Transaction Support for an ANSA-Based Platform

Maria Beatriz Felgar de Toledo

Departamento de Ciência da Computação
Universidade de Campinas
Campinas, Brasil 13081-970

Gordon S. Blair

Computing Department
Lancaster University
Lancaster, UK LA1 4YR

Abstract

This article considers the problems of ensuring reliability in distributed cooperative systems through the transaction methodology. From the analysis of several transaction models, it is argued that the diversity of requirements in the distributed environment demands a novel approach. The transaction model proposed in this article aims at providing the required flexibility. In this model, applications have more control over transaction management through the specification of policies for services such as concurrency control and recovery. The approach also features a number of mechanisms specifically tailored for distributed cooperative systems including notifications and reservations. The model feasibility is demonstrated by layering the transaction facilities on top of a distributed platform based on ANSA.

1. Introduction

The transaction concept was introduced to guarantee consistency when data is shared by concurrent applications and when failures may disrupt execution [14].

Transactions were first developed for commercial/financial applications characterized by competitive access to resources. Mechanisms designed for this application area are well understood and effective. However, more flexible models are necessary to meet the demands of applications in distributed cooperative environments. This article proposes a novel approach in which applications can tailor transaction management according to their requirements through the specification of policies for concurrency control and recovery.

The article is organized as follows. Section 2 focuses on previous work. Section 3 describes a new transaction model to support applications in the distributed environment. Finally, conclusions are presented in section 4.

2. Related work

Several transaction models have been developed focusing on specific areas of application. Examining the range of approaches, these models can be classified into traditional models, distributed models and cooperative models which are discussed below.

2.1 Traditional models

The transaction construct was originally developed for commercial/financial applications in order to maintain data consistency in spite of concurrent access and failures. Transactions for this application area are usually simple, consisting of read/write operations on a limited range of data types. Moreover, transaction mechanisms take into account the short duration of these applications and the necessity of resource sharing in a controlled way. The properties associated with this traditional view of transactions are atomicity, serializability and isolation [5].

2.2 Distributed models

Transaction models for distributed systems extend the original model in order to increase its flexibility and performance. Two major approaches have been developed: models based on nesting and models based on abstract data types.

The first approach [19] allows applications to be hierarchically structured as nested transactions. New forms of concurrency control and recovery resulting from the nesting structure will also have an impact on performance. Gain in concurrency is achieved by running child transactions under the same parent in parallel. The recovery property of atomicity can be relaxed such that the failure of a child does not require the rollback of its parent. Parents can try their own recovery either by retrying the failed subtransaction or by performing another action.

The other approach provides programmers with a wider range of data types defined according to applications' need. Moreover, the integration of abstract

1063-6927/94 $03.00 © 1994 IEEE

data types with the transaction construct allows more efficient concurrency control and recovery mechanisms to be built through the semantic knowledge about types and their operations [1,3,9,10,11,26]. Whereas in the traditional approach, concurrency control algorithms are based only on the classification of operations into readers and writers, type-specific concurrency control can take into account the operations semantics to achieve a better interleaving of operations under serializability. For some applications, even departure from serializability will not compromise correctness. Type semantics can also be used in recovery mechanisms helping in decisions such as what part of the object state to save.

2.3 Cooperative models

In the cooperative environment, applications are partitioned in tasks performed by a group of users and usually last for a long period. Examples of applications are multi-user editors, electronic meeting rooms, design development systems and conferencing systems. Transactions must, therefore, be concerned with the interactions among users working on a common task and also the structure of groups within an application. In case of synchronous interactions, real-time constraints must also be considered in transaction design.

While the traditional approaches developed for competitive environments isolate users from each other, transactions for cooperative systems must not impair visibility among users working in collaboration. Enforcing serializability among the members of a group is therefore unacceptable. Atomicity is another property from the traditional model that is not suitable in an environment where applications have long duration.

Support for the propagation of changes within a group can be provided in the form of notifications [17] or activity logs [12,15]. The first technique allows users to be notified of conflicting accesses in which case they can try some compensating procedure such as reading the updated data. An activity log stores information about accesses to an object. A user can retrieve this information to find out about other users' actions on the shared object.

In design environments, projects are complex requiring tasks to be distributed among small groups. As a result, designers can be structured into hierarchies. It is therefore important that these groups are modelled in an efficient way. Nested transactions can be used to reflect group hierarchies such that designers working in direct cooperation will run subtransactions embedded into a higher-level transaction responsible for the interactions among the designers in the group [18,24]. Many levels can exist. For each group, a specific synchronization policy can be specified. Such policy determines the degree of cooperation among group members and the required isolation from other groups.

With respect to long transaction support, research has focused on extending concurrency control mechanisms in order to allow early release of resources [20,21,23] and recovery mechanisms to support periodic checkpoints.

The Grove editor [13] is an example of the use of semantics to allow synchronous update to a shared space. An update issued by an user is instantaneously executed on a local replica and propagated to remote replicas where transformed operations are executed.

Some other methods used in cooperative environments are discussed in [4,13,16].

3. A flexible approach for transaction management

Due to the variety of applications imposing different requirements on transaction management, one single fixed approach does not seem suitable for all applications. Rather, it is necessary for applications to have more control on transaction management. The approach to be discussed in this section splits responsibilities between the support environment, providing mechanisms for transaction management, and applications, selecting services according to their requirements [25]. In addition to supporting a variety of applications, the model is also concerned with group work and real-time. To allow group work, the model provides applications with notification of conflicting accesses to warn users of potential non-serializable behaviour. Support for real-time computing is provided through pre-allocation of resources.

The proposed model integrates transaction services into an ANSA-based architecture to be described in 3.1. This is followed by the extensions to ANSA in order to provide transaction support in 3.2. The objects providing transaction services are described in 3.3. Features such as real-time and group work are discussed in 3.4. An example of use is given in 3.5 and an evaluation of the model is presented in 3.6.

3.1 The Lancaster platform

This section describes the Lancaster multimedia project [8] for which the transaction model was developed. This project extends a distributed platform based on ANSA [2] in order to consider the real-time requirements of multimedia computing. Basic concepts about ANSA are described in 3.1.1 and required extensions for multimedia computing in 3.1.2.

3.1.1 The ANSA platform

The ANSA project defines a complete framework for the design and construction of distributed systems. The complexity of the architecture required it to be partitioned into five viewpoints:

. enterprise: describes interactions between the system, the organization and the environment in which the system and organization are placed.

. information: comprises the identification and location of information in addition to the description of information processing activities.

. computation: provides a description of how distributed programs may be written. It is based on distributed objects and their modes of interaction.

. engineering: describes the realization of the abstract computational model at a systems level.

. technology: deals with the implementation of distributed systems in terms of real components such as hardware, operating systems and software packages.

In the ANSA computational model, all services are treated uniformly as objects. Objects are accessed through their interfaces. An interface is defined as a set of named operations. Services are made available for access by exporting an interface to a trader which acts as a database of available services. Each entry in the trader describes an interface in terms of an abstract data type signature for the object and a set of attribute values. This will be matched against the available offers in the trader and a suitable candidate will be selected. Finally, once an interface has been selected, the system binds the client to the appropriate implementation of the requested service and operations can then be invoked.

At the engineering level, the relevant concepts are objects as the units of structure of a system, capsule as the address domain, the nucleus as a resource manager, thread as an independent execution path and channel as a path between communicating capsules.

3.1.2 Multimedia computing

One of the ANSA deficiencies in dealing with multimedia computing is its inability to model continuous media such as voice or video. The use of such media implies the need for continuous data transfers over time. To solve this, two sorts of objects are added to the platform, namely, *devices* and *streams*. Devices are abstractions of physical devices or stored continuous media. They can represent either sources or sinks of multimedia data. Streams are abstractions over transport protocols. A stream represents a unidirectional connection between two devices.

Another requirement in multimedia systems is the support of synchronization mechanisms. It is necessary synchronization between the sender and the receiver of continuous media and also across multiple continuous media streams [6].

3.2 Extensions to ANSA for transaction management

For achieving flexible management, it is necessary to add two extensions to the ANSA model. Firstly, the introduction of *generic factories* allows the creation of objects with a range of characteristics concerning concurrency control and recoverability. Secondly, it is necessary to manage these objects in a more application-oriented manner. For this, a *new approach to transparency* is proposed. These two aspects of flexible management are described below.

3.2.1 Generic factories

The ANSA architecture features the concept of factories as a means of creating objects in the distributed environment. A factory creates an object from a template where a template is the code implementing the object behaviour. The creation operation creates and activates a new instance of an object choosing one of the several templates available from a database (figure 1).

The notion of factory taken from the ANSA model must be extended to include parameters for the creation operation. These parameters express policies of services such as concurrency control and recovery. The policy will determine the mechanism to be linked to the created object. Thus objects with different characteristics can be created by the same factory. For instance, the same factory can create an object with optimistic concurrency control and another with pessimistic concurrency control depending on the specified policy. The extended notion is called a *generic factory*.

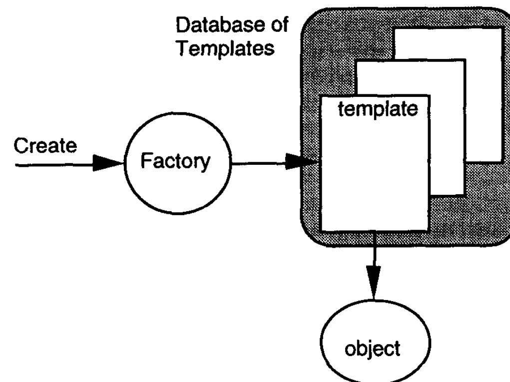

Figure 1 - Creation of object instances

3.2.2 Flexible approach to transparency

The ANSA project has refined the traditional notion of transparency by breaking it down into a number of constituent parts:

. access: the difference between local and remote services are hidden from the application.

. location: an application can access an object without knowing its physical location.

. migration: a service can move its location while being used by an application without its knowledge.

. replication: the application is not aware if a service is replicated.

. concurrency: the effects caused by on-going concurrent applications using a service can not be observed by another application using the same service.

. failure: the effects of partially completed interactions that failed are hidden from users.

ANSA allows programmers to choose the required transparency for each aspect in an independent way. This is called selective transparency.

Although providing some flexibility, selective transparency is not enough for cooperative environments in which applications have different requirements. For this environment, it is necessary to increase the user's control on distributed management. Selective transparency can be extended by allowing a management policy to be specified for each distributed aspect. In the proposed model, it is not only possible to mask out the concurrency and failure aspects but also to determine the style of management according to the application preferences. For this, policies can be specified for a given object at its creation time and for a group of objects within the scope of a transaction. Further discussion on this topic is found below.

3.3 The transaction services

In the proposed model, transaction management caters for concurrency control and recovery. The concurrency control and recovery services are embedded into each object and they depend on the policies specified by the application at the object creation time. Transaction support is based on two types of objects: *reliable objects* and *transaction managers* to be described in 3.3.1 and 3.3.2 respectively.

3.3.1 Reliable objects

Reliable objects are complex objects composed of a concurrency control component and a storage component in addition to the usual functional component (figure 2).

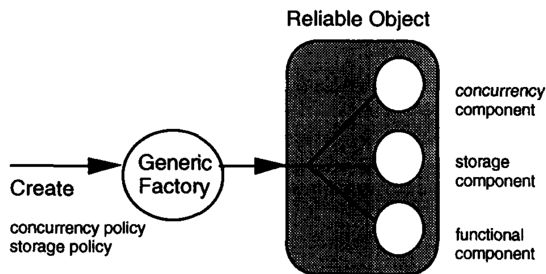

Figure 2 - Creation of reliable objects

Reliable objects are created by generic factories. The parameters of the creation operation express policies for the objects. Policies must be specified for the following services: concurrency control and storage. There are two templates for the concurrency control service implementing the locking mechanism and the certification mechanism. The templates for the storage service implement update-in-place, logging and versioning mechanisms.

A factory creating an object is responsible for mapping the service policy on to the appropriate template implementing the service as there is a range of templates implementing a given service. Thus the policy will determine the object component implementing the corresponding service. For example, a pessimistic concurrency policy will map on to a locking scheme while an optimistic policy will map on to a certification scheme. The supported services are described below.

(a) concurrency control service

The concurrency control service is responsible for controlling interleavings of operations according to the compatibility rules specified by the implementor of a given type. Differently from other systems, the concurrency control mechanism is not fixed. Instead, one of a range of mechanisms can be chosen according to the specified policy.

The policies for concurrency control are divided into two main categories: pessimistic and optimistic. Optimistic strategies work better when the expected number of conflicts is low otherwise pessimistic approaches are more suitable. The complete list of policies is the following:

. no concurrency control: no conflict check is performed applications are always allowed to proceed.

. pessimistic: conflict checking is made prior to each access to an object. In case of conflict, the application is blocked. Otherwise the object can be accessed.

. optimistic: conflict checking is performed at the end of the transaction. In case of conflict, the application is aborted. Otherwise it is allowed to complete with updates becoming effective.

. pessimistic with notification: conflict checking is made prior to each access but the requester never waits. In case of conflict, a notification is sent to the other applications that also accessed the object in a conflicting mode.

. optimistic with notification: conflict checking is made at the end but no applications are aborted. Instead, the application is allowed to finish and a notification is sent to the other applications that accessed the object in a conflicting mode.

Pessimistic policies map on to a locking mechanism [5] while optimistic policies map on to schemes based on certification [5].

(b) storage service

The storage service is responsible for updating objects in secondary memory. It provides a range of facilities for safe update and recovery from failures. It is up to the application to choose one of the supported storage mechanisms.

The policies for the storage service are divided into two main categories: pessimistic and optimistic. The optimistic schemes assume that failures will not happen often. Thus updates are made in place after the old state of the object is saved in a separate space. In this case, there is more overhead for failing transactions than succeeding ones, as a failed transaction requires old states to be recovered. The pessimistic approach, on the other hand, postpones changes to the object state until the transaction is committed. For failing transactions, the recovery process is simpler. The complete list of policies is the following:

. no recoverability: objects are updated in place. There is no provision for failure recovery.

. pessimistic without versions: objects are never updated in place. Instead changes to the object state are made in a separate area and added just at transaction commit time.

. pessimistic with versions: objects are never updated in place. Updates requested by an application cause a new version of the object to be created. If the transaction commits, this new version is added to the object version history.

. optimistic: updates are made in place but the old object state must be saved before the object is modified. The old version is necessary for the recovery procedure.

The pessimistic policies map into versioning or log-intention mechanisms [5]. The optimistic policy maps into a log-undo recovery mechanism [5].

3.3.2 Transaction manager objects

Applications can choose between *transparent* or *non-transparent* transaction management. If transaction management is non-transparent, the application will have to control concurrency and recovery for the set of objects within its domain. The application interacts directly with the objects.

In case of transparency, a transaction manager will be responsible for controlling transaction services. The application interacts with the transaction manager. First, the application must create a transaction manager. The creation operation of a generic factory will allow management policies to be specified as parameters. Policies for concurrency control and storage services are meant to force constraints on the objects participating in the transaction.

The transaction policies for *concurrency control* and their compatible object policies are the following:

(a) non-serializable: compatible with any object policies.

(b) non-serializable with notification: compatible with *pessimistic with notification* or *optimistic with notification* policies.

(c) serializable: compatible with *pessimistic* or *optimistic* policies.

(d) optimistic: compatible with *optimistic* or *optimistic with notification* policies.

(e) pessimistic: compatible with *pessimistic* or *pessimistic with notification* policies.

The transaction policies for *storage* and their compatible object policies are the following:

(a) non-recoverable: compatible with any object policy.

(b) recoverable: compatible with *pessimistic* or *optimistic* policies.

(c) optimistic: compatible with *optimistic* policy only.

(d) pessimistic without versions: compatible with *pessimistic without versions* only.

(e) pessimistic with versions: compatible with *pessimistic with versions* only.

Having created a transaction manager, the application initiates a transaction issuing a *begin transaction*. Before accessing any object, it is necessary to execute a *join* operation to make the transaction manager aware of a new participating object. When a new object joins a transaction, a consistency check between object's attributes and application's specification is made and the application is allowed to proceed if there is no incompatibility. Otherwise the object is not allowed to join the transaction. Facilities for checkpointing and rolling back the transaction state are also provided. The application finishes the transaction by issuing either an *end transaction* operation for committing the transaction state or an *abort transaction* for discarding the transaction updates.

3.4 Additional features

In addition to dealing with concurrency control and recovery, the transaction service is also concerned with other relevant issue for cooperative systems, namely real-time and group work as discussed below.

(a) real-time: multimedia applications require some form of real-time support. The approach taken in the proposed model incorporates a reservation mechanism within the transaction facilities. An application can reserve a group of resources atomically. This guarantees that requested resources will be available without delay when they are actually required. For example, to guarantee the transmission of continuous media between two devices, the application must reserve these devices prior to the transmission.

(b) group work: group work requires mechanisms to make a user aware of other users' actions. In the proposed model, this is done through a notification mechanism provided by the concurrency control service. An application can register itself with the transaction manager in order to receive a notification of conflicting access. After the registration, if any object accessed by the application is later accessed by another application in a conflicting mode, a notification is sent to the first application. For example, two users editing a document in collaboration have to be aware when the document has been updated. For that, they both must registered

themselves with their respective transaction manager. If one user is editing and the other user requests access to the same document, a notification will be sent to the first user.

3.5 Example of use

In a software design environment, two designers are working on two separate modules A and B. Designer 1 while debugging module A finds a problem in a routine of module C. This same routine is also used by module B which is eventually debugged by designer 2.

To solve this problem using the model proposed in this article, modules A, B and C are created with the *pessimistic with notification* policy for concurrency control and a *pessimistic with versions* policy for recovery. Designer 1 starts debugging A, finds a problem with one of the routines in C, changes C and continues to debug A. While debugging B, designer 2 receives a notification that module C is being updated. This will require module B to be tested with C's new version when it is released.

With respect to versions, designer 1 creates new versions for the two modules s/he updates (A and C) while designer 2 works on module B creating a new version for it. The latter designer starts using module C's latest version (c_1) but when notification is received that another user is updating the module, the new released version (c_2) must be read and module B re-compiled with this new version of C. The designer is responsible for finding out when C's new version is released. See figure 3.

3.6 Model evaluation

Distributed applications impose several requirements on transaction management. These requirements are discussed below in the context of the proposed model.

3.6.1 Support for data consistency

Objects can be created with different concurrency control and recoverability properties. It is up to the application to decide which properties will be required to preserve consistency. For some applications, serializability and atomicity will be required. Others may accept non-serializability. In case serializability is too restrictive and non-serializability endangers consistency, the application can rely either on a notification scheme or versioning.

Note that objects with different properties may participate in the same transaction if their policies are compatible with the transaction policies. It is left to the application to define the appropriate policies to achieve the required consistency level.

3.6.2 Performance

The ability of expressing application semantics is a major factor in improving performance. The flexible transaction model allows applications to create objects with the required properties. In order to improve performance, an application may avoid the extra overhead of concurrency control or recovery (or both) for some of the objects it creates. For that, the application should specify the *no concurrency control* policy for the concurrency control service or the *no recoverability* policy for the storage service. Alternatively, in case concurrency control properties are incorporated into the object, greater concurrency can be achieved through multiple versions or type-specific specification of operation compatibility. Moreover, versioning can be used to increase concurrency and security in collaborative work.

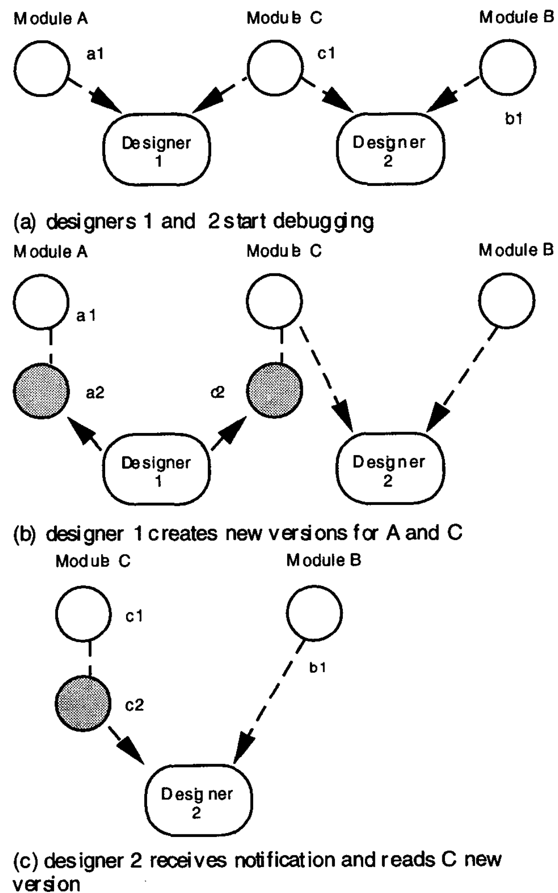

Figure 3 - A software design example

3.6.3 Support for long transactions

In the proposed approach, it is possible to create objects with non-serializable behaviour. This avoids blocking when requesting a lock in a pessimistic concurrency control scheme or rollbacks in an

optimistic method. Non-serializable behaviour may however allow two transactions to update the same object. To avoid this type of interference, an object can be defined with versioning properties. In that way, inconsistency and long waits can be avoided without requiring expensive mechanisms such as cascading aborts [20,21].

The process of recovery can be improved with checkpoint mechanisms. After a failure, the effects of an interrupted transaction can be recovered until the last checkpoint. Other models [23] use boundaries of subtransactions as checkpoints.

3.6.4 Support for group work

There are three aspects related to group work support: information sharing, mechanisms to improve visibility and group modelling.

Information Sharing In the proposed model, the collaboration is actually achieved through common objects with special properties. In particular, the concurrency control properties of shared objects must be specified to allow non-serializable behaviour and thus not to restrict interactions between users. This provides much of the flexibility required by groupware applications.

Visibility Mechanisms Cooperative work requires some mechanism to make users aware of each other's actions. In the proposed transaction model, this is done through a notification service. A user must register herself/himself with the associated transaction manager for receiving notification of a conflicting access. If an object within the application domain is accessed by another transaction in a conflicting mode, a notification will be sent to the registered user that non-serializability may occur. It is up to the notified user to take appropriate action. Notifications are also available in Observer [17] when objects are accessed in a conflicting way or conflicting requests are made by other clients. Gordion [12] and Quilt [15] maintain, for each object, a log of changes which can be read by cooperating users. Real-time conferencing systems support a shared space with synchronous update [13,22].

Group Modelling In the proposed transaction model, users, although working in collaboration, issue independent transactions. There is no support for group modelling as found in Skarra's transaction groups [24] or Korth's cooperative transactions [18] in which group structure is reflected into the hierarchy of nesting transactions. A parent transaction manages all the enclosed designers' subtransactions to make collaboration possible. In [24], synchronization specification at each level of the transaction hierarchy defines the allowable schedules for the members of a group. In [18], concurrency control mechanisms are designed to preserve serializability across projects but not between designers in the same project. In the model described in this article, once an object is created with non-serializable behaviour, it can be accessed by any user whether the user belongs to the group or not. Group transactions would be a valuable extension to the model.

3.6.5 Support for real-time

In the proposed model, two strategies can be used to tackle the problem of real-time constraints. The first avoids application delays by defining an object with non-serializable behaviour. The other approach is based on a pre-allocation scheme. Other systems use semantics to avoid application delays [13] or timing constraint specifications [7].

3.6.6 Support for a variety of applications

The proposed model is flexible enough to support a variety of applications. Its approach differs from traditional approaches in which transaction management is fixed providing the same concurrency control and recovery mechanisms for applications.

In order to achieve flexible transaction management for an environment in which applications have very different requirements, policies are separated from mechanisms and policy decisions are delegated to applications. While applications are responsible for specifying concurrency control and recovery policies, the support environment must map a policy on to an appropriate mechanism within the range of implementations provided for a given service. These policies are specified when an object is created with concurrency control and recovery properties being incorporated into the object itself. The application can, therefore, create the objects in its domain with appropriate properties to match its characteristics. For example, within a conference, a video image could be associated with a stream connected to a camera and a video window in a workstation. The stream, the source and sink devices must have concurrency control properties as they must be pre-allocated to guarantee real-time transmission but they do not require recoverability as the image will be discarded at the end of the conference.

4. Conclusions

This article has presented several transaction models classified according to the application areas they have been developed for. However, it is argued that a novel approach is necessary to deal with the great variety of applications in the distributed environment. Such environment still requires some form of concurrency control and recovery from failures although traditional mechanisms are too restrictive and transaction properties such as atomicity, serializability and isolation may not be required for some application types.

The model proposed in this article provides the flexibility required in such environment. This is achieved by increasing user's control over transaction management. The model is concerned with three main

issues: the variety of applications in the distributed environment, group work and real-time support for multimedia applications.

An application requiring reliability can be constructed from reliable objects which incorporate concurrency control and recovery properties. These properties are determined by policy specification at object creation time. In addition, an application can specify policies for its associated transaction manager. This transaction manager will be responsible for enforcing the specified policies on the objects joining the transaction. This approach allows applications to tailor transaction management according to their specific requirements.

Although much research is still necessary to find out more effective mechanisms for distributed cooperative applications, the authors believe that the proposed model is another step in providing more flexible transaction management in the distributed environment.

References

[1] Allchin, J. E., McKendry, M. S., "Synchronization and Recovery of Actions", Proc. 2nd ACM Symp. on Principles of Distributed Computing, August, 1983.

[2] ANSA Technical Report, ANSA Reference Manual, Release 01.00, 1989.

[3] ISA Project, Access-specific concurrency control in distributed object systems, APM/TR.010.00, 1990.

[4] Barghouti, N. S., Kaiser, G. E, "Concurrency Control in Advanced Database Applications", ACM Computing Surveys, vol. 23, n. 3, September, 1991.

[5] Bernstein, P. A., Hadzilacos, V., Goodman, N., "Concurrency Control and Recovery in Database Systems", Addison-Wesley, 1987.

[6] Blair, G. S., Coulson, G., "Meeting the Real-Time Synchronization Requirements of Multimedia in Open Distributed Systems", Technical Report, !993.

[7] Cheng, S. C., Stankovic, J. A., Ramamritham, K., "Scheduling Algorithms for Hard Real-Time Systems: A Brief Survey", Hard Real-Time Systems, Stankovic, J. A., Ramamritham, K., eds., Computer Society Press of the IEEE, 1988.

[8] Coulson, G., Blair, G. S., Davies, H., Williams, N., "Extensions to ANSA for Multimedia Computing", to appear in Computer Networks and ISDN Systems, 1992.

[9] Decouchant, D., Krakowiak, S., Meysembourg, M., Riveill, M., Rousset de Pina, X., "A synchronization mechanism for typed objects in a distributed system", SIGPLAN Notices, vol.24, n. 4, April, 1988.

[10] Detlefs, D. L., Herlihy, M. P., Wing, J. M., "Inheritance of Synchronization and Recovery Properties in Avalon/C++", IEEE Computer, vol. 21, n. 12, 57-69, December, 1988.

[11] Dixon, G. N., Shrivastava, S. K., "Exploiting Type Inheritance Facilities to Implement Recoverability in Object Based Systems", IEEE, 107-114, 1987.

[12] Ege, A., Ellis, C. A., "Design and Implementation of GORDION, an Object Base Management System", Proc. 3rd International Conference on Data Engineering, February, 1987.

[13] Ellis, C. A., Gibbs, S. J., "Concurrency Control in Groupware Systems", Proceedings of the ACM SIGMOD Conference on Management of Data, 399-407, May, 1989.

[14] Eswaran, K. P., Gray, J. N., Lorie, R. A., Traiger, I. L., "The Notion of Consistency and Predicate Locks in a Database System", Communication of the ACM, vol. 19, n. 11, 624-633, November, 1976.

[15] Fish, R. S., Kraut, R. E., LeLand, M. D. P., "Quilt: a collaborative tool for cooperative writing", Proc. of the Conference on Office Information Systems, 30-37, 1988.

[16] Greif, I., Sarin, S., "Data Sharing in Group Work", ACM Transactions on Office Information Systems, vol. 5, n. 2, 187-211, April, 1987.

[17] Hornick, M. F., Zdonik, S. B., "A Shared, Segmented Memory System for an Object-Oriented Database", ACM Transactions on Office Information Systems, vol. 5, n.1, 70-95, January, 1987.

[18] Korth, H. F., Kim, W., Bancilhon, F., "On Long-Duration CAD Transactions", Information Sciences 46, 73-107, 1988.

[19] Moss, E. B., "Nested Transactions and Reliable Distributed Computing", Proc. 2nd Symp. on Reliability in Distributed Software Systems, July, 33-39, 1982.

[20] Pu, C., Kaiser, G. E., "Split-Transactions for Open-Ended Activities", Proceedings of the 14th VLDB Conference, 26-37, 1988.

[21] Salem, K., Garcia-Molina, H., Alonso, R., "Altruistic Locking: A Strategy for Coping with Long Lived Transactions", Proceedings of the 2nd International Workshop on High Performance Transaction Systems, 175-199, September, 1987.

[22] Sarin, S., Greif, I., "Computer-Based Real-Time Conferencing Systems", IEEE Computer, 33-45, October, 1985.

[23] Shrivastava, S. K., Wheater, S. M., "Implementing Fault-Tolerant Distributed Applications Using Objects and Multi-Coloured Actions", Technical Report Series, n. 315, June, 1990.

[24] Skarra, A. H., Zdonik, S. B., "Concurrency Control and Object-Oriented Databases", in Kim, W., Lochovsky, F. H., eds., Object-Oriented Concepts, Databases, and Applications, ACM Press, New York, 395-421, 1989.

[25] Toledo, M. B. F., "A Flexible Approach to Transaction Management in a Distributed Cooperative System", PhD thesis, Department of Computing, Lancaster University, 1992..

[26] Weihl, W., Liskov, B., "Implementation of Resilient, Atomic Data Types", ACM Trans. Prog. Lang. and Systems, vol. 7, n. 2, 244-269, April, 1985.