Exception Handling in Transactional Object Groups

M. Patiño-Martínez[†], R. Jiménez-Peris[†] and S. Arévalo[‡]

[†] Universidad Politécnica de Madrid
Facultad de Informática, 28660 Boadilla del Monte (Madrid), Spain
{mpatino,rjimenez}@fi.upm.es
[‡] Universidad Rey Juan Carlos
Escuela de Ciencias Experimentales, 28933 Móstoles (Madrid), Spain
s.arevalo@escet.urjc.es

Abstract

Transactions and group communication are two fault-tolerance techniques that have been evolved separately for a long time. However, these two techniques are complementary and distributed systems can benefit from their integration. In particular, transactional object groups can be used in clustered systems to provide high availability and high performance transactional services. Replicated object groups tolerate failures without interrupting running services increasing their availability. Cooperative object groups perform services in parallel taking advantage of distribution and thus, improving their performance. Exceptions play an important role in this kind of systems. Firstly, exceptions can be used in combination with transactions to cope with anticipated errors within a transaction. Transactions act as firewalls providing automatic backward recovery for unhandled exceptions. Secondly, object groups can raise concurrent exceptions due to their concurrent and distributed nature. Exception resolution is used to yield a single exception to the client. We propose an exception resolution algorithm that resolves exceptions at two levels: local and distributed. Local resolution is first applied to concurrent exceptions raised within an object. Then, distributed resolution is applied to resolve the resulting exceptions of the object group. This two-level hierarchical resolution is more rational than a global resolution, as local exceptions, that are more related among themselves, are resolved first.

1 Introduction

There are several systems providing group communication [BR93, HT93] primitives for distributed objects, namely, Electra [Maf95], Eternal [MMSN98], object group service (OGS) [GFGM98] and Newtop object group service (NOGS) [MSEL99]. These systems are usually referred as object groups [GFGM98] and all of them provide CORBA interfaces. Replicated and cooperative object groups can be built based on these communication primitives. Replicated object groups are used to provide high available services running each object replica on a different site. Therefore, site crashes are tolerated. Cooperative ones are used to increase the performance of their services by distributing and running in parallel a service among the objects of a group.

Transactions [GR93] provide data consistency in the presence of failures and concurrent accesses. Transactions are not only used in database systems, they are the key to build reliable distributed applications that might access data across different systems. Transactions are incorporated in general purpose programming languages or libraries, such as Avalon [EMS91] and Arjuna [Shr95]. The importance of transactions has also been recognized in the CORBA standard. It includes as one of its services the object transactional service (OTS).

Transactions and group communication systems have been used separately. However, these techniques are complementary and many applications can take advantage from their integration. In this integration, object group methods are executed as transactions, which can be replicated or cooperative. In the former case availability is increased, while in the latter performance is increased. As methods are executed as transactions, object consistency is guaranteed in the presence of failures and concurrency. This approach has been taken in TransLib [JPAB00] an object oriented library providing transactional group services. Exception handling plays an important role in this system. More concretely, exceptions have been integrated in the context of transactions acting on groups of objects providing forward (exception handling) and backward (transactions) recovery to guarantee the consistency of objects. This paper focuses in this topic.

This paper concentrates on two main issues. First, it is discussed how to integrate forward and backward recovery provided by exceptions and transactions, respectively. And second, it is shown how to deal with concurrent exceptions that can be raised by transactional object groups. We discuss these two issues at a conceptual level in the next section. Section 3 proposes linguistic support for our exception model. Section 4 discusses how this exception model can be implemented. Finally, we compare our proposal with related work and present our conclusions.

2 Exceptions in Transactional Object Groups

Transactions [GR93] have been proposed as a fault-tolerance mechanism that guarantees the consistency of data in the presence of failures and concurrent accesses. A transaction either finishes successfully (commits) or fails (aborts). Transactions provide the so-called *ACID* properties: *Atomicity* ensures that the effect of a transaction is all-or-nothing. A transaction provides *consistency* by transforming a consistent state into a new consistent one; *isolation* ensures that the final effect of concurrent transactions is equivalent to execute them sequentially in some order; and *durability* guarantees that the effect of a committed transaction will remain. If a transaction aborts the atomicity property provides backward recovery as the state is restored to a *previous* (consistent) one.

Transactions can be nested [Mos85]. Nested transactions or subtransactions can be executed concurrently, but isolated from each other. They also allow confining failures within a transaction. If a subtransaction aborts only the work of that subtransaction is undone, the parent transaction does not abort. However, if a transaction aborts all its subtransactions will be aborted.

Multithreaded transactions [PJA98] are more powerful than traditional nested transactions as transactions may have several threads. Any transaction thread may spawn additional local threads. These threads are part of the same transaction and do not conflict among them, as subtransactions do. This allows to take advantage of multiprogramming and multiprocessor facilities.

Exceptions are used to signal an abnormal situation. Exception handling tries to produce from the current state a *new* consistent one (forward recovery). We propose to use exceptions to attain forward recovery within transactions. Thus, a new consistent state within the transaction can be obtained in the advent of foreseen errors. Unfortunately, not every exception (error) can be foreseen nor every exception can be handled. In our proposal, transactions act as firewalls for unhandled exceptions applying automatically backward recovery (transaction abort) when an unhandled exception is propagated outside the transaction boundary. Furthermore, this exception is used as a notification of the transaction abort to the enclosing scope.

Transactional object groups provide atomic services, that is, they guarantee transaction properties. An object group is a set of objects with a common interface (methods). Objects of a group can be distributed among a set of sites. Methods are called within a transaction using reliable multicast [HT93]. So, all available objects of a group will receive the calls. A method invocation is executed as a subtransaction that is run by all objects of the group. Each object creates a new thread to execute a method. Hence, one of these subtransactions has as many distributed threads as objects has the group.

Two kinds of transactional objects groups are provided: replicated and cooperative ones. Replicated object groups provide hardware fault-tolerance. If each object of a group is at a different site, it can tolerate up to $k-1$ failures, being k the number of object groups. Objects of a replicated group behave as an state machine [Sch90]. They have the same state and code and they behave as a single object in front of clients. In order to preserve the consistency of replicated objects, no concurrency is allowed within their methods. Cooperative object groups can be used to decrease transaction latency by executing a method in parallel by all the group objects. For instance, each object can represent a set of accounts of a bank. The group can provide an operation to pay the interest to each account at the end of the month. That operation will be performed in parallel by all the group objects. Hence, the state of the objects of a cooperative group can be different, as in the previous example. Even the code can be different. Objects of a cooperative group know each other and they can communicate among them. They can also spawn local threads to execute concurrently an invocation. Thus, two levels of concurrency can be used to execute a method invocation, the inherent parallelism provided by the object group distribution and the local threads of each object of the group.

Concurrent exceptions can happen both in a local transaction (a transaction that only has local threads) or a distributed one (an object group method invocation). Hence, exception resolution is needed. As all the objects of a replicated group have the same code and state, and no threads can be created within a method, they will raise the same exception in a given method invocation. So, no exception resolution is done and the raised exception will be propagated to the client. However, if the group is a cooperative one, each object can raise a different exception when a method is run. What it is more, as each object can create threads during the execution of a method, concurrent exceptions can be raised within an object. As the nature of the exceptions raised at other objects might be quite different, we have found more adequate to apply exception resolution in two stages instead of a single one, as it is usual. Concurrent exceptions raised within a method of an object (local exceptions) are more related among them that with other exceptions raised at different (distributed exceptions) objects. For this reason, we propose to apply a two-level exception resolution. The first level is a *local resolution* performed among the threads of a method in a given object. This resolution can be different for each distributed object of a group. As a result of this resolution, each object will yield at most one exception. The second level, *distributed resolution*, will be applied to the resulting exceptions of the local resolution, thus obtaining a single exception. Thus, these two levels yield a *hierarchical exception resolution*.

3 Linguistic Support

In this section we propose linguistic support for the exception model presented above. The linguistic support is presented as an extension of Ada95 [BW95], but it can be applied to any other object oriented concurrent/distributed language/system.

Ada 95 is a language that provides both, concurrency and exception handling. Ada exception model is based on the termination model, that is, raising an exception causes the termination of the scope where the exception is raised. Scopes in Ada are subprograms, task bodies, accept statements, block statements, ... Unhandled exceptions are propagated from one scope to the enclosing scope until they are handled successfully or they reach the outermost scope, that can be the main program or a task. An unhandled exception in the main program causes its termination with a run-time error, whilst in a task, it causes the task termination.

Our proposal consists of introducing two new constructs: the *transactional block*, and *transactional object groups*. A transactional block allows to initiate a transactional scope that will guarantee the *ACID* properties. Transactional blocks can be nested yielding to nested transactions. Data items declared within a transactional block are atomic and can also be persistent. Ada tasks are used to create local threads. If tasks are declared within a transaction, they will be local threads of the associated transaction. Local threads can access the same data without conflicting and can cooperate to perform the transaction goal. A transactional block cannot terminate until its threads terminate, as happens with the regular block statement. A transaction will commit if all its threads finish successfully. Otherwise, if at least one of them finishes exceptionally the transaction will be aborted.

Transactional object groups are provided extending the distributed systems annex of Ada introducing a new kind of partition corresponding to an object group. There is a peculiarity about cooperative object groups that is that there is a single specification, but there might be multiple implementations (up to one per object of the group) [PJA98, MÁAG96].

Exception resolution clauses are provided to associate resolution functions for concurrent exceptions either to a transactional block or to a group object method. These functions take two exceptions as arguments and return the resulting exception[1]. If n local (distributed) exceptions are concurrently raised within a transaction, the local (distributed) resolution function will be called $n - 1$ times by the runtime system to obtain the final exception.

The above linguistic mechanisms have been included in an Ada 95 extension, *Transactional Drago* [PJA98]. The linguistic support discussed here can be easily applied to any programming language that provides objects, concurrency/distribution and exceptions as Ada 95, for instance, Java.

4 Implementation

The main implementation issue is how to implement exception handling and resolution within transactions. Termination of single-threaded transactions is trivially determined, as a transaction finishes when its code executes the last statement of a transaction; it finishes successfully or not depending on whether it executes a commit or an abort statement. However, termination of a multithreaded transaction is not that easy. First, it is necessary to find out *when* it terminates. And second, it must be determined *how* it terminates, that is, whether it commits or aborts, and in case of an abort which exception is propagated as abort notification. In order to achieve this task, we have combined three different algorithms: commit protocol, abort protocol, and exception resolution algorithm into one. We have called it *hierarchical termination algorithm*.

This algorithm is performed in two stages corresponding to the two levels of exception resolution. The execution of a method in a object group will finish when all its local threads have finished, that is, its termination is synchronized with the one of its local threads. It finishes successfully if all its local threads do so. If one or more threads (including itself) finish exceptionally, the distributed thread will also finish exceptionally. In the advent of concurrent exceptions the returned exception is obtained by applying the local resolution function. The distributed transaction termination is delayed until all its distributed threads (one per group object) terminate. A transaction will commit, if all of the group objects finish successfully. Otherwise, it will abort. If more than one object raises an exception, the distributed resolution function will be applied to obtain a single exception. This exception will be propagated to the transaction enclosing scope to notify its abort.

5 Related Work

There have been a few attempts to integrate transactions and exceptions. One of first ones was Argus [Lis88], a distributed transactional programming language. Their approach is an orthogonal integration where transactions can commit or abort terminating normally or exceptionally. In our opinion this implies some dangers. In particular, committing a transaction terminating exceptionally is quite dangerous as a result the state can be left inconsistent, and in the best case it can result misleading.

A more recent proposal appears in the CORBA object transaction service. In [SBR98] some of the problems of this integration are discussed. The main one is that the client is in charge of handling a server raised exception in order to abort the transaction on the client site. We think that this must be done automatically as part of the transaction semantics to prevent inconsistencies as we suggest in our proposal.

[Rom96] presents an approach for integrating coordinated atomic actions and exceptions. This work deals with a different context where processes join explicitly on-going transactions at different moments to cooperate within them. In our approach, all the participants in a distributed transaction (group objects) start a transaction implicitly when an object group method is invoked. This

[1] In fact, as exceptions cannot be passed as arguments in Ada 95, exception identities are used for this purpose.

fact makes the context of transactional object groups less complex than the one of coordinated atomic actions, thus simplifying the algorithm for exception resolution.

6 Conclusions

A new use of exceptions has been presented in the context of transactional object groups. This approach is novel in that backward and forward recovery provided by transactions and exceptions have been integrated. This allows to use forward recovery within a transaction. When an exception is raised, the exception handler will try to obtain a new consistent state. If this is not possible, the previous consistent state (the one before the transaction begun) is recovered. Thus, data consistency is guaranteed in any case.

The second contribution is a hierarchical exception resolution for multithreaded transactions. This resolution is more rational than a global resolution, as it resolves first local exceptions which are more related and later distributed exceptions. Additionally, it has been proposed how to integrate the commit and abort protocols of transactions with exception resolution in a single algorithm. Thus, no additional cost is paid for the exception resolution.

Finally, we believe that transactional object groups will play an important role in simplifying the programming of fault-tolerant distributed systems, hence a clear semantics for exceptions should be provided for this kind of systems.

References

[BR93] K. P. Birman and R. Van Renesse. *Reliable Distributed Computing with Isis Toolkit.* IEEE Computer Society Press, Los Alamitos, CA, 1993.

[BW95] A. Burns and A. Wellings. *Concurrency in Ada.* Cambridge University Press, 1995.

[EMS91] J. L. Eppinger, L. B. Mummert, and A. Z. Spector, editors. *Camelot and Avalon: A Distributed Transaction Facility.* Morgan Kaufmann Publishers, San Mateo, CA, 1991.

[GFGM98] R. Guerraoui, P. Felber, B. Garbinato, and K. Mazouni. System Support for Object Groups. In *Proc. of the ACM Conference on Object Oriented Programming Systems, Languages and Applications*, Vancouver, Canada, October 1998. ACM Press.

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

[HT93] V. Hadzilacos and S. Toueg. Fault-Tolerant Broadcasts and Related Problems. In S. Mullender, editor, *Distributed Systems*, pages 97–145. Addison Wesley, Reading, MA, 1993.

[JPAB00] R. Jiménez Peris, M. Patiño Martínez, S. Arévalo, and F.J. Ballesteros. TransLib: An Ada 95 Object Oriented Framework for Building Dependable Applications. *Int. Journal of Computer Systems: Science & Engineering*, 15(1):113–125, January 2000.

[Lis88] B. Liskov. Distributed Programming in Argus. *Communications of the ACM*, 31(3):300–312, March 1988.

[MÁAG96] J. Miranda, Á. Álvarez, S. Arévalo, and F. Guerra. Drago: An Ada Extension to Program Fault-tolerant Distributed Applications. In A. Strohmeier, editor, *Proc. of Int. Conf. on Reliable Software Technologies*, volume LNCS 1088, pages 235–246, Montreaux, Switzerland, June 1996. Springer.

[Maf95] S. Maffeis. Adding Group Communication and Fault-Tolerance to CORBA. In *Proc. of 1995 USENIX Conf. on Object-Oriented Technologies*, June 1995.

[MMSN98] L.E. Moser, P.M. Melliar-Smith, and P. Narasimhan. Consistent Object Replication in the Eternal System. *Theory and Practice of Object Systems*, 4(2):81–92, 1998.

[Mos85] J. E. B. Moss. *Nested Transactions: An Approach to Reliable Distributed Computing.* MIT Press, Cambridge, MA, 1985.

[MSEL99] G. Morgan, S.K. Shrivastava, P.D. Ezhilchelvan, and M.C. Little. Design and Implementation of a Corba Fault-Tolerant Object Group Service. In *Proc. of the 2nd Conf. on Distributed Applications adn Interoperable Systems*, June 1999.

[PJA98] M. Patiño Martínez, R. Jiménez Peris, and S. Arévalo. Synchronizing Group Transactions with Rendezvous in a Distributed Ada Environment. In *Proc. of ACM Symp. on Applied Computing*, pages 2–9, Atlanta, Georgia, February 1998. ACM Press.

[Rom96] A. Romanovsky. Atomic Actions Based on Distributed/Concurrent Exception Resolution. Technical Report 560, University of Newcastle upon Tyne, November 1996.

[SBR98] R. Stewart, S. Bhattacharya, and D. Rai. Distributed Exception Handling in CORBA-Based C++ Applications. *C++ Report*, pages 33–43, March 1998.

[Sch90] F. B. Schneider. Implementing Fault-Tolerant Services Using the State Machine Approach: A Tutorial. *ACM Computing Surveys*, 22(4):299–319, 1990.

[Shr95] S. K. Shrivastava. Lessons Learned from Building and Using the Arjuna Distributed Programming System. In K.P. Birman, F. Mattern, and A. Schiper, editors, *Theory and Practice in Distributed Systems*, volume LNCS 938, pages 17–32. Springer, 1995.