An Adaptable Transaction Model for Traditional and Advanced Applications

Maria Beatriz Felgar de Toledo
George Marconi de Araújo Lima

Relatório Técnico IC–98-19

Maio de 1998

An Adaptable Transaction Model for Traditional and Advanced Applications

Maria Beatriz Felgar de Toledo
George Marconi de Araújo Lima*

Abstract

Transaction processing systems provide the required tools to maintain data consistency transparently. However, application diversity imposes conflicting requirements on transaction management. This report presents a framework of adaptable transaction management supporting short-duration applications, long-duration applications and cooperative applications. In addition to allowing an application to choose a transaction manager according to its characteristics, the proposed framework provides version management and support for synchronous and asynchronous interactions among users working in collaboration.

1 Introduction

The maintenance of data consistency requires complex mechanisms when applications execute concurrently or when failures interrupt the execution of an application. To make application development easier, responsibility to deal with these issues is transferred from the programmer to the underlying system.

The range of applications for which consistency is fundamental is very wide. They can be classified as:

- traditional applications: commercial/financial applications such as banking and flight reservation systems are in this category. They are characterized by:
  - short duration: applications may last in general fractions of seconds.
  - competitive access to shared data: sharing must be controlled in order to avoid interferences among applications.
- advanced applications: CAD/CAM and CASE are examples of advanced applications. They are characterized by:
  - long duration: applications may last hours or weeks.

*Instituto de Computação, Universidade Estadual de Campinas, 13081-970 Campinas, SP. Pesquisa desenvolvida com suporte financeiro parcial do CNPq — Conselho Nacional de Desenvolvimento e FAPESP — Fundação de Amparo à Pesquisa do Estado de São Paulo

- cooperative access to shared data: users working cooperatively need to be aware of actions performed by other users in their work group.

Traditionally concurrency control and recovery mechanisms provided within the transaction framework were fixed and designed to satisfy a single class of applications. More recently, advanced applications required a more flexible approach. In an environment characterized by application diversity, it is not acceptable to provide a single fixed mechanism. On the contrary, an application must tailor transaction management according to its specific requirements.

The framework presented in this report provides a variety of management styles allowing an application to choose the most suitable. The rest of the paper is organized as follows. In section 2, the requirements imposed on transaction management are analysed. The framework is introduced in section 3 with a discussion on transaction management. In the following section, other mechanisms for cooperative work within the framework are described. In section 5, a simple example of use is presented. Conclusions and future work are presented in the final section.

2 Requirement Analysis

A flexible transaction model to support both traditional and advanced applications will have to meet the requirements analysed below.

2.1 Adaptable Management

Transaction management must be adaptable to the characteristics of each specific application. Some applications may consist of a few updates or queries lasting for a short time, others may take a long time updating large portions of the database and finally many of them require group work. Thus one single style of management can not satisfy these wide range of requirements, each application must have the suitable support for its specific needs.

2.2 Flexible Correctness Criteria

Correctness criteria based on serializability [4] are suitable for short-duration applications with a competitive access pattern. Advanced applications, 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 [12, 1]. However, the difficulty of specifying consistency constraints and detecting violations automatically originated alternative proposals. In [11], the group of users developing a project cooperatively defines the appropriate updates dynamically before they are applied to the database.

2.3 Support for Long Duration

Some applications may last hours or even weeks. Traditional methods for concurrency control [4] are based on transaction blocking or rollback in order to solve conflicts. Recovery methods [4] are usually based on transaction rollback to guarantee the atomicity property. When transactions have long duration, it is not acceptable making transactions waiting locks for a long period or undoing completely a transaction. These strategies would degrade the system performance.

2.4 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 [3, 12, 16]. 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 [7] have been criticized [2, 6] due to its inadequacy to operate in real working environments. Some propose [2, 10] a shift to workflow models to address a wider range of requirements.

2.5 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 [8] are thus required in addition to mechanisms for grouping the valid versions of components of a complex object [8].

3 Adaptable Transaction Management

Traditional and advanced applications impose a wide range of requirements on transaction management. Some of them can not be provided by a single style of management. Therefore the main objective of this work is to focus on application diversity. An adaptable transaction model is proposed to satisfy the above applications and their specific requirements. Three transaction managing styles are distinguished:

- traditional transaction management for short-duration applications.
- traditional transaction management extended to allow early release of resources and periodic checkpointing. This style of management is oriented to long-duration applications.

- cooperative transaction management allowing groups of designers to be modelled by hierarchical transactions and objects to be transferred between work areas.

The diagram in figure 1 defines the model of adaptable management according to the OMT notation [14]. In this diagram, each class represents a transaction manager for a specific category of applications. The relationships between classes and their operations are also presented. An application must instantiate a class implementing the transaction manager suitable for its characteristics. Each of the classes in the framework is described below. More details about the specification of classes can be found in [5].

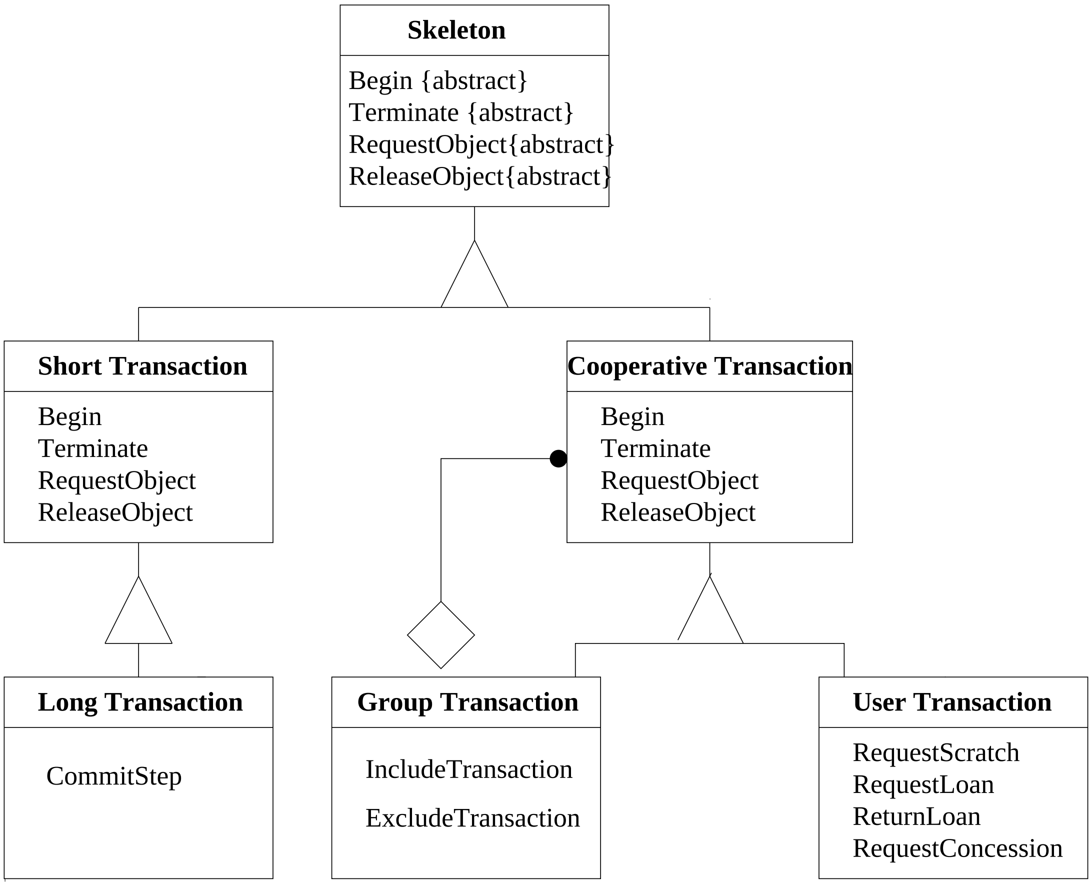

Figure 1: Transaction management within the framework

3.1 The Skeleton Class

The Skeleton class as the root of the class hierarchy (figure 1) defines the common interface for all kinds of transaction managers supported by 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.

3.2 The Short Transaction Class

The Short Transaction class inherits the interface of the Skeleton class (figure 1) defining the implementation of the inherited operations. An instance of the Short Transaction class provides traditional transaction management for short-duration applications.

Concurrency control is based on locking with access mode for reading and deriving versions. Locks can be requested at transaction beginning or during transaction execution. Locks are released during transaction execution or at transaction termination. Versions released before transaction termination are made available for reading only and are discarded if the transaction aborts later.

Recovery from failures uses information about transaction states and participating objects stored in a log. Updates requested on an object are recorded as a new version which is made effective after transaction commitment.

Therefore the properties of traditional transaction management can be preserved. Recovery mechanisms guarantee atomicity and durability. Locking maintains serializability but isolation is abandoned when early release of locks is requested. Strict two-phase locking can be achieved if locks are only released at transaction termination.

The operations defined in the class are the following:

1. **Begin** initiates a transaction.
2. **Terminate** finishes a transaction either by committing or aborting it according to the specified parameter. In the first case, new versions are added to the version graphs of updated objects. Otherwise, they are discarded. Requested locks are released.
3. **RequestObject** includes an object as a participating object of the on-going transaction if the object is successfully locked in the requested mode. The object identifier and the access mode are parameters of the operation.
4. **ReleaseObject** releases the object specified as a parameter for reading access only. If the transaction aborts afterwards, the object will be restored to its old state.

3.3 The Long Transaction Class

The Long Transaction class inherits from the Short Transaction class defining the additional operation CommitStep. This operation extends the traditional transaction management provided by the Short Transaction class allowing long-duration applications to release locks in advance and to establish checkpoints.

3.4 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 1). It will not be instantiated by an application. Instead, a cooperative application can be structured 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 cooperative transaction may only be committed if all lended 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.

3.5 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.

3.6 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. Requesting of a version may be denied if the lock mode held on the version does not allow the corresponding operation (copy, loan or concession). Lock modes are either for reading or deriving a version with an additional specification for the derivation mode indicating if the operations copy, loan or concession are allowed.

The operations defined in the class are the following:

1. **RequestScratch** searches the most recent version of the specified object in the hierarchy of work areas making a copy of this version into the requesting transaction work area. 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** searches the most recent version of the specified object in the hierarchy of work areas making a copy of this version into the requesting transaction work area. The original transaction maintains a copy which will be updated by the second transaction

   when it invokes the ReturnLoan operation. If the timeout specified as a parameter of the operation expires, the original transaction re-acquires its previous privileges on the object and the transaction which borrowed the object looses the right of updating the object in the source work area.

3. **ReturnLoan** transfers the version of a borrowed object to the source work area if the specified timeout has not expired. Otherwise it is discarded.

4. **RequestConcession** searches the most recent version of the specified object in the hierarchy of work areas transferring it to the requesting work area. The original transaction looses its copy and concedes all privileges on the object to the requesting transaction.

4 Support for Cooperative Work

Within the framework, support for cooperative applications is provided by the classes Cooperative Transaction, Group Transaction and User Transaction. These classes can be instantiated to construct hierarchical transactions modelling hierarchical groups of designers.

Furthermore, the framework provides version management to satisfy the requirements of data evolution and improved concurrency. Version management is discussed in [5, 9].

Presently, the model is being extended to support synchronous and asynchronous interactions among users working in collaboration. Notifications were added to the operations of the User Transaction class to warn a user that his/her version is being requested or returned by another user. With respect to synchronous work, a special class was defined to provide session support among groups of users. Users in a group transaction can start a session with an associated shared environment. The turn to update this environment is given according to an update list. More details can be found in [15].

5 Example of Use

A typical cooperative application is a software system developed by a team of programmers. 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 and then each programmer creates an instance of the User Transaction class. 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 in each other's area. 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 work area. This same user who will be the session coordinator will include in the session his/her collaborating partner. Then both will have access to all objects in the session area being able to update them in turn.

6 Conclusions

This report analyses the set of requirements imposed on transaction management by traditional and advanced applications concluding that a single style of management can not be adopted. Instead, it is necessary a framework providing a variety of styles. An application can thus choose the transaction manager more suitable for its specific requirements. Three transaction management styles are distinguished within the proposed framework:

- traditional management for short-duration applications.
- extended management to allow early release of resources and periodic checkpointing for long-duration applications.
- cooperative management to allow group modelling and transference of objects between work areas for applications based on group work.

Furthermore, it is also integrated within the framework:

- **support for synchronous collaboration:** in addition to cooperative transactions, collaboration can be achieved using sessions. Within a session synchronous work can be developed by a group of users.
- **support for data evolution:** as version management has a major impact on mechanisms of concurrency control and recovery, it was also integrated with transaction management into the same framework.

Following the specification of the framework, a prototype is under development using a distributed platform. CORBA [13] 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] F. Bancilhon, Won Kim, and Henry F. Korth. A Model of CAD Transactions. In *Proc. of* 11^{th} *VLBD, Stockholm*, pages 25–33, 1985.

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

[5] George Marconi de Araujo Lima. Gerenciamento de transacoes: Um estudo e uma proposta. Master's thesis, Universidade Estadual de Campinas, 1996.

[6] 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.

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

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

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

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

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

[12] 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.

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

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

[15] Maria Beatriz F. Toledo and George M. A. Lima. An integrated framework supporting transaction management and synchronous interactions for cooperative applications. 1998. submitted for publication.

[16] 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, 1994.