Transactions, Messages and Events: Merging Group Communication and Database Systems *
– extended abstract –

Bettina Kemme Gustavo Alonso
Information and Communication Systems Research Group
Institute of Information Systems, Swiss Federal Institute of Technology (ETH)
ETH Zentrum, CH-8092 Zürich
E-mail: {kemme,alonso}@inf.ethz.ch

Abstract

In this paper, we report on the DRAGON (Database Replication on Group Communication) project and the results obtained so far at ETH Zürich. The goal of DRAGON is to improve database replication strategies by using group communication primitives. As a first step in this direction, we have developed a suite of protocols implementing database replication based on broadcast primitives. We are currently embedding these protocols into a database management system to show the feasibility of the approach. In this short paper, we briefly discuss these protocols and analyze how they can be included in a database system. Our contribution is to further clarify the effort involved in the process and to explicitly state some of the issues commonly not considered by existing work. These issues are analyzed in detail and solutions are provided taking as example our current development efforts.

1 Introduction and Motivation

Data replication is essential in distributed systems to enhance performance and to provide fault tolerance. Fast access to a local copy instead of expensive remote calls and the possibility to fall back on a second copy when the nearest replica is not accessible make replication an attractive mechanism. However, maintaining consistency can be quite expensive and in some application areas it is considered to be infeasible. In particular, in distributed databases existing solutions can be roughly divided into two categories [GHOS96]: synchronous and asynchronous. In *synchronous* replication, updates are propagated within the scope of the transaction where they take place, that is, by the time the transaction commits, all copies have been updated. In *asynchronous* replication, on the other hand, changes are propagated only after the transaction making the changes has committed. From a conceptual point of view, synchronous replication is clearly preferable because it provides full correctness and consistency. Commercial systems, however, rarely implement synchronous replication. Instead, they offer a wide range of asynchronous solutions which are a compromise between performance and correctness (leaning more to the former than to the latter). The reason behind this choice is that synchronous replication is widely believed to have serious drawbacks. Many of the protocols are too complex to be feasible in a real database system (e.g., we do not know of any commercial database system using quorums [BHG87]). Furthermore, the poor response times (due to sending messages within transaction boundaries) and the lack of scalability of existing protocols (due to different overheads and deadlock rates) [GHOS96] make synchronous replication unattractive in practice.

While these concerns are understandable and justified in several scenarios, we believe there are cases in which synchronous database replication is a feasible and the preferable option. In particular, when working with clusters of computers, advances in LAN technology have eliminated much of the overhead associated with communication [KPAS99]. Databases are increasingly being deployed over such clusters, and in these environments it makes sense to provide full consistency among the copies. However, the complexity issue still needs to be resolved.

It has been suggested that a promising approach to solve this issue is to use group communication systems [HT93]. In the last few years, there has been a significant amount of theoretical work in this direction. For instance, Agrawal et al. [AAES97] discuss determinism in databases as a pre-requisite to link them to group communication sys-

* Part of this work has been funded by ETH Zürich within the DRAGON Research Project (Reg-Nr. 41-2642.5)

tems. Kemme and Alonso [KA98] have proposed a suite of database replication protocols based on group communication showing how group communication can help with fault tolerance issues and the concurrency and replica control mechanisms of the database system. Pedone et al. [PGS97, PGS98] have tackled the group communication side of the problem and have also discussed protocols using certification mechanisms. In spite of these promising first steps, there is still a clear gap between the protocols proposed and practical and implementable solutions. This short paper fills this gap by discussing the conceptual and practical difference between databases and group communication systems and outlining the architecture of our replicated database system DRAGON, an extension of the publicly available database system PostgreSQL [Pos98]. We believe the experiences reported in this paper are relevant even when databases are not directly involved. For instance, similar problems arise when integrating transactional based services (such as CORBA's OTS, Object Transactional Service [Gro98]) and group communication based services (like OGS – Object Group Service) [Fel98]). In this regard, the lessons learned from DRAGON are likely to be applicable to a wide range of practical fault tolerance problems.

In what follows, the architecture of DRAGON is described in detail and we show in which way the traditional database structure must be extended, reconstructed and combined with group communication semantics to provide an efficient, fault tolerant system. For reasons of space, we limit the discussion to the issues of serialization order, ordering of events, the message structure and recovery.

2 Model

2.1 Basic Modules

Messages will be exchanged using group communication primitives. These primitives offer reliable multicast mechanisms within a group of sites, providing different levels of fault tolerance and ordering mechanisms. Out of all the possibilities, the properties of *total order broadcast* (broadcasted messages are delivered to all sites in the same order), *reliable* and/or *uniform reliable message delivery* (if a (correct) site delivers a message, all sites deliver it) and *virtual synchrony* (all group members perceive site failures and recovery at the same logical time) are of special interest.

A distributed database consists of a number of sites communicating via message passing, and a number of objects. Objects are accessed by executing transactions. A transaction is a set of read and write operations on objects. Transactions are executed atomically, i.e., a transaction either commits or aborts the results of all its operations [BHG87]. Two operations conflict if they are from different transactions, access the same object and at least one of them is a write.

In order to guarantee correct executions, transactions with conflicting operations must be isolated from each other. We do this via *locking protocols*. Before accessing an object, a transaction acquires a lock on the object and conflicting locks on the same object are not allowed. An alternative method is the use of *multiple versions* of the same object. While write operations still use locks on the current versions of the objects, read operations access older versions and hence, do not directly conflict with write operations. Care has to be taken that transactions still read consistent data and that read/write dependencies are resolved correctly.

The usual correctness criterion is *serializability* which requires the concurrent execution of transactions to be equivalent to a serial execution. In practice, also other correctness criteria are common. They allow different levels of isolation defined in terms of possible inconsistencies [BBG+95].

Our algorithms for replica control are based on a modified version of the *all available copies* approach [BHG87]. We assume a fully replicated system, i.e., all nodes have copies of exactly the same objects. Read operations are performed locally on one site while write operations are sent to all available sites. Due to performance reasons, we bundle all write operations into a single message [KA98].

2.2 Basic Replication Protocol

The basic idea of the algorithms presented in [KA98] is to use the total order broadcast to support the serialization of transactions. While [KA98] provides an in-depth discussion of the algorithms and a detailed performance analysis, in this paper we concentrate more on architectural aspects and hence, present only the rough outline of the algorithms.

A transaction T invoked at a site S is said to be *local* at S and *remote* at all other sites. All read operations of T are performed locally. The write operations are deferred until all read operations have been executed and bundled into a single message; the message is then sent to all available nodes (including the local site) using total order broadcast. Upon arrival at a site, conflicting transactions are ordered according to the total order determined by the communication system. This is done by requesting all write locks for a given transaction in an atomic step before processing the next message. The serial processing of lock requests in the order of message delivery guarantees that conflicting write operations are ordered in the same way at all sites.

Regarding the case of read/write conflicts we have proposed several alternatives in [KA98] providing different degrees of isolation. Our goal is to keep the overhead of read operations as small as possible. Hence, remote sites should not have to know about the read set of the transaction. This means that read/write conflicts can only be seen and resolved at the local site of the reading transaction. The simplest way to treat read/write conflicts is to abort a local transaction keeping a read lock whenever a conflicting write operation arrives. With this, any kind of deadlock is avoided and the protocol provides full serializability. To reduce abort rates, other possibilities include short read locks or a multiversion concurrency control mechanism that allows read operations to read from a consistent snapshot of the data whereby read/write conflicts are indirectly detected by keeping track of the time write operations take place [BBG+95]. These mechanisms, although not guaranteeing serializability, provide a high degree of isolation and are common practice in existing systems [Ora95]. However, a detailed description of the handling of read/write conflicts goes beyond the scope of this paper, since we treat them applying purely database related techniques and hence, they do not influence our integration of database and group communication systems.

3 Merging Database and Group Communication Systems

Databases and group communication systems are not compatible, both conceptually and physically. Additional work needs to be done to develop a working system combining both. This section will discuss some of the differences between group communication and database semantics and how they have been resolved in DRAGON.

3.1 Database and Communication Systems

As core system for DRAGON we use the object-relational database system PostgreSQL [Pos98] which implements an extended subset of SQL and is a single server system. It has over 250,000 lines of well structured C code. The actual changes to PostgreSQL have mainly involved the lock and the process management modules, and some parts of the flow control.

With respect to group communication, we could use any of the existing systems since our algorithms do not modify the communication primitives. The only requirements are the ones described in section 2. Our current version of DRAGON uses the Transis system [DM96], but we are extending the initial prototype to other systems [vRBM96, MMSA+96] for comparison and flexibility purposes.

3.2 The DRAGON Architecture

As shown in Figure 1, DRAGON consists of a number of servers, each of which runs an instance of the DRAGON database system. At each server, the process structure follows that of PostgreSQL. In the figure, all clear shapes represent original PostgreSQL constructs. When a *client* wants to use the database system, it sends its request to the postmaster. For each client, the postmaster creates a *backend* process and all further communication is done directly between backend and client. The client can submit an arbitrary number of transactions to the backend one at a time.

To implement replication, this structure must be extended to include the communication system and to deal with remote operations. The shadowed shapes of the Figure 1 depict this extension. A client may connect to any of the DRAGON servers. The transactions of this client are called the *local* transactions of the server. For each client the postmaster creates a *local backend*. Control of the replication protocol takes place at the *replication manager*. Local backends transfer the write operations of local transactions to the replication manager. The replication manager forwards them via the *communication manager* to the group communication system which, in turn, broadcasts them to all sites. The existence of the communication manager allows us to hide the interface and characteristics of the group communication system. Upon delivery of a message, the replication manager looks up the owner transaction. If the message refers to a local transaction it is forwarded to the corresponding local backend. Otherwise, the replication manager creates a *remote backend* who handles the local part of the *remote transaction*.

3.3 Serializability vs. Total Order

The first obstacle to overcome when using group communication primitives in conjunction with databases is the different concepts of ordering they employ. The different orders usually provided by group communication systems (FIFO order of messages of each site, causal [BJ87] or total order) tend to reflect temporal (global time) precedence of events. In the case of a total order the arrangement for concurrent events is usually arbirtray [HT93]. Databases, on the other hand, use serializability, which is a partial order based on explicit data flow dependencies. Although serializability implies the equivalence to a total – serial – order [BHG87], in practice transactions are always processed concurrently whenever they do not conflict. However, although serializability represents a weaker order than the total order, both FIFO and causal order are not strong enough to capture its data dependencies. Hence, if we want to exploit group communication ordering primitives the only alternative is to use the total order. The question is then how to combine these two notions without introducing overhead and respecting the correctness of both database and communication primitive.

We have resolved this issue by using the two orderings

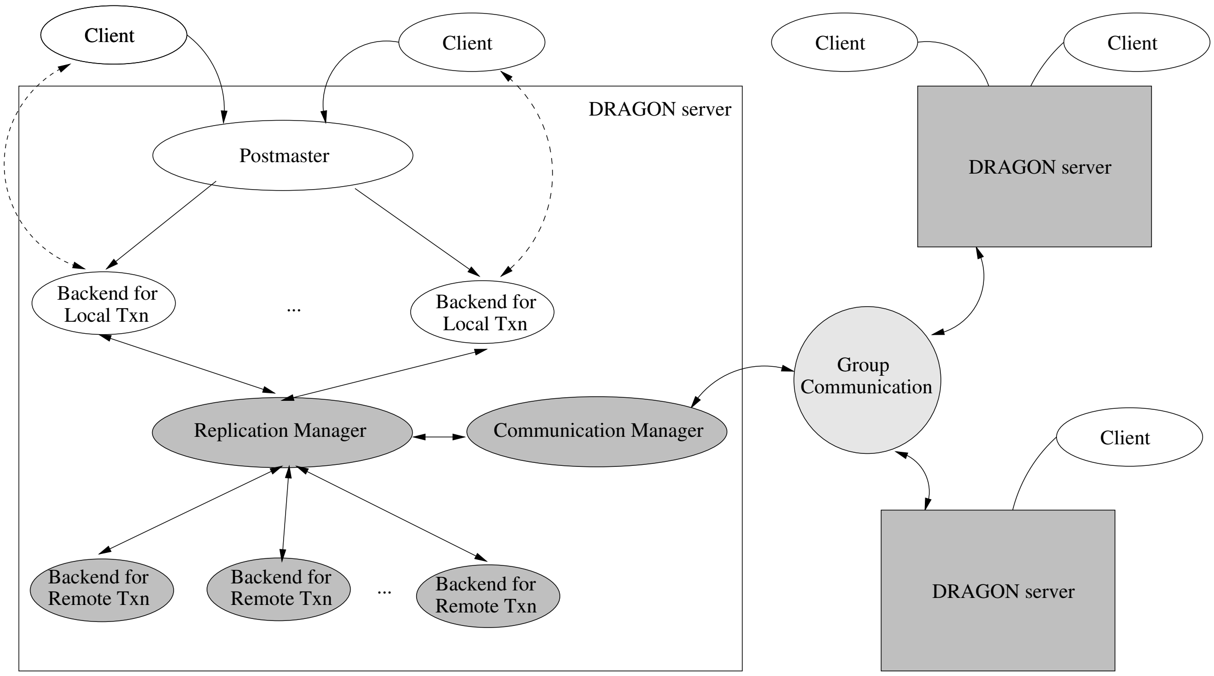

Figure 1. Architecture of DRAGON

for different purposes and allowing the serialization order (which is less restrictive) to contradict the total oder in concrete cases which do not affect correctness. Thus, we use the total order, which is identical at all sites, as a *pattern to follow* when establishing the serialization order. In particular, we only require conflicting operations to be ordered according to the total order while non-conflicting operations might be executed out of sequence, and possibly in different orders at different sites without affecting serializability. This approach takes advantage of the properties of both systems. The total order provides the basis for coordination and the serialization order looks at the transaction semantics to determine what must be ordered and what is allowed to be executed concurrently.

3.4 Read/write Operations vs. Events

In group communication, the state of a process is determined by the history of *events* that take place at that process. Some events are local while others refer to sending and receiving messages [Lam78]. In the context of databases, we can consider the read and write operations of transactions as the main events in the system.

In typical database applications, a client transaction submits one operation after the other, waiting for the results of the previous operation before submitting the next one. Often, write operations depend on the results received in previous read operations. To support this application type, traditional database replication protocols [BHG87] use single operations as the units of exchange for coordination. While this provides an elegant framework to prove correctness, it introduces serious overheads in practice. Assuming a typical transaction with about 20 operations [Tra96], a system with 5 sites will then require around 200 messages per transaction (usually pairs of messages are sent), which at say 50 transactions per second, results in a total of 10000 messages per second. Such a protocol does not scale and introduces a significant problem with deadlocks [GHOS96].

To avoid the overhead, we do not send a message per write operation but a single message containing all write operations. Since this write set is not known in advance we have to defer the execution of the single write operations until the end of the transaction, requiring a reordering of the events in the system.

The resulting steps in the execution of a transaction are shown in Figure 2. The first phase takes place at the local backend, where all read operations of a transaction are performed (*read phase*). Depending on the concurrency method applied for read operations, the transaction either acquires read locks before accessing the data or performs the read operations on a snapshot of the data. Write operations are deferred. Once the client submits the commit of the transaction, the entire write set is sent to all sites using the total order broadcast (*send phase*). Once the replication manager informs a backend about the delivery of a write set, the backend requests all locks necessary to per-

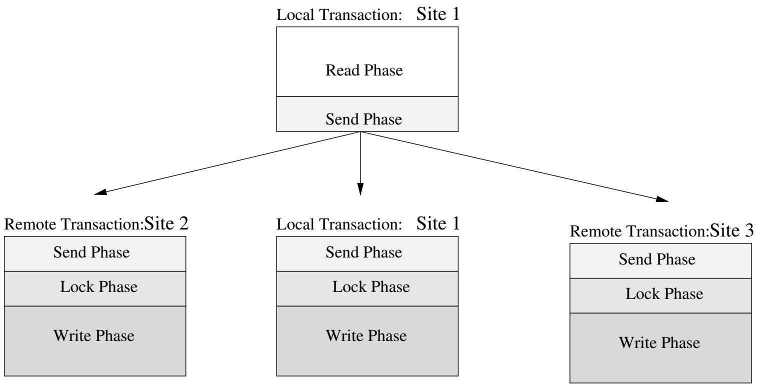

Figure 2. Transaction Processing

form the write operations in an atomic step (*lock phase*) before it starts the execution of the operations (*write phase*). A lock might be granted immediately if no conflicting lock is active, otherwise it is appended to the waiting list. Only after all lock requests are included in the lock table (granted or waiting), the replication manager is allowed to process the next write set. Hence, by correlating the lock phase of the transactions with the delivery of their write sets and the fact that write sets are delivered in total order, conflicting write operations are ordered in the same way at all sites. However, since non-conflicting operations have no conflicts in their locks and since the transaction and data managers of a database are usually decoupled, there are no ordering guarantees on non-conflicting write operations. During the lock phase, read/write conflicts are detected and treated.

This procedure requires to modify the traditional lock and backend management (as implemented in PostgreSQL). Usually, a transaction requests a lock and performs an operation on the object before it requests the next lock. In between the two lock requests, other transactions can also acquire locks. In the replicated environment, the lock management must be enhanced to allow for an atomic request of more than one lock to be able to request all write locks in a single step. This also means that, in contrast to usual transaction processing, a transaction can have more than one lock waiting and more than one lock granted without the corresponding operation on the object being executed, requiring modifications of the backend management.

3.5 Local vs. Global Events: Local Write Operations

Until now, we have simply assumed that write operations are deferred. However, read operations might depend on previous write operations of the same transaction. For instance, a transaction might first insert a couple of records in a table and then want to read them. If the insert is simply delayed, the read operation will not see the new records. If we want to avoid this behavior, the local backend must actually perform modifications at the time they are submitted. However, these modifications should not become visible until after the write set is delivered. To do so, a DRAGON backend modifies shadow copies which the following read operations can access. In practice, this means we have to include local events visible only within a single backend.

3.6 SQL-Statements vs. Messages

In practice, the question of the granularity of events that will be communicated in a message needs to be resolved. As pointed out above, a transaction can be seen as a series of read/write events which are first visible only at the local backend and later on at all sites.

However, clients do not merely submit simple reads or writes but rather higher level operations. Looking at relational database systems [ANS92], SQL statements may access several records of different tables. For instance, the following statement increases the salary of all records of the table `employee` whose salary has been below 2000.

```
UPDATE employee
SET salary = salary + 100
WHERE salary < 2000
```

To process an SQL-statement, PostgreSQL parses the statement and transforms it into an execution tree. In the example above, the execution involves a scan which checks step by step whether each record of the table `employee` fulfills the salary condition. If this is the case, the record is updated. This means that a SQL-statement is usually a sequence of read and write operations.

Now the question is what exactly should be sent to the remote servers. Basically, there exist two alternatives. The first alternative is the write set to contain the logical statements, i.e., SQL-statements like UPDATE, DELETE, IN-

SERT. As a first approach, this is advantageous for two reasons. Firstly, message creation is very simple: the statement string is directly added to the message. Secondly, the size of the message is small. Taking into account that a typical update transaction does not have more than 20 updating statements [Tra96], the message size is only a couple of hundreds bytes. However, the solution has also two major drawbacks. As a first problem, the workload on the nodes is considerably higher since all of them have to process the entire SQL statements, and hence also the read operations of the statement. For instance, the UPDATE statement above contains a considerable search overhead. The second problem refers to the locking granularity. The replica control algorithm requires all locks to be requested before any execution can start. In a general statement like the UPDATE statement above, however, we do not know exactly which records of the table will be updated, since we do not know a priori the specific primary keys (primary key is the attribute of a record that uniquely identifies it). This means, we have to choose a coarser locking granularity covering all possible records, i.e., all tables that are accessed by the statement must be locked.

A second alternative is to transfer update information recordwise. Advantages and disadvantages are exactly the opposite of those of the previous case. In the example above, the local server alone performs the scan through the table to find the matching records since actual computation takes place only at the local site. Once a matching record is found its primary key value is determined. This is then appended to the message along with the new value of the record. The remote servers locate the records through indices (which usually exist for primary keys) and enter the updates directly. Furthermore, the knowledge of the primary keys allows fine granularity record level locking and hence, higher concurrency. On the other hand, when many records are updated, the message size can grow very large. In these cases, we may be better off sending the SQL statement instead of the actual record updates. We are currently evaluating the message cost and the impact on the overall delay and overhead of both strategies before deciding which one to use.

3.7 Fault Tolerance in Group Communication and Database Systems

As a last topic, we want to look at the concepts of fault tolerance of both systems. Distributed systems should be able to continue processing despite of site failures and network partitioning. This means that even in the case of failures and the unavailability of copies the replicated database system has to fulfill transactional atomicity (once a site commits a transaction no other site may abort it), database consistency (the copies of all available sites must stay consistent) and non-blocking behavior (transactions should be able to continue processing). Furthermore, adequate primitives must be provided for the recovery from failures, i.e., the rejoining of sites or the merging of partitions.

In [KA98] we pointed out how different reliability semantics of group communication primitives can directly be used to guarantee atomicity, database consistency and allow for continuous transaction processing on the available sites despite of site failures. In particular, our algorithms turn out to belong to the "read once/ write all *available* copies" approach (which allows to ignore unavailable copies) without any additional work in the database system but simply relying on the delivery guarantees and failure detection mechanisms of the group communication systems. In [KA98], we suggested different usage of the reliability semantics of group communication primitives to provide different degrees of fault-tolerance for the database system. In any case, the provided degrees of fault tolerance are far better than provided by any asynchronous replication protocol.

However, the recovery of a DRAGON server or the joining of a new server is only partially solved by the virtual synchrony approach. It is not enough for an executing site to be informed that another site joined the system. Database replication introduces a state to the system, namely the database. Hence, before becoming operational, the database state must be transferred from an available site to the new site [SR96]. However, copying the database must be coordinated with the processing of concurrent transactions on the available server.

To do so, we assume that a specific server of the group is the peer server providing the copy. The virtual synchrony guarantees that all servers receive group change information (joining of a new site) within the flow of normal messages. Hence, it is easy for the peer server to determine which is the last transaction the new server has not received. The peer server processes all pending transactions up to this transaction but delays all new transactions. Local transactions that are still in their read or send phase are not considered and may continue. After the last commit the server sends the copy of the database to the new server and only then continues processing new messages. The new server installs the database and starts processing the first transaction it received through the broadcast mechanism. At the same time it can start receiving requests of clients.

4 Conclusion and Future Work

This paper presented the architecture of the replicated database system DRAGON. DRAGON is an extension of the object-relational database system PostgreSQL, integrating replication and distribution to provide a fully functional network of replicated database systems. Using group communication as the basic concept for message transfer, DRAGON represents a feasible approach to synchronous replication. The focus of the paper was to discuss various issues related to the combination of group communication primitives and replica control mechanisms.

Aside of completing the implementation, further development goes in two directions. Firstly, we have to provide a suitable interface to replication management. This includes the coordination of the DRAGON servers regarding table creation and deletion, performance monitoring etc. Secondly, we are going to extend the system to provide partial replication. This requires mechanisms to subscribe and unsubscribe to data, a location management to log who has subscribed to which data and a refinement of our protocols to handle partial replication efficiently.

Acknowledgements

We would like to thank A. Schiper, F. Pedone and M. Wiesmann for the many helpful discussions on the DRAGON project.

References

[AAES97] D. Agrawal, G. Alonso, A. El Abbadi, and I. Stanoi. Exploiting atomic broadcast in replicated databases. In *Euro-Par'97*, Passau (Germany), August 1997.

[ANS92] ANSI X3.135-1992. *American National Standard for Information Systems – Database Languages – SQL.* November 1992.

[BBG+95] H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O'Neil, and P. O'Neil. A critique of ANSI SQL isolation levels. In *Proc. of the ACM SIGMOD Int. Conf. on Management of Data*, pages 1–10, June 1995.

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

[BJ87] Kenneth P. Birman and Thomas A. Joseph. Reliable communication in the presence of failure. *ACM Transactions on Computer Systems*, 5(1):47–76, 1987.

[DM96] D. Dolev and D. Malki. The Transis approach to high availability cluster communication. *Communications of the ACM*, 39(4):63–70, April 1996.

[Fel98] P. Felber. *The CORBA Object Group Service: A Service Approach to Object Groups in CORBA*. PhD thesis, École Polytechnique Fédérale de Lausanne, 1998.

[GHOS96] J. Gray, P. Helland, P. O'Neil, and D. Shasha. The dangers of replication and a solution. In *Proc. of the ACM SIGMOD Int. Conf. on Management of Data*, Montreal, Canada, June 1996.

[Gro98] Object Management Group. *The Common Object Request Broker: Architecture and Specification.* OMG, February 1998.

[HT93] V. Hadzilacos and S. Toueg. Fault-tolerant broadcasts and related problems. In S. Mullender, editor, *Distributed Systems*, pages 97–145. Addison-Wesley, 1993.

[KA98] B. Kemme and G. Alonso. A suite of database replication protocols based on group communication primitives. In *Proc. of the Int. Conf. on Distributed Computing Systems*, Amsterdam, The Netherlands, May 1998.

[KPAS99] B. Kemme, F. Pedone, G. Alonso, and A. Schiper. Processing transactions over optimistic atomic broadcast protocols. In *Proc. of the Int. Conf. on Distributed Computing Systems*, Austin, Texas, June 1999. to appear.

[Lam78] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.

[MMSA+96] L. E. Moser, P. M. Melliar-Smith, D. A. Agarwal, R. K. Budhia, and C. A. Lingley-Papadopoulos. Totem: A fault-tolerant multicast group communication system. *Communications of the ACM*, 39(4):54–63, April 1996.

[Ora95] Oracle. *Concurrency Control, Transaction Isolation and Serializability in SQL92 and Oracle7*, 1995. White Paper.

[PGS97] F. Pedone, R. Guerraoui, and A. Schiper. Transaction reordering in replicated databases. In *16th IEEE Symp. on Reliable Distributed Systems (SRDS'97)*, Durham, USA, October 1997.

[PGS98] F. Pedone, R. Guerraoui, and A. Schiper. Exploiting atomic broadcast in replicated databases. In *Proc. of EuroPar*, Southampton (England), September 1998.

[Pos98] PostgreSQL. *v6.4.2 Released*, January 1998. http://www.postgresql.org.

[SR96] A. Schiper and M. Raynal. From group communication to transactions in distributed systems. *Communications of the ACM*, 39(4):84–87, April 1996.

[Tra96] Transaction Processing Performance Council (TPC). *TPC Benchmark C, Standard Specification, Revision 3.4*, February 1996.

[vRBM96] R. van Renesse, K. P. Birman, and S. Maffeis. Horus: A flexible group communication system. *Comm. of the ACM*, 39(4):76–83, April 1996.