Group Transactions: An Integrated Approach to Transactions and Group Communication*

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

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

Abstract

Transactions and group communication are two techniques to build fault-tolerant distributed applications. They have evolved separately during a long time. It has been in the last years when researchers have proposed an integration of both techniques. Transactions were developed in the context of database systems to provide data consistency in the presence of failures and concurrent accesses. On the other hand, group communication was proposed as a basic building for reliable distributed systems, and it deals with consistency in the delivery of multicast messages. The difficulty of the integration stems from the fact that the two techniques provide very different kinds of consistency.

This work addresses the following questions: Is it possible to build a system providing both models? Is it possible to propose an integrated model? This work discusses how applications using group communication can benefit from transactions and vice versa. Groups of processes can deal with persistent data in a consistent way with the help of transactions and transactional applications can take advantage of group communication to build distributed cooperative servers as well as replicated ones. An additional advantage of an integrated approach is that it can be used as a base for building transactional applications taking advantage of computer clusters.

In this paper we address the integration of transactions and group communication proposing a new transaction model, *GroupTransactions*, where transactional servers can be groups of processes.

*This research has been partially funded by the Spanish National Research Council CICYT under grant TIC98-1032-C03-01.

1 Introduction

Two well-known techniques to build fault-tolerant distributed systems are transactions and group communication. Transactions [GR93] were developed to provide data consistency in the presence of concurrent accesses and failures. Group communication (multicast) [HT93] was proposed as a building block for reliable distributed systems. Group communication provides different levels of consistency in the delivery of multicast messages. These techniques have evolved quite independently, transactions in the context of databases and group communication to build distributed systems. It has not been until the last years when researchers have tried to integrate both techniques.

During the mid-nineties a debate [CS93, Bir94a, Ren94, Shr94] in the distributed systems community took place about whether group communication was enough to build any kind of fault-tolerant application. One of the conclusions of this discussion was that group communication and transactions are two complementary fault-tolerance techniques. Since then, several research groups have become interested in the integration of both models.

For instance, in [GS94] it is studied a basic mechanism, *Dynamic Terminating Multicast*, that can be used to build both transactional and group systems. This work takes advantage of the fact that commit and multicast algorithms are special cases of the consensus problem. They propose *Dynamic Terminating Multicast* as a basic mechanism to build both kinds of algorithms on top of it. Although their work deals with the integration of transactions and group communication, it just deals with the implementation of the commit protocol.

[Bir94b] proposes an integration of two models of consistency namely, virtual synchrony [Bir93] and linearizability [HW90]. In that integration services can be requested to groups of objects. Virtual synchrony guarantee that all group members perceive membership (view) changes at the same virtual time. Linealribility is a relaxation of serializability. It guarantees that the result of a set of concurrent invocations on a given object is equivalent to a serial execution. Serializability [BHG87] guarantees serial execution of concurrent transactions (sets of operations on different objects). This approach combines group communication with a relaxed concurrency control method, but it does not deal with the bulk of transactional systems that imply a stronger isolation condition, serializability, as well as failure atomicity. However, the paper points out that the inclusion of these properties, serializability and failure atomicity, in the model must be addressed.

[SR96] presents a more complete approach. In this paper, the authors explore the role of group communication in building transactional systems. The point of the paper is that group communication primitives are an application structuring mechanism that provides by itself transactional semantics. A transaction is sent in a single reliable total-ordered multicast message to all the servers the transaction needs to contact. Transaction atomicity is provided by multicast atomicity. That is, a message is delivered to all group members or to none of them. This approach has some penalties: groups must be dynamic, increasing transaction latency as a consequence. There are some other issues that are not addressed in the paper like recovery and transaction nesting.

In contrast the approach taken in [LS00, LS98] provides transactions as a basic mechanism while multicast is hidden from application programmers. Transactions can access replicated objects and therefore, they provide high availability. However, the locking granularity is an object group, which restricts concurrency. On the other hand, they try to avoid ordering guarantees for multicast, which increases performance. However, this can produce client starvation, when it cannot lock enough replicas.

Some programming languages [ST95, MÁAG96] have incorporated group communication primitives and features for replication, recovery and failure notification. Although, no facilities for transaction processing are available.

A different integrative approach has been the use of group communication as a building block to implement database replication. This approach has been taken in [PJKA00, PGS98, KPAS99, HAA99]. In these papers, reliable total ordered multicast is used to propagate updates from a replica where a transaction has been executed to the rest of the replicas. However, the emphasis is on improving the implementation of database replication rather than providing an integrated model.

Corba [OMGa] provides a transaction service (OTS) and many research projects [MMSN99, LM97, FGS98, MFX99, MSEL99] have integrated group communication in that framework. Corba has been recently enhanced with replication (FT-Corba [OMGb]). However, in [FG99] it is stated that the composition and FT-Corba does not result in any meaningful combination of their strengths.

In [MDB01] it is stated that none of the previous systems considers the problem of integrating group communication with the transactional frameworks they extend. The paper also describes how to integrate group communication with Jini [AOS+99] transactions to provide transparently transactions over replicated objects.

In all the previously mentioned works the integration of transactions and group communication has been identified as a key issue "to extend the power and generality of group communication as a broad distributed computing discipline for designing and implementing reliable applications" [SR96]. However, all the mentioned just consider group communication to build replicated systems, but groups can be used for other purposes [LCN90]. In this paper we address a complete integration of transactions and group communication. *Group Transactions* is a new transaction model in which transactional servers are groups of processes, either cooperative or replicated. Clients interact with these transactional group servers by multicasting their requests to them.

The paper is structured as follows, section 2 presents some definitions. Section 3 presents the proposed model, *Group Transactions*. Section 4 shows some applications of the model. In section 5 some implementation aspects are discussed. Finally, we present our conclusions in section 6.

2 Model and Definitions

2.1 System

A distributed system consists of a number of nodes with no shared memory among them that communicate exchanging messages. Network partitions are not considered.

In each node there is a set of processes. Each process belongs to a group. A group is seen as an individual logical entity, which does not allow its clients either to view its internal state, nor the interactions among its members. Processes belonging to the same group share a common interface and application semantics. A group interface is a description of remotely callable services, which must be implemented inside each group member.

Group communication primitives [HT93] are used to communicate with groups. A request to a group is multicast to all the processes of a group. Multicast messages are reliable. That is, a message is delivered at all available sites or none of them. Regarding message ordering we consider multicast primitives providing *FIFO order* (messages from the same sender are delivered in the order they were multicast) or *total order* (all messages are delivered at all processes in the same order). We assume a virtual synchronous model [Bir93], which ensures that a message is delivered in the same view by all processes that deliver the message.

We distinguish two kinds of groups, *replicated* and *cooperative* groups, according to the state and behavior of its members.

Replicated groups implement the active replication model, that is, they behave as state machines [Sch90]. According to this model all the group members are identical replicas, that is, they have the same state and should run on failure-independent nodes. Clients use total ordered multicast to submit their requests to replicated groups (Fig. 1.b). Therefore, all group members receive the same requests and produce the same answers.

A replicated group can act as a client of another group. Replication transparency is provided by the underlying communication system that filters the replicated requests so that a single message is issued. This is known as "n-to-1" communication [LCN90] (Fig. 1.c). This type of communication allows building programs with active replication and minimal additional effort for the programmer, that is, as if the group were made out of a single process. To our knowledge *Group_IO* [GMÁA97] is the only library that supports this kind of communication.

On the other hand, a *cooperative group* (Fig. 1.a) do not need to have neither the same state nor the same code. They are intended to divide data among its members and/or to express parallelism taking advantage of multiprocessing or distribution capabilities in order to increase the throughput. A simple example is matrix multiplication. Each member of a cooperative group can compute a row. Each member of the group has a copy of the matrix and knows which elements it has to multiply.

Members of a cooperative group are aware of each other and they can communicate multicasting messages to the group. This kind of communication is called *intragroup* communication. Invocations from a cooperative group are independent so they are not filtered by the communication system.

2.2 Transactions

A transaction is a sequence of operations that are executed atomically, that is, they are all executed (the transaction commits) or the result it is as if they had not been executed (it aborts). Two operations on the same data item conflict if they belong to different transactions and at least one of them modifies the data item. Transactions with conflicting operations must be isolated from each other to guarantee serializable executions [BHG87].

A transaction that is executed in a single node is called a *local transaction*, while those that are executed in several nodes are *distributed transactions*.

Transactions can be nested [Mos85]. Nested transactions or *subtransactions* can be executed concurrently, but isolated from each other. This kind of concurrency is competitive and only allows dividing a task into independent chunks (isolation forbids any cooperation). Transactions that are not nested inside another transaction are called *top-level transactions*. If a top level transaction aborts, all its subtransactions and their descendants will also abort, no matter whether they have committed or aborted. However, a subtransaction abortion does not compromise the result of its parent transaction (the enclosing one). Hence, subtransactions allow failure confinement.

3 Group Transactions

Group Transactions is a transaction model that integrates nested transactions and groups of processes (i.e., group communication). Whenever data consistency in the presence of concurrent accesses and failures is needed, data should be accessed within a transaction. Applications are made out of groups in our model. Group services can be invoked from transactions. To guarantee data consistency, groups invoked from a transaction cannot be invoked outside a transaction. Therefore, we distinguish transactional groups and non-transactional ones. Transactional group services are executed as subtransactions, hence, concurrent services are executed atomically and isolated from each other. Group data can be persistent or volatile. Subtransactions guarantee data consistency. Top level transactions are always started at a non-transactional process.

Traditional transactions have a single flow of execution. However, in order to use transactions in a more general setting, for instance to build fault-tolerant and high available distributed applications, transactions should have multiple flows of execution. These flows of execution (threads) or *intratransactional concurrency* allow to transform easily concurrent programs into fault-tolerant programs. This intratransactional concurrency is coooperative, in constrast to the concurrency provided by subtransactions that is competitive. Transactions with local flows of execution are called *multithreaded transactions*. Threads of the same transaction (siblings) can communicate among them.

Concurrency control mechanisms are used to guarantee the isolation of different transactions, however, those mechanisms do not apply to the local flows of execution of a transaction. A local thread of a transaction can write a data item while sibling is reading it. The underlying system must provide some kind of mutual exclusion to guarantee physical consistency, for instance *latches* [MHL+98].

Any transaction flow can start subtransactions. As a transaction can have several threads, a subtransaction and its parent transaction can run concurrently. Traditional nested

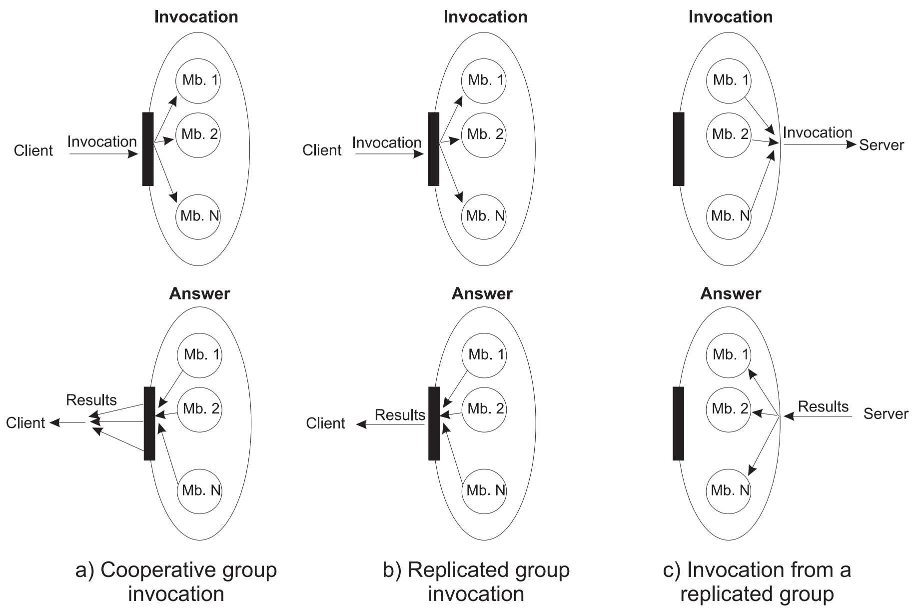

Figure 1: Group invocations and invocations from groups

transactions do not allow parent and child transactions to run concurrently. In our model, due to multithreading parent and child transactions can run in parallel. The semantics provided is that subtransactions are seen atomically by all the threads of its parent transaction. [HR93] studies different forms of parent/child transaction concurrency, but they are based on explicit synchronization, whilst our approach provides an implicit synchronization closer to the transaction philosophy.

Services of transactional groups must be called within a transaction and are executed as a distributed subtransaction or *multiprocess subtransaction*. The invocation of transactional groups within a transaction allows the isolation and atomicity of a set of group invocations, which it is not possible without transactional support.

Group members can have a persistent state. During recovery it is guaranteed that group members obtain a consistent state.

3.1 Transactional Replicated Groups

An invocation to a replicated group is executed as a *replicated subtransaction*, the same in all the group members. If a member of a replicated group fails during the execution of a replicated subtransaction, the subtransaction will not abort as far as there is at least one available group member. Therefore, replicated groups can tolerate $k-1$ failures, being k the number of group members. Transactional replicated groups provide high availability of both data and processing. If all the groups involved in a transaction (including the client) are replicated, the transaction will not abort in the presence of failures on client and server replicas, hence, transactions will be highly available.

A transaction on a replicated group is executed by a thread at each group member. Those threads cannot create additional threads in order to maintain replica determinism. However, a replicated group can execute several transactions concurrently to provide an adequate level of concurrency. A transactional replicated group uses a deterministic scheduler [JPA00], which ensures that all the replicas execute the same sequence of steps despite their multithreaded nature.

Since multicast messages are totally ordered and a deterministic scheduler is used, deadlocks on a single data item cannot happen. It is not possible an execution of two concurrent transactions where one of them locks an item in a subset of the replicas and the other locks the same item in another subset of the replicas. This problem happens in many replicated transactional systems.

Invocations from a transactional replicated group are filtered, so that only one invocation is made. These invocations are also executed as subtransactions of the calling transaction. The results of an invocation are sent back to all the members of the calling group. In Figure 2, there is a transactional replicated group (*gt1*) that invokes another transactional group (*gt2*). The *client group* invokes service *E1* in *gt1*. As a consequence a replicated subtransaction (*T1.1*) is created. If any of the two members of the replicated group fails, the subtransaction can still commit. When the replicas invoke service *E2* in *gt2*, a single request is made. The request result is returned to both members of the calling group.

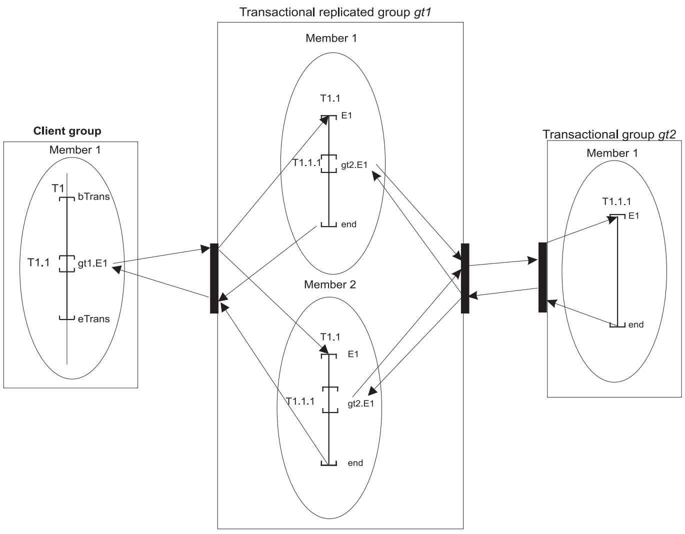

Figure 2: A transactional replicated group

When a failed member recovers, it performs a recovery process in order to undo the effects of uncommitted transactions on persistent data. Before joining the group, the state of the new member is updated with the state of a correct member. This state transfer is needed because the group could have been working while that member was down. The state transfer can be automatically made, since all group members have the same state. The state transfer is performed after all the subtransactions executing in the group at the time of the join message delivery have finished. The new member will execute the group invocations corresponding to transactions initiated after the delivery of the join message.

3.2 Transactional Cooperative Groups

Cooperative group invocations are executed as a *cooperative subtransaction*. All the members of the group execute in parallel a thread of the subtransaction. Since members of a cooperative group are aware of each other, they can use intragroup communication to cooperate. Members of a cooperative group can create new local threads to perform concurrently a service. The scope of these thread is restricted to the service where they are created.

Participants of a cooperative transaction can invoke another groups. These invocations are also executed as subtransactions. Figure 3 shows the interaction with a transactional cooperative group. Subtransaction *T1.1* is a cooperative subtransaction, where its participants can communicate (collaborate) using intragroup communication. Although the two group members invoke service *E2* in group *gt2*, invocations are independent and are executed as different subtransactions. Members of a cooperative group can also cooperate using another group. For instance, in the figure, if subtransaction *T1.1.1* executes before subtransaction *T1.1.2*, the latter will see the effects of the former subtransaction as both are subtransactions of the same transaction (*T1.1*).

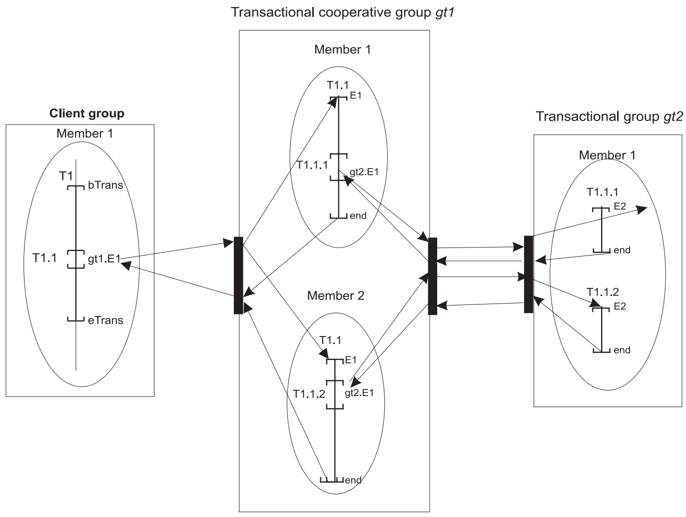

Figure 3: A transactional cooperative group

Cooperative groups can have either of the following failures modes: all-commit-or-none and any-commits. In the former failure mode a transaction commits only if all its processes end successfully. In the latter mode, if a node, where one of the group members resides, crashes while processing requests, the client will be notified and it will receive less answers from the group, but the transaction will not be aborted. The group could process further requests without all its members. This failure mode can be used when it is possible to perform services in a possibly degraded mode.

When a failed node restarts, group members in that node can join again their corresponding groups. Before joining the group a failed member will perform a recovery process. Since the rest of the group members could have been working in the meantime, the restarted member might need information from the group in order to update its state. The state transfer cannot be made transparently as it happens with replicated groups. The reason is that, in general, group members do not share the same state. Members of a cooperative group should define a recovery section to define the exchange of information needed before the new member joins the group.

4 Applications of the model

4.1 Cooperative Agenda

Traditional transaction models have precluded cooperation within transactions due to their isolation property. Some advanced transaction models [JK97] have been proposed to deal with cooperative applications. However, their approach has been quite different. Cooperative transactions [NRZ92] relax serializability and offer different kinds of locks less restrictive than read/write ones, so transactions corresponding to different clients can cooperate. This is useful in cooperative applications like CAD environments.

Let us see an example to illustrate the kind of cooperative applications for which cooperative groups are well-suited. The application under consideration will be in charge of the maintenance of the set of agendas of an organization or organization agenda. The application will register collective appointments (like department or section meetings) as well as private ones (go to the dentist). Each department of the organization keeps the agendas corresponding to its members or department agenda. A transactional cooperative group can be used to implement the organization agenda. The group provides services to access the distributed organization agenda. Each group member will keep a department agenda. Since the group is transactional, consistency of agendas is guaranteed in presence of concurrent accesses and node failures. This would be impossible with a non-transactional group. Using a traditional non-transactional group this would not be possible.

A user can add, remove or modify entries in her agenda. An agenda resides in a single group member, and thus, this service does not require any cooperation. A more interesting service is the one of making a reservation for a set of people (i.e. a meeting). A reservation is made in two steps. First, the user asks for the free common slots in the set of agendas within a particular period of time. Then, the user chooses one of the slots and the reservation is made. Each step is implemented with a different group service. Clients want to make the reservation atomically, hence, both services should be called from within a transaction. The server group provides the `FindFreeSlots` service to search for all the common free slots in a set of agendas during a particular period of time. It also provides the `MakeCollectiveReservation` service to perform the reservation.

The `FindFreeSlots` service requires cooperation among the members of the server group due to common free slots cannot be found locally at one member. Common free slots can be obtained by intersecting the free slots in the requested period of time of all the set of agendas.

This intersection can be performed in two steps. A member of the group can coordinate this process. In the first step, the coordinator will request (multicasting it) to rest of the group members to do the intersection of the involved local agendas. As a result of this request the coordinator will receive all the local intersections. In the second step, it will intersect all the local intersections to obtain the global intersection that will be returned to the client.

The global result can be computed in a more balanced way, if the coordinator sends its local intersection to the rest of the members. Thus, the rest of the members will intersect the coordinator local intersection with their local one, returning a (hopefully) smaller intersection and thus, the global intersection to be computed by the coordinator will be smaller.

As a result of `FindFreeSlots`, the client will receive the set of available free common slots. In the second step, the client will choose one of them to make the reservation through the `MakeCollectiveReservation` service. The reservation request will be multicast to all the group members and in response they will make the reservations corresponding to their local agendas involved in the reservation.

If a client wants to both make a reservation in a set of agendas as well as to book a meeting room. A server can be devoted to the reservations of meeting rooms in all the organization. In this case, the reliability requirements can be stronger and the organization can be interested in a highly available service, so even in the advent of node crashes the information about meeting rooms will be still available. Thus, a transactional replicated server can be devoted to held the information about meeting rooms providing the required availability. The client will make the reservation in the set of agendas and for a meeting room within the same transaction to guarantee that both reservations are done or none of them.

4.2 Fault-tolerant parallel computing

Most parallel programming languages have ignored the issue of fault tolerance. The reason is that parallel programs are usually written to increase the performance of a computation. Fault tolerance decreases performance and it is not worth in the current small-scale systems where processor crashes are not very frequent.

Nowadays, there is an increasing trend of using large-scale systems. This combined with the increasing need for bigger computations will make fault tolerance a topic to be dealt with in parallel computing. In parallel computations running for days in hundreds or thousands of processors failures will be likely.

This topic has been addressed in [Bal92] where the introduction of fault-tolerance in parallel applications using Argus [Lis88] is studied. One of the examples proposed in [Bal92] uses a master-slave scheme. The master can be seen as a client of the slaves. The master distributes subcomputations to the slave processes. This master can be seen as a client of the slaves. The master splits a computation up into subcomputations that are executed on different slaves. Thus, subcomputations are executed as transactions what confines processor crashes to a single subcomputation, preventing the repetition of the whole computation. To prevent the loss of the whole computation due to a crash, checkpoints are written into stable memory. The master to achieve the checkpoints executes each checkpointed computation within a different top-level transaction. Thus, after a crash of the master it is only necessary the repetition of subcomputations not checkpointed before the master crash.

However, it is not always feasible to split a computation into totally independent subcomputations. Some subcomputations might need to know intermediate results of other subcomputations. Traditional languages, like Argus, and models fail here, as transactions cannot cooperate. *Group Transactions* is an adequate model for fault-tolerant parallel computing, especially for those applications that need cooperation among subcomputations. Multiprocess and multithreaded transactions allow the work distribution in a cooperative way. Multiprocess transactions can used to distribute a computation among a set of nodes, whilst multithreaded transactions can take advantage of multiprocessing (or multiprogramming) capabilities to run multiple threads of a transaction in parallel.

In [Bal92] stable memory is used to save the results of subcomputations to prevent its lose in the advent of failures. The use of stable memory results quite expensive. In *Group Transactions* the creator of a transaction can be a replicated group. That group acts as a replicated master. Thus, subcomputations received by a replicated master can be stored in volatile memory and they will not be lost due to the availability provided by replication. Although group communication has a cost, it is much cheaper than stable memory that requires careful writes [Lam81].

In the approach using Argus, each subcomputation checkpoint is achieved by running the subcomputation as a top-level transaction. Top-level transactions are expensive due to the atomic commitment protocol. Thus, another advantage of the proposed model is that the whole computation can be executed as a single top-level transaction in the replicated master. Subcomputations can be run as subtransactions, thus failure confinement is guaranteed with a cheaper solution.

Some parallel algorithms can perform more efficiently by using broadcast [TKB92, YLC90]. For instance, the parallel version of shortest path Floyd's algorithm [Bal92]. As Argus only provides transactions, broadcasts can only be achieved by point-to-point messages with the corresponding loss of performance. Another advantage of *Group Transactions* is that members of cooperative group can multicast messages to the other group members, this yields to an increase of performance with respect to a traditional transactional system without group communication like Argus.

4.3 Other applications

4.3.1 Database replication

Another interesting application of the model is the implementation of replicated databases. It has been recently studied how to take advantage of group communication in the implementation of replicated databases [PJKA00, KPAS99, PGS98, HAA99]. These approaches allow to manage replication in database systems that do not support replication, although they require the modification of the database system implementation. *Group Transactions* provides a complementary approach to support replication in this context. It is possible to use a replicated group to manage a replicated database based on a database system that does not support replication. Each group member is in charge of interacting with one of the database copies. What it is more, databases can be heterogeneous (different vendors), each group member will hide the interaction with each different database.

4.3.2 Cluster computing

Cluster computing has become a new paradigm for high performance computing. A cluster of computers is a collection of computers connected with a high speed reliable LAN that provides a set of services. Traditional transactions cannot take advantage of this kind of architectures, while *Group Transactions* suits it perfectly. A transactional group server can be run on top of a cluster to provide transactional services. These transactional services can be distributed among the cluster to reduce their latency.

4.3.3 Transaction Processing in Multiprocessors

Multithreaded transactions can be used to take advantage of multiprocessors. Threads of a transaction can cooperate by means of shared memory and can be run on different processors to reduce the latency of the transaction.

5 Practical aspects for improving performance

One of the disadvantages of using reliable multicast are the incurred costs. The integration of transactions and group communication has the advantage that it allows cheaper implementations of multicast, in particular, reliable total ordered multicast. It has been found that in a local area network (LAN) like ethernet, multicasts are spontaneously total ordered in a LAN with a very high probability [PS98]. In this work they take advantage of this fact to implement an optimistic total ordered multicast. This optimistic multicast can be used in a transactional setting combined with a transaction manager, such that multicast requests are delivered immediately and only in the unlikely event of a violation of the total order, the involved transaction is aborted [PJKA00, KPAS99]. This approach helps to reduce the latency of total ordered multicasts. Our model can also take advantage of this approach.

Another technique that can be used to reduce the number of multicasts to replicated groups consists in grouping set of interactions with their control code in what we denominate a *CompositeCall* [PBJ+99, BJP+00] and send it to the replicated group where it will be interpreted in the style of stored procedures. On the server side the interpretation of the *CompositeCall* will yield local calls to the replica what will be much cheaper than crossing the network. In database systems a similar approach has been taken with stored procedures [Eis96].

6 Current Work and Conclusions

Group Transactions has been incorporated into a programming language called *Transactional Drago* that it is an Ada 95 extension. The language has been already defined [PJA98] and a preprocessor is under implementation. The run-time support is provided by an object oriented library *TransLib* [JPAB00] that implements *Group Transactions*.

In this paper we have presented an integration of transaction and group communication models into the new transaction model, *Group Transactions*. The proposed integration has been compared with others in the literature. This new model provides multithreaded and multiprocess transactions. This kind of transactions are useful in many kinds of applications and they allow to reduce the latency of transactional services as transactions can be parallelized taking advantage of multiprocessors and/or clusters of computers. With the proposed model it is possible to build transactional group servers, either cooperative or replicated. A formal description of the model has also been described with the ACTA framework.

Some applications of the model have been shown such as a cooperative application, the agenda of an organization. This example has shown how some inherent distributed services can take advantage from an integrated approach to process groups and transactions. Fault-tolerant parallel computing is a field where group transactions can be extensively used. A whole parallel computation can be run as a single top-level transaction with high availability. Multicast is useful for many parallel algorithms. With our model it is possible to make them fault-tolerant this kind of algorithms in contrast to Argus where multicast is substituted by point-to-point communication.

Finally, some hints about how to improve efficiency of the model implementation are provided.

References

[AOS+99] K. Arnold, B. O'Sullivan, R. Sheifler, J. Waldo, and A. Wollrath. *The JINI Specification*. Addison Wesley, 1999.

[Bal92] H. E. Bal. Fault-tolerant parallel programming in Argus. *Concurrency: Practice and Experience*, 4(1):37–55, February 1992.

[BHG87] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison Wesley, Reading, MA, 1987.

[Bir93] K. P. Birman. The Process Group Approach to Reliable Distributed Computing. *Communications of the ACM*, 36(12), December 1993.

[Bir94a] K. P. Birman. A Response to Cheriton and Skeen's Criticism of Causal and Totally Ordered Communication. *Operating Systems Review*, 28(1):11–20, January 1994.

[Bir94b] K. P. Birman. Integrating Runtime Consistency Models for Distributed Computing. *Journal of Parallel and Distributed Computing*, 23:158–176, 1994.

[BJP+00] F. Ballesteros, R. Jiménez-Peris, M. Patiño-Martínez, S. Arévalo, F. Kon, and R. H. Campbell. Using Interpreted CompositeCalls to Improve Operating System Services. *Software: Practice and Experience*, 30:589–615, 2000.

[CS93] D. R. Cheriton and D. Skeen. Understanding the Limitations of Causally and Totally Ordered Communication. In *Proc. of 14th ACM Symp. on Operating Systems Principles, Asheville, North Carolina*, pages 44–57, December 1993.

[Eis96] A. Eisenberg. New Standard for Stored Procedures in SQL. *SIGMOD Record*, 25(4):81–88, December 1996.

[FG99] S. Frolund and R. Guerraoui. Corba Fault-Tolerance: Why it does not add up? In *Proc. of the IEEE Workshop on Future Trends in Distributed Computing*, Cape Town, December 1999.

[FGS98] P. Felber, R. Guerraoui, and A. Schiper. The Implementation of a Corba Object Group Service. *Theory and Practice of Object Systems*, 4(2):93–105, 1998.

[GMÁA97] 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.

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

[GS94] R. Guerraoui and A. Schiper. Transaction model vs Virtual Synchrony model: bridging the gap. In K.P. Birman, F. Mattern, and A. Schiper, editors, *Theory and Practice in Distributed Systems*, volume LNCS 938, pages 121–132. Springer, 1994.

[HAA99] J. Holliday, D. Agrawal, and A. El Abbadi. The Performance of Database Replication with Group Communication. In *Proc. of 29th of Int. Symp. On Fault-Tolerant Computing FTCS'99*, Wisconsin, June 1999.

[HR93] T. Haerder and K. Rothermel. Concurrency Control Issues in Nested Transactions. *Very Large Databases Journal*, 2(1):39–74, 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.

[HW90] M.P. Herlihy and J. M. Wing. Linearizability: A Correctness Condition for Concurrent Objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, July 1990.

[JK97] S. Jajodia and L. Kerschberg, editors. *Advanced Transaction Models and Architectures*. Kluwer Academic Publishers, 1997.

[JPA00] R. Jiménez Peris, M. Patiño Martínez, and S. Arévalo. Deterministic Scheduling for Transactional Multithreaded Replicas. In *Proc. of IEEE Int. Symp. On Reliable Distributed Systems (SRDS)*, pages 164–173, Nürenberg, Germany, October 2000. IEEE Computer Society Press.

[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.

[KPAS99] B. Kemme, F. Pedone, G. Alonso, and A. Schiper. Processing Transactions over Optimistic Atomic Broadcast Protocols. In *Proc. of 19th Int. Conf. on Distributed Computing systems (ICDSCS)*. IEEE Computer Society Press, 1999. Also in Technical Report No. 325 ETH Zürich, Department of Computer Science.

[Lam81] B. W. Lampson. Atomic Transactions. In G. Goos and J. Hartmanis, editors, *Distributed Systems - Architecture and Implementation: An Advanced Course*, pages 246–265. Springer, 1981.

[LCN90] L. Liang, S. T. Chanson, and G. W. Neufeld. Process Groups and Group Communications. *IEEE Computer*, 23(2):56–66, February 1990.

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

[LM97] S. Landis and S. Maffeis. Building Reliable Distributed Systems with Corba. *Theory and Practice of Object Systems*, April 1997.

[LS98] M. C. Little and S. K. Shrivastava. Understanding the Role of Atomic Transactions and Group Communications in Implementing Persistent Replicated Objects. In *Proc. of 8th Workshop on Persistent Object Systems: Design, Implementation and Use*, Sept. 1998.

[LS00] M. C. Little and S. K. Shrivastava. Integrating Group Communication and Transaction to Implement Persistent Replicated Objects. volume LNCS 1752, pages 238–253, 2000.

[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.

[MDB01] A. Montresor, R. Davoli, and O. Babaoglu. Enhancing JINI with Group Communication.

Technical Report UBLCS-2000-16, Department of Computer Science, University of Bologna, January 2001.

[MFX99] S. Mishra, L. Fei, and G. Xing. Design, Implementation and Performance Evaluation of a Corba Group Communication Service. In *Proc. of the 29th Int. Symp. on Fault Tolerant Computing FTCS'99*, June 1999.

[MHL+98] C. Mohan, D. Haderle, B. Lindsay, H. Pirahesh, and P. Schwarz. ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging. In V. Kumar and M. Hsu, editors, *Recovery Mechanisms in Database Systems*, pages 145–218. Prentice Hall, NJ, 1998.

[MMSN99] L. E. Moser, P. M. Melliar-Smith, and P. Narasimhan. A Fault Tolerance Framework for Corba. In *Proc. of the 29th IEEE Int. Symp. On Fault Tolerant Computing, FTCS'96, Madison*, June 1999.

[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 IFIP WG 6.1 Int. Working Conf. on Distributed Applications and Interoperable Systems, DAIS'99*, June 1999.

[NRZ92] M. H. Nodine, S. Ramaswamy, and S. B. Zdonik. A Cooperative Transaction Model for Design Databases. In A. K. Elmagarmid, editor, *Database Transaction Models*, chapter 3, pages 53–85. Morgan Kaufmann Publishers, San Mateo, CA, 1992.

[OMGa] OMG. Corba services: Commong object services specification.

[OMGb] OMG. Fault tolerant corba draft adopted specification.

[PBJ+99] M. Patiño Martínez, F. Ballesteros, R. Jiménez Peris, S. Arévalo, F. Kon, and R. H. Campbell. Composite Calls: A Design Pattern for Efficient and Flexible Client-Server Interaction. In *Proc. of the Int. Conf. on Pattern Languages of Design*, Illinois (USA), August 1999.

[PGS98] F. Pedone, R. Guerraoui, and A. Schiper. Exploiting Atomic Broadcast in Replicated Databases. In D. J. Pritchard and J. Reeve, editors, *Proc. of 4th International Euro-Par Conference*, volume LNCS 1470, pages 513–520. Springer, September 1998.

[PJA98] 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.

[PJKA00] M. Patiño Martínez, R. Jiménez Peris, B. Kemme, and G. Alonso. Scalable Replication in Database Clusters. In *Proc. of the Int. Conf. on Distributed Computing DISC'00*, volume LNCS 1914, pages 315–329, Toledo (Spain), October 2000.

[PS98] F. Pedone and A. Schiper. Optimistic Atomic Broadcast. In S. Kutten, editor, *Proc. of 12th Distributed Computing Conference*, volume LNCS 1499, pages 318–332. Springer, September 1998.

[Ren94] R. Van Renesse. Why bother with CATOCS? *Operating Systems Review*, 28(4):22–27, October 1994.

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

[Shr94] S. K. Shrivastava. To CATOCS or not to CATOCS, that is the ... *Operating Systems Review*, 28(4):11–14, October 1994.

[SR96] A. Schiper and M. Raynal. From Group Communication to Transactions in Distributed Systems. *Communications of the ACM*, 39(4):84–87, April 1996.

[ST95] R. Schlichting and V. T. Thomas. Programming Language Support for Writing Fault-Tolerant Distributed Software. *ACM Transactions on Computer Systems*, 44(2):203–212, 1995.

[TKB92] A. S. Tanembaum, M. F. Kaashoek, and H. E. Bal. Parallel Programming Using Shared Objects and Broadcasting. *ACM Transactions on Computer Systems*, 25(8):10–19, August 1992.

[YLC90] C. Yang, R.C.T. Lee, and W. Chen. Parallel Graph Algorithms Based Upon Broadcast Communications. *ACM Transactions on Computer Systems*, 39(12):1468–1472, December 1990.