2016 IEEE 2nd International Conference on Collaboration and Internet Computing

A Transaction Model with Multilevel Consistency for Shared Data in Distributed Groupware Systems

Anand Tripathi
Department of Computer Science & Engineering
University of Minnesota, Minneapolis, Minnesota, 55455 USA
Email: tripathi@umn.edu

***Abstract*—In groupware systems a broad range of requirements for user coordination and data consistency need to be supported. The notions of event causality and user awareness are central in such requirements. Traditional transaction models supported in general purpose database management systems with strong consistency guarantees have been found to be unsuitable for groupware systems. Weaker models for data consistency are needed for user awareness and cooperative activities. Objects in the shared workspace need to be managed with different consistency guarantees. Towards such requirements, we examine here the applicability of a distributed transaction management model which supports multilevel consistency. The consistency levels supported in this model include serializable transactions for strong consistency and weaker consistency models such as Causal Snapshot Isolation (CSI), CSI with commutative updates, and CSI with asynchronous updates. We review the coordination and data consistency requirements in groupware systems. We show using two examples how replicated shared data in distributed groupware systems can be managed with multiple consistency levels using this model.**

I. Introduction

Groupware systems support collaborative activities of a group of users towards some common shared goals and tasks. The goals of groupware applications can be wide-ranging such as supporting an engineering design project team, collaborative document preparation, or activities of a social group. Such systems tend to be distributed in structure with the group participants and their computing systems connected by a network, and the participants often working offline.

Three decades of research in the area of groupware systems has identified several important requirements for user coordination, user awareness, and concurrency control in updating of shared objects. It is well-recognized that the transaction models provided by the traditional database management systems are not suitable for groupware applications. Such transaction models are designed to isolate the activities of one user from another in the sense that the updates of a transaction become visible to the other users only on its commitment. On the other hand, the requirement of supporting awareness [8] where a user can observe results of in-progress activities of others is important for effective collaboration in many situations [35]. In groupware applications, transactions can be of long durations. Supporting strong consistency requirement for managing shared data in the presence of such transactions can negatively impact user productivity and awareness. If locking schemes are used for concurrency control, then some objects may be locked for long durations, which can potentially block activities of some group members. Use of optimistic concurrency control methods may lead to high probability of long transactions getting aborted during the validation phase, thereby resulting in lost work. Such observations have led the groupware researchers to adopt various approaches such as fine-grain locking, support for concurrent operations on shared objects using operation transformation, and use of short transactions within long duration transactions with weaker isolation guarantees. Most importantly, it has been observed that different kinds of data in a collaboration space require different levels of consistency guarantees.

One approach for managing shared data objects pertaining to the group activities is to replicate them on the participants' computing systems for performance and offline operations. Management of replicated data in distributed systems poses fundamental trade-offs between data consistency, scalability, and availability [12]. Transactions with strong consistency for serializability may impose scalability and availability limitations due to distributed coordination. Replication management protocols with weaker consistency models provide lower latencies for transactions and high availability, but they guarantee only eventual consistency [6] or causal consistency [29], [20]. Causal consistency provides more useful semantics than eventual consistency and it can be supported under asynchronous replication. With such considerations, several distributed systems [6], [5], [29], [20] have adopted weaker consistency models for updating replicated data.

In this paper we study the applicability of a multi-consistency transaction model [32] in groupware applications for managing replicated shared data with different levels of consistency. We developed this multilevel transaction model in a more general context with the observation that rather than providing a single consistency model for transactions in replicated data management systems, it is desirable to support transactions with different levels of consistency guarantees. This allows application developers to use suitable transaction models for data objects with different consistency requirements. Thus one can use low latency transaction management models for data items with weak consistency requirements, but at the same time use transactions with strong consistency for critical data. This enables the application developers to make suitable trade-offs between data consistency and interactive response time in managing replicated data in groupware

978-1-5090-4607-2/16 $31.00 © 2016 IEEE
DOI 10.1109/CIC.2016.39

IEEE computer society

applications.

The transaction management model [32] considered here for supporting multilevel consistency in groupware applications is based on Causal Snapshot Isolation (CSI) [23], which guarantees causal consistency of the snapshots observed at a site. The CSI [23] model forms the core building-block for the multilevel consistency model, which simultaneously supports transactions with different consistency levels. The consistency levels supported include serializability for strong consistency, and weaker consistency models which include CSI, CSI with commutative updates, and CSI with asynchronous concurrent updates. Consistency requirements are associated with data items, and data is organized in a hierarchy which is based on consistency levels and the associated transaction management protocols. The focus of our work here is on examining how this model can be utilized in groupware applications.

In the next section we discuss the related work in groupware systems in relation to concurrency control and coordination. In Section III we review and summarize the unique requirements of concurrency control in groupware systems and the need for supporting weaker models of consistency and isolation levels in such systems. Section IV presents an overview of the CSI model. Section V presents the multilevel consistency model. In Section VI we consider two examples of groupware applications and show how the replicated shared data objects in these systems can be managed using the multilevel consistency model.

II. Related Work

Groupware systems have been developed and used in a broad range of collaborative environments with different modes of coupling between user interactions, which can be broadly classified into two categories: real-time systems with tightly coupled synchronous online interactions, and non-real-time systems with asynchronous and offline interactions. In groupware systems, the importance of user awareness is highlighted in a seminal work by Dourish and Bellotti [8]. Some of the facets of awareness are noted in [15]. These include information pertaining to users, their groups, permitted access to shared objects, and view of their current activities.

Some of the early groupware systems provided simple mechanisms for coordination such as the floor-control model with users taking turns as in [26]. GROVE [10] was one of the first systems to support concurrent real-time editing operations on replicated shared documents in a distributed environment by merging them using *operation transformation* [31]. A range of issues and requirements of concurrency control and coordination in real-time groupware systems were identified in the context of GroupKit [14] system for collaborative drawing and editing. Similar kind of requirements in hypertext based document editing groupware systems are presented in [35]. These include need for fine-grain locking, use of short transactions, long transactions with weaker isolation levels, and use of notifications for awareness support. Concurrency control methods utilizing fine-grained locks in accessing hierarchically structured objects were investigated in Suite [21], where different components of a tree-structured object can be concurrently locked by different users. The approach there is based on the fine-grain locking model presented in [13].

Several groupware systems such as GroupKit [14], Suite [21], DistView [24] have based their architecture on replicated shared data, utilizing different approaches for concurrency control. GroupKit used optimistic locking, i.e. a lock requester could proceed with operations on an object before acquiring lock on it. It further considered two options; one is *fully-optimistic* where updated objects are made visible before lock is acquired, and the other option called *semi-optimistic* releases the updated object only after acquiring the lock. Fully-optimistic schemes can introduce complexity in design to deal with cascaded rollbacks if the lock request fails. The semi-optimistic model conceptually follows the well-known optimistic concurrency control method of Kung and Robinson [17]. In that method, a transaction reads committed data, buffers its updates locally, then goes through a validation phase to check that none of the objects in its read and write sets have been updated by any concurrent transaction. The updates are made visible (written to the global storage) only on successful validation. We refer to this as the *optimistic model* in our work. The approach in DistView is *non-optimistic*, it blocks a user until lock is acquired on the object to be updated. Suite [21] uses fine-grain locking and defines merge procedures for resolving conflicting operations. The Corona [15] system is based on a reliable communication protocol and publish-subscribe mechanisms for group coordination. The system presented in [30] adopts optimistic concurrency control method for updating replicated data. It uses virtual time which is used in conflict detection and for maintaining timestamped versions of an object. A transaction is committed if no conflicts are detected. In this system, users can choose to see only committed version of a data or uncommitted updates on an object.

In contrast to the systems noted above, the approach in GROVE [10] permits concurrent operations without locking, and applies them on different replicas in possibly different order after appropriate *operation transformation* to preserve *intention*. In this context, the notion of operation commutativity [34] is useful in supporting concurrent operations on an object without requiring synchronization. Thus commutative operations can be executed concurrently and applied in different orders on the replicas, but eventually the final state of all replicas would be the same. The topic of concurrency control using operation semantics and exploiting commutativity properties has been extensively researched in the past [22], [2], [25], [28], [9].

In the area of transaction management in databases and distributed data management systems, several recent projects have pursued the goal of simultaneously supporting different consistency models. The system presented in [3] provides mechanisms for supporting causal consistency in systems with eventual consistency. The RedBlue consistency model [19] requires commutativity analysis of operations to be performed

across a set of transactions, which can span multiple objects. The Salt [36] model requires rewriting an ACID transaction as a BASE transaction consisting of a series of *alkaline* nested transactions. Both RedBlue and Salt models require analysis and rewriting of application level transactions. The approach in [16] places data into different consistency categories and adaptively changes the category of an item based on its access patterns. Consistency categories are associated with data and not transactions. This system provides probabilistic guarantees of consistency, which may get violated at times. The multilevel consistency model which we adopt for groupware systems in this study associates consistency levels with both data and transactions, and it ensures that the consistency guarantees are always satisfied. It is based on some simple static rules, and the analysis of object operation semantics is confined to an individual object rather than a collection of objects.

III. Concurrency Control Requirements in Groupware Systems

Several groupware projects [10], [35], [14], [21] in the early 1990s highlighted the unique requirements for concurrency control mechanisms in groupware systems. Here, we briefly review and summarize these requirements.

User Awareness: Effective collaboration is facilitated when a user has greater awareness of the presence and current activities of other users. There are several aspects of awareness which can be useful; these include view of current group memberships, members currently active and online, objects being used by other members, recent updates to shared data, and ability to view updates of in-progress tasks, such as viewing a document which is being edited by another group member in an exclusive mode. The awareness of user locations can also be important in location-aware groupware systems [33], [27].

Notifications: Asynchronous event notifications are useful for coordination and supporting awareness in a pro-active manner. For example, the view of an object by a participant can be continuously updated in real-time when the object is updated. Notifications are equally useful in asynchronous groupware applications to inform a participant when an object is updated. Mechanisms are needed for applications to register for event subscriptions, specifying the object or some system state to be monitored for changes. It is noted in [35] that fine-grain notification mechanisms should be supported for monitoring of updates at object component level and lock/unlock operations. As a general requirement, any context-awareness [7] related aspect of a group application may need to be monitored and communicated through event notifications.

Multiversion Data Management: Support for multiversion data management is useful in groupware systems in several ways. It can be used to support optimistic concurrency control techniques, without requiring long duration exclusive locks for updates. While a group member is updating an object, others can view previous committed versions of it. In a long update session a series of new versions can be created for group members to observe the progress of an activity. It also enables application level rollback of an object to some previous state. Multiversion data management also allows users to view snapshots of old data to review and understand the progression of an activity.

Fine-grain Locking: The benefits of fine-grain locking for supporting higher concurrency in database systems were originally detailed in [13]. In groupware applications, the use of fine-grain locking and its benefits in supporting parallel activities of users are detailed in [21], [35]. Fine-grain locking mechanisms permit different components of a tree-structured object to be concurrently locked by different users to work in parallel, rather than one user locking the entire object in exclusive mode. For example, different members in a group can be permitted to concurrently edit different sections of a document.

Short vs. Long Transactions: Transactions in groupware systems can be of long durations because they are generally part of some interactive task. Locking based concurrency control for such transactions can potentially block the activities of other participants for long durations, and that can have adverse effect on user experience and group productivity, Shared objects inaccessible for a long duration can also negatively impact the awareness requirement. Long transactions with optimistic concurrency control schemes [17] may face higher probability of aborts during the validation phase, thereby resulting in lost work and undesirable experience for the users. However, optimistic approaches can permit group members to view the current or the last committed state of an object being modified. Update models based on optimistic techniques have been adopted in several systems [14], [30], [21].

User-level Locking and Application-defined Transactions: It is generally desirable to structure a long update activity as a sequence of short transactions which can be part of an application-controlled long transaction. The incremental updates of such short transactions can be committed and made visible to the group, thus increasing awareness. Effectively, such long duration transactions can be designed to permit a weaker isolation level than that enforced by ACID transactions in database systems. Supporting such long transactions can be achieved by application-controlled coordination mechanisms [35] such as explicit locking of objects by the users. Such locks are defined and managed by application level mechanisms, which should also support revocation of locks and recovery on transaction abort or user inactivity.

Operation Semantics-based Concurrency Control: It has been well-recognized that concurrency control techniques based

on read/write sets are general in applicability but restrictive in nature. Utilizing operation semantics to support concurrent operations is useful for achieving greater concurrency and also for eliminating the need for synchronization protocols in distributed systems. As noted in Section II, commutativity of operation is one way to achieve this. Another approach is based on operation transformation [31] methods to preserve the intent and semantics of operations when applying them in different order on different replicas. Optimistic concurrency control methods can also benefit by taking into account semantics of concurrent operations during the validation phase; rather than checking for conflicts based on read/write operations, the validation procedure can allow concurrent commutative operations.

Weak Isolation and Consistency Levels: Database transactions enforce strong isolation levels among concurrent transactions, preventing a transaction from observing intermediate states produced by another. In collaborative environments, the need to support awareness requires that users should be able to view the results of in-progress transactions by allowing weaker isolation levels. Also, total ordering of all operations on a shared object is not necessary in many situations. For example, when several participants are adding their comments on some document, the comments can be collected in any order. In some situations, eventual consistency of the set of comments may suffice. However, in some application scenarios, preserving the causal ordering of actions can be an important requirement. For example, when a group leader removes a user from the group and later sends a notification to all current group members, then the need of preserving the causality of these two actions may be critical in some cases. Certainly there can also be many cases that require total ordering of operations. For example, an auction application may need to record bids in the order in which they are made by the users.

A typical groupware application contains many objects and not all are required to be managed with strong consistency guarantees. Consider an engineering design project which is divided into several teams working on different parts or aspects of a product design. In this collaboration system, the documents concerning the master plan, specifications for the integration of different parts, team memberships, task assignments, and task completion schedule would be generally required to be updated with strong consistency using serializable transactions. Such updates are likely to be infrequent. On the other hand, within a team, some of the shared objects such as a set of design documents and design notes could be updated concurrently with causal consistency. Event notifications in the system can be maintained and communicated with causal consistency. Certain objects could be updated with even weaker models, permitting concurrent commutative updates on some objects. For example, each design team may frequently update an object maintaining an inventory of some off-the-shelf parts and their quantity required in building the target product. User activity logs can be maintained with a weaker model of eventual consistency.

IV. Building Block: Causal Snapshot Isolation

The building block for the multilevel consistency model [32] considered here is the Causal Snapshot Isolation (CSI) model [23]. The core principles underlying the CSI model come from the basic Snapshot Isolation (SI) model originally presented in [4]. The SI model is an optimistic scheme based on multi-version data maintained in update timestamp order. In this model, a transaction obtains a snapshot timestamp and it reads the latest committed version of data up to its snapshot time. Thus read-only transactions never abort. An update transaction is committed only if no concurrent committed transaction has modified any of the items in its write-set. This requires execution of a validation phase at the commit time, which is executed in a sequential order by concurrent transactions. A committed transaction is assigned a timestamp which is used for creating new versions for the data items it updates. The SI model is not suitable in distributed systems because of the sequential execution of the validation step and timestamp assignment.

The CSI model is based on a weaker form of snapshot isolation model presented in Parallel Snapshot Isolation (PSI) [29], which addressed the above noted limitation of using the SI model in distributed systems. It allows parallel execution of the validation step by each site. In the CSI model, the system consists of a set of sites in the network. Each site is identified by a unique $siteId$, S_i for $i \in (1..n)$. The database is fully replicated at each site and it supports multi-version data management using timestamps. Transactions can execute at any site. The transactions executing and committing at a site are total ordered using a local sequence counter, which is monotonically increasing. 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 them only after it has applied updates of all causally preceding transactions.

Each site maintains a vector clock with one element for each of the sites, indicating the latest update sequence number of the transactions received from that site and applied locally. This vector clock is used for maintaining causal consistency in the system. It is also used for assigning the snapshot times for local transactions. A transaction reads the latest version of the items in its read-set based on this snapshot time. All writes are performed on local copies of data items.

After the transaction has completed its computation phase, it goes through a validation phase to check for *write-write* conflicts with other concurrently committed transactions. A transaction is committed only if none of the items in its write-set have been updated by any concurrent transaction. For each data item, there is a designated *conflict resolver* for checking *write-write* conflicts. It maintains the update timestamp of the latest version of the item, and a *write-write* conflict is detected if the snapshot time of the transaction being validated is less than this timestamp value. The transaction performs validation using a *two-phase* protocol with the conflict resolvers of all items in its write-set. The transaction commits if all conflict re-

solvers give "commit" votes. It is assigned a commit sequence number $seqno$ obtained from the monotonically increasing local counter at its execution site. The commit timestamp for a transaction is a pair <*siteId, seqno*>. The updates are applied to the local database with this commit timestamp as version numbers. The local site's vector clock is advanced appropriately and a commit/abort message is sent to all the conflict resolvers. On commit, the conflict resolver for an item records the update timestamp of its latest version.

The CSI model provides the following guarantees for transaction execution:

- Transaction Ordering: A partial order relationship ($\prec$) is defined over the set of transactions, as described below:
  - *causal ordering*: If transaction T_j reads any of the updates made by transaction T_i, then T_i causally precedes T_j ($T_i \prec T_j$).
  - *per-item global update ordering*: $T_i \prec T_j$ if T_j creates a newer version for any of the items modified by T_i, i.e. T_i commits before T_j.
- Causally Consistent Snapshot: A transaction observes a *consistent snapshot* which satisfies the properties of *atomicity* and *causality*. In a consistent snapshot, either all or none of the updates of a transaction are visible. If a snapshot contains updates of transaction T_i, then updates of all transactions causally preceding T_i are also visible.

In the CSI model, it is possible for concurrent transactions to result in non-serializable execution because it is based on the Snapshot Isolation (SI) model, which can lead to non-serializable transaction executions [4]. An *anti-dependency* [1] between two concurrent transactions T_i and T_j is a *read-write (rw) dependency*, denoted by $T_i \overset{rw}{\rightarrow} T_j$, implying that some item in the read-set of T_i is modified by T_j. This is the only kind of dependency that can arise in the SI model between two concurrent transactions. For the SI model, it is shown in [11] that a non-serializable execution must always involve a dependency cycle among transactions in which there exists one *pivot* transaction with both in-coming and out-going anti-dependency edges. Therefore, one approach for ensuring serializability of SI based transactions is to prevent anti-dependencies from occurring. This approach can be used in the CSI model for ensuring transaction serializability by also checking for *read-write* conflicts, in addition to *write-write* conflicts, during the validation phase. This approach is taken in developing the multilevel consistency model presented below.

V. Multi-Level Consistency Model

The multilevel model [32] supports transactions with different consistency guarantees in a system. The items in the replicated data store are organized along a hierarchy of consistency levels. A higher level in the hierarchy corresponds to a stronger consistency and isolation level. A data item can belong to only one level. Similarly each transaction in the system is designated to execute at exactly one of the levels. Certain rules are enforced for ensuring the consistency guarantees for each level. These rules ensure that information never flows from a weaker consistency level data item to an item at a higher consistency level.

A transaction executes at a specific level in this hierarchy. The following rules are enforced to ensure the consistency properties of the data items organized in different consistency levels by constraining information flow across levels.

- *Read-Up* A transaction at a level in this hierarchy can read only those data items that are at the same level or at stronger consistency levels in this hierarchy.
- *Write-Down* A transaction can update items that are at its own level or at weaker consistency levels.

We refer to the above rules as *read-up/write-down*. These rules prevent a transaction from reading information from weaker consistency level data items to update a stronger consistency level data item. A transaction can update data items that are at its own level or at weaker consistency levels.

To realize the above model, we use the CSI model as a building block to develop a hierarchy of transaction management protocols with different consistency guarantees. All transactions execute with the basic protocol of the CSI model, but with different conflict resolution policies. Therefore, all transactions exhibit the isolation properties of the SI model [4]. A transaction always reads committed data, and the updates of a transaction become visible only when it commits. The atomicity property of causal snapshots guarantees that either all or none of the updates of a transaction are visible in a snapshot. Moreover, all updates are applied at remote sites in their causal order.

In Table 1 we present a hierarchy of data consistency levels and the associated transaction management protocols. This table outlines the consistency properties of the transactions at different levels of this hierarchy. The highest level is the *SR level*, which guarantees serializable transactions. A transaction at the SR level can read items only at the SR level but it can update items at any of the levels. The next level below this is the CSI model which provides the consistency properties described in Section IV. We refer to it as *CSI level*. The transactions are causally ordered with updates to a data item total ordered. The next lower level weakens the CSI model to allow concurrent updates to an item if they are commutative. We refer to this consistency level as *CSI-CM level*. The lowest level in the hierarchy allows asynchronous updates to an item. We refer to it as *ASYNC level*. Conflict checking is not required for updates to data items at this level. This consistency level is suitable for appending records to logs or inserting items in a set.

We briefly outline below the additional mechanisms that are included in the CSI model for implementing the SR and CSI-CM level transactions. The consistency guarantees of the transactions at these levels are also outlined in the following discussion.

A. SR Level – Serializable Transactions

The approach for ensuring serializability of SR level transactions is to also check for *read-write* conflicts among con-

TABLE I
HIERARCHY OF CONSISTENCY LEVELS FOR DATA AND TRANSACTIONS

Level	Consistency Properties	Transaction Model
SR	Strong consistency and isolation - Globally serializable transactions;	Serializable transactions
CSI	Causal ordering of transaction updates; Per-item update ordering; Fork-join model of snapshots for concurrent updates on different items	Causal Snapshot Isolation
CSI-CM	Causal ordering of transaction updates; Permits concurrent commutative updates on an item; Fork-join model of snapshots for all concurrent updates, including commutative concurrent updates on an item	Causal Snapshot Isolation with commutative updates
ASYNC	Causal ordering of transaction updates; E.g. append-only Logs, CRDT [18] objects	Causal Snapshot Isolation without validation

current transactions, in addition to the *write-write* conflict checking performed by the CSI model. For this approach, the validation phase also involves the conflict resolvers for the items in the read-set of the transaction. The conflict resolver for an item at the SR level performs *read-write* conflict checks in addition to checking for *write-write* conflicts. We refer to this type of resolver as *SR resolver*. Each data item at the SR level is associated with an instance of this type of conflict resolver.

A transaction at the SR level is guaranteed to be serializable with other transactions at the SR level, but only with respect to the data items at this level. This property follows from the observation that an SR level transaction can never become a pivot. The basis for this observation is that such a transaction cannot have an outgoing anti-dependency because (1) its read-set can contain only the data items that are at the SR level, and (2) the transaction is committed only if no *read-write* conflicts with other concurrent transactions are found for any of the items in its read-set.

SR level transactions may not be serializable with respect to data items which they may update at the weaker consistency levels. For example if two SR level transactions concurrently append log records to two logfiles at the ASYNC level, their records may appear in different order in the logfiles. The consistency properties of data at the SR level are guaranteed as no information flows from weaker consistency levels to the data items at this level.

B. CSI-CM – Commutative Level

One can exploit operation commutativity to support greater concurrency by reducing the probability of transaction aborts due to *write-write* conflicts. Fundamental concepts in exploiting commutativity of operations for concurrency control are presented in [34]. For supporting concurrent commutative updates, the basic resolver in the CSI model is extended as described below. The validation request to the conflict resolver for an item contains the operation identifier along with the parameters. The conflict resolver checks that all newer versions of the item, not present in the requesting transaction's snapshot, have been created by operations that commute with the operation in the validation request. If so, it gives 'commit' vote for that transaction. Otherwise it aborts the transaction. In case of a 'commit' vote, the resolver keeps track of all commit-pending requests for which a 'commit' vote has been given but the commit/abort decision is not yet known.

For a CSI-CM level object, the resolver is defined based on the notion of *method license* [9] to check if an operation commutes with a group of concurrent operations. The resolver maintains some information about the state of the object and the concurrent commit-pending operations. For example, consider a *Hashtable* object which maintains a set of keys. Its operations *insert(key,value)*, *delete(key)*, and *isMember(key)* all commute with each other if the values of their *key* parameter are distinct. Therefore, a resolver for such an item needs to maintain only the set of keys for which these operations are currently commit-pending. A typical resolver also classifies methods into different commutative groups such that the methods in a group always commute with each other, but methods in different groups do not commute. For example, in case of a *Hashtable* object, the operation *listKeys* to enumerate all keys does not commute with the *insert* and *delete* operations.

In the CSI-CM model it is possible for concurrent transactions modifying the same item with commutative operations to commit. Their update propagation messages contain the operation name and the parameters rather than the updated values. A remote site recomputes the updated value of an item based on this information. The commutative updates of such concurrent transactions may get applied in different orders at different sites. Thus snapshots of different sites can fork even with respect to a single item. Eventually they will converge to the same value when all such concurrent updates have been applied. This is illustrated in more details in [32].

C. ASYNC Level – Asynchronous Updates

This is the weakest consistency level in the hierarchy presented in Table 1. For data items at this level, no conflict checking is performed. This level is useful for data items such as logs or sets, where a transaction appends a record to a log, or inserts an item in a set. The order in which these operations are performed does not matter. For example, consider a log where it does not matter if the records appear in different order at different sites, but the only requirement is that eventually all records are appended to the log. In case of commutative replicated data type (CRDT) [18] objects, all

operations always commute unconditionally, thus no conflict checking is required. Such objects can be placed at the ASYNC level. The causality property in transaction update propagation is still preserved at this consistency level. This consistency level can be used for managing event-stream objects. Transactions defined at any of the four consistency levels can insert notification events in such event-stream objects.

VI. Examples of Multilevel Consistency Model in Groupware Applications

We consider here two examples of groupware applications to illustrate how the multilevel consistency model can be applied for managing replicated data items with transactions providing different consistency levels. The first example is a document editing groupware application. This example shows use of application-level locking primitives in addition to the use of the transaction primitives provided by the multilevel model. In this example long duration transactions are implemented using application-level locking. A long duration transaction contains a series of short transactions, which are implemented using the primitives provided by the multilevel model. This example also illustrates fine-grain locking in a hierarchically structured object. Second example pertains to an engineering design groupware for a group of people working on the design of a product. The group is divided into multiple teams to design different components to be used for building the product. The activities of the group and the collaborating teams involve development of design documents, specification of components and their integration requirements in the final product, group/team organizations, project schedule, and coordination of the activities within and across teams.

A. Document Editing Groupware

A document object is tree structured. It contains a title, list of authors, and a list of *Chapter* objects. Each *Chapter* object has a title and contains a list of *Section* objects. A *Section* object has a title and it contains a list of *Paragraph* objects. A *Paragraph* object contains either a list of lines or simply a blob of characters. As shown in Figure 1, each object of these four types noted above also contains other objects such as an *EditNotes* object which is a set of author-notes maintained in causal order, an *ActivityLog* to record user actions, and an event-stream object called *Notifier* for notifications. Each object contains a reference called *parent* for its container object. The reference is a system-wide unique-id (UID) for the container object. For example, a *Section* object contains a reference to the *Chapter* object in which it is contained.

Each object also contains a *Lock* object for implementing application-defined mechanisms for user activity coordination policies. Long duration interactive transactions are implemented using these lock objects. Such transactions of long duration interactive sessions are application-defined, and they are composed of a sequence of short transactions executed using the mechanisms provided by the multilevel model. The CSI model, which is based on optimistic concurrency control, forms the basic building-block for transaction management in the multilevel model. In this model, if two concurrently executed transactions contain a common data item in their write-set, then the validation procedure permits only one of them to commit and the other is aborted. This model is fine for non-interactive or short duration transactions. For long interactive transactions this can lead to undesirable user experience. Consider a case where two users concurrently perform edits on a document object, and then the edit transaction of one of them is aborted in the validation phase, thus resulting in lost work. Application-defined coordination mechanisms help in avoiding such situations.

In designing the group editing application using the multilevel consistency model, the various data items of a document are placed at different levels in the four-level hierarchy of consistency levels, as shown in Figure 1. This figure also shows various transactions associated with different levels. The data items and transactions are placed at four levels: SR,

Data Items: Document	Data Items: Chapter	Data Items: Section	Data Items: Paragraph	Level	Transactions
Lock: LockManager Title: String Authors: List<String>	Lock: LockManager Title: String parent: Document UID	Lock: LockManager Title: String parent: Chapter UID	Lock: LockManager parent: Section UID	**SR**	*Request/Release Lock* *UpdateTitle* *UpdateAuthors* *Add/Delete Chapter* *Add/Delete Section* *Add/Delete Paragraph*
Content: List<UID> of Chapter objects	Content: List<UID> of Section objects	Content: List<UID> of Paragraph objects		**CSI**	*ReorderChapters* *ReorderSections* *ReorderParagraphs*
EditNotes: Set<String>	EditNotes: Set<String>	EditNotes: Set<String>	Text: List<Line>	**CSI–CM**	*Add/Delete Line* *Edit Line* *AddNotes* *ViewNotes* *ViewDocument*
ActivityLog: Logger Notifier: EventStream	ActivityLog: Logger Notifier: EventStream	ActivityLog: Logger Notifier: EventStream	ActivityLog: Logger Notifier: EventStream	**ASYNC**	*ViewLogs* *NotifyEvents*

Fig. 1. Multilevel Hierarchy of Data and Transactions in a Document Editing Groupware

CSI, CSI-CM, and ASYNC. Transactions access the data items according to the *read-up* and *write-down* rules.

All *Lock* objects are managed with strong consistency, because these objects are critical in proper coordination of the group activities. These objects and the related transactions for acquiring and releasing locks are also placed at the SR level as shown in Figure 1. Similarly, some other data items for which strong consistency is required are also placed at the SR level. They include data items such as the title, author-list, and parent references in chapters, sections, and paragraphs. Corresponding update transactions are also defined at the SR level. For example, operations to add or delete a chapter, or section, or paragraph are defined at the SR level to ensure that the parent pointers are maintained consistently for the document tree structure.

In a *Document* object, the *Content* object is an ordered list of pointers for the chapter objects. The *Content* object is placed at the CSI level. Transactions defined at the SR level to insert/delete a chapter in a *Document* object can update its *Content* object at the CSI level. A transaction is also defined at the CSI level to reorder the chapters in the document. Similarly the *Content* objects of the *Chapter* and *Section* objects are placed at the CSI level. The *EditNotes* objects are of *Set* type, supporting operations to insert notes by the reviewers and the authors. Because these operations are commutative, these objects are placed at CSI-CM level, thus providing concurrent update operations. Because the *Text* object of a paragraph could be designed to support concurrent edit operations using commutativity or operation transformation techniques, it is placed at the CSI-CM level, with the corresponding update transactions also defined at that level. The ASYNC level contains logger and event-notification stream objects, which are updated without any conflict checking. The operations on them include logging of records or insertion of notification events into a stream.

The goal in this system is to allow different users to edit different parts of a document concurrently by appropriate coordination using application-controlled locking operations. For example, two users may be editing in parallel different sections of a chapter, or different paragraphs of a section. They would want to hold an *exclusive* lock on the object while editing it. While a user is editing a paragraph, it is desired to prevent another user from obtaining an *exclusive* lock on the section containing that paragraph. This can be realized by using the *intention mode* locking approach presented in [13]. Following the locking model in [13], to grant an *exclusive* lock on an object, all its parent objects in the hierarchy are locked in the *intent exclusive (IX)* mode to ensure that an *exclusive* lock should not be granted on any of the parent objects in the hierarchy. This can prevent situations such as deletion of a section by one user while another user is editing a paragraph in it. However, an *IX* mode lock on an object permits an *exclusive* or *shared* lock on any of the components contained in the tree structure below it. Similarly, in [13] a *shared* lock on an object is granted after acquiring *intent share (IS)* on all of its parent objects in the tree. The compatibility between various modes is given in [13]. For example, *IS* and *IX* modes are mutually compatible and also with the *shared* mode.

Application-defined locks are *advisory* in nature and they are not enforced by the system. Thus, the programs for the tasks or transactions in an application are expected to acquire locks, in appropriate modes, on the objects they access. An application would define the various lock modes together with their compatibility and how various transactions would use those locking modes. For example, a group editing application may want to allow certain kinds of concurrent editing operations such as reordering of the sections in a chapter while some user is editing a section. Some application may want to support a new mode called *edit shared (ES)* to allow two users to concurrently modify an object because their edit operations either commute or can be combined using methods such as operation transformation. For example, the *Text* object in a *Paragraph* is placed at the CSI-CM level to allow concurrent commutative operations. For such operations, two or more users could be granted *ES* mode lock instead of just one given an *exclusive* mode lock. This mode would still require the parent objects to be locked in the *IX* mode.

The transactions to view all contents of a *Document* object or the comments in the notes collected in the *EditNotes* objects are associated with the CSI-CM level. These transactions can read all objects at this and the levels above it. The ASYNC level contains transactions to read the loggers or the *Notifier* objects to generate notifications for the users. Transactions at any of the four consistency levels can add records to the loggers or events to the *Notifier* event-streams.

B. Engineering Design Groupware

This example considers a groupware application for supporting an engineering project for designing a product. The placement of the various data items and transactions required in this product design project are shown in Figure 2. The group working on the project is divided into teams to work in parallel on different components of the product. Certain objects in this group collaboration environment are shared by all teams, and strong consistency guarantees are required for some of them. For example, the documents for the master design, component specifications, and the requirements for the component integration in the final product are needed to be managed with strong consistency because they are critical for assembling the components to make the final product. Other objects which are important for group coordination include the project schedule, group membership, and team organizations. All these objects and the associated transactions for updating them are placed at the SR level to ensure strong consistency.

The CSI level contains objects corresponding to the activities of different teams. Some of the shared documents within a team include assignment of roles and tasks to team members, task schedule, and the design documents pertaining to various tasks. These are managed using the CSI model which ensures that updates to any of the objects are sequential and they are applied to all replicas in causal order. A transaction updating data at this level can read data from the SR level. For example,

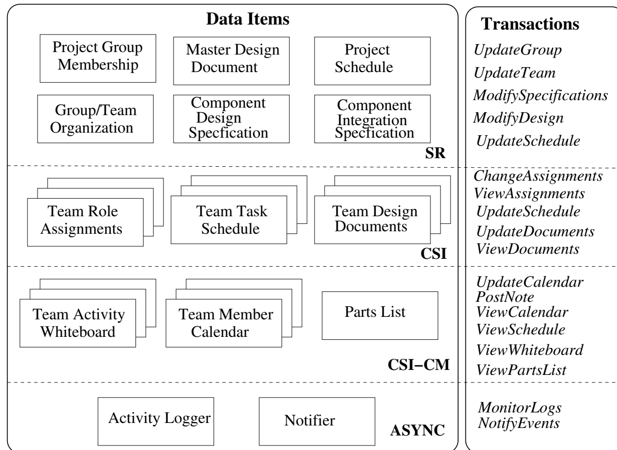

Fig. 2. Multilevel Hierarchy of Data and Transactions in a Product Design Groupware

the transaction updating a team's schedule can read the project schedule and update the team members' calendars at the CSI-CM level. The CSI-CM level contains objects whose conflict resolvers support concurrent updates based on commutativity of operations. Some of these objects include calendars, whiteboards for team communication, and inventory of parts-list for the component being designed by the team. Updates to these objects can be performed by transactions defined at the CSI-CM or higher levels, and they are propagated to their replicas in causal order. At the ASYNC level, logger and event-stream objects are defined for notifications. A transaction can read objects at its own level or at the higher levels. For example, the transaction *ViewSchedule* can read the project schedule, a team's schedule, as well as the team members' calendars.

VII. Conclusion

In this paper we have examined the applicability of a transaction model with multiple consistency levels in groupware applications. This multilevel model was developed for management of replicated data in distributed systems, supporting transactions with different consistency guarantees. The Causal Snapshot Isolation (CSI) model serves as the building-block for this transaction management framework. The multilevel model supports serializable transactions for strong consistency, and weaker consistency models which include CSI, CSI-CM for supporting commutative updates, and ASYNC for asynchronous updates. Data and transactions are organized in a hierarchy which is based on these consistency models. This model ensures the consistency guarantees of data at each level in this hierarchy by constraining the information flow across different levels. In this study we reviewed the unique requirements of concurrency control in groupware systems. These requirements indicate the need of managing different data items with different levels of consistency guarantees. Using two groupware applications we have shown here how these requirements can be supported by the transaction model with multilevel consistency.

Acknowledgments: This work was supported by the National Science Foundation award 1319333. Computing resources for this work were provided by the Minnesota Supercomputing Institute.

References

[1] A. Adya, B. Liskov, and P. E. O'Neil. Generalized isolation level definitions. In *International Conference on Data Engineering ICDE*, pages 67–78, 2000.

[2] B. R. Badrinath and K. Ramamritham. Semantics-based concurrency control: Beyond commutativity. *ACM Trans. Database Systems*, 17(1):163–199, Mar. 1992.

[3] P. Bailis, A. Ghodsi, J. M. Hellerstein, and I. Stoica. Bolt-on causal consistency. In *Proc. of ACM*, SIGMOD '13, pages 761–772, 2013.

[4] 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*, pages 1–10. ACM, 1995.

[5] 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.*, 1:1277–1288, August 2008.

[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.*, 41:205–220, October 2007.

[7] A. K. Dey. Understanding and Using Context. *Journal Of Personal And Ubiquitous Computing*, 5(1):4–7, 2001.

[8] P. Dourish and V. Bellotti. Awareness and coordination in shared workspaces. In *Proceedings of the 1992 ACM Conference on Computer supported Cooperative Work*, CSCW '92, pages 107–114, New York, NY, USA, 1992. ACM.

[9] J. Eberhard and A. Tripathi. Semantics Based Object Caching in Distributed Systems. *IEEE Trans. on Parallel and Distributed Systems*, December 2010.

[10] C. A. Ellis and S. J. Gibbs. Concurrency control in groupware systems. In *Proceedings of the 1989 ACM SIGMOD International Conference on Management of Data*, SIGMOD '89, pages 399–407, New York, NY, USA, 1989. ACM.

[11] A. Fekete, D. Liarokapis, E. O'Neil, P. O'Neil, and D. Shasha. Making snapshot isolation serializable. *ACM Trans. Database Syst.*, 30:492–528, June 2005.

[12] S. Gilbert and N. Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News*, 33(2):51–59, 2002.

[13] J. N. Gray, R. A. Lorie, G. R. Putzolu, and I. L. Traiger. Readings in database systems. chapter Granularity of Locks and Degrees of Consistency in a Shared Data Base, pages 94–121. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1988.

[14] S. Greenberg and D. Marwood. Real time groupware as a distributed system: Concurrency control and its effect on the interface. In *Proceedings of the ACM CSCW Conference on Computer Supported Cooperative Work*, pages 207–217, Chapel Hill, NC, USA, oct 1994.

[15] R. W. Hall, A. Mathur, F. Jahanian, A. Prakash, and C. Rassmussen. Corona: A communication service for scalable, reliable group collaboration systems. In *Proceedings of the 1996 ACM Conference on Computer Supported Cooperative Work*, CSCW '96, pages 140–149, New York, NY, USA, 1996. ACM.

[16] T. Kraska, M. Hentschel, G. Alonso, and D. Kossmann. Consistency rationing in the cloud: Pay only when it matters. *Proc. VLDB Endow.*, pages 253–264, Aug. 2009.

[17] H. T. Kung and J. Robinson. On optimistic methods for concurrency control. *ACM Transactions on Database Systems*, 6(2), June 1981.

[18] M. Letia, N. Preguiça, and M. Shapiro. Consistency without concurrency control in large, dynamic systems. *SIGOPS Oper. Syst. Rev.*, 44(2):29–34, Apr. 2010.

[19] 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*, pages 265–278, 2012.

[20] 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*, pages 401–416, 2011.

[21] J. Munson and P. Dewan. A concurrency control framework for collaborative systems. In *Proceedings of the ACM 1996 Conference on Computer Supported Cooperative Work*, pages 278–287, Boston, Massachusetts, United States, 1996.

[22] T. Nakajima. Commutativity based concurrency control and recovery for multiversion objects. In *Distributed Object Management*, pages 231–247. Morgan Kaufmann, Los Altos, CA, USA, 1994.

[23] V. Padhye and A. Tripathi. Causally Coordinated Snapshot Isolation for Geographically Replicated Data. In *Proc. of IEEE Symposium on Reliable Distributed Systems (SRDS)*, pages 261–266, 2012.

[24] A. Prakash and H. S. Shim. DistView: Support for Building Efficient Collaborative Applications Using Replicated Objects. In *Proceedings of the Conference on Computer Supported Collaborative Work*, pages 153–164, Chapel Hill, NC, Oct. 1994.

[25] R. F. Resende, D. Agrawal, and A. E. Abbadi. Semantic locking in object-oriented database systems. In *OOPSLA'94*, Oct. 1994.

[26] S. Sarin and I. Greif. Computer-based real-time conferencing systems. *Computer*, 18(10):33–45, Oct. 1985.

[27] B. Schilit, N. Adams, and R. Want. Context-Aware Computing Applications. In *IEEE Workshop on Mobile Computing Systems and Applications*, pages 85–90, Santa Cruz, CA, US, 1994.

[28] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski. Conflict-free replicated data types. In *Proceedings of the 13th International Conference on Stabilization, Safety, and Security of Distributed Systems*, SSS'11, pages 386–400, Berlin, Heidelberg, 2011. Springer-Verlag.

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

[30] R. Strom, G. Banavar, K. Miller, M. Ward, and A. Prakash. Concurrency control and view notification algorithms for collaborative replicated objects. *IEEE Trans. Comput.*, 47(4):458–471, Apr. 1998.

[31] C. Sun and C. Ellis. Operational transformation in real-time group editors: Issues, algorithms, and achievements. In *Proc. of 1998 ACM Conference on Computer-Supported Cooperative Work*, pages 59–68, Seattle, USA, Nov. 1998.

[32] A. Tripathi and B. Thirunavukarasu. A Transaction Model for Management of Replicated Data with Multiple Consistency Levels. In *Proc. of IEEE Intl. Conference on Big Data*, 2015.

[33] R. Want and B. Schilit. Expanding the horizons of location-aware computing. *IEEE Computer*, 34(8):31–34, August 2001.

[34] W. E. Weihl. Commutative-based concurrency control for abstract data. *IEEE Transactions on Computers*, 37(12):1488–1505, December 1988.

[35] U. K. Wiil and J. J. Leggett. Concurrency control in collaborative hypertext systems. In *Proceedings of the Fifth ACM Conference on Hypertext*, HYPERTEXT '93, pages 14–24, New York, NY, USA, 1993. ACM.

[36] C. Xie, C. Su, M. Kapritsos, Y. Wang, N. Yaghmazadeh, L. Alvisi, and P. Mahajan. Salt: Combining acid and base in a distributed database. In *Proc. of USENIX OSDI'14*, pages 495–509, 2014.