Unification of Replication and Transaction Processing in Three-Tier Architectures

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 a software infrastructure that unifies replication and transaction processing in three-tier architectures and, thus, provides high availability and fault tolerance for enterprise applications. The infrastructure is based on the Fault Tolerant CORBA and CORBA Object Transaction Service standards, and works with commercial-off-the-shelf application servers and database systems.

The infrastructure replicates the application servers to protect the business logic processing. In addition, it replicates the transaction coordinator, which renders the two-phase commit protocol non-blocking and, thus, avoids potentially long service disruptions caused by coordinator failure. The infrastructure handles the interactions between the application servers and the database servers through replicated gateways that prevent duplicate requests from reaching the database servers. The infrastructure implements client-side automatic failover mechanisms, which guarantees that clients know the outcome of the requests that they have made. The infrastructure starts the transactions at the application servers, and retries aborted transactions, caused by process or communication failures, automatically on the behalf of the clients.

1 Introduction

Many enterprise applications use the Common Object Request Broker Architecture (CORBA) as the middleware bus. Recently, the Object Management Group (OMG) adopted two specifications related to fault tolerance, namely, the Fault Tolerant CORBA (FT CORBA) [13] and the CORBA Object Transaction Service (OTS) [14]. The OTS provides reliability for enterprise applications by providing commits and aborts to protect the consistency of the data even in the presence of faults and, thus ensures that aborted transactions can be retried after a fault. FT CORBA provides increased reliability for enterprise applications by replicating CORBA objects so that, if one of the replicas of an object fails, the surviving replicas can continue the processing of the business logic and can provide continuous service to the clients.

The design of enterprise applications is often based on a logical decomposition into presentation, logic and data, which is readily implemented as a three-tier architecture consisting of clients, application servers and database servers. The front-end provides an easy-to-use interface for the clients, the application servers implement the business logic, and the database servers manage data and transactional operations at the back-end. Within this structure, the application servers use a transaction processing programming model. When an application server receives a client's request, it initiates one or more transactions. When the application server finishes processing the request, it commits the transaction, stores the resulting state at the back-end database, and returns the results to the client.

Transaction processing provides reliability for traditional enterprise applications, but lacks the high reliability that many current and future enterprise applications will require. In the highly distributed and networked environment of such enterprise applications, faults in a transaction processing system can lead to problems. When a transaction coordinator fails, the distributed transaction commit protocol requires the participants to wait for the coordinator to recover, which might be quite a long time. Continued processing, without waiting for the coordinator to recover, can compromise the consistency of the data. Consequently, enterprises are seldom willing to participate in transactions across a communication network with another enterprise.

The "transaction outcome determination" problem can arise if the client does not participate directly in the distributed transaction, which is usually the case for clients interacting across the Internet, including both Web browsers and computer systems of other enterprises. A fault might occur in the middle of a transaction, in which case the transaction will be rolled back, or a fault might occur after the transaction has completed and its results are committed but before the reply is sent back to the client. In both cases, the client receives no reply, and the client does not know whether the request has been serviced completely, or not at all. It is not safe for the client to retry the same request because the request might then be processed twice.

Stronger fault tolerance for enterprise applications can be achieved through the unification of replication and transaction processing. However, existing replication frameworks lack the mechanisms that are needed to support three-tier transaction processing systems [7]. In this paper, we present an infrastructure that unifies replication and transaction processing within a three-tier architecture. The infrastructure uses the replication and recovery provided by FT CORBA to protect the processing of the business logic, and the transaction processing provided by the CORBA OTS to protect the data.

2 Background

2.1 FT CORBA

FT CORBA [13] defines interfaces, policies and services that provide robust support for applications requiring high availability and reliability through CORBA object replication. FT CORBA handles object, process and host crash faults, but not Byzantine or network partitioning faults. In FT CORBA, replicated objects are managed through the object group abstraction. FT CORBA uses an Interoperable Object Group Reference (IOGR) to access an object group. An IOGR contains multiple profiles, each corresponding to a member (replica) in the group, or to a set of gateways providing access to the group. The IOGR enables transparent client reinvocation and redirection to provide unreplicated clients with replication and failure transparency. The FT CORBA standard addresses the three aspects of fault tolerance: replication management, fault management and recovery management. To maintain strong replica consistency, FT CORBA requires that all CORBA objects that are replicated must be deterministic (or rendered deterministic), *i.e.*, for the same input (request), all replicas of an object must produce the same output (reply).

2.2 CORBA OTS

The CORBA Object Transaction Service (OTS) [14] provides services and interfaces for atomic execution of transactions that span one or more objects in a distributed environment. A transaction is composed of a set of activities, defined in terms of CORBA remote method invocations and communication with a database server. For each distributed transaction, the OTS generates a unique transaction identifier, which we refer to subsequently as XID. OTS uses the two-phase commit (2PC) protocol to commit a distributed transaction.

OTS supports a flexible programming model for transaction context management and propagation. All user messages in the scope of a transaction contain a transaction identifier within a transaction context. The transaction context can be managed either directly, by manipulating the `Control` Object and the other OTS service objects associated with the transaction, or indirectly, by using the `Current` object provided by the OTS. The `Current` object hides the interactions with the OTS service objects, and provides a convenient interface for programmers of the OTS to start, suspend, resume and commit a transaction. The transaction context is associated implicitly with the request messages sent by a client involved in a transaction, and is propagated with those messages to remote objects, without the client's direct intervention. The transaction context can also be propagated explicitly to remote objects in the form of parameters of the request. Indirect context management, and implicit propagation, are often referred to as the *implicit programming model* for transaction processing in OTS. Our intention is to support multi-tier applications that use the implicit programming model and to use flat transactions that involve one or more database servers as a back-end through the XA interface [18].

3 Replication and Transaction Processing

The unified infrastructure adopts a non-intrusive integrated approach for fault tolerance to application servers [19]. It uses Totem [9] to provide reliable totally-ordered multicasts. The infrastructure implements all of the mechanisms specified in FT CORBA, including the server-side and client-side fault tolerance mechanisms, and the gateway mechanisms that facilitate communication between the unreplicated and replicated CORBA objects. Many mechanisms that FT CORBA defines are complimentary to transaction processing. In particular, the problem of transaction outcome determination is solved by FT CORBA's client redirection and transparent reinvocation mechanisms.

In addition, the infrastructure augments the existing replication mechanisms, and introduces novel mechanisms to cope with the interactions between replicated transactional objects, and those between replicated transactional objects and the database servers that are outside the fault tolerance domain:

- *Replication mechanisms*. In addition to the typical duplicate detection, suppression and operation scheduling, the replication mechanisms must also detect the creation and completion of a transaction, and the association of each user message with the ongoing transactions. This provides a basic service for other transaction-related mechanisms listed below.

- *Logging and checkpointing mechanisms*. The logging and checkpointing in FT CORBA is rather general. The basic idea is that the state of the application object is checkpointed at some appropriate time, and all subsequent requests and replies are logged. When a new replica is added into the object, the replica can catch up with other existing replicas by applying the checkpoint and replaying the logged messages. When a backup is promoted to be the primary, the logging mechanisms ensure that the new primary continues to provide service to the clients. In non-intrusive FT CORBA implementations, such as Eternal, the checkpoint is carried out when an application object is operational quiescent, *i.e.*, the object is not involved with any ongoing request. This strategy works fine if all remote invocations on the object are independent. However, in transaction processing, there are many remote invocations on many objects of a distributed transaction. The use of an operational quiescence point to perform a checkpoint is not sufficient, because of the need to retrieve and subsequently assign potentially large amount of transaction-related state. The checkpointing of the state of a transactional object must be done when the object is *transactional quiescent* to avoid accessing and manipulating the transaction-related state. Transactional quiescence requires that an application object is operational quiescent, and also that it is not involved in an ongoing transaction. Such knowledge is provided by the augmented replication mechanisms.

- *Recovery mechanisms*. The mechanisms specified by FT CORBA must be augmented to handle the transactional objects.

- *2PC optimization*. Novel mechanisms speed the execution of 2PC by exploiting the reliable totally-ordered broadcast capability of Totem.

- *Gateway mechanisms*. The gateway mechanisms defined in FT CORBA allow unreplicated CORBA clients to access replicated CORBA servers inside the fault tolerance domain. Those mechanisms are not sufficient for replicated transactional objects. Another type of gateway, which we refer to as out-bound gateways, are necessary to provide a single-copy image of the replicated transactional objects to the database servers. Otherwise, the exactly-once semantics of remote invocations cannot be guaranteed.

- *Automatic transaction retry mechanisms*. A distributed transaction can be aborted by the database servers as a result of deadlock avoidance, or it can be caused by communication failure between the transactional objects and the database servers, or failure of the primary out-bound gateway failure itself. In such cases, the transaction is automatically retried, without the client's involvement. This ensures exactly-once invocations from the clients to the replicated transactional objects. Care must be taken to guarantee consistency of the state of the transactional objects. To the best of our knowledge, other exactly-once approaches work only with *stateless* servers.

In the unified infrastructure, the application servers are replicated (Database replication, although an interesting topic, is beyond the scope of this paper). Unreplicated clients access the replicated servers through the in-bound gateways. The in-bound gateways are replicated using passive replication. The out-bound gateways are used between the replicated transactional objects and the database server to prevent duplicate requests from escaping the fault tolerance domain to the database servers. The out-bound gateways are implemented in the process group layer of the Totem group communication system. The transactional processes communicate with the gateway via local interprocess communication channels, and the gateway communicates with the database servers via TCP/IP. An example of a three-tier application running with the unified infrastructure is illustrated in Figure 1.

3.1 Augmented Fault Tolerance Mechanisms

Replication mechanisms. The replication mechanisms monitor the creation/completion of each transaction, keep track of the request/reply messages that belong to the transaction, and parse all requests to, and replies from, the OTS server. For each transaction, they maintain a transaction identifier (XID). For the OTS management objects and application-controlled management objects, they maintain a list of object keys together with their object group identifiers. The mechanisms recognize invocations that are part of a transaction by comparing the object key in a request message with the object keys in the object key list. To indicate the start or stop of a transaction, the infrastructure multicasts special messages to the transactional objects, and updates the transaction tables accordingly.

Logging/recovery mechanisms. An object can be checkpointed only when it is transactional quiescent. The user can specify the checkpoint frequency as guidance for the logging mechanisms; however, the mechanisms might not be able to checkpoint the object at the exact specified frequency because they must wait for, or force, a transactional quiescent state. Forcing a transactional quiescent state means queuing requests that do not belong to the transactions in which the replica is currently involved, until all current transactions have completed. Again, this requires the association of user messages with ongoing transactions.

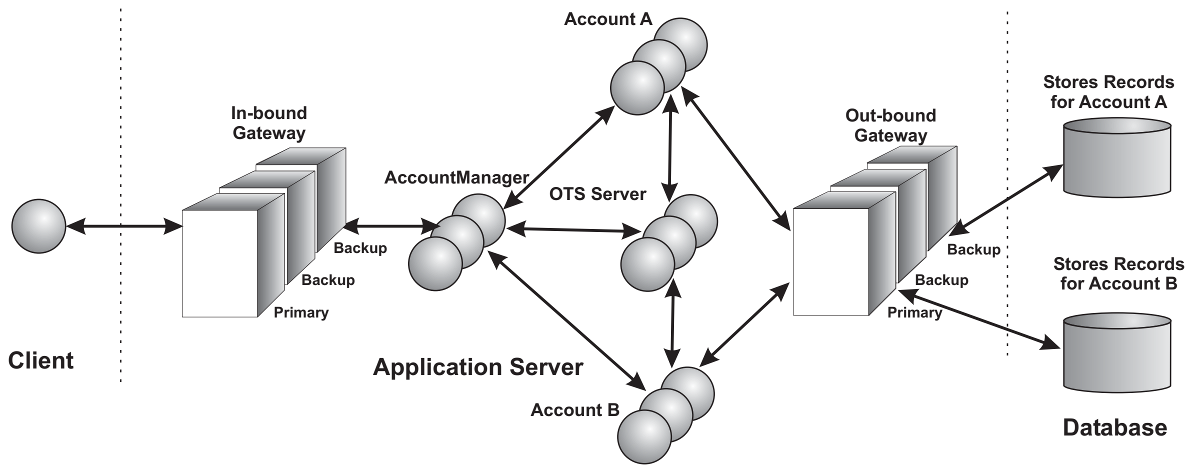


Figure 1. An example of a three-tier application running on top of the unified infrastructure.

All incoming requests and replies for a transactional object following a checkpoint are logged. The logged messages taken before a checkpoint are garbage collected after a new checkpoint is taken.

Checkpointing is performed periodically for all replication styles (active, semi-active and passive replication styles). For active and semi-active replication, although the checkpoint operation can be delayed until a replica is recovered, it is not desirable because it can result in excessive recovery time due to the need for transactional quiescence. Waiting for transactional quiescence blocks the processing of invocations of new transactions for an indefinite period of time that depends on the length of the ongoing transaction. By periodically checkpointing the replica's state, the recovery time can be much less. After the mechanisms transfer the state to the recovering replica, they enable delivery of new invocations to the existing replicas immediately and replay the log at the recovering object while they queue new invocations.

On starting a new replica, or restarting a failed replica, the mechanisms at the new or restarting replica multicast a Recovery_Start message. On receiving the Recovery_Start message, the mechanisms hosting the existing replicas insert the Recovery_Start message into the replicas' logs and later queue incoming messages in the log. When the mechanisms have delivered all of the incoming messages ahead of the Recovery_Start message and the existing replicas are operational quiescent, the mechanisms supporting those replicas multicast the consolidated log to the new or recovering replica and start delivering queued messages.

The mechanisms that host the new or recovering replica start queuing messages after the Recovery_Start message. When they receive the consolidated log from the existing replicas, they extract the checkpoint (if any) and messages from the consolidated log and insert them in the same order into the message queue before the Recovery_Start message. (If the recovering replica is stateless (*e.g.*, the OTS server), there might be no checkpoint in the logs.) If a checkpoint is present in the log, the mechanisms that host the new or recovering replica fabricate a set_state message, based on the checkpoint, and deliver it to the new or recovering replica. If the recovering replica assumes the role of a backup replica, no further messages from the message queue are delivered during recovery.

If the new or recovering replica assumes the role of primary replica, the mechanisms deliver all messages in the log to the replica. They suppress outgoing messages before the Recovery_Start message. If an outgoing message is a request, the reply might be present in the log. If so, the mechanisms deliver the reply to the recovering replica after they suppress the corresponding request.

If an existing (not new and not recovering) backup replica assumes the role of primary replica, the steps are similar but with a few subtle differences. Firstly, the log is already present at the backup replica and the backup replica already has its state initialized to the last checkpoint of the primary. Secondly, the primary replica might not have handled the last incoming request. Therefore, all outgoing messages from the new primary replica, after the last incoming request is delivered, must be multicast to their destinations. If some of the outgoing messages are duplicates, the infrastructure suppresses them at the destinations.

3.2 Optimization of the 2PC Protocol

To simplify the discussion, we assume that all requests/replies are suppressed at the source of the message.

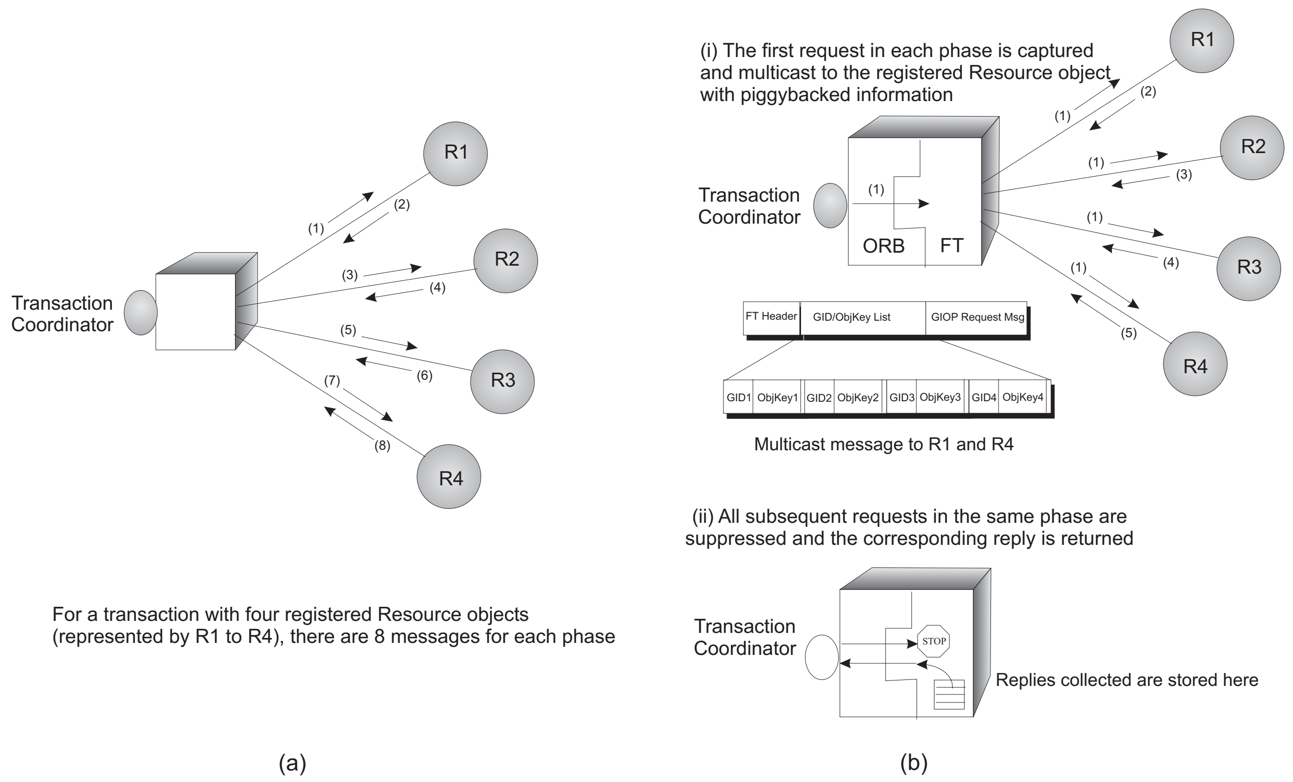

R1
R2
R3
R4
(1)
(2)
(3)
(4)
(5)
(6)
(7)
(8)
Transaction Coordinator
For a transaction with four registered Resource objects (represented by R1 to R4), there are 8 messages for each phase
(a)
(i) The first request in each phase is captured and multicast to the registered Resource object with piggybacked information
ORB
FT
FT Header
GID/ObjKey List
GIOP Request Msg
GID1
ObjKey1
GID2
ObjKey2
GID3
ObjKey3
GID4
ObjKey4
Multicast message to R1 and R4
(ii) All subsequent requests in the same phase are suppressed and the corresponding reply is returned
STOP
Replies collected are stored here
(b)

Figure 2. Steps of the 2PC protocol for $n = 4$ participants (a) without the optimization, and (b) with the optimization. Only one phase of the 2PC protocol is shown in the figure.

In a replicated system, each two-way interaction between two replicated objects is converted into (at least) two multicasts, one for the request and the other for the reply. In the 2PC protocol, assuming that the number of registered Resource objects (transaction participants) is n, the total number of multicasts is reduced from $4n$ to $2n + 2$, by aggregating the multiple request messages from the transaction coordinator to the transaction participants. Figure 2 shows the steps needed to achieve the optimization of the 2PC protocol for $n = 4$ participants. The non-optimized case is also shown. As can be seen, the total number of messages drops from 16 to 10 with the optimization.

As shown in Figure 2, Step (i), the infrastructure piggybacks the list of object group IDs and object keys for all of the Resource objects onto the first request message for each phase of the 2PC protocol, and multicasts the message to all of the Resource objects in the transaction. The infrastructure decides if the message should be delivered to a replica based on the object group ID. Before delivering the message to a replica, the infrastructure replaces the object key that is contained in the request message with the object group ID that corresponds to the object group ID of the receiving group. The infrastructure supporting the transaction coordinator collects responses from the Resource objects and delivers the reply corresponding to the first and subsequent requests from the transaction coordinator. All requests in each phase except the first are suppressed, as shown in Step (ii) in Figure 2.

3.3 Distributed Out-Bound Gateways and Automatic Transaction Retry

Rather than building a single set of replicated out-bound gateways between the replicated application servers and the database system, we designed and implemented distributed out-bound gateways. Unlike the traditional design in which a primary gateway failure can result in the abort and retry of *all* of the ongoing transactions in the replicated system, the failure of one of the distributed gateways aborts only those transactions that go through that particular gateway. This feature significantly increases the scalability and robustness of the infrastructure.

The distributed gateway mechanisms are built into the Totem process group layer. A Totem instance that supports objects that interact with the database system activates the mechanisms when the objects start to communicate with one or more database servers. Rather than letting the replicas connect directly to the database servers, the connections from the objects are routed to the Totem instance that runs on the same host through local IPC. For active replication, one of the replicas in the object group is designated as the primary replica for the purpose of handling the interactions between the replicas and the database server. In the case of semi-active or passive replication, the same primary replica is chosen. The gateway that supports the primary replica, in turn, opens TCP/IP connections to the database servers. The secondary gateways do not establish the connections and do not forward the messages to the database server. The primary gateway forwards the messages received from the database servers back to the primary replica, and multicasts the replies to the other object group members.

When the primary gateway fails, one of the remaining replicas is promoted to be the primary. The fault tolerance mechanisms that support the remaining replicas checks the outstanding transaction list and the status of each transaction on that list.

If a transaction is not yet prepared, the mechanisms shut down the connection with the replica to induce the replica to rollback the transaction because the database server might have aborted the transaction when it detected that the connection was down. In the meantime, the fault tolerance mechanisms broadcast a special message to the transaction originator to request a retry of the aborted transaction. Before retrying the aborted transaction, the states of all transaction objects that are involved in the transaction are reset. This reset is achieved by restoring the last checkpoint at each replica, and replaying the logged messages, up to, but not including, the message that took the object into the transaction. The logged messages within the aborted transaction are discarded. Finally, the message that initiated the transaction is replayed at the transaction initiator.

On the other hand, if a transaction is already prepared, the connections for the transaction remain open and the new primary gateway reestablishes TCP/IP connections to the database servers. The fault tolerance mechanisms (collocated with the transactional object and the transaction coordinator) then reissue the logged request for the second-phase of the 2PC protocol.

4 Performance Measurements

We have developed a prototype of the infrastructure that unifies replication and transaction processing, based on the ORBacus OTS implementation [15] from Object Oriented Concepts, Inc., and our own FT CORBA implementation based on ORBacus's pluggable protocols framework. We used the Oracle 8i Database Management System (DBMS) as the XA resource manager. Our experiments were carried out on six Pentium III PCs over a 100 Mbit/sec local-area network. Each PC is equipped with a single 1GHz CPU and 256 MBbytes of RAM, and runs the Mandrake Linux 7.2 operating system.

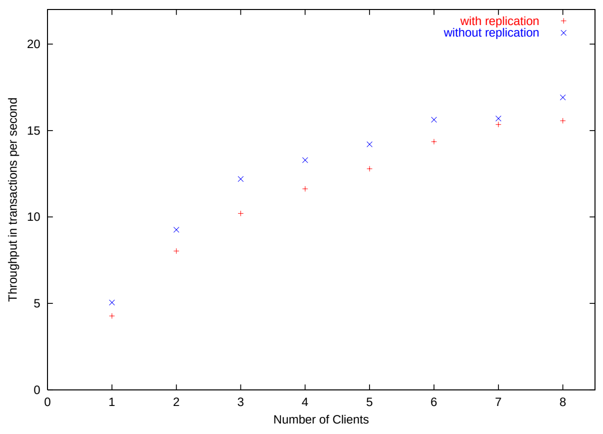


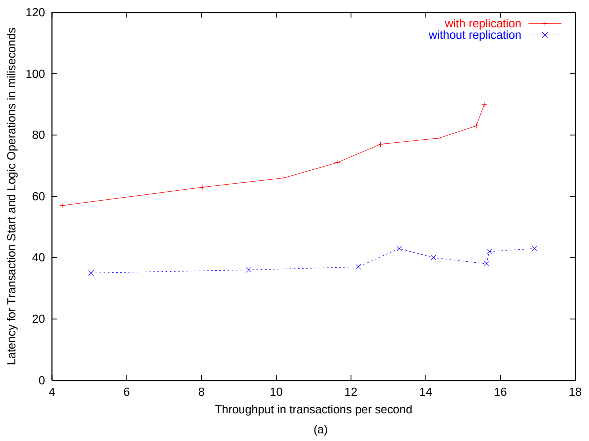
Figure 3. Throughput of the application server, in terms of number of transactions per second, with and without replication.

The experimental three-tier application setup is shown in Figure 1. The clients invoke the replicated server to do a simple bank balance transfer operation. The server then updates the associated tables in the Oracle database management system. The servers are replicated using semi-active replication, because semi-active replication achieves a good balance between the runtime overhead in the fault-free case and the recovery time when a fault occurs.

All server-side processes, including the OTS server, are three-way semi-actively replicated on four of the six PCs. Up to eight unreplicated clients are distributed as evenly as possible on the two remaining PCs. For each run, each client starts and commits a total of 1,000 transactions, one after the other without any delay.

The overall throughput in terms of the number of transactions per second is reduced by only 10% to 20% over the unreplicated case, as shown in Figure 3. Considering the increased availability and reliability provided by replication, this overhead is acceptable. The resource utilization of the unified infrastructure is moderate. The unified infrastructure takes about 10-20% of the CPU time on each node, and consumes about 4 Mbits/sec network bandwidth for the reliable totally-ordered multicast protocol.

In addition to the throughput, we also measured the latency of the following two activities: (1) the start of a transaction and the business logic operations (*i.e.*, bank balance transfer), and (2) the two-phase commit (*i.e.*, the termination of the distributed transaction). The mean latencies of each of the two activities, measured at the clients, for different throughputs, compared to the unreplicated case, are shown in Figure 4.



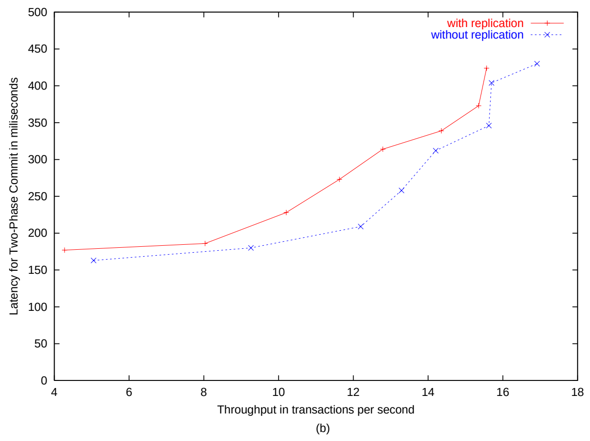


Figure 4. Mean latency for (a) transaction startup and business logic operations, and (b) two-phase commit, for different throughputs, with respect to the unreplicated case.

Figure 4 (a) shows the latency for the first activity, the communication and processing overhead incurred by the unified infrastructure. As the throughput increases, the overhead increases from about 50% to more than 100%. Detailed measurements the cost of the token-based fault tolerance infrastructure can be found in [20].

Figure 4 (b) shows the latency for the second activity, *i.e.*, two-phase commit. The infrastructure overhead becomes relatively small in terms of a percentage of the non-replicated case because the 2PC protocol involves expensive (in terms of latency) disk IO operations. The standard deviation of the measured latencies for this activity, both with and without replication, is quite large, about 10% of the mean latency, probably because of the disk IO.

5 Related Work

Several researchers [2, 3, 6, 10] have investigated object replication and fault tolerance for CORBA prior to the adoption of the Fault Tolerance CORBA standard [13]. Since then, other researchers [8, 11, 12] have developed partial or complete implementations of the Fault Tolerant CORBA standard that the middle-tier application servers might use. To the best of our knowledge, none of the other researchers has addressed the unification of fault tolerance and transactions in a three-tier architecture.

Little and Shrivastava [7] have proposed a high availability solution for CORBA applications written in Java. Their system replicates the application objects to achieve forward progress and provides consistency by means of transactions. They employ a transactional naming service to manage replication and persistent state. Their implementation is based on the CORBA Object Transaction Service but not on the Fault Tolerant CORBA standard.

Frolund and Guerraoui [4] have pointed out the deficiencies of both the Object Transaction Service and Fault Tolerant CORBA as high availability and fault tolerance solutions. They have recognized the difficulties of combining the two services and have proposed a set of proprietary protocols as a solution [5]. Their protocols (called e-transactions) are based on passive replication and guarantee exactly once semantics for transactions. In e-transactions a client retries a request until that request is eventually committed. E-transactions include a 2PC protocol equivalent to distributed transaction completion, but with no-blocking guarantees rendered by passive replication of the transaction coordinator. Commercial clustering techniques are used to replicate the database servers.

Much work has been done on improving the reliability of database systems, the last tier in the three-tier architecture. Vaysburd [17] has given an excellent survey of commercially available packages that provide fault tolerance for the database tier, with respect to such requirements as persistence, data consistency and high availability of service. Several researchers [1, 16] have investigated the use of group communication in combination with transactions for database replication. Database replication is an interesting and important topic, but it is not the focus of this paper.

6 Conclusion

The infrastructure described in this paper transparently unifies replication and transaction processing in three-tier architectures by replicating the application servers and the transaction coordinators. With replication and automatic transaction retry, the infrastructure guarantees non-blocking of distributed transaction completion and provides roll-forward semantics for business operations as perceived by the clients. The infrastructure makes it possible to use the CORBA Object Transaction Service together with Fault Tolerant CORBA to achieve higher availability and reliability for enterprise applications.

References

[1] D. Agrawal, G. Alonso, A. El Abbadi and I. Stanoi, "Exploiting atomic broadcast in replicated databases, *Proceedings of the Third International Euro-Par Conference*, Passau, Germany (September 1997), pp. 496-503.

[2] 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," *Proceedings of the IEEE 17th Symposium on Reliable Distributed Systems*, West Lafayette, IN (October 1998), pp. 245-253.

[3] P. Felber, R. Guerraoui and A. Schiper, "The implementation of a CORBA object group service," *Theory and Practice of Object Systems*, vol. 4, no. 2 (1998), pp. 93-105.

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

[5] S. Frolund and R. Guerraoui, "Implementing e-transactions with asynchronous replication," *Proceedings of the IEEE 2000 International Conference on Dependable Systems and Networks*, New York, NY (June 2000), pp. 449-458.

[6] S. Landis and S. Maffeis, "Building reliable distributed systems with CORBA," *Theory and Practice of Object Systems*, vol. 3, no. 1 (1997), pp. 31-43.

[7] M. C. Little and S. K. Shrivastava, "Implementing high availability CORBA applications with Java," *Proceedings of the IEEE Workshop on Internet Applications*, San Jose, CA (July 1999), pp. 112-119.

[8] C. Marchetti, M. Mecella, A. Virgillito and R. Baldoni, "An interoperable replication logic for CORBA systems," *Proceedings of the International Symposium on Distributed Objects and Applications*, Antwerp, Belgium (September 2000), pp. 7-16.

[9] 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*, vol. 39, no. 4 (April 1996), pp. 54-63.

[10] L. E. Moser, P. M. Melliar-Smith and P. Narasimhan, "Consistent object replication in the Eternal system," *Theory and Practice of Object Systems*, vol. 4, no. 2 (1998), pp. 81-92.

[11] P. Narasimhan, L. E. Moser and P. M. Melliar-Smith, "State synchronization and recovery for strongly consistent replicated CORBA objects," *Proceedings of the IEEE 2001 International Conference on Dependable Systems and Networks*, Goteberg, Sweden (June/July 2001), pp. 261-270.

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

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

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

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

[16] F. Pedone, R. Guerraoui and A. Schiper, "Exploiting atomic broadcast in replicated databases," *Proceedings of the 4th International Euro-Par Conference*, Lecture Notes in Computer Science 1470 (September 1998), pp. 514-520.

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

[18] X/Open Company Ltd. Distributed Transaction Processing: The XA Specification. The Open Group (February 1992).

[19] W. Zhao, L. E. Moser and P. M. Melliar-Smith, "Design and implementation of a pluggable Fault Tolerant CORBA infrastructure," *Proceedings of the International Parallel and Distributed Processing Symposium*, Fort Lauderdale, FL (April 2002).

[20] W. Zhao, L. E. Moser and P. M. Melliar-Smith, "End-to-end latency of a fault-tolerant CORBA system," *Proceedings of the IEEE International Symposium on Object-Oriented and Real-time Distributed Computing*, Washington, DC (April 2002).