Reconciling Replication and Transactions for the End-to-End Reliability of CORBA Applications

Pascal Felber[1] and Priya Narasimhan[2]

[1] Institut EURECOM, 2229 route des Crêtes, BP 193
06904 Sophia Antipolis, France
pascal.felber@eurecom.fr

[2] ISRI, School of Computer Science, Carnegie Mellon University
Pittsburgh, PA 15213-3890
priya@cs.cmu.edu

Abstract. The CORBA standard now incorporates support for reliability through two distinct mechanisms — replication (using the Fault Tolerant CORBA standard) and transactions (using the CORBA Object Transaction Service). Transactions represent a roll-back reliability mechanism, and handle a fault by reverting to the last committed state, and by discarding operations that were in progress at the time of the fault. Replication represents a roll-forward reliability mechanism, and handles a fault by re-playing any operations that were in progress at another operational replica of the crashed server. Most of today's enterprise applications have a three-tier structure, with simple clients in the first tier, servers in the middle-tier to perform the processing, and databases in the third tier to store information. For such applications, replication is required to protect the middle-tier processing, while transactions are required to protect the third-tier data. This requires the reconciliation of roll-forward and roll-back reliability mechanisms in order to protect both data and processing, and to provide consistent end-to-end reliable operation. This paper looks at the issues of integrating replication with transactions for three-tier enterprise CORBA applications, with particular emphasis on reconciling the Fault Tolerant CORBA standard and the CORBA Object Transaction Service.

1 Introduction

The emergence of Internet-based enterprise computing and Web-based electronic commerce has led to the development of system architectures with advanced features to address technical and quality-of-service (QoS) issues such as security, data integrity, high-availability, reliability, scalability, atomicity, and session management. Reliability, in particular, forms the cornerstone of every enterprise, with the market becoming increasingly intolerant of downtime, and with enterprise server crashes leading to bad publicity, prohibitive financial losses, reduction in stock prices, and loss of customers. With its adverse impact on the economy [4] and on our quality of life, the lack of fault tolerance is increasingly more unacceptable, particularly for enterprise applications. Therefore, continuous perceived uptime and reliability are key requirements of electronic commerce servers.

There are several approaches to providing fault tolerance. Hardware redundancy solutions are insufficient because they focus primarily on detecting, and tolerating, hardware defects. Enterprise systems contain a lot of software, for the most part, and use both hardware and software components that can fail, thereby disrupting service. To provide the necessary degree of availability, these systems must use hot swappable components (to replace a failed component on-the-fly with a working component) and failover (to switch all clients to working with the replaced component, instead of the old failed one). There are two distinct aspects of fault tolerance for enterprise systems — protection of the data and protection of the processing (operations that are in progress) when a fault occurs. To protect against the loss or the corruption of data, databases and transaction processing systems are often employed. To protect against the loss of operations or processing, the servers that perform the processing are often replicated, so that there exist redundant servers to perform the computing and redundant network resources for running the distributed application.

Most of today's enterprise applications have a three-tier structure, with simple front-end clients (usually browsers) in the first tier, servers (business logic) in the middle-tier to perform the processing, and databases and legacy applications in the third tier to store information. Different technologies form the basis for each of the three tiers — for instance, application servers are usually the environment of choice for the middle-tier business logic components. The architecture of most enterprise applications is based on client-server middleware, where a client requests services across the network, and a server performs the services and returns results to the client.

Middleware platforms such as the Common Object Request Broker Architecture (CORBA) [8] are increasingly being adopted because they simplify client-server application programming by rendering transparent the low-level details of networking, distribution, physical location, hardware, operating systems, and byte order. However, despite their many attractive features, middleware platforms have still not found favor for deployment in applications that have high reliability requirements. Recognizing this deficiency, the Object Management Group (OMG), the CORBA standards body, has attempted to incorporate specifications for reliability into the CORBA middleware standard. The reliability support within CORBA takes two different forms: (i) the CORBA Object Transaction Service (OTS) [10], and (ii) the recently adopted Fault Tolerant CORBA (FT-CORBA) standard [9].

Unfortunately, both of these specifications were developed independently, and it is, in fact, difficult to reconcile replications and transactions in general. The reason for this is that replication represents a *roll-forward* mechanism, where a fault (crash of a server) is tolerated by switching over to a backup replica of the server, and re-doing, or moving forward with, the operations in progress at the failed server. On the other hand, transactions represent a *roll-back* mechanism, where the crash of a server is tolerated by discarding all of the operations in progress at the failed server, and by reverting to the last well-known, or committed, state persisted in a database. Clearly, the focus of each reliability mechanism differs — transactions focus on protecting data, while replication focuses on protecting processing. Stated another way, a roll-forward mechanism promotes *liveness*, while a roll-back mechanism promotes *safety*.

For true reliability, enterprise applications clearly require elements of both roll-forward and roll-back reliability strategies, in order to protect *both* data and processing. In fact, each of the two complementary strategies stands to benefit greatly from the other. However, in today's state-of-the-art and state-of-the-practice, this is challenging to achieve.

There exist two orthogonal, but equally essential, properties in building mission-critical distributed systems: *availability* and *consistency*. Availability provides clients with the abstraction of a continuous service, despite the failure of some server components, and is generally achieved using the replication of critical resources, so that the failure of the copy of a critical component can be masked by another copy. Consistency guarantees informally that the system will always remain in a coherent state, despite the occurrence of faults. The partial execution of an operation might lead to the violation of the consistency property. For instance, when money is transferred from one bank account to another, the system is in an incoherent state if the money is withdrawn from the source account, but not deposited in the destination account. Consistency is generally maintained through the use of transactional facilities. By integrating replication and transactions, we can achieve two additional objectives: we provide stronger consistency to replicated systems by supporting non-deterministic operation, and we provide higher availability and failure transparency to transactional systems.

In this paper, we explore the underlying problems in composing the roll-forward capability provided by replication with the roll-back capability provided by transactional systems, in order to achieve end-to-end reliability for distributed enterprise CORBA systems. We have chosen CORBA as our vehicle for exploring the research problems of integrating replication and transactions because of our prior experience[1] with building replication-based CORBA systems, and also because CORBA is unique in being the only middleware that currently incorporates specifications for both replication (FT-CORBA) and transactions (OTS). It is our hope that the ideas in this paper will lay the foundations for reconciling these two separate specifications and, thereby, for deriving more powerful capability by combining FT-CORBA with OTS. It is not our intention, in this paper, to invent novel protocols, but rather to present a pragmatic approach for leveraging the best of the transactional and replication worlds, and for combining their power to achieve both reliability and availability for critical CORBA applications.

[1] The co-authors of this paper have independently developed fault-tolerant CORBA systems [2,7], well before the FT-CORBA standard was approved. In addition, both co-authors have contributed to, and participated in, the FT-CORBA standardization process.

The rest of this paper is organized as follows: Section 2 introduces the necessary background concepts, and presents the fundamental system models that we consider in the rest of this paper. Section 3 describes and compares roll-back and roll-forward reliability in greater detail. Section 4 outlines our novel scheme for integrating the best of the FT-CORBA and the OTS mechanisms to achieve end-to-end availability and consistency. Section 5 discusses other research efforts that are relevant to our work in this area. Finally, Section 6 presents some concluding remarks.

2 Background

The architectures that form the focus of this paper are three-tier distributed enterprise middleware applications. Thin first-tier clients communicate with application servers that implement the application's logic, typically using a middleware component model such as Enterprise JavaBeans (EJB) [14] or the CORBA Component Model (CCM) [11]. These middle-tier servers are transactional and have access to back-end systems, which are typically highly available parallel database servers. Figure 1 illustrates such a three-tier configuration in which multiple application servers are used to process requests from possibly thousands of thin clients.

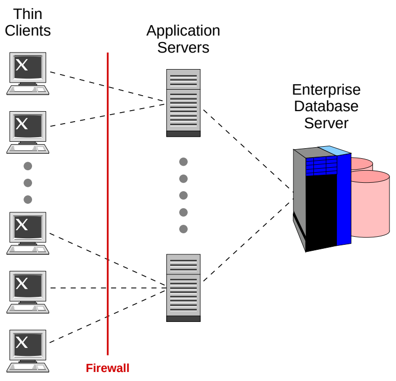

Fig. 1. A Typical Three-tier Architecture

Although communication between clients and servers can happen over various protocols, such as HTTP, HTTP/S, RMI-IIOP or SSL, in this paper, the focus is on the use of CORBA for distributed first-to-second tier client-server interactions. Similarly, we assume that communication between the middle-tier application servers and the third-tier database system are performed using CORBA. Thus, for the rest of this text, we assume that the communication between the various tiers of this distributed client-server architecture occurs over CORBA's

Internet Inter-ORB Protocol (IIOP). However, the mechanisms discussed in this paper are generic, and can be readily extended to other middleware and protocols, as well as to additional levels of nesting (n-tier architectures).

In the rest of this section, we describe the current support for transactions and fault tolerance within the CORBA standard.

2.1 The CORBA Object Transaction Service

CORBA incorporates support for roll-back reliability through the Object Transaction Service (OTS) [10]. OTS forms a part of the rich suite of services (such as Naming, Events, Notification, etc.) that CORBA incorporates, and that vendors provide, in order to free CORBA programmers from having to write such commonly-used functionality themselves.

OTS essentially specifies interfaces for synchronizing a transaction across the elements of a distributed client-server application, as shown in Figure 2. A transaction satisfies the four so-called ACID properties: Atomicity, i.e., transactions executes completely or not at all; Consistency, i.e., transactions are a correct transformation of state; Isolation, i.e., even though transactions execute concurrently, it appears for each transaction, T, that other transactions execute either before T, or after T, but not both; and Durability, i.e., modifications performed by completed transactions survive failures.

In OTS, a transaction is typically initiated by a client, and can involve multiple objects performing multiple requests. The scope of the transaction is defined by a transaction context, which is shared by the participating objects. The transaction context is logically bound to the thread of the client that initiated the transaction, and is implicitly associated with subsequent requests that the client issues, until the client decides to terminate the transaction. If no fault occurs for the duration of the transaction, the changes produced as a consequence of the client's requests are committed, or preserved, in accordance with the durability property above. In case a fault occurs during the transaction, any changes to data that have occurred within the duration and scope of the current transaction are rolled back and discarded.

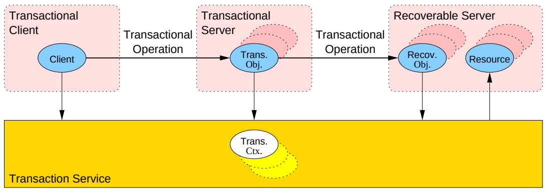

Fig. 2. Architectural Overview of OTS

Apart from the transactional client and the transaction context, a distributed transaction typically involves three other kinds of objects — transactional, recoverable and resource objects. *Transactional objects* are CORBA objects whose behavior and/or state is affected by being invoked within the scope of a transaction (e.g., objects that refer to persistent data that can be modified by transactional requests). *Recoverable objects* are transactional objects that explicitly participate in the transaction service protocols. They do so by registering *resource objects* with the OTS which, in turn, drives the commit protocol by issuing requests to the resources registered for the transaction. Transactional (recoverable) objects are hosted in transactional (recoverable) servers, which participate to the two-phase commit protocol executed upon completion of a distributed transaction.

OTS implements roll-back reliability in the sense that the effect of the requests issued in the context of a failed transaction are undone on all recoverable servers. Roll-back might be implicitly triggered on the occurrence of a fault, or explicitly requested by a transactional object. Upon roll-back, the client might re-try the transaction or take some other appropriate action.

2.2 The Fault-Tolerant CORBA Standard

Support for roll-forward reliability in CORBA is provided by the recently adopted fault-tolerant CORBA (FT-CORBA) [9] specification. FT-CORBA implements reliability by replicating critical objects: if a server replica fails while processing a client's request, then another replica can take over the processing of the request, generally without the client noticing the failure.

The FT-CORBA specification includes minimal fault-tolerant mechanisms to be included in any CORBA implementation, as well as interfaces for more advanced management facilities intended to be provided by a fault-tolerant CORBA implementation. FT-CORBA implementors are free to use proprietary mechanisms (such as reliable multicast protocols) for their actual implementation, as long as the resulting system complies with the interfaces defined in the specification, and the behavior expected from those interfaces.

The client-side mechanisms to be included in all CORBA implementations — regardless of whether they implement FT-CORBA or not — have been intentionally kept minimal. They essentially specify object references that can contain multiple profiles, each of which designates a replica (multi-profile IORs[2]), and simple rules for iterating through the profiles in case of failure. These mechanisms ensure that unreplicated clients can interact with replicated FT-CORBA servers in a fault-tolerant manner.

The server-side components of FT-CORBA are shown in Figure 3. The Replication Manager handles the creation, deletion and replication of both the application objects and the infrastructure objects. The Replication Manager replicates objects and distributes the replicas across the system. Although each replica of

[2] An Interoperable Object Reference (IOR) is a standardized form of a reference to a CORBA object, and can contains one or more profiles. Each profile contains sufficient information to contact the object using some protocol, usually TCP/IP; this information often includes the host name, port number, and object key associated with the CORBA object.

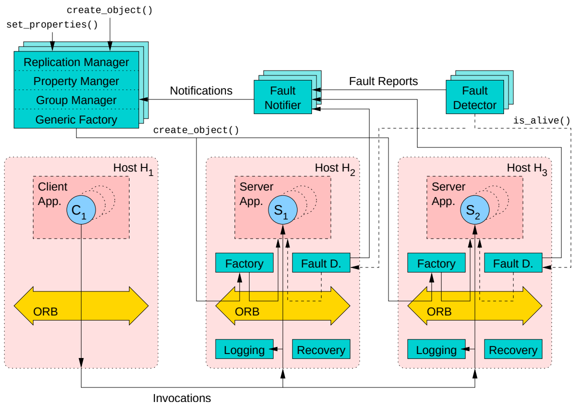

Fig. 3. Architectural Overview of FT-CORBA

an object has an individual object reference, the Replication Manager fabricates an object group reference that clients use to contact the replicated object. The Replication Manager's functionality is achieved through the Property Manager, the Generic Factory, and the Object Group Manager.

Host, process, and object faults are detected by the Fault Detector. The occurrence of faults are reported to the Fault Notifier, which filters them and distributes fault event notifications to the Replication Manager. Based on these notifications, the Replication Manager initiates appropriate actions to enable the system to recover from faults.

3 Reliability Strategies

In this section, we compare the replication-based and transaction-based reliability strategies introduced in the previous section, with particular focus on their respective benefits and drawbacks.

3.1 Replication-Based Reliability

Replication is intended at protecting computational resources through the use of redundancy: if a processor fails, then another processor can take over the processing of the failed processor.

In distributed systems, the two best-known replication styles are *active* [13] and *primary-backup* [1] (or *passive*) replication. A replicated object is often represented by an object group, with the replicas of the object forming the members of the group. The object group membership may be static or dynamic. Static membership implies that the number, and the identity, of the replicas do not change over the lifetime of the replicated object; on the other hand, dynamic replication allows replicas to be added or removed at run-time.

With active replication, all of the replicas of the object play the same role: every active replica receives each request, processes it, updates its state, and sends a response back to the client. Because the client's invocations are always sent to, and processed by, every server replica, the failure of any of the server replicas can be made transparent to the client. With primary-backup replication, one of the server replicas is designated as the primary, while all of the other replicas serve as backups. A client typically sends its request only to the primary, which executes the request, updates it own state, updates the states of the backups, and sends the response to the client. The periodic state updates from the primary to the backups serve to synchronize the states of all of the server replicas at specific points in their execution.

Replication implements *roll-forward* recovery mechanisms that promote liveness by continuing processing where it had been left at the time of the failure. In active replication, in the event of a fault (one of the active replicas crashes), the other replicas continue processing the current request, regardless, thereby implicitly implementing a roll-forward mechanism. In primary-backup replication, in the event of a fault (the primary replica crashes), one of the backup replicas takes over as the new primary and re-processes any requests that the previous primary was performing before it failed. If a backup replica crashes, then, there is no loss in processing. Thus, the roll-forward mechanism is explicitly implemented in the re-election of a new primary replica, and the re-processing of requests by the new primary. Consistency is maintained for both active and primary-backup replication by guaranteeing that partial request execution will not harm since the request will be eventually completed (by "rolling forward").

A major limitation of replication-based system is that consistency may not be preserved in the presence of non-determinism. Indeed, a frequent assumption in building replicated CORBA systems is that each CORBA object is deterministic in behavior. Determinism implies that if distinct distributed replicas of the object, starting from the same initial state, receive and process the same set of operations in the same order, they will all reach the same final state. It is this reproducible behavior of the application that lends itself so well to reliability. Unfortunately, pure deterministic behavior is rather difficult to achieve, except for very simple applications. Common sources of non-determinism include the use of local timers, multi-threading, operating system-specific calls, processor-specific functions, shared memory primitives, etc.

Non-deterministic behavior is an inevitable and challenging problem in the development of fault-tolerant systems. For active replication, determinism is crucial to maintaining the consistency of the states of the replicas of the object. Passive replication is often perceived to be the solution for non-deterministic applications. There is some truth in this perception because, with passive replication, invocations are processed only by the primary, and the primary's state is captured and then used to update the states of the backup replicas. If the primary fails while processing an invocation, any partial execution is discarded, and the invocation is processed afresh by the new primary. Because the state updates happen only at one of the replicas, namely, at the primary replica, the results of any non-deterministic behavior of the replicated object are completely contained, and do not wreak havoc with the consistency of the object.

There exist situations, however, where passive replication is not sufficient to deal with non-determinism. This is particularly true of scenarios where the non-deterministic behavior of a passively replicated object is not contained because the behavior has "leaked" to other replicated objects in the system. Consider the case where the primary replica invokes another server object based on some non-deterministic decision (e.g., for load balancing, the primary replica randomly chooses one of n servers to process a credit-card transaction). If the primary replica fails after issuing the invocation, there is no guarantee that the new primary will select the same server as the old primary; thus, the system will now be in an inconsistent state because the old and the new primary replicas have communicated with different servers, both of whose states might be updated.

For passive replication to resolve non-deterministic behavior, there should be no persistent effect (i.e., no lingering "leakage" of non-determinism) resulting from the partial execution of an invocation by a failed replica. This is possible if the passively replicated object does not access external components based on non-deterministic inputs, or if all accesses are performed in the context of a transaction aborted upon failure. Sources of non-determinism (such as thread scheduling) can also be controlled by careful programming. In general, however, passive replication is no cure for non-determinism.

3.2 Transaction-Based Reliability

Unlike replication, transaction processing systems essentially aim at protecting data. When a failure occurs in the context of a transaction, the objects involved in the transaction are reverted to their state just prior to the beginning of the transaction. All of the state updates and all of the processing that occurred during the transaction are discarded, often with no trace left in the system. Some systems support nested transactions, where a new (child) transaction can be initiated within the scope of an existing (parent) transaction. If the nested (child) transaction fails, the enclosing (parent) transaction needs not automatically roll back; the application can attempt to correct the problem, and subsequently retry the nested transaction. However, if the enclosing transaction encounters a fault, then all the nested transactions roll back, along with the enclosing transaction.

Transactions use roll-back recovery mechanisms that guarantee consistency by undoing partial request processing. Data is protected from the undesirable side-effects of failures, but computational resources may become unavailable for arbitrary durations. Transactions are thus an effective mechanism for preserving consistency, but not for achieving high availability, as they sometimes trade liveness for safety.

3.3 Roll-Back vs. Roll-Forward

Consider a simple three-tier bank application. A client C issues a transfer request to a passively replicated bank object B, which in turn withdraw money from a bank account X and deposit it on another account Y (see Figure 4). The bank is essentially a stateless object that coordinates the money transfer between stateful bank account objects, typically hosted in a database server. As the data

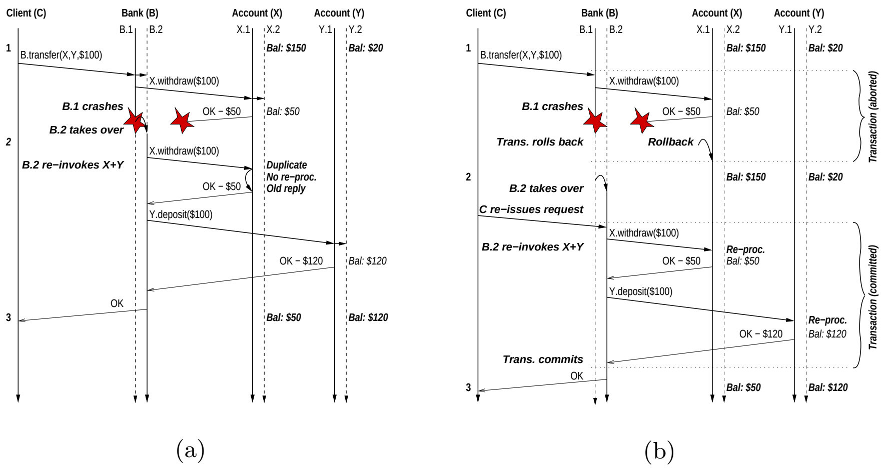

Fig. 4. Reliability in Multi-tier Applications. (a) Roll-forward. (b) Roll-back.

pertaining to the bank accounts is possibly managed by distinct entities, e.g., different branches, the account objects execute on different servers and are not co-located with the bank object. Any of the bank or account objects can therefore fail independently.

If the primary bank object B_1 fails while performing a transfer, neither the invoker C, nor the backup B_2 generally know at what point B_1 failed — whether it was before/while invoking X, between invocations to X and Y, or after/while invoking Y. Roll-forward and roll-back reliability strategies adopt two approaches to address this problem, as illustrated in Figure 4, where the flow of requests/replies is represented by arrows, and the flow of time occurs downward, toward the bottom of the figure.

With roll-forward reliability strategies, invocations are traditionally sent using reliable multicast (also known as reliable group communication), so that all of the replicas of an object receive every request. This is evident in an active replication configuration, where a client does not need to re-issue the request if one of the active server replicas fails (in fact, the client is typically not even aware of this failure). In a primary-backup setting, when the primary has finished processing a request, it multicasts both the response and a state update to the backups before returning the response to the client. The state update allows the backups to synchronize their state with that of the primary. The response is also cached by the backups for retrieval, should the primary fail. If the primary fails, then a backup assumes the role of the new primary transparently. If the primary fails before returning a response to the client, the client will re-issue the request to one of the backups (now the new primary); if the new primary has a cached response and the last state update of the old primary, it can readily return a response; if it doesn't have the cached response, it will re-process the request.

In Figure 4 (a), B_1 fails after having successfully invoked X, but before invoking Y and replying to C. When taking over as the new primary, B_2 does not know when B_1 failed, and it re-processes the request. If we assume that objects have a deterministic behavior, B_2 will re-invoke X and Y. Unless these invocations are idempotent (i.e., repeated executions of the invocation leave the state of the invoked object unaltered), re-processing these requests will corrupt the state of X. Thus, such duplicate processing of requests should not be allowed to occur. To guarantee this behavior, using FT-CORBA, the invoker embeds a request identifier within each request. This identifier is used at the invoked server X to detect a duplicate invocation, and to return previously cached replies instead of re-processing the entire request. The invocation to Y is normally processed and, finally, B_2 sends a reply to C. Note that, for C, the server moves from one consistent state (1) to another consistent state (3); the intermediary inconsistent state (2) is note exposed to C.

With roll-back reliability strategies, an invocation is typically sent using point-to-point communication and, upon failure of the invoked server, all effects of the invocation are first wiped out from the system, and the invoker then re-issues the request to a backup server. This retransmission is generally performed transparently by the middleware infrastructure, without the knowledge of the client application. However, the roll-back phase of the recovery requires mechanisms to enable a component to undo some changes to its state, and any effects of processing an invocation, in order to avoid inconsistencies. This can be achieved using transactional facilities to reset the component to a previous well-known committed state if the transaction aborts.

Roll-back becomes more interesting when the participants in the transaction are distributed entities. In such distributed transactions, it is possible to roll back the actions even on remote components that have been invoked in the context of the transaction. In Figure 4 (b), B_1 starts a new transaction before processing the request from C, and fails just after having successfully invoked X. Since B_1 did not commit the transaction, the actions performed on X as part of this transaction are undone, and the state of the system (i.e., the balance of the accounts) is reset to what it was prior to the the invocation of B_1. With this roll-back strategy, the entire invocation sub-chain whose root is the crashed object is reverted to its previous consistent state. Since C does not receive a reply, it eventually re-issues the request to B_2. If no fault occurs, invocations to X and Y are then processed normally, and a reply is returned to C. Note that, if C does not re-issue its request, the server will be left in a consistent state (2), identical to the initial state (1); after re-invocation, the server reaches a consistent final state (3).

Because of their potential for faster recovery (there is no roll-back phase prior to the retry-recovery phase), roll-forward approaches are well suited to systems that have real-time requirements or that need predictable response times, such as embedded systems or supervision and control applications. In particular, when roll-forward recovery is used with active replication, recovery time can be significantly faster than with roll-back recovery. On the other hand, roll-back approaches are well adapted to transactional systems where the integrity of data far outranks the recovery time, such as electronic commerce and banking applications.

Because each reliability strategies have distinct properties, critical distributed application can significantly benefit from combining their particular strengths (strong consistency with transactions and high availability with replication) while simultaneously alleviating their respective limitations (deterministic behavior with replications and unpredictable response time with transactions).

4 Combining Replication and Transactions

In this section, we outline a protocol to provide end-to-end reliability between clients and replicated servers. It supports non-deterministic servers and nested invocations, and can be used in transactional environments, such as enterprise application servers. Although we illustrate this protocol in the context of a three-tier architecture, it extends naturally to n-tier systems.

With this protocol, the client can issue a request (remote invocation) to a "highly available" server. The outcome of the request is preserved, despite the failure of the client, network, or server. In the event of a failure, the client can re-issue the same request to obtain a reply, without worrying about duplicate processing and its potential for the corruption of the server state.

The protocol makes use of FT-CORBA and OTS to replicate computational resources and to maintain consistency. It also assumes that servers have access to a logging infrastructure — similar to the logging mechanisms specified by FT-CORBA — for storing, and retrieving, information. The log must be accessible by all the replicas of an object and support transactional operation. It can be implemented by various mechanisms, such as a database, or communication primitives that guarantee atomic exchange of data among replicas at commit time.

The important feature of this architecture is the fact that the middle-tier servers perform the core processing, and initiate transactions on the third-tier database servers, which store and persist the data. We exploit the FT-CORBA infrastructure to handle the server replication and client-side failover. We then exploit the OTS mechanisms to enable the servers to perform their processing (to handle a client request) in the scope of nested transactions that they initiate. Thus, as emphasized in the following sections, we "marry" the best of the OTS and FT-CORBA mechanisms to achieve end-to-end reliability and availability all the way from the first-tier client to the third-tier database.

4.1 Client-Side Mechanisms

Server objects are passively replicated and hosted by an FT-CORBA infrastructure. A replicated server is represented by a multi-profile CORBA Interoperable Object Reference (IOR), with each profile enumerating the address (host name, port number, object key) of a server replica. A non-replicated first-tier client addresses the middle-tier primary server replica using this fault-tolerant object reference.

On its part, the client does not need to perform any additional processing, apart from conforming to the client-side FT-CORBA specification. According to the FT-CORBA standard, the client-side ORB runtime first invokes the primary server replica. If the primary is suspected to have failed, then, the client-side ORB runtime transparently iterates through the addresses contained in the multi-profile IOR, invoking each address in search of an operational server replica.

A `ServiceContext` field,[3] embedded within the request by the client-side ORB runtime, contains a unique request identifier that permits the middle-tier servers to detect if the request is a duplicate, i.e., it has been seen before. This allows the server-side ORB runtimes to detect, and discard, duplicate requests and, therefore, to prevent the server state from being corrupted. Note that the client's invocation does *not* need to execute in the context of a transaction, or use OTS at all (thereby eliminating the need for embedding transactional service context within the client's request). Thus, the only service context information carried in the request from the first-tier to the middle-tier is that for FT-CORBA duplicate-request detection.

4.2 Server-Side Mechanisms

On the server side, the protocol relies on OTS and FT-CORBA to achieve both consistency and availability. Consistency is implemented not only through the OTS' mechanisms, but also through the server-side fault-tolerant ORB runtime. If a client mistakenly invokes a backup replica (rather than the primary replica), the server-side FT-CORBA runtime intercepts the request (before it reaches the server replica), and transparently notifies the client of the primary's identity using a LOCATION_FORWARD reply message. The client-side ORB runtime can use the address embedded within that message to contact the real primary replica. The client request is never executed on a non-primary replica, thereby ensuring that the states of the server replicas are not rendered inconsistent by accidental diversion of requests to the wrong server replica.

On receiving a request from a first-tier client, the primary first checks in the log to see if the request has already been processed (the unique duplicate-detection context embedded in the client's request is used precisely for this purpose). If the current client request is a duplicate, the primary returns the previously generated (and cached) reply. If this is a fresh non-duplicate request, the primary server replica initiates an OTS transaction. Note that, in a multi-tier architecture, this might be a nested transaction, if the incoming request was already part of another transaction. The scope of the transaction encloses all the operations performed by the replica, including any interactions with other components.

The client's request is then processed within the scope of this server-initiated transaction. By hosting the middle-tier servers over an FT-CORBA infrastruc-

[3] CORBA allows a client to propagate additional information to the server, in order to influence the processing of a specific invocation. This additional information is embedded into the `ServiceContext` field of an IIOP invocation message, and is interpreted by the server-side ORB runtime.

ture that is OTS-enabled, the servers can initiate transactions, while benefiting from the FT-CORBA mechanisms. This allows us to handle both stateless and stateful servers in the middle-tier, as opposed to most three-tier enterprise architectures, which typically handle only stateless servers. If the server object is stateful, then, the state (or state update) is retrieved from the primary replica through the `Checkpointable` (or `Updateable`) interface that every object must support, according to the FT-CORBA standard. This state is written to the log, together with the response for the client. The replica then commits the transaction and returns the response to the client.

If the primary fails before committing, the transaction rolls back and undoes all of the operations performed by the failed replica (the complete invocation chain whose root is the failed primary is "rolled back", including data written to the log). If the primary fails after committing, the reply is available at the backup replicas, one of which is elected as the new primary and will return this reply if the invocation is encountered again.

This protocol applies recursively to further tiers. Thanks to the use of nested transactions, a failure can be contained into just one branch of the invocation tree and a roll-forward strategy can be used within the scope of that branch.

This scheme is best illustrated by an example. Consider the bank application introduced in Section 3.3. Figure 5 shows a run of the protocol with two failures. The client C first sends a transfer request to the replicated bank object B. The primary replica, B_1, starts an transaction TX_1 and sends a request for withdrawal to account X. In turn, the primary replica of the account, X_1, starts a transactions $TX_{1.1}$ nested within TX_1, performs the withdrawal, logs the state update and the response, commits the transaction, and returns the response to B_1, which fails before receiving the response. As B_1 fails before committing TX_1, the transaction rolls back. As a consequence, nested transaction $TX_{1.1}$ also rolls back and the state of the account is reverted to its previous value. At the time B_2 takes over, the global state of the system is consistent thanks to the roll-back recovery mechanism of OTS.

The client runtime of the FT ORB at C detects the failure of B_1 and transparently re-issue the transfer request to the second profile in the IOR, B_2. This roll-forward mechanism effectively shields the client from the failure of the server. B_2 notices that the transfer request has not yet been processed — there is no associated entry in the log. Therefore, B_2 initiates a new top-level transaction TX_2 and invokes the primary replica X_1. X_1 performs a withdrawal in the context of a new nested transaction $TX_{2.1}$, returns a reply, and fails (after committing the transaction). Note that, in this scenario, the failure occurs at a 3^{rd} tier server and transaction $TX_{2.1}$ does not roll-back. The FT ORB runtime at B detects the failure and re-issues the request to the backup server X_2, again using a roll-forward mechanism. Since transaction $TX_{2.1}$ completed successfully, X_2 detects that the request has already been processed, fetches the state update and the reply from the log, updates its state, and returns the previous reply to B_2. Finally, B_2 performs the deposit on account Y, logs the response, commits transaction TX_2, and returns the response to the client.

The key idea here is the combination of the replication and transaction models. The beauty of the server-initiated OTS transaction scope is that any fault

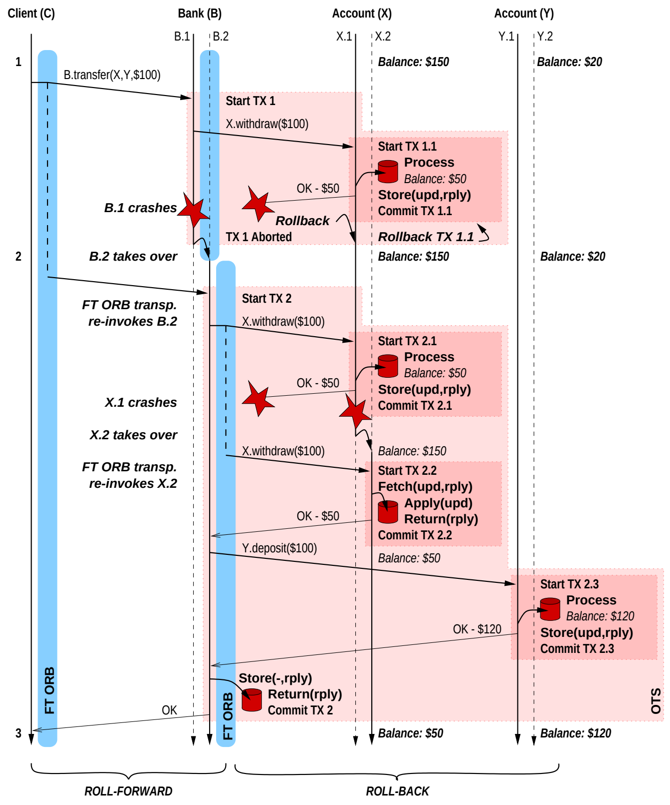

Fig. 5. Protocol Run with Failures in the 2nd and 3rd Tiers.

that occurs during the server's processing of the client's request will not violate data consistency (because it merely triggers a roll-back of the processing and state changes). At the same time, because the server is replicated, and the client-side ORB runtime is equipped with the FT-CORBA failover infrastructure, the crash of a server replica does not lead to loss of availability, either. Thus, the best of both worlds — replication for availability (liveness) and transactions for consistency (safety) — is achieved.

5 Related Work

There exist several commercial implementations and research prototypes of the FT-CORBA specification, as well as of the OTS specification. These implementations employ either the roll-back or the roll-forward reliability strategies, but do not attempt to reconcile the two different approaches. Because our focus in this paper is on the *integration* of the two approaches, and not on systems that satisfy only one of the two approaches, we will not discuss the merits and drawbacks of the various FT-CORBA and OTS implementations.

Instead, we will highlight other research efforts that have attempted to solve specific aspects of this integration problem, albeit from a different viewpoint. Related efforts include research on integrating transactional protocols with group communication protocols, where group communication forms the foundation for maintaining consistency in most replication-based systems.

GroupTransactions [12] aim to take advantage of both group communication and transactions through a new transactional model, where transactional servers can, in fact, be groups of processes. This allows for transactional applications to be built on top of computer clusters.

An e-Transaction [3] is one that executes exactly once despite failures, and is targeted at three-tier enterprise architectures with stateless middle-tier servers that are replicated. This overcomes the limitations of current transactional technologies that, for the most part, ensure at-most-once request processing, which is not sufficiently reliable. The e-Transaction abstraction builds upon an asynchronous replication scheme that provides both the liveness feature of replication, as well as the safety feature of transactions.

Another CORBA-related effort [6] aims to compare the two different kinds of systems — one with group communication and no transactions, and the other with transactions and no group communication — from the viewpoint of replicating objects for availability. Their study leads them to conclude that although transactions are effective in their own right, using group communication infrastructures to support transactional applications can lead to benefits, such as faster failover in the event of a fault.

While all of the above replication schemes refer to objects or servers, the notion of integrating group communication into a transactional model has been extended to the replication of the entire database itself [5]. This work attempts to eliminate the centralized and, therefore, unreliable approach that databases adopt today. The proposed family of replication protocols exploit group communication semantics to eliminate deadlocks, improve performance, and enhance reliability.

IBM Research's Dependency-Spheres [15] aims to integrate (asynchronous) messaging and (synchronous) transactions for distributed objects, with the intention of increasing the level of reliability provided for enterprise Web Services. Dependency-Spheres provide a new kind of global transaction context that allows both synchronous and asynchronous distributed messaging style exchanges to occur within a single transaction-like operation.

To the best of our knowledge, our research represents the first use of transactional mechanisms to implement replication, and to address the determinism problem of nested interactions between replicated objects.

6 Conclusion

Today's enterprise applications have a three-tier structure, with simple clients in the first tier, servers in the middle-tier to perform the processing, and databases in the third tier to store information. For such enterprise applications, replication is required to protect the middle-tier processing, while transactions are required to protect the third-tier data. The CORBA middleware standard now incorporates support for reliability through these two distinct mechanisms — roll-forward replication (using the new Fault Tolerant CORBA standard) and roll-forward transactions (using the CORBA Object Transaction Service). In the current state-of-the-art and state-of-the-practice, it is difficult to reconcile these two techniques.

For true reliability, however, enterprise applications clearly require elements of both roll-forward reliability (to protect processing, and for liveness) and roll-back reliability (to protect data, and for safety). In this paper, we presented a novel combination of replication and transactions to achieve the best of both worlds, and to obtain end-to-end consistency and availability all the way from the first-tier client to the third-tier database.

We exploit the FT-CORBA infrastructure to handle the server replication and client-side failover. We then exploit the OTS mechanisms to enable the servers to perform their processing in the scope of nested transactions that they initiate. To our knowledge, this is the first use of transactional mechanisms to implement replication and to address the determinism problem of nested interactions between replicated objects. Although our solution has been presented in the context of CORBA, it is equally applicable to other transactional environments.

Early results from experimental evaluation with off-the-shelf ORBs demonstrate that little effort is required to combine replication and transaction in a real-world application, and the overhead remains small under normal operation. In the presence of failures, the performance of the replication and recovery mechanisms strongly depends on where and when the failures occur — a failure occurring in the context of a top-level transaction will force a roll-back, which might be costly when many resources are involved in the transaction. In addition, the overhead of recovery is highly dependent of the quality of the FT-CORBA and OTS implementations, and in particular the performance and accuracy of their monitoring mechanisms.

References

1. N. Budhiraja, K. Marzullo, F. Schneider, and S. Toueg. *Distributed Systems*, chapter 8: The Primary-Backup Approach, pages 199–216. 2nd edition, 1993.
2. P. Felber. *The CORBA Object Group Service: A Service Approach to Object Groups in CORBA.* PhD thesis, Swiss Federal Institute of Technology, Lausanne, Switzerland, 1998.
3. S. Frolund and R. Guerraoui, "Implementing e-Transactions with Asynchronous Replication," *IEEE Transactions on Parallel and Distributed Systems*, vol. 12, no. 2, pp. 133-146 (2001).
4. IBM Global Services. *Improving Systems Availability*, 1998.
5. B. Kemme and G. Alonso, "A Suite of Database Replication Protocols Based on Group Communication Primitives," *Proceedings of the IEEE International Conference on Distributed Computing Systems*, Amsterdam, pp. 156-163 (May 1998).

6. M. C. Little and S. K. Shrivastava, "Integrating Group Communication with Transactions for Implementing Persistent Replicated Objects," *Lecture Notes in Computer Science*, vol 1752, Springer-Verlag, 2000.
7. P. Narasimhan. *Transparent Fault Tolerance for CORBA*. PhD thesis, Department of Electrical and Computer Engineering, University of California, Santa Barbara, December 1999.
8. Object Management Group. *The Common Object Request Broker: Architecture and Specification*, 2.6 edition, OMG Technical Committee Document formal/02-01-02, January 2002.
9. Object Management Group. *Fault Tolerant CORBA*, OMG Technical Committee Document formal/01-12-29, December 2001.
10. Object Management Group. *Object Transaction Service Specification*, OMG Technical Committee Document formal/01-11-03, May 2001.
11. Object Management Group. *The CORBA Component Model,* OMG Technical Committee Document ptc/01-11-02, January 2002.
12. M. Patino-Martinez, R. Jimenez-Peris and S. Arevalo, "Group Transactions: An Integrated Approach to Transactions and Group Communication," *Concurrency in Dependable Computing*, Kluwer Academic Publishers, 2002.
13. F. Schneider. *Distributed Systems*, chapter 7: Replication Management using the State-Machine Approach, pages 169–197. 2nd edition, 1993.
14. Sun Microsystems. *Enterprise Java Beans*, version 2.0.
15. S. Tai, T. A. Mikalsen, I. Rouvellou and S. M. Sutton, Jr., "Dependency-Spheres: A Global Transaction Context for Distributed Objects and Messages", *Proceedings of the 5th IEEE International Enterprise Distributed Object Computing Conference*, Seattle, WA (September 2001).