Exception Handling and Resolution for Transactional Object Groups*

Marta Patiño-Martínez[1], Ricardo Jiménez-Peris[1] and Sergio Arévalo[2]

[1] School of Computer Science
Technical University of Madrid (UPM)
28660 Boadilla del Monte, Madrid, Spain
{mpatino,rjimenez}@fi.upm.es
[2] Escuela de Ciencias Experimentales
Rey Juan Carlos University
28933 Móstoles, Madrid, Spain
s.arevalo@escet.urjc.es

Abstract. With the advent of new distributed applications like on-line auctions and e-commerce, the reliability requirements are becoming tighter and tighter. These applications require a combination of data consistency, robustness, high availability and performance. However, there is no single mechanism providing these features. Data consistency is preserved using transactions. Robustness can be obtained by foreseeing and handling exceptions. Objects groups can help in increasing the availability and performance of an application. In order to attain the growing demand of higher levels of reliability it is necessary to integrate these mechanisms with a consistent semantics. This article addresses this topic and studies the role of exceptions in this context.

1 Introduction

With the increasing importance of new distributed applications such as on-line auctions and e-commerce, stronger reliability guarantees are required. These applications need availability, data consistency and high throughput. Traditional reliability techniques by themselves, like transactions or group communication, only provide a subset of these properties. With the integration of these techniques is possible to satisfy the surging need for higher levels of reliability.

Group communication [2, 10] is one of the basic building blocks to build reliable distributed systems. Although, group communication primitives were proposed in the context of groups of processes, they have been integrated with the object oriented paradigm [15, 17, 8, 16] resulting in what has been named object groups.

A *group of objects* is a set of distributed objects that share the same interface and behave as a single logical object. Clients interact with object groups as with

* This work was partially supported by the Spanish Research Council, CICYT, under grant TIC94-C02-01 and the Madrid Regional Research Council (*CAM*), contract number CAM-07T/0012/1998.

regular objects. Transparently to the client, invocations are multicast to all group members. Object groups have traditionally been used to increase either system availability or performance. If all the group objects are exact replicas, object failures can be masked. On the other hand, distributing a method execution among the group objects can increase performance.

Transactions [6] provide data consistency in the presence of failures and concurrent accesses. Transaction properties have become crucial for building reliable applications. Their use has spreaded from databases to a more general setting, namely distributed systems. The importance of transactions has been recognized in the CORBA object transactional service (OTS) [19], Java transaction service (JTS) [26], and Enterprise Java Beans [25] standards. But, also several general purpose programming languages and libraries have incorporated them, such as Avalon [4] and Arjuna [24].

Our research has been motivated by the need to provide a consistent integration of these mechanisms. In our proposal, clients can enclose a set of group invocations within a transaction to preserve their atomicity. Object group methods can be executed as transactions. In this way, object consistency is guaranteed in the presence of failures and concurrent accesses.

In this context the semantics of exceptions need to be precisely defined. Exception handling plays a key role in our approach. First, the abort of a transaction is notified by means of an exception. Second, exceptions have been integrated in the context of transactions acting on groups of objects, providing forward (exception handling) and backward (transactions) recovery to guarantee data consistency. As the nature of object groups is concurrent several exceptions can be raised concurrently. Concurrent exception resolution is provided to notify the abortion of the corresponding transaction with a single and meaningful exception.

This article concentrates on two main issues. First, it is discussed how forward and backward recovery provided by exceptions and transactions, respectively, have been integrated. And second, it is shown how to deal with concurrent exceptions within the context of transactional groups.

The rest of the article is organized as follows. Next section describes the features of transactional object groups. Section 3 discusses exception handling in this framework. Section 4 proposes linguistic support for our exception model. Implementation issues of this exception model are presented in Section 5. Finally, we compare our proposal with related work and present our conclusions.

2 Transactional Object Groups

2.1 Object Groups

An object group can be considered a distributed implementation of a class. Members (objects) of an object group share the same interface (the one of the class) and can be located at different sites of a network. Method invocations are reliably multicast to all the group members. Reliable multicast messages are delivered to all the group members or none of them. This property helps to keep the consistency among group members, as all of them will process the same method invocations. Multicast is also virtually synchronous [1], that is, membership (view) changes are delivered at the same logical instant at all the members. This means that the members that transit from one view to the next one have processed the same set of method invocations before the view change. Therefore, the programming of reliable object groups is simplified.

We distinguish two kinds of object groups based on their functionality: replicated and cooperative object groups.

Replicated object groups (replicated groups to abbreviate) provide hardware fault-tolerance. They implement active replication. That is, all the objects process each method invocation. In a replicated group all the objects are exact replicas. They have the same state and deterministic code.

Objects of a replicated group behave as a state machine [23]. Method invocations are reliable multicast and also total ordered [10] to guarantee that behavior. Total order means that all the members of a group receive method invocations in the same order. All group members start from the same state and process the same method invocations in the same order. This feature together with the restriction of not allowing concurrency within methods ensures the determinism of replicated groups [12]. If each object of a group is placed at a different site, the group can tolerate up to $k-1$ site failures, being k the number of objects in the group. Therefore, the distribution and replication of objects is used to increase the availability of a logical object. For instance, let´s consider a name service that maps services to servers in a distributed system (e.g., CORBA). This service is critical in the sense that when it is not available, clients cannot contact the servers as they cannot find out their location, and the system blocks. Replicating the name server object prevents this situation, providing the required availability.

Replication is transparent both in front of clients and servers. Clients of a replicated group invoke group methods as if the object were non-replicated. Since all the group members have the same state and code, they will produce the same answers, therefore a single answer is returned to group clients (Fig. 1.b). Replicated groups can invoke other objects. That is, they can act as clients. When a replicated group invokes another object (replicated or not), duplicated requests must be avoided. In our approach only one method invocation takes place (Fig. 1.c) to preserve the single object behavior of the group. Answers are returned to all group members. To our knowledge only *GroupIO* [7], a group communication library, implements such a behavior.

On the other hand, in a *cooperative object group* (or cooperative group) distribution is used to increase the throughput of the system. The state of an object is distributed among the group members and thus, method invocations are executed in parallel, decreasing latency. Hence, the state of the objects of a cooperative group can be different. Even method implementation can be different. For instance, a cooperative group can represent a bank and each member can represent a branch of a bank. In this case, the state of the objects is different.

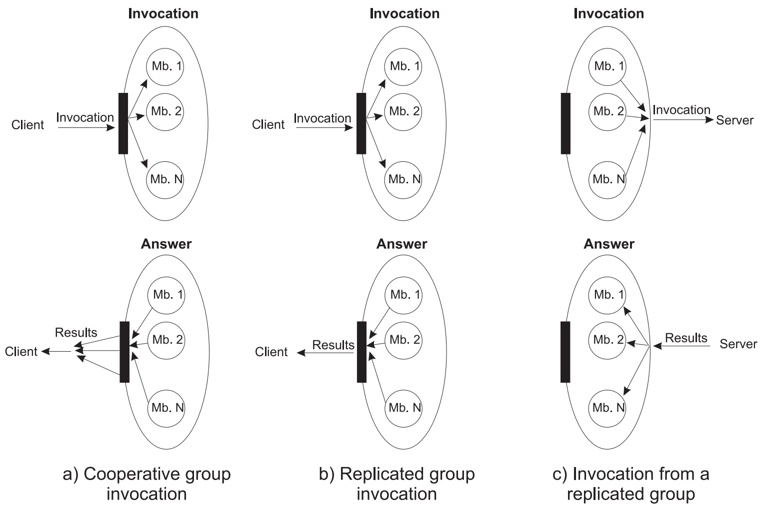

Fig. 1. Interaction with object groups

The group can provide an operation to pay the interest to each bank account at the end of the month. That operation will be performed in parallel by all the group objects.

During the execution of a method in a cooperative group, each object of the group can create new threads to execute concurrently that method. This feature allows taking advantage of the multiprocessing capabilities of the underlying system. Thus, two levels of concurrency can be used to execute a method invocation in a cooperative group, the inherent parallelism provided by object distribution and local multithreading at each object.

Cooperative groups also behave as a single logical object in front of clients and servers. Since the state and the code can be different, each object can return a different result. The final result is composed at the client site before delivering it to the client application (Fig. 1.a). For instance, a method computing the total balance of a set of accounts will compose the results adding all the object results.

Unlike replicated objects, objects of a cooperative group are aware of each other and they can communicate among them. For instance, a cooperative group can store the agendas of the staff of a company, where each group member holds a department agenda. An agenda contains information about the schedule of an employee. The group can provide a service to set meetings among members of several departments in a given period of time. When this service is invoked, group objects communicate among them to notify the availability of the members of their departments to find a common free slot to set the meeting.

2.2 Transactions

Transactions [6] are used to preserve data consistency in the presence of failures and concurrent accesses. A transaction either finishes successfully (commits) or fails (aborts). A transaction provides the so-called ACID properties. *Atomicity* ensures that a transaction is completely executed (it commits) or the result is as it were not executed (it aborts). If a transaction aborts, the atomicity property ensures that the state is restored to a *previous* (consistent) one. Hence, transaction atomicity provides backward recovery. *Isolation* or serializability guarantees that the result of concurrent transactions is equivalent to a serial execution of them. *Durability* ensures that the effect of committed transactions is not lost even in the advent of failures.

Transactions can be nested [18]. Nested transactions or subtransactions can be executed concurrently, but isolated from each other. They cannot communicate among them due to the isolation property. No concurrency is allowed in the traditional nested transaction model apart from concurrent subtransactions. If a subtransaction aborts only that subtransaction is undone, the parent transaction does not abort. Therefore, subtransactions also allow confining failures within a transaction. However, if a transaction aborts, all its subtransactions will abort to preserve the atomicity of the parent transaction. We propose a more general model, *group transactions* [20], where a transaction can have several concurrent threads, either local or distributed. Those threads can communicate among them and share data as they belong to the same transaction.

2.3 Transactional Object Group Services

If a client interacts with several groups and the atomicity of the whole interaction must be preserved, multicast by itself does not help. The reliability property of multicast is concerned with a single message (method invocation). To keep the atomicity of several group invocations, a *super-group* [22] can be created. This super-group contains all the groups the client will contact. However, this solution is very expensive. Creating groups dynamically takes some time and the groups' programming gets more complicated. Messages must be decomposed to know which part belongs to which group of the super-group. Additionally, this approach does not deal with recovery (needed in case of aborts or failures) nor with concurrency control (needed for concurrent clients). A simpler approach is to enclose the whole interaction within a transaction. The transaction automatically guarantees the atomicity property.

Transactional object groups provide atomic services. Clients must interact with transactional object groups within a transaction. Methods of a transactional group are executed as subtransactions, which are run by all the group objects. A subtransaction corresponding to a method invocation on an object group is a *distributed transaction* that has as many distributed threads as there are objects in the group.

Subtransactions on replicated groups are highly available. They survive site failures without aborting. A subtransaction (method invocation) in a replicated group will commit as far as there is an available member. This contrasts with the traditional approach where the failure of a single replica aborts all ongoing transactions [9].

When a transaction (method invocation) is executed in a replicated group, it just has a thread per object to enforce the determinism of the group. However, this restriction does not apply to cooperative groups. Each object of a cooperative group can create new threads on behalf of the (sub)transaction the object is running. The lifetime of those threads does not expand beyond the method execution. As the objects of a cooperative group work to achieve a common goal, it is required that all the group members finish successfully to commit a transaction. That is, a subtransaction in a cooperative group will commit, if all its threads finish successfully, otherwise it will abort.

3 Exceptions in Transactional Object Groups

3.1 Exceptions and Transaction Aborts

The operation domain is decomposed into *standard* and *exceptional domains* [3]. An operation invoked within its standard domain terminates successfully. On the other hand, an operation invoked within its exceptional domain leads to an exception *raising*, if the situation is detected. If the exception was foreseen, an *exception handler* can fix the situation and bring the system to a new consistent state (forward recovery), that is, the exception is *handled*. Exception handlers are attached to *exception contexts*, i.e., regions where exceptions are treated uniformly. Nested operation invocations yield to (dynamic) exception context nesting. An unhandled exception in an exception context causes its termination and it is *propagated* to the outer exception context.

We propose to use exceptions within transactions to attain forward recovery. In this way we integrate backward and forward recovery provided by transactions and exceptions, respectively. In the advent of foreseen errors a new consistent state within a transaction (Fig. 2) can be obtained (those that the transaction programmer has considered), preventing the transaction abort.

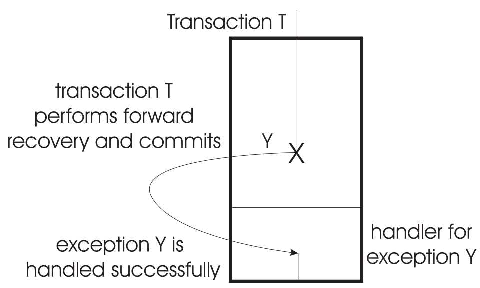

Fig. 2. Exception handling within a transaction

Unfortunately, every exception (error) cannot 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 (Fig. 3).

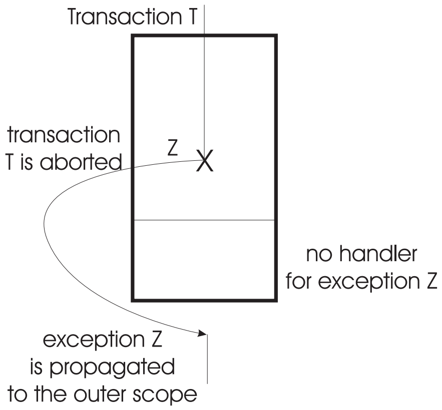

Fig. 3. Exception propagation outside a transaction

As exceptions are used to notify abnormal situations, any exception that is propagated outside the scope of a transaction causes its abort. If the transaction had been able to handle the exception internally, it would mean that forward recovery was successfully applied within the transaction. However, if the error could not be handled, backward recovery (undoing the transaction) is automatically performed. If the transaction commits, no exception is raised.

Fig. 4 shows how forward and backward recovery are combined. Subtransaction $T1.1$ raises an exception (Y). The exception is not handled in $T1.1$ and therefore, it is propagated to the enclosing scope ($T1$). As a consequence, $T1.1$ aborts (backward recovery). Thread $th0$ handles the exception (forward recovery) and transaction $T1$ continues.

We propose the use of exceptions to notify transaction aborts. Since transaction programmers can define their own exceptions, using exceptions to notify aborts provides more information about the cause of an abort than the traditional abort statement. This integration can be seen as the identification of transaction commit with the standard domain and transaction abort with the exceptional domain of a transaction.

3.2 Concurrent Exceptions

In our model, client transactions can be multithreaded to increase performance. Due to multithreading, two or more exceptions can be raised concurrently. In

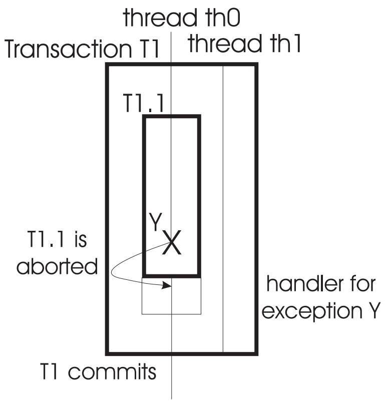

Fig. 4. Combined forward and backward recovery

this case, the transaction is aborted, as it happens with a single-threaded transaction, and an exception is propagated to notify the abort. However, when there are concurrent exceptions, it is necessary to perform exception resolution (*local resolution*) to choose a single exception to notify the transaction abort. This situation is depicted in Fig. 5.

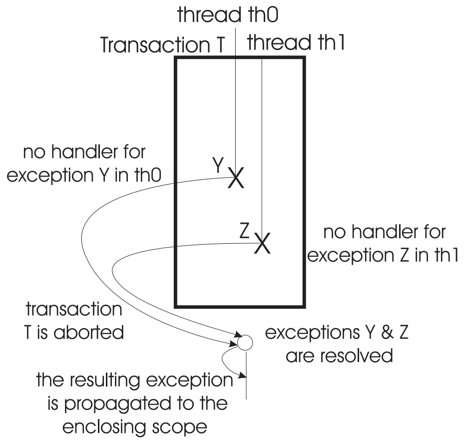

Fig. 5. Concurrent exceptions and exception resolution

A similar scenario can happen during the execution of a group method. An object group method is executed concurrently by all the group members. If multiple exceptions are raised, a single exception should be propagated to the outer scope, in this case, the scope where the method was invoked. Hence, a mechanism for *distributed exception resolution* is also needed. We call it distributed exception resolution, since it is performed among the distributed objects of a group. Two cases must be considered: resolution in replicated groups and in cooperative ones.

Replicated object groups behave deterministically. If a group object raises an exception, all of them should raise the same exception. However, there are some situations where the determinism of a replicated group is no longer respected. For instance, when group members are writing to a file and one of the members cannot write because of a local disk failure. If a member of a replicated group raises an exception that the rest of the members do not raise, it is considered faulty and removed from the group. Generalizing, a voting process is used for distributed exception resolution in replicated groups, and those members that are not in the majority are considered faulty and are removed from the group. If no majority it is obtained, the `abort_error` exception is raised provoking the transaction abort. Handling concurrent exceptions in this way avoids state divergence among the replicas.

In a cooperative group, each object can raise a different exception during the execution of a method. What it is more, as each object can create local threads during the execution of a method, concurrent exceptions can be raised even within a single object. Concurrent exceptions raised within an object (local exceptions) are more related among them that those exceptions raised at different objects (distributed exceptions). It is our opinion that it is more adequate to apply exception resolution in two stages instead of a single global one, as it is usual.

The first level is a *local exception resolution* performed among the threads of a method at a given object. This resolution can be different for each object of a group. As a result of this resolution, each object will yield at most one exception. If two or more objects of a group raise exceptions, distributed resolution is applied. This resolution constitutes the second level. Distributed exception resolution takes the exception raised by each group member (if any) and returns a single exception. Observe that each object will propagate at most one exception. If more than one exception is raised in an object, local exception resolution will return a single exception. Therefore, only an exception is propagated at each object. These two levels of exception resolution in cooperative object groups yield to a *hierarchical exception resolution*.

The situation is depicted in Fig. 6. A method has been invoked in the group. The group is executing the method. The three objects of the group (*obj.1, obj.2* and *obj.3*) have two threads. At object *obj.1*, thread *th1* raises the exception Y and thread *th11* raises Z. None of the exceptions is handled in its corresponding thread. Object obj.1 applies local exception resolution. As a result, transaction T finishes at *obj.1* raising exception W. At *obj.2* a single exception is unhandled (X). Therefore, no local resolution is applied. Transaction T finishes at *obj.2* raising exception X. *obj.3* finishes transaction T succesfully. As two objects have finished the transaction raising an exception, distributed exception resolution is applied among the exceptions (W and X) raised by the group objects. The exception resulting from this resolution (A) is propagated to the client to notify the abort of transaction T.

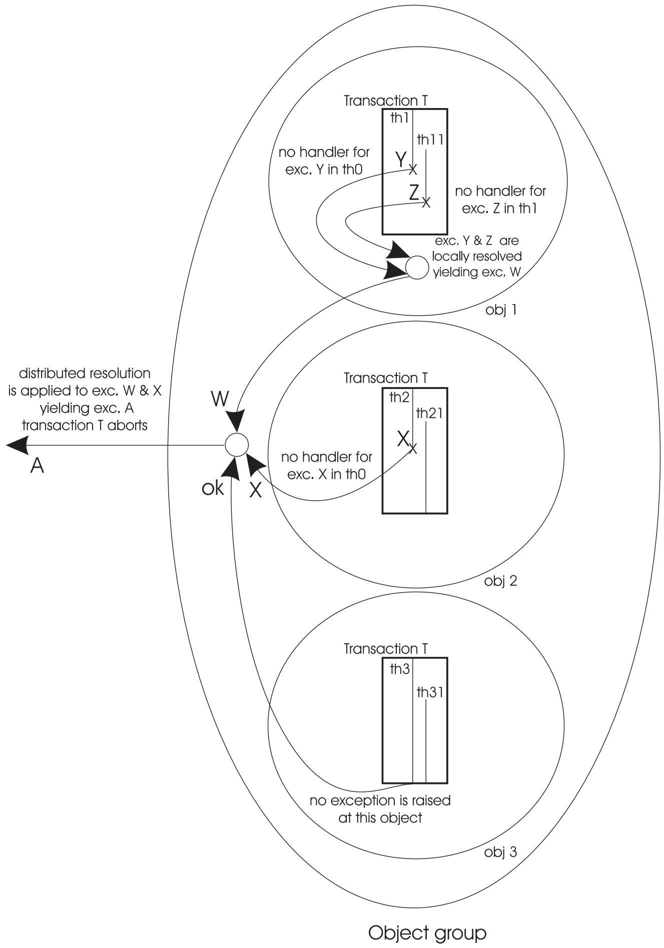

Fig. 6. Local and distributed exception resolution

4 Linguistic Support

The mechanisms previously described have been included in an Ada 95 extension, *Transactional Drago* [21]. Ada 95 is a programming language that provides objects, concurrency and exception handling, therefore, the extension of the language is quite natural. Although, in this section we refer to Ada, the same linguistic mechanisms can be easily applied to any programming language that provides objects, concurrency/distribution and exceptions, for instance, Java. The runtime of Transactional Drago is provided by TransLib [13] an object oriented library that can be used in combination with Ada.

Our proposal consists of introducing two new constructs: the *transactional block*, and *transactional object groups*. A transactional block allows to initiate a transaction (or transactional scope). Transactional blocks have a similar syntax to the Ada block statement. A keyword is used to distinguish a regular block statement from a transactional one. As block statements can have attached exception handlers, no new instruction is needed to handle aborts (since they are propagated as exceptions). Nesting of transactional blocks is used to implement nested transactions. Data items declared within a transactional block are subject to concurrency control (they are atomic) and can also be persistent. In order to ease the programmer task, concurrency control is implicitly set in Transactional Drago. In particular, read/write locking is used.

Ada tasks are used to create local threads where they are allowed (transactional blocks and methods of cooperative groups). If tasks are declared within a method of a cooperative object, they will be local threads of the associated transaction. A method (and a transactional block) cannot terminate until all its threads terminate, as happens with the regular block statement in Ada.

The Ada exception model is based on the termination model [5]. In this model when an exception is raised, the scope where the exception is raised terminates. Scopes in Ada are subprograms, task bodies, block statements, ... Unhandled exceptions are propagated from one scope to the enclosing one until they are successfully handled or they reach the outermost scope, which can be either 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. We have modified the behavior of unhandled exceptions in tasks when they belong to a transaction. Instead of losing the exception, the Transactional Drago runtime handles it to prevent its loss and enforce the semantics presented above.

Transactional object groups are provided, extending the Ada distributed systems annex by introducing a new kind of partition (Ada unit of distribution) corresponding to an object group. There is a peculiarity about cooperative object groups. In these groups there is a single class specification, but there might be multiple implementations (up to one per object of the group). Replicated object groups have a single implementation and only the number of replicas needs to be defined in this case.

In Ada there is no resolution of concurrent exceptions. In Transactional Drago exception resolution clauses are provided to associate resolution functions for concurrent exceptions either to a transactional block or to a cooperative object group method. These functions take two exceptions as arguments and return the resulting exception[1]. If n exceptions are concurrently raised within a transaction, the resolution function will be called $n-1$ times by the runtime system to obtain the final exception.

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

5 Implementation

The main implementation issue in the integration of transactional object groups and exceptions is how to combine exception handling and resolution with transaction termination protocols (commit and abort). Termination of traditional single-threaded transactions is trivially determined. A transaction finishes when its code executes the last statement of a transaction, which determines the outcome of the transaction. It finishes successfully, if a commit was executed. Otherwise, it aborts. However, the 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. If the transaction aborts, an exception must be choosen. 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*.

First, let's discuss the behavior of the protocol in the case of a non-distributed (client) multithreaded transaction. Initially, a thread (the main thread) starts a transaction. Once the transaction has started, this thread may spawn additional threads (secondary threads) that will also work on behalf of the transaction. Unlike in the traditional single-threaded model, it is necessary to perform a termination protocol where the main thread waits for the outcomes of all the transaction threads. If the transaction aborts (due to the exceptional termination of one or more threads), the termination protocol will apply the resolution function to obtain a unique exception. Then, it will abort the transaction and propagate the resulting exception to the enclosing exception context. If the transaction commits, the appropriate actions will be taken to make the results permanent.

Method invocations of object groups are performed as subtransactions. If the group is cooperative, these subtransactions might have two levels of concurrency. At the first level there is a thread at each group object that executes a method invocation. Those threads are distributed. The second level (is optional, and only available in cooperative groups) corresponds to local threads created in an object method. The termination algorithm is performed in two stages corresponding to the two concurrency levels. First, each object waits for the outcome of its local threads. It produces a successful outcome or an exception. In case of concurrent exceptions, the local resolution function is applied. Then, the caller acts as coordinator of the distributed termination. It waits for the outcome of each of the objects. Again, the final outcome is commit if all the objects finished successfully. In case of an abort, the exception propagated is chosen applying the distributed resolution function to the exceptions propagated by the distributed objects (that can be the result of a local resolution).

Transactions in replicated object groups also need a termination protocol. The caller plays the role of coordinator and waits for the results of all the group members. The majority decides the outcome of the transaction. If the outcome is abort, the exception raised by the majority is propagated to the caller. The objects not included in the majority are removed from the group. If the outcome is commit, no exception is propagated. If no majority is reached, the transaction aborts propagating the `abort_error` exception to the caller.

6 Related Work

There have been few attempts to integrate transactions and exceptions in the literature. One of the first ones was Argus [14], a distributed transactional programming language. Its approach is an orthogonal integration where transactions can commit or abort, independently of how they terminate (normally or exceptionally). In our opinion this implies some dangers. In particular, committing a transaction that terminates exceptionally is quite dangerous. An exception indicates that an operation has been called in its exceptional domain and hence, that the postcondition is not guaranteed. Therefore, if the transaction commits, a state that might be inconsistent is being made permanent.

[27] presents an approach for integrating coordinated atomic actions and exceptions. This work deals with a different context where processes join explicitly on-going atomic actions at different moments to cooperate within them. These processes are autonomous entities (for instance, different devices of a manufacturing system) that at some points cooperate to perform a particular action. For this reason, when an exception is raised within a coordinated atomic action, the exception is propagated to all the participants in the action.

Although, this approach is quite indicated for autonomous (active) entities, it is not applicable to a transactional system, where servers are passive entities that only perform work on behalf of clients. In particular, existing threads cannot join on-going transactions. In our approach, threads are created when an object group method is called. Those threads terminate with the method. Exceptions raised by any of these threads are not propagated to other group members. Resolution is applied and the result is propagated to the enclosing scope.

Arche [11] is a parallel object-oriented programming language. In this language a notion of object groups is provided. An object group is defined as a sequence of objects. The signature of a group operation (or multi-operation) results from converting each parameter from the original class (the base class of the group) into a sequence of parameters of the original type. This object group definition is targeted to the explicit parallelization of algorithms, and strongly contrasts with the one provided in our approach, where distribution is hidden behind the object group, and thus, it is transparent to the client of the group. This definition also differs in that it does not provide any fault-tolerance. Object group invocations are unreliably multicast to the group members. In our approach reliable multicast and transactions provide fault-tolerant atomic services.

Arche also provides exception handling. Exceptions are defined as objects to allow their extension/redefinition in subclasses. Two kinds of exceptions are defined: global and concerted. When a member of a group raises a global exception, the exception is propagated to all the group members if they try to synchronize with the signaler of the global exception. Concerted exceptions are used in synchronous multi-party communication. Exception resolution (possibly, user-defined) takes place for this kind of exceptions. If during cooperation, one or more members of a group raise an exception, a concerted exception is then locally computed and raised within each of the members.

Concerted exceptions have some similarities with concurrent exceptions in our approach. In both cases, exception resolution takes place and can be user-defined. However, resolution functions take different forms. In Arche, resolution functions take as parameter a sequence of raised exceptions, whilst in our approach, a binary resolution function is used. Arche's approach is more flexible, but it is also more complex for the programmer as it is necessary to iterate through the sequence of exceptions. Our approach is less flexible in this aspect, but it is much simpler from the programmer viewpoint, as the code of the resolution function only deals with two exceptions. Additionally, the iteration through the sequence of exceptions is performed by the underlying system for the sake of reliability. Additionally, in Arche there is a single level of resolution, while in our approach, there are two levels due to the nature of cooperative groups. Another difference between Arche and the approach we have presented comes from differences in the host languages. Arche uses an object oriented exception model, while our proposal uses the Ada exception model, that is not object-oriented, and extends it to deal with object groups.

There are several proposals for implementing distributed object groups [15, 17, 8, 16], but none of them deals with the semantics of exceptions.

7 Conclusions

All the concepts we have used in the article, namely transactions, exceptions, multithreading, and object groups, are implemented in modern object-oriented languages and systems. The importance of transactions as a mechanism to program reliable distributed systems has been recognized in current standards. Examples of this are CORBA object transaction service (OTS) [19], Java transaction service (JTS) [26], or the transactional support of Enterprise Java Beans [25]. Exceptions are already part of current object-oriented languages, like C++ and Java, and systems, e.g., CORBA. Multithreading has been around for more than a decade and it is supported by Java and CORBA, and by all modern operating systems. Object groups are becoming increasingly important and there several research efforts in this direction. However, a consistent integration of all of them has not been addressed to our knowledge in any language or system. The work presented in this article addresses how to integrate these existing mechanisms in a consistent way.

In this article we have presented a new use of exceptions in the context of transactional object groups. This approach is novel in that integrates backward and forward recovery provided by transactions and exceptions. This allows the use of forward recovery within a transaction.

The second contribution is a proposal of semantics for exceptions raised by object groups. Two cases have been considered depending on the group functionality. In replicated groups, the implicit exception resolution is used to avoid divergence of replicas' state. In cooperative groups exception resolution is user defined. Additionally, this semantics have also been integrated in the context of transactions. 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 exception resolution.

We believe that transactional object groups will play an important role in simplifying the programming of future reliable distributed systems, and therefore, a clear semantics for exceptions should be provided for these kinds of systems.

References

1. K. P. Birman and R. Van Renesse. *Reliable Distributed Computing with Isis Toolkit.* IEEE Computer Society Press, Los Alamitos, CA, 1993.
2. K.P. Birman. *Building Secure and Reliable Network Applications.* Prentice Hall, NJ, 1996.
3. F. Cristian. Exception Handling and Software Fault Tolerance. *ACM Transactions on Computer Systems*, C-31(6):531–540, June 1982.
4. 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.
5. J. B. Goodenough. Exception Handling: Issues and a Proposed Notation. *Communications of the ACM*, pages 683–696, 1975.
6. J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques.* Morgan Kaufmann Publishers, San Mateo, CA, 1993.
7. F. Guerra, J. Miranda, Á. Álvarez, and S. Arévalo. An Ada Library to Program Fault-Tolerant Distributed Applications. In K. Hardy and J. Briggs, editors, *Proc. of Int. Conf. on Reliable Software Technologies*, volume LNCS 1251, pages 230–243, London, United Kingdom, June 1997. Springer.
8. R. Guerraoui, P. Felber, B. Garbinato, and K. R. Mazouni. System support for object groups. In *ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'98)*, October 1998.
9. R. Guerraoui, R. Oliveira, and A. Schiper. Atomic Updates of Replicated Objects. In *Proc. of the Second European Dependable Computing Conf. (EDCC'96)*, volume LNCS 1150, Taormina (Italy), October 1996. Springer Verlag.
10. 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.
11. V. Issarny. An exception-handling mechanism for parallel object-oriented programming: Toward reusable, robust distributed software. *Journal Object-Oriented Programming*, 6(6):29–40, October 1993.
12. R. Jiménez Peris, M. Patiño Martínez, and S. Arévalo. Deterministic Scheduling for Transactional Multithreaded Replicas. In *Proc. of the Int. Symp. on Reliable Distributed Systems (SRDS)*, pages 164–173, Nürnberg, Germany, October 2000. IEEE Computer Society Press.
13. 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.
14. B. Liskov. Distributed Programming in Argus. *Communications of the ACM*, 31(3):300–312, March 1988.
15. S. Maffeis. Adding Group Communication and Fault-Tolerance to CORBA. In *Proc. of 1995 USENIX Conf. on Object-Oriented Technologies*, June 1995.
16. 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 Second IFIP WG 6.1 International Working Conference on Distributed Applications and Interoperable Systems, DAIS'99*, June 1999.
17. L. E. Moser, P. M. Melliar-Smith, P. Narasimhan, L. Tewksbury, and V. Kalogeraki. The Eternal System: An Architecture for Enterprise Applications. In *International Enterprise Distributed Object Computing Conference*, pages 214–222, September 1999.
18. J. E. B. Moss. *Nested Transactions: An Approach to Reliable Distributed Computing.* MIT Press, Cambridge, MA, 1985.
19. OMG. *CORBA services: Common Object Services Specification.* OMG.
20. M. Patiño Martínez, R. Jiménez Peris, and S. Arévalo. Integrating Groups and Transactions: A Fault-Tolerant Extension of Ada. In L. Asplund, editor, *Proc. of Int. Conf. on Reliable Software Technologies*, volume LNCS 1411, pages 78–89, Uppsala, Sweden, June 1998. Springer.
21. 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.
22. A. Schiper and M. Raynal. From Group Communication to Transactions in Distributed Systems. *Communications of the ACM*, 39(4):84–87, April 1996.
23. F. B. Schneider. Implementing Fault-Tolerant Services Using the State Machine Approach: A Tutorial. *ACM Computing Surveys*, 22(4):299–319, 1990.
24. 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.
25. Sun. *Enterprise JavaBeans.* http://java.sun.com/products/ejb/index.html.
26. Sun. *Java Transaction Service.* http://java.sun.com/products/jts/.
27. J. Xu, A. Romanovsky, and B. Randell. Coordinated Exception Handling in Distributed Object Systems: from Model to System Implementation. In *Proc. of Int. Conference on Distributed Computing Systems, ICDCS-18*, May 1998.