Unification of Transactions and Replication in Three-Tier Architectures Based on CORBA

Wenbing Zhao, L. E. Moser and P. M. Melliar-Smith *

Index Terms: Fault tolerance, transaction processing, replication, CORBA, three-tier architectures

Abstract: In this paper we describe a software infrastructure that unifies transactions and replication in three-tier architectures, and provides data consistency and high availability for enterprise applications. The infrastructure uses transactions based on the CORBA Object Transaction Service to protect the application data in databases on stable storage, using a roll-backward recovery strategy, and replication based on the Fault Tolerant CORBA standard to protect the middle-tier servers, using a roll-forward recovery strategy. The infrastructure replicates the middle-tier servers to protect the application 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 failure of the coordinator. The infrastructure handles the interactions between the replicated middle-tier servers and the database servers through replicated gateways that prevent duplicate requests from reaching the database servers. It implements automatic client-side failover mechanisms, which guarantee that clients know the outcome of the requests that they have made, and retries aborted transactions automatically on behalf of the clients.

*W. Zhao is with the Department of Electrical and Computer Engineering, Cleveland State University, Cleveland, OH 44115. E-mail: wenbing@ieee.org. L.E. Moser and P.M. Melliar-Snith are with the Department of Electrical and Computer Engineering, University of California, Santa Barbara, Santa Barbara, CA 93106. E-mail: moser, pmms@ece.ucsb.edu.

1 Introduction

Enterprise applications are typically implemented as three-tier architectures that consist of clients in the front tier, servers that perform the application business logic processing in the middle tier, and databases that store the application data in the back-end tier. An example of such an enterprise application is an online banking application, where a customer accesses his bank accounts over the Internet through a Web browser, the client. After appropriate authentication, the customer is authorized to view the balances in his bank accounts, and transfers money from one account to another (say, from a checking account to a savings account), using the middle-tier servers. The two accounts are managed by two different database servers, and the fund transfer is executed as a distributed transaction.

In such architectures, the Common Object Request Broker Architecture (CORBA) is often used as the middleware bus. Indeed, the Enterprise Java Beans/Java 2 Enterprise Edition (EJB/J2EE) standard derives from the CORBA standard, and mandates the use of CORBA's Internet Inter-ORB Protocol (IIOP). The CORBA Object Transaction Service (OTS) [26] provides data consistency through atomic commitment of distributed transactions and, thus, protects the application data against faults. The Fault Tolerant CORBA (FT CORBA) standard [25] provides high availability by replicating the application objects and, thus, protects the application objects against faults. If one of the replicas fails, the surviving replicas continue the business logic processing and provide continuous service to the clients.

1.1 Problems with Existing Three-Tier Architectures

First, we consider some of the key problems that must be solved to provide both data consistency and high availability for enterprise applications, using the above online banking application as an example.

Traditional transaction processing based on the two-phase commit (2PC) protocol provides

data consistency for enterprise applications, but lacks the strong protection mechanisms that many current and future distributed enterprise applications will require.

If the transaction coordinator becomes faulty and all of the participants (*i.e.*, the XA resources representing the different accounts) in a transaction have voted to commit but have not received a commit message from the coordinator, the 2PC protocol requires the participants to wait for the coordinator to recover, which might take quite a long time. By the time the coordinator recovers, the Web browser might have timed out the server and the customer would see a "page-not-available" page instead of the expected "transfer-succeeded" page. The problems created by failure of the coordinator are three-fold: (1) The customer wastes his time in waiting to see the outcome of the fund transfer transaction. (2) The customer is not notified of the outcome of the transaction. To discover the outcome, he would have to wait until the server is available, re-login, and view his account balance to see whether the previous fund transfer request had been executed successfully. (3) Because resources are locked by the transaction, other transactions that need to access those resources would also be delayed.

Although practical implementations of the 2PC protocol allow a participant to make heuristic decisions regarding the outcome of a transaction, continued processing without waiting for the coordinator to recover can compromise the consistency of the data. Even though such inconsistencies can be addressed by the applications, in principle, the solutions are usually provided in ad-hoc manners and are expensive to design, implement and maintain. A three-phase commit (3PC) protocol [33] can reduce the probability of a hiatus but with substantial cost in overhead; thus, such a protocol is seldom used in practical systems. However, if a small subset of the participants implement the functionality of a replicated transaction coordinator, the cost of the traditional 3PC protocol can be reduced [13].

If the middle-tier server that performs the business logic processing becomes faulty, the transaction in which the server is involved might have to be aborted. In the online banking application, the

customer would then see a “transfer-failed” page and he would have to make a new fund-transfer request. This situation is not as bad as the previous one, but still is unpleasant for the customer. Such a fault would result in lost processing for the server and wasted time for the customer.

1.2 Benefits of Unifying Transactions and Replication

By employing both transactions and replication in a three-tier architecture, it is possible to achieve stronger fault tolerance for enterprise applications [8, 10, 11, 20, 29, 36]. In particular, by replicating the transaction coordinator, the 2PC protocol can be rendered non-blocking [15] and exactly-once semantics can be provided for the clients’ invocations. Moreover, by replicating the middle-tier servers and using transparent transaction retry, roll-forward recovery can be provided. Consequently, a higher degree of abstraction and transparency can be achieved, so that the unpleasant scenarios mentioned above can be avoided, with lower cost and in an application-independent manner.

Existing commercial transaction processing systems do not provide replication of the application logic, and existing replication infrastructures do not support three-tier transaction processing. Work has been done on replication of databases [1, 16], the third tier in three-tier architectures, but that is not the focus of this paper.

There are several ways to unify transactions and replication. For example, to provide the exactly-once guarantee, the e-transactions approach [11] introduces a set of protocols that explicitly combine replication with distributed transaction commitment and recovery. In principle, such protocols can be implemented in the middleware layer and can operate transparently to the applications. However, replacement of, or extensive modification to, existing transaction monitoring middleware might be required. We favor a more transparent approach, so that minimal changes to the application programs are required and commercial-off-the-shelf software can be used.

The unified infrastructure that we have developed uses the transaction processing of the CORBA

OTS to protect the application data, and the replication and recovery of FT CORBA to protect the business logic processing. The infrastructure provides strong replica consistency transparently to the applications and, thus, simplifies the programming of the applications. No modifications of the applications (other than those mandated by the CORBA OTS and FT CORBA standards) are required. To the best of our knowledge, our unified infrastructure is the first such infrastructure to combine the two standards seamlessly and, thus, render three-tier applications fault tolerant by providing both data consistency and high availability.

1.3 Challenges in Unifying Transactions and Replication

Next we discuss some of the challenges that one must face in designing and implementing an infrastructure that provides fault tolerance for enterprise applications by unifying transactions and replication.

The infrastructure must enable as much application concurrency as possible. Serializing all incoming requests is not a practical solution for enterprise applications that involve multiple clients and multiple connections.

For roll-forward recovery, the infrastructure must know the relationships between the clients' invocations for the purpose of logging and replay. The infrastructure must be aware of the status of each middle-tier server (including the transaction coordinator) to take a checkpoint, introduce a new replica, and restore the replica from the checkpoint. To retry a failed transaction transparently, the infrastructure must properly reset the state of each middle-tier server involved in a transaction to the state it had immediately before the start of the transaction and it must properly coordinate the replay of messages.

Even though the state of the CORBA objects in the middle-tier servers can be obtained through standard interfaces, there exists state associated with a process in other forms that tends to be hidden and overlooked. When restarting a failed replica, or adding a new replica, the state of the

new or restarted replica cannot be fully restored if only the states of the application objects are checkpointed. Examples of hidden state include the state of the middleware and the state of the third-party libraries. It is difficult to identify, capture and restore such hidden state.

Appropriate inbound and outbound gateway mechanisms must be designed and implemented, so that the replicated middle-tier servers can interact with non-replicated clients and database servers. Failure of the gateways between the middle-tier servers and the database servers might cause the abort and retry of transactions, both because the TCP connections are lost and because the progress of communication with the database is uncertain.

2 Background

We provide here a brief synopsis of the CORBA Object Transaction Service (OTS) [26] and the Fault Tolerant (FT) CORBA [25] standards on which our fault tolerance infrastructure is based.

2.1 CORBA OTS

The CORBA OTS provides interfaces and services for atomic execution of transactions that span one or more objects in a distributed environment. A distributed transaction consists of CORBA remote method invocations of middle-tier objects and communications with database servers. For each distributed transaction, the OTS generates a unique transaction identifier, which we refer to as the XID. The OTS uses the two-phase commit (2PC) protocol to commit a distributed transaction.

The OTS supports a flexible programming model for transaction context management and propagation. The transaction context is 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 transaction context is associated implicitly with the request messages sent by the client involved in the transaction, and is propagated in those messages to remote objects without the client's intervention. In the OTS, indirect context management and implicit propagation are referred to as the *implicit programming model* for transaction processing.

2.2 FT CORBA

The FT CORBA standard defines interfaces, policies and services that provide robust support for applications requiring high availability and fault tolerance through object replication. The standard provides fault tolerance for object, process and host crash faults, but not Byzantine or network partitioning faults.

FT CORBA addresses three aspects of fault tolerance: replication management, fault management and recovery management. In FT CORBA, replicated objects are managed through the object group abstraction. To maintain strong replica consistency, FT CORBA requires objects that are replicated to be deterministic, or rendered deterministic by the fault tolerance infrastructure, *i.e.,* for the same input (request) all replicas of an object must produce the same output (reply). FT CORBA introduces the notion of *fault tolerance domain*, which consists of multiple hosts, a single replication manager object group, and many application object groups.

The FT CORBA standard defines how clients, both replicated and unreplicated, interact with replicated server objects, so that a client is not disrupted by failover from one server replica to another. In particular, each client request message contains a unique identifier. If a delay or hiatus occurs, a client can retransmit its request message, and the server object recognizes the retransmitted message as a duplicate of a request that it has already processed, discards it, and retransmits the reply corresponding to the original request. Similarly, clients recognize and discard duplicate replies.

The FT CORBA standard allows the applications to choose either application-controlled or infrastructure-controlled consistency and membership styles. If application-controlled consistency is chosen, the application is responsible for checkpointing, logging, activation and recovery, and for maintaining whatever kind of consistency is appropriate for the application. If infrastructure-controlled consistency is used, the fault tolerance infrastructure provides the aforementioned services and maintains strong replica consistency. If infrastructure-controlled membership is used, the

infrastructure automatically chooses an appropriate host to launch a new replica, when the number of replicas falls below the specified minimum number of replicas and ensures that the new replica is synchronized with the existing replicas.

The FT CORBA standard provides a state transfer mechanism and the corresponding interfaces to facilitate adding a new or recovered replica to a group. To make its state available for the infrastructure to retrieve and restore, an object must implement the `get_state()` and `set_state()` methods specified in the `FT::Checkpointable` interface.

Neither the Fault Tolerant CORBA standard nor the XA database standard define how replicated middle-tier servers and database servers interact in the presence of replication and failover.

3 The Model

We now present the model within which our fault tolerance infrastructure operates, including the distributed system model, fault model and programming model. We also discuss the services provided by the infrastructure.

3.1 Distributed System Model

To circumvent the impossibility results related to reaching agreement (*e.g.*, group membership, totally-ordered multicast) in asynchronous distributed systems where processes may crash, we assume an an asynchronous system augmented with (unreliable) failure detectors [5]. We assume that the network does not partition.

The distributed system supports three-tier enterprise applications that comprise clients, middle-tier servers and database servers. The front tier acts as an interface for the clients, the middle-tier servers perform the application business logic processing, and the database servers manage data and transactional operations in the back-end tier. A *middle-tier server* comprises one or more CORBA objects that invoke methods of each other, either locally or remotely via messages sent over a network in typical Remote Procedure Call (RPC) fashion.

Because of our focus on unifying transactions and replication, we refer to a CORBA object that is involved in a transaction as a *transactional object*. Sometimes it is necessary to consider processes, rather than objects, and we refer to a process that is involved in a transaction as a *transactional process*.

The multicast group communication system that we employ operates over a local-area network; however, nothing else in the architecture or its implementation requires that the processors are located on the same local-area network. The multicast group communication system provides a reliable totally-ordered multicast with agreed and safe delivery [22]. For efficiency reasons, we choose to use agreed delivery.

3.2 Fault Model

We assume a crash fault model in which objects, processes and processors perform correctly prior to becoming faulty, and perform no operations thereafter. We assume that commission faults and network partitioning faults do not occur. Faults in distinct processors are assumed to be independent. A fault can affect one or more objects in a process or all of the objects hosted by a processor. Multiple faults can occur, and additional faults can occur during recovery. Our replication and recovery mechanisms require that at least one replica in each group is operational, *i.e.*, catastrophic faults do not occur. However, if a catastrophic fault occurs, traditional transactional rollback recovery from data in stable storage is employed.

3.3 Programming Model

We assume that the middle-tier servers use a distributed transaction processing programming model. When a middle-tier server receives a client's request, it initiates one or more distributed transactions. When it finishes processing the request, the middle-tier server commits the transaction, stores the resulting state in the back-end database, and returns the result to the client. The OTS transaction coordinators handle all transactions started by the middle-tier servers; no transaction is

started by an external transaction manager. The middle-tier servers use the implicit programming model provided by the OTS. They use flat transactions [12] with only one layer of application control, rather than multiple nested layers within a hierarchy. The transactions involve one or more database servers at the back-end through the XA distributed transaction processing interface [34].

A pool of threads is dedicated for communication between a transactional process and a database server. The threads are created during initialization of the transactional process. Each thread manages a single connection to the database server, and is associated with one transaction at a time. Using the connection, the thread performs a number of remote procedure calls with the database server, but it has at most one request outstanding at a time (*i.e.,* it makes a synchronous request on which it blocks waiting for the reply from the database server before it makes another request).

The concurrency control algorithm of the database server determines the serialization of concurrent requests, and our fault tolerance infrastructure respects that ordering. If either the database server or the infrastructure cannot complete a transaction, whether because of concurrency control or because of a fault, the transaction is aborted.

3.4 Services Provided

Our fault tolerance infrastructure enables the middle-tier server applications to provide a highly available service to the clients, a service that continues without disrupting the clients, even when a replica of a middle-tier server fails. The clients can invoke the methods of a middle-tier server substantially as they would invoke the methods of a non-fault-tolerant middle-tier server. The clients are unaware of the three-tier architecture and the database, and of faults in the middle-tier servers and the gateways.

The infrastructure protects the business logic processing in the middle-tier servers, using the replication and recovery defined by FT CORBA with the infrastructure-controlled consistency and membership styles. It supports the standard distributed transaction processing model used by the

CORBA OTS, and protects the application data in databases on stable storage to guard against catastrophic faults. In addition, it renders the two-phase commit protocol non-blocking by replicating the transaction coordinator and, thus, avoids potentially long service disruptions caused by failure of the coordinator.

The infrastructure guarantees that clients know the outcome of the requests that they have made. It handles the interactions between the replicated middle-tier servers and the database servers through replicated gateways that prevent duplicate requests from reaching the database system. It automatically retries aborted transactions on behalf of the clients unless the application itself does so.

The infrastructure uses the standard XA interface and protocol over TCP/IP connections to invoke the database system. However, the transactions are more robust, because the transaction coordinator is replicated. No modifications to the database servers are required. The infrastructure maintains the transactional recoverability of the middle-tier servers and their data in the database. In particular, it ensures that the persistent data created by the replicas of the transaction coordinator do not overwrite each other, which is critical for restoring the coordinator to a correct state after a catastrophic fault.

4 The Fault Tolerance Infrastructure

Our fault tolerance infrastructure, shown in Figure 1, consists of the Totem group communication system [22], the replication, logging and recovery mechanisms, the inbound and outbound gateways at the boundaries of the fault tolerance domain, the client-side fault tolerance mechanisms, and the application-level replication manager and fault notifier. The external clients and the middle-tier servers communicate through the inbound gateways, and the middle-tier servers and the database servers communicate through the outbound gateways. The infrastructure does not replicate the external clients or the database servers; however, such replication is not precluded.

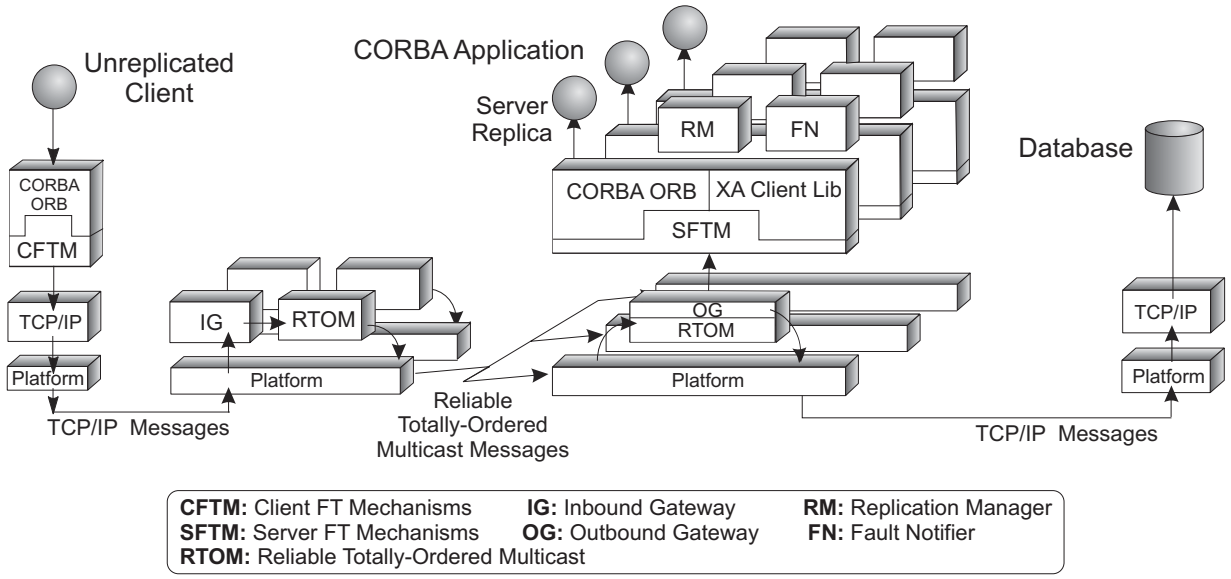

Figure 1: The fault tolerance infrastructure that unifies transactions and replication.

Some commercial database systems provide fault protection and recovery mechanisms internally.

The infrastructure uses a non-intrusive approach to fault tolerance for CORBA by exploiting the pluggable protocols framework [18] provided by many CORBA Object Request Brokers (ORBs) and, thus, differs from the Eternal system [23, 24], which does not use the pluggable protocols framework. More importantly, unlike the Eternal system, the infrastructure unifies transactions and replication and, thus, provides both data consistency and high availability for three-tier enterprise applications.

4.1 Replication of Transactional Objects

To replicate transactional objects consistently, the fault tolerance infrastructure maintains the association of request and reply messages with ongoing transactions. Moreover, the infrastructure copes with the additional complexity introduced by the transaction processing middleware and application objects. Examples of such complexity are multithreading and input (output) from (to) disk.

Association of Messages with Transactions

The fault tolerance infrastructure monitors the status of each transaction by keeping track of request and reply messages within the transaction, and by parsing all requests to, and responses from, the transaction coordinator. For each distributed transaction, the infrastructure maintains a transaction identifier (XID). For both the OTS and the application-controlled management objects, it maintains a list of object keys together with their object group identifiers. The infrastructure recognizes 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 end of a transaction, the infrastructure multicasts control messages to the transactional objects, and updates the transaction tables accordingly.

Replication of the Transaction Coordinator

The fault tolerance infrastructure replicates the transaction coordinator of the CORBA OTS. Thus, if one replica of the coordinator fails, the other replicas continue processing the transaction without disruption to the client. Protecting the transaction coordinator against faults, by replication, reduces the risk that a transaction becomes blocked if the coordinator fails. Continued operation of the transaction coordinator depends on data held locally by the replicas of the coordinator, rather than on persistent data held in stable storage. Note that, unlike [13, 33], the replicas of the transaction coordinator are distinct from the participants in the transaction.

To ensure that the persistent data are not compromised by replication of the transaction coordinator, the disk-write operations carried out by the replicas of the coordinator must be synchronized. For each transaction, a coordinator replica writes the persistent data to stable storage only when it has received acknowledgments from all of the other coordinator replicas that they have received ”yes” votes from all of the participants and the coordinator replica has made its decision.

In principle, uniform totally-ordered multicast [32], or equivalently safe delivery in the absence of network partitioning, should be used for replica synchronization and recovery when there are

visible side-effects of operations resulting from message delivery, such as disk-write operations. However, safe delivery significantly impacts the latency of message delivery and, therefore, we do not use it in our infrastructure. Rather, we implement functionally similar mechanisms, such as those discussed above for synchronization of disk-write operations, that delay the side-effects until all operational replicas reach the same stage of the computation. A drawback of this approach is that it is necessary to identify the places where such synchronization mechanisms are needed, and to design and implement appropriate mechanisms.

Concurrency and Message Scheduling

Transaction processing in the middle tier is unavoidably multithreaded, which is a source of replica non-determinism. One strategy for sanitizing or masking such non-determinism is to employ a message scheduler that forces a multithreaded application into running under a single thread of control by serializing all incoming requests at each replica [24]. A forced logical single thread of control strategy not only decreases the performance but also can cause deadlock if two transactional objects send nested invocations to each other concurrently.

Another message scheduling strategy for transactional multithreaded applications [14] uses a special-purpose proprietary threading library, called *Transactional Drago* [28], that controls the input message buffers and the creation, deletion and scheduling of threads. When all threads are blocked or there are no running threads, the scheduler delivers a message to its handler thread. That strategy has the advantage that actively replicated processes can perform multithreaded operations without extra intra-group communication to coordinate the deterministic processing of the replicas. However, it does not address or consider the problem of integrating transactions and replication in standard middleware or in an industrial system, as our infrastructure does.

We have devised a deterministic message scheduling algorithm for multithreaded applications that is similar, in principle and in the degree of concurrent processing achieved, to the algorithm in [14]. It is implemented transparently to both the applications and the middleware, and works with

the standard pthreads library. Due to space limitations, we provide here only a brief description of the deterministic scheduling algorithm; more details can be found in [37].

For our deterministic message scheduling algorithm, we assume that invocations that belong to the same transaction are serialized. If all of the threads in a replica are blocked waiting for new requests, the replica is *quiescent*. If all of the threads in a replica are blocked waiting for either requests or replies, the replica is *blocked*. A thread can be blocked because it is waiting for a request or a reply after issuing a nested request.

Starting from a quiescent state, the delivery of a new request activates a thread in the replica and the replica becomes unblocked. When the replica issues a nested request or a reply to a previous request, the replica becomes blocked. Because no thread is active, it is safe to deliver the next request to the replica as soon as the replica is blocked. The delivery of a new request can unblock another thread and, therefore, the replica can continue processing in a new thread while the previous thread is blocked waiting.

The algorithm is complicated by the fact that a replica might receive a non-transactional invocation, or some operation might acquire a lock and not release it until the nested invocations have finished executing. The algorithm is conservative, and waits to deliver a non-transactional request message until a quiescent point is reached. Moreover, before it delivers the next request, the algorithm waits until the corresponding reply for a nested invocation, made within a critical section, is received and delivered. To increase the degree of concurrency further, the messages and mutex operations could be scheduled as conflicting operations using the techniques described in [14].

4.2 Checkpointing and Logging

Checkpointing and logging are needed to recover from a fault and to bring a new or restarted replica into the system. A checkpoint is taken by invoking the `get_state()` method, defined by the Fault Tolerant CORBA standard, on one of the operational replicas. The value returned by

`get_state()` is the checkpoint, which is buffered locally and transmitted subsequently to the new or restarted replica. The operational replica cannot service other requests while it is executing the `get_state()` method; however, it can resume processing other requests immediately after execution of the `get_state()` method.

In an ideal world, where the middleware is stateless, only the application objects need to be checkpointed. However, in practice, the middleware, including the fault tolerance infrastructure, that supports the application objects constitutes a significant amount of state. To recover a replica fully, the middleware-related state of the new replica must be made consistent with that of the existing replicas.

The state related to the transport layer in the middleware is fully controlled by the fault tolerance infrastructure due to our plug-in approach. For other kinds of state that are in general difficult to access, we checkpoint a replica when it is transactional quiescent and such state is essentially nonexistent, as described below. There exist other approaches that are transparent to the applications and middleware, such as the user-level checkpointing library developed by Dieter and Lumpp [6]. The drawbacks of such approaches include large checkpoints and strong operating system dependencies.

Service State

In addition to the application object state, ORB/POA state and fault tolerance infrastructure state identified in [24], we recognize a fourth kind of state, which we call *service state*.

In CORBA, an application object might be supported not only by an ORB/POA but also by objects that implement Common Object Services, such as the OTS. Furthermore, an application object might depend on third-party libraries, such as the Oracle client-side library for remote SQL and XA operations. The service objects and third-party libraries are not stateless but have service state. In general, most service state is hidden from, and cannot be directly manipulated by, a transactional object, in the implicit programming model that we support.

Transactional Quiescence

We take a full checkpoint of a replica when it reaches a *transactional quiescent* point (*i.e.*, the replica is not involved in any processing or ongoing transactions), because then the ORB/POA state, fault tolerance infrastructure state and service state are minimized. Our fault tolerance infrastructure maintains a table of ongoing transactions in which each replica is currently involved. A replica is transactional quiescent when that table becomes empty.

Because a replica might continuously receive new invocations for an extended period of time, sometimes it is necessary to force transactional quiescence to avoid an excessively long log and recovery time. To force transactional quiescence, our infrastructure queues requests that start a new transaction, or that belong to a transaction different from the transaction in which the replica is currently involved. Note that forcing transactional quiescence reduces system throughput.

Logging Mechanisms

The logging mechanisms of our fault tolerance infrastructure store messages and checkpoints in volatile memory in the same address space as the replica process. This kind of logging is not to be confused with the logging that is used in enterprise database systems, where logs are stored in stable storage on disk.

The message log contains one or more checkpoints that are interleaved with incoming requests and replies. When a transaction commits, the request and reply messages for that transaction are retained in the log. When a transaction aborts, the logged messages for the aborted transaction are removed from the log.

The fault tolerance infrastructure checkpoints a replica periodically when the replica is transactional quiescent, and logs subsequent incoming request and reply messages. If transactional quiescence does not naturally occur frequently enough, the infrastructure forces transactional quiescence, as mentioned above.

When the infrastructure takes a new checkpoint, it garbage collects all previously logged mes-

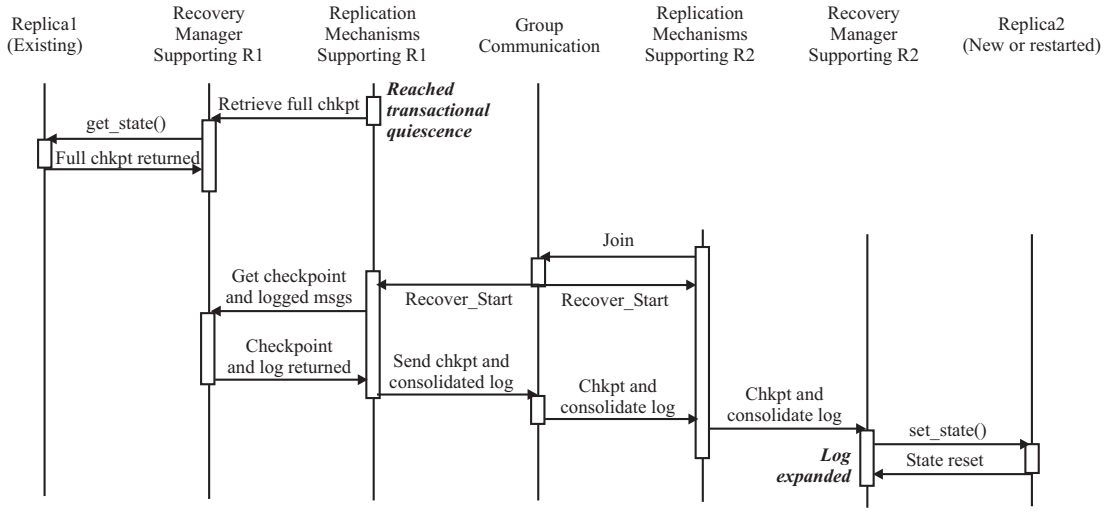

Figure 2: Sequence diagram of the recovery of a replica.

sages and checkpoints. In addition, it checkpoints each transactional object on entry to a transaction. Typically, a prior checkpoint suffices, and the infrastructure recovers the state on entry to the transaction by replaying the request and reply messages in the log after the checkpoint.

There are other "online recovery" techniques such as those described in [17] that allow checkpoints to be taken while a replica is processing. Such techniques require knowledge of the application program and strict programming models. Although such techniques are feasible for a database system, they are less appropriate for the application programmers that write the middle-tier application business logic.

4.3 Recovery of Transactional Objects

The fault tolerance infrastructure uses the checkpoint of an existing replica to initialize the state of a new or restarted replica. It then replays the messages in the message log at the new or restarted replica. The key steps of the recovery process are shown in Figure 2, and are described below.

First, the new or restarted replica is added to the group membership, and the infrastructure at the replica starts to log messages. Shortly after, but not at the same point in time, it checkpoints an existing replica, and the multicast group communication system orders the checkpoint request

message within the message sequence and conveys the state to the new or restarted replica. The infrastructure initializes the state of the new or restarted replica and then replays subsequent logged messages to ensure that the state transfer occurs at the correct point in the message sequence.

To establish the synchronization point, and thus ensure that the state of a new or restarted replica is consistent with that of the existing replicas, our infrastructure uses a `Recovery_Start` protocol message. The `Recovery_Start` message is multicast and is delivered reliably, in a total order with respect to other messages, to all of the replicas in the group.

At an existing replica, the infrastructure transfers the message log, starting from the most recent checkpoint of the replica (which was obtained by invoking the `get_state()` method), followed by the incoming request and reply messages for the existing replica, through the message prior to the `Recovery_Start` message. Subsequent incoming messages at the existing replica are logged but are not transferred.

At a new or restarted replica, the infrastructure buffers messages following the `Recovery_Start` message, and discards messages preceding the `Recovery_Start` message. When it receives the log containing the checkpoint, the infrastructure installs the checkpoint by invoking the `set_state()` method. It then replays, to the new or restarted replica, first the messages that are contained in the log, and then those that it has buffered since the `Recovery_Start` message, until the new or restarted replica catches up with the other replicas.

The time at which the group membership changes and the time at which the checkpoint is taken are logically distinct. Consequently, this strategy is not a classical virtual synchrony strategy, but it does maintain the essential feature of virtual synchrony that the state transfer is synchronized with the message sequence. By not using view change state transfer, we prevent the blocking of processing of multicast messages, while a new or restarted replica is being added to the group.

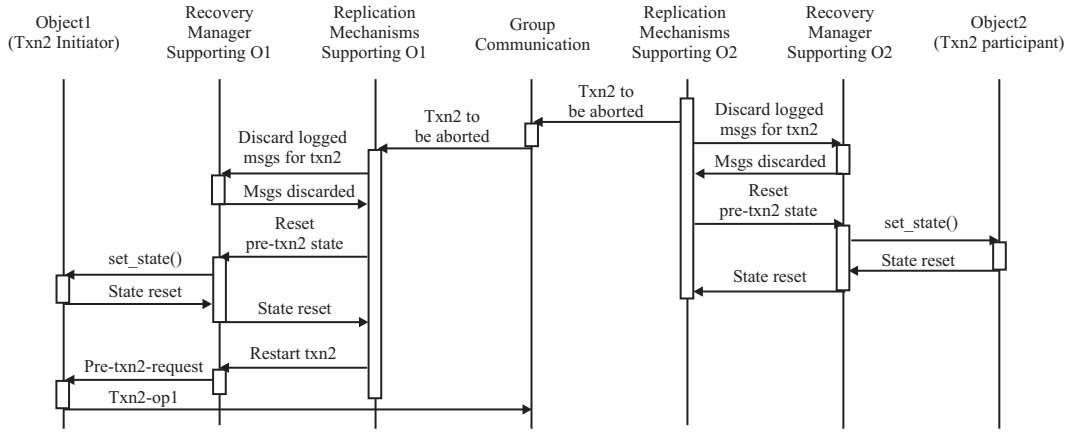

Figure 3: Sequence diagram for the recovery of an aborted transaction.

4.4 Recovery of Aborted Transactions

A transaction might be aborted for several reasons, including deadlock prevention, invalid authentication, and process or communication faults. The fault tolerance infrastructure and group communication system shield the application against process, processor and communication faults and, thus, against the rollback of a transaction. However, if any replica decides to abort a transaction, the transaction is aborted.

For flexibility, the fault tolerance infrastructure provides APIs that allow the application to indicate whether or not the infrastructure should retry an aborted transaction. The application can disable the retry of an aborted transaction if it has code for retrying the aborted transaction. Unless the application indicates otherwise, the infrastructure automatically retries and recovers aborted transactions. The key steps for recovery of an aborted transaction are shown in Figure 3.

When the fault tolerance infrastructure becomes aware that a transaction has been aborted, it sends a notification to all of the participants in the transaction (including the initiator). Then, the infrastructure resets the states of the application objects involved in the aborted transaction by applying the most recent checkpoints and replaying the logged request and reply messages up to, but not including, the message that took the objects into the transaction. It discards the

logged messages within the aborted transaction. Finally, the infrastructure replays the message that initiated the transaction, at the initiator.

Resetting the state of an application object is different from recovering a replica. To reset the state of an application object, the infrastructure applies, to the object, the most recent checkpoint of the object in the message log. It does *not* reset the middleware or service state. The retried transaction must be regarded as a new transaction that has a new transaction identifier; otherwise, the database system would regard the retried database operations as duplicates, and the transaction coordinator would abort the retried transaction because it has the same identifier as the aborted transaction.

If an application object is the initiator of a transaction and the transaction is aborted, the infrastructure initializes the object with the most recent checkpoint in the log and then replays the request and reply messages since the checkpoint up to, but not including, the message that initiated the transaction. The infrastructure discards logged request and reply messages within the aborted transaction, and then restarts the application object, which reinitiates the transaction.

If an application object is a participant in a transaction, but is not the initiator, and the transaction is aborted, the infrastructure initializes the object with the most recent checkpoint in the log and then replays the request and reply messages since the checkpoint up to, but not including, the request message that took the object into the transaction. The application object waits to receive the request message as part of the retried transaction. Retrying the transaction results in regeneration of request and reply messages that were previously logged within the aborted transaction, which is why these logged messages must be discarded at the beginning of the retry process.

4.5 Gateway Replication

Our fault tolerance infrastructure for three-tier applications employs two kinds of gateways at the boundaries of the fault tolerance domain, the inbound gateway between the external clients and

the replicated middle-tier servers, and the outbound gateway between the replicated middle-tier servers and the database system, as shown in Figure 1.

Both the inbound and the outbound gateways are replicated using a hybrid replication scheme that resembles semi-active replication [30], with a primary replica and one or more backup replicas. However, from within the fault tolerance domain, the inbound and outbound gateways appear to be actively replicated. Because the inbound gateway mechanisms have been specified in the FT CORBA standard [25], we do not discuss them further in this paper.

Only the primary outbound gateway establishes TCP/IP connections with the database servers. However, both the primary and the backup outbound gateways receive messages generated by the replicated middle-tier servers, and pass the messages from the database servers back to the replicated middle-tier servers.

The outbound gateway mechanisms reside in the same address space as the Totem group communication process. The connections from the application objects to the outbound gateway mechanisms are implemented using Unix sockets. Library interpositioning is used to capture the connection function calls and messages between a transactional process and the outbound gateway.

The primary outbound gateway does not write a request message for the database server to the TCP/IP connection immediately when it receives the message from the application. Instead, as shown in Figure 4, it waits for acknowledgments of the request message from all of the backup gateways, before sending its request message to the database server. This avoids the loss of messages if the primary outbound gateway fails, and provides certainty for database updates equivalent to the certainty provided by safe delivery, without incurring the overhead of such a multicast for other messages.

The reply messages, including both GIOP and SQL/XA reply messages, are multicast to all of the outbound gateway replicas serving a replica group. The primary outbound gateway delivers reply messages only when the multicast group communication protocol has totally ordered them,

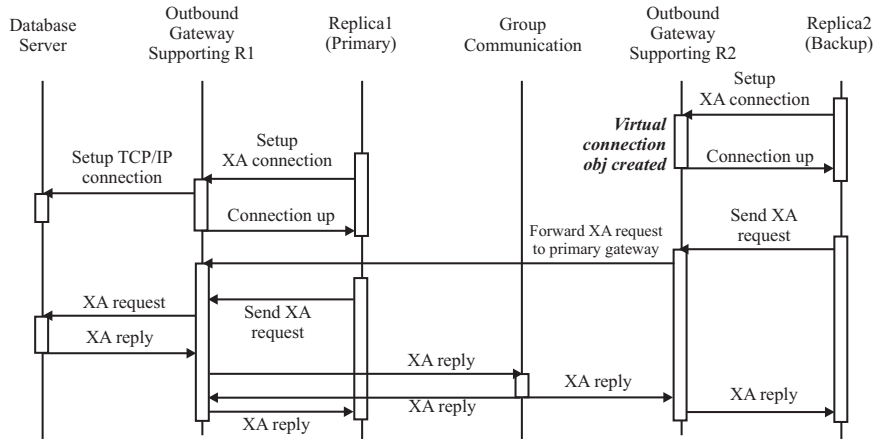

Figure 4: Sequence diagram for normal operations of an outbound gateway.

as shown in Figure 4. Because there is at most one outstanding request per connection, there is no ambiguity in matching a reply message from the database server with the corresponding request message. There is no need to know the wire protocol used by the database vendor.

When a middle-tier server processes multiple transactions concurrently, the database servers process multiple requests in an arbitrary unknown serialization order. This arbitrary order does not violate the strong replica consistency guarantees of our fault tolerance infrastructure, because each database server appears as a single entity to our infrastructure, and each database request is submitted once only to the database server and is never repeated. If the infrastructure retries an aborted transaction, *e.g.*, due to failure of the primary gateway, the retried transaction appears to the database server as a new transaction with a different XID. If we were to replicate the database servers, we would also need to be concerned with consistent ordering of transactions within the replicas of the database.

4.6 Gateway Recovery

Modern commercial databases, such as Oracle 8i, allow a client of the database server (in our case, the primary outbound gateway) to reestablish a connection to the same or a different database server endpoint and continue the transaction, when a fault occurs in the midst of a transaction.

The database server matches such a reconnection with an ongoing transaction using the XID. It also uses a timeout mechanism, so that the database server can abort a transaction unilaterally, if a client fails (otherwise, the database server might wait forever for the client to reconnect).

If the database system does not support reconnection, all ongoing transactions, except for transactions that have been prepared, must be rolled back. Likewise, if the primary outbound gateway fails before the transaction has been prepared and the new primary gateway is not sure of the status of the transaction, the transaction must be rolled back. Continuing an ongoing transaction during the failover of a gateway might result in the database server's processing an invocation twice. The automatic transaction-retry mechanisms of our infrastructure transparently retry a rolled-back transaction to achieve roll-forward recovery.

The group communication system multicasts messages, including transaction prepare and commit messages, and delivers them in total order to both the primary and the backup gateway replicas. Each gateway replica maintains a table of outstanding request messages. When it receives a reply corresponding to an outstanding request, it removes the request from its request table. If the primary gateway fails, a backup gateway uses its request table to achieve consistent recovery, as described below.

Note that the request table is necessary regardless of the type of delivery (agreed or safe) used for request messages, because the reply messages originate at the database servers and are sent via point-to-point TCP/IP communications. It is possible that the primary gateway receives a reply message (and the underlying TCP/IP stack acknowledges the reception of the message to the node running the sending database server) and fails before it multicasts the reply message to all of the gateway replicas. Note also that, if the database server were capable of multicasting the reply messages or if the gateway were collocated with the database server, it could multicast them directly to the replicated middle-tier servers. Safe delivery could then be used to eliminate the uncertainty and to render the request table unnecessary.

On failure of the primary gateway, the group communication system forms a new membership and broadcasts a membership change message, from which a backup gateway can determine that the primary gateway failed. The backup gateway checks whether there are any requests in its request table. It regards such requests as *uncertain outstanding requests*.

For each uncertain outstanding request, the backup gateway contacts the infrastructure for the status of the transaction to which the request belongs. In some cases, the infrastructure might have to query the transaction coordinator to determine whether the transaction is in the prepared state, or to query an XA resource to determine whether the XA resource has been prepared. If the transaction has not been prepared and there is an uncertain outstanding request, the backup gateway notifies the infrastructure to abort the transaction (we assume that the database server supports reconnection). In the second phase of the 2PC protocol, an uncertain outstanding request does not cause the rollback of the current transaction, because the commit notification to the database server is idempotent [26].

If the database server times out a connection while the infrastructure is failing over the outbound gateway, the database server aborts the transaction using the new connection if the transaction has not been prepared, and propagates the abort decision to the application object in the middle tier. The infrastructure notices the abort, enables the automatic retry mechanism, and restarts the transaction. If the database server times out a connection and makes a heuristic decision to abort a prepared transaction, it is up to the application to resolve the conflict.

Note that the abort of an on-going transaction when the primary gateway fails is due to our assumption that we have no access to the database tier except for TCP/IP connections. Otherwise, the risk of loss of transactions due to failure of the primary gateway could be reduced by collocating the primary gateway with the database server.

A remaining issue to be resolved is how to avoid critical runs during recovery. For example, if the transaction coordinator is recovering very fast in the presence of very slow participants, the

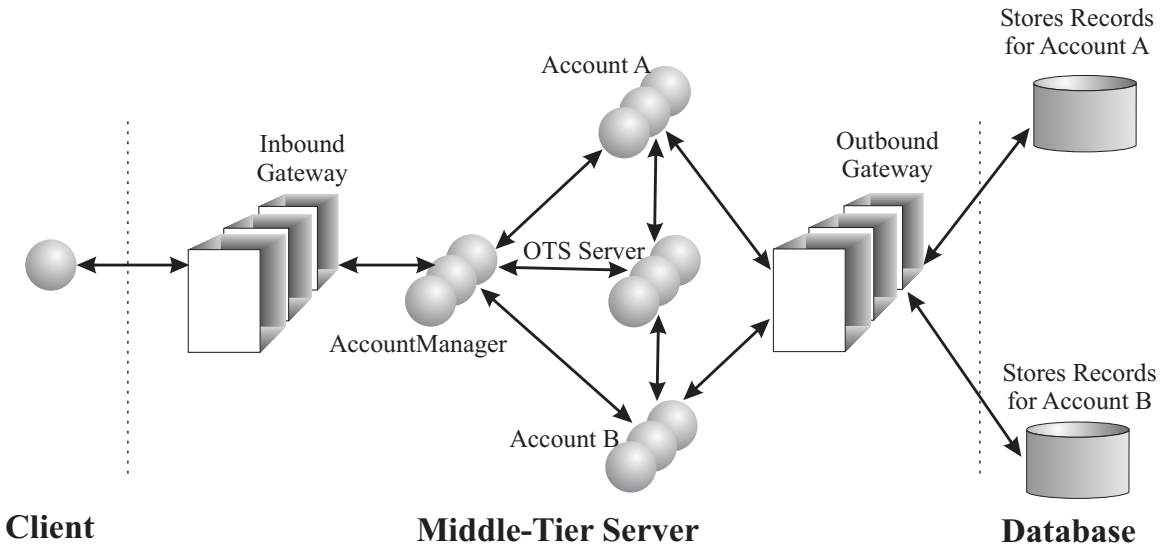

Figure 5: A three-tier banking application running on top of our fault tolerance infrastructure.

infrastructure might misread messages and, consequently, the system might not recover properly.

5 Implementation and Performance

We have implemented a prototype of the fault tolerance infrastructure that unifies transactions and replication for three-tier enterprise applications. The prototype works with the ORBacus CORBA ORB and its OTS implementation [27] from Object Oriented Concepts, Inc (now Iona). The Oracle 8i database management system is used as the XA resource manager. We ran our experiments 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. Although it would be more desirable and realistic to deploy the clients and database servers on different networks, this more ideal setup is not yet attainable due to the limitations of our current testbed.

The three-tier banking application that we used in our experiments is shown in Figure 5. A client invokes the replicated middle-tier server for a fund transfer operation between two different accounts that are managed by two different database servers, which update the tables in the

corresponding databases. In the middle tier, there are four distinct server processes (groups) running: an account manager, two account servers, and the OTS server. The account manager accepts remote method invocations from clients, initiates a distributed transaction for each fund transfer request, contacts the two account servers for the fund transfer, and commits the transaction. Each distributed transaction involves one (read-only) query and one update between each middle-tier account server and its associated database server. For each run, each client initiates a total of 1000 transactions; each transfer request follows completion of the prior request without any delay.

The middle-tier servers, including the OTS server, are three-way actively replicated on four of the six processors. Replicas of the same server (including the OTS server) are deployed on different processors (*i.e.,* no two replicas of the same server are collocated on the same processor). Thus, there are three replicas of the different servers on each processor. Inevitably, some server replicas execute on the same processor as an OTS server replica. Up to eight unreplicated clients are evenly distributed (whenever possible) on the two remaining processors. Two Oracle database management systems also execute on these two processors.

For comparison purposes, we measured the performance when the middle-tier servers are not replicated. In this case, the account manager server, the two account servers, and the OTS server each run on a separate processor while the clients and the Oracle database servers run on the remaining two processors.

As shown in Figure 6 (a), with replication, the overall system throughput, in transactions per second, is reduced by 10-20% over the unreplicated case. This minor reduction in throughput does not, however, reflect the true cost of our fault tolerance infrastructure in general. The overhead of our fault tolerance infrastructure is better reflected in Figure 6 (b), which considers the time for the middle-tier servers to start a new transaction and to carry out the fund transfer operation. The latency overhead of the fault tolerance infrastructure for the application business logic operations now increases from about 50% when the load is low to more than 100% when the load is high. The

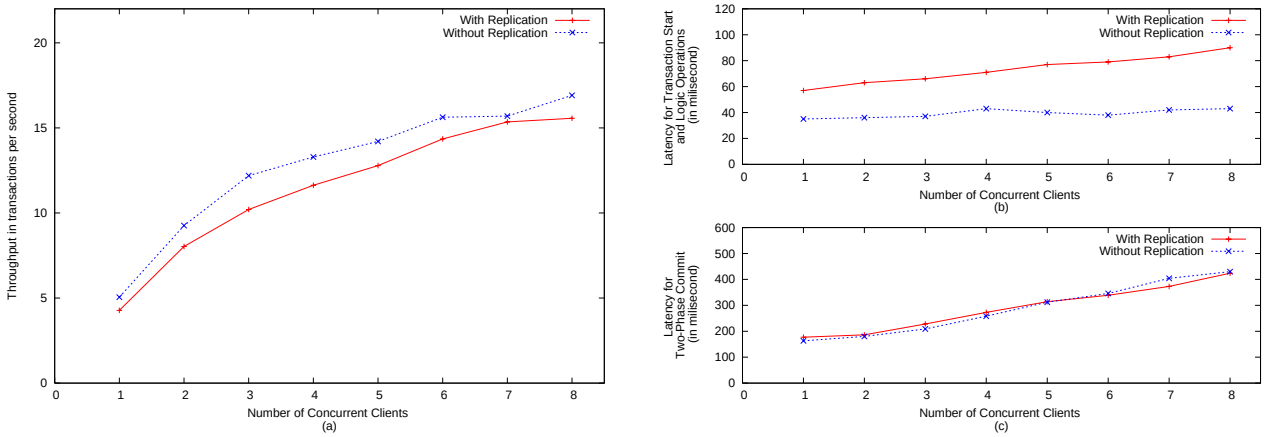

Figure 6: Performance measurement results as a function of the number of concurrent clients, with and without replication. (a) Throughput of the middle-tier server. (b) Mean latency for transaction startup and business logic. (c) Mean latency for the two-phase commit protocol.

low system throughput overhead is due to the dominant cost of the two-phase commit operations, as can be seen from Figure 6 (c).

To investigate the actual cost, we added a method to each middle-tier account manager to query (read-only) the new account balance from the database server and to return that balance to the account manager. For the sake of benchmarking, the account manager invokes this method multiple times on each account manager before committing a transaction. The results of this experiment are shown in Figure 7, where a single client invokes the account manager for a fund transfer with different numbers of account-balance-read operations inserted by the account manager. As can be seen in the figure, when the number of such read-only operations is larger, the relative replication overhead for the end-to-end latency is correspondingly higher.

In summary, the overhead of the fault tolerance infrastructure is primarily due to: (1) communication cost, where a message is multicast by the group communication system and some of the messages are redirected through the gateways, (2) processing cost, where some of the CPU cycles are dedicated to token handling in Totem, duplicate detection and suppression, and message parsing and patching, and (3) loss of concurrency, because of total ordering of messages and mes-

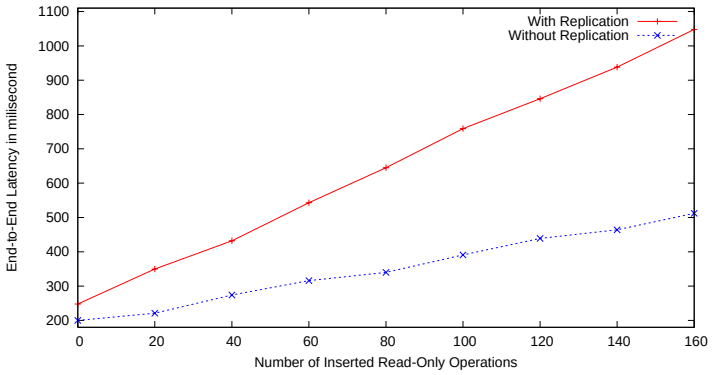

Figure 7: Mean end-to-end latency in milliseconds, with and without replication, as a function of the number of read-only operations of the middle-tier server.

sage scheduling to guarantee strong replica consistency. All three factors result in larger overheads when the load is higher.

In addition to the fault-free runtime performance, we measured the fault detection time and recovery time for our infrastructure. For a process crash fault, a Totem instance records a start time before it sends a kill signal to the process, and an end time when it receives the notification regarding the lost socket event. The process crash fault detection time is the difference between the end time and the start time. On average, the process crash fault detection time is about 3 ms. The processor fault detection time depends on the timeout value used in Totem which, in turn, depends on the characteristics of the network. In our testbed, a 1 second timeout suffices and, therefore, the processor fault detection time is approximately 1 second.

The recovery time for a process replica includes the time for the replica to join the group membership, the time to retrieve the state of an existing replica, the state transfer time, and the time to inject the state into the new or restarted replica. In the best case, when there is no queueing involved in retrieving and restoring the application state, and no other process is competing with the replica for retrieving and restoring the application state, it takes about 100 ms to recover a replica with a state size of 100 kBytes. If the communication and/or computation loads are higher, the recovery time is correspondingly longer. The fault detection time is also longer under higher

load.

6 Related Work

Several researchers [7, 23, 31] have investigated object replication and fault tolerance for CORBA prior to the adoption of the Fault Tolerant CORBA standard [25]. Since then, other researchers [21, 24] have developed partial or complete implementations of Fault Tolerant CORBA that middle-tier servers might use. To the best of our knowledge, none of those researchers has implemented an infrastructure that unifies transactions and replication in three-tier architectures.

Frolund and Guerraoui [9, 10, 11] have pointed out the deficiencies of both the CORBA OTS and the FT CORBA standards. They have proposed an exactly-once transactions (e-transactions) specification for three-tier architectures that integrates transactions and replication. They have also introduced a set of protocols as an implementation of their e-transactions specification. The e-transaction approach aims to combine replication with distributed transaction commitment and recovery to achieve higher availability and better performance.

In [8] Felber and Narasimham have presented a discussion of the issues involved in, and the benefits of, reconciling transactions and replication for CORBA applications. They have outlined a protocol for use, in transactional environments, that provides end-to-end reliability between the clients and the replicated servers. They have not provided any implementation details or performance measurements.

In [19] Little and Shrivastava have proposed a high availability solution for CORBA applications written in Java. Their system replicates the application objects to achieve forward progress and uses transactions to provide consistency. Their implementation is based on the CORBA OTS, but not on FT CORBA. In [20] they have further explored ways in which transactions and group communication can be used together. They conclude that process groups can be used with transaction processing for binding service replication, faster failover, and active replication.

JBoss is an open-source Java EJB/J2EE application server that has been extended with the JavaGroups group communication toolkit to provide session state replication and failover [2, 4]. JBoss uses an abstraction framework to isolate communication layers that resembles CORBA's pluggable protocols framework. Thus, like our infrastructure, it achieves transparency to the applications and other middleware. However, JBoss extended with JavaGroups does not address all of the difficult issues that our infrastructure addresses in unifying transactions and replication.

In [29] Patino-Martinez, Jimenez-Peris and Arevalo have investigated the integration of transactions and group communication, and have introduced the group transactions model, where a transactional server is a group of processes, and clients interact with the transactional server by multicasting requests to the group. Jimenez-Peris, Patino-Martinez, Alonso and Arevalo [15] have described a non-blocking atomic commitment protocol that exploits replication to achieve the non-blocking property and that reduces the latency by employing optimistic techniques. Unlike our fault tolerance infrastructure, which can be plugged into the middleware transparently, their approaches require modification of the applications to use a particular programming and communication model, or use proprietary protocols that are difficult to integrate into existing middleware.

In [14] Jimenez-Peris, Patino-Martinez and Arevalo have investigated deterministic scheduling for transactional multithreaded replicas, using a special-purpose library, called *Transactional Drago* [28]. Jimenez-Peris and Patino-Martinez [17] have also presented techniques for deterministic scheduling and online recovery for transactional multithreaded replicas. In [3] Basile, Whisnant and Iyer have presented a deterministic scheduling algorithm for multithreaded replicas; their algorithm is based on preemption.

7 Conclusions and Future Work

We have described a fault tolerance architecture that unifies transactions and replication to achieve data consistency and high availability for enterprise applications. We have developed mechanisms

that solve the transaction outcome non-determinism problem, render the two-phase commit protocol non-blocking, and automatically retry aborted transactions. We have also developed replication mechanisms that guarantee strong replica consistency, not only during fault-free conditions but also when adding a new or restarted replica to a group. Based on these mechanisms, we have implemented a prototype infrastructure that works with CORBA middle-tier servers and Oracle database servers running over the Linux operating system.

To realize the widespread use of transaction processing and fault tolerance technology to protect not only the data but also the transactions and business logic processing, a number of other issues still remain to be addressed. In particular, it would be desirable to integrate additional capabilities, such as scalability, load balancing and security, into a three-tier architecture that provides high availability, data consistency and fault tolerance. Moreover, developing an infrastructure that can support other distributed computing protocols and platforms, such as Web Services, would be desirable. Many of the techniques that we have developed in the context of CORBA are general and can be adapted to other protocols and platforms.

Acknowledgment

We wish to thank the associate editor, Professor Andre Schiper, and the reviewers for their valuable comments, which have greatly improved this paper. This paper expands on material that was previously published in the Proceedings of the IEEE 12th International Symposium on Software Reliability Engineering, Hong Kong, China, November 2001 [35] and the Proceedings of the IEEE 23rd International Conference on Distributed Computing Systems, Vienna, Austria, July 2002 [36]. This research was supported by DARPA/ONR Contract N66001-00-1-8931 and MURI/AFOSR Contract F49620-00-1-0330.

References

[1] Y. Amir and C. Tutu. From total order to database replication. In *Proceedings of the IEEE 22nd International Conference on Distributed Computing Systems*, pages 494–503, Vienna, Austria, July 2002.

[2] B. Ban. Design and implementation of a reliable group communication toolkit for Java, September 1998. *http://www.cs.cornell.edu/home/bba/Coots.ps.gz*.

[3] C. Basile, K. Whisnant, and R. Iyer. A preemptive deterministic scheduling algorithm for multithreaded replicas. In *Proceedings of the IEEE International Conference on Dependable Systems and Networks*, pages 149–158, San Francisco, CA, June 2003.

[4] B. Burke and S. Labourey. Clustering with JBoss 3.0, July 2002. *http://www.onjava.com/pub/a/onjava/2002/07/10/jboss.html*.

[5] T. D. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2):225–267, 1996.

[6] W. R. Dieter and J. E. Lumpp, Jr. A user-level checkpointing library for POSIX threads programs. *Proceedings of the IEEE 29th International Symposium on Fault-Tolerant Computing Systems*, pages 224–227, Madison, WI, June 1999.

[7] 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.

[8] P. Felber and P. Narasimhan. Reconciling replication and transactions for the end-to-end reliability of CORBA applications. In *Proceedings of the Confederated International Conferences CoopIS, DOA, and ODBASE 2002*, Lecture Notes in Computer Science 2519, pages 737–754, Irvine, CA, January 2002.

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

[10] S. Frolund and R. Guerraoui. Implementing e-transactions with asynchronous replication. *IEEE Transactions on Parallel and Distributed Systems*, 12(2):133–146, February 2001.

[11] S. Frolund and R. Guerraoui. e-Transactions: End-to-end reliability for three-tier architectures. *IEEE Transactions on Software Engineering*, 28(4):378–395, April 2002.

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

[13] R. Guerraoui, M. Larrea, and A. Schiper. Reducing the cost for non-blocking in atomic commitment. In *Proceedings of the IEEE 16th International Conference on Distributed Computing Systems*, pages

692–697, Hong Kong, May 1996.

[14] R. Jimenez-Peris, M. Patino-Martinez, and S. Arevalo. Deterministic scheduling for transactional multithreaded replicas. In *Proceedings of the IEEE 19th Symposium on Reliable Distributed Systems*, pages 164–173, Nurnberg, Germany, October 2000.

[15] R. Jimenez-Peris, M. Patino-Martinez, G. Alonso, and S. Arevalo. A low-latency non-blocking atomic commit service. In *Proceedings of the IEEE 15th International Conference on Distributed Computing*, Lecture Notes in Computer Science 2180, pages 93–107, Lisbon, Portugal, October 2001.

[16] R. Jimenez-Peris, M. Patino-Martinez, B. Kemme, and G. Alonso. Improving the scalability of fault-tolerant database clusters. In *Proceedings of the IEEE 22nd International Conference on Distributed Computing Systems*, pages 477–484, Vienna, Austria, July 2002.

[17] R. Jimenez-Peris and M. Patino-Martinez. Deterministic scheduling and online recovery for replicated multithreaded transactional servers. In *Proceedings of the Workshop on Dependable Middleware-based Systems, International Conference on Dependable Systems and Networks*, Washington, D.C., June 2002.

[18] F. Kuhns, C. O'Ryan, D. C. Schmidt, O. Othman, and J. Parsons. The design and performance of a pluggable protocols framework for object request broker middleware. In *Proceedings of the IFIP Sixth International Workshop on Protocols For High-Speed Networks*, pages 81–98, Salem, MA, August 1999.

[19] M. C. Little and S. K. Shrivastava. Implementing high availability CORBA applications with Java. In *Proceedings of the IEEE Workshop on Internet Applications*, pages 112–119, San Jose, CA, July 1999.

[20] M. C. Little and S. K. Shrivastava. Integrating group communication with transactions for implementing persistent replicated objects. In *Advances in Distributed Systems*, Lecture Notes in Computer Science 1752, pages 238–253, 1999.

[21] 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.

[22] 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.

[23] 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.

[24] P. Narasimhan, L. E. Moser, and P. M. Melliar-Smith. Strongly consistent replication and recovery of fault-tolerant CORBA applications. *Computer System Science and Engineering Journal*, 17(2):103–

114, March 2002.

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

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

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

[28] M. Patino-Martinez, R. Jimenez-Peris, and S. Arevalo. Synchronizing group transactions with rendezvous in a distributed Ada environment. In *Proceedings of the ACM Symposium on Applied Computing*, pages 2–9, Atlanta, GA, February 1998.

[29] M. Patino-Martinez, R. Jimenez-Peris, and S. Arevalo. Group transactions: An integrated approach to transactions and group communication. In *Concurrency in Dependable Systems*, P. Ezhilchelvan and A. Romanovsky (eds.), pages 1-19, Kluwer Academic Publishers, 2002.

[30] D. Powell. *Delta-4: A Generic Architecture for Dependable Distributed Computing*. Springer-Verlag, Berlin, Germany, 1991.

[31] Y. Ren, D. E. Bakken, T. Courtney, M. Cukier, D. A. Karr, P. Rubel, C. Sabnis, W. H. Sanders, R. E. Schantz, and M. Seri. AQuA: An adaptive architecture that provides dependable distributed objects. *IEEE Transactions on Computers*, 52(1):31–50, January 2003.

[32] A. Schiper and A. Sandoz. Uniform reliable multicast in a virtually synchronous environment. In *Proceedings of the IEEE 13th International Conference on Distributed Computing Systems*, pages 561–568, Pittsburgh, PA, May 1993.

[33] D. Skeen. Nonblocking commit protocols. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 133–142, April–May 1981.

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

[35] W. Zhao, L. E. Moser, and P. M. Melliar-Smith. Increasing the reliability of three-tier applications. In *Proceedings of the IEEE 12th International Symposium on Software Reliability Engineering*, pages 138–147, Hong Kong, China, November 2001.

[36] W. Zhao, L. E. Moser, and P. M. Melliar-Smith. Unification of replication and transaction processing in three-tier architectures. In *Proceedings of the IEEE 23rd International Conference on Distributed Computing Systems*, pages 290–297, Vienna, Austria, July 2002.

[37] W. Zhao, L. E. Moser, and P. M. Melliar-Smith. Deterministic scheduling for multithreaded replicas. In *Proceedings of the IEEE Workshop on Real-Time Object-Oriented Dependable Systems*, Sedona, AZ, February 2005.