Causally Coordinated Snapshot Isolation for Geographically Replicated Data

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

***Abstract*—We propose a Snapshot Isolation based transaction execution and consistency model, referred to as *causally-coordinated snapshot isolation*, for geographically replicated data. The data replication is managed through asynchronous update propagation. Our approach provides snapshot-isolation model over multiple sites and ensures causal ordering of transactions. We present here an efficient protocol for precisely capturing the causal data dependencies of transactions and ensuring the causal ordering based on these dependencies when applying transactions' updates at remote sites. Through experimental evaluations, we demonstrate the benefit of this protocol over an alternative approach for providing causal consistency for geo-replicated data. We further extend this model to support *session consistency guarantees* such as *read-your-writes* and *monotonic-reads*. Additionally, we provide a notion of *group-session* where a group of users are involved in a collaborative session. We provide various *group-session consistency* guarantees for users collaborating in a group. We present the mechanisms for providing these session consistency guarantees and evaluate their performance.**

I. Introduction

We address here the problem of providing transaction support for data replicated across geographically distributed sites. Our transaction model uses Snapshot-Isolation (SI) [1]. There are several issues related to scalability and performance that need to be addressed while applying the SI transaction model to a geographically distributed environment. The first issue is related to managing data replication across multiple geographically distributed sites. Strong consistency of data replicated across multiple sites, which would require synchronous update propagation, is typically not practical in wide-area setting. Due to this limitation, various geo-replicated data management systems [2] use asynchronous update propagation. In the context of SI, this means that transactions may see an older database snapshot. The second issue is related to global ordering of transactions. Under the SI model, transactions are assigned monotonically increasing timestamps. This requires a global sequencer mechanism for ordering transactions. Such a global sequencer in wide-area environments can become a performance bottleneck.

Recently, the work presented in [10] has recognized the above limitations and proposed a weaker snapshot isolation model called Parallel Snapshot Isolation (PSI) [10]. In the PSI model, transactions execute at a single site and are ordered within the site using a local sequence number. Transactions are propagated asynchronously to remote sites. The model does not enforce a global ordering of transactions, however, it ensures causal ordering while applying transactions at remote sites. This enhances the scalability and performance of the system, while still providing a useful consistency model to applications. Our transaction model, referred to as the *Causally-coordinated Snapshot Isolation (CSI)* is based on PSI, however, it differs from PSI with respect to the approach for ensuring causal consistency. The PSI scheme [10] can lead to false causal dependencies, which can unnecessarily delay the application of a transaction's updates at remote sites. Due to these delays, a remote site would see an older snapshot of data even though the updates are propagated at that site. This can also increase the number of transaction aborts due to *write-write* conflicts as the transactions at the remote site would observe an older snapshot. In contrast to PSI, we propose an approach which captures true causal dependencies of transactions to eliminate such false dependencies.

We further extend the CSI model to support session consistency guarantees [11] for a user session. Such guarantees are often required to ensure that a particular user session sees a consistent view of data. Our basic transaction model ensures session guarantees as long as the user is connected to a single site throughout her session. This is because all the transaction updates performed at the local site are made immediately visible to the subsequent transactions. However, if a user connects to a different site, she may not see the recent updates if those updates are not yet propagated there. We describe the mechanisms to support session guarantees under the CSI model. Furthermore, we propose a consistency model useful for multi-user group applications, such as collaborative editing, workflow applications, or multi-player games. For this purpose, we propose a consistency model for a *group session* involving operations by group members on some shared data. We propose various consistency levels for such group sessions and describe the mechanisms for implementing them under our CSI model.

In the next section we describe the CSI model. In Section III, we discuss mechanisms for providing session consistency guarantees for single-user sessions. In Section IV, we describe the group session model and its implementation. Evaluations of the proposed models and mechanisms are presented in Section V. Next we discuss the related work followed by the conclusions.

1060-9857/12 $26.00 © 2012 IEEE
DOI 10.1109/SRDS.2012.15

IEEE computer society

II. CAUSALLY-COORDINATED SI (CSI) MODEL

We present here the Causally-coordinated SI model for multi-site transaction management. The CSI model ensures the following properties. First, transactions read from a consistent (but possibly an older) snapshot, i.e. the transactions observe only the updates made by committed transactions. Second, when two or more concurrent transactions update a common data item, only one of them is allowed to commit. Third, if updates of a transaction are visible at a site, then all updates that causally precede it are also visible.

A. Background: Snapshot Isolation Model

Snapshot isolation (SI) model [1] is based on multi-version concurrency control where a data item version is assigned the timestamp of the transaction that created the version. When a transaction T_i's execution starts, it obtains the timestamp of the most recently committed transaction. This represents the *snapshot timestamp* TS_s^i of the transaction, and a read operation by the transaction on a data item returns its most recent committed version up to this snapshot timestamp. When T_i commits, it is assigned a monotonically increasing commit timestamp TS_c^i. The commit timestamps of transactions reflect the logical order of their commit points. A transaction T_i commits only if there is no *write-write (ww)* conflict, i.e none of the items in its write-set have been modified by any committed concurrent transaction T_j i.e. $TS_s^i < TS_c^j < TS_c^i$.

B. System Model

The system consists of multiple geographically distributed database sites S_i such that $i \in (1..n)$. Each site is identified by a unique $siteId$. Each site has a local database that supports multi-version data management and transactions. It can be either an RDBMS system or a key-value store with transaction support [9]. Users connect to the site closest to them and execute transactions on local database. Data items are replicated at all the sites. For each data item, there is a designated *conflict resolver site* which is responsible for checking for ww conflicts for that item. Transactions can execute at any site. Read-only transactions can be executed locally without needing any coordination with remote sites. Update transactions need to coordinate with conflict resolver sites for ww conflict checking for the items in their write-sets.

C. CSI Model

As noted earlier, the total ordering on transactions is not enforced. This eliminates the need of a global sequencer. Instead, a transaction is assigned a commit sequence number $seqno$ from a monotonically increasing local sequence counter maintained by its execution site. Thus, the commit timestamp for a transaction is a pair <*siteId, seqno*>. Similarly, a data item version is identified by a pair <*siteId, seqno*>. The local sequence number is assigned only if the transaction is guaranteed to commit, i.e. only if there is no ww conflict. Thus, there are no gaps in the sequence numbers of the committed transactions. A transaction first commits locally and then its updates are propagated to other sites asynchronously. A remote site, upon receiving a remote transaction's updates, applies the updates provided that it has also applied updates of all the causally preceding transactions. The transactions from a particular site are always applied in the order of their sequence numbers. All the updates of a transaction are applied to the local database as an atomic operation, which also includes updating a local vector clock.

Each site maintains a vector clock, which we denote by VC, indicating the transactions from other sites that it has applied to the local database. Thus, a site S_i maintains a vector clock VC_i, where $VC_i[j]$ indicates that S_i has applied the updates of all transactions from S_j up to this timestamp, moreover, S_i has also applied all the other updates that causally precede these transactions. In the vector clock, $VC_i[i]$ is set to the sequence number of the latest transaction committed at S_i.

Snapshot-based access: A transaction p executing at site S_i is assigned, when it begins execution, a *snapshot timestamp vector* VT_s^p, which is set equal to the current vector clock VC_i value. When p performs a read operation for item x, we determine the latest version of x that is visible to the transaction according to its snapshot timestamp vector. For each data item x, we maintain a version log which basically indicates the order of the versions. When p performs a read operation on x, we check for every version $< j,n>$, starting from the version that is applied most recently, if the version is visible in the transaction's snapshot, i.e. if $VT_s^p[j] \geq n$. We then select the latest version that is visible in p's snapshot. All writes by p are kept in a local buffer until the commit time. The execution protocol is detailed in Algorithm 1.

Algorithm 1 Execution protocol for transaction p at site S_i

```
function BEGIN(p)
    VT_s^p ← VC_i
function READ(itemId)
    for each v ∈ versionLog(itemId) do
        /* performed in descending order of versions */
        if VT_s^p[v.siteId] ≥ v.seqno then return v.data
function WRITE(itemId, data)
    put itemId, data in local buffer of p
```

Commit protocol: When transaction p is ready to commit, it performs the commit protocol as shown in Algorithm 2. If p has modified one or more items, then it performs ww conflict checking using a two-phase commit (2PC) protocol with the conflict resolver sites responsible for those items. In the prepare message to each site, p sends VT_s^p and the list of items it has modified for which that site is the conflict resolver. Each site checks, if the latest versions of those items are visible in p's snapshot and that none of the item is locked. The locking is performed to avoid conflicts with any concurrent conflict checking operations by other transactions. If this check fails, then the resolver sends a 'no' vote. Otherwise, it locks the corresponding items and sends a 'yes' vote. If p receives 'yes' votes from all conflict resolvers, p is assigned a monotonically increasing local sequence number by p's local

site, obtained from $localSequencer$ counter. First, p commits locally, applying the updates to the local database. The local site's vector clock is advanced appropriately. It now sends a commit message, containing the sequence number, to all the conflict resolvers. Otherwise, in case of any 'no' vote, p is aborted and an abort message is sent to all the conflict resolvers. Upon receiving a commit or abort message, a conflict resolver releases the locks, and in case of commit it records the new version number as a 2-tuple: ($<siteId, seqno>$). After performing these operations, the local site asynchronously propagates p's updates to all the other sites.

Algorithm 2 Commit Protocol for transaction p at site S_i

/* [..] denotes an atomic region */
function COMMIT(p)
  $sites \leftarrow$ conflict resolver sites for items $\in p.writeset$
  for each $s \in sites$ **do**
    $itemList \leftarrow$ list of $itemId \in p.writeset$, such that resolver($itemId$)= s
    send prepare message to s : ($itemList$, VT_s^p)
  if all votes are 'yes' **then**
    [$p.seqno \leftarrow S_i.localSequencer{+}{+}$
    apply p's updates at S_i
    $VC_i[i] \leftarrow p.seqno$ **]**
    for each $s \in sites$ **do**
      { send to s a commit message:
      ($itemList$, i, $p.seqno$) }
    propagate asynchronously to all sites:
    (VT_e^p, i, $p.seqno$, $p.writeset$)
  else
    send abort message to each $s \in sites$
/* Functions executed by Conflict Resolver */
function RECVPREPARE($items$, VT_s)
  if $\forall\ x \in items$: if the latest version of x is visible in $VT_s \wedge x$ is unlocked **then**
    lock all $x \in items$ and vote 'yes'
  else
    vote 'no'
function RECVCOMMIT($items$, $siteId$, $seqno$)
  for each $x \in items$ **do**
    record version $<siteId,seqno>$ in versionLog(x)
    release lock on x
function RECVABORT($items$)
  release locks on all $x \in items$

Update propagation: For ensuring causal consistency, p's updates are applied at remotes sites only when all the causally preceding transactions are applied. PSI model uses the transaction's snapshot vector timestamp VT_s to indicate causal dependencies. Thus, p's updates are applied at a remote site only when that site's vector clock is advanced up to VT_s^p, so that all the events that were visible when p started its execution are visible at the remote site when p is applied there. However, this can induce false dependencies, as not all the updates that were visible at p's execution site when p started are necessarily seen by p. Therefore, we propose an alternative approach that precisely captures a transaction's causal dependencies by considering only its read and write sets. We define the *effective causal snapshot*, which indicates, for each site, the latest event from that site which is 'seen' by the transaction based on the items it read or modified. In other words, we capture causal dependencies with respect to a transaction rather than a site. The effective causal snapshot for a transaction p, executed at a site S_i is defined as a vector timestamp denoted by VT_e^p, and it is determined as follows. $VT_e^p[i]$ is set equal to n-1 where n is p's sequence number. This indicates that p can be applied only when the transaction immediately preceding p at site S_i is applied. The other elements of VT_e^p, i.e. those corresponding to the remote sites, are determined as follows:

$$\forall j; j \neq i : VT_e^p[j] = max\{seqno \mid \exists x \ s.t.(x \in read\text{-}set(p) \vee x \in prevwrites(p)) \wedge (version(x) = < j, seqno >)\}$$

Here, $prevwrites(p)$ is the set of latest versions visible at site S_i for the items that p modified. If this information is not included, then it may happen that for a particular data item x modified by p a remote site may store the version created by p without having all the preceding versions of x. We call it the *missing version* problem. This can violate the basic multi-version semantics of snapshot-based access in cases such as time-travel queries, which read from a specific older snapshot. It also complicates the version ordering logic described above. It should also be noted that the *effective causal snapshot* vector for a transaction is determined at the end of the transaction execution, and therefore the information about the read/write sets is needed only after the transaction execution has finished.

The update propagation protocol uses the VT_e value of a transaction while propagating its updates. Upon receiving updates, the remote site compares its vector clock with VT_e vector to ensure that an update is applied at that site only when it has seen all the causally preceding updates. On applying the updates, the vector clock of the site is advanced appropriately.

III. SINGLE USER SESSION CONSISTENCY MODEL

In this section we consider session guarantees with respect to a *single-user session*. A *single-user session* cosnists of a sequence of transactions. During a session the user may connect to different sites. The session model provides primitives for suspending a session at a site and resuming it later at a different site. We support the following four session guarantees [11].

- *Read-Your-Writes (RYW)*: A read operation on a data item must read a version that is equal to or newer than the most recent write on that item in that session.
- *Monotonic-Read (MR)*: A read operation on a data item must never see a version of that item which is older than the version seen by any previous read operation.
- *Monotonic-Writes (MW)*: A write operation on a data item follows any preceding writes on that item in the session.
- *Write-Follows-Reads (WFR)*: A write operation on a data item is performed on a version that is equal to or newer than the version seen by a preceding read operation.

The MW and WFR guarantees are supported, not only on a session level but across the entire system, through the ww conflict checking protocol of the basic SI model. The RYW and MR guarantees are supported implicitly when the user is connected to a single site during the entire session. However, these guarantees may be violated if the user connects to a different site and the updates seen at the previous site have not been propagated there.

Our basic model for supporting session guarantees is as follows. For a session, we maintain the *session state* as the events (i.e. transactions) seen by the session. We maintain the state of session s as a vector clock, called *session vector clock* and denoted by $\mathcal{C}_s$, in which each element corresponds to a site and identifies the sequence number of the latest event corresponding to that site that must be visible in order to support the required session guarantees. A simple approach to determine the session state vector is to consider the current vector clock value of the site where the session is executing. However, this can induce false dependencies, i.e. the events not seen by the session. In our approach we consider only the events seen by the session. This includes updates seen by the read operations (we denote it as read-event set) and updates performed during the session (denoted as the write-event set). The read-event set R_s of session s is a set of 2-tuples $<j, n>$ such that a read operation in the session read the version created by a transaction executed at site j with sequence number n. Similarly, write-event set W_s is a set of 2-tuples $<j,n>$ such that the session executed an update transaction at site j with sequence number n. We then determine the session vector clock $\mathcal{C}_s$ as follows:

$\forall j;\ 1 \leq j \leq numSites$: $\mathcal{C}_s[j] = max\{seqno | s.t.$
$< j, seqno > \in R_s \vee < j, seqno > \in W_s\}$

The session vector clock is itself maintained in the database as a *session object* like any other database item. A sequence of modifications to this object are committed as a transaction and propagated to other sites. When a user connects to some other site, say S_j, the $\mathcal{C}_s$ value is obtained by reading the session object. For this, S_j contacts the conflict resolver for the session object to obtain the latest version of that object. Then, S_j compares its vector clock VC_j with $\mathcal{C}_s$ to check if the required events are visible. If VC_j is advanced up to $\mathcal{C}_s$, then it means that both the updates read and written by the session are visible at S_j. Thus, both MR and RYW guarantees can be supported, and hence the session can be resumed at S_j. If S_j is not enough up to date, then we need additional mechanisms to provide session guarantees. In our approach a site pulls the required updates to advance itself up to the session vector clock, so that the session can be resumed at that site. To do this, the site first determines the updates that need to be pulled, and then contacts the corresponding sites.

IV. Group Session Consistency Model

A *group session* involves a group of users/clients, referred to as the *group members*, accessing a shared pool of data items, called the *session object space*. We assume that the session object space contains a well-defined set of data items and these items are not modified by users outside the group session. Furthermore, we assume that the updates on the data items outside the session object space are not relevant to the group session and need not be seen by the session. The group members may be connected to different sites and they may concurrently execute transactions.

For each member in the group, we store the information about the site the member is currently connected to. It may be possible that more than one are connected to a single site. For a member in a group, all the single-user level session guarantees are provided. Additionally, we propose the following consistency models for group sessions.

A. Strong Consistency Model

In strong consistency model we provide the following guarantee for the read operations in a group-session transaction.

- *Read-Latest-Global (RLG)*: A read operation on an item reads its latest global committed version at the transaction's start time, created in the group session.

This model implicitly supports the RYW and MR session guarantees for the single-user level. Supporting this model requires synchronous update propagation before locally committing the transaction. However, it is performed only for the transactions in the group session and only among the sites involved in the group session.

We take advantage of the fact that the effect of the transactions outside the group session need not be seen by the group session, since they do not modify any items in the session object space. For a group session, we maintain a *group-session snapshot vector ($\mathcal{C}_g$)*, which is different from the sites' vector clock VC. Transactions in the group session see the database snapshot according to $\mathcal{C}_g$ instead of their site's current vector clock. Thus, for transaction p the snapshot timestamp VT_s^p is set equal to $\mathcal{C}_g$. The $\mathcal{C}_g$ value is maintained synchronously, i.e all sites in the group session have the same view of $\mathcal{C}_g$. The $\mathcal{C}_g$ value would be advanced only due to transactions in the group session. Hence, synchronous propagation of only the group session transactions ensures that the $\mathcal{C}_g$ value is kept consistent across the group session sites. The non-group-session transactions are propagated asynchronously.

There are two advantages of maintaining a separate snapshot timestamp vector for a group session. First,, it ensures that all group members see the same consistent state. Second,, it allows to eliminate false dependencies caused by non-group-session transactions. We elaborate this below. Suppose a site S_j in a group session receives the updates of a group-session transaction p from site S_i with sequence number n. It may be possible that S_j has not received some preceding, non-group-session, transaction p' from S_i with sequence number k, $k < n$. In this case, since the updates of p' are not relevant to the group session, we can make the updates of p visible to the group session without applying the updates of p'. This is done by setting $\mathcal{C}_g[i]$ to p's sequence number. However, with respect to S_j's vector clock, since the vector clock elements must be always advanced sequentially, $VC_j[i]$ cannot be advanced until p' is received. Thus, maintaining a separate

snapshot timestamp vector for group sessions allows to avoid unnecessary delays caused by such false dependencies.

B. Weak Consistency Model

The purpose of this consistency model is to eliminate the need of synchronous propagation, but provide certain primitives for users to read a consistent state whenever desired. We refer to this as *on-demand consistency*. Specifically, we provide a *sync* primitive for user-driven synchronization, either on all data items or a specific data item. The *sync(x)* primitive fetches the most recent version of item x created in the group session, and *syncSession()* fetches the recent versions for all items. In order to guarantee the snapshot isolation semantics, these operations should be executed only at the time of starting a transaction.

When a group member connected to a site S_i performs $sync(x)$, S_i contacts the conflict resolver site for x, which has knowledge of the site that has the most recent version of x. The $syncSession$ operation is performed a bit differently. In this case, S_i contacts all the sites in the group session and sends them it's vector clock VC_i value. Each site then checks if it has created any version for any item that is not visible in VC_i and sends the required updates to S_i, if any.

Mixed Consistency Execution Mode: In this execution mode, different users may opt for different consistency levels. For example, a mobile user may opt for a weaker consistency level and periodically synchronize with other members. Moreover, a user may opt for strong consistency for some critical items whereas for other items it may tolerate weak consistency level. For this, we provide a *setUpdateMode(x, SYNC/ASYNC)* primitive to define consistency level per item. The *setUpdateMode* operation indicates whether updates of x should be synchronously propagated.

V. Experimental Evaluations

Our primary goals in these evaluations are as follow: (1) compare the CSI approach of ensuring causal consistency to that of PSI; (2) measure transaction performance in terms of response times; (3) measure the overhead of providing session consistency guarantees; (4) measure the overhead of providing strong group session consistency.

Experiment setup: We deployed our system on 30 PlanetLab nodes, distributed over the Internet. Each node acts as a single site in our experiments. The geographic distribution of nodes is as follow: 13 nodes from US/Canada, 9 nodes from Europe, 6 nodes from Asia, and 2 nodes from Aus/NZ. The average *ping* latency between the sites was 158 ms with standard deviation of 140 ms. The database contained 10,000 items replicated at each site. Each site executed 6000 transactions per minute and transactions were propagated periodically with the period value randomly selected between 1 second to 1 minute. In our experiments, we varied the size of the combined read-write sets of a transaction from 1 to 10, with equal number of items in the read-set and the write-set.

PSI vs CSI: To measure the benefits of the causal consistency model of CSI over that of PSI, we first measured the number of false dependencies induced by PSI. For this, we measured for each transaction, the total difference between the vector clock element values for VT_s and VT_e. Table I shows this data. In this table, f denotes the average number of false dependencies. We observed that the number of false dependencies tends to decrease with increase in number of items in read/write set. This is because for a transaction the actual dependencies are likely to increase with increase in the number of items in its read/write set. Thus, as the actual dependencies increase the number of false dependencies tends to decrease. Note that, not all the false dependencies would cause delays in application of updates at the remote site, as it depends on that site's vector clock. We measured the delays in applying updates in two ways. The first measure, denoted by M_1 is the duration between the time a site receives an update transaction till the time the transaction's updates are applied at that site. The second measure, denoted by M_2, is the number of transactions not seen by the site that are required in order to apply the given updates. This indicates the number of transactions that need to be applied before the given updates can be applied. We show the average values of M_1 and M_2 in Table II for PSI and CSI approaches. The average values were computed for sample size of approximately 3.5 million transactions. We observe that the update delays are negligible when CSI is used. Note that, the values for these two measures depend on the transaction rate and update propagation delay. However, we observed that CSI incurs significantly lower delays in applying updates compared to PSI.

size of read-write set	1	2	4	6	8	10
f	14804	14018	13612	13461	12917	12384

TABLE I
Average Number of false dependencies in PSI

Number of items per transaction	1	2	4	6	8	10
M_1 (update delay in ms)						
PSI	1123	1303	2234	2359	2019	1725
CSI	0	0	12	19	27	119
M_2 (number of missing events)						
PSI	1987	3040	2310	2591	2876	2342
CSI	0	0	42	48	97	180

TABLE II
Update Delays in PSI vs CSI

Transaction response time: We observed that, in case of read-only transactions, as they do not require any remote coordination, the response times were typically below 20 ms. For update transactions, the response times are typically dominated by delays due to two-phase-commit. Figure 1 shows the average and 90-percentile response times for update transactions with varying number of conflict resolver sites. In case of update transactions with local conflict resolver, the response times typically ranged between 20-30 ms.

Overhead of session consistency guarantees: In this case, our goal is to measure the delays incurred in resuming a session when a user migrates to a different site. Obtaining

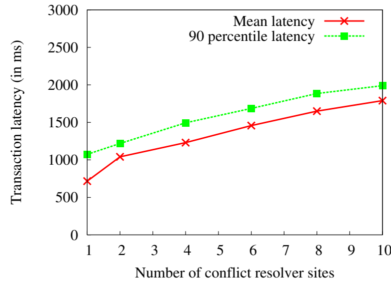

Fig. 1. Transaction response time with 2PC

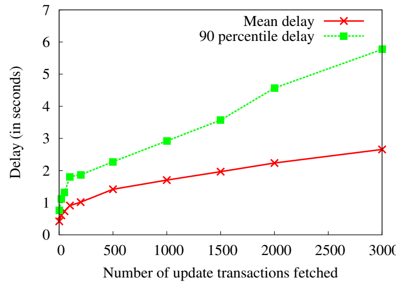

Fig. 2. Overhead of pulling updates for *sync*

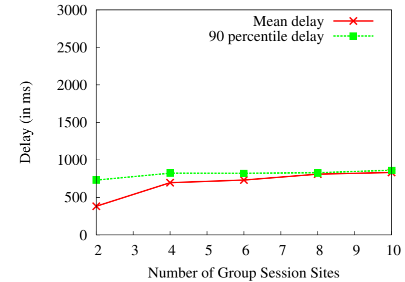

Fig. 3. Overhead of synchronous propagation

the most recent version of the session object typically requires 200-300 ms. This delay depends largely on the update propagation rate. Therefore, for better insight, we performed micro-benchmarking of our mechanism for pulling the updates and measured its overhead. Figure 2 shows the time required to pull updates from a random site for varying number of transactions. In this case, each transaction updated 10 items with the total size of the transaction's updates being 1KB.

Overhead of group session consistency guarantees: Figure 3 shows the delays incurred due to synchronous propagation with varying number of group session sites. This data indicates the relative increase in the response times for group session transactions. In this case, since the updates are propagated in parallel to different sites, the delays are essentially dominated by the slowest peer. For weak consistency level, users basically utilize the *sync* primitive. Depending on the number of updates to be fetched, the latency of the *sync* operation would vary as shown in Figure 2.

VI. Related Work

Database replication using Snapshot Isolation has been studied widely in the past [3], [4], [5], [6], [7], [8], [10], [12]. Many of the proposals for SI-based database replication using update-everywhere model, such as [4], [7], [12], used eager replication with atomic broadcast to ensure that the replicas observe a total ordering of transactions. Serializable snapshot isolation based replication management is presented in [5]. COPS [8] provides causal consistency, but does not provide transaction functionality except for snapshot-based read-only transactions. PSI [10], which is the most closely related work to ours, provides transaction functionality with asynchronous replication and causal consistency. Our work differs from PSI in two ways. First, we provide an efficient protocol for ensuring causal consistency. Second, we provide session consistency guarantees for single-user as well as group session model. Session consistency models have been investigated in the context of the Bayou system [11]. In contrast to the approach in Bayou, which requires maintaining the entire read and write sets of the session, our model uses a vector clock based approach to support session guarantees. Session guarantees under SI model using the primary-backup approach are investigated in [3]. In contrast to that work, we support update-everywhere model and also provide a group session model for collaborative applications.

VII. Conclusion

We have presented here the CSI model for executing transactions on geographically replicated database systems. This model uses asynchronous replication for scalability and ensures causal ordering of transactions. The approach taken here avoids any false causal dependencies in propagating updates. We have shown the performance benefits of CSI over the PSI approach. Building upon the CSI framework, we have developed models for different consistency levels suitable for session level interactions in single-user as well as group-based applications. We have presented here a set of mechanisms that can be used by an application to guarantee the desired consistency level.

References

[1] 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.

[2] 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.

[3] K. Daudjee and K. Salem, "Lazy database replication with snapshot isolation," in *Proc. of the VLDB*, 2006, pp. 715–726.

[4] S. Elnikety, F. Pedone, and W. Zwaenepoel, "Database replication using generalized snapshot isolation," in *24th IEEE Symposium on Reliable Distributed Systems*, 2005, pp. 73 – 84.

[5] H. Jung, H. Han, A. Fekete, and U. Roehm, "Serializable Snapshot Isolation for Replicated Databases in High-Update Scenarios," in *VLDB*, 2011.

[6] B. Kemme and G. Alonso, "A new approach to developing and implementing eager database replication protocols," *ACM Trans. Database Syst.*, vol. 25, pp. 333–379, September 2000.

[7] Y. Lin, B. Kemme, M. Patiño Martínez, and R. Jiménez-Peris, "Middleware based data replication providing snapshot isolation," in *Proc. of the ACM SIGMOD*, 2005, pp. 419–430.

[8] 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.

[9] V. Padhye and A. Tripathi, "Scalable Transaction Management with Snapshot Isolation on Cloud Data Management Systems," in *Proc. of IEEE 5th Intl. Conference on Cloud Computing*, 2012.

[10] 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.

[11] D. Terry, A. Demers, K. Petersen, M. Spreitzer, M. Theimer, and B. Welch, "Session guarantees for weakly consistent replicated data," in *PDIS'94*, 1994, pp. 140–149.

[12] S. Wu and B. Kemme, "Postgres-R(SI): Combining Replica Control with Concurrency Control Based on Snapshot Isolation," in *Proc. of IEEE ICDE*, 2005, pp. 422–433.