System Support for Robust, Collaborative Applications

Muthusamy Chelliah *Mustaque Ahamad*

College of Computing

Georgia Institute of Technology

Atlanta, Georgia 30332-0280 USA

Phone: (404) 894-2593

(chelliah,mustaq@cc.gatech.edu)

GIT-CC-95-09

Abstract

Traditional transaction models ensure robustness for distributed applications through the properties of view and failure atomicity. It has generally been felt that such atomicity properties are restrictive for a wide range of application domains; this is particularly true for robust, collaborative applications because such applications have concurrent components that are inherently long-lived and that cooperate. Recent advances in extended transaction models can be exploited to structure long-lived and cooperative computations. Applications can use a combination of such models to achieve the desired degree of robustness; hence, we develop a system which can support a number of flexible transaction models, with correctness semantics that extend or relax serializability. We analyze two concrete CSCW applications - collaborative editor and meeting scheduler. We show how a combination of two extended transaction models, that promote split and cooperating actions, facilitates robust implementations of these collaborative applications. Thus, we conclude that a system that implements multiple transaction models provides flexible support for building robust, collaborative applications.

Key Words: Atomicity, Groupware, Extended Transaction Models, Programming Support, Operating Systems.

1 Introduction

Distributed computing systems are increasingly being used to support interactions between users that go beyond file sharing and electronic mail, e.g., collaborative editors allow users distributed across many nodes to concurrently work on shared documents. Outline generators (e.g., Cognoter [12]), coauthoring tools (e.g., Quilt [11]), and meeting schedulers (e.g., Visual Calendar [2]) are concrete instances of applications that constitute the broad domain of Computer-Supported Cooperative Work (CSCW). Collaborative applications have several unique characteristics: users in these applications operate on persistent data for long durations, and there is *cooperation* due to interleaved resource sharing among different users. An editing session may last hours and users may take turns updating parts of a shared document.

Persistent data manipulated by distributed applications necessitates support for *robustness* to maintain consistency despite partial failures and concurrency inherent in a distributed system. Several distributed systems provide transactional facilities for building robust applications. Flat [10] and nested [24] atomic transactions ensure *failure atomicity* (either all or none of a transaction's changes to data items persist despite failures), and *view atomicity* (concurrent access to common data items by different transactions are made to appear serial). The atomicity properties are necessary for traditional databases, however not well-suited for robust, collaborative applications. In fact, it was demonstrated that the Argus system [23], which supported nested transactions, could not effectively be used to program a robust version of a CSCW tool - the Collaborative Editing System (CES) [17].

In this paper, we explore the restrictions associated with atomic transactions not only for collaborative editing, but also for other CSCW applications. In particular, view atomicity does not permit interleaved resource sharing among cooperating components of CSCW applications, and reduces potential concurrency since locks will be retained by atomic computations over extended durations. Failure atomicity, also provided by atomic transactions, prohibits early committal of changes even to select data items thereby making the changes vulnerable to failures in long-running computations. We address these limitations of atomic transactions by exploiting the recent advances in extended transaction models [9]. Specifically we show that two such models, that promote *split* [26] and *cooperating* [13] transactions, can be used to program robust CSCW applications. Our approach thus advocates the use of multiple transaction models for programming robust applications.

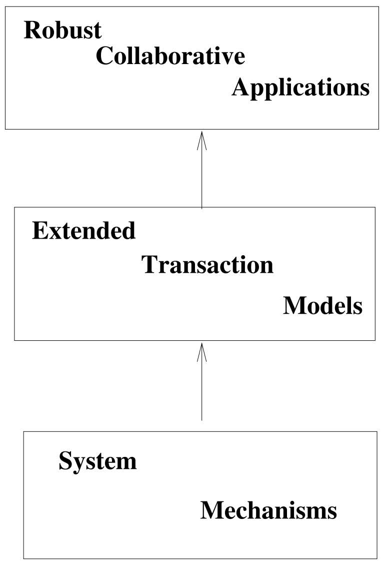
Robust Collaborative Applications
Extended Transaction Models
System Mechanisms

Figure 1: System Architecture

As shown in Figure 1, we propose a layered system architecture - operating system (OS) mechanisms for concurrency and recovery control that can be used to realize a variety of transaction models, a library that implements multiple transaction models using the low-level mechanisms, and robust collaborative applications realized using the transaction models provided by the library. Such an approach is easy-to-use since the only support facility the CSCW application programmer needs to exploit is the set of well-defined transaction models. Our approach is efficient also since we exploit the low-level OS primitives for accommodating such a flexible, multi-model programming paradigm, still simple since we tailor mechanisms that are readily available in modern systems to implement multiple transaction models. We will elaborate on these virtues of our system support later in the paper.

Section 2 explores in detail two applications that come from the domain of *groupware* (tools for CSCW) - collaborative editor and meeting scheduler. We analyze the robustness needs of these applications as dictated by their concurrency and recovery requirements. We then describe how existing techniques for robustness do not meet the requirements in Section 3. We present brief descriptions of some extended transaction models and show how each of them meets some specific needs of CSCW applications in Section 4. Section 5 shows outlines of robust editor and scheduler applications implemented using a combination of split and cooperating transactions. We discuss how these multiple transaction models can be implemented using a uniform set of system mechanisms in Section 6. The contributions of the paper are summarized in Section 7.

2 Robust Collaborative Applications

Collaborative applications contain a shared workspace that is manipulated by concurrent users in a relatively uninhibited fashion. The users might have varying degrees of collaboration depending on whether they are competing or cooperating on a certain task. The concurrency and recovery control techniques employed by the system for robustness should neither force the competing users to wait for extended durations nor prohibit the cooperating users from interleaving their data accesses. In the following description of the sample applications, we identify situations which require such support for robustness, and thereby demand more flexibility than that provided by traditional atomic transactions.

2.1 Collaborative Editor

A shared document is the workspace manipulated in collaborative editing. The document consists of disjoint sections and different users can operate concurrently on these sections. The users can be colleagues who, at the outset, negotiate their roles in writing different sections of a research paper. Coauthors write complete sections of the paper whereas critical reviewers participate in the writing process only by commenting on the written sections, after substantial work has been done.

An author who originally accepted the responsibility for writing certain sections of the paper might want to periodically save his changes without relinquishing control, so as not to lose them in the event of failures. After a while, on consultation with his reviewer, the author might delegate some of the sections to the reviewer for further polishing while continuing to work on the remaining sections. This dynamic division of labor is desirable because the reviewer does not need to wait for the author to complete all the sections, especially if both agree in the interim that the entire writing process might take a long time. Even before the author completes a certain section, he can allow his coauthor to either read the section for a quick reference, or write parts of it.

In a collaborative editing session, one can model a single user writing all the relevant sections of a research paper as an atomic transaction. However, the task of writing the paper might last for a few weeks which could make the completed sections vulnerable to failures and prevent the reviewers from accessing those sections for a long time. Also, interleaved access to different sections by cooperating authors is not feasible. Contrarily, if editing each section itself is treated as an atomic transaction a priori, atomicity of accesses to multiple sections by a single author cannot be ensured.

2.2 Meeting Scheduler

A meeting scheduler is a program which coordinates collaborative activities like arranging a group conference in an office environment. A shared calendar is the workspace manipulated in this application. The calendar consists of a set of diary objects, each with disjoint fragments representing different appointment slots. Each individual group member as well as inanimate entities, like a meeting room or an overhead projector, maintain a diary object. The scheduling task is performed by a coordinator (secretary) and consists of activities like polling the group members, arriving at a consensus date (time) and reserving the necessary facilities. This task naturally involves many rounds of negotiation, which may last even a few days, before a consensus is achieved.

After each round of negotiation, the coordinator needs to release the unwanted slots for other meetings, and start another round of negotiation with the current compromise slots as the input. Once a consensus is reached among the group members, possibly after many rounds of negotiation, the coordinator can proceed to reserve the room and overhead projector. In the meantime, it is essential to guard the group's consensus slots from failures and competing coordinators who may try to grab these slots. Also, a negotiating coordinator can allow a colleague to cooperate either by concurrently reserving a room, or rejecting a meeting based on the unavailability of the room.

If the scheduling task with the many negotiation phases is modelled as an atomic transaction, locks held on the unwanted slots at the end of each negotiation phase might prevent other coordinators from progressing till all the phases complete. Furthermore, intermediate states like a consensus achieved among the meeting participants without involving the equipment (e.g., overhead projector) cannot be reliably saved. Also, cooperating coordinators cannot freely access shareable resources like the meeting room. Each negotiation phase (releasing the unacceptable and compromise slots) can be modelled as an atomic transaction; however, this option does not guard against concurrent access of compromise slots by competing coordinators between two phases.

2.3 Requirements of Robust Collaboration

From the preceding discussion, we conclude that robustness guarantees, primarily facilitated by atomic transactions of course, are desirable in collaborative applications; however, the following extensions and relaxation, of the serializability correctness criterion [10], can be beneficial:

1. enhancing concurrency through early committal of select data items and releasing the associated locks,
2. reducing loss of data due to failures through periodic checkpoints of a subset of accessed data items,
3. providing flexible functionality whereby interleaved resource sharing (cooperation) is feasible among robust computations.

3 Related Work

We now investigate how well existing techniques for maintaining consistency in distributed systems satisfy the requirements of robust collaboration; the goal of this discussion is to highlight the lack of support for robust, long-lived and cooperative computations in any one distributed system.

Low-level mechanisms used directly for flexible concurrency control in collaborative applications, like *triggers*, *reservations* and *tickle locks*, have been surveyed by Greif and Sarin [16]. Many systems that facilitate non-serializable executions, e.g., Locus [32], Profemo [22], Nexus [30] and Arjuna [29], provide mechanisms that relax the atomic transaction model in certain ways. However, none of these systems support a variety of user-level facilities that help realize robust, collaborative applications easily and flexibly. It is possible to design ad hoc techniques at the user level for relaxing view as well as failure atomicity. However, the burden of validating the resulting executions then lies on the application programmers. We next explore several systematic approaches with well-defined, but flexible, consistency semantics that could provide a unifying paradigm for robust collaboration.

DistEdit [19] - a group editor - uses the Isis toolkit [5] which supports process groups and associated broadcast protocols that update the state encapsulated by processes corresponding to different users. Failure notification and message ordering guarantees provided by the Isis system can ensure consistency of information shared among the users. Process groups facilitate forward progress by preserving consistency of replicated data items; however, without additional mechanisms, they cannot efficiently maintain the integrity of a collection of updates to persistent data items as a whole in the presence of failures. A team of coauthors can encapsulate their computations in one process group so as to view an update by a single author in a consistent manner across the group whereas each coauthor can structure a set of updates he/she performs more naturally as an atomic transaction. This is mainly because of the orthogonality of the underlying correctness criteria - virtual synchrony vs. serializability and failure atomicity [4].

Our basic premise is that atomic transactions, a powerful abstraction that ensures consistency of a sequence of accesses to persistent data items, simplifies the structuring of robust computations. In addition, techniques that enhance the atomic transaction model can provide the flexible consistency semantics that is needed by robust CSCW applications. Simple extensions of the atomic transaction model like nesting and type-specific concurrency control do not address all the problems associated with the basic model. We first discuss these techniques and then outline our approach for robust collaboration.

Nesting of atomic transactions [24] localizes failures by making the ancestor transactions responsible for the permanence of data items manipulated by their descendants. This technique protects a transaction from its children's failure, but a child transaction's changes are vulnerable to the parent's failure. Also, a parent transaction might force its siblings to wait longer for accessing its children's resources. Moreover, the hierarchical structuring does not promote cooperation among the atomic subactions. Atomic transactions alternatively can exploit opearation semantics associated with the data items [28, 31, 1] for enhanced concurrency and hence improved performance. All transactions invoking the particular object methods can cooperate using this technique. It is however not suitable for applications where such object methods cannot be statically determined and dynamic sharing of resources among only a few transactions, representing select users, is desired.

We narrow down our search, for techniques that support robust collaboration, to extended transaction models [9], because such models extend and/or relax the serializability correctness criterion selectively at run-time. The models were developed by database researchers to address the needs of advanced engineering applications such as CAD and software development. Composite models that combine the desirable features from many simpler models are now being proposed, e.g., Cooperative Transaction Hierarchies [25], NT/PV model [21], and Flexible Transaction Model [18]. Generalized correctness semantics advocated by such models could very well accommodate the requirements of robust collaboration. However we prefer a set of simple models because they can be implemented efficiently using existing system mechanisms and an application can choose a combination of such models to avoid the specific limitations associated with atomic transactions.

We have found that a combination of split [26] and cooperating [13] transaction models readily match the requirements of robust collaborative applications that were identified earlier. We exploit these models to structure long-lived and cooperative computations common in such applications. We already have a proof-of-concept prototype system for implementing multiple transaction models on the same software platform [6]. Low-level mechanisms that support many transaction models have been investigated in other systems like ASSET [3] and TSME [15]; however, they focus on advanced database applications and not on distributed applications from the CSCW domain.

In summary, we employ a combination of extended transaction models to address the limitations of flat and nested atomic transactions. Such an approach more naturally fits the paradigm in which users frequently interact among themselves by manipulating shared, persistent data. A computation in our system is spawned originally as an atomic transaction; however it remains flexible since it can dynamically change its structure using one or more extended transaction models. We refer to such active entities in a distributed application as robust computations since they - despite failures - preserve the well-defined consistency semantics dictated by the various transaction models.

4 Extended Transaction Models Match CSCW Applications' Needs

We now focus on two extended transaction models, and the abstract application characteristics they are suitable for. We describe in a generic fashion how the robust collaboration requirements are met through extensions and relaxation to the serializability correctness criterion, allowed by split [26] and cooperating [13] transactions respectively, without violating application-specific data consistency. In the next section, we illustrate these desirable features by showing how concrete examples from groupware exploit the multiple transaction models.

Dirty reads and lost updates are two common problems cited to motivate the atomic transaction model which prohibits a transaction from reading its peer's uncommitted results as well as writing onto them. However, such unconstrained sharing is desirable in CSCW applications to enhance concurrency and promote cooperation. In that case, a dependency set can be formed for each concurrent transaction to regulate the consequent side-effects; a *dependent* transaction T_2 is one that can read or write some data items manipulated by the *depended-upon* transaction T_1. If T_2 is alive after T_1 aborts, atomicity of T_1 could be violated since T_2 could propagate the uncommitted results of failed T_1. In traditional database applications (e.g., banking and air-line reservation) where atomicity is necessary, such violation could lead to cascading aborts of dependent transactions like T_2 when the transaction T_1 abnormally terminates.

Cascading aborts, on the otherhand, could be costly in the CSCW domain due to its inherently long-lived and cooperative computations. Many extended transaction models obviate cascading aborts in a systematic manner; the models clearly identify the application characteristics that make the transactions independent of one another despite sharing uncommitted results. We will now explore, in particular, how split and cooperating transactions alleviate the restrictions associated with atomic transactions for robust collaboration as well as avoid the overhead of keeping track of and aborting dependent transactions.

4.1 Enhancing Concurrency - Independent Split Transactions

A robust computation, structured originally as an atomic transaction T_1, can dynamically split itself into many independent components, say T_{11} and T_{12}, divide its resources (data items as well as associated locks) into disjoint partitions, and make one atomic transaction component responsible for each resource partition. Due to the lack of a priori knowledge of the resource partitions and transaction components, the option of using separate atomic transaction for each resource partition initially itself is not possible. The split transaction T_{11} can be committed or aborted independent of T_{12} thereby making its resources available for other transactions without making such transactions wait for T_{12} to complete.

Thus, the independent split transaction T_{11} enhances concurrency through early committal of a subset of data items originally owned by T_1 and releasing the associated locks. This is not possible if T_1 remains atomic without any restructuring. The resources that T_{11} and T_{12} access are disjoint, hence failure of T_{11} does not affect T_{12} and vice-versa. Since these newly created transactions still remain atomic, there is no danger of other transactions sharing their uncommitted results. However, the set of eventually committed transactions (T_{11} and T_{12}) need not be the same as that existed initially (T_1) since the creation of transactions is dynamic through the splitting of original atomic transactions. Early committal of a few of these newly created transactions enhances concurrency.

4.2 Reducing Data Loss - Serial Split Transactions

The transaction components created dynamically by splitting the original atomic transaction can become related if the resources they access are not disjoint. They are defined as serial split transactions and can be used to periodically save some or all of the data items accessed by an atomic transaction. The atomic transaction T_1 can split itself into two transaction components T_{11} and T_{12}, delegate all or some of its resources to both the components, make one component T_{12} dependent on the other component T_{11}, commit the effects of the depended-upon component T_{11}, and allow the dependent component T_{12} to proceed concurrently.

Failure of the dependent split transaction T_{12} does not affect the changes made by the depended-upon transaction T_{11}, already committed as a checkpoint. If T_1 had remained atomic as structured originally, on its failure even the changes committed by T_{11} in the new structure could have been lost. Thus, the serial split of T_1 into T_{11} and T_{12} reduces data loss through a checkpoint by T_{11}. Since T_{11} is committed immediately after the split, there is also no danger of aborting its dependent transaction T_{12} later. The application semantics might require that T_{12}, till its completion, does not release the locks that it inherited from T_{11} so as not to affect the correctness of other concurrent transactions. If T_{12} should fail, however, a concurrent transaction T_2 could obtain its locks and thereby even undo the changes made by its depended-upon transaction T_{11}.

4.3 Accommodating Interleaved Resource Sharing - Cooperating Transactions

An application could have a group of cooperating computations whose results are made visible to the external world only after the whole group completes its assigned task. Each such computation can be encapsulated in a transaction and thus the application is programmed as a collection of transactions. In the CSCW domain, transactions that belong to a certain group of users are treated as peers and need to cooperate among themselves. In particular, a computation encapsulated by a transaction T_1 may need to selectively allow its peers (say T_2) to access (read or write) its resources even before T_1, or one or more of its split components T_{11}, T_{12}, etc. complete.

If the cooperating computation T_1 fails, its results may have to be undone; however, the application-specific knowledge about cooperation might avoid cascading aborts across the groups, or even among group members when the transactions interact only in a certain manner. The need for this execution behaviour, that allows violations to concurrency atomicity but not failure atomicity, was first identified by Garcia Molina who introduced cooperating transactions [13]. The following restrictions are enforced on such seemingly non-recoverable schedules - that permit accesses to uncommitted data and still avoid cascading aborts [20]:

1. only those reads which do not affect any further writes are allowed,
2. only writes into disjoint fragments of data items which do not change the previous uncommitted writes are permitted.

Thus cooperating transactions permit interleaved resource sharing between computations like those encapsulated by T_1 and T_2. This flexible behaviour could not have been possible if T_1 and T_2 had remained atomic. If a cooperating transaction T_2 belonging to a group member reads another's (T_1) results, but does not use the results for further modifications, none of the cooperating computations need to be aborted. If uncommitted results do lead to further modifications, compensating transactions [14] could avoid cascading aborts. Similarly, the two group members T_1 and T_2 might exploit the information that they will be concurrently modifying disjoint fragments of a shared workspace, or if the fragments overlap they will adopt a merging policy on completion.

4.4 Systematic Approach for Flexible Robustness

Extended transaction models thus add flexibility to the traditional notion of atomicity, potentially without sacrificing consistency in certain application domains. The scenarios under which they can be used are well-defined, and hence application programmers can use the models without worrying about the undesirable side-effects of failures and concurrency. Each model alleviates a certain limitation associated with atomic transactions and hence we rely upon a combination of these transaction models for supporting robust collaboration.

5 Programming Support for Robust Groupware

We now show how a combination of extended transaction models simplify the task of programming CSCW applications. We present code fragments from robust collaborative editor and scheduler applications discussed earlier. A variant of C++, the extensions being highlighted through boldface fonts, is employed for this purpose. We will first describe the run-time scenario in terms of the objects and transactions that model each application. Then, we walk the reader through the code describing the program behavior. We conclude the program description with a discussion of the ease of mapping application scenarios to well-defined execution behaviours of extended transaction models. In particular, we highlight the need for multiple transaction models, so as to exploit their unique virtues, in each of the applications.

[1]We assume that a session manager, a standard service in collaborative applications, provides the end users with transaction and resource identifiers at run-time. This service, in turn, may need to consult with the corresponding users and operating system services for ensuring coordination and cooperation.

```
Class Document{
     typedef struct Resource{
          Data Section;
          Lock SectLock;
}
public:
     write();
}
Document::Write()
{
     TransactionId[1] Coauthor, System, Reincarnation, Reviewer;
     ResourceId CommonSections, CheckPointSections,
          FinishedSections, ToBePolishedSections;
          .
     Cooperate(CommonSections, Coauthor)
          .
     If(Reviewer is not ready))
          SerialSplit(CheckpointSections, System, Reincarnation)
     else
          IndependentSplit(FinishedSections, ToBePolishedSections, Reviewer, Reincarnation)
}
```

Figure 2: Robust Collaborative Editing

5.1 Collaborative Editor

We have shown the code for a collaborative editing session in Figure 2. The document file is an object Document with an array of disjoint data fragments Section and a single operation Write. All the transactions in the editor application are spawned by invoking the object method Document::Write(). The editing task of accessing multiple sections of the document by a single author starts as an atomic transaction Author which interacts with concurrent transactions Reviewer and Coauthor, representing a reviewer and coauthor respectively. Author can restructure itself dynamically as another transaction Reincarnation for periodic checkpoints and releasing unwanted locks.

The robust editing task, which the Author transaction encapsulates, can share only the desired sections CommonSections with the cooperating transaction Coauthor in conflicting modes (e.g., read-write). If actual conflicts do arise, they can eventually be resolved through user intervention.

Also, the Author transaction can save its most recent changes CheckpointSections on stable storage without relinquishing control, by delegating the corresponding data items to System which can be viewed as the root transaction and locks to the serial split transaction Reincarnation. Alternatively upon request from a reviewer, the Author transaction can divide its resources into disjoint partitions FinishedSections and ToBeFinishedSections, among the independent split transactions Reviewer and Reincarnation.

Now, we will describe why the robust collaborative editing task needs to exploit multiple transaction models for accessing various resources. The editing transaction Author has to share CommonSections in an interleaved manner with Coauthor. Also, Author needs to delegate the sections that it has completed, FinishedSections, to Reviewer. Cooperating transactions allow the sharing between Author and Coauthor whereas independent split transactions facilitate the delegation from Author to Reviewer. The two distinct types of interaction, exemplified by the above scenarios, cannot be implemented using either cooperating or split transactions alone. For example, it is not natural to use cooperating transactions for delegating FinishedSections from Author to Reviewer, because Author may not be willing to cooperate with Reviewer since any potential conflicts between them cannot be resolved as easily as those between coauthors. Similarly, split transactions do not capture interleaved resource sharing between Author and Coauthor. The editor application thus should foresee many such scenarios and facilitate authors to choose dynamically a combination of transaction models.

5.2 Meeting Scheduler

We have shown the code for the meeting scheduler program in Figure 3. The scheduler application can be programmed as an object Calendar that encapsulates a particular meeting. The Calendar object has many constituent objects of type Diary belonging to various participants, and a single operation Negotiate visible to the users trying to coordinate appointments. The diary objects themselves consist of the data items Slot corresponding to the many hourly slots in a single day and synchronization variables SlotLock protecting against concurrent access. We consider the application scenario in which concurrent transactions FirstPhase, LastPhase and ResourceSchedule, representing the various phases of a scheduling task, are initiated by a coordinator (secretary). These transactions interact with each other and the Colleague transaction belonging to another coordinator. All the transactions are spawned by invoking the object method Calendar::Negotiate().

```
Class Calendar {
     Diary diary[NumParticipant];
public:
     Negotiate();
}

Class Diary{
     typedef struct Resource{
          Data Slot;
          Lock SlotLock;
     }
     ResourceId AppSlot[NumSlots];
protected:
     Schedule();
}

Calendar::Negotiate()
{
     TransactionId System, LastPhase, ResourceSchedule;
     ResourceId UnAvailable[], Available[];

     // asynchronously invoke Diary::Schedule() method for each
     // participant and consult for the availability of desired slots

     If(no consensus yet)
          IndependentSplit(UnAvailable, Available, System, LastPhase)
     else
          SerialSplit(Available, System, ResourceSchedule)
}

Diary::Schedule()
{
     TransactionId Colleague;
     For each desired slot i obtained as input
          Cooperate(AppSlot[i], Colleague)
          decide whether AppSlot[i] is available and output the status
}
```

Figure 3: Robust Meeting Scheduling

The robust computation FirstPhase consults the individual diary objects and partitions its original resources into unwanted and compromise time slots. It then delegates the compromise slots Available to the independent split transaction LastPhase. The unwanted time slots UnAvailable are released to any other coordinator wanting them through the root transaction System. The negotiation transaction LastPhase reaches a consensus among the group members. It then saves the groups's consensus slots on permanent storage through the System transaction and entrusts the associated locks to the serial split transaction ResourceSchedule. The ResourceSchedule transaction cooperates with another coordinator transaction Colleague only on the desired resources - the hourly slots for the meeting room, through a nested invocation of the corresponding object method Diary::Schedule().

Now, we will examine why the robust meeting scheduling task needs to employ multiple transaction models for flexibility. On the completion of the negotiation transaction FirstPhase, the compromise slots Available have to be passed to the LastPhase transaction. On the otherhand, ResourceSchedule and Colleague transaction share the resources AppSlot - belonging to the diary object of the meeting room - in an interleaved manner. Independent split transactions facilitate the delegation of Available from FirstPhase to LastPhase. Cooperating transactions cannot model the interaction between FirstPhase and LastPhase because of the lack of application-specific knowledge to resolve any potential conflicts. However, cooperating transactions naturally model the sharing between ResourceSchedule and Colleague transactions. Such sharing cannot be implemented using independent or serial split transactions; because even though split transactions can share resources, only one of the transactions actively executes whereas the other one should commit its results immediately. Thus, the scheduler application could exploit the various transaction models for accessing different resources.

5.3 Ease of Programming

The key task in programming robust collaborative applications thus is to identify scenarios that befit a certain transaction model. We believe that application programmers familiar with such scenarios can exploit the corresponding well-defined consistency semantics of a variety of extended transaction models. Thus, they can easily choose the particular model that satisfies their unique needs. System designers for robust collaboration, in turn, should appreciate the need for, and provide, many such transaction models on a single software platform.

6 System Mechanisms for Implementing Extended Transactions

Our goal in this section is to outline an implementation of multiple transaction models using only low-level system mechanisms. However, the desired flexibility requires a few minor modifications to such primitive mechanisms that are already provided by existing systems. This design decision to tailor existing low-level facilities arises from efficiency considerations of providing minimal system support for robust applications. Our approach is simple also, because it requires no major changes to the basic abstractions provided by common operating systems. We only add a few additional operations or enhance the semantics of a few existing ones.

Objects provide an attractive structuring paradigm in system and application software, particularly in a distributed environment [8]. For implementing extended transaction models, we assume a system in which a *thread* corresponds to a computation structured using a transaction, and *objects* encapsulate passive data and code. Objects are made of a lower level abstraction called *segment.* A thread has to invoke an operation defined by the object and execute its code to manipulate the data encapsulated by the object. Multiple threads can concurrently execute in the same object. We introduce a low-level abstraction *synchronizer* which encompasses multi-mode locks used for coordinating such concurrent threads.

Two major functions: recovery and concurrency control, need to be supported by the system mechanisms for flexible robustness. We describe below the primitive operations for segments and synchronizers which can be used to implement multiple transaction models. These operation can be built on top of virtual memory and file system support provided by the Unix operating system. Light-weight recoverable virtual memory [27] is one instance of such a recovery manager. We have however implemented this functionality at the OS level [6] since it was more natural to do so in the context of our testbed, the Clouds operating system. We next show how these mechanisms simplify implementing multiple transaction models.

6.1 Recovery Control

To deal with failures, we allow segments to have multiple versions. Versions can reside on volatile or permanent storage. A segment can be initialized by copying the contents of another segment to it using the **clone** call. Multiple segments can be atomically committed (copied) to another set of segments using the **delegate** call. **Clone** and **delegate** calls facilitate recovery control needed to implement many transaction models by helping avoid in-place data modification. These calls can be used to establish relationships between the versions of a segment, e.g., a segment can be cloned from and delegated to different versions. Creation of such relationships is dictated by the encapsulating transaction model and enforced by choosing appropriate parameters for these calls. Thus our system does not assume any particular transaction model.

6.2 Concurrency Control

Acquire and **release** operations allow the synchronizers to be transferred between threads. These operations can be performed tentatively on specification; acquired locks need to be released when a robust thread fails, and locks are not released till the outcome of the thread is determined. A thread may obtain a synchronizer in a conflicting mode with other threads specified through the **acquire** operation. **ResilientSyncBegin** call can be used to specify the coordinator thread that should commit or abort the synchronization operations performed by a robust thread. **SyncCommit** and **SyncAbort** operations respectively can be used by the coordinator to make the tentative synchronization operations performed by the cohort thread permanent on its success, or to undo the operations on its failure.

6.3 An Implementation of Extended Transactions

The code in Figure 4 shows the skeletons of an implementation of split and cooperating transactions using the proposed system mechanisms. For brevity, we have omitted the code executed by the Coordinator thread which commits or aborts the synchronization effects of the robust computation depending on its outcome. An application programmer has to specify just the extended transaction model that dictates the particular resource access, not the system calls that manipulate the data items and synchronization variables. The run-time library and preprocessor support for implementing many extended transaction models we have proposed earlier [7] thus simplify the programmer's task by making the underlying system primitives for recovery and concurrency control transparent.

We assume that the object RobustObj, as shown in Figure 4, has data segments CoopSeg, DepSeg and IndepSeg protected by the synchronizers CoopSync, DepSync and IndepSync respectively. The **acquire** call is used for obtaining the lock CoopSync, even if the Cooperator thread holds the lock in a mode that conflicts with CoopHoldMode. Next, the segment SourceSeg used for

```
RobustObj::method(){
     // We assume a system service, thread manager, that keeps track of these identifiers
     // Segment CoopSeg, DepSeg, IndepSeg;
     // Synchronizer CoopSync, DepSync, IndepSync;
     // Thread Coordinator, Cooperator, Dependent, DependedUpon, Independent;
                    .
     ResilientSyncBegin([CoopSync,DepSync,IndepSync],Coordinator);
                    .
     /* Cooperate */
     acquire(CoopSync,ReqMode,[Cooperator, CoopHoldMode]);
     SourceSeg = HasSegment(Cooperator, CoopSeg);
     clone(SourceSeg, CurrentSeg);
                    .
          /* method code */
                    .
     /* SerialSplit */
     release(DepSync, Dependent, DepMode)
     DepDestSeg = HasSegment(DependedUpon, DepSeg);
                    .
     /* IndependentSplit */
     release(IndepSync, Independent, IndepMode)
     IndepDestSeg = HasSegment(Independent, IndepSeg);
                    .
     delegate(DepSeg, IndepSeg, DepDestSeg, IndepDestSeg);}
```

Figure 4: Object Being Invoked as a Robust Computation

initializing a version of the data segment CoopSeg is determined. This is used in the **clone** call. The transient segment CurrentSeg, thus created, can be manipulated by the robust computation. On completion, the computation, structured originally as an atomic transaction, uses the **release** call to propagate the lock DepSync to the serial split transaction Dependent and checkpoints the changes in the corresponding data segment DepSeg by delegating it to its counterpart DependedUpon. The lock IndepSync is released as well as the corresponding data segment IndepSeg is delegated to the independent split transaction Independent.

Thus, we have basically decomposed the traditional transaction manager calls as follows: **Be-**

ginTransaction is split into primitives for acquiring the relevant synchronization variables (**acquire**) and initializing data versions (**clone**). **EndTransaction** consists of releasing the synchronization variables (**release**) and committing the corresponding data versions (**delegate**). This separation of concurrency and recovery control mechanisms necessitates specifying a coordinator transaction (**ResilientSyncBegin**) responsible for reliably making the executing transaction's synchronization effects permanent (**SyncCommit**), or void (**SyncAbort**), on its termination.

6.4 Data Sharing among Flexible Robust Computations

We now provide a new perspective on data sharing among flexible, robust computations so as to clearly distinguish the salient features of the extensions and relaxation of the serializability correctness criterion. This discussion should show the simplicity and efficiency of our system mechanisms for implementing extended transactions.

A flat atomic transaction shares its changes to all the manipulated data items with the external world, only upon its committal, by releasing the corresponding locks. A nested transaction on the other hand allows such changes to be selectively shared among its subactions. Type-specific concurrency control facilitates sharing among all the transactions invoking particular methods of an object. However, it is desirable only a few such cooperating transactions share data in an interleaved fashion. A serial split transaction promotes data sharing between its dynamically spawned dependent and depended-upon split transactions. Independent split transactions conversely divide shared data into disjoint partitions.

Based on these data sharing patterns in extended transaction models, we can now justify the design rationale for our system primitives. The **acquire** system call facilitates select interleaved resource sharing among cooperating transactions. Independent split transactions exploit the **delegate** and **release** primitives, that split and entrust the ordinary variables and associated synchronization variables accessed by a transaction atomically to multiple transactions. Serial split transactions call for the separation of the **release** and **delegate** primitives since sometimes it is necessary to maintain concurrency atomicity, but not failure atomicity.

7 Conclusion

We summarize the main contributions of this paper as follows:

1. identifying the characteristics of collaborative applications - long duration and cooperation - that cannot be accommodated by the atomic transaction model,
2. trace the evolution of transaction models that match the advanced application needs,
3. show how collaborative applications need more than one extended transaction model and a combination of split and cooperating transactions simplifies their programming,
4. illustrate the applicability of low-level system mechanisms for implementing multiple transaction models.

Thus we have developed flexible support for realizing robust groupware in a systematic manner. We have not provided any measurements from our prototype system; because such metrics which quantify the virtues of extended transaction models are not obvious yet. One of our future directions is to identify such parameters and compare performance gains through modelling, and to validate our results using experimentation.

References

[1] B. Badrinath and K. Ramamritham. Semantics-based concurrency control: beyond commutativity. *ACM Transactions on Database Systems*, 17(1):163–99, March 1992.

[2] D. Beard et al. A Visual Calendar for Scheduling Group Meetings. In *Proceedings of the Conf. on Computer-Supported Cooperative Work*. ACM, 1990.

[3] A. Biliris et al. ASSET: A system for supporting extended transactions. In *1994 ACM SIGMOD International Conference on Management of Data*. ACM, June 1994.

[4] K. Birman. Integrating runtime consistency models for distributed computing. *Journal of Parallel and Distributed Computing*, 23(2), November 1994.

[5] K. Birman and T. Joseph. Exploiting Virtual Synchrony in Distributed Systems. In *Proceedings of the 11th Symposium On Operating System Principles*, pages 123–137. ACM, October 1987.

[6] M. Chelliah and M. Ahamad. System Mechanisms for Distributed Object-Based Fault-Tolerant Computing. Technical Report GIT-CC-92/23, Georgia Tech., 1992. An abridged version of this paper appeared in the 1994 IEEE Workshop on Fault-Tolerant Parallel and Distributed Systems.

[7] M. Chelliah and M. Ahamad. Multi-model Fault-tolerant Programming in Distributed Object-Based Systems. Technical Report GIT-CC-93/78, Georgia Tech., 1993.

[8] R. S. Chin and S. T. Chanson. Distributed Object-based Programming Systems. *ACM Computing Surveys*, 3(1):91–124, March 1991.

[9] A. Elmagarmid. *Database Transaction Models for Advanced Applications.* Morgan-Kaufmann Publishing Company, 1992.

[10] K. Eswaran et al. The Notion of Consistency and Predicate Locks in a Database System. *Communications of the ACM*, November 1976.

[11] R.S. Fish et al. Collaborative document production using quilt. In *Proceedings of the Conf. Computer-Supported Cooperative Work.* ACM, 1988.

[12] G. Foster and M. Stefik. Cognoter, theory and practice of a collaborative tool. In *Proceedings of Computer-Supported Cooperative Work.* ACM, 1986.

[13] H. Garcia-Molina. Using Semantic Knowledge for Transaction Processing Databases. *ACM Transactions on Database Systems*, 8(2):186–213, March 1983.

[14] H. Garcia-molina and K. Salem. SAGAS. In *Proceedings of the SIGMOD Intl. Conf. On Management Of Data.* ACM, 1987.

[15] D. Georgakopoulos, M. Hornick, P. Krychniak, and F. Manola. Specification and management of extended transactions in a Programmable Transaction Environment. In *Proceedings of the 10th International Conference on Data Engineering.* IEEE, February 1994.

[16] I. Greif and S. Sarin. Data Sharing in Group Work. *ACM Transactions On Office Information Systems*, April 1987.

[17] I. Greif, R. Seliger, and W. Weihl. A Case Study of CES: A Distributed Collaborative Editing System Implemented In Argus. *IEEE Transactions On Software Engineering*, September 1992.

[18] G. Kaiser. A Flexible Transaction Model for Software Engineering. In *Proceedings of the Intl. Conf. on Data Engineering*, 1990.

[19] M.J. Knister and A. Prakash. Distedit: A distributed toolkit for supporting multiple group editors. In *Proceedings of the Conference on CSCW.* ACM, 1990.

[20] H. Korth, E. Levy, and A. Silberschatz. A Formal Approach to Recovery by Compensating Transactions. In *Proceedings of the 16th Conference on VLDB*, 1990.

[21] H. Korth and G. Speegle. Formal Model of Correctness without Serializability. In *Proceedings of the SIGMOD Intl. Conf. on Management of Data.* ACM, 1988.

[22] R. Kroeger et al. RelaX - An Extensible Architecture Supporting Reliable Distributed Applications. In *Proceedings of the 9th Symposium On Reliable Distributed Systems.* IEEE, 1990.

[23] B. Liskov et al. Implementation of Argus. In *Proceedings of the 11th Symposium On Operating Systems Principles*, pages 111–22. ACM, 1987.

[24] E. Moss. *Nested Transactions: An Approach to Reliable Distributed Computing.* MIT Press, 1985.

[25] M. Nodine and S. Zdonik. Cooperative Transaction Hierarchies: A Transaction Model to Support Design Applications. In *Proceedings of the 16th Conference on VLDB*. IEEE, 1990.

[26] C. Pu and G. Kaiser. Split-Transactions for Open-Ended Activities. In *Proceedings of the 14th Conference on VLDB*. IEEE, September 1988.

[27] M. Satyanarayanan. Light-Weight Recoverable Virtual Memory. In *Proceedings of the 14th Symposium On Operating System Principles*. ACM, December 1993.

[28] P.M. Schwarz and A.Z. Spector. Synchronizing Shared Abstract Types. *ACM Transactions On Computer Systems*, August 1984.

[29] S.K. Shrivastava and S. Wheater. Implementing Fault-tolerant Distributed Applications Using Objects and Multi-coloured Actions. In *Proceedings of the 10th Intl. Conf. on Distributed Computing Systems*. IEEE, March 1990.

[30] A. Tripathi. An Overview of the Nexus Distributed Operating system Design. *IEEE Transactions On Software Engineering*, June 1989.

[31] W. Weihl. Specification and Implementation of Resilient Atomic Data Types. Technical Report MIT/LCS/TR-314, Massachusetts Institute of Technology, 1984.

[32] M. Weinstein et al. Transactions and Synchronization in a Distributed Operating System. In *Proceedings of the 10th Symposium on Operating System Principles*. ACM, October 1985.