Lightweight Fault Tolerance in CORBA

Pascal Felber
Bell Labs – Lucent Technologies
600 Mountain Ave, Rm 2B-303
Murray Hill, NJ 07974, USA
pascal@research.bell-labs.com

Abstract

Although fault-tolerant implementations of CORBA have been available for several years, the standard specification of fault-tolerant CORBA (FT-CORBA) has been finalized only recently. This specification defines simple, minimal mechanisms for regular clients to deal with fault-tolerant servers, as well as a wide spectrum of services and APIs for implementing replicated, fault-tolerant servers. While extremely powerful, these advanced server-side mechanisms come with significant complexity both for the FT-CORBA implementor and the application developer. This paper proposes an alternative, lightweight approach to fault tolerance for applications that do not have strong requirements in terms of data consistency. This approach builds on the client-side mechanisms of FT-CORBA and takes advantage of semantic knowledge of the server objects to mediate distributed interactions in an efficient and fault-tolerant manner. Although the approach proposed in this paper is not *applicable to any application, it can be deployed in existing systems to transparently increase their reliability and availability without requiring any re-engineering.*

1. Introduction

Mechanisms for fault tolerance and high availability of distributed applications has been an intensive field of research during the past two decades. This research led to a number of distributed environment for reliable computing [1] based on various computing paradigms such as replication, group communication, or distributed transactions.

More recently, in an effort to provide a unified programming model for distributed computing across heterogeneous platforms and programming languages, the distributed community has been promoting open middleware infrastructures such as the CORBA distributed object architecture [16]. The first generation of reliable CORBA implementations used various approaches to embed fault-tolerant mechanisms in the infrastructure (e.g., [9, 10, 4, 13, 9, 11, 15]), leading to ad-hoc solutions with limitations in terms of interoperability, code portability, compliance to the standard, or performance.

A standard specification of fault-tolerant CORBA (FT-CORBA) has been finalized last year [17]. This specification describes minimal fault-tolerant mechanisms to be included in any CORBA implementation, as well as interfaces for the more advanced facilities provided by a fault-tolerant CORBA implementation. The specification also includes several replication styles and multiple levels of compliance, which informally define the degree of fault tolerance provided by a compliant 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.

In this paper, we argue that some types of applications can be rendered fault-tolerant without paying for the cost and complexity of a full-featured implementation of FT-CORBA. For these applications, we propose a lightweight approach to fault tolerance and high availability that leverages on the basic client-side mechanisms of FT-CORBA, but implements much simpler mechanisms for server-side replica management.

The middleware architecture presented in this paper acts as a replicated gateway and mediates interactions between CORBA clients and servers in a fault-tolerant manner. This system is fully transparent to both clients and servers (FT-CORBA requires servers to explicitly deal with fault tolerance) and can thus be used to increase reliability and availability of legacy applications. The replicated gateway relies on FT-CORBA's core specification for transparent client failover. Reliability and high availability are achieved through replication and load-balancing of requests to redundant instances of application components. The gateway uses semantic knowledge of the distributed objects for smart request processing, in order to preserve application consistency and optimize distributed interactions. Request processing includes inspection, filtering, routing, and non-trivial transformations such as reply aggregation or reference substitution to avoid direct references to "leak out" to clients.

Our approach does *not* intend to replace FT-CORBA and is only applicable to a limited number of applications that do not require the additional flexibility and guarantees provided by a complete fault-tolerant CORBA implementation. We do however believe that such applications can benefit from the simplicity, transparency, and low cost of our approach in situations where FT-CORBA is too onerous.

The remainder of this paper is organized as follows. Section 2 discusses related work and gives an overview of the FT-CORBA specification. Section 3 presents background information on replication protocols and the notion of semantic knowledge. Section 4 describe the architecture of our gateway and Section 5 elaborates on the technical challenges and the various mechanisms used for transparently increasing reliability and availability of distributed objects. Section 6 discusses some deployment and runtime considerations. Finally, Section 7 concludes this paper.

2. Related Work

Early fault-tolerant implementations of CORBA have adopted various approaches for reliability. While some form of reliability can be provided through the use of transactions, high availability is generally achieved through replication mechanisms. CORBA implementations have initially dealt with redundancy through one of three approaches: (1) by integrating group communication mechanisms within the ORB (e.g., [10, 9]), (2) by using transparent interception mechanisms (e.g., [13]), or (3) via a service dedicated to group management (e.g., [4, 15, 11]). However, these systems suffer from some important limitations (e.g., regular clients cannot interact with replicated servers in a fault-tolerant manner) and each of them exposes a different, non-standard APIs to the application, making development of portable applications almost impossible. In addition, most of these systems suffer from a number of problems related to the mismatch between the underlying process group model and distributed objects, such as group proliferation, request duplication, or mixed scheduling [7].

To alleviate these limitations, the Object Management Group (OMG) has initiated a standardization process for fault-tolerant CORBA. Several individuals involved in previous fault-tolerant CORBA implementations (namely [15, 13, 4]) have contributed to this specification, and therefore the FT-CORBA specification [17] partially builds on experiences from previous systems. Due to the fundamental differences between these systems and diverging goals of the companies involved in the standardization process, the resulting specification evolved into a very general, but also complex, replication framework. As an example, the service specification makes it possible to implement fault tolerance through such different mechanisms as group communication and replicated databases. The client-side mechanisms to be included in all CORBA implementations — regardless of whether they implement FT-CORBA or not — have however been kept minimal. They basically specify object references that can contain multiple profiles, each of which designates a replica (multi-profile IORs), and simple rules for iterating through the profiles in case of failure. These mechanisms ensure that non-replicated clients interact with replicated servers in a fault-tolerant manner. We know of two partial implementations of FT-CORBA, Eternal [13] and DOORS [15], as well as a few CORBA implementations that already include support for FT-CORBA's client-side mechanisms.

On the other end of the spectrum, some ORB implementations (e.g., VisiBroker [2]) provide simple fault-tolerant mechanisms through which an object reference can be resolved into one of multiple objects at connection time. These mechanisms are strongly limited and cannot be used for replication since there is no provision for (even minimal) state consistency.

The architecture proposed in this paper uses semantic knowledge of the server objects to efficiently mediate interactions between clients and servers in a fault-tolerant manner. Although the idea of using semantic knowledge to improve performance has been investigated in the context of databases (e.g., [6, 3]), little attention has been paid to its application in the context of distributed systems. In the context of replication, OGS [4] first introduced mechanisms for describing the semantics associated with replicated server, and let the infrastructure (rather than the application) use that information to choose the most efficient algorithm for distributed interactions. For instance, a read operation may not need to be delivered atomically by all servers, thus reducing the latency experienced by the client and the load on the servers. At the algorithmic level, Pedone and Schiper introduced a generalized version of atomic broadcast, called generic broadcast [18], that implements different ordering guarantees depending on the semantics of messages that are broadcast. Informally, generic broadcast behaves either as a reliable broadcast if messages do not conflict, or as an atomic broadcast otherwise. (In this context, two messages do not conflict if they can be processed in any order, i.e., they commute.) This permits to implement active replication with lower latency than using traditional atomic broadcast protocols when the semantics of messages are known. Algorithms for generic broadcast have been proposed in [18, 12].

3. Background

The various mechanisms used to implement replicated objects and maintain the state of individual copies consistent generally have a high cost. For some applications that have weaker consistency requirements, replication may be implemented in a more lightweight manner. For this purpose, some semantic knowledge of the application is necessary. This section discusses replication protocols in the context of fault-tolerant applications. We also introduce the notion of semantic knowledge and point out how it can be used to transparently increase the reliability and availability of distributed applications.

3.1. Replicated Objects

The basic purpose of replication is to implement fault-tolerant and highly available distributed services. The literature distinguishes between two main classes of replication techniques: passive replication and active replication [14]. In passive replication the client only interacts with one replica, called the *primary*: the primary handles the client request and sends back the response. The primary also issues messages to the secondaries (the other replicas) in order to update their state. In active replication the client sends its request to all the replicas, which all handle the request and send back the response to the client. The client waits only for the first reply. Note that active replication requires the servers to be deterministic.

A replicated server should appear as a single logical entity to its clients. For this purpose, the copies of a replicated services must be consistent which other. Ensuring consistency of the replicas is the main difficulty of replication techniques. With active replication, consistency is ensured by having the clients invoke a group communication primitive called *atomic broadcast* (also called *total order broadcast*). Atomic broadcast guarantees that the requests sent by the clients are received by all replicas in the same order. With passive replication, consistency requires a group communication infrastructure that provides a *group membership service* (to select the primary), and a *view synchronous broadcast* (to be used by the primary to update the state of the secondaries) [8].

Additionally, when the number of the replicas can change dynamically, the replication infrastructure must synchronize the state of a new replica with that of other replicas. This is generally achieved by a so-called *state transfer* protocol that transmits the state from an up-to-date replica to the new one. The state transfer is generally driven by the replication infrastructure that calls back to the application for accessing the state. Hence, replicas must typically provide operations for "getting" and "setting" their state.

Managing replicated servers is not trivial and requires complex protocols. There are however situations where advanced mechanisms for strong consistency or dynamic membership are not necessary, in particular when the focus is more on service availability than data replication. A few examples are:

- *Servers that do not maintain state.* For example, a distributed service can offer "processing power" to its clients and process independent, computing-intensive requests on their behalf.
- *Servers that act as a front-end to a stable storage system.* In many deployment scenarios, valuable information is kept in a database (which may be replicated) and distributed objects are used as a presentation layer for accessing this data. Since the data is kept in the database, the distributed objects can always retrieve a consistent state. In addition, when data is updated through the database rather than via the distributed objects, maintaining state consistency does not require complex protocol.
- *Read-only servers.* Sometimes, it is desirable to provide only read-only access to data. For instances, databases, web servers, name servers, are example of applications that often prevent clients from performing updates and leave this task to the administrator.
- *Statically-deployed servers.* When there is no need to dynamically add replicas to a replicated service, distributed protocols become significantly simpler.

In addition, the protocols used to maintain service consistency can often be simplified through the use of semantic knowledge, i.e., when servers exhibit some specific properties that are known by the replication infrastructure.

3.2. Semantic Knowledge

As discussed in [5], semantic knowledge of objects can help implement intelligent behavior in a distributed infrastructure and choose optimal protocols for component interactions. Until recently, little attention has been paid to application of semantic knowledge in the context of distributed systems. Informally, object semantics describes relationships between a request and the object's state (e.g., read-only, deterministic, idempotent) and relationships between multiple requests (e.g., commutative, parallelizable, combinable). This description typically occurs at deployment time and does not involve programming. Semantic knowledge can then be used for several purposes.

At the level of message protocols, the infrastructure can take advantage of semantics to select the least expensive protocol that maintains consistency. For instance, a request to a read-only operation does not need to be processed by all servers; two requests that commute do not need to be totally ordered; etc.

At the level of replica management, semantics are important to guarantee that replicas remain consistent. For instance, a request to a deterministic operation can be safely processed by multiple replicas (active replication); in case of communication failure with a replica, a request to an idempotent operation can be re-issued; etc.

At the level of request processing, semantic knowledge can help the infrastructure optimize interactions with multiple clients. For instances, some requests may be processes in parallel by the same server (e.g., because they modify disjoint parts of the state) or may need to be serialized; some requests may be combined into a single request; some requests may be load-balanced to multiple servers; etc.

Finally, the infrastructure can use semantics to transparently cache information and avoid unnecessary invocations to the servers. For instance, the reply to a read-only request may be cached until invalidated by a conflicting request.

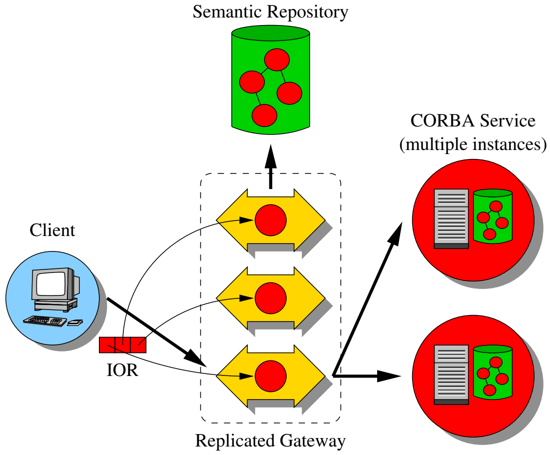

Figure 1. The gateway masquerades as the actual CORBA service and intercepts requests and replies. The client holds a multi-profile IOR that refers to redundant service proxies managed by the gateway.

4. Overall Architecture

This section introduces the architecture that we propose as a lightweight alternative to a full-featured FT-CORBA implementation. Clients interact with multiple instances of a CORBA service indirectly through a gateway. The service instance are *loosely coupled*, in that they do not communicate with each other and none of them is aware of redundancy. The gateway implements the replication logic and uses data stored in a semantic repository to perform non-trivial tasks such as load balancing, caching, and consistency management.

The gateway is itself replicated and clients interact with it using CORBA's standard client-side fault-tolerant mechanisms. The client holds a multi-profile interoperable object reference (IOR) that contains references to individual copies of the gateway. (We shall discuss later how the client obtains such an IOR.) In case of failure of some gateway, the client transparently reconnects to another gateway. FT-CORBA also requires clients to embed enough information in their requests for gateways to detect duplicate requests. This may happen for instance when a client incorrectly suspects a gateway and re-issue its request to another gateway.

Unlike the servers, the multiple instances of the gateway can communicate with each other using atomic broadcast primitives and are thus *strongly coupled*. This permits the gateways to deal with unexpected events like duplicate requests and gateway/server failures. Note that the number and identity of the gateway replicas might change over time; in this situation, the gateway can update outdated client references transparently to the application using so-called "location forward" messages.

The overall architecture of the system is shown in Figure 1 (thin arrows represent references and thick arrows show the actual invocation paths). Clients invoke the service using a multi-profile IOR that actually designates proxy objects in the gateways. The standard behavior of a compliant client is to try invoking the object associated with the reference of each profile in turn, until it gets a reply. If all profiles fail, then the ORB runtime throws an exception to the client application.

Clients do *not* use any form of multicast communication: they only use regular point-to-point IIOP[1] messages. To the client, a gateway appears as the actual service in that it accepts requests to operations of the service's interface, processes them, and returns replies. In fact, the gateway forwards the request to one or several servers for actual processing, and relays the reply to the client. This process is completely transparent to the client, which ignores the identity of the server that actually processed its request. If the gateway fails or is unreachable, the client transparently reconnects to another gateway present in the multi-profile IOR. Note that the degree of replication of the servers is independent of that of the gateway.

5. Fault Tolerance Support

A replication infrastructure must deal with complex issues such as group addressing, consistency management,

[1] IIOP is the instantiation of CORBA's standard communication protocol over TCP/IP.

and failure detection. This section discusses the technical challenges of object replication and describes the various mechanisms implemented in our replicated gateway.

5.1. Dealing with Object References

The architecture described above can be easily deployed with read-only or stateless CORBA services, for which consistency management is trivial. With such services, one might wonder whether the gateway is necessary and why the client could not simply have references to the actual servers in its multi-profile IOR. The answer is to avoid object references to leak out to clients.

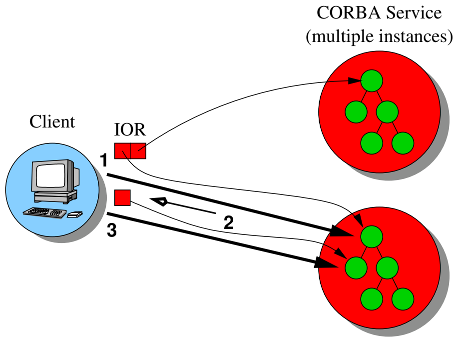

Figure 2. Without gateway, replicated objects do not return multi-profile IORs.

One of the limitations present in early fault-tolerant CORBA implementations is the lack of support for applications that return object references. A classic example of this problem is illustrated by the CORBA naming service. Names are organized in a hierarchy, with each node containing zero or more other nodes and name bindings. If a client holding a multi-profile IOR to the root of the hierarchy traverses down the hierarchy, it will lose the benefits of replication because references to subsequent nodes will *not* be multi-profile IOR (Figure 2). Indeed, the client interacts with a single instance of the naming service, which is not aware of replication.

In the naming service scenario, the proper behavior is to invoke several replicas (even if the request is read-only) and aggregate the returned references to construct a new replicated object. With the gateway approach, this task is easy to achieve because the gateway can intercept replies and create a multi-profile IOR on-the-fly. However, the gateway cannot simply returns the aggregated reference to the client, because the same problem would occur again recursively. Therefore, the gateway must take the following steps (Figure 3): (1) relay the client requests to the replicas, (2) aggregate the returned references to construct a new replicated

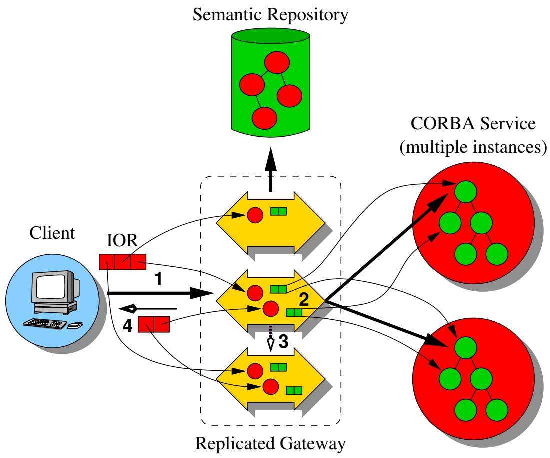

Figure 3. The gateway creates a proxy on-the-fly when a replicated object returns a reference to another replicated objet, and constructs a multi-profile IOR for the client.

object, (3) create a proxy for this replicated object on some[2] or all of the gateway replicas, and (4) return a multi-profile IOR with references to these proxies.

It is possible in some situations to let the client interact directly with the replicated server. This is the case when the server is read-only and the gateway can determine that it never returns references (this can be established by analyzing the server's interface). This approach leads to increased performance because requests do not have to traverse the gateway. An intermediate approach adopted in our prototype implementation consists in returning multi-profile IORs with the first profile (tagged as the primary) referencing an actual server and the next profiles referencing the gateway (Figure 4). That way, if the server fail, the client reconnects to the gateway that can act accordingly (e.g., restart the server, direct the request to a non-failed replica, update the client's reference, etc.).

Reference aggregation is not always desirable. For instance, a replicated server might return a reference to a singleton object and because of CORBA's weak identity model, the gateway might not be able to determine that the replicas have returned references to the same object. There is no universal way to distinguish between references that should and should not be aggregated. Therefore, it is necessary in some situations to indicate when the gateway needs to perform reference aggregation. This information is part of the semantics describing the server's interface.

[2]In Figure 3, only two out of three gateway replicas have been used for the second node.

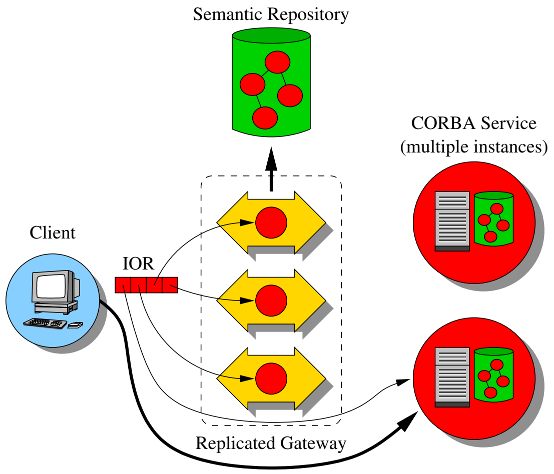

Figure 4. In some situations, the gateway can leak out direct references to the actual CORBA service as part of a multi-profile IOR. To deal with service failures, the IOR also contains references to service proxies managed by the Gateway.

5.2. Consistency Management

One of the most complex issue of a replication infrastructure is to maintain the replicas consistent with each other. In our context, consistency informally means that the state of the replicas must remain identical. Much of the complexity of the FT-CORBA specification comes from consistency management, with dedicated APIs for tasks like state transfers and state updates.

Maintaining server consistency transparently to the server is a challenging task, and is often not possible. For instance, passive replication requires the primary and backups to explicitly deal with state updates, and dynamic group management requires support for copying an object's state. Also, as discussed in [19], transparency typically requires generic protocols that behave in a conservative way and cannot perform optimizations, thus resulting in poor performances.

Because it performs transparent replication, the gateway architecture discussed in this paper also suffers from the these limitations (although some of them are alleviated by the use of semantic knowledge). There are however situations where transparent replication can be implemented without risks of breaking consistency. For instance, consistency management is trivial if the servers are read-only because there is no need for state transfer and state update mechanisms.

When the servers can be modified but behave in a deterministic manner, the gateway forwards update requests to all replicas. Read requests can be sent to only one up-to-date replica. Since the gateway is replicated and multiple clients can concurrently send requests through different copies of the gateway, update request must be serialized by the replicated gateway, and read requests must be totally ordered with respect to updates. To achieve this, the gateway replicas communicate with each other to agree upon request ordering.

If a server fails or becomes unreachable, the gateway stops sending requests to that server. If it becomes available at a later point in time, its state may be different from that of the other replicas. Therefore, the gateway should never invoke a server that has been assumed to be failed (unless it is read-only) because of the absence of state transfer and state update mechanisms.

Consistency may be broken in a more subtle way. If a gateway fails while invoking an update operation on several servers, the client will contact another gateway. The latter cannot know which servers have been successfully invoked by the first gateway. If the operation invoked on the server is idempotent, then the gateway can invoke it multiple times without breaking consistency. Otherwise, this problem is not solvable without adding additional logic to the server application.

In summary, consistency management cannot performed in a generic way suitable for all applications and transparency further limits its use to situations where state transfer and state update are not necessary. Therefore, the gateway approach is inadequate for some types of applications and a full-featured FT-CORBA implementation should be used in these cases.

5.3. Combining Fault Tolerance and Load Balancing

The use of semantic knowledge allows us to solve most of the limitations discussed in [19], and let the infrastructure perform many kinds of optimizations while maintaining consistency. One of the most obvious example is the combination of replication and load-balancing of client requests. Other uses of semantic knowledge are discussed in [5].

Without semantic knowledge, a transparent replication infrastructure cannot determine if a request modifies the state of a server and must conservatively assume that it does. On the other hand, when the infrastructure can distinguish between read-only and update requests, it can perform several obvious optimizations. Update requests must be sent to all replicas and serialized. Read-only requests can be arbitrary interleaved with each other as long as they are totally ordered with respect to updates. In addition, they do not need to be processed by every replica. Therefore, read-only requests can be load-balanced to any of the replicas, thus increasing experienced throughput and responsiveness. Note that the failure of a replica in this situation may increase the latency of the request because the infrastructure has to reissue the request to another replica; to avoid this situation, the infrastructure can dispatch read-only request to a small subset of the replicas (typically two) to decrease the odds of delays upon failure while still experiencing the benefits of load-balancing. Note that, in any case, failover is completely transparent to the client.

A refinement of this approach can be performed when we can further distinguish between read-only requests to static data and read-only requests that access the state of an object. This is the case for instance of a server that implements online shopping. The contents of the shopping cart represent the object's state; adding an item to the cart corresponds to an update request, listing the contents of the cart to a read-only request to state data, and consulting the catalog to a read-only request to static data. In this situation, assuming that there exists a large number of copies of the shopping server, the infrastructure can use a small subset of replicas to maintain the user's state (i.e., the shopping cart). Update requests are delivered to all copies of that subset, read-only requests to state data are load-balanced to any copies of the subset, and read-only requests to static data can further be load-balanced to any of the replicas, whether or not it maintains a copy of the user's state. Such optimizations are only possible with semantic knowledge, which lets the infrastructure take "intelligent" decisions without involving the servers directly.

6. Deployment and Runtime Issues

The gateway architecture presented in this paper transparently manages replica consistency and client accesses to multiple instances of a CORBA service. This approach does however add overhead to the system (as compared to a non-replicated service) and cannot be used for all CORBA services. This section discusses how a replicated service can be deployed and advertised, how the gateway overhead can be minimized, and when an application should use a full-featured FT-CORBA implementation instead of the lightweight gateway approach.

6.1. Obtaining Initial References

We have previously assumed that the client interacts with a replicated gateway using multi-profile IORs. We also showed how the gateway can transparently create multi-profile IORs from references returned by the servers to the clients. We now discuss how the clients obtain the initial references to a replicated service maintained by the gateway.

Clients obtain initial references to services through various mechanisms: by reading the IOR from the command line or from a file, through the naming service or the trading service, etc. To ensure that clients obtain correct multi-profile IORs to replicated services, one has to make sure that references to the individual replicas are not advertised in the system. The approach that we advocate is to replicate the naming service using the gateway and export a multi-profile IOR to the root naming context. This context will subsequently be used by clients to find references stored in the naming service. Since the gateway performs reference aggregation, traversal of the naming service is performed in a fault-tolerant manner.

The deployer can then create a replicated service by starting multiple instance of the server and register a multi-profile IOR into the naming service. The profiles of this IOR can directly reference the server's replicas (instead of proxies in the gateways), because the gateway will create proxies on-the-fly if necessary. Note that the deployer should generally not store references to individual replicas of the server in different instances of the naming service, even though the gateway can perform reference aggregation. Indeed, the replication degree of the server would directly depend on that of the naming service and, in some failure scenarios, different clients may update different subsets of the server's replicas in an inconsistent manner. Therefore, the deployer should always store multi-profile IORs when the servers are not read-only.

The major advantages of using the naming service to distribute references to the clients are that (1) the naming service is a regular CORBA service and can thus be rendered fault-tolerant and highly available, and (2) the gateway can create proxies for the replicated servers only when required, therefore simplifying service deployment and avoiding unnecessary resources consumption. Indeed, the naming service is a strategic component for controlling the creation and distribution of references in the system.

Note that the problem of multi-profile IOR creation and distribution must also be addressed by full-featured implementations of FT-CORBA; the gateway approach has the advantage of being able to create "replicated references" from individual, non-replicated servers on demand.

6.2. Performance Considerations

Like any CORBA bridge, the gateway uses the CORBA's dynamic request facilities [16] to process requests targeted at objects not known at compile time: the Dynamic Skeleton Interface (DSI) to dynamically create proxies for a given objet; the Interface Repository (IR) to discover server interfaces at runtime; and the Dynamic Invocation Interface (DII) to create requests and invoke operations on servers.

The use of these dynamic facilities is necessary to inspect CORBA requests, because messages are not fully self-describing. These facilities have a non-negligible runtime overhead, but there are ways to speed-up dynamic request processing significantly. When the gateway initially creates a proxy for a replicated server, it finds out the associated interface and performs a static analysis of that interface (if the same interface has not already been analyzed previously). During analysis, the gateway check which operations may leak out references, and which one may not. This information is used later to decide whether the gateway needs to inspect the data returned by the servers (which is a costly operation) and possibly perform reference aggregation, or if it simply needs to return the reply back to the client without extra processing. In addition, the gateway caches important information about the server's interface and pre-computes the data structures (lists of name-value pairs) used to receive and create dynamic requests. The gateway effectively accesses the interface repository only once, the first time a proxy is created for an unknown interface.

Another source of performance overhead comes from proxy life cycle management: it is generally no possible to determine when a proxy is no longer be needed and its resources can be recycled. As the number of proxies grows, so does memory consumption and performance may degrade. This problem can be solved by periodically storing on disk and reclaiming the resources of proxies that have not been accessed for some time. Proxies that have been destroyed can be re-constructed on demand using standard CORBA mechanisms, namely custom servant locators.

Although we have experienced a significant speedup when implementing the optimizations described above, the use of a gateway adds a non-negligible overhead to the system. Furthermore, because of its dynamic nature and its genericity, the gateway cannot be implemented without CORBA's dynamic request processing facilities. We do however believe that the advantages offered by the gateway, namely fault tolerance and high availability, together with semantic-based smart request processing (e.g., combination of replication and load-balancing) can prove to be valuable and pay off for the gateway's overhead.

6.3. Limitations

As previously mentioned, the gateway architecture that we propose suffers from a few shortcoming, which limit its use to specific application domains. In particular, due to its transparency requirements, this approach cannot be applied to passively-replicated or non-deterministic servers (unless they manage their consistency themselves, as in the case of a replicated database).

A more subtle limitation is the risk of inconsistency when a gateway instance fails while issuing an update request to a non-idempotent operation of a replicated service. In this situation, it may be impossible to determine which replicas have already processed the request, and which ones have not.[3] A similar problem may arise if communication links between the gateway and the service fail in such a way that the gateway cannot determine if a service replica has processed the request. The only option to preserve consistency in this situation is to assume that server to be failed.

These limitations are direct consequences of the server transparency feature of the architecture, where servers do not provide support for state management and duplicate request suppression. As a matter of fact, the gateway architecture is only adapted to specific kinds of applications (as described in Section 3.1) and is not intended as a universal replication infrastructure.

7. Conclusion

In this paper, we have discussed the provision of fault tolerance in the CORBA middleware infrastructure through the recent fault-tolerant CORBA (FT-CORBA) standard. We have argued that, while some advanced mechanisms of the FT-CORBA specification are necessary to ensure strong consistency of replicated objects, they are not necessary for all types of applications and they expose significant complexity both to the FT-CORBA implementor and the application developer.

We have presented a lightweight approach to fault tolerance in CORBA that leverages on the minimal client-side mechanisms defined by the FT-CORBA specification, but does not use any of its more advanced facilities. This approach relies an a replicated gateway that mediates interactions between clients and servers in a fault-tolerant manner. The gateway is transparent to both clients and servers, and high availability is achieved through the use of multi-profile IORs and transparent client failover.

The gateway additionally uses semantic knowledge of the server objects to both maintain consistency on behalf of the application and optimize client-server interactions. Such optimization include selective data caching and load balancing. The gateway includes advanced mechanisms to prevent references to leak out to clients and create dynamic proxies on the fly. These mechanisms permit complex applications to be replicated transparently, without re-engineering.

The approach advocated in this paper is not universal and is limited to specific types of applications. It is not meant to replace FT-CORBA, but rather to offer a lightweight alternative in situations where using a full-featured FT-CORBA implementation would be too onerous. We have been successful in deploying fault-tolerant instances of the CORBA

[3] In fact, at the price of additional communication overhead it is possible to guarantee that at most one of the replicas becomes inconsistent and prevent subsequent invocations to that replica.

naming service and interface repository using a prototype gateway implementation. For this purpose, we have used the services provided with our CORBA implementation without modifying them in any way. Quantitative analysis of our gateway in real-world deployment scenarios is part of our future research directions.

References

[1] K. Birman. *Building Secure and Reliable Network Applications*. Manning Publications Co., 1996.

[2] Borland. *VisiBroker Programmer's Guide*. Borland, 2000.

[3] A. Farrag and M. Özsu. Using semantic knowledge of transactions to increase concurrency. *ACM Transactions on Database Systems*, 14(4):503–525, Dec. 1989.

[4] P. Felber, R. Guerraoui, and A. Schiper. The implementation of a CORBA object group service. *Theory and Practice of Object Systems*, 4(2):93–105, 1998.

[5] P. Felber, B. Jai, R. Rastogi, and M. Smith. Using semantic knowledge of distributed objects to increase reliability and availability. In *Proceedings of the 6th International Workshop on Object-oriented Real-time Dependable Systems (WORDS'01)*, Roma, Italy, Jan. 2001.

[6] H. Garcia-Molina. Using semantic knowledge for transaction processing in distributed database. *ACM Transactions on Database Systems*, 8(2):186–213, Mar. 1983.

[7] R. Guerraoui, P. Felber, B. Garbinato, and K. Mazouni. System support for object groups. In *Proceedings of the ACM Conference on Object Oriented Programming Systems, Languages and Applications (OOPSLA'98)*, Oct. 1998.

[8] R. Guerraoui and A. Schiper. Software-Based Replication for Fault Tolerance. *IEEE Computer*, 30(4):68–74, April 1997.

[9] IONA and Isis. *An Introduction to Orbix+Isis*. IONA Technologies Ltd. and Isis Distributed Systems, Inc., 1994.

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

[11] 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.

[12] H. F. M.K. Aguilera, C. Delporte-Gallet and S. Toueg. Thrifty generic broadcast. In *Proceedings of the 14th International Symposium on Distributed Computing (DISC'00)*, Oct. 2000.

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

[14] S. Mullender, editor. *Distributed Systems*, chapter 7 and 8. Addison-Wesley, 2nd edition, 1993.

[15] B. Natarajan, A. Gokhale, S. Yajnik, and D. 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.

[16] OMG. *The Common Object Request Broker: Architecture and Specification*. OMG, October 2000.

[17] OMG. *Fault Tolerant CORBA Specification, V1.0*. OMG, April 2000. OMG document: ptc/2000-04-04.

[18] F. Pedone and A. Schiper. Generic broadcast. In *Proceedings of the 13th International Symposium on Distributed Computing (DISC'99)*, Sept. 1999.

[19] W. Vogels, R. V. Renesse, and K. Birman. Six misconceptions about reliable distributed computing. In *Proceedings of the 8th ACM SIGOPS European Workshop*, Sept. 1998.