems (ICDCS), 2012 IEEE 32nd International Conference on*, pages 455–465, June 2012.

[13] S. K. Sarin and N. A. Lynch. Discarding obsolete information in a replicated database system. *Software Engineering, IEEE Transactions on*, (1):39–47, 1987.

[14] N. Schiper, P. Sutra, and F. Pedone. P-store: Genuine partial replication in wide area networks. In *Proceedings of the 2010 29th IEEE Symposium on Reliable Distributed Systems*, SRDS '10, pages 214–224, Washington, DC, USA, 2010. IEEE Computer Society. ISBN 978-0-7695-4250-8.

[15] D. Sciascia, F. Pedone, and F. Junqueira. Scalable deferred update replication. In *Proceedings of the 2012 42Nd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, DSN '12, pages 1–12, Washington, DC, USA, 2012. IEEE Computer Society. ISBN 978-1-4673-1624-8.

[16] G. T. Wuu and A. J. Bernstein. Efficient solutions to the replicated log and dictionary problems. In *Proceedings of the third annual ACM symposium on Principles of distributed computing*, pages 233–242. ACM, 1984.