Technical Report

Department of Computer Science
and Engineering
University of Minnesota
4-192 Keller Hall
200 Union Street SE
Minneapolis, MN 55455-0159 USA

TR 15-012

Scalable Transactions in Partially Replicated Data Systems with Causal Snapshot Isolation

Anand Tripathi, Gowtham Rajappan, Vinit Padhye

July 14, 2015

Scalable Transactions in Partially Replicated Data Systems using Causal Snapshot Isolation

Anand Tripathi, Gowtham Rajappan and Vinit Padhye
Department of Computer Science & Engineering
University of Minnesota, Minneapolis, Minnesota, 55455 USA
Email: (tripathi, rajappan, padhye)@cs.umn.edu

***Abstract*—**
We present here a transaction management protocol, which enhances the Partitioned Causal Snapshot Isolation (PCSI) protocol, to support scalable transactions with non-local partition writes in a partially replicated multi-version database. The PCSI protocol is scalable for update transactions that involve local read and writes. However, it faces scalability limitations with non-local partition writes, when partitions are updated from any of the sites. In PCSI, to support non-local partition writes, a ghost replica of that partition is created at the transaction execution site. It is used primarily for assigning update sequence numbers to the local transactions and does not store any data. We present here an alternate approach for supporting non-local partition writes. The update sequence number for a non-local partition update is obtained from one of the remote sites that stores a replica of that partition. This approach raises several problems in regard to stalling of transactions at the remote replica's site. We develop here a protocol based on the notion of sequence number escrow to address these problems. Through experimental evaluations, we show that this approach for supporting non-local partition updates scales almost linearly.

I. Introduction

In large-scale systems data replication across different geographically distributed sites is needed for several reasons. Many large-scale web applications have geographically distributed client population. With geo-replication, clients in a particular geographic region can be directed to the servers located at a site close to that region. Geo-replication is also desired for applications that need to be disaster-tolerant. Providing transaction support is an important requirement of a database system, since transaction semantics are often needed in many applications for atomically executing a sequence of operations and support certain consistency guarantees.

Replication management techniques in large-scale systems have to make certain fundamental trade-offs between data consistency, scalability and availability [11]. There are several factors that can affect the scalability of replication management techniques. A high degree of replication can become a major obstacle in achieving scalability as noted in [12]. This has motivated investigation of techniques for managing data with partial replication [10], [13], [19], [18], [20], [22], i.e. a data item is not required to be replicated at all sites, and a site contains replicas of only a subset of the items in the database. From the considerations of performance and scalability, full replication is undesirable in large-scale systems.

Another obstacle in regard to scalability is the requirement of strong consistency, as in serializable transactions, which can require distributed coordination such as the execution of a two-phase commit protocol among multiple sites. Synchronous replication, in which the updates of a transaction are propagated synchronously to other sites before committing the transaction, provides strong consistency but incurs high latencies for transactions. Moreover, this may not be practical in wide-area environments. This has motivated designs of many large-scale systems to adopt weaker yet useful models of consistency [6], [4], which can be supported using asynchronous replication management techniques. The weak consistency based models for replication management in large-scale systems have ranged from eventual consistency [6], per-record timeline consistency [4], causal consistency [15], and snapshot isolation (SI) [3] with causal consistency [23].

We present here a scalable protocol for providing transaction support in partially replicated databases in large-scale systems. A database is partitioned in multiple disjoint partitions and each partition is replicated at one or more sites, and a site may contain any number of partitions. Each partition stores a set of items using the key-value based multi-version data storage model. The protocol which we present here is based on our previous work [17] on the PCSI (*Partitioned Causal Snapshot Isolation*) model. The PCSI model provides snapshot isolation (SI)-based transaction support with causal consistency for managing databases with partial replication.

In this paper we address the scalability limitations of the PCSI protocol and present a mechanism which we developed for remote partition updates to overcome these limitations. In the PCSI model, a transaction may be executed at any site and it may read or modify items in any of the partitions, local or remote. A transaction is committed locally at a site, and then, at some later time, its updates are propagated asynchronously to other sites and applied there. Transactions are ordered according to a causal ordering. At a remote site, transactions are applied according to their causal ordering. For this purpose a vector clock is maintained for each partition and the length of the vector clock is equal to the number of its replicas.

We measured the scale-out capability of the PCSI model [17] in terms of the peak throughput achieved for different system configurations as the number of sites was varied. In these scale-out experiments, the number of partitions

was set equal to the number of sites, which means that systems with larger configurations served proportionally larger data sets. We found that the PCSI model [17] is scalable when transactions update only the items that are present in the local partitions at their execution site. It is also scalable when the workload consists of transactions with updates to non-local partitions such that a partition is updated by transactions executing only at some small bounded number of sites that do not have that partition. With workloads in which every partition may be updated by transactions executing at any of the sites in the system, the scalability of this protocol is limited to about 20 sites.

The scalability limitation noted above motivated us to re-examine the mechanism that we had developed for supporting transactions updating non-local partitions. For supporting updates to partitions not present at a transaction's execution site, the PCSI model uses the notion of a *ghost replica*, which is created at the execution site. A ghost replica of a partition does not store any data and no update related information is propagated to it from other sites. It acts as a virtual replica. Its purpose is to locally assign sequence numbers for the updates to this partition whose replicas are all located on remote sites. The motivation behind this approach was to avoid communication with any remote sites containing the updated partitions to obtain update sequence numbers. We found that this mechanism becomes the limiting factor for scalability when partitions are updated from all of the sites. That causes the vector clock of a partition to have an element for each of the ghost replica, in addition to those for the real replicas. As the vector clock lengths become large, the vector clock related computations start affecting the performance.

We present here an alternate approach for supporting non-local partition updates in the PCSI model to make it scalable when partitions may be updated from any of the sites in the system. This approach is based on obtaining a sequence number for a transaction's updates to a remote partition from one of the replicas of that partition. This approach raises several issues which we address in this paper. One is to ensure that the local transactions at the replica's site which grants a sequence number to a remote transaction are not stalled due to sequence ordering of transaction updates. To address this issue, we develop the notion of *escrow sequence numbering*. When granting a sequence number to a remote transaction, a replica reserves as an escrow a range of sequence numbers for local transactions and gives out to the remote transaction an advance sequence number following the escrow range. We elaborate on the various issues in adopting this approach and present the mechanisms that we developed to address them. Another issue that arises is the possibility of deadlocks when several transactions update a common set of multiple remote partitions. Deadlocks can arise in applying updates of two transactions which obtained remote sequence numbers from a common set of partition replicas but the sequence order implied by two replicas are not consistent. In Section 3 we elaborate on these problems and present solutions for them.

In the next section we discuss the related work on transaction support for partially replicated data in cloud/cluster computing systems. Section III presents a brief overview of the PCSI model and in Section IV we discuss scalability issues due to the ghost partition scheme. In Section V we present the escrow based advance sequence number assignment scheme for transactions updating remote partitions. We elaborate on the various problems that arise in adopting this approach and present our design to address these problems. In Section VI we present the results of our performance evaluation experiments and demonstrate the scalability of this new mechanism.

II. Related Work

The issues with scalability in data replication with strong consistency requirements are discussed in [12]. Such issues can become critical factors for data replication in large-scale systems and geographically replicated databases. This has motivated use of other models such as snapshot isolation (SI) [3], weaker consistency models such as eventual on and causal consistency, and partial replication of data.

The techniques for transaction management in systems with partial replication of data have been investigated by others in the past [13], [22], [18], [10], [20], [21]. The notion of *genuine partial replication* [19] requires that the messages related to a transaction should only be exchanged between sites storing the items accessed by the transaction. The approach presented in [13] guarantees serializability. It uses epidemic communication that ensures causal ordering of messages using a vector clock scheme where each site knows how current is a remote site's view of the events at all other sites. Other approaches [22], [20], [19] are based on the state machine model, utilizing atomic multicast protocols. These approaches support *1-copy serializability*. In contrast, the approach presented in [21] is based on the snapshot isolation model, providing the guarantee of *1-copy snapshot isolation*. Non-Monotonic Snapshot Isolation (NMSI) [1] and PCSI are both based on the causal snapshot isolation model. The PCSI model differs from NMSI in following ways. NMSI [1] protocol requires maintaining dependency information either on per object version level or on per partition level with the restriction that updates within a partition should be serialized. In contrast, PCSI does not require that updates on a partition be serialized; the updates are serialized only on per-replica basis, and only the updates on a given object are serialized which is a result of the no write-write conflict requirement.

Recently, many data management systems for cloud data-centers distributed across wide-area have been proposed [6], [4], [2], [15], [23]. Dynamo [6] uses asynchronous replication with eventual consistency, whereas PNUTS [4] provides a stronger consistency level than eventually consistency, called as *eventual timeline consistency*, but both these systems do not support transactions. Megastore [2] provides transactions over a group of entities using synchronous replication. COPS [15] provides causal consistency, but does not provide transaction functionality, except for snapshot-based read-only transactions. Eiger [16] provides both read-only and update transactions

with causal consistency but requires maintaining causal dependencies on per object level. PSI [23] provides snapshot isolation based transaction support guaranteeing causal consistency.

Orbe [8] supports partitioned and replicated databases with causal consistency, but it does not support multi-key update transactions. It uses two-dimensional vector clocks for capturing causal dependencies. GentleRain [9] uses physical clocks to eliminate dependency check messages, but similar to Orbe it lacks a general model of transactions. Spanner [5] provides strong consistency with serializable transactions under global-scale replication. However, it relies on special purpose hardware such as GPS or atomic clocks to minimize clock uncertainty. The work presented in [14] provides a consistency scheme called *red-blue consistency*, which uses operation commutativity to relax certain ordering guarantees for better performance.

III. Background: Overview of the PCSI Model

We first present here an overview of the PCSI model for transaction management in partially replicated key-value based data storage systems. More detailed description of this model can be be found in [17]. The PCSI model provides the following guarantees in regard to the transaction updates. A transaction's updates are applied at a site only after the updates of all transactions causally preceding it have been applied there. Atomicity of a transaction's updates to multiple partitions is ensured, i.e. either all or none of the updates of a transaction become visible at a site. If two or more concurrent transactions update the same item, then only one is able to commit and all others are aborted, following the snapshot isolation model. This means that updates to an item are globally ordered. Concurrent updates to different partitions are permitted. Also different items in the same partition but at different replicas can be concurrently updated by transactions.

Before executing read/write operations, a transaction must obtain a globally consistent snapshot for the partitions to be accessed. All read operations on a partition are performed according to the snapshot obtained for that partition. If the partition to be read is stored at the local site, it executes read operation on the local database, otherwise the transaction performs a read from a remote site. The writes are buffered till the commit time. When a transaction is ready to commit, it checks for update conflicts with concurrently committed transactions. In case of no conflicts, the transaction is committed and applied at the local site and its updates are asynchronously propagated to other sites that store the partitions updated by the transaction. For ensuring causal consistency, the causal dependencies of the transaction are computed and this information is communicated with the update propagation message. A remote site applies the updates only if it has applied updates of all the causally preceding transactions.

A. Partition Vector Clocks and Causal Dependency View

Each site is identified by a unique $siteId$. A site maintains a sequence counter for each of its local partitions, which is used to assign sequence numbers to the local transactions modifying items in that partition. A transaction obtains, during its commit phase, a timestamp for each partition it is modifying. A timestamp is a pair $<siteId,\ seq>$, where seq is a local sequence number assigned to the transaction, by the replica of that partition at the site identified by $siteId$. The *commit timestamp vector* ($\mathcal{C}_t$) of transaction t is a set of timestamps assigned to the transaction by the replicas of the modified partitions. For an item modified by transaction t in partition q, the commit timestamp assigned by q is denoted by $\mathcal{C}_t^q$.

Each replica of a partition maintains a vector clock referred to as the *partition view ($\mathcal{V}_p$)*. The vector clock value of a partition replica indicates the sequence numbers of transaction updates from other replica sites which have have been applied to this replica.

For capturing causality and atomicity dependencies, a site also maintains, for each local partition p, a *partition dependency view ($\mathcal{D}_p$)*, which is a set of vector clocks. For causality, it identifies for each of the other partitions the events that have occurred in that partition and that causally precede the current state of partition p. The *partition dependency view* $\mathcal{D}_p$ consists of a vector clock for each other partition. Formally, $\mathcal{D}_p$ is a set of vector clocks $\{\mathcal{D}_p^1, \mathcal{D}_p^2, \cdots, \mathcal{D}_p^q, \cdots, \mathcal{D}_p^n\}$, in which an element $\mathcal{D}_p^q$ is a vector clock corresponding to partition q. Each element of the vector clock $\mathcal{D}_p^q$ identifies the transactions performed on partition q that causally precede the transactions performed on partition p identified by $\mathcal{V}_p$.

When obtaining a snapshot, we use the partition dependency views to check if a snapshot is globally consistent for a given set of partition replicas to be accessed. For each pair of partitions p, q to be accessed by the transaction, we can consider $\mathcal{V}_p$ and $\mathcal{V}_q$ as snapshots for p and q provided the following conditions holds: $\mathcal{D}_p^q \leq \mathcal{V}^q \wedge \mathcal{D}_q^p \leq \mathcal{V}^p$. This means that all those update events of q on which the snapshot state of p depends are visible in the snapshot state of q, and a symmetric condition is to be satisfied for the update events of p on which the snapshot state of q depends.

It is possible that a site executes a transaction that accesses some partitions not stored at that site. This requires reading/writing items from remote site(s). An important requirement for a multi-site transaction is to ensure that it observes a consistent global snapshot. The protocol for obtaining a globally consistent snapshot is detailed in [17].

B. Transaction Validation

For each data item, there is a designated *conflict resolver site* which is responsible for checking for update conflicts for that item. A conflict resolver site maintains an ordered list of the commit timestamps ($<siteId, seq>$) of all the committed versions of the corresponding item. The transaction coordinates with the conflict resolver sites responsible for the items in its write-set to check if some other transaction has created an item version newer than the latest version

visible in the transaction's snapshot. Such a situation indicates an update conflict. This coordination is done using a two-phase-commit (2PC) protocol. The transaction commits only if none of the items in its write-set have an update conflict. On successful validation the transaction is committed and a commit timestamp is obtained from each of the partitions being updated by the transaction. These commit timestamps form the timestamp vector $\mathcal{C}$ for the transaction. The commit decision and the commit timestamps are communicated to the conflict resolvers. The updates to local partitions are applied and then the update messages are sent to the remote sites using the update propagation mechanism described below.

C. Update Propagation

The causal dependencies of a transaction are captured in form of a set of vector clocks, called *transaction dependency view* ($\mathcal{TD}$). A vector clock in this set corresponds to a partition and identifies all the causally preceding transactions pertaining to that partition. The atomicity dependencies are captured by the commit timestamp $\mathcal{C}$ value. With the update propagation message for transaction t. the $\mathcal{TD}_t$ and $\mathcal{C}_t$ values are communicated to the remote sites. At the remote site, say site k, for every partition p specified in $\mathcal{TD}_t$, if p is stored at site k, then site checks if its partition view $\mathcal{V}_p$ is advanced up to $\mathcal{TD}_t^p$. Moreover, for each modified partition p for which the remote site stores a replica, the site checks if $\mathcal{V}_p$ of the replica contains all the events preceding the sequence number value present in $\mathcal{C}_t^p$. i.e., for each modified partition p the following check is done.

$$\mathcal{V}_p[\mathcal{C}_t^p.siteId] = \mathcal{C}_t^p.seq - 1 \quad (1)$$

If this check fails the site delays the updates until the vector clocks of the local partitions advance enough. If this check is successful, the site applies the updates to the corresponding local partitions. Updates of transaction t to any non-local partitions are ignored.

Furthermore, when applying the transaction t, the partition dependency views of the modified partitions are updated using $\mathcal{TD}_t$ and $\mathcal{C}_t$ values to indicate the partition's causal and atomic dependencies on other partitions as described in [17].

D. Remote Partition Update

In the PCSI protocol we used the notion of *ghost replica* to support updates to remote partitions by a transaction. If a transaction involves updating any remote partition, the execution site creates a local *ghost replica* for that partition. A ghost replica does not store any data but provides following functions. The main function is to assign local commit timestamps, using a local sequencer, to transactions updating this partition. The PCSI protocol does not propagate updates to the ghost replicas of any partition.

IV. Scaling Problems with Remote Partition Updates

PCSI protocol scales well for local reads, local writes and remote reads. But, when we tested the PCSI protocol (with the ghost replica scheme) using a benchmark in which every partition was updated by transactions from every site, the system scaled only for a small number of sites - 8, 12 and 16. As the number of nodes reaches 20, the performance of PCSI for remote writes starts deteriorating. In the PCSI protocol, the size of the vector clock for a partition is equal to its total replication degree. For instance, if partition P_1 has a replication degree of 3 and is present in sites: S_1, S_2 and S_3, then the vector clock of P_1 in all the aforementioned sites will exactly have three entries. Using the ghost replica of a partition, the number of entries in the vector clock of the partition increases as and when a site, that does not have partition executes a transaction that involves a write to the partition. This means that the number of entries in the vector clock of the partition begins to grow, and in the worst case becomes equal to the total number of sites.

Vector clock is a core component of PCSI - it's used for obtaining a globally consistent snapshot, checking for conflicts, recording the commit sequence numbers (number of events that have happened so far in the site), capturing the dependency view of partition(s), and ensuring causal ordering of when applying a transaction's updates at a remote site. As all these operations involve checking the vector clock, when the size of the vector clock increases so does the computational complexity of all these operations.

We evaluated the ghost partition scheme using a single partition update. The transaction read one partition and updated one partition. The number of items read/written per partition is 4 and the size of each item is 10KB. The notation x% means that x% of the transactions update a remote partition and (100 - x)% update a local partition. A comparison of transactions involving local partition writes (0%) *vs* remote partition writes(1%, 5%) using PCSI with the ghost replica approach is shown in Figure 1. Clearly, the ghost replica scheme does not scale when the number of sites > 16. This motivated us to come up with a new solution to tackle transactions involving remote partition writes.

Fig. 1: Throughput Scalability of PCSI with the ghost replica approach

V. Escrow based remote partition update protocol

We now present an alternate approach in which a transaction updating items in a non-local partition gets the sequence num-

ber from one of the sites which has a replica of that partition. With this approach, consider the case where a site, say S_1, has the sequence number (SN) 100 for partition P_1, and a transaction at some remote site S_2 needs a sequence number from S_1 for partition P_1. If S_1 gives the "next" sequence number 101, then this approach has a major drawback because all the local transactions at S_1 will stall (as they cannot get the next sequence number) until the remote transaction commits at site S_2 and its update is applied at site S_1. This is because the PCSI protocol orders transactions updating a partition replica on the sequence numbers assigned to them by the replica.

The stalling of local transactions as noted above affects the transaction response times and system throughput. The number of stalled transactions depends on several factors. It depends on the local load and the time it takes for a remote transaction's update to be applied at the site issuing the sequence number.

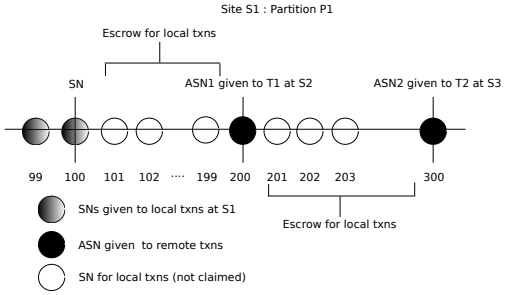

Fig. 2: Notion of ASN and Escrow

To address the problem mentioned above, the idea we developed is to reserve a range of sequence numbers for local transactions when a transaction from another site requests a sequence number. This reserved range of sequence numbers serves as an *escrow* for local transactions. A remote site requesting a sequence number is given a number after the escrow range. We call it *Advance Sequence Number* (ASN). Figure 2 shows that the current sequence number at partition P_1 in site S_1 is 100 and the escrow value is set to 100. When remote transactions T_1 from site S_2 and T_2 from site site S_3 request for sequence numbers from P_1, ASNs 200 and 300 are given out to the T_2 and T_3 respectively.

In our implementation of the escrow technique, each partition keeps track of two sequence numbers: Sequence Number, as in the PCSI protocol, and the last Advance Sequence Number that was given out by this partition, ASN_{last}. Whenever a remote transaction requests an ASN, it's computed using the formula given below:

$$ASN_{next} := ASN_{last} + escrow \tag{2}$$

Each partition also maintains an ASN List, a list of sequence numbers that the partition has given out to other site(s). Also, ASN_{first} is the first ASN in the ASN List. Naturally, the next question is how to configure the value of escrow.

Figure 3 shows the sequence of events involved in the transactions T_1 and T_2: obtaining ASN, committing the transaction, and propagating the updates. The notation $T_1 : w(P_1, P_2)$

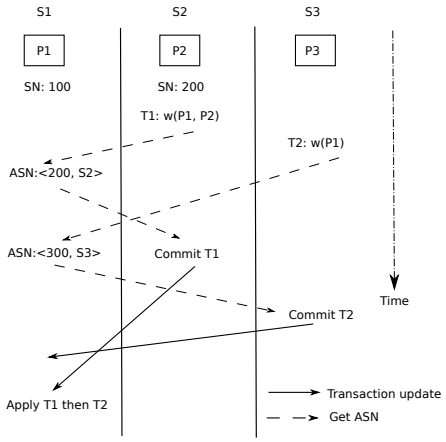

Fig. 3: Single ASN Scenario

means that the transaction T_1 updates the partition replicas P_1 and P_2. Note that as $T_1 \prec T_2$, T_1's update will be applied at S_1 before applying T_2's update, even if T_2's update reaches S_1 first, thus guaranteeing transaction ordering.

A. Configuration of Escrow value

Case 1: If the value of escrow is too small, then the local transactions will consume all numbers in the escrow range before the remote transaction's update, the one that has the sequence number ASN_{first}, is applied at the local site. This will result in increased stalling of transactions, therefore leading to reduced throughput and commit rate at the local site. In our design we abort such stalled transactions instead of blocking them while they are holding locks on the conflict resolver sites, required by the commit protocol.

Case 2: If the value of escrow is too big, local transactions will not fill the escrow range. By the time the update of the remote transaction with the sequence number ASN_{first} for the partition reaches the partition replica, we've two options. One option is to queue the remote transaction and let the local transactions use the remaining escrow range. The second option is to generate a special message to fill the remaining escrow range and to advance the vector clock of the partition. This message is then propagated to all the other sites that has the partition's replica. The former option is undesirable because it blocks all transactions that causally follow the remote transaction, thus causing increased queuing delay and reduced throughput.

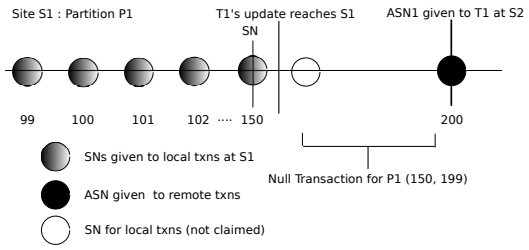

Fig. 4: Null Transaction

To help understand the second option, consider Figure 4, when transaction T_1's update reaches site S_1 with ASN 200, say the current SN of P_1 at site S_1 is 150. If all the causal dependencies for the remote transaction are satisfied, we would like to apply the transaction right away. Hence, we generate a special message that says: if the value of vector clock reads $\mathcal{V}_{P_1}[S_1] = 150$, then increment the value to $(ASN_{first} - 1)$, which in this case is 199. We call this special message, *null transaction* as shown in Figure 4, as its sole purpose is to advance the vector clock by generating null events.

If we can dynamically set the escrow for a partition, based on its local transaction rate, then we can make sure that local transactions would not stall. Figure 5 shows the events (commit/abort) that happen at partition P_1 on site S_1, from the time an ASN is given out by P_1 to a remote transaction to the time its updates are applied at S_1.

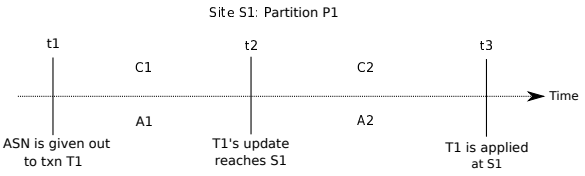

Fig. 5: ASN Timeline

- *C1* - Number of commits between the time *t1* and *t2*.
- *C2* - Number of commits between the time *t2* and *t3*.
- *A1* - Number of aborts between the time *t1* and *t2*. These aborts are due to transaction stalling.
- *A2* - Number of aborts between the time *t2* and *t3* due to stalling. *A2* will be zero, if we generate *null transaction* and apply the transaction's update right away.

The local transaction rate at each partition, for a given site, might vary with time and so will the values A1, A2, C1 and C2. The sum of A1, A2, C1 and C2 denotes the number of attempts made by local transactions to get a sequence number for partition P_1, from the time an ASN was given out to a transaction to the time the transaction with ASN is applied. In a given time period, a number of ASNs will be given out by a partition. To adaptively set the escrow value, over an observation period we record the values of A1, A2, C1 and C2 for each ASN given out over that period. We set the escrow value to the sum of their mean values, i.e. $\overline{A1}$, $\overline{A2}$, $\overline{C1}$, $\overline{C2}$, multiplied by a scaling factor $\alpha > 1$. We also make sure that the escrow is never set less than some minimum value $escrow_{min}$, which was set to 50 in our experiments.

$$escrow := min(escrow_{min}, \alpha * (\overline{A1} + \overline{A2} + \overline{C1} + \overline{C2}) \quad (3)$$

This way the escrow value is set adaptively based on the local transaction rate at the partition.

B. Protocol for Single Remote Partition Update

The single remote partition update algorithm is straightforward. Whenever a transaction executes at a site, which does not have a replica of the partition P, the transaction requests a sequence number from a site which has a replica of the partition P. This sequence number is generated by adding a escrow to the largest value of ASN given out by the replica of the partition P. The reason we've to use the largest value of ASN is because we want the ASN given out to be strictly increasing as it's used to maintain the ordering of events in a partition.

Algorithm 1 Obtaining Single ASN

function GETSINGLEASN($\mathcal{T}$, $\mathcal{P}$, $\mathcal{S}$)
// $\mathcal{T}$ is the txn ID of the txn that access the remote partition $\mathcal{P}$
// $\mathcal{P}$ this partition's ID
// $\mathcal{S}$ is the site at which transaction $\mathcal{T}$ is executing
[begin atomic region
$ASN_{next} \leftarrow ASN_{last} + escrow$
add $<ASN_{next}, \mathcal{S}>$ to ASN List
end atomic region]
return ASN_{next}

When the transaction T's update from S reaches the site that gave out the ASN, P's ASN_{first} is checked against the P's commit timestamp of the transaction, C_T^P. If $C_T^P.seq$ is equal to ASN_{first} at partition P, and if all the other causal dependencies for T are satisfied except for P, then a *null transaction* is generated to fill the remaining unconsumed part of the escrow range and the transaction's update is applied. As soon as the transaction's update is applied, the $<ASN_{first}, S>$ entry is removed from the ASN List.

C. Issues with Multiple Remote Partition Update (Multiple ASNs)

Now, let's consider the case where multiple remote partitions are updated by a transaction. In scenarios where two transactions concurrently try to update a common set of remote partition replicas, if the ASNs are not handed out in a total ordered fashion there is a potential of deadlocks when trying to apply the transactions' updates.

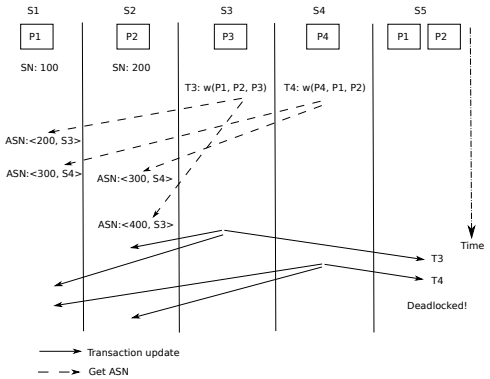

Fig. 6: Multiple ASN Scenario

In Figure 6, transactions $T_3 : w(P_1, P_2, P_3)$ and $T_4 : w(P_4, P_1, P_2)$ get the ASNs from sites S_1 and S_2 for partitions P_1 and P_2 respectively. T_3 gets the ASN 200 for partition P_1 from site S_1 before T_4, whereas T_4 gets the ASN 300 for

partition P_2 from site S_2 before T_3. As T_3 got the ASN from P_1 before T_4, $T_3 \prec T_4$ for partition P_1, but $T_4 \prec T_3$ for partition P_2. The above situation makes T_3 and T_4 to violate the transaction ordering property as explained in [17].

Moreover, when the update messages of T_3 and T_4 reach site S_5, both updates get stuck. This is because w.r.t P_1, $T_3 \prec T_4$ and w.r.t P_2, $T_4 \prec T_3$, and hence both transactions cannot be applied resulting in deadlock. Consequently, what this means is that all transactions that have a sequence number obtained from site S_1, after transaction T_3, for partition P_1 or a sequence number from site S_2, after transaction T_4, for partition P_2 cannot be applied at S_5.

A solution to this problem is to total order the transactions which update a common set of partition replicas. This means that the requests for obtaining ASNs by transactions with common set of partition replicas must be total ordered, i.e., all such replicas handle the ASN requests in the same order. In essence, what we need here is a *total order multicast protocol* [7] for delivering ASN requests to partition replicas such that all transactions requesting ASNs from a common set of remote partition replicas are total ordered based on commit timestamps.

D. Protocol for obtaining Multiple ASNs

We present here a total ordered multicast protocol for requesting ASNs from multiple remote partition replicas. In order to avoid deadlocks, each transaction $\mathcal{T}$ requests ASNs from remote partition replicas in an ascending order of partition ids. Also, each ASN request to a remote partition replica, $\mathcal{P}$, contains a list of $<remotePartitionId, siteId>$, $\mathcal{P}_{list}$, ordered by partition ids and a predecessor transaction list, $\mathcal{T}_{list}$, which is explained next. The response to an ASN request will contain the ASN itself and $\mathcal{T}_{list}$. Each transaction, $\mathcal{T}_i$, in the $\mathcal{T}_{list}$ will satisfy all the following conditions:

- $\mathcal{T}_i$ obtained ASN from $\mathcal{P}$ before $\mathcal{T}$
- $\mathcal{T}_i$'s updates haven't been applied yet at $\mathcal{P}$
- $\mathcal{T}_i$ has a common set of remote partition replicas with $\mathcal{T}$

The following information about $\mathcal{T}_i$ is stored in the $\mathcal{T}_{list}$: its site ID, $\mathcal{S}$, and $\mathcal{P}_{list}$.

Consider the scenario explained in Figure 6. When transaction T_4 requests ASN from the partition replica P_1 at site S_1, in addition to the ASN 300 it also receives the predecessor list which contains an entry for T_3. When T_4's request for ASN reaches the partition replica P_2 at site S_2, P_2 looks at the predecessor list and finds that T_3 should be given ASN before T_4 to ensure total ordering. It checks if it has already *seen* T_3, i.e. it has either seen T_3's request and given out an ASN to T_3, or it has decided not to handle in the future any delayed request from T_3, thereby forcing it to abort. If P_2 at site S_2 has not *seen* T_3, we've two options. One option is to make T_4 wait until T_3's request arrives at S_2. This way if T_4 is delayed by a long time period then the $responseTime$ of T_4 takes a hit and so does the throughput at site S_4. The other option, adopted in our design, is to make T_4 wait only for a time-out period, which was set to 200 msec in our experiments. If T_3 arrives before the time-out period, then T_3 precedes T_4 else P_2's ASN_{next} can be given to T_4 and T_3 will be aborted when its request reaches S_2. A pseudocode of this protocol is provided in Algorithm 2.

Algorithm 2 Obtaining Multiple ASN

function GETMULTIPLEASN($\mathcal{P}$, $\mathcal{T}$, $\mathcal{S}$, $\mathcal{T}_{list}$, $\mathcal{P}_{list}$)
// This is executed when transaction $\mathcal{T}$ at site $\mathcal{S}$
// requests ASN for partition $\mathcal{P}$ at site $mySiteId$
// $\mathcal{T}_{list}$ is the predecessor transaction list of $\mathcal{T}$
// $\mathcal{P}_{list}$ list of all $<remotePartitionId, siteId>$ that $\mathcal{T}$ updates
if $(seenTxn[\mathcal{T}] = ABORT)$ **then** return $NULL$
if $(\exists\ T_i \in \mathcal{T}_{list} \mid \{seenTxn[\mathcal{T}_i] \neq$ TRUE $\wedge$ $<\mathcal{P}, mySiteId> \in \mathcal{P}_{list}$ of $\mathcal{T}_i\})$ **then**
wait(time-out period)
[begin atomic region
for all $\mathcal{T}_i \in \mathcal{T}_{list}$ **do**
if $(seenTxn[\mathcal{T}_i] = NULL)$ **then**
$seenTxn[\mathcal{T}_i] \leftarrow ABORT$
$ASN_{next} \leftarrow getMaxASN() + escrow$
$\mathcal{T}_{list} \leftarrow$ appendPredecessors($\mathcal{T}_{list}$, ASN List)
add $<ASN_{next}, \mathcal{S}, \mathcal{T}, \mathcal{P}_{list}>$ to ASN List
$seenTxn[\mathcal{T}] \leftarrow TRUE$
end atomic region]
return $<ASN_{next}, \mathcal{T}_{list}>$

E. Increased abort rate due to Multiple ASNs

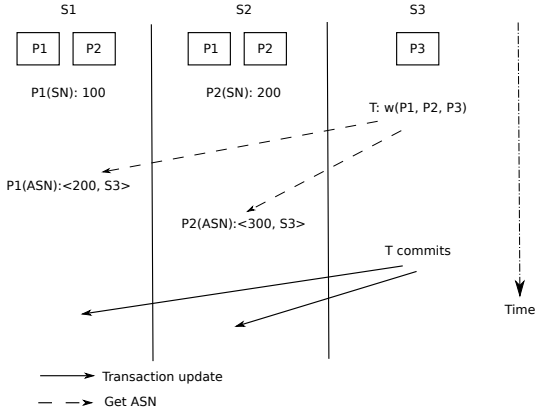

Fig. 7: Increased abort rate due to Multiple ASN

When the number of remote partitions updated by a transaction is more than one, then the *null transaction* generation can result in increased abort rate. Consider the scenario shown in Figure 7, $T : w(P_1, P_2, P_3)$ has two remote partitions P_1 and P_2 but the ASN for P_1 is obtained from site S_1 and the ASN for partition P_2 is obtained from site S_2. Note that the sites S_1 and S_2 both have replicas of partitions P_1 and P_2. Consider the case when transaction T's update reaches site S_1 and the causal dependencies of T on all update partitions but P_1 and P_2 are satisfied. Even if P_1 at S_1 generates a *null transaction* advancing its SN to $(C_T^{P_1}.seq - 1)$, which is

$ASN_{first} - 1$, site S_1 cannot apply the transaction T because of its causal dependency on partition P_2. This will cause all local transactions updating P_1 at S_1 to stall and abort. Also, the effect is compounded if the same process takes place at site S_2 for partition P_2. This effect gets even more pronounced when the number of remote partitions per transaction is more than two. The problem highlighted here indicates a sharp increase in the abort count *A2* as shown in Figure 5.

F. Null Transaction generation in case of Multiple ASNs

Due to multiple remote partition updates, we found the commit rates to be as low as 76%. $A2$, shown in the Figure 5, contributed to the abort of 20% of the transactions. In order to reduce the number of aborts due to transactions involving multiple remote partition updates, the *null transaction* generation across all the remote update partitions of a transaction has to be coordinated. We want that the gap between the current sequence number (SN) and ASN of each of the remote update partition should reduce to zero within small real-time difference between them. At each site, a thread periodically checks if the updates can be applied and it's during this check/step the ASN gaps are reduced. The reason why we reduce the aforementioned gap by a small amount in each step is to keep some part of the escrow for local transaction(s) to get the sequence numbers thus reducing abort rate. It's important that the gap between the ASN and the current SN has to be reduced in each step as this ensures that the algorithm converges ultimately. We call this technique, *random walk based synchronization* as each partition takes steps towards convergence of the gaps to zero. There are two cases to be considered here:

Case 1: Site which generated an ASN does not have replicas of any other remote update partitions. In Figure 7, if site S_1 did not have partition P_2, then this would be the case. In this case, each partition generates a *null transaction* to fill the gap equal to the difference between its current sequence number and the ASN handed out. This is how transactions with single remote partition update are applied.

Case 2: Site which generated an ASN has replicas of all or some of the other remote partitions accessed by the transaction. We illustrate our approach using the example in Figure 7, where sites S_1 and S_2 have a replica of all the remote partitions updated by the transaction T.

The notation, $G^{S_j}_{P:S_i}$ denotes the view at site S_j of the gap corresponding to the ASN generated by partition P's replica at site S_i. The view of gaps for transaction T are computed in the following way:

- If S_i and S_j are the same, then the gap for P is $ASN_{first} - SN$. This view of the gap is exact and it is controllable at S_i using *null transactions*.
- Otherwise, $G^{S_j}_{P:S_i} := C^P_T.seq - \mathcal{V}_P[S_i]$. This value is always smaller than the gap $G^{S_i}_{P:S_i}$ because $\mathcal{V}_P[S_i]$ at S_j is less than or equal to $\mathcal{V}_P[S_i]$ at S_i.

In our example, site S_1 can only reduce the gap $G^{S_1}_{P_1:S_1}$ as the ASN was generated by S_1. Similarly, site S_2 can reduce the gap $G^{S_2}_{P_2:S_2}$. We would like the time difference between the gaps' convergence to zero to be as small as possible so that the local transaction aborts at S_1 and S_2 are reduced.

In our scheme, at site S_1, the gap for partition P_1 is reduced according to the following heuristics:

- If both the gaps are less than a *threshold*, then we fill the gap by generating a *null transaction*.
- Otherwise, we reduce the gap by half or more. i.e., the gap is reduced to $min(G^{S_1}_{P_1:S_1}/2, G^{S_1}_{P_2:S_2})$.

A similar process will be followed at S_2. The reason why we don't fill the gap completely is to provide some part of the escrow for local transactions until the time the gaps for all remote partitions converge to zero. This way, both the partitions will converge more or less at the same time and the transaction T can be applied at both sites S_1 and S_2.

The synchronization of the escrow gap for a transaction can be generalized to more than two remote partitions. Suppose a transaction obtained ASNs for four remote partitions P_i, P_j, P_k, and P_l, and site S_i generated ASNs for replicas P_i and P_j then the gap, for P_i and P_j at S_i, is reduced to:

$$min(min(G^{S_i}_{P_i:S_i}, G^{S_i}_{P_j:S_i})/2, median(G^{S_i}_{P_k:S_k}, G^{S_i}_{P_l:S_l}))$$

VI. Evaluations

We implemented the escrow based remote write mechanism in a system based on the PCSI protocol. The focus of our evaluation was on the following performance measures: (a) scalability of the enhanced PCSI protocol measured by peak throughput for different system sizes, (b) scalability of single vs. multiple remote partition update, (c) impact of remote partition updates on response time, 2PC latency, the update application delay due to causal dependencies and the commit rate.

A. System Configurations

We evaluated the scalability of the escrow based remote write protocol for system configurations with various number of sites. The number of partitions was set equal to the number of sites, thus larger configurations served larger data-sets to reflect system scale-out. We emulated an environment where a geo-scale system is distributed over multiple regions, and each region has several sites. About half of the replicas of a partition are present in one region, and the other replicas are located at sites in different regions. In our experiments, all sites had the same number of partitions, and all regions had the same number of sites, which was set to 4 in our experiments. Each partition contained 100,000 items of 1K bytes each. Each partition was replicated at three sites, and a site contained replicas of three partitions.

B. Benchmarks and Transaction Workloads

We developed a custom benchmark to evaluate the impact of various parameters such as the degree of partial replication, number of partitions read or written by a transaction, number of remote partitions read or written, number of items in a partition read or written by a transaction, and the percentage

of transactions updating remote partitions. These parameters can be specified to generate a particular type of workload.

We defined three workload classes as shown in Table I. In these workloads all transactions were update transactions. Each transaction read from 2 partitions, and modified 1 to 3 partitions, depending on the workload class. The number of remote update partitions is shown in the table for each of the three workload classes. For each accessed partition, the transaction read 4 items and modified 2 items, which were selected randomly with uniform distribution. With 1KB item size, each transaction read 8KB data, and wrote 2KB, 4KB, and 6KB data for workloads A, B, and C respectively. In each workload class, it is possible to control the fraction of the transactions that update remote partitions. In Table I, *x* means that *x%* of the transactions update one or more remote partitions and *(100 - x)%* update only local partitions.

Transaction workload	No. of read partitions	No. of write partitions	No. of Remote Write Partitions	
			x%	(100-x)%
A	2	1	1	0
B	2	2	1	0
C	2	3	2	0

TABLE I: Transaction workload configurations

C. Testbed Environment:

We performed the evaluations on a computing cluster at the Minnesota Supercomputing Institute (MSI). In this cluster, each node had 8 CPU cores with 2.8 GHz Intel X5560 Nehalem EP processors, and 22 GB main memory. Each node in the cluster served as a database site in our experiments. We set the update propagation period to 1 second.

D. Scalability of Escrow based remote write protocol

For each system size, we measured the peak throughput, average transaction response time, time required for validation with remote conflict resolver sites, delays in applying updates at remote sites due to causal dependencies and the total commit rate. Figures 8, 9, and 10 show the peak throughput for workload A, B, and C, respectively. The degree of replication in all these evaluations was set to 3.

For each workload, we measured the peak throughput for two cases: all local update transactions (i.e. parameter x set of 0), and 5% remote update transactions. To evaluate the scalability, we measured the peak throughput for system configurations consisting of 16, 32, 48, and 64 nodes. The protocol is scalable for workloads A and B. The peak throughput for workload B tends to be lower than that for A due to overhead of the commit protocol and ASN coordination. The performance for workload C tends to be much lower compared to that for workloads A and B. The performance of workload C scales for up to 48-node configuration, and the peak throughput tends to be lower than that for workloads A and B as at least 3 sites are involved in commit protocol execution and ASN coordination. This workload also involves more update messages, and each message contains 3 times more data as compared to workload A.

We observe that the escrow based remote write protocol provides near-linear scalability for workloads A and B; the peak throughput achieved scales almost linearly with the number of sites. For example, as we scaled-out the system configuration from 16 sites to 64, throughput increased by a factor of 3.5 in case of workload A. For workload B, the corresponding performance scale-out factor was 2.86, and it was 1.82 for workload C.

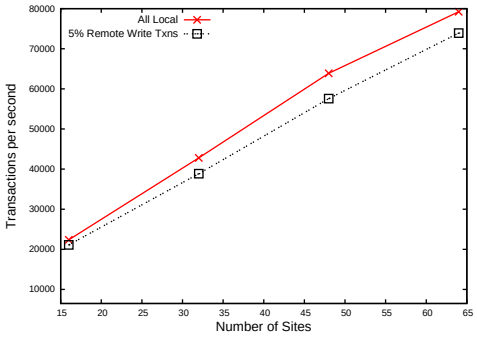

Fig. 8: Throughput for workload A

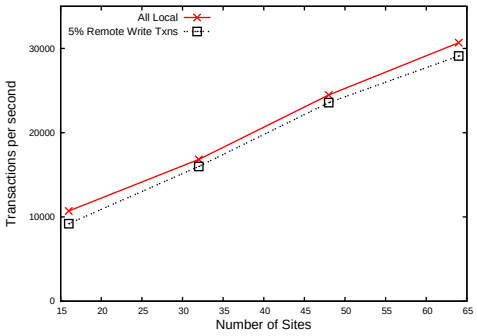

Fig. 9: Throughput for workload B

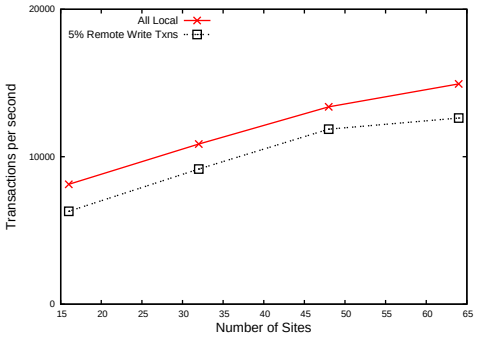

Fig. 10: Throughput for workload C

E. Performance Impact of Remote Partition Updates

For the transaction workloads A, B and C the average transaction response time, 2PC latency, update delay and commit

rates were measured and are shown in Table II. The numbers shown in Table II are for a system consisting of 40 sites. The 2PC latencies and the transaction response time increased as the number of remote update partitions increased, indicating the cost incurred for communication with the conflict resolvers and for ASN coordination. Also, the update delay increased indicating the time delay due to synchronizing the ASN gaps of multiple partitions. The commit rate decreased as the number of remote update partitions increased.

Transaction workload	Response time (msec)	2PC latency (msec)	Update delay (msec)	Commit rate %
A	102.96	154.25	78.17	96.82
B	139.47	176.36	101.54	92.18
C	199.15	197.88	150.05	86.92

TABLE II: Response time latencies and commit rates

VII. Conclusion

We have presented here the escrow based remote partition write protocol, which enhances the scalability of the PCSI model for transaction workloads involving remote partition updates in systems with partial replication of data in cloud/cluster computing environments. We have presented here the unique problems associated with the escrow technique. The escrow technique, in combination with the PCSI protocol, enables scalable data replication management supporting local and remote partition updates while providing causal consistency. Our evaluations show that the escrow based remote write model can scale for transaction workloads in which the number of remote update partition is limited to one.

Acknowledgements: This work was supported by National Science Foundation grant 131933 and Minnesota Supercomputing Institute.

References

[1] M. S. Ardekani, P. Sutra, and M. Shapiro, "Non-monotonic snapshot isolation: Scalable and strong consistency for geo-replicated transactional systems," in *Proc. of SRDS '13*, 2013, pp. 163–172.

[2] J. Baker, C. Bond, J. Corbett, J. J. Furman, A. Khorlin, J. Larson, J.-M. Leon, Y. Li, A. Lloyd, and V. Yushprakh, "Megastore: Providing scalable, highly available storage for interactive services," in *CIDR*, 2011, pp. 223–234.

[3] H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O'Neil, and P. O'Neil, "A critique of ANSI SQL isolation levels," in *Proc. of ACM SIGMOD'95*. ACM, 1995, pp. 1–10.

[4] B. F. Cooper, R. Ramakrishnan, U. Srivastava, A. Silberstein, P. Bohannon, H.-A. Jacobsen, N. Puz, D. Weaver, and R. Yerneni, "Pnuts: Yahoo!'s hosted data serving platform," *Proc. VLDB Endow.*, vol. 1, pp. 1277–1288, August 2008.

[5] J. C. Corbett and et al, "Spanner: Google's globally distributed database," *ACM Trans. Comput. Syst.*, vol. 31, no. 3, Aug. 2013.

[6] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, "Dynamo: Amazon's highly available key-value store," *SIGOPS Oper. Syst. Rev.*, vol. 41, pp. 205–220, October 2007.

[7] X. Défago, A. Schiper, and P. Urbán, "Total order broadcast and multicast algorithms: Taxonomy and survey," *ACM Comput. Surv.*, vol. 36, no. 4, pp. 372–421, Dec. 2004.

[8] J. Du, S. Elnikety, A. Roy, and W. Zwaenepoel, "Orbe: Scalable causal consistency using dependency matrices and physical clocks," in *Proceedings of the 4th ACM Symposium on Cloud Computing*, ser. SOCC'13, 2013.

[9] J. Du, C. Iorgulescu, A. Roy, and W. Zwaenepoel, "GentleRain: Cheap and Scalable Causal Consistency with Physical Clocks," in *Proceedings 5th ACM Symposium on Cloud Computing*, ser. SOCC '14, 2014.

[10] U. Fritzke, Jr. and P. Ingels, "Transactions on partially replicated data based on reliable and atomic multicasts," in *Proc. of the Intl. Conference on Distributed Computing Systems (ICDCS'01)*, 2001, pp. 284–291.

[11] S. Gilbert and N. Lynch, "Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services," *SIGACT News*, vol. 33, no. 2, pp. 51–59, 2002.

[12] J. Gray, P. Helland, P. O'Neil, and D. Shasha, "The dangers of replication and a solution," in *Proc. of ACM SIGMOD'96*, 1996, pp. 173–182.

[13] J. Holliday, D. Agrawal, and A. El Abbadi, "Partial database replication using epidemic communication," in *Proc. of the Intl. Conference on Distributed Computing Systems (ICDCS'02)*, 2002.

[14] C. Li, D. Porto, A. Clement, J. Gehrke, N. Preguiça, and R. Rodrigues, "Making geo-replicated systems fast as possible, consistent when necessary," in *Proc. of USENIX OSDI'12*, 2012, pp. 265–278.

[15] 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 *Proc. of the 23rd ACM SOSP*, 2011, pp. 401–416.

[16] ——, "Stronger semantics for low-latency geo-replicated storage," in *Proc. of USENIX NSDI'13*, 2013, pp. 313–328.

[17] V. Padhye, G. Rajappan, and A. Tripathi, "Transaction Management using Causal Snapshot Isolation in Partially Replicated Databases," in *Proceedings of the IEEE Symposium on Reliable Distributed Systems (SRDS'2014)*, 2014.

[18] S. Peluso, P. Romano, and F. Quaglia, "Score: a scalable one-copy serializable partial replication protocol," in *Proc. of Middleware'12*, 2012, pp. 456–475.

[19] N. Schiper, R. Schmidt, and F. Pedone, "Optimistic Algorithms for Partial Database Replication," in *Proc. of OPODIS*, 2006, pp. 81–93.

[20] N. Schiper, P. Sutra, and F. Pedone, "P-store: Genuine partial replication in wide area networks," in *IEEE SRDS*, 2010, pp. 214–224.

[21] D. Serrano, M. Patiño Martínez, R. Jiménez-Peris, and B. Kemme, "Boosting database replication scalability through partial replication and 1-copy snapshot isolation," in *Proceedings of PRDC.07*, 2007.

[22] A. Sousa, F. Pedone, R. Oliveira, and F. Moura, "Partial replication in the database state machine," in *Proc. of NCA'01*, 2001.

[23] Y. Sovran, R. Power, M. K. Aguilera, and J. Li, "Transactional storage for geo-replicated systems," in *Proc. of ACM SOSP*, 2011, pp. 385–400.