Real-Time Concurrency Control in Groupware[*]

Paul Jensen   Nandit Soparkar

Electrical Engineering & Computer Science
The University of Michigan
Ann Arbor, MI 48109-2122

{pjensen,soparkar}@eecs.umich.edu

Abstract

Concurrency control has been identified as an important issue for groupware systems in which several users may access shared data resources. The unique real-time responsiveness and consistency requirements in groupware environments suggest that traditional approaches (e.g., from transaction-processing) need to be modified in order to be deployed. We describe how recently developed techniques from real-time transaction systems may be applied to groupware. In doing so, we provide novel ways to partition data and concurrency control to permit weaker consistency constraints in order to facilitate meeting of the responsiveness requirements. Our research addresses these issues as pertinent to both the local as well as the distributed environments in the context of groupware.

1 Introduction

Groupware systems are becoming increasingly important in organizations to facilitate the collaborative efforts of team projects (e.g., see [ea93b, EGR91, KP90, KBL93, GM94, GSW92, Wu95]). Users, often geographically distributed, coordinate on accessing shared data, and some means for *concurrency control* (CC) is necessary to facilitate their inter-dependent, and sometimes conflicting, manipulation of shared data. Such shared data represents not only the particular documents or designs used by the team, but also the presentation environments such as the views, pointers etc., that are made available to the users.

It has been pointed-out by researchers in groupware (e.g., see [EG89, GM94, PK94]) that traditional distributed databases and transaction processing (e.g., see [BHG87]) are not suitable for such collaborative environments. Furthermore, as some of the examples in this paper exhibit, traditional distributed computing techniques (e.g., see [ea90, CS93]) are not entirely appropriate either. The reasons are essentially that the high responsiveness necessary in terms of performance, and human-computer interaction considerations, are paramount — which are not handled well by the traditional techniques. We discuss these issues below, and mention that in our view techniques from several domains, complemented by new ones, will eventually emerge as the appropriate approach for groupware.

The issues to consider for CC in groupware include the specific needs for consistency, real-time responsiveness, undoing of actions, human-interaction etc., in ways that differ from similar issues faced in traditional

[*]This work was supported in part by a grant from the Centre for Advanced Studies, IBM Toronto Laboratory.

concurrent processing. Furthermore, several of the requirements for groupware are mutually incompatible — in particular we consider the *real-time responsiveness* (RTR) and *consistency requirement* (CR) issues as they relate to CC. In broad terms, RTR in groupware refers to the performance of the system being such as to closely approximate a shared workspace for physically co-located users. That is, as far as is feasible, each user's actions should be made visible right away to the others, and vice versa. RTR has two factors in this respect: "response time" (i.e., the difference between the time at which an action is submitted up to the time at which the system responds back having completed the action), and "latency" (i.e., the difference between the time at which an action is submitted up to the time at which its effect is made visible at a remote location) of a user's action. In contrast, the CR issues are often application dependent.

RTR and CR are often not mutually satisfiable in the context of groupware, and this is best illustrated by examples. Consider a shared document being edited by several users simultaneously. RTR suggests that the changes being effected by a user should be made visible locally to the user immediately, and also, propagated to the other users (in consequence incurring unavoidable communication delays). However, if a remote user simultaneously makes conflicting changes (local to the remote location), then a CR problem may arise in terms of the contents of the document. Conversely, if an attempt were made to ensure a consistent document across all locations (e.g., by the use of "write locks"), then the CC itself may cause inadequate RTR (e.g., due to the time-consuming acquisition of remote locks). This example illustrates how RTR may adversely affect CR considerations, and vice versa (discussed further in Section 3). This fundamental incompatibility between the two requirements has been also identified in environments such as transaction-processing and real-time systems (e.g., see [Sop93, SLKS94]).

On the other hand, the shared data resources in a groupware system may exhibit characteristics which permit greater RTR because their CR considerations are less crucial (e.g., as compared to a database environment). Consider data that represents the "presentation" services in groupware (e.g., a shared pointer, or a shared view of the underlying document or design). The utility of such data lies in providing "instantaneous" interaction among the users (limited, of course, by unavoidable communication delays). However, it may not matter much if, occasionally, a shared pointer or a shared view gets out of synchronization for some users. In such cases, if an optimistic CC approach is used to improve RTR, the occasional inconsistencies introduced due to conflicting changes may be rectified by restoring the shared resources to a consistent state chosen in some arbitrary manner, or by human intervention.

In this paper, we examine how techniques developed recently for real-time CC may be effectively applied to groupware systems. Our aim is not so much to provide new CC protocols as to show how some existing ones may be used. We delineate the differences in CC as dictated by RTR considerations at local as well as the distributed sites. Furthermore, we discuss a logical design to segregate the data and CC into several distinct levels such that the RTR and CR within a level may be addressed in a similar manner, whereas these may be distinct from the other levels. By exploiting the characteristics required for the groupware, we are able to suggest several CC strategies suitable for use in groupware.

2 Logical System Model

We address the design of the facilitating functions in groupware as a database problem. Therefore, we use standard database terminology to describe the data and the applications.

2.1 Using a Standard Model

The *database* is a set of data *entities*, and it is *persistent* in that the data may have lifetimes greater than the processes that access them. Associated with each entity is a *name*, and a *value*; the *state* of a database is a mapping of the entity names to their corresponding values.

The access of database entities is effected by an *operation* which is an *atomic* access of the database; an operation is one among *read*, *write*, *delete*, and *create* of the database entities (often abbreviated to $R[\ldots]$, $W[\ldots]$, etc.) as well as the special operations for *commit* and *abort* with their usual semantics. A *transaction*, T_i, is a data accessing part of an application program, and may be regarded as a sequence on a finite set of operations that is guaranteed to execute correctly in isolation. Exactly one of the abort or the commit operation is part of a transaction T_i, and all the other operations in T_i precede it. A transaction is the unit for the consistent access of the database as well as the unit for recovery in case of an abort. The precise meaning of a *consistent* access is left unspecified except that it is to accurately reflect the groupware application semantics. Therefore, a transaction preserves the consistency of a database when executed in isolation.

Example 1. Instances of "transactions" occur in groupware whenever a set of operations, possibly with some partial order imposed on them, must be executed atomically on shared data. Note that in the following examples, human protocols may be used to coordinate the activities — although the provision of the requisite protocols by the system would minimize the need for such protocols, and would improve the RTR.

In collaborative editing of a text document, a sequence of changes made to a portion of the document by one user would need to be protected from other users' actions in order to reflect correctly the desired changes. This may be accomplished by treating the changes made by a user as one transaction. As another example, consider a situation where one user creates a picture, and a different user moves the picture to a specific location on the screen. These two activities would need to be atomic w.r.t. each other to ensure that an incomplete version of the picture is not moved. In our approach, each of the two activities may be treated as a separate transaction acting on the shared workspace of the picture.

There may also be situations where a single multi-site transaction is defined for grouping together the actions by several users. For instance, suppose that a telescope is to be focused by several geographically dispersed users who are responsible for different controls. One user may be responsible for the horizontal movement of the device, the second may handle the vertical movement, and a third may actually adjust the focal length to maintain a sharp image. Assume that there are many other entities adjusting the telescope for various purposes. In such a situation, to keep the view observed through the device focused, the actions of the three users may be grouped together as a single transaction. Note that in this example, there may be additional timing considerations that need to be observed among the users' actions to maintain focus. 2

A *schedule* of transactions, is a sequence on the set of operations from the transactions which subsumes each sequence of a transaction T_i that is represented in it. We use the criterion of *conflict serializability* (CSR) as the logical correctness criterion for a schedule with concurrently executing transactions; more liberal criteria (e.g., see [BHG87, KS94]) may also be considered without affecting the development of our ideas significantly. We develop the use of more liberal criteria in the descriptions within this paper. Furthermore, operations that do not need transaction protection may be submitted as such — the CC would be regarded then to be transactions with single operations (and locks etc. would be unnecessary for them).

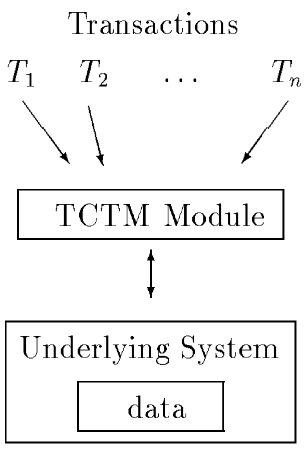

Figure 1: Centralized CC for transactions.

For a given site we assume the software architecture illustrated in Figure 1 to manage CC (e.g., see [BHG87]) and scheduling. Transactions are submitted to the CC module which resides above the underlying system that stores the data. The CC module must handle both, the logical correctness as well as the RTR for the executions.

In the distributed environment, the CC for groupware has features common with transaction processing at different locations that have autonomous, centralized CC (illustrated in Figure 3 of Section 5). This is dictated by the RTR requirements: managing CC for all transactions from a single site would be too slow. Transactions that access data at a particular site include *local* transactions as well as sub-transactions from *global* transactions. We assume that the local clocks at the separate sites are well-synchronized, and that individual CC modules manage local scheduling.

One of the unique features of groupware is the desirability of allowing intermediate changes generated during atomic actions being made visible to other users. For instance, even though the drawing of a figure by one user may be regarded as a single transaction, the process by which the figure is generated may need to be displayed to all the users. Clearly, such uncommitted changes being made visible raises some issues and we briefly discuss them. In contrast, transaction systems do not permit uncommitted changes to be made visible to other transactions (rather than human users), and we continue to support that approach for reasons such as recoverability etc. of the generated schedules. However, we do assume that human users will not take any actions based on uncommitted values made visible to them — the analogous situation in transaction systems is to prevent access of uncommitted, and therefore possibly inconsistent, data.

The above observations regarding visibility imply that uncommitted changes observable by human users must be demarcated as being subject to confirmation by commitment. Therefore, temporary changes may be portrayed using different graphics (e.g., not unlike the manner in which a shadow window is created in window-based systems when the size or location of a window is being changed). We do not discuss such user-interface issues any further in this paper.

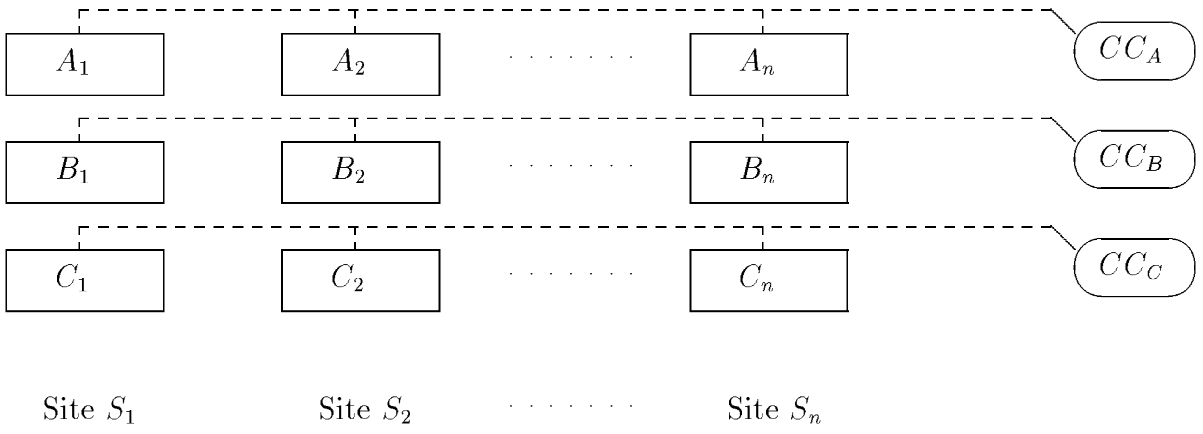

Figure 2: Logical system structure with levels.

2.2 Logical Separation of Data

Figure 2 depicts a logical separation of data and CC into "levels" in a groupware environment. Sites $S_1, S_2, \ldots, S_n$ represent the separate locations where there are users (one user per location). At any given site S_i, the levels A_i, B_i, and C_i, represent three disjoint, "loosely coupled," data sets, with different RTR and CR requirements. The data within a given level across the sites (e.g., $A_1, A_2, \ldots, A_n$) would usually represent replicated data. Replication would be necessary in groupware to manage situations where large latencies may be encountered if the requisite objects need to be read remotely. Such data may have various consistency constraints between them (e.g., in terms of replication, the constraint is "equality").
The levels A, B, and C correspond to the following RTR and CR requirements:

Level A: This level corresponds to high RTR and low CR. In particular, in order to provide quick responsiveness to an action initiated by a user, it is acceptable for the local user as well as other users to occasionally see the action being effected, and then being undone (possibly by human intervention) due to conflicts with other actions, or to encounter occasional inconsistencies.

Level B**:** This level corresponds to high RTR and medium CR. Again, it is acceptable for the user to occasionally see an action being done, and then being undone due to conflicts with other concurrent actions. However, unlike the case for level A, the data would need to be restored to a consistent state by the CC. As expected, this may adversely affect the RTR requirement.

Level C**:** This level corresponds to high CR, even if RTR is relatively poor. The effect of an action will be made permanent only if the action is guaranteed to be committed.

Different concurrency control strategies, CC_A, CC_B, and CC_C, may be used across the sites to manage the data levels A, B, and C, respectively. The reason that separate CC is possible is that the levels may be regarded as separate, independent databases. Such approaches have been considered in other transaction processing contexts with respect to distributed databases as well (e.g., see [KKB88, SLJ88]). In all cases, the CC strategies should ensure that eventually the data at various sites become mutually consistent.

3 Responsiveness and Consistency

In general, providing better RTR performance entails using relaxed CR — since "optimistic" approaches need to be used (e.g., see [KR81]). However, in most cases, the effectiveness of various forms of optimism depend on factors such as the degree of conflicts etc., and therefore, it is not very simple to provide a general approach suited to all situations. Below, mainly by means of examples, we demonstrate the various RTR and CR considerations for the logical levels described in Section 2.

We expect that level A data will represent the presentation data, such as shared pointers and views, with high RTR requirements, but relatively low CR. In groupware, these constitute very important considerations for the usability of a system.

Example 2. Consider an application where several scientists are studying a geophysical terrain together, and the image is provided on their computers(e.g., see [ea93a]). A single pointer on the screen may be used to draw the attention of the entire team to particular points of interest. Since the team may be involved in discussions on what they observe, the pointer manipulations would need to be effected in real-time, and in order. Furthermore, conflicts arising due to the simultaneous access of the pointer may be resolved in a simple manner (e.g., by restoring the pointer to its original position, by giving one movement preference over others, or by combining the movements by using T_{op}-style transformations as described in [EG89]). 2

Example 3. Consider a flight-simulation environment where different controls are handled by separate user (e.g., there may be separate controls for motion in the vertical and horizontal directions). The resulting view should be depicted on every user's computer screen in real-time during the simulation. However, the effect as seen on the screens needs to be the *net* effect of combining both directions of motion (as opposed to an interleaved sequence of vertical and horizontal motions). Furthermore, small inconsistencies between the views observed by two users may be inconsequential, as long as views of all users become consistent in an eventual quiescent state. 2

Level B is expected to represent shared data in a groupware environment that exhibit both RTR and CR to a certain degree — and since both cannot be provided to their highest degrees simultaneously, there is a need here for innovative, groupware-specific approaches to CC. The approaches to CC may try to make allowances in RTR and CR, based on application semantics, in order to approach the goal of providing both criteria simultaneously. As one may expect, the CC for this level is likely to be very challenging.

Example 4. Consider again the viewing of geophysical terrain data as described in Example 2 above. Suppose that it is deemed unreasonable to make changes to the views being seen by other users, until the point when the changes made are committed by the CC. Then a given user would need to obtain permission (via the CC) for making changes in the shared view prior to effecting the changes. Of course, once the permission is granted by the others (e.g., if locks are obtained on others' views), then the RTR considerations may be brought into play. 2

Example 5. Consider a groupware environment that is used in teaching a geographically distributed class, and suppose that a shared blackboard is depicted on the computer screens. It may be the case that the instructor requires, from time-to-time, exclusive access to the board to ensure that the entire class sees his writing, and also, no unexpected conflicts occur for the shared board. In addition, the instructor may want RTR for the changes made at the local view of the board. In such situations, the exclusive access would be obtained first (via the CC), and subsequently, RTR may be provided. 2

Attempts to satisfy RTR and CR requirements simultaneously may entail undoing (by utilizing application semantics) a user's actions that were effected at some sites (e.g., see [PK94]). Since several other actions may have occurred that were dependent on the changes made by the action to be undone, an undo may prove difficult. Examples for such situations are available in the literature, and the approaches that appear suitable involve some form of compensatory actions derived from the applications (e.g., see [PK94, SLKS94, Lev91]). In terms of a transaction-oriented approach, this could even mean the undoing of committed actions based on "compensating" transactions (e.g., see [Lev91]).

Example 6. Consider an observation system in scientific domains for mobile objects (adapted from [SLKS94]). The system consists of several tracking stations (i.e., "sites"), each of which has its own computing and scheduling resources. That is, there are several processing sites that manage object-sensors, cameras, and store data pertaining to the readings, positions, etc. locally.

Periodically, the sensors update the data regarding the objects as tracked at each local site, and this data is also sent to specific coordinator sites. The coordinator site receives track data from several sites and correlates the information gathered to create the global tracking information. It is necessary to do the correlation since the data obtained at each site may be individually insufficient to identify the objects accurately. The globally correlated data is also disseminated among the sites, and this data affects local decisions at the sites. Finally, global decisions may be taken sporadically for a variety of actions to be executed simultaneously among several sites. For instance, cameras may be activated to take photographs of a particular object from several angles at a particular time, to be then provided to the groupware users.

We assume that at each site, local transactions update the local track data (e.g., see *external-input* transactions of [KSS90]). Also, we assume that the collection and correlation of the local track data from the different sites, and the dissemination of the global track data, together constitute one type of global transaction. The reading of the local track data and subsequent writing of the global track data at each site constitute the local subtransaction for the global transaction.

Suppose that an erroneous local track is recorded at one of the locations — perhaps due to a malfunctioning sensor. This fault may be detected only after the local track data is collected and correlated with (correct) track data from other sites (but before the corresponding global track is committed). Consequently, erroneous global track data may be generated and disseminated to several sites. Such a global transaction should be aborted as soon as possible. In standard transaction processing, the execution of a commit protocol (e.g., 2PC) ensures that all the subtransactions of the aborted global transaction do indeed abort.

The price paid for employing a standard commit protocol may be high. Blocking may cause a situation where none of the sites have recent global track data, and awaiting the coordinator's final decision may unnecessarily cause poor RTR. The fast *local commit* of a subtransaction would be much more suitable — optimistically assuming that the global transactions usually commit. However, uncoordinated local commitment may cause some sites to commit the erroneous global track data they receive, and subsequently to expose the data to other transactions. For instance, a transaction that positions the camera at a site may base its computation on the prematurely committed, and hence inaccurate, global track data. Therefore, there is a need to recover from the effects of the incorrectly committed data by compensatory actions. In our example, the compensatory actions re-position the camera based on the past history of the execution. 2

The level C data may be regarded as representing shared data with a very stringent CR, even if the RTR is relatively poor. It may be necessary to ensure that changes made by a user to such data can be guaranteed to be effected (provided there are no unprecedented system failures). Therefore, potentially conflicting accesses would need to be carefully identified, and avoided or resolved, before allowing a user to embark on the changes proposed. Or, at least, standard transaction commitment semantics would need to be provided (i.e., committed transactions execute correctly, and have their changes effected permanently). The role played by standard, conservative CC techniques (e.g., from database management systems) should be evident in these cases.

Example 7. Consider the cooperative editing of a document where each user has the need to make some intricate and time-consuming changes to parts of the document. It may be the case that a user would then be unwilling to repeat the work in case conflicting changes invoked by some other user prevents the intended changes from being effected. In such environments, it may be worthwhile for the CC to provide a guarantee that any changes reflected on the local computer screen would indeed be effected (and not "aborted" due to others' actions). This implies that the required exclusive access to the relevant part of the document must be ensured before allowing the changes made by a user to be initiated. 2

4 Centralized Scheduling

In this section, we consider the groupware issues of scheduling a user's or application program's actions to provide the RTR and CR necessary. We may use "soft" deadlines to incorporate response time needs for users (i.e., a deadline would represent the acceptable time period within which the transaction in question may commit without causing any problem to the user).

Example 8. Consider two transactions, T_1 and T_2, with associated deadlines as follows.

- T_1: $R_1[y]R_1[x]W_1[x]W_1[y]$; deadline: 600 ms.
- T_2: $R_2[x]W_2[x]$; deadline: 400 ms.

Assume that the time taken for each operation is 100 ms. The following schedules start at time 0.

- **Schedule** a**:** CSR and only 1 deadline is met.

$$\begin{array}{lllll} R_1[y] & R_1[x] & W_1[x] & W_1[y] & \\ & & & R_2[x] & W_2[x] \end{array}$$

- **Schedule** b**:** Serial schedule, and both deadlines are met.

$$\begin{array}{llllll} & & R_1[y] & R_1[x] & W_1[x] & W_1[y] \\ R_2[x] & W_2[x] & & & & \end{array}$$

- **Schedule** c**:** Logically incorrect, and both deadlines are met.

$$\begin{array}{llll} R_1[y] & R_1[x] & W_1[x] & W_1[y] \\ R_2[x] & & & W_2[x] \end{array}$$

Several points should be noted. First, for the application, schedule b is preferred over schedule a even though schedule b takes longer to execute. Second, based on our correctness criteria, schedule c is deemed unacceptable. Although with other correctness criteria, it may still be considered acceptable. Third, since we are dealing with soft deadlines, the aim is to try and meet these time constraints, without necessarily being able to guarantee them. 2

4.1 Traditional Concurrency Control

We re-examine traditional transaction CC since the limitations in merely improving concurrency become apparent (e.g., see [Sop93]), and the options available to the CC module get clarified. The traditional approach has the goal of certifying as large a number of schedules as possible to be logically correct (e.g., see [Pap86]). The expectation is that this would enable the CC module to adversely affect the performance minimally. Therefore, every logically correct schedule is regarded as being equally desirable (e.g., either of the two logically correct schedules, a and b in Example 8). That is, the goal of the CC is to preserve consistency (i.e., logically correct executions) while maintaining a high level of parallelism (i.e., the number of allowable executions).

The CC may be regarded as a function F that permutes an input sequence with n operations, s_i, into an output sequence, s_o, that belongs to an acceptable logically correct class of schedules. There is an implicit assumption that the CC must not delay any operation unless it is necessary to do so: any input sequence belonging to the logically correct class of schedules handled, is required to be output unchanged. That is, CC F is said to generate a correctness class C only if for any $s_i \in C$, it is the case that $s_o = s_i$. We refer to such CC that does not delay any operations unnecessarily for a class C as a *no-delay scheduler* for the class C. The no-delay characterization has been used to assess the utility of different concurrency control strategies in terms of the concurrency level supported by them. Potentially, the characterization includes all common CC protocols such as *two-phase locking* (2PL), *time-stamp ordering* (TO) etc.; in the case of a *certification* protocol, the only difference is that the check to ensure that an acceptable execution is being generated is done at the end of a transaction rather than at each operation (e.g., see [BHG87]). This characterization includes CC developed using semantics-based approaches.

No-delay schedulers have several inadequacies when considered in the context of groupware:

- In practice, delays may be encountered for operations that arrive in an input sequence that is logically correct even in CC techniques potentially characterized as being no-delay schedulers. For example, in 2PL, an operation O_i must wait if it arrives while a conflicting operation O_j from another transaction is holding a lock. In such cases, the CC delays the operation O_i, and outputs some other operation O_k, thereby effecting a change in the input sequence.

- The performance impact of aborted transactions is not handled by the no-delay scheduler characterization since the commit and abort operations are not considered. As a result, a scheduler with a high degree of concurrency is regarded as being desirable even if it encounters a high rate of aborts (e.g., in situations with a high degree of data contention).

- No-delay schedulers do not attempt to provide better performance explicitly. That is, it is assumed implicitly that the input sequence will be one that provides good performance, and the sequence is output unchanged. This is a consequence of regarding different logically correct schedules as being equal in terms of performance. In fact, allowing a larger number of schedules implies that schedules exhibiting *worse* performance may also be generated.

4.2 Optimizing for Performance

To improve performance a CC module for groupware must *choose* among the logically correct schedules — i.e., there is a need for optimization. Increasing the degree of concurrency provides more choices for optimization. However, such increases in concurrency do not help in making the search problem (which is usually computationally expensive) any easier. To effect the optimization, a CC module may influence either the ordering of the operations in the generated schedule, or it may abort certain transactions — or do both. To change the ordering of the operations, a CC module cannot handle operations one at a time (i.e., in a manner similar to a no-delay scheduler); instead, it must handle a set of several operations together. If such a set is not used or is unavailable, this approach reduces to no-delay scheduling. Since RTR considerations are important, the CC should be invoked sufficiently often (i.e., every few milliseconds) to ensure that the response time requirements are not adversely affected. Each such invocation of the scheduler may then execute on the outstanding operations regarded as being a set. As example 3 in Section 3 suggests, in some situations there is an inherent need to schedule a set of operations. To simplify the scheduling, the operations are assumed to take a fixed amount of time to execute (e.g., if the data is main-memory resident, this is a reasonable assumption), and that they have deadlines derived from those on the transactions.

We now focus on the optimization strategies for the CC module, and we examine the options available. Meeting the logical correctness and performance criteria "simultaneously" is not simple. Logical correctness criteria, such as CSR, effectively impose a precedence order on the operations to be scheduled, and typically, this precedence order is generated dynamically based on the executed portion of a schedule.

We identify three different approaches that may be adopted by a centralized CC module — in each, the limitations with regard to improving concurrency are evident.

- **Ensuring logical correctness first.** The set of operations can be precedence ordered to ensure logical correctness by means described in other research (e.g., see [Kri82]). Thereafter, a separate scheduler that handles precedence ordering may be used to achieve the desired performance. The approach would be to regard each operation as a task for scheduling purposes.

- **Addressing performance first.** A CC module may optimize performance criteria prior to imposing logical correctness. In practice, this approach corresponds to techniques such as prioritizing the input queue to the CC (e.g., see [Ram93]). A simple example would be to effect the "earliest deadline first" heuristic on deadline-constrained operations by maintaining an input queue sorted in the order of the deadlines.

  The optimistic assumption being made in addressing the performance first is that imposing the logical correctness criteria would not significantly affect the order indicated by the performance criteria. That is, the logical correctness phase would be best achieved by a no-delay scheduler which attempts to minimize the changes made to a given input sequence of operations.

- **Handling both criteria simultaneously.** The scheduling problem in this context is unresolved w.r.t. complexity. The interested reader is referred to [Sop93].

Regardless of which of the above approaches is used, the scheduling would be essentially a static scheduler executed repeatedly to achieve dynamic scheduling. The question arises as to how often the static scheduler would need to be executed. A simple approach would be to execute the static scheduler every few units of time or every few new submitted operations – whichever occurs first. The specific number of time units or operations would need to be empirically determined from the groupware application at hand.

In terms of obtaining locks etc. for a lock based scheme, local locks should be obtained first, followed by an optimistic execution at the local site (made visible locally). Concurrently, remote locks and executions may be initiated. Thereafter, if a sufficient number of sites respond, the local change, and remote changes thereafter, may be committed. It is not difficult to envisage several variations of such approaches.

Since RTR requirements are high for level A, we describe some of the specifics that need to be met by CC_A. CC_A should be such that changes invoked by a user are optimistically reflected immediately at the local site, and the changes are propagated to the remote sites using messages that are intended for rapid delivery (if the communication system has different speeds for delivering messages depending on their real-time requirements). Furthermore, we require that any conflicts that occur at a site w.r.t. the changes invoked are resolved by mechanisms that operate locally at each site (e.g., see [SLKS94, Lev91]). This is to ensure that time-consuming negotiations with other sites (thereby exacerbating problems by degrading RTR) are avoided. Example 2 in Section 3 provided an instance of such a strategy in the context of conflicting accesses to a shared pointer.

5 Distributed Concurrency Control

A Multi-site Groupware System (MGS) consists of n sites, S_1, S_2, ..., S_n, interconnected by a computer network as shown in Figure 3. Each site S_i has a local groupware system, GPM_i, with a local CC module, LCC_i. The data, which may be replicated, is stored in LDB_i. The groupware is distributed among the sites in the form of n software agents. Each agent, MGS_i, located at site S_i is interconnected with the other MGS_j modules by a communications network, but is otherwise independent of any other MGS_j module. All interaction between the sites, including the synchronization between the sites is managed by the MGS_i. All transactions and subtransactions at site S_i are executed by LCC_i, which does not distinguish between transactions and subtransactions — hence, we refer to either by the common term "transaction." A subtransaction for a global transaction G_x that executes at site S_j is denoted by T_{xj}. We assume that each LCC_i generates CSR schedules.

The presence of a distributed environment creates several problems in realizing a groupware system. Not only are latencies introduced in the interactions, but also, coordinating among the different sites is very challenging. The reason is that with autonomous CC at the constituent sites, there may be incompatible, conflicting executions in the system. The issue is primarily one of maintaining the CR across the sites. Therefore, in this section, we place more emphasis on the levels C and B of the software architecture.

For the relatively straightforward case of level C, the CC_C mechanisms need to be very conservative in their approach to CC. Techniques which first ensure that the intended access by a user will indeed be achieved (e.g., by acquiring exclusive "write locks" at every site) fall into this category. In fact, these techniques may be made even more conservative as compared to a replicated distributed database system. For instance, database systems in which only a single copy of a data item is present may exhibit characteristics of CC that are similar to those needed in CC_C.

For level B, it is clear that fairly innovative techniques are needed for CC_B to try and satisfy the RTR and CR considerations simultaneously. Approaches based on undo/redo, compensation, semantics etc. (e.g.,

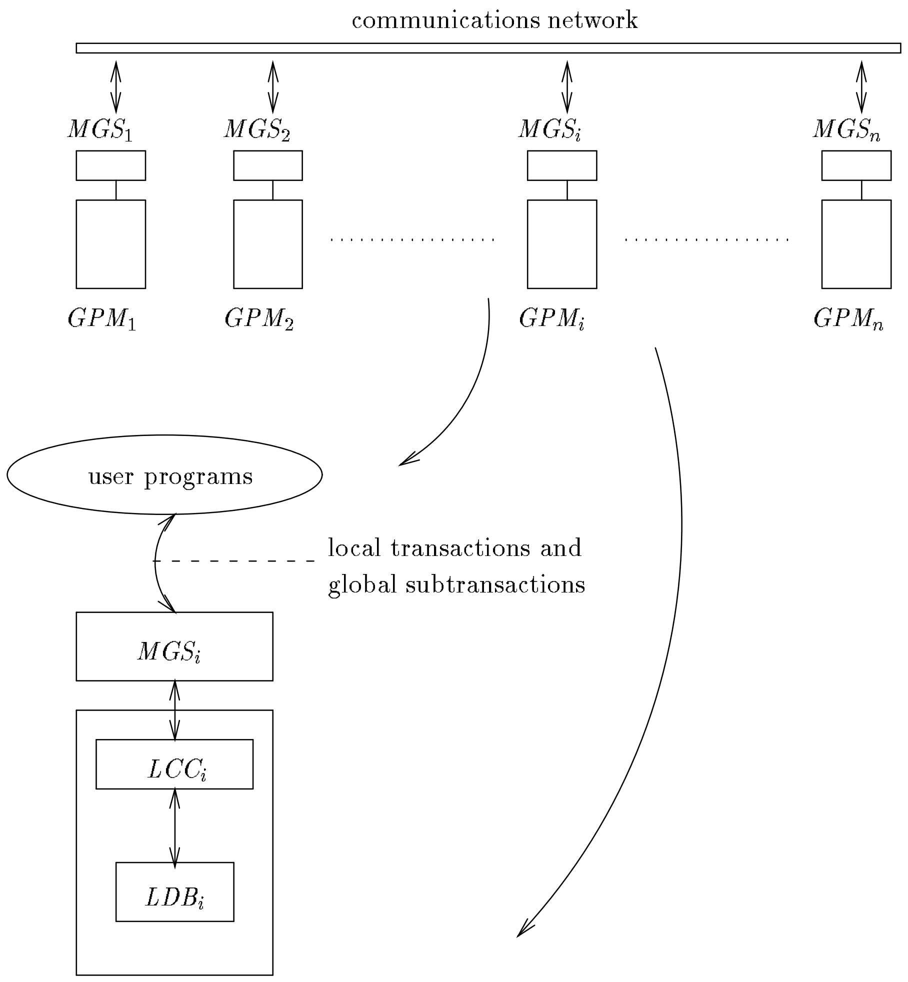

Figure 3: MGS structure

see [PK94]), may all be regarded as attempts to achieve these essentially incompatible goals. The use of semantics to improve the performance of concurrent systems, especially in the context of new and advanced applications, has been the subject of study in related disciplines (e.g., see [Sop93, SLKS94, Lev91]), and below, one of the techniques is detailed in the context of groupware systems.

5.1 Difficulties in Distributed Synchronization

Consider the issue of ensuring the correct execution of a multi-site global transaction. The standard approach to distributed atomic commitment is to use the *two-phase commit* (2PC) protocol (e.g., see [BHG87]), where a transaction is executed under the control of a centralized coordinator. If for any reason a site does not obtain the final message of the 2PC protocol, then the data held at that site for the concerned transaction may be *blocked* (i.e., become inaccessible until the necessary message is received). This would lead to very poor RTR. A somewhat different problem that may arise is that the local CC may dictate that a subtransaction be aborted in favor of other (sub)transactions with higher response time priorities, and that may be impossible to achieve for similar reasons.

While it is advisable to use the standard CR where possible in order to accrue their obvious advantages [GR93, BHG87], the problems noted above cannot be tolerated in groupware applications. Therefore, in order to use the transaction paradigm in a groupware environment, some of the stringent CR must be *relaxed* (e.g., see [Sin88, Sta88]). We adapt the concept of relaxed atomicity from [SLKS94] and apply them to groupware CC. The key idea in our approach lies in the idea of *compensation.* If a transaction T commits "erroneously" (i.e., it is discovered *ex post facto* that T should actually have been aborted), a compensating transaction CT for T is used to perform a "semantic undo" of T (e.g., see [KLS90]). This undo returns the database to a consistent state that is equivalent, in an application-specific semantic sense, to a state resulting from an execution in which T never executed. Compensation is accomplished without resorting to cascaded aborts.

Compensatory actions may be deferred as compared to traditional undo operations which need to be performed immediately. This allows the execution of the recovery process during periods of light system load. Moreover, it is not necessary for a transaction T to hold data pending the execution of CT; the data may be later (re-)acquired by CT. We use standard 2PC as the norm for transaction commitment, with compensation-based techniques invoked only when required; this reduces the overhead associated with commitment and uses standard CR normal operating conditions.

5.2 Compensating Transactions

A *compensating* transaction is a recovery transaction that is associated with a specific *forward* transaction that is committed, and whose effects must be undone. The purpose of compensation is to "undo" a forward transaction *semantically* without causing cascading aborts. Compensation guarantees that a consistent state is established based on semantic information. The state of the database after compensation takes place may only approximate the state that would have been reached, had the forward transaction never been executed (e.g., see [KLS90, Lev91]).

Regard a transaction to be a function from one database state to another. An execution imposes a serialization order among a set of concurrently executing transactions. Therefore, an execution defines both a total order among the transactions, as well as a function from states to states that is the functional composition (denoted by '$\circ$') of the transactions. That is, $X = T_1 \circ \ldots \circ T_n$ denotes the function from states to states defined by applying the functions denoted by the transactions in the same order X. We use $X(St)$ to denote the state resulting from applying the function X to the state St.

To formalize the criteria of compensation, we use a binary relation between the transactions as follows. Two transactions, T_1 and T_2, commute with respect to a relation $\mathcal{R}$ on states (in short, $\mathcal{R}$-commute), if for all states St, $(T_1 \circ T_2)(St)\ \mathcal{R}\ (T_2 \circ T_1)(St)$. If $\mathcal{R}$ is the equality relation, the two transactions commute in the usual mathematical sense. For a forward transaction T and its compensating transaction CT, the execution $T \circ X \circ CT$ is atomic with respect to $\mathcal{R}$ (in short $\mathcal{R}$-atomic), if $T \circ X \circ CT(St)\ \mathcal{R}\ X(St)$. If a compensating transaction is serialized immediately after its associated forward transaction, then compensation amounts to the traditional undoing of the forward transaction; formally, $(T \circ CT)(St) = St$. If CT $\mathcal{R}$-commutes with each of the transactions mentioned in X, then the execution $T \circ X \circ CT$ is $\mathcal{R}$-atomic. The relation $\mathcal{R}$ serves to constrain CT, thereby preventing it from violating CR.

In groupware, each global transaction is decomposed into a collection of local subtransactions, each of which performs a semantically coherent task at a single site. The subtransactions are selected from a well-defined library of routines at each site. For global transactions that can be compensated-for, each forward subtransaction is associated with a predefined compensating subtransaction. Compensating for a global transaction need not be coordinated as a global activity (e.g., see [LKS91a, Lev91, LKS91b]). Consequently, the compensating subtransactions are assumed to have no inter-dependencies, share no global information, and to not need the use of a commit protocol (i.e., local sites run the compensations autonomously).

5.3 Relaxing the CR

Our CR is stated in terms of a *serialization graph* (SG) that are an extended version of the standard SGs (e.g., see [BHG87]). We model an aborted subtransaction as a committed subtransaction followed immediately by the corresponding compensating transaction that simply undoes the committed subtransaction (i.e., simply as a syntactic device for uniformity of presentation).

Let $\mathcal{T}$ be a set of global transactions. Let C_p be a site with the set of subtransactions, $\mathcal{T}_p$ corresponding to $\mathcal{T}$, and the set $\mathcal{CT}_p$ corresponding to the set of compensating subtransactions. Aside from the compensating subtransactions used to model an aborted subtransaction, not all elements of $\mathcal{T}_p$ need necessarily have a corresponding element in $\mathcal{CT}_p$. The *local serialization graph* at site C_p for a complete local history H_p (e.g., see [BHG87]) is a directed graph $SG_p(H)=(V_p, E_p)$. The set of nodes V_p consists of a subset of transactions in $\mathcal{T}_p \cup \mathcal{CT}_p$. An edge $A \rightarrow B$ is in E_p if and only if one of A's operations precedes and conflicts with one of B's operations in H_p.

A *global SG* is the union of all the local serialization graphs. For a set of local SGs, represented by $SG_p = (V_p, E_p)$, the corresponding global SG is defined as $SG_{global} = (\cup\ V_p,\ \cup\ E_p)$. Observe that each compensating subtransaction is assigned a separate node in the global SG (in accord with the local execution of compensating subtransactions as described above).

A global history is logically correct if the following two conditions hold: (1) serializability (i.e., the global SG is acyclic); and (2) semantic atomicity (i.e., for each transaction T_i, either all local subtransactions are committed — thereby committing T_i, or for each committed subtransaction of T_i, the corresponding compensating transaction is executed at some point following the commitment of the subtransaction in question — thereby aborting T_i).

5.4 Using Compensation

We illustrate compensation and $\mathcal{R}$-atomicity by referring back to Example 6. We shall be concerned here only with the setting of the camera. Consider the following transactions:

T_1 : The global transaction that collects and correlates local track data, and disseminates the resultant global track data to the various sites. Suppose that the erroneous reading occurs in subtransaction T_{11} at site S_1. Thus, subtransaction T_{12} at site S_2 contains the recording of the erroneous correlated data into the local database.

T_{22}**:** A local camera positioning transaction at site S_2.

CT_{12}**:** The subtransaction for T_1 at site S_2 that performs compensatory actions at that site.

We now consider the compensating subtransactions and the concept of $\mathcal{R}$-commutativity in this context at site S_2. We shall use the following data variable:

GT_2**:** A sequence of data elements at site S_2, each of which is global track data. This sequence records all global track data in chronological order with the most current one forming the head of the sequence. Each element in the sequence is associated with a time-stamp to specify how current that information is. Such a sequence is assumed to be stored at each site — possibly in the form of log records.

Manipulation and access of GT_2 is done through the following:

- *tail*(*sequence*): Returns the tail of the sequence (i.e., all elements but the head).
- *head*(*sequence*): Returns the head element of the sequence.
- *extrapolate*(*sequence*): Computes and returns global track data which is the extrapolation of the sequence of global track data provided to it.

Next, we provide the pseudo-code for the compensating subtransaction CT_{12}:

```
$GT_2 \leftarrow tail(GT_2)$
if $GT_2$ was read by $T_2$ since it was updated by $T_{12}$ then
    begin
        $x \leftarrow extrapolate(GT_2)$
        set the camera according to the global track data $x$
    end
```

The first step of CT_{12} is simply to undo T_{12} by removing the head of GT_2. Observe that once this head element is removed, the time-stamp of the new head indicates that the value is outdated. Only if there are transactions that used the (erroneous) value of GT_2, is compensation actually needed. Checking this condition (i.e., "if GT_2 was read since...") can be done as part of the execution of CT_{12} by accessing the log records that contain GT_2. Actual compensation is performed by the routine that sets the camera based on the extrapolated value stored in the variable x. The aim is to try to set the camera based on its past trajectory since the local site does not know the precisely correct current position for it.

The pseudo-code for T_{22} is as follows:

```
$y \leftarrow head(GT_2)$
**if** the time-stamp of $y$ shows the value is up-to-date **then**
        set the camera according to the global track data $y$
```

Let X be the local execution before T_{12} at site S_2. Consider the following equations:

$$St_1 = (X \circ T_{12} \circ T_{22} \circ CT_{12})(St_0) \qquad (1)$$

$$St_2 = (X \circ T_{12} \circ CT_{12} \circ T_{22})(St_0) = (X \circ T_{22})(S_0) \qquad (2)$$

Equation 1 represents an execution where T_{12} was committed erroneously, and was later compensated-for. Equation 2 represents an execution where T_{12} was aborted on time and its effects were entirely undone. In the execution represented by Equation 2, T_{22} is unable to position the camera since it follows CT_{12}, which rendered outdated the new head value of GT_2. Observe that had T_{12} actually aborted, T_{22} would have been prevented from positioning the camera in exactly the same manner. We do not speculate on what T_{22} does in such a situation for this example.

Finally, we define an appropriate relation $\mathcal{R}$ as follows:

$St_1 \ \mathcal{R} \ St_2 \ \equiv$ the last value in the sequence that is used as input to compute the value of x by extrapolation (in St_1) = the value of y (in St_2)

Observe that $St_1 \not\equiv St_2$ which is a consequence of guaranteeing only semantic atomicity rather than traditional atomicity. Also, notice that the above description holds regardless of whether T_{12} is committed locally, or it is committed following the completion of a 2PC protocol. Thus, the availability of a compensating transaction allows for the flexibility of making local decisions.

5.5 Adaptive Atomicity

For the baseline operation of the system, the 2PC protocol can be used to ensure the traditional criteria of correctness. Its use in conjunction with strict 2PL effectively synchronizes all the subtransactions according to a distributed 2PL policy (e.g., see [BHG87, SKS91]).

The above protocol suffers from the following shortcomings. First, once a participant indicates its preparedness to commit in 2PC, it cannot allow the subtransaction in question to relinquish the locks held by committing or aborting until such time that the final decision is obtained from the coordinator. This is problematic since it may cause other local subtransactions awaiting the decision to miss their deadlines. Second, if the final commit message arrives after the deadline for the local subtransaction has passed, the commitment must still be effected to achieve traditional atomicity. Also, note that since an indication of preparedness requires that the participant be in a position to change the database as dictated by the subtransaction in question, it is necessary to save the appropriate log records prior to a notification of preparedness. This may be a time-consuming activity, and thus, there is an additional delay before the participant may notify its preparedness.

To alleviate the above problems, a protocol based on an optimistic 2PC protocol (e.g., see [LKS91a]) may be used adaptively under conditions requiring improved RTR. This protocol is similar to the traditional 2PC up to the point that the request for the state of preparedness is received by the participating sites. At this point, the synchronization to ensure CSR is achieved (e.g., in a distributed 2PL protocol, the coordinator knows that all locks, for all subtransactions, have been obtained), and the phase that follows ensures semantic atomicity. Due to the flexibility offered by semantic atomicity, no message need be sent to the coordinator unless the subtransaction is aborted. In the event that the subtransaction is aborted, the coordinator is alerted by a message, and this causes the coordinator to trigger compensations at all sites that committed their subtransactions in order to maintain semantic atomicity. We assume in this discussion that the requirements of $\mathcal{R}$-commutativity are met, as detailed above.

Notice that in the above protocol, locks may be released at any time after the first phase by simply aborting the subtransaction — which is not possible after the point that preparedness is guaranteed in the traditional 2PC protocol. Therefore, the problem of blocking due to remote failures or delays is avoided. Furthermore, if a subtransaction attempting to commit fails to do so — say due to the lack of resources — then the site could choose to abort it, and to subsequently inform the coordinator. Note that the log records can be saved at any time prior to the final commit for the subtransaction, and therefore, the delay does not severely affect the commitment.

We now describe the adaptive strategy that assures semantic atomicity as a contingency measure. The idea is that blocking becomes imminent, sites should decide locally to switch from the traditional 2PC to the optimistic 2PC. This decision may be taken at any time after the first phase of the 2PC protocol. The decision as to when exactly that should take place is application-dependent. Note that an abort can always be effected unilaterally during the first phase of the 2PC protocol. In the event that the global coordinator decides to abort the entire transaction, compensating subtransactions may be executed at each site where the subtransactions in question were locally committed. This ensures semantic atomicity as described above.

Our adaptive strategy provides a method to deal with the question of a fast approaching deadline due to RTR for a subtransaction T_{ip} (corresponding to a global transaction T_i) executing at site S_p. In the above protocols, if site S_p has not yet sent a message indicating preparedness to commit to the coordinator of T_i, then T_{ip} may be unilaterally aborted. On the other hand, if that message has already been sent, then T_{ip} may be optimistically committed — the expectation is that the final decision regarding the fate of a global transaction will usually be to commit it. Notice that it would be inadvisable to abort T_{ip} prior to receiving a final decision to abort from the coordinator of T_i since such an action would be contrary to the intent of the message sent by S_p that indicated a state of preparedness to commit T_{ip}.

Some important points regarding the above protocols need to be stated. Although the above scheme uses strict 2PL to guarantee serializability, similar mechanisms could be devised for other CC techniques by using careful synchronization (e.g., see [SKS91]). Also, we note that in the techniques described above, concurrent global transactions may each use a different criterion of atomicity. Furthermore, even for the same global transaction, the constituent subtransactions may actually be engaged in different commit protocols. This is important because not all subtransactions may be compensatable. Such subtransactions must always follow the traditional 2PC protocol, whereas its sibling subtransactions may continue to use an optimistic 2PC approach. Thus, the non-compensatable subtransactions are informed of a final decision to commit only after the coordinator ascertains that all the compensatable subtransactions also commit. This is achieved by using a complete version of the optimistic 2PC protocol [LKS91a] where each participant informs the coordinator after it commits its corresponding subtransaction.

6 Conclusions

We have considered the concurrency control issues for groupware systems, and have exhibited that recent developments in real-time transaction processing may be appropriate approaches. We have provided a novel architectural framework for the design of concurrency control mechanisms for use in groupware systems. Our approach logically separates the data and concurrency control into levels based on the different real-time responsiveness and consistency requirements within the same groupware application. In our approach, we indicate how several existing traditional and newly developed techniques may be used to provide the characteristics desired in groupware systems. Experiences of others using several experimental systems (e.g., see [ea93a, Wu95]) suggest that our approach may be profitably used in managing concurrent executions in groupware environments.

There are several issues regarding concurrency control for groupware, and we consider only a few of them here. First, although we have discussed the different levels of CC, decisions must be made on where the separations are effected and how each CC level will be implemented (i.e., locking or other schemes), and how these will impact the user interface. For instance, in each CC scheme, it is necessary to consider how to achieve the visibility appropriate for the CC level (i.e., when the changes are to be made visible to users). Second, there are considerations for the CC schemes that are common to transaction systems. For example, the granularity and types of locks, the use of logging facilities, the specific communication protocols etc. are all relevant issues. We hope that system builders will gain insight into these issues for groupware as prototype systems are built and deployed.

Finally, we note that a transaction-oriented concurrency control approach is not necessarily the panacea for managing concurrent executions in groupware systems. It may be the case that hybrid systems that use both transaction-oriented as well as process-group approaches will prove most appropriate in groupware systems. We postulate that there is a need for an advanced transaction model that is specifically geared to groupware systems.

References

[BHG87] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems.* Addison-Wesley, Reading, MA, 1987.

[CS93] D.R. Cheriton and D. Skeen. Understanding the limitations of causally and totally ordered communication. *Operating System Review*, pages 44–57, December 1993.

[ea90] K. Birman et al. The isis system manual, version 2.0, 1990.

[ea93a] C.R. Clauer et al. Uarc: A prototype upper atmospheric research collaboratory. *EOS Trans. American Geophysical Union*, pages 267–274, 1993.

[ea93b] Ed. Reddy et al. Computer support for concurrent engineering. *IEEE Computer*, 26(1), January 1993.

[EG89] C.A. Ellis and S.J. Gibbs. Concurrency control in groupware systems. In *Proceedings of ACM-SIGMOD 1989 International Conference on Management of Data, Portland, Oregon*, pages 399–407, June 1989.

[EGR91] C.A. Ellis, S.J. Gibbs, and G. Rein. Groupware: Some issues and experiences. *Communications of the ACM*, 34(1), January 1991.

[GM94] S. Greenberg and D. Marwood. Real time groupware as a distributed system: Concurrency control and its effect on the interface. *CSCW '94*, pages 207–217, October 1994.

[GR93] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques.* Morgan Kaufmann, San Mateo, California, 1993.

[GSW92] I. Greif, R. Seliger, and W. Weihl. A case study of CES: A distributed collaborative editing system implemented in Argus, 1992.

[KBL93] Alain Karsenty and Michel Beaudouin-Lafon. An algorithm for distributed groupware applications, 1993.

[KKB88] H. F. Korth, W. Kim, and F. Bancilhon. On long duration CAD transactions. *Information Sciences*, 46:73–107, October 1988.

[KLS90] H. F. Korth, E. Levy, and A. Silberschatz. Compensating transactions: A new recovery paradigm. In *Proceedings of the Sixteenth International Conference on Very Large Databases, Brisbane*, pages 95–106, August 1990.

[KP90] M.J. Knister and A. Prakash. DistEdit: A distributed toolkit for supporting multiple group editors. In *Third Conference on Computer-Supported Cooperative Work*, pages 343–355, October 1990.

[KR81] H. Kung and J. Robinson. On optimistic methods for concurrency control. *ACM Transactions on Database Systems*, June 1981.

[Kri82] R. Krishnamurthy. Concurrency control and transaction processing in a parallel database machine environment. Ph.D. dissertation. Department of Computer Sciences, University of Texas at Austin, December 1982.

[KS94] H. F. Korth and G. Speegle. Formal aspects of concurrency control in long-duration transaction systems using the NT/PV model. *ACM Transactions on Database Systems*, 1994.

[KSS90] H. F. Korth, N. R. Soparkar, and A. Silberschatz. Triggered real-time databases with consistency constraints. In *Proceedings of the Sixteenth International Conference on Very Large Databases, Brisbane*, August 1990. Also included in Readings in *Advances in Real-Time Systems*, IEEE Comp.Soc. Press, 1993.

[Lev91] E. Levy. Semantics-based recovery in transaction management systems. Ph.D. dissertation. Department of Computer Sciences, University of Texas at Austin, July 1991.

[LKS91a] E. Levy, H. F. Korth, and A. Silberschatz. An optimistic commit protocol for distributed transaction management. In *Proceedings of ACM-SIGMOD 1991 International Conference on Management of Data, Denver, Colorado*, pages 88–97, May 1991.

[LKS91b] E. Levy, H. F. Korth, and A. Silberschatz. A theory of relaxed atomicity. In *Proceedings of the ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, August 1991.

[Pap86] C. Papadimitriou. *The Theory of Database Concurrency Control.* Computer Science Press, Rockville, Maryland, 1986.

[PK94] A. Prakash and M.J. Knister. A framework for undoing actions in collaborative systems. *Transactions on Computer Human Interactions*, 1994. Also available as Tech.Rep. CSE-TR-196-94 from The University of Michigan, Ann Arbor, EECS department.

[Ram93] K. Ramamritham. Real-time databases. *International Journal on Parallel and Distributed Databases*, 1(2), 1993.

[Sin88] M. Singhal. Issues and approaches to design of real-time database systems. *ACM SIGMOD Record*, 17(1):19–33, March 1988.

[SKS91] N. R. Soparkar, H.F. Korth, and A. Silberschatz. Failure-resilient transaction management in multi-databases. *IEEE Computer*, 24(12):28–36, December 1991.

[SLJ88] L. Sha, J.P. Lehoczky, and E.D. Jensen. Modular concurrency control and failure recovery. *IEEE Transactions on Computers*, 37(2):146–159, February 1988.

[SLKS94] N. R. Soparkar, E. Levy, H. F. Korth, and A. Silberschatz. Adaptive commitment for real-time distributed transactions. In *Third International Conference on Information and Knowledge Management*, December 1994.

[Sop93] N. R. Soparkar. Time-constrained transaction management. Ph.D. dissertation. Department of Computer Sciences, University of Texas at Austin, December 1993.

[Sta88] J. A. Stankovic. Misconceptions about real-time computing. *IEEE Computer*, pages 10–19, October 1988.

[Wu95] Gwobaw A. Wu. Concurrency control issues in collaborative systems (a thesis proposal), June 1995.