Chapter 1

*GROUP TRANSACTIONS**:

An Integrated Approach to Transactions and Group Communication

Marta Patiño-Martínez
Ricardo Jiménez-Peris
Facultad de Informática
Technical University of Madrid
Boadilla del Monte E-28660 Madrid Spain
{mpatino,rjimenez}@fi.upm.es

Sergio Arévalo
Escuela de Ciencias Experimentales
Universidad Rey Juan Carlos
Móstoles E-28933 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 over a long time. Only in recent years 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 block for reliable distributed systems. Group communication 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 chapter addresses the integration of both models and how applications using group communication can benefit from transactions and

*This research has been partially funded by the Spanish National Research Council CICYT under grant TIC98-1032-C03-01.
P. Ezhilchelvan and A. Romanovsky (eds.), Concurrency in Dependable Computing

vice versa. On one hand, groups of processes can deal with persistent data in a consistent way with the help of transactions. On the other hand, 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.

Keywords: Transactions, Reliable Multicast, Cooperative and Competitive Concurrency.

1. Introduction

Two well-known techniques to build fault-tolerant distributed systems are transactions and group communication. Transactions [1] were developed to provide data consistency in the presence of concurrent accesses and failures. Group communication (multicast) [2, 3] 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 in the context of reliable distributed systems. It has not been until the last years when researchers have tried to integrate both techniques.

During the mid-nineties a debate [4, 5, 6, 7] in the distributed systems community took place about whether group communication was enough to build any kind of distributed 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 [8] a basic mechanism it is studied, *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 consensus-like problems and therefore similar. 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. Another approach integrating group communication with atomic commitment is taken in [9] where the lower bound of three rounds for non-blocking atomic commitment [10, 11] is overcome by using optimistic delivery of uniform multicast.

[12] proposes an integration of two models of consistency namely, virtual synchrony [2] and linearizability [13]. In that integration services

can be requested to groups of objects. Virtual synchrony guarantees that all group members perceive membership (view) changes at the same virtual time. Linearizability is a relaxation of serializability [14] (that guarantees serial execution of concurrent transactions). Linearizability ensures that the result of a set of concurrent invocations on a given object is equivalent to a serial execution of them. This approach 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 in the model of these properties, serializability and failure atomicity, must be addressed.

[15] present 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 transactional semantics by itself. 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. The isolation is achieved by using total-order and by processing requests sequentially. 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 [16] provides transactions as a basic mechanism while multicast is hidden from application programmers. Transactions can access replicated objects and therefore provide high available data, but no support is provided for cooperative transactional applications.

Another approach combining competitive and cooperative concurrency control are coordinated atomic actions [17]. In this model, processes can join on-going atomic actions to cooperate within them. This is useful in those applications where processes are autonomous entities, that need to cooperate with some atomicity guarantees. However, this model is not suitable for transactional systems where servers are passive entities that are only activated when clients request services.

Some programming languages [18, 19] 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 [20, 21, 22, 23, 24]. 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 [25] did not provide fault-tolerance and many research projects [26, 27, 28, 29] have addressed this topic using replication and group communication. As a result, Corba has been recently enhanced with replication (FT-Corba [30]). However, in [31] it is stated that the composition of Object Transaction Service (OTS) and FT-Corba does not result in any meaningful combination of their strengths.

In [32] 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 transactions [33] to provide transparently transactions over replicated objects.

In all the previously mentioned approaches 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" [15]. However, all the mentioned approaches just consider group communication to build replicated systems, but groups can be used for other purposes [34]. 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 introduces some definitions. Section 3 presents the proposed model, *Group Transactions*. Section 4 shows some applications of the model. Finally, we present our conclusions in Section 5.

2. Model and Definitions

2.1 System

The system consists of a set of nodes $S = \{S_1, S_2, ..., S_N\}$ that communicate by exchanging messages through reliable channels. We assume an asynchronous system where nodes fail by crashing (no Byzantine failures).

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.

Sites are provided with a group communication system supporting strong virtual synchrony [35]. Group communication systems provide communication primitives and the notion of view (current connected sites). Changes in the composition of a view are delivered to the application. We assume a primary component membership [3]. Strong virtual synchrony ensures that messages are delivered in the same view they were multicast.

Group communication primitives [2] 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 to all sites in the view or to 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 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 [36]. 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 [34] (Fig. 1.c). This type of communication allows building programs with active replication and minimal additional effort from the programmer. That is, the programmer programs the group as if the group were made out of a single process. To our knowledge *Group_IO* [37] is the only protocol that supports "n-to-1" communication.

On the other hand, members of a *cooperative group* (Fig. 1.a) do not need to have either the same state or 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. For instance, a cooperative group can be used to perform matrix multiplication. Each member of a cooperative

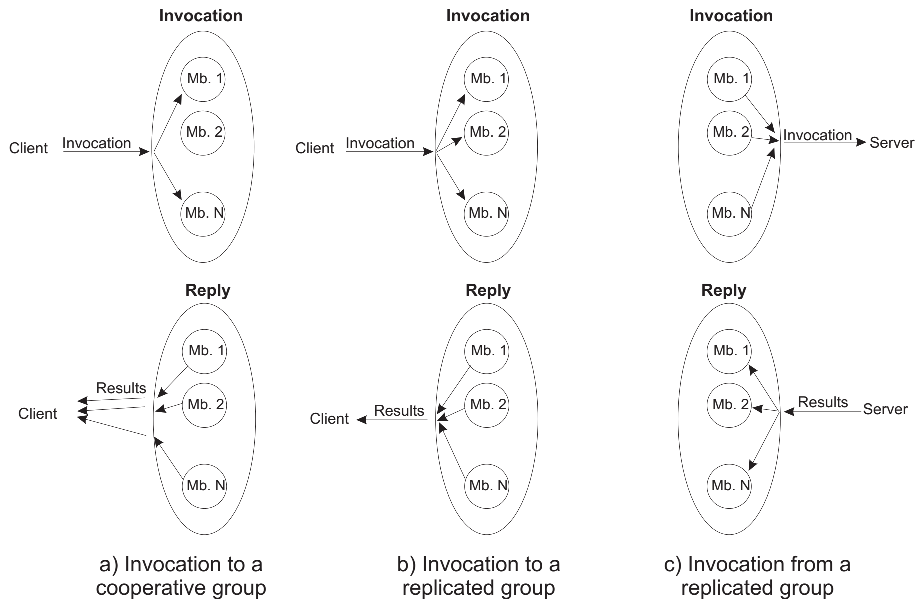

Figure 1. Group invocations and invocations from groups

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 by 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 is as if none of them had 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 [14].

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 [38]. 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 process groups. In this model, transactional servers can be groups of processes. This kind of server is invoked from within transactions to enforce data consistency. Transactional group services are executed as distributed subtransactions or *multiprocess subtransactions.* The invocation of transactional groups within a transaction allows the isolation and atomicity of a sequence of group invocations, which it is not possible without transactional support.

Traditional transactions are single threaded. However, in order to use transactions in a more general setting, for instance to build fault-tolerant and high available concurrent and distributed applications, transactions might need to have multiple threads. The *intratransactional concurrency* provided by multithreading allows to transform easily concurrent applications into transactional ones. This intratransactional concurrency is cooperative, in contrast to the concurrency provided by subtransactions that is competitive. Transactions with local threads 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 threads of a transaction. A local thread of a transaction could write a data item while a sibling is reading it. The underlying system must provide some kind of mutual exclusion to guarantee physical consistency, for instance *latches* [39].

Any transaction thread can start subtransactions. As a transaction can have several threads, a subtransaction and its parent transaction can run concurrently. Traditional nested 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. [40] study 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.

Members of a transactional group might have persistent state, which consistency is guaranteed by the transactional semantics. In case of failure, a recovery procedure takes place to recover the last 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 (either at the client or server side), 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 [41], 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.

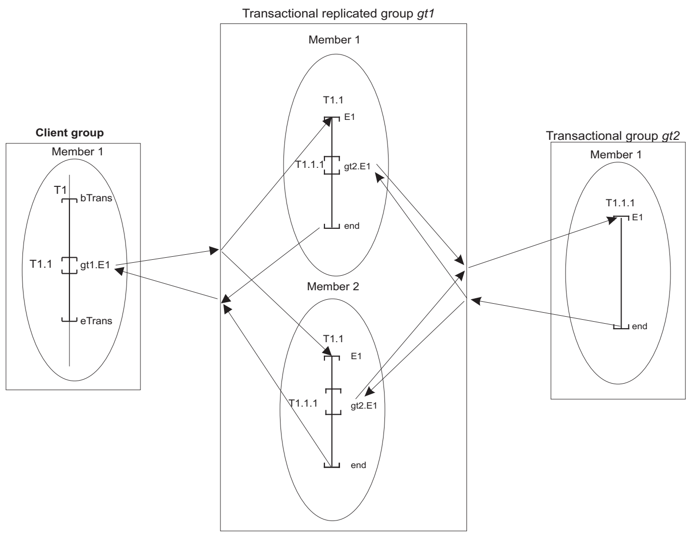

Figure 2. A transactional replicated group

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 performed, since all group members have the same state. State transfer is started once all the transactions, that were active when the join message was delivered, have finished their execution. 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.* Each of the group members executes a thread of the subtransaction in parallel. 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 threads is restricted to the service where they are created.

Participants of a cooperative transaction can invoke other 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*).

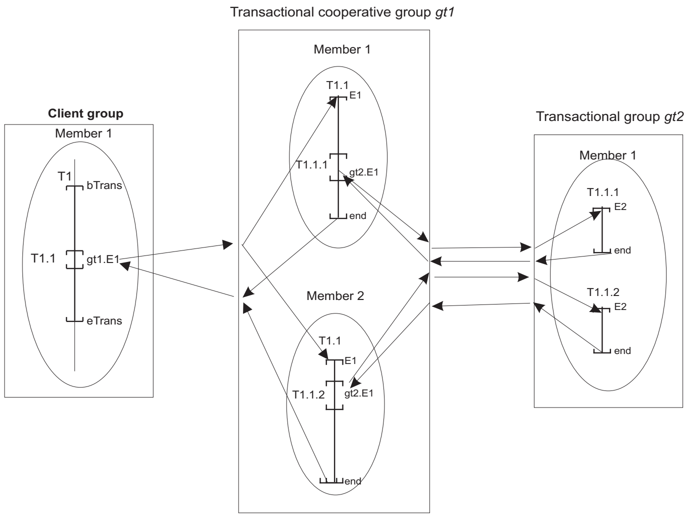

Figure 3. A transactional cooperative group

Cooperative groups can have either of the following failure modes: all-commit-or-none and any-commits. In the former failure mode a transaction commits only if all the group members finish 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.

Traditional transaction models have precluded cooperation within transactions due to their isolation property. Some advanced transaction models [42] have been proposed to deal with cooperative applications. However, their approach has been quite different. Cooperative transactions [43] 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.

4. Applications of the model

4.1 Cooperative Agenda

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 have been impossible with a traditional non-transactional group.

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 by 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 involved 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 multicast a request to the rest of the group members to perform 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 involved local agendas.

A client might want 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 book 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 because they focus on increasing the performance of a computation. Nowadays, there is an increasing trend to use large-scale distributed systems for parallel programs, as well as an increasing need for bigger computations. Parallel computations might run for days in hundreds or thousands of nodes, and in this context failures will be more likely. Therefore, fault tolerance of parallel computations should be addressed.

This topic has been addressed in [44] where the introduction of fault-tolerance in parallel applications using Argus [45] is studied. One of the examples proposed in [44] uses a master-slave scheme. 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 crashed.

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 be 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 [44] stable memory is used to save the results of subcomputations to prevent its lose in the advent of failures. The use of stable memory is quite expensive in terms of latency. 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 [46].

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 [47]. For instance, the parallel version of shortest path Floyd's algorithm [44]. 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 a 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

Database replication is another interesting application of the model. It has been recently studied how to take advantage of group communication in the implementation of replicated databases [20, 48, 23]. These approaches allow to manage replication of database systems. Replicated groups of *Group Transactions* can be used similarly to implement a replicated database.

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. A transactional group server can be run on top of a cluster to provide transactional services. The execution of these transactional services can be distributed among the cluster to reduce their latency. Traditional transactions preclude any cooperation within a transaction, and therefore cannot take advantage of cluster computing.

Transaction processing in multiprocessors is a different context where multithreaded transactions are interesting. 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. Current Work and Conclusions

Group Transactions have been incorporated into a programming language called *Transactional Drago* that is an Ada 95 extension. The language has already been defined [49] and a preprocessor is under implementation. The integration of exception handling with the model is addressed in [50]. The run-time support is provided by an object oriented library *TransLib* [51, 52] that implements *Group Transactions*.

In this paper we have presented an integration of transaction and group communication models into a new transaction model, *Group Transactions*. This new model provides multithreaded and multiprocess transactions, which are useful in many kinds of applications. The proposed model allows to build transactional distributed servers, either cooperative or replicated. Cooperative groups can be used to reduce the latency of transactional services by parallelizing transactions and thus, taking advantage of multiprocessors and/or clusters of computers. Replicated groups provide highly available transactional processing. The model has been formally described elsewhere [53] using the Acta framework [54].

Some applications of the model have been shown such as the agenda of an organization (a cooperative transactional application). This example has shown how some inherent distributed services can take advantage from an integrated approach for group communication and transactions. Fault-tolerant parallel computing is another field where group transactions can be extensively used. Our model allows the addition of fault-tolerance to parallel algorithms.

References

[1] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques.* Morgan Kaufmann, 1993.

[2] K.P. Birman. *Building Secure and Reliable Network Applications.* Prentice Hall, NJ, 1996.

[3] G. V. Chockler, I. Keidar, and R. Vitenberg. Group Communication Specifications: A Comprehensive Study. *ACM Computer Surveys*, 2001.

[4] D. R. Cheriton and D. Skeen. Understanding the Limitations of Causally and Totally Ordered Communication. In *Proc. of ACM SOSP*, pages 44–57, 1993.

[5] 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.

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

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

[8] R. Guerraoui and A. Schiper. Transaction model vs Virtual Synchrony model: bridging the gap. In *Theory and Practice in Distributed Systems, LNCS 938.* Springer, 1994.

[9] R. Jiménez-Peris, M. Patiño-Martínez, G. Alonso, and S. Arévalo. A Low Latency Non-Blocking Atomic Commitment Protocol. In *Proc. of the Int. Conf. on Distributed Computing, DISC'01, LNCS-2180* , pages 93–107, 2001.

[10] C. Dwork and D. Skeen. The Inherent Cost of Nonblocking Commit. In *Proc. of ACM PODC*, pages 1–11, 1983.

[11] I. Keidar and S. Rajsbaum. On the Cost of Faul-Tolerant Consensus Where There No Faults - A Tutorial. Technical Report MIT-LCS-TR-821, 2001.

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

[13] 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.

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

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

[16] 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*, Sept. 1998.

[17] A. Romanovsky, S.E. Mitchell, and A.J. Wellings. On Programming Atomic Actions in Ada 95. In *Proc. of Int. Conf. on Reliable Software Technologies, LNCS 1251*, pages 254–265. Springer, June 1997.

[18] R. Schlichting and V. T. Thomas. Programming Language Support for Writing Fault-Tolerant Distributed Soft. *ACM Trans. on Comp. Syst.*, 44(2):203–212, 1995.

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

[20] 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.

[21] F. Pedone, R. Guerraoui, and A. Schiper. Exploiting Atomic Broadcast in Replicated Databases. In *Proc. of Euro-Par Conference, LNCS 1470*, pages 513–520. Springer, 1998.

[22] B. Kemme and G. Alonso. A new approach to developing and implementing eager database replication protocols. *ACM TODS*, 25(3):333–379, September 2000.

[23] J. Holliday, D. Agrawal, and A. El Abbadi. The Performance of Database Replication with Group Communication. In *Proc. of FTCS'99*, 1999.

[24] U. Fritzke and Ph. Ingels. Transactions on Partially Replicated Data based on Reliable and Atomic Multicasts. In *Proc. of IEEE ICDCS'01*, pages 284–291, 2001.

[25] OMG. Corba services: Common object services specification, 1995.

[26] 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*, June 1999.

[27] S. Landis and S. Maffeis. Building Reliable Distributed Systems with Corba. *TAPOS*, April 1997.

[28] 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.

[29] 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 Int. Working Conf. on Distributed Applications and Interoperable Systems*, 1999.

[30] OMG. Fault tolerant corba specification, 1999.

[31] 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*, December 1999.

[32] A. Montresor, R. Davoli, and O. Babaoglu. Enhancing JINI with Group Communication. Technical Report UBLCS-2000-16, Computer Science Dep., University of Bologna, 2001.

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

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

[35] R. Friedman and R. van Renesse. Strong and Weak Virtual Synchrony in Horus. Technical report, CS Dep., Cornell Univ., 1995.

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

[37] F. Guerra, J. Miranda, Á. Álvarez, and S. Arévalo. An Ada Library to Program Fault-Tolerant Distributed Applications. In *Proc. of Int. Conf. on Reliable Software Technologies, LNCS 1251*, pages 230–243. Springer, June 1997.

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

[39] C. Mohan, D. Haderle, and B. Lindsay. ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging. In *Recovery Mechanisms in Database Systems*, pages 145–218. Prentice Hall, 1998.

[40] T. Haerder and K. Rothermel. Concurrency Control Issues in Nested Transactions. *Very Large Databases Journal*, 2(1):39–74, 1993.

[41] 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.

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

[43] M. H. Nodine, S. Ramaswamy, and S. B. Zdonik. A Cooperative Transaction Model for Design Databases. In *Database Transaction Models*, pages 53–85. Morgan Kaufmann, 1992.

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

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

[46] B. W. Lampson. Atomic Transactions. In *Distributed Systems*, pages 246–265. Springer, 1981.

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

[48] B. Kemme, F. Pedone, G. Alonso, and A. Schiper. Processing Transactions over Optimistic Atomic Broadcast Protocols. In *Proc. of ICDCS'99*, 1999.

[49] M. Patiño-Martínez, R. Jiménez-Peris, and S. Arévalo. Integrating Groups and Transactions: A Fault-Tolerant Extension of Ada. In *Proc. of Int. Conf. on Reliable Software Technologies, LNCS 1411*, pages 78–89. Springer, June 1998.

[50] M. Patiño-Martínez, R. Jiménez-Peris, and S. Arévalo. Exception Handling in Transactional Object Groups. In *Advances in Exception Handling, LNCS-2022*, pages 165–180. Springer, 2001.

[51] 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.

[52] J. Kienzle, R. Jiménez Peris, Alexander Romanovsky, and Marta Patiño Martínez. Transaction Support for Ada. In *Proc. of Int. Conf. on Reliable Software Technologies*, volume LNCS-2043, pages 290–304. Springer, May 2001.

[53] M. Patiño-Martínez. *Language and Model for Cooperative and Replicated Distributed Transactional Systems.* PhD thesis, Technical University of Madrid (UPM), 1999.

[54] P. K. Chrysanthis and K. Ramamritham. Synthesis of Extended Transaction Models Using ACTA. *ACM Transactions on Database Systems*, 19(3):450–491, 1994.