Practical Considerations in Making CORBA Services Fault-Tolerant

Priya Narasimhan
Institute of Software Research International
School of Computer Science
Carnegie Mellon University, Pittsburgh, PA 15213-3890
priya@cs.cmu.edu

Abstract

This paper examines the CORBA Naming, Event, Notification, Trading, Time and Security Services, with the objective of identifying the issues that must be addressed in order make these Services fault-tolerant. The reliability considerations for each of these Services involves strategies for replicating the Service objects, and for keeping the states of the replicas consistent. Of particular interest are the sources of non-determinism in each of these Services, along with the means for addressing the non-deterministic behavior in the interests of ensuring strong fault tolerance.

1 Introduction

The integration of distributed computing with object-oriented programming leads to distributed object computing, where objects are distributed across machines, with client objects invoking operations on, and receiving responses from, remote server objects. Both the client's invocations, and the server's responses, are conveyed in the form of messages sent across the network. The Common Object Request Broker Architecture (CORBA) [18], was established by the Object Management Group, as a standard for distributed object computing.

CORBA uses a purely declarative language, the OMG Interface Definition Language (IDL), to define interfaces to objects. The IDL interfaces are subsequently mapped, through an IDL compiler provided by an implementor of CORBA, onto specific programming languages, such as C, C++, Java or Smalltalk. CORBA's language transparency implies that clients need to be aware of only the IDL interface, and never of the language-specific implementation, of a server object. CORBA's interoperability implies that a client can interact with a server object, despite differences in their platforms and operating systems. CORBA's location transparency implies that client objects can invoke server objects, without worrying about the actual locations of the server objects. The key component of CORBA, the Object Request Broker (ORB), acts as an intermediary between the client and the server objects, and shields them from differences in platform, programming language and location.

CORBA encompasses a rich suite of Services, including Naming, Event, Notification, Time, Trading and Security services, also specified through IDL interfaces, and typically implemented by ORB vendors. This frees application developers from having to write the code for such commonly-used functionalities. More recently, the Object Management Group adopted a Fault-Tolerant CORBA standard [15] that allows CORBA applications to be made reliable through the replication of the application objects, along with mechanisms and services for replica consistency, fault detection and recovery. The Fault-Tolerant CORBA standard requires applications to support certain interfaces, and to exhibit deterministic behavior. Unfortunately, because most of the CORBA Services were specified and implemented well before the Fault Tolerant CORBA standard emerged, they do not necessarily exhibit the kind of behavior that the standard mandates, and do not lend themselves readily to fault tolerance.

This makes it difficult for application developers who have long enjoyed the freedom and the ability to use any of the CORBA Services – by introducing an unreliable implementation of a CORBA Service into their application, these application programmers risk the failure of their applications. Thus, CORBA developers are forced to reconsider, and perhaps even to forgo, employing these Services if fault tolerance is essential for the operation of their application.

There has been considerable work in implementing fault-tolerant CORBA systems, such as AQuA [1], DOORS [10], Electra [5], Eternal [9], FRIENDS [2], FTS [4], IRL [7], Newtop [8] and OGS [3]. For the most part, these systems enhance normal CORBA applications with fault tolerance – to the best of my knowledge, these systems have addressed the challenges surrounding reliability for CORBA Services. However, a fault-tolerant name server [6] was proposed for a very early version of CORBA using the Electra ORB.

This paper takes an objective look at some of the more commonly used CORBA Services, purely from the viewpoint of fault tolerance. The focus of this study is to identify the reliability issues and concerns that the specifications and the implementations of these Services raise and, more importantly, to present solutions that overcome the identified fault tolerance problems in these Services.

2 Reliability for CORBA Services

Each CORBA Service is specified through a set of well-defined IDL interfaces, and realized through one or more CORBA server objects which can be invoked either locally or remotely. The application typically exploits a CORBA Service by acting as the client of the Service's objects. In keeping with the spirit of CORBA, both the specification and the implementation of a CORBA Service are independent of the type, nature, or location of its clients.

When a CORBA application that wishes to exploit a CORBA Service, clearly, the objects of the application must interact with, invoke, and respond to, the Service objects. Thus, for the end-to-end dependability of the CORBA application, the reliability of the application, as well as that of the CORBA Service that it uses, must be considered. To protect the application against fault using the Fault Tolerant CORBA standard, both the application objects and the Service objects must be replicated, with the replicas distributed across different processors in the distributed system.

For specific cases, the straightforward replication of the Service objects is simply not sufficient to guarantee fault tolerance, primarily because the intrinsic behavior of the Service objects leads to problems when the objects are replicated. Therefore, it is essential to identify the sources of the undesirable side-effects that replication triggers in these Services, as well as to find strategies to overcome these side-effects. In this paper, the dependability of each Service is assessed along different aspects.

Dependability architecture. For each Service, only some of its objects need to be replicated; the remaining objects can be regarded as pure unreplicated clients. To avoid a single point of failure, individual replicas of an objects must be allocated onto different processors. In keeping with the Fault Tolerant CORBA standard, each object that is to be replicated must support the `Monitorable` interface (to allow it to be "pinged" periodically for liveness) and the `Checkpointable` interface (to allow its state to be captured periodically, and restored later during recovery). A part of the dependability architecture involves identifying the nature of the operations of the Service objects (whether they are idempotent or not) in order to determine whether support for detecting and suppressing duplicate operations (which are inevitable in a fault-tolerant system that uses replication) is required.

State. For a true fault-tolerant system, it is also necessary to enable the failed replicas of an object to be re-instated to normal operation. If a Service object's replica has to be recovered, its state must first be initialized to that of the other functioning replicas of the object. For this to be possible, we must first be able to define the state of each of the Service objects, and then transfer it (through the `Checkpointable` interface) to each new or recovering replica. Of course, an important factor in the definition of state is the consideration of performance-related issues, such as the size of the state, and the distribution of the entire Service's state across its different constituent objects. These issues ultimately determine the speed of recovery of a replica of a Service object.

Non-determinism. A frequent assumption in providing reliability for a component is the deterministic behavior of the component. This means that, if the component starts from a specified initial state, and is fed a sequence of inputs, the component should reach the same final state. It is this reproducible behavior of the component that lends itself well to reliability techniques. Unfortunately, many applications, and the CORBA Services themselves, do exhibit non-deterministic behavior. Common sources of non-determinism include local timers (*e.g.*, when an application invokes `gettimeofday`), input-output to local devices (*e.g.*, CDROM drives, floppy drives), multithreading, etc. For each of the CORBA Services that we consider, we must also identify the non-deterministic features of its constituent objects, and then discover ways of "sanitizing" these features.

2.1 CORBA Naming Service

Simplistic CORBA applications publish the server's object reference into a file, which the client then reads to discover the server's location, and to contact the server subsequently. Clearly, this is not a satisfactory mechanism for field deployment, particularly because we cannot assume that both the client and the server will have access to the same shared file system. To support the run-time discovery of server object references in a much more elegant and scalable manner, CORBA provides for a Naming Service.

The CORBA Naming Service [16] allows the application programmer to associate the server objects of the application with human-interpretable and more intuitive stringified *names*. A name is represented by a CORBA `Name` object, which is constructed from a hierarchical sequence of name components, each component being uniquely identifiable within the naming context of its preceding component, much in the way that file systems organize directories and files within directories; for example, the name */usr/bin* contains the component *bin*, which is uniquely identifiable only within the naming context of its hierarchically preceding component, *usr*. For each of its server objects, an application can register, or *bind*, a `Name` object within the Naming Service's repository; a client of one of the server objects can present the corresponding `Name` to the Naming Service, which will then *resolve* the embedded name components to discover the server's reference, and to return the reference to the client. Armed with the server's reference, the client can now contact the server, without needing to contact the Naming Service further (unless the client requires the reference to some other server). The advantage is that while a server's reference (containing the server's hostname and port number) can change over its lifetime, the server's name can be maintained relatively constant over the server's lifetime.

Typically, the Naming Service is implemented as a single server object that supports three interfaces: the `NamingContext` (to bind, re-bind, unbind, resolve, list and destroy names of objects) the `BindingIterator` (to traverse sequentially the list of names stored within the Naming Service) and the `NamingContextExt` (to convert between the Naming Service's representation of a name and a human-readable name).

Dependability architecture. The fault tolerance architecture for the Naming Service is relatively straightforward because there is typically only one server process that needs to be replicated, with multiple replicas distributed across distinct processors. To enable the recovery and the fault detection of the Naming Service, each of the three server objects, `NamingContext`, `BindingIterator` and `NamingContextExt` must support both the `Checkpointable` and the `Monitorable` interfaces, as required by the Fault Tolerant CORBA standard. For the sake of efficiency, it might be appropriate to implement a fourth collocated CORBA server object, called `FTCorbaSupport` that supports only the `Checkpointable` and the `Monitorable` interfaces, and whose only function is to act as a front-end, handling all of the fault detection and recovery communication on behalf of the other three objects. Some of the Naming Service's methods are idempotent, $e.g.$, two consecutive executions of the `resolve()` operation will yield identical results. However, the Naming Service's interfaces do contain non-idempotent operations, $e.g.$, the `delete()` operation is likely to lead to exceptions if performed twice in a row. Thus, there is the need to log and to identify uniquely every operation before allowing the Naming Service to execute it.

State. The Naming Service can contain a large amount of state, depending on the size and the number of names that have been registered with it. At the point of recovering a new or a failed replica of the Naming Service, this entire state first needs to be retrieved from an existing operational Naming Service replica. The time to retrieve the state and, therefore, the time to recover a new replica, will depend on the format in which the names are stored. Note that, over the course of its lifetime, the Naming Service instantiates many internal CORBA objects, particularly the `Name` objects that have been registered with it. Thus, when a new replica of the Naming Service is started, its recovery involves not only transferring the list of `Name` objects registered with an existing replica, but also painstakingly instantiating each of the name-related CORBA objects within the new replica. In the interests of speeding up recovery, "smart" name storage and list traversal mechanisms that improve performance could, of course, be used to result in the efficient retrieval and transfer of the Naming Service's state. However, the performance impact on recovery time due to the instantiation of possibly hundreds of `Name` objects on the launch of a new Naming Service replica is a significant source of concern.

Non-determinism. Another concern is the possibility of non-deterministic behavior in the implementation of the Naming Service. It is likely that the Naming Service will have many application clients and servers (otherwise, there would have been no need for a CORBA Naming Service because the simpler but less scalable strategy like the shared file between the application's clients and servers would have sufficed). To maintain its scalability and its ability to handle many clients simultaneously without substantial performance degradation, most Naming Service implementations are multithreaded. Despite its many performance advantages, multithreading is a significant source of non-determinism if threads can modify shared data; unfortunately, the list of registered names stored within the Naming Service constitutes data that threads might share. Thus, the possibly different orders of thread execution within any two replicas of the Naming Service can lead to inconsistencies in their states. One solution is to allow concurrent operations within the Name Server only if the operations do not update any shared state. This implies that multiple read-only operations ($e.g.$, `resolve()`) can proceed concurrently, but update operations ($e.g.$, a `delete()` and a `bind()`) cannot execute concurrently because they might modify the shared data in different orders at different replicas of the Name Server.

2.2 CORBA Event and Notification Services

The CORBA Event Service [14] provides support for decoupled distributed communication between the CORBA objects of an application. There exist *suppliers* that generate events, and *consumers* that have an interest in receiving these events. The events may propagate from the suppliers to the consumers through either a pull model, where the consumers request events of the suppliers, or a push model, where the suppliers automatically dispatch the events to the suppliers, as they occur. An *event channel* is an intermediary object that allows multiple consumers and multiple suppliers to communicate with each other asynchronously.

In their mediating role, event channels function simultaneously as both consumers and suppliers of events. Because the decoupling of the supplier-to-consumer communication occurs through the event channels, the suppliers and consumers of events can produce and receive events, respectively, without knowing of each others' identities, locations or types. The CORBA Event Service specification allows for disconnected operation, where consumers and suppliers may be unable to reach other sporadically. Unfortunately, when the Event Service is used as a part of a CORBA application that must be reliable, the intermittent operation of the Event Service objects, accompanied by the possible loss of event data, is unacceptable. In contrast to the two-tiered Naming Service, the Event Service represents a more challenging three-tiered architecture, with the suppliers (first tier) communicating with the event channels (second tier), which, in turn, communicate with the consumers (third tier).

The CORBA Notification Service [11] extends the CORBA Event Service with support for specific kinds of notification, including the filtering of events, the definition of events as structured data types, the ability to define quality of service attributes on a per-channel or per-event basis, *etc.* From a fault tolerance viewpoint, the Notification and Event Services are very much alike in architecture and in the reliability issues that they present. Thus, the reliability considerations for the Event Service are equally applicable to the Notification Service.

Dependability architecture. Clearly, the most critical elements of the Event Service is the event channel. Thus, the event channel objects must be replicated, with the replicas distributed across the processors in the system. To enable their recovery and fault detection, the event channel objects of the Event Service server object must support both the `Checkpointable` and the `Monitorable` interfaces, respectively, in accordance with the Fault Tolerant CORBA standard.

State. The event channel maintains some state, in order to perform the co-ordination between the suppliers and the consumers. A part of this state includes the event data that the consumers require, as well as the identities of the consumers that have already received this data, and the identities of those that are yet to receive this data. When a new replica of the event channel is started, in addition to receiving this state from an existing replica, the new replica must also be provided with the order in which the existing replica has formed connections with its event suppliers and event consumers. This is important because if any two replicas of the event channel have established connections in different orders, then, the two replicas will detect I/O activity on those connections in different orders (by the very nature of the operating system's I/O polling routines, such as `select()`, which polls connections in the order in which they were formed). Thus, the order in which connections to the suppliers and consumers were established must be made known, and respected, at the new replica of the event channel. In the case of the Notification Service, a new replica needs to instantiate, and then register, the appropriate notification filters in the right order. The recovery of a new event channel server object might be time-consuming because of the overhead of instantiating multiple objects. Furthermore, because the event channel plays the roles of both client (to the consumers) and server (to the suppliers), the client-side ORB-level state [9], which typically includes the value of request identifier that is encapsulated into the most recent outgoing IIOP request message, must be recovered and restored at a new event channel replica.

Non-determinism. Because the event channel must handle multiple consumers and suppliers simultaneously, it is often multithreaded. As with the Naming Service, the multithreading of the event channel represents a source of non-determinism. One solution to this problem would be to serialize all of the operations within the event channel object, thereby ensuring that all of its replicas behave deterministically and, therefore, reach the same states as they execute method invocations. However, this might be rather an extreme solution, particularly because of its implications on performance; a better alternative might be to ensure that the event channel objects are implemented in the following ways so that no two concurrently executing operations modify or access shared state:

- The event channel must use distinct internal objects (each object with its exclusive region of allocated memory) to handle the different suppliers and consumers, *i.e.*, for every supplier or consumer that it handles, the event channel should restrict the interactions with that supplier or consumer to a distinct region of memory within its process.
- The event channel must adopt a thread-per-object model for each of its constituent objects (where each object represents a consumer or supplier of events). This ensures that there will be at most one thread accessing any object within the event channel at any time. The implication of this is that the event channel can allow for the simultaneous execution of any two operations, as long as both do not pertain to the same consumer or supplier.
- Any operations on the event channel that must deal with all of the producers and consumers at once (for instance, an operation that deletes all of the consumer-related objects) must be allowed to execute only when all of the other threads have quiesced, and all of the other operations currently being executed by the event channel have completed.

2.3 CORBA Trading Object Service

The CORBA Trading Service [13] facilitates the offering and subsequent discovery of the instances of services of particular types. In this context, a *trader* is an object that advertises the capabilities of other objects in the system, and allows objects to match their needs against advertised capabilities. An object that desires to advertise, or export, a service simply contacts the trader with a description of the service, and the location in the system of an interface that supports the service. An object desirous of using, or importing, a service contacts the trader with a description of the characteristics of the service that it seeks, and the trader responds with the location of such a service, if it exists. One of the challenging aspects of a trading service is the fact that its data, or state, can be partitioned into multiple different servers. The reason for such an implementation of the trader is that it is likely that service offerings themselves can be partitioned into some sensible groupings and, moreover, such a partitioning would make the trading service scalable, and simultaneously accessible to many users. Each partition consists of a group of importers and exporters that are interested in a service with specific kinds of characteristics. If the service offerings within a partition are found to be inadequate, the importers can seek other partitions. This hierarchical structuring of traders is often referred to as a *federation* of traders. Once an importer makes explicit contact with an exporter of a service, the Trading Service can step out of the way.

Dependability architecture. For the Trading Service, each of the traders must be replicated, with replicas distributed across the system. Because there are multiple distinct traders that must communicate with each other, there exist a number of networked replicated objects that hold some information about each other. Each trader object must support the `Checkpointable` and `Monitorable` interfaces.

State. Each trader must maintain links to other traders in its federation, and must also maintain exhaustive lists of the services that exporters in its partition have registered. The trader must maintain information about its exporters and its importers. The state of each trader also consists of the list of traders offering services outside of its partition. Thus, when a new trader replica needs to be recovered, this state must be restored, as well as its connections with other traders in the system. The order of registration of, and connection establishment to, external traders within the federation is important for recovery, because it decides the search order of services. This order must be captured and restored to new trader replicas. Furthermore, because each trader plays the roles of both client (to the traders outside its partition) and server (to the exporters and importers within its partition), the client-side ORB-level state, which typically includes the value of request identifier that is encapsulated into the most recent outgoing IIOP request message, must be recovered and restored at a new trader replica.

Non-determinism. A significant reliability concern with the trading service is the multithreading that is inevitable because the trader must support multiple importers and exporters. The trader supports two different kinds of operations – read-only operations when an importer looks up a service offering, and update operations, when an exporter registers on unregisters a service. Read-only operations can proceed concurrently because they do not modify any shared state across different threads; however, the update operations are trickier because they manipulate the list of services registered with the trader. Furthermore, because the trader pairs up an importer with an exporter, both of which are explicitly aware of each other (unlike the Event Service, where the consumer and supplier are decoupled from each other), there exists the possibility that certain trader implementations use short-cuts to share data between an importer and an exporter, in the interests of efficiency. Care must be taken in the implementation of the Trading Service to respect the protection of memory across multiple executing threads that update shared data; in the absence of such guarantees, the serialization of invocations on the trader, and its negative impact on performance, are unavoidable.

2.4 CORBA Security Service

The CORBA Security Service [17] allows the protection of CORBA objects in the system, allowing authorized users to access objects, protecting confidential data from access by unauthorized users, ensuring that users cannot masquerade as each other and gain illegal access to data, and protecting data and inter-object communication from being tampered by users. To this end, the security interfaces proposed by the specification are often not visible to the application as they are buried within the ORB. Thus, an ORB that purports to support the CORBA Security Service specifications is significantly different from an insecure ORB, and embeds mechanisms for audit, authentication, access control, repudiation and secure messaging.

Security and reliability are pervasive system properties in the sense that they both require end-to-end consideration of not only the application, but the ORB, the processors, the network, and the distributed system itself. Thus, in contrast to the other CORBA Services, which merely provide additional functionality to an existing application, the Security Service has more far-reaching implications on the system architecture and system properties. Furthermore, security and reliability are sometimes conflicting goals, and the resulting trade-offs must be taken into account when the two properties are simultaneously required.

Reliability Considerations One of the first sources of conflict between the Security Service's definition and reliability is the fact that the Security Service rightfully does not allow for the tampering of communication between objects, in the interests of ensuring the secure exchange of messages. Unfortunately, reliability necessarily requires some degree of probing at the message level, because a reliable system must protect the application from message loss, and must be able to ascertain if messages are duplicates of each other, *etc.*

This calls for a secure way of allowing messages to be inspected and possibly modified, in the interests of reliability. One approach to this is to use Portable Interceptors, which form a part of standard CORBA implementations. Portable interceptors can be used to examine requests, replies and messages in the system. This requires the Security Service to support the concept of portable interceptors, and furthermore, to allow for requests, replies and messages to be manipulated, before enforcing any security policies on them.

Another source of conflict is that the presence of corruption in the system presents a source of non-determinism. For instance, it might be possible for a three-way replicated object to have one of its replicas running on a maliciously corrupted processor, and to have its other two replicas running on correctly functioning processors. In this case, it is likely that the corrupted replica will behave differently from the correct replicas, and could generate different responses to incoming requests. Thus, in the presence of corruption, determinism can no longer be guaranteed. To protect against corruption, and simultaneously to eliminate the effect of non-determinism, some form of majority voting on the messages sent by the replicas of an object is required. Thus, both the objects of the Security Service, as well as the objects of the application, must be actively replicated,[1] with majority voting applied on both their requests and responses. On the other hand, the introduction of replication also poses a threat to security; the fact that there exist identical copies of data distributed across the system increases the possibility of compromise of the data through any one of the multiple copies. These are the kinds of trade-offs that must be taken into account when incorporating the Security Service into an application that is simultaneously required to be reliable.

Another reliability consideration is the fact that a secure ORB contains additional state, over and above a normal CORBA-compliant ORB. This state might comprise the ORB's security policies, the list of authenticated users, the cryptographic mechanisms required for authentication, *etc.*. In the case of recovering a new replica of an application object, this additional ORB-level state must be taken into consideration. Of course, we must exercise caution in transferring this ORB state onto a fresh processor that has not yet been authenticated. As a part of initializing the ORB of a new replica, we might need to have the new replica, its ORB, and its hosting processor authenticated before allowing the new replica to be re-instated to normal operation.

[1]With active replication, all of the replicas receive, process, and respond to all invocations.

2.5 CORBA Time Service

The Time Service [12] provides for a way for CORBA application objects to obtain the current time, or to obtain a time interval that they can use for other purposes. With the CORBA Time Service, the `TimeService` object manages Universal Time Objects (UTOs) and Time Interval Objects (TIOs). Each UTO represents a time, and each TIO represents a time interval. The Time Service returns the time, along with an uncertainty interval. The CORBA Time Service acknowledges that the time obtained by a client of the `TimeService` object may be obsolete by the time that the client receives the response over the network. Possible solutions that have been proposed to overcome this problem include the provision of a local `TimeService` object collocated within the process of each of its clients.

The `TimerEventService` object manages timer-event handler objects. Each timer-event handler is born associated with an push-mode event channel (one of the components of the CORBA Event Service, as described in Section 2.2). The timer-event handler can be used to manipulate the timing and content of the events associated with its event channel. Typically, at the expiration of the timers, events will be pushed through the associated event channel to all interested event consumers.

Dependability architecture. The critical elements of the CORBA Time Service are the `TimeService` and the `TimerEventService` objects. If these are in separate server processes, then, each of the two objects needs to support the `Checkpointable` and `Monitorable` interfaces, and both server processes should then be replicated. It might happen that both the `TimeService` and the `TimerEventService` objects are collocated within the same process, in which case, the process itself can be replicated. In this case, for the sake of efficiency, a third object supporting the `Checkpointable` and `Monitorable` interfaces can handle the fault detection and the state transfer on behalf of the two Time Service-related objects.

State. Each of these two objects manages other CORBA Time Service objects, including the TIOs, UTOs and timer-event handler objects. Thus, there is some amount of state that is associated with the `TimeService` and the `TimerEventService` objects. Because the `TimeService` and the `TimerEventService` create additional CORBA objects within their processes, their state must include the list of these additionally created objects and the order in which these objects were created. To recover a new `TimeService` replica, the list of the TIOs and UTOs managed by an existing replica must be transferred to the

CORBA Service	Replication Architecture	State	Non-Determinism
Naming	Replicate the server process (containing the three objects, `NamingContext`, `BindingIterator` and `NamingContextExt`). Add a fourth collocated server object solely to support the `Checkpointable` and `Monitorable` interfaces, and to handle all the fault detection and checkpointing for the other three objects.	List of registered names, as well as the individual `Name` objects, need to be restored within every new replica. This might be time-consuming because several `Name` objects might need to be instantiated during recovery.	To overcome non-determinism due to multiple threads updating shared data (such as the list of registered names), allow read-only operations (*e.g.*, `resolve()`) to proceed concurrently, but not update operations (*e.g.*, a `delete()` and a `bind()`).
Event and Notification	Replicate the event channel server process, and ensure that the event channel object supports the `Checkpointable` and `Monitorable` interfaces.	Identities of consumers, event data that needs to be sent, selective notification event filters, as well as ORB-level state, such as the order of connection establishment with consumers and suppliers, and the client-side ORB's most recent outgoing request identifier to each consumer	To overcome non-determinism due to multithreading, implement the event channel object to (i) create distinct objects to represent each supplier and consumer internally, (ii) use a thread-per-object model, and (iii) serialize concurrent operations that necessarily share data.
Trading Object	Replicate each of the multiple trader objects within the federation, and ensure that each trader object `Checkpointable` and `Monitorable` interfaces.	List of services registered by exporters in its partition, list of traders (and their service offerings) in other partitions, as well as ORB-level state, consisting of the order of connection establishment with external traders, and the client-side ORB's most recent outgoing request identifier to each external trader.	Read-only service lookup operations from importers can proceed concurrently, while service registration and unregistration operations from exporters must be serialized because they modify shared state (the trader's list of service offerings).
Security	Because the Security Service is built into the ORB, dependability requires the replication of objects that use a secure ORB.	ORB's security policies, list of authenticated users, cryptographic mechanisms, all of which are buried within the ORB, and not easy to access from the application.	Security requires per-host-level authentication, which is a source of non-determinism. Active replication with majority voting can tolerate value faults, and hide processor-level security violations.
Time	Replicate the `TimeService` and the `TimerEventService` objects, and have them both support the `Checkpointable` and `Monitorable` interfaces.	List of TIOs and UTOs for a `TimeService` object, and the list of the timer-event handler objects and the associated event channel objects for a `TimerEventService` object	Passive replication of the `TimeService` object, along with checkpointing of its state, overcomes the non-deterministic nature of time itself. Alternatively, the `TimeService` objects should use a GPS facility.

Table 1: Considerations in providing dependability for various CORBA Services.

new replica, where these TIO and UTO objects must be instantiated afresh within the new replica. To recover a new `TimerEventService` replica, the list of the timer-event handler objects and the associated event channel objects managed by an existing replica must be transferred to the new replica, where these additional objects must be instantiated afresh within the new replica. The performance implications of this object instantiation on the speed of recovery of a new replica of either the `TimeService` or the `TimerEventService` is a significant concern.

Non-determinism. Time, by its very nature, is non-deterministic because the local clocks of different processors in a distributed system cannot be assumed to be always synchronized. Thus, when the Time Service is replicated, and its replicas are distributed across distinct processors in the system, the replicas are likely produce different values of the current time, each based on the local clock of its processor. Clients of the Time Service might receive responses from different replicas (depending on which replica is the fastest) with different invocations of the Time Service. In fact, it is quite possible that successive invocations of the Time Service result in clients obtaining monotonically decreasing values of time!

There are two ways of ensuring that the replication of the Time Service objects does not lead to confusing results at its clients. One approach is to use some kind of Global Positioning Service (GPS) support within the Time Service to avoid dependence on processors' local clocks. Another approach, which assumes the more typical lack of access to a GPS system, is to use passive replication[2] for the ob-

[2]With passive replication, the primary replica processes all invocations, and transfers its state periodically to the backup replicas.

jects of the Time Service so that only one copy of the Time Service determines the time to be returned to the clients. However, the backup replicas also need to be informed of the primary's current value of time so that they can maintain offsets of their respective local times with that of the primary's. Thus, even if the primary fails, and one of the backups takes over as the new primary, the new primary will know the correct value of time to return to the clients.

3 Conclusion

The CORBA standard encompasses the Common Object Services that represent some useful functionality required by a variety of CORBA applications. When these Services form part of a CORBA application that is required to be fault-tolerant, the reliability of the Services is also essential. Unfortunately, most of the CORBA Services were specified and implemented well before the adoption of the Fault Tolerant CORBA standard, with the result that the issues underlying the Services' reliability have been largely ignored.

This paper examines the specifications of the CORBA Naming, Event, Notification, Trading, Time and Security Services, purely from a fault tolerance perspective. For each of these Services, we examine the strategies for replicating the Service objects, and for keeping the states of the replicas consistent. These results include the means of enforcing deterministic behavior of the Service objects, recommendations for implementing these Services, and performance issues that impact the speed of recovery of new or failed replicas of the Service objects. Of all of these Services, perhaps the most challenging is the Security Service because it does not merely add new functionality to an existing CORBA application, but is a system-level property.

Acknowledgments

I would like to thank Rajeev Gandhi for suggesting the idea for this paper, and Luis E. T. Rodrigues for his feedback and his suggestions which greatly improved this paper.

References

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

[2] J. C. Fabre and T. Perennou. A metaobject architecture for fault-tolerant distributed systems: The FRIENDS approach. *IEEE Transactions on Computers*, 47(1):78–95, 1998.

[3] 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.

[4] R. Friedman and E. Hadad. FTS: A high performance CORBA fault tolerance service. In *Proceedings of IEEE Workshop on Object-oriented Real-time Dependable Systems*, San Diego, CA, January 2002.

[5] S. Maffeis. *Run-Time Support for Object-Oriented Distributed Programming*. PhD thesis, University of Zurich, Feb. 1995.

[6] S. Maffeis. A fault-tolerant CORBA name server. In *Proceedings of the IEEE Symposium on Reliable Distributed Systems*, pages 188–197, Niagara-on-the-Lake, Ontario, Canada, 1996.

[7] C. Marchetti, M. Mecella, A. Virgillito, and R. Baldoni. An interoperable replication logic for corba systems. In *Proceedings of the 2nd International Symposium on Distributed Objects and Applications (DOA'00)*, pages 7–16, Feb. 2000.

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

[9] P. Narasimhan. *Transparent Fault Tolerance for CORBA*. PhD thesis, Department of Electrical and Computer Engineering, University of California, Santa Barbara, December 1999.

[10] B. Natarajan, A. Gokhale, S. Yajnik, and D. C. Schmidt. Doors: Towards high-performance fault tolerant corba. In *Proceedings of the 2nd International Symposium on Distributed Objects and Applications (DOA'00)*, pages 39–48, Feb. 2000.

[11] Object Management Group. The Notification Service specification. OMG Technical Committee Document formal/2000-06-20, June 2000.

[12] Object Management Group. The Time Service specification. OMG Technical Committee Document formal/2000-06-26, June 2000.

[13] Object Management Group. The Trading Object Service specification. OMG Technical Committee Document formal/2000-06-27, June 2000.

[14] Object Management Group. The Event Service specification. OMG Technical Committee Document formal/2001-03-01, March 2001.

[15] Object Management Group. Fault tolerant CORBA. OMG Technical Committee Document formal/2001-09-29, September 2001.

[16] Object Management Group. The Naming Service specification. OMG Technical Committee Document formal/2001-02-65, February 2001.

[17] Object Management Group. The Security Service specification. OMG Technical Committee Document formal/2001-03-08, March 2001.

[18] Object Management Group. The Common Object Request Broker: Architecture and specification, 2.6 edition. OMG Technical Committee Document formal/02-01-02, jan 2002.