A Framework Supporting Integrated Transaction and Session Management for Cooperative Applications

Maria Beatriz Felgar de Toledo

Relatório Técnico IC–98-18

Maio de 1998

A Framework Supporting Integrated Transaction and Session Management for Cooperative Applications

Maria Beatriz Felgar de Toledo [*]

Abstract

This report presents a framework of integrated transaction and session management to meet the requirements of applications based on group work. Transaction management is provided in order to guarantee data consistency and recovery from failures with the additional benefit of structuring complex applications as nested transactions. Visibility between users is achieved by allowing unfinished work to be transferred between users or by establishing a synchronous session. Within a session users can update shared objects in turns and be notified of others' actions synchronously.

1 Introduction

This report focuses on applications based on group work analysing their requirements and proposing a framework to meet their demands. It is argued that these applications should have support for transaction and session management in an integrated manner. Firstly it recognizes the benefits of transactions with their primary goal of achieving consistency [5]. Traditional mechanisms were, however, replaced in order to offer more visibility to users working in collaboration. In spite of its advantages, transaction support was not considered enough and novel features were included into the framework to allow users' interactions in synchronous or asynchronous mode.

The framework presented in this report aims at opening transactions and thus providing the required consistency and visibility for developing cooperative applications. Another effort towards a synchronous mode of interactions is made through the integration of session management within the same framework.

The rest of the report is organized as follows. In section 2, the requirements of cooperative applications are analysed. The framework is introduced in section 3 with a discussion on version management. In the following section, transaction and session management are proposed. In section 5, an example of use is presented. Conclusions and future work are discussed in the final section.

[*]Instituto de Computação, Universidade Estadual de Campinas, 13081-970 Campinas, SP.

2 Requirement Analysis

Applications based on group work will have to meet the requirements analysed below.

2.1 Flexible Correctness Criteria

Correctness criteria based on serializability [5] are suitable for short-duration applications with a competitive access pattern. Applications based on group work, on the other hand, require more flexible criteria which take into consideration their long duration and the interactions among members of a group. New criteria based on data semantics were proposed in [1, 19]. However, the difficulty of specifying consistency constraints and detecting violations automatically originated alternative proposals. In [18], the group of users developing a project cooperatively defines the appropriate updates dynamically before they are applied to the database.

2.2 Support for Long Duration

Some applications may last hours or even weeks. Traditional methods for concurrency control [5] are based on transaction blocking or rollback in order to solve conflicts. Recovery methods [5] are usually based on transaction rollback to guarantee the atomicity property. When transactions have long duration, making transactions waiting locks for a long period or undoing completely a transaction would degrade the system performance.

Alternative strategies allow early release of results [10, 13] and relaxed atomicity through periodic checkpoints or subtransaction commitment [23].

2.3 Support for Group Work

Advanced applications related to project development involve several designers working cooperatively. A complex project is subdivided into simpler tasks which are assigned to a designer or group of designers. Each designer may interact more intensively with a subset of the system data but may also need to interact with other designers and the work developed by them.

In cooperative environments, groups of designers can be modelled by hierarchical transactions [4, 19, 25]. Moreover, transactions must not impose visibility barriers among designers working cooperatively. Methods for concurrency control based on serializability would restrict cooperation and should be abandoned.

More recently advanced transaction models [8] have been criticized [2, 7] due to its inadequacy to operate in real working environments. Some propose [2, 17] a shift to workflow models to address a wider range of requirements.

2.4 Support for Data Evolution

As projects are developed incrementally, each stage must be stored not only for documentation purposes but also to aid the development process. For instance, if errors are detected in the current stage of a project, development may be restarted from any of the previous stages. Moreover, the maintenance of versions can improve concurrency.

The stages of a project can be represented by distinct versions organized by ancestor/descendant relationships. When complex objects are supported, additional structure must be provided to maintain the hierarchy of component objects. Mechanisms for storing and accessing versions [14] are thus required in addition to mechanisms for grouping the valid versions of components of a complex object [14].

2.5 Support for Synchronous and Asynchronous Interactions

Cooperative applications including some groupware applications require support for interactions among users working in collaboration. This support may be provided asynchronously as in [9, 12] where locks have two dimensions. The first dimension is the lock mode while the other one specifies when notifications should be received. This allows a transaction to be notified when others update or request one of its allocated data items.

Real-time groupware applications, on the other hand, require users to be aware of others' actions synchronously. This is the approach of the multiuser editor Grove [7] where an update by any user in the session is immediately reflected into the shared environment.

However many applications often switch from one interaction mode to another. This characteristic demands from the underlying system support for both modes of interaction. Furthermore an application should be allowed to change modes rapidly.

2.6 Support for Flexible Interactions

Users in cooperative applications are often distributed over many locations and connected by a communication network. With the greater availability of portable computers and wireless communication, mobility issues must also be addressed. Thus users can work from remote locations possibly shifting from one location to another without the need of being connected to a traditional network.

The Coda File System [22] was one of the first systems to support users in three modes of operation depending on the type of network he or she is connected: disconnected, weakly connected (to low-bandwidth networks) and fully connected (to high speed networks). Other works [24, 11] have also treated the problem of mobility. More recently, the Exotica project [3] integrates mobility to a cooperative environment.

2.7 Support for Control Flow

Traditionally, the control flow between steps within an application was managed by the application itself. Thus a long-running computation could be structured as a sequence of short-duration transactions without any system support for synchronization or failure handling across steps.

Gradually some support started to appear in models such as [6, 10, 26, 27]. Sagas [10] allows explicit control flow with an automatic compensation capability which can be used for backward recovery after a failure. More flexible approaches based on the specification of dependencies between transactions can be found in [6, 27]. The Contract model [26] provides two independent notions to support complex activities: steps that are elementary units of work and scripts which describe the structure of an activity in terms of sequence, branch, loop and parallel constructors.

3 Version Management

Advanced applications related to project development require support for the evolution of complex data. Within the proposed framework, two classes provide the required support: Version History and Version Class.

An application object to be versionable must inherit the properties of Version class. An instance of Version History class organizes the derivation hierarchy for a given application object. Moreover, an application object may be composed of other objects and participate in one or more transactions. The diagram in figure 1 shows the relationships between classes in the proposed model. The Transaction class and its specializations will be discussed in the next section.

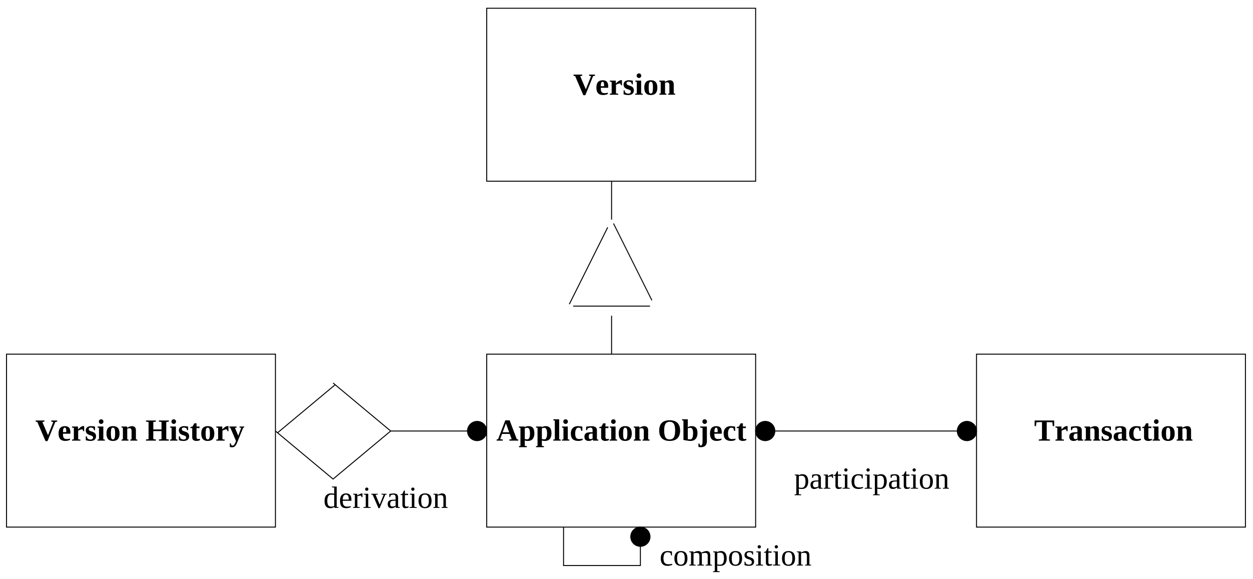

Figure 1: Version management within the framework

Version management in the proposed framework provides the features below:

- **identification of objects and their versions:** an object is composed of one instance of the Version History class and a series of versions representing distinct stages in the object history. A Version History instance receives an identifier at creation time consisting of two parts: a work area identifier and a counter which uniquely identifies the object in the work area. When a version of an object is created, it is assigned an identifier consisting of two parts: the Version History identifier and a counter which uniquely identifies the version in the object history.

- **resolution of dynamic references:** applications can use static or dynamic references to objects. A static reference points to a specific version of the object while a dynamic reference points to the Version History component of the object. This component stores the current version of the object to which the dynamic reference is resolved.

- **concurrency control:** the unit of locking is the version which can be requested either in reading or deriving mode. Many transactions can simultaneously allocate a version for reading but just one transaction can lock the current version in order to derive a new version from it. When a lock is requested on a complex object, each component must be locked as the composition hierarchy is traversed. Locking is implemented in two phases to guarantee atomicity: in the first phase, each component is locked in a prepare mode (for reading or deriving) and if all components can be successfully locked the composition hierarchy is traversed a second time for committing the locks. Otherwise, the prepare locks are released.

- **creation and manipulation of version histories:** the Version History class provides operations for creating/removing Version History instances and check in/check out Version History instances to/from transaction work areas.

- **creation and manipulation of versions:** the Version class provides operations for creating/removing object versions and check in/check out object versions to/from transaction work areas.

- **update of derivation hierarchy:** the version histories implemented by the Version History class are linear. The current version is always the latest effective version from which a new version can be derived. The Version History class has an attribute for storing the current version of an object and operations for inserting new versions into the version history as they are derived.

- **update of composition hierarchy:** each version of an object may be composed of other versions. The Version class has an attribute to store references to component objects. Either static or dynamic references are allowed. The Version class also provides operations for updating the composition hierarchy.

The Version History class provides information about the version history of application objects and operations to manipulate it. The Version class together with the Version History class provide structure and operations related to version support. More details can be found in [15, 16].

4 A Framework for Group Work

Cooperative applications, in particular those for project development, use databases intensively. Concurrency control is required in order to allow data sharing and still preserve consistency. Automatic recovery from failures is also important. Transaction support is therefore necessary although not in the traditional from. In the proposed framework, it is provided through hierarchical transactions modelling groups of users working cooperatively. Each level in the hierarchy has its working area where objects can be checked out from higher level areas. Unfinished work can be copied or transferred between work areas. Thus increased visibility among collaborating users is achieved.

In order to satisfy the requirements of a wider range of applications, other mechanisms allowing more flexible interactions between users are also provided within the framework:

- mechanisms for asynchronous interactions: when objects are requested for loan or concession, a notification is sent to the user holding a lock on the object. This notification should contain the requesting user identification, the user's working area, the object identification and the desired operation. The user holding a lock may agree with the operation transferring the object to the requesting user's area. If the object is not transferred within a certain period, the operation returns unsuccessfully.
- mechanisms for synchronous interactions: users can initiate a session and update an object in turns. Updates invoked by a user will be propagated to the other users in the session when his/her period of time to access the object expires.

The diagram in figure 2 presents the framework according to the OMT notation [21]. In this diagram, the classes Cooperative Transaction, Group Transaction and User Transaction provide transaction management and the Session class supports synchronous interactions between users in a session. The relationships between classes and their operations are also presented. Each of the classes in the framework is described below.

4.1 The Skeleton Class

The Skeleton class as the root of the class hierarchy (figure 2) defines the common interface for all other classes in the model.

The operations of the Skeleton class are the following:

1. **Begin**
2. **Terminate**
3. **RequestObject**
4. **ReleaseObject**

The above operations are all abstract and their implementation will be defined in the subclasses of the Skeleton class.

4.2 The Cooperative Transaction Class

The Cooperative Transaction class inherits the operations from the Skeleton class defining the common operations of its two subclasses: Group class and User class (figure 2). It will not be instantiated by an application. Instead, a cooperative application can be structured

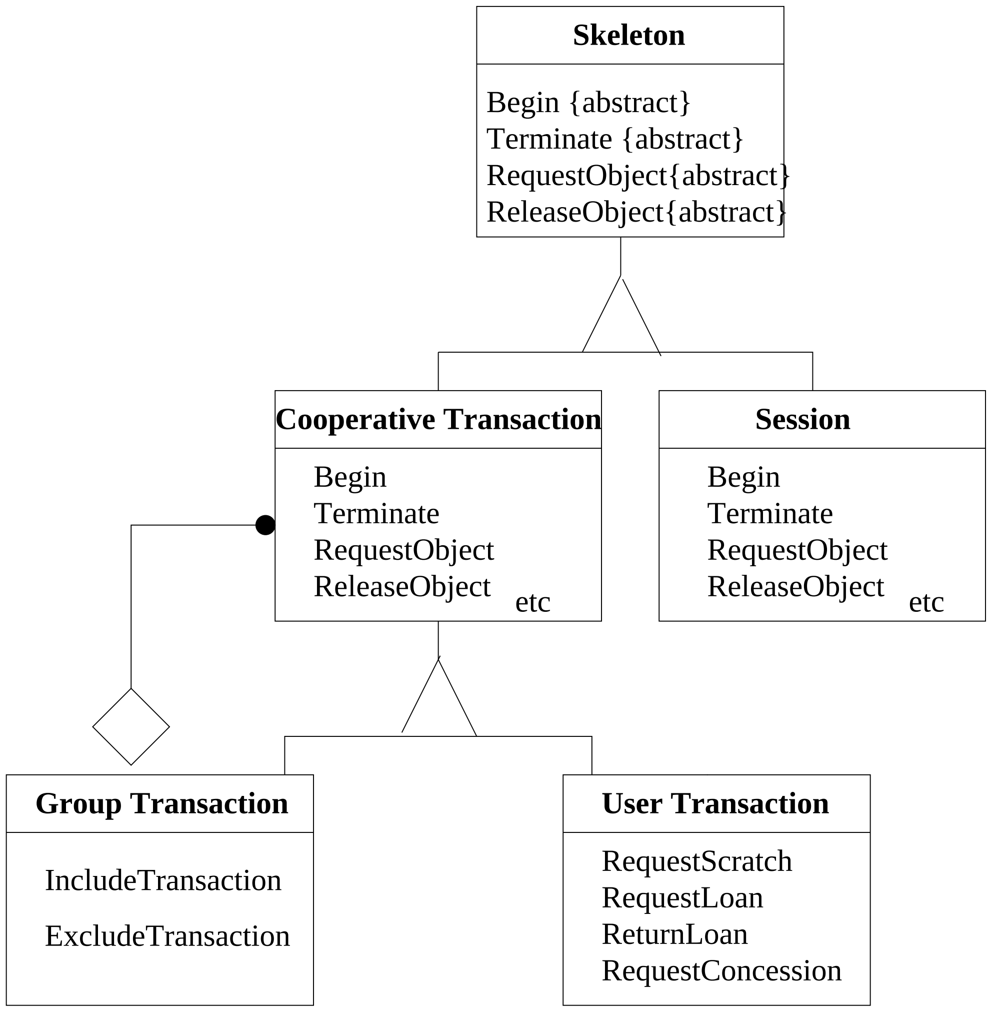

Figure 2: Structure of classes within the framework

as a hierarchy of transactions whose internal nodes are instances of the Group class and leaves are instances of the User class. This hierarchy can model the groups working in collaboration. The mechanisms to allow collaboration among users are based on check in/check out protocols to transfer objects between transaction work areas.

The operations defined in the class are the following:

1. **Begin** initiates a cooperative transaction creating a new work area for the initiating transaction.

2. **Terminate** finishes the transaction according to the type of termination specified as parameter. The possible types of termination are: aborting, committing the child transactions which successfully committed, committing only if all child transactions committed and committing only if the majority of child transactions committed. A co-

operative transaction may only be committed if all lent objects were returned back to the committing transaction. At termination time, borrowed objects are returned back to their original transactions, scratch copies are discarded and the other participating objects are checked into the parent work area and unlocked.

3. **RequestObject** locks the specified object and checks it out from the parent work area (or other closest ancestor area) to the requesting transaction area.

4. **ReleaseObject** if the specified object is locked for updating, it is checked into the parent work area and released for reading only. If the object is locked for reading, it is unlocked and discarded from the cooperative transaction area.

5. **GetUsers** returns the list of users involved in the cooperative transaction.

6. **GetObjects** returns the list of objects stored in the cooperative transaction working area.

4.3 The Group Transaction Class

The Group Transaction class inherits the operations from the Cooperative Transaction class adding operations to construct the hierarchy of cooperative transactions.

The operations defined in the class are the following:

1. **IncludeTransaction** includes an initiating cooperative transaction as a child transaction of an on-going group transaction.

2. **ExcludeTransaction** notifies the parent transaction of the termination of its child transaction and the type of termination.

4.4 The User Transaction Class

The User Transaction class inherits the operations from the Cooperative Transaction class adding operations to transfer objects between transaction work areas.

Managers, instances of the User Transaction class, provide operations to allow collaboration among users. These operations relax serializability improving the visibility among users. Versions being derived by on-going transactions can be copied from one work area to another, borrowed from a transaction and later returned to the original transaction, and, finally, conceded to a second transaction that acquires the rights of the original transaction.

The operations requesting objects cause notifications to be sent to the user owning a lock on the object. A notification contains the requesting user's identification, the user's working area, the object identification and the requested operation (scratch, loan or concession). Upon its receipt, the user may decide to transfer or not the requested object version. If the request is unacceptable, the user does not need to take any special action as the operation will return after a time-out.

The operations defined in the class are the following:

1. **RequestScratch** requests an object from a specified transaction causing a notification to be sent to the user owing a lock on the object. This user may concede its copy invoking the transfer operation. The transaction which owns the version retains its copy and original privileges. The requesting transaction can invoke any operation on its version but has no right to check this version into its parent work area.

2. **RequestLoan** requests an object from a specified transaction causing a notification to be sent to the user owing a lock on the object. This user may agree with the loan invoking the transfer operation. The object is returned back to its original owner when the ReturnLoan operation is invoked. Then the original transaction re-acquires its previous privileges on the object.

3. **ReturnLoan** transfers the version of a borrowed object to the source work area. A notification is sent to warn the user that an object was returned.

4. **RequestConcession** requests an object from a specified transaction causing a notification to be sent to the user owing a lock on the object. This user may agree with the concession invoking the transfer operation. The original transaction looses its copy and concedes all privileges on the object to the requesting transaction.

5. **Transfer** transfers an object from the working area of the invoking transaction to a specified transaction area. The type of transfer must also be specified (copy, loan or concession) to determine the privileges of the requesting transaction on the object.

4.5 The Session Class

The Session class inherits the operations from the Skeleton class defining the implementation of a synchronous session among a set of users. When a session is created a coordinator is assigned for it and other members can be specified. The coordinator is responsible for inserting participating users and objects into the session. Participating users can access any of these objects according to the object update list. The first user in the list will have a certain period of time fixed by the coordinator to update its associated object. When this period is expired the next user in the list will receive a notification of his/her turn and all updates are propagated to the other users in the session. A session can be associated with a cooperative transaction or be independent.

The operations defined in the class are the following:

1. **Begin** initiates a session with an associated set of participants. The user who invoked the Begin operation will be the coordinator of the initiating session with the rights to insert/remove users into/from this session, to request objects to be used within the session, to release objects when updates are finished and to terminate the session.

2. **Terminate** finishes a session according to the type of termination specified as parameter. The possible outcomes are: discarding/committing all work done, committing partially (updating only the list of specified objects). At termination time, updated objects are checked into the parent work area (or the public area) and unlocked. Updates or objects released during the session can not be discarded.

3. **InsertSession** associates a session with a specified cooperative transaction. Objects can be checked out from the cooperative transaction area only and participating users must be a subset of the users in the cooperative transaction. It can be invoked by the coordinator only.

4. **RequestObject** inserts the specified object into the session object list and locks it.

5. **ReleaseObject** checks the specified object into the transaction work area (or public area), unlocks it and removes it from the session object list.

6. **IncludeUser** inserts the user into the user list for the associated session. This user can participate in the session updating its associated objects. It can be invoked by the coordinator only.

7. **RemoveUser** removes the user from the session user list. It can be invoked by the coordinator only.

8. **IncludeUserUpdateList** inserts the user into the end of an update list. This user will receive a notification when his/her turn to update the specified object arrives. It can be invoked by the user only and will succeed if the user belongs to the session user list.

9. **RemoveUserUpdateList** removes the user from an update list.

10. **GetUsers** returns the list of users involved in this session.

11. **GetUpdateList** returns the users in the update list for a given object. The list is ordered by the users' turn to update the object.

12. **SetTime** sets the amount of time to be used for each user in the specified update list. A user in the beginning of the list will receive a notification of his/her turn when s/he will be able to update the associated object. A notification will also be sent when his/her time expires. Then s/he will be put into the end of the list and her/his updates are propagated to the other users in the session. It can be invoked by the coordinator only.

5 Example of Use

A typical cooperative application is a software system developed by a team of programmers (in figure 3). For example, programmer Paul is responsible for developing module A but needs to see changes in module B developed by programmer Helen. While developing B, Helen also needs to see module A. This could be done creating an instance of the Group Transaction class (group transaction Tg in figure 3) and then each programmer creates an instance of the User Transaction class to be included into this group transaction (user transactions Tp and Th for Paul and Helen respectively). Paul will check out module A to his private area while Helen will check out module B.

Collaboration between Paul and Helen is achieved through the request of unfinished work of each other. Objects can be copied (RequestScratch operation), transferred temporarily (RequestLoan operation) between work areas or conceded (RequestConcession operation) from a user transaction to another.

If synchronous collaboration is required at some stage, either Paul or Helen can start a session and insert modules A and B into its object list. The session initiator who is also its coordinator will include in the session his/her collaborating partner and modules A and B into its object list. Then both will be able to update them in turn.

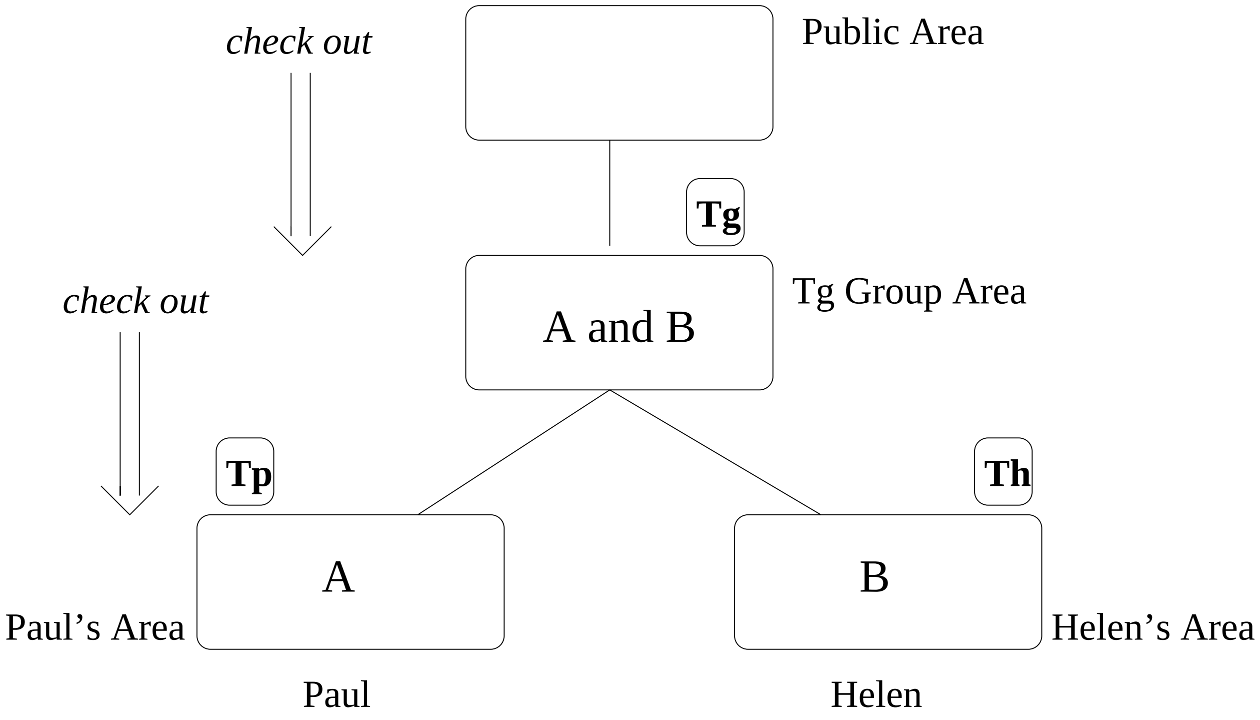

Figure 3: Example of collaboration

6 Conclusions

This reports starts by analysing the requirements of applications based on group work. Taking into account these requirements, it proposes a framework providing the following features:

- consistency: the maintenance of consistency is guaranteed by transaction mechanisms which support concurrency control based on locks and recovery from failures based on logging and versioning.
- flexibility: the required flexibility to model hierarchical groups working in collaboration is provided by the nesting of transactions. There are two types of transactions:

group transactions which are internal nodes of the hierarchy and user transactions which are the leaves of the hierarchy.

- visibility: users working cooperatively can see unfinished work of other users' by requesting special operations. These operations allow copying, borrowing or concession of objects.
- collaboration through asynchronous interactions: a user can request an object held by other user either for reading or updating it. The object's owner will receive a notification warning about the request and may agree or not with the proposed operation.
- collaboration through synchronous interactions: users can start a session and take turns in updating an object. Updates by an user are propagated to other users in the same session.
- maintenance of historical data: updates within a transaction creates another version for an object. Older versions have to be explicitly removed.

Transaction management within the framework is quite similar to other models [9, 25] in its support of hierarchical transactions for project development applications. However it has new features to open transactions for cooperative work. Firstly it includes notifications associated with operations for the transference of unfinished work between users. Moreover, in order to satisfy the requirements of a wider range of groupware applications, the framework integrates transaction and session management. In that way, it is possible for applications to choose a suitable style of interaction changing from asynchronous to synchronous mode or vice-versa as appropriate.

Future work includes the study of other features described in section 2 and not addressed by the framework. Moreover, a prototype is under development using a distributed platform. CORBA [20] was chosen as it allows the development of distributed applications based on objects.

References

[1] D. Agrawal, J. Bruno, El Abbadi, and V. Krishnaswamy. Managing concurrent activities in collaborative environments. In *CoopIS*, 1995.

[2] G. Alonso, D. Agrawal, El Abbadi, M. Kamath, R. Gunthor, and C. Mohan. Advanced transaction models in workflow contexts. In *Proc. 12th Int. Conf. Data Engineering*, February 1996.

[3] G. Alonso, R. Gunthor, M. Kamath, D. Agrawal, El Abbadi, and C. Mohan. Exotica/FMDC: Handling disconnected clients in a workflow management system. In *Proc. 3rd Int. Conf. Cooperative Information Systems*, May 1995.

[4] F. Bancilhon, Won Kim, and Henry F. Korth. A Model of CAD Transactions. In *Proc. of* 11^{th} *VLBD, Stockholm*, pages 25–33, 1985.

[5] Philip A. Bernstein, Vassos Hadzilacos, and Nathan Goodman. *Concurrency Control and Recovery in Database Systems.* Addison-Whesley Publishing Company, 1987.

[6] U. Dayal, M. Hsu, and R. Ladin. Organizing long-running activities with triggers and transactions. In *ACM SIGMOD*, 1990.

[7] C. A. Ellis, S. J. Gibbs, and G. L. Rein. Groupware: Some Issues and Experiences. *Communication of the ACM*, 1(34):39–58, January 1991.

[8] A. K. Elmagarmid, editor. *Transaction Models For Advanced Database Applications.* Morgan Kaufmann, 1992.

[9] M. Fernandez and S. Zdonik. Transaction groups: A model for controlling cooperative transactions. In *Third International Workshop On Persistent Object Systems*, January 1989.

[10] H. Garcia-Molina and K. Salem. Sagas. In *ACM SIGMOD*, pages 249–259, 1987.

[11] J. S. Heidemann, T. W. Page, R. Guy, and G. Popek. Primarily disconnected operation: Experiences with Ficus. In *Proc. Second Workshop on the Management of Replicated Data*, November 1992.

[12] M. F. Hornik and S. B. Zdonik. A Shared, Segmented Memory System for an Object-Oriented Database. *ACM Transaction on Office Information Systems*, 5(1):70–95, January 1987.

[13] Gail E. Kaiser and Calton Pu. Dynamic Restructuring of Transactions. In Ahmed K. Elmagarmid, editor, *Database Transaction Models for Advanced Applications*, 1992.

[14] Randy H. Katz. Toward a Unified Framework for Version Modeling in Engineering Databases. *ACM Computing Surveys*, 22(4):375–408, December 1990.

[15] G. M. A. Lima. Gerenciamento de transacoes: Um estudo e uma proposta. Master's thesis, Universidade Estadual de Campinas, 1996.

[16] G. M. A. Lima and M. B. F. Toledo. Um modelo de transacoes cooperativas integrado a um modelo de versoes. In *XII Simposio Brasileiro de Banco de Dados*, October 1997.

[17] C. Mohan. Advanced transaction models - survey and critique. In *ACM SIGMOD Int. Conf. Management of Data*, 1994. Tutorial.

[18] K. Narayanaswamy and Neil Goldman. Lazy Consistency: A Basis for Cooperative Software Development. In *CSCW Proceedings*, pages 257–264, November 1992.

[19] Marian H. Nodine, Sridhar Rasmaswamy, and Stanley B. Zdonik. A Cooperative Transaction Model for Design Databases. In Ahmed K. Elmagarmid, editor, *Database Transaction Models for Advanced Applications*, pages 53–85, 1992.

[20] Object Management Group. *The Common Object Request Broker: Architecture and Specification*, 1992. OMG Document Number 91.12.1, Revision 1.1.

[21] James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy, and William Lorensen. *Object-Oriented Modeling and Design*. Prentice-Hall, 1991.

[22] M. Satyanarayanan. Mobile information access. *IEEE Personal Communications*, 3(1), February 1996.

[23] S. K. Shrivastava and S. M. Wheater. Implementing Fault-Tolerant Distributed Applications Using Objects and Multi-Coloured Actions. *ICDCS-10*, pages 1–9, June 1990.

[24] D. Terry, A. Demers, K. Petersen, M. Spreitzer, M. Theimer, and B. Welch. Session guarantees for weakly consistent replicated data. In *Proc. Int. Conf. Parallel And Distributed Information Systems*, September 1994.

[25] Rainer Unland and Gunter Schlageter. A Transaction Manager Development Facility for Non Standard Database Systems. In Ahmed K. Elmagarmid, editor, *Database Transaction Models for Advanced Applications*, pages 53–85, 1992.

[26] H. Wachter and A. Reuter. The ConTract Model. In Ahmed K. Elmagarmid, editor, *Database Transaction Models for Advanced Applications*, 1992.

[27] A. Zhang, M. Nodine, B. Bhargava, and O. Bukhres. Ensuring relaxed atomicity for flexible transactions in multidatabase systems. In *ACM SIGMOD*, 1994.