Increasing the Reliability of Three-Tier Applications *

W. Zhao, L. E. Moser and P. M. Melliar-Smith
Department of Electrical and Computer Engineering
University of California, Santa Barbara, CA 93106
wenbing@alpha.ece.ucsb.edu, moser@ece.ucsb.edu, pmms@ece.ucsb.edu

Abstract

In this paper we describe an infrastructure that provides increased reliability for three-tier applications, transparently, using commercial off-the-shelf application servers and database systems. In this infrastructure the application servers are actively replicated to protect the business logic processing. Replicating the transaction coordinator renders the two-phase commit protocol non-blocking and, thus, avoids potentially long service disruptions caused by coordinator failure. A thin interpositioning library provides client-side automatic failover, so that clients know the outcome of their requests. The interaction between the application servers and the database servers is handled through replicated gateways that prevent duplicate requests from reaching the database servers. Aborted transactions, caused by process or communication faults, are automatically retried on the client's behalf.

1. Introduction

Three-tier architectures, that are composed of thin clients, application servers and database management systems (DBMS), have become mainstream in business applications. The structure of three-tier architectures matches the logical decomposition (presentation, logic and data) of the applications. The front-end clients provide a user interface for service access. The application servers implement the business logic. The database servers manage data and transactional operations at the back-end. Typically, one or more transactions are initiated by an application server when it receives a request from a client. After processing the request, the resulting state is stored at a back-end database server, and the result is returned to the client. It is critical that the services provided by a three-tier architecture are reliable and continuously available.

* This research has been supported by MURI/AFOSR Contract F49620-00-1-0330 and UC MICRO Grant 00-068.

Transaction processing provides fault tolerance for applications by rolling back the current transaction to a previous consistent state when a fault occurs. This approach works best if simple, local transactions are involved. For distributed transactions, the OMG's Object Transaction Service (OTS) [13], traditional TP monitors, or an equivalent service must be used to coordinate transaction execution and completion. The two-phase commit (2PC) protocol is commonly used for distributed transactions. However, 2PC is a blocking protocol, *i.e.*, the transaction participants can be blocked forever waiting for the transaction coordinator to recover. In practice, timeouts are often used to avoid the indefinite blocking problem; however, when the timeout occurs, heuristic decisions made by the transaction participants might not be the same, putting the integrity of the data in danger.

Another well-known problem for three-tier applications is the "outcome determination" problem. One or more transactions can be initiated by a request from a thin client, but the client usually does not participate directly in the transactions, *e.g.*, a Web browser that acts as a thin client typically does not support distributed transactions. A fault might occur in the middle of a transaction and the transaction will be rolled back, or a fault might occur after the transaction has completed but before the reply is sent to the client. Thus, the client does not know whether the request has been serviced completely, or not at all. Simply retrying the same request is not safe because the client's request might be processed twice, *e.g.*, the client might be charged twice for an order.

Fault tolerance can also be achieved by entity redundancy, *i.e.*, an object or a process can be replicated on distinct processors to mask the failure of one of the replicas from the rest of the system. Roll-forward semantics are provided by such a fault-tolerant system unless there is a total failure (all replicas fail simultaneously).

It is highly desirable to combine replication and transaction processing, so that the business logic processing is protected by replication and the data are protected by transaction processing. With appropriate failover and request

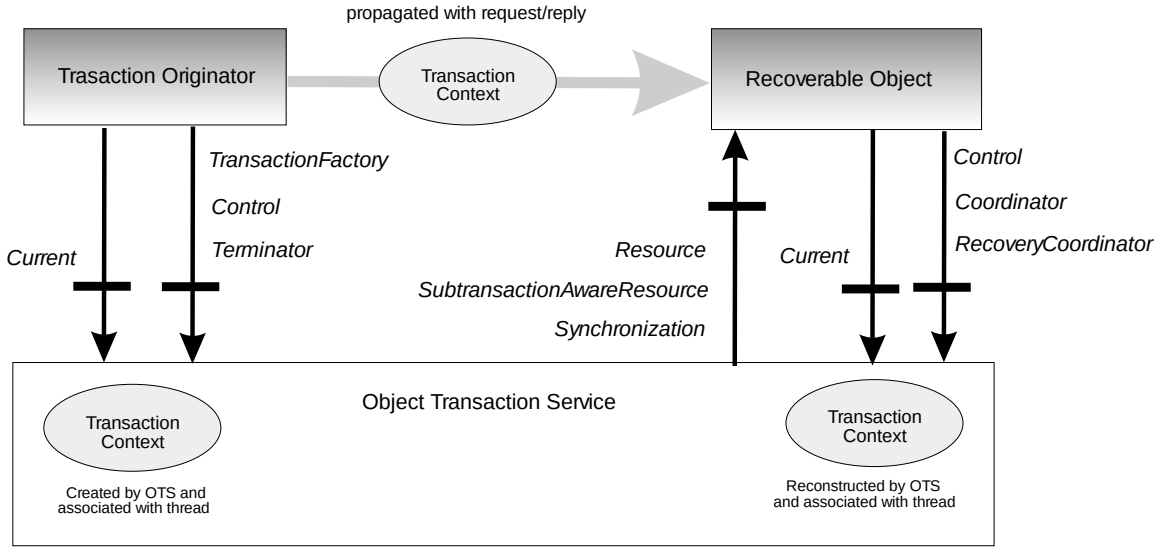


Figure 1. The service architecture and interfaces of the OTS.

retry mechanisms, the outcome determination problem can be eliminated. A naive combination of the two approaches does not make sense, as pointed out in [15], because current replication frameworks lack the necessary support for three-tier transaction processing systems.

In this paper we describe mechanisms that enable the integration of replication and transaction processing in the scope of three-tier architectures. We present an infrastructure that is based on the replication provided by the OMG's Fault Tolerant CORBA (FT CORBA) standard [12] and the transaction processing provided by the OMG's OTS standard [13]. We assume that one or more endpoints to commercial DBMS are available.

We have built a prototype that implements the mechanisms described in this paper to enable the integration of replication and transaction processing. The prototype is based on the OTS implementation from Object Oriented Concepts, Inc [14] and the FT CORBA implementation from Eternal Systems, Inc [9, 10]. The overhead of three-way active replication of application servers and transaction coordinators, compared to the non-replicated case, is approximately 16%.

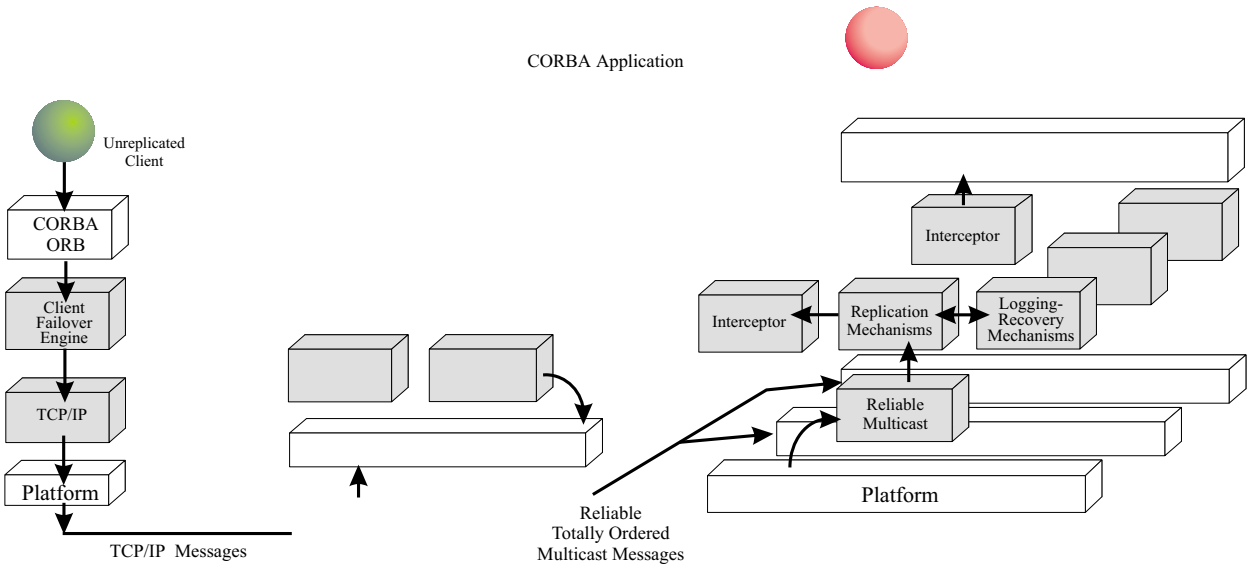
2. Transaction Processing with OTS

The OMG's Object Transaction Service (OTS) [13] provides services and application programming interfaces (APIs) for atomic execution of transactions that span one or multiple objects in a distributed environment. A transaction is composed of a set of activities in terms of CORBA remote invocations and communication with one or more DBMS. The OTS uses the two-phase commit (2PC) protocol for transaction termination. The OTS specifies the following interfaces within the CORBA CosTransaction module: `Current`, `TransactionFactory`, `Control`, `Terminator`, `Coordinator`, `RecoveryCoordinator`, `Resource`, `Synchronization`, and `SubtransactionAwareResource`. The OTS provides services through these interfaces, as shown in Figure 1.

The OTS can be implemented using one of two different approaches. One approach is to implement all of the functionality in one or more libraries so that the transaction management is completely distributed, *i.e.*, there is no centralized TransactionFactory process. The other approach is to use a centralized process (OTS server) to handle transaction management together with a set of client-side libraries. We assume that the second approach is used, because it is conceptually easier to handle. The mechanisms that we have developed based on this model can be adapted quite easily to support the distributed model.

The management of both flat transactions and nested transactions is specified by the OTS, but support is mandatory only for flat transactions. The OTS supports many flexible programming models. Programmers may choose implicit or explicit context management and transaction propagation. It is also capable of importing/exporting transactions from/to traditional TP monitors. Multiple OTS servers can collaborate to commit a distributed transaction. In this paper we consider only flat transactions, only one OTS server and the implicit programming model.

In the implicit programming model the application invokes `Current::begin` to start a transaction. The OTS runtime then invokes the `TransactionFactory` inter-


CORBA Application
Unreplicated
Client
CORBA
ORB
Client
Failover
Engine
TCP/IP
Platform
TCP/IP Messages
Interceptor
Interceptor
Replication
Mechanisms
Logging-
Recovery
Mechanisms
Reliable
Multicast
Platform
Reliable
Totally Ordered
Multicast Messages

The Replication Engine is implemented below the ORB for efficiency and transparency. It consists of the Replication Mechanisms, the Logging-Recovery Mechanisms, and the Totem reliable totally-ordered multicast protocol [8]. The Replication Engine daemon runs on each host that supports a fault-tolerant application. On the server-side, the multicast protocol delivers the messages, in order, to the local Replication Engine, which logs the messages. The Interceptor then delivers the messages to the ORB, as though they were received from TCP/IP. Similarly, the Interceptor diverts the reply messages to the Replication Engine, so that they can be logged and multicast.

The Fault Notifier is implemented as a CORBA object above the ORB and is replicated for fault tolerance. The Fault Notifier is responsible for notifying the registered consumers of faults based on the supplied criteria.

3.3. Supporting Unreplicated Clients

The Client Failover Engine implements the Client Redirection and Transparent Reinvocation Mechanisms specified by the FT CORBA standard. The replicated server publishes its IOGR, which contains the gateway information used for automatic redirection and reinvocation. The Client Failover Engine intercepts all of the messages from the unreplicated client and diverts them to the Gateway. The Gateway acts as an entry point for the unreplicated client to the replicated server, and is passively replicated for fault tolerance.

The Gateway receives requests from the clients through TCP/IP, and multicasts the requests to the replicated server. If the primary gateway fails, the Client Failover Engine automatically reconnects to a backup gateway. For each request, the Client Failover Engine inserts a service context that is composed of a `client_id`, a `retention_id` and an `expiration_time` for the request. The `client_id` and `retention_id` serve as a unique identifier for the client's request and allow the server to recognize that the request is a duplicate of a previous request. The `expiration_time` defines a lower bound on the time within which the server can safely discard the logged request and reply messages.

4. Integrating Transaction Processing and Replication

Many mechanisms defined by the FT CORBA standard are complimentary to transaction processing. In particular, the problem of transaction outcome determination can be readily solved by the Client Redirection and Transparent Reinvocation mechanisms of FT CORBA. However, FT CORBA is defined only in the scope of CORBA object interactions. It does not address issues related to interactions between CORBA objects and non-CORBA systems, such as the DBMS. Furthermore, in FT CORBA the granularity of an operation is a single CORBA invocation. In transaction processing the granularity is a single transaction that is composed of several CORBA invocations and several interactions with the DBMS. It is necessary to keep track of the association of each message with current transactions so that the logging and recovery, including transaction retry, can be done atomically for each transaction.

We discuss below the additional mechanisms that are needed for integrating transaction processing and replication. We focus on replicating the middle tier, *i.e.*, the application servers, using the active replication style. To enable unreplicated clients to use the middle tier services, Gateways between the clients and the application servers are employed, as discussed in Section 3.3.

Here we introduce another type of Gateway between the application servers and the DBMS, as shown in Figure 3. This Gateway functions as a proxy for the database, and is replicated using the passive replication style. The application servers connect to the DBMS through this Gateway. Messages from the application server objects are intercepted and diverted to the Replication Engine, which multicasts them to the Gateway. The Gateway detects and suppresses duplicate messages, and then transmits a single message to the database, using TCP/IP.

The DBMS may or may not be replicated; the topic of database replication is beyond the scope of this paper. However, we assume that the DBMS implements appropriate reliability mechanisms, that it supports the standard XA interface [21] and that it has one or more endpoints open for connections.

4.1. Retrieving and Monitoring Transaction Information

Knowledge of the outstanding transaction status and the association of each message with a transaction is essential for managing a replicated transaction processing system.

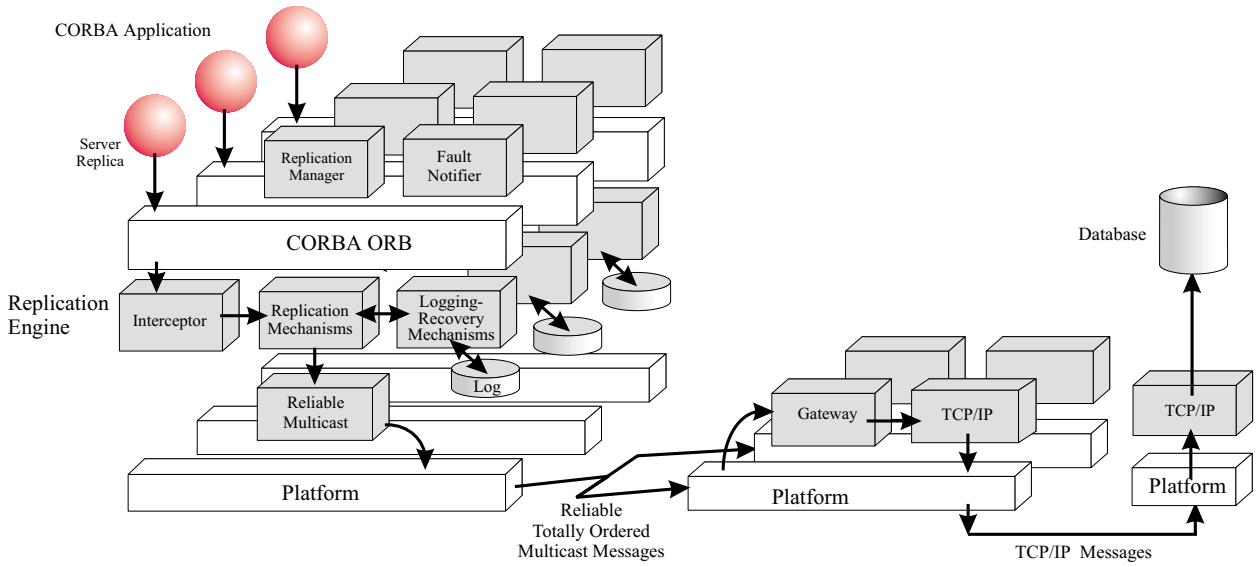


Figure 3. Architecture for access from a replicated application server to a database system.

object keys for the management objects are retrieved from
the reply messages of the `create()` operations.
`Control` object reference.`get`

-

-

-

-

state() and set

_state() on an exist-

_state() to

_state() and set_state() methods. Therefore,to provide

both the start of new transactions and the commit or abort of existing transactions. The Replication Engine holds the status of all current transactions. Messages to the OTS server are logged only during the recovery process and are discarded as soon as they are delivered to the OTS server replica. Starting a new OTS server replica involves the following steps:

1. To indicate the creation of a new OTS server replica, the Replication Engine multicasts a `OTS_RECOVERY_START` message.
2. Upon receiving the `OTS_RECOVERY_START` message, the Replication Engines supporting the existing OTS server replicas start filtering messages for the OTS server and queueing request messages for new transactions, but allow messages for existing transactions to be delivered.
3. The Replication Engine supporting the new OTS server replica discards request messages for existing transactions but queues request messages for new transactions.
4. The Replication Engines supporting the existing OTS server replicas multicast an `OTS_RECOVERY_STOP` message when they detect that all existing transactions have committed or aborted.
5. Upon receiving the `OTS_RECOVERY_STOP` message, the Replication Engines supporting both the existing OTS server replicas and the new OTS server replica start delivering the queued request messages for new transactions.

By using active replication for the transaction coordinator, we render the 2PC protocol non-blocking under fault conditions. The 2PC protocol no longer blocks completion of a transaction as a result of a coordinator failure. A fault at one coordinator replica is masked by the other surviving coordinator replicas.

4.2.3 Logging and Recovery of Transactional Objects

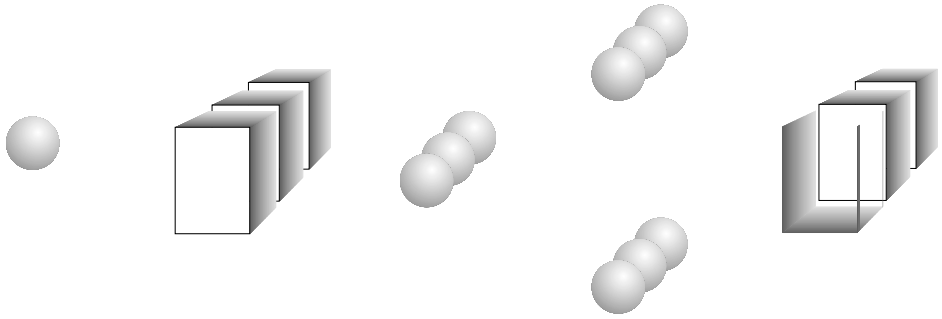
If a transactional object is stateful, it must inherit the `Checkpointable` interface specified by the FT CORBA standard, which provides `get_state()` and `set_state()` methods to transfer the application state. For a transactional object, transferring only application state from an existing object replica to the recovering object replica does not ensure strong replica consistency, because there is also hidden OTS management state that is maintained on behalf of the object. Unfortunately, the `get_state()` and `set_state()` methods are not available for the OTS libraries that hold this state. Consequently, we allow state transfers to occur only when an object reaches a quiescent state, *i.e.*, the object involves no transactions. Thus, to recover a transactional object, we employ the following steps:

1. When the Replication Engine creates a new transactional object replica, it multicasts an `OBJ_RECOVERY_START` message.
2. Upon receiving the `OBJ_RECOVERY_START` message, the Replication Engines that support the existing object replicas start queueing request messages that do not belong to the ongoing transactions associated with the object, while delivering messages associated with current transactions.
3. The Replication Engine that supports the recovering object replica discards request messages for current transactions but queues request messages for new transactions.
4. When all current transactions associated with an object complete, the Replication Engines that support the existing object replicas multicast an `OBJ_RECOVERY_STOP` message. If the object is stateful, they each fabricate a `get_state()` message and deliver it to the existing replicas. After they receive the reply for the `get_state()` message, the Replication Engines fabricate and multicast a `set_state()` message based on the state that the `get_state()` message returns.
5. The Replication Engine that supports the recovering replica delivers the `set_state()` message to the recovering replica. As soon as it sees the reply for the `set_state()` message, the Replication Engine that supports the recovering replica starts delivering the queued messages. The Replication Engines that support the existing replicas can start delivering queued messages as soon as they have multicast the `OBJ_RECOVERY_STOP` message or the `set_state()` message.

Note that the recovery of an OTS server requires a system-wide quiescent state, while the recovery of a transactional object requires only an object-wide quiescent state.

4.3. Automatic Transaction Retry

Certain cases exist for which retrying an aborted transaction on behalf of the application is desirable. For example, the crash of the primary gateway facing the DBMS, or a communication failure between the primary gateway and the DBMS, will cause the DBMS to roll-back all associated transactions if the transactions are not yet prepared. We mask this type of failure from the clients by retrying aborted transactions.



"Start," "SQL" and "Termination" latency is denoted by "Other." The latency for "Other" is due primarily to the round-trip communication between the client and the front-end server object. Most of the communication and processing overhead is due to the underlying reliable totally-ordered multicast protocol [20]. The integrated system incurs about 0.5ms round-trip overhead for synchronous remote invocations with messages of size less than 1KB. The overhead is somewhat less significant in a replicated transaction processing system, because the system does a lot of disk IOs, which are more time-consuming.

It is somewhat surprising to see that the integrated system incurs minimum overhead (less than 2%) for the transaction termination stage. This stage consists of five CORBA synchronous invocations among the front-end server object, the back-end server object and the OTS server, and four XA RPCs between the back-end server object and the database server. Analysis shows that the overhead from the integrated system is offset by eliminating the expensive TCP/IP connection set-up and tear-down for the 2PC in the unreplicated system. The integrated system reuses the local IPC (*i.e.*, Unix stream socket) connection between the replicated application and the Replication Engine.

The end-to-end overhead as seen by the client is about 16% for the integrated system, compared to the non-replicated case. Considering the increased reliability and availablity provided by replication, this overhead is acceptable. The resource utilization of the integrated system is moderate. The integrated system takes about 10-20% CPU time on each node, and consumes about 4 Mbps additional network bandwidth for the rotating token that the reliable totally-ordered multicast protocol uses.

6. Related Work

Much work has been done on improving the reliability of database systems, the third tier in the three-tier architecture [2, 3]. Many commercial DBMS nowadays include parallel or replicated database processing functionalities to meet the demand [18].

There are also a number of fault-tolerant CORBA solutions [1, 4, 5, 6, 7, 11, 17, 19] that the application servers might use. To the best of our knowledge, none of them has considered the interaction with a DBMS in a three-tier architecture.

Recently, Frolund and Guerraoui [16] proposed a suite of protocols to support exactly-once transactions (e-transactions) in three-tier applications that provide an end-to-end fault tolerance solution based on passive replication. The client retries a request until it is eventually committed. They assume that the database servers are replicated using commercial clustering technologies. Their protocols include a 2PC protocol equivalent to distributed transaction completion, but with no-blocking guarantees rendered by passive replication of the transaction coordinator. They have demonstrated low overheads for their approach in a prototype implementation. Our approach differs from their approach in several respects.

In our approach fault tolerance is provided to the application servers in a transparent manner. COTS application servers can be readily used without modification. In contrast, the e-transaction approach requires the application server to implement the protocols and store additional information in the database servers for recovery.

In our approach the application servers are protected by active replication. Failure of a replica does not cause the current transactions to roll-back. In the e-transaction approach the application servers are protected by passive replication. Failure of the primary application server might cause all current transactions to roll-back.

In our approach an aborted transaction is automatically retried on behalf of the client if it is caused by a failure of the Gateway facing the DBMS, or by a communication failure with the DBMS. In the e-transaction approach aborted transactions are retried by the client-side protocol. Retrying a transaction from the client-side can incur additional delay compared to our approach.

7. Conclusion

We have described an infrastructure that transparently enables the integration of replication and transaction processing by replicating the application servers and the transaction coordinators. With our replication and automatic transaction retry mechanisms, the infrastructure guarantees non-blocking of distributed transaction completion and provides roll-forward semantics for business operations as perceived by the clients unless a total failure occurs. Three-tier business applications can achieve higher reliability by incorporating our solution.

References

[1] M. Cukier, J. Ren, C. Sabnis, W. H. Sanders, D. E. Bakken, M. E. Berman, D. A. Karr, and R. Schantz. AQuA: An adaptive architecture that provides dependable distributed objects. In *Proceedings of the IEEE 17th Symposium on Reliable Distributed Systems*, pages 245–253, West Lafayette, IN, October 1998.

[2] D. Agrawal, G. Alonso, A. El Abbadi and I. Stanoi. Exploiting atomic broadcast in replicated databases. In *Proccedings of EuroPar*, pages 496–503, Passau, Germany, September 1997.

[3] F. Pedone and S. Frolund. Pronto: A fast failover protocol for off-the-shelf commercial databases. In *Proceedings of the 19th IEEE Symposium on Reliable Distributed Systems*, pages 176–185, Nuremberg, Germany, October 2000.

[4] 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.

[5] S. Maffeis. Adding group communication and fault tolerance to CORBA. In *Proceedings of the 1995 USENIX Conference on Object-Oriented Technologies*, pages 135–146, Monterey, CA, 1995.

[6] C. Marchetti, M. Mecella, A. Virgillito, and R. Baldoni. An interoperable replication logic for CORBA systems. In *Proceedings of the International Symposium on Distributed Objects and Applications*, pages 7–16, Antwerp, Belgium, September 2000.

[7] G. Morgan, S. Shrivastava, P. Ezhilchelvan, and M. Little. Design and implementation of a CORBA fault-tolerant object group service. In *Proceedings of the Second IFIP WG 6.1 International Working Conference on Distributed Applications and Interoperable Systems*, pages 361–374, Helsinki, Finland, June 1999.

[8] 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.

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

[10] P. Narasimhan, L. E. Moser, and P. M. Melliar-Smith. Strong replica consistency for fault-tolerant CORBA applications. In *Proceedings of the IEEE 6th Workshop on Object-Oriented Real-Time Dependable Systems*, Rome, Italy, January 2001.

[11] B. Natarajan, A. Gokhale, S. Yajnik, and D. C. Schmidt. DOORS: Towards high-performance fault-tolerant CORBA. In *Proceedings of the International Symposium on Distributed Objects and Applications*, pages 39–48, Antwerp, Belgium, September 2000.

[12] Object Management Group. Fault tolerant CORBA (final adopted specification). OMG Technical Committee Document ptc/2000-04-04, April 2000.

[13] Object Management Group. Transaction service specification v1.2 (final draft). OMG Technical Committee Document ptc/2000-11-07, January 2000.

[14] Object Oriented Concepts, Inc. *ORBacus OTS*, 1.0 beta 2 edition, 2000.

[15] S. Frolund and R. Guerraoui. CORBA fault-tolerance: Why it does not add up. In *Proceedings of the 7th IEEE Workshop on Future Trends of Distributed Systems*, pages 229–234, Cape Town, South Africa, December 1999.

[16] S. Frolund and R. Guerraoui. Implementing e-transactions with asynchronous replication. In *Proceedings of the International Conference on Dependable Systems and Networks*, pages 449–458, New York, NY, June 2000.

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

[18] A. Vaysburd. Fault tolerance in three-tier applications: Focusing on the database tier. In *Proceedings of the 18th IEEE Symposium on Reliable Distributed Systems*, pages 322–327, Lausanne, Switzerland, October 1999.

[19] A. Vaysburd and K. Birman. The Maestro approach to building reliable interoperable distributed applications with multiple execution styles. *Theory and Practice of Object Systems*, 4(2):73–80, 1998.

[20] W. Zhao, P. Narasimhan, L. E. Moser and P. M. Melliar-Smith. Experimental evaluation of a fault-tolerant CORBA system. In *Proccedings of the International Conference on Parallel and Distributed Processing Techniques and Applications*, pages 390–396, Las Vegas, NV, June 2001.

[21] X/Open Company Ltd. *Distributed Transaction Processing: The XA Specification*. The Open Group, February 1992.