Architectures for Fault-Tolerant Middleware Services

Erez Hadad
Architectures for Fault-Tolerant Middleware Services

Research Thesis

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy

Erez Hadad

Submitted to the Senate of
the Technion — Israel Institute of Technology
Nisan 5766 Haifa April 2006
The Research Thesis Was Done Under The Supervision of Dr. Roy Friedman in the Computer Science Department.

I would like to thank Roy Friedman for his inspiring guidance.

I thank Prof. Hagit Attiya and Dr. Idit Keidar for tutoring me in the theoretical aspects of distributed systems and failure models, as well as for referring me to further sources of knowledge. I also thank Assoc. Prof. Danny Raz for valuable lessons in issues of probability.

I thank Dan Kenigsberg for assisting me with \LaTeX/Hebrew.

A special thanks is sent to Yardena Kolet for efficient administrative support.

Almost last but foremost, I have only but my deepest love and gratitude to my wonderful wife Orit and my sweet children Ohad and Ella who helped me maintain my joy of life even in the most trying moments. Also, to my parents Meir and Michal Hadad, to whom I owe my life and education from the very beginning. Your love and support carried me here.

The generous financial support of the Idi and Robert Netter Memorial Fellowship, the Eshkol Fund of the Israeli Ministry of Science, the VATAT fund of the Israeli Board of High Education and the Technion is gratefully acknowledged.
## Contents

Abstract .......................................................... 1

1 Introduction ..................................................... 3
  1.1 Background .................................................. 3
  1.2 Research Overview .......................................... 5
  1.3 Thesis Layout ............................................... 7
  1.4 CORBA ....................................................... 8
  1.5 Middleware Fault-Tolerance ................................. 11
     1.5.1 Replication Techniques ............................... 12
  1.6 Group Communication ...................................... 14

2 Related Work ................................................... 17
  2.1 Adding Replication to CORBA ............................. 17
     2.1.1 The Integration Approach ............................ 17
     2.1.2 The Interception Approach ........................... 19
     2.1.3 The Service Approach ................................ 22
     2.1.4 FTS: The Object-Adaptor Approach .................. 25
  2.2 The Fault-Tolerant CORBA (FT-CORBA) Specification ..... 27
  2.3 Request Batching/Packing .................................. 32
  2.4 FTS Request Cache and Selective Acknowledgements .... 34
3 Introducing FTS: A Fault-Tolerance Service 38

3.1 Design Guidelines .............................................. 38
    3.1.1 Fault-Tolerance and Survivability ......................... 38
    3.1.2 Portability and Interoperability ......................... 39
    3.1.3 Performance and Scalability ............................. 40
    3.1.4 Simplicity ............................................. 42

3.2 FTS: Top-Level Description ................................. 42
    3.2.1 Group Object Adaptor ................................ 43
    3.2.2 Interfacing With GC .................................. 47
    3.2.3 FTS Interceptors .................................... 49
    3.2.4 Object Replica ...................................... 49

4 Operational View of FTS 51

4.1 Replication Model .......................................... 51
4.2 Associating Objects and Replicas ....................... 52
4.3 Fault-Detection ........................................... 53
4.4 Replica Consistency ..................................... 54
    4.4.1 Protocol Description ................................ 55
    4.4.2 Properties of the Protocol ........................... 55
4.5 Cascading Invocations .................................. 61
    4.5.1 Inter-Context Communication ......................... 61
    4.5.2 Global Consistency .................................. 62
4.6 Dynamic Service Configuration ......................... 63
    4.6.1 LMR Protocol Properties .............................. 68
4.7 Age and State-Transfer .................................. 69
4.8 Request Identification and Caching .................. 72
4.9 Dynamic Object Management ............................... 73
4.10 Discussion ................................................. 74
    4.10.1 Replication Granularity .............................. 74
4.10.2 Selecting Consistency Modes ........................................ 76
4.10.3 Rationale of Handling Dynamic Configuration ................. 78
4.10.4 Replicated Object Reference Semantics ........................... 79

5  FTS Performance Evaluation .............................................. 82
  5.1 Test Environment ..................................................... 82
  5.2 Test Configuration .................................................. 83
  5.3 Test Procedure ..................................................... 83
     5.3.1 Measurements ................................................ 84
  5.4 Summary of Results ................................................ 84
  5.5 Analysis ............................................................. 88
     5.5.1 Invocation Latency ........................................... 88
     5.5.2 Cluster Throughput .......................................... 92
  5.6 Conclusions ........................................................ 96

6  Adaptive Batching ....................................................... 98
  6.1 Characterizing The Problem ......................................... 98
     6.1.1 The Need For Adaptivity ................................... 99
     6.1.2 Model of Operation ......................................... 100
  6.2 Adaptive Batching .................................................. 100
  6.3 Timed-Adaptive Batching ........................................... 102
  6.4 Performance Evaluation ............................................ 102
  6.5 Discussion .......................................................... 105
  6.6 Conclusions ........................................................ 108

7  Reducing FTS Server Memory Consumption .......................... 109
  7.1 Motivation .......................................................... 109
  7.2 Implementing SACK in FTS ........................................ 110
     7.2.1 Description .................................................. 110
     7.2.2 Rationale ...................................................... 111
8 Summary and Future Directions 118

8.1 Summary of Results ................................. 118
8.2 Acquired Lessons ................................. 120
  8.2.1 Middleware Support for Replication .......... 120
  8.2.2 Comparison With FT-CORBA .......... 125
8.3 Future Directions ................................. 131
  8.3.1 Porting to Different Middlewares .......... 131
  8.3.2 Extending FTS ................................. 132
  8.3.3 Proposed Design Modifications .......... 138

Abstract in Hebrew
List of Figures

1.1 CORBA IOR Structure ............................................. 8
1.2 Client-Server Interaction in CORBA .............................. 9
2.1 The Integration Approach ........................................... 19
2.2 The Interception Approach ......................................... 23
2.3 The Service Approach ............................................... 25
2.4 FTS: The Object-Adaptor Approach ................................. 26
2.5 The FT-CORBA Architecture ........................................ 28
3.1 Sharing replication resources between objects ..................... 41
3.2 Top-Level View of FTS .............................................. 43
3.3 GOA Structure ..................................................... 45
4.1 Request Processing Protocol - Data Definitions .................. 56
4.2 Request Processing Protocol - Client Part ........................ 56
4.3 Request Processing Protocol - Server Part ........................ 57
4.4 LMR Sub-service .................................................. 64
4.5 LMR Server Operation .............................................. 65
4.6 Service Acquisition at Component Coordinator ................... 67
4.7 Client Redirection .................................................. 68
5.1 Invocation Latency Test Results ..................................... 85
5.2 Write Test: Throughput vs. Cluster Size .......................... 85
5.3 Write Test: Throughput vs. Client Load ............................ 86
5.4 Read Test: Throughput vs. Cluster Size ................. 86
5.5 Read Test: Throughput vs. Client Load ................. 87
5.6 Mixed Test: Throughput vs. Cluster Size ................. 87
5.7 Mixed Test: Throughput vs. Client Load ................. 88
5.8 Comparison of FTS write throughput with non-replicated server ........ 89
5.9 Comparison of FTS read throughput with non-replicated server ........ 89
5.10 Comparison of FTS mixed throughput with non-replicated server ........ 90
5.11 Wait-time utilization problem in FTS servers ............. 95
5.12 Write test: 2000 clients vs. 2,4,6,8 servers with only 2 active servers ........ 97

6.1 Adaptive Batching ........................................... 101
6.2 Performance Evaluation: Throughput vs. Client Load .......... 103
6.3 Performance Evaluation: Cast Latency vs. Client Load .......... 103
6.4 Performance Evaluation: Batch Length vs. Client Load .......... 104
6.5 Performance Evaluation: (Batch Length / Cast Letency) vs. Client Load 107

7.1 Performance Evaluation: Cache Size vs. Client Load .......... 113
7.2 Performance Evaluation: Server Memory vs. Client Load .......... 114
7.3 Performance Evaluation: Cluster Throughput vs. Client Load .......... 114

8.1 Decoupling invocation and reply using a proxy in FTS ............. 139
List of Tables

4.1 Summary of Consistency Modes .................................................. 59
5.1 Write invocation latency profiling (msec) ................................. 91
5.2 Write latency profiling for 1500 clients (msec) ......................... 93
7.1 Performance Evaluation: SACK Overhead vs. Client Load ............ 115
Abstract

Distributed computing is the key technique for constructing enterprise-scale software systems, both now and in the foreseeable future, for reasons such as flexibility, component re-usability and scalability. Middleware platforms such as CORBA, J2EE and .NET facilitate the construction of distributed systems by standardizing and abstracting various aspects of distribution including component interaction, management and isolation.

Partial failures are a fundamental problem in distributed systems, arising from the actual distribution of system functions among multiple components that can fail independently of one another. In the context of client-server interaction, such failures might disrupt the service provided to clients, which is undesirable, especially for mission-critical applications. Consequently, much research was devoted to increasing the availability of distributed services by making them fault-tolerant. Specifically, an extensive effort was put into adding generic fault-tolerance capabilities to the middleware, which is our initial research problem.

In this thesis we present and explore FTS, a CORBA Fault-Tolerance Service that operates through active replication of objects. FTS embodies a novel approach at solving our research problem, called the object-adapter approach. In essence, this new approach melds together qualities of preceding techniques, resulting in a unique combination of portability, interoperability, performance and simplicity. Furthermore, through FTS we provide constructive criticism of the Fault-Tolerant CORBA (FT-CORBA) standard, which aims to be a comprehensive solution to the above problem. Last, we study the performance of FTS, and deduce general limitations of CORBA’s ability to support sophisticated high-level services.

As additional contributions, we study a couple of aspects of FTS operation that have general relevance to middleware services and/or state machine replication. First, we focus on applying batching (or packing) of multiple requests into a single ABCAST message as a technique to boost its throughput, and consequently the throughput of an active replication service that uses it. We observe that maintaining good throughput using ABCAST protocols under varying client request rates involves adapting the
batching threshold with the request rate. Consequently, we provide two variants of an adaptive batching mechanism and show their advantage compared to classic fixed-threshold batching mechanisms.

Last, we present a scheme for reducing the memory footprint of replication services such as FTS and FT-CORBA compliant systems. One of the largest memory-consuming components is the reply cache, which is used for maintaining at-most once semantics of request execution. We show how to apply Selective Acknowledgments (SACKs) to speed up cache purging beyond its time-based expiration mechanism, in a way that matches FTS’s client-to-single-server interaction. This technique can also be applied to many types of passive replication systems as well. We further test this technique and show its effectiveness.
Chapter 1

Introduction

1.1 Background

A distributed software system is a collection of software components that can be created/destroyed, operate and fail independently of one another. The components interact with each other either by passing messages through a network or by reading from and writing to shared memory. The resulting programming model is much more complex than that of a single sequential program since the system designer has to deal with issues of inter-component coordination and timing as well as with possible partial failures of the system’s operation prior to end of computation. Such failures result either from individual component faults, e.g., due to process/computer crash, or from faults in the communication medium, e.g., switch failure or network routing errors.

On the other hand, distributed systems present a number of significant advantages over single sequential programs, such as the ability to harness more computing resources than what a single machine can produce, by deploying the system’s components over multiple networked machines. This way, one can construct a powerful computing platform using a set of cheap PCs instead of a single and much more expensive mainframe. Such a platform can also be scaled out - gradually expanded to suit growing computational needs by adding hardware, as opposed to upgrading a mainframe by replacing it as a whole. In addition, the loose component coupling of distributed systems facilitates development of isolated generic components that can be re-used in various combinations with tailored business logic to form different applications. For example, different instances of the same database server can be used for inventory management as well as accounting. Because of these advantages and others, distributed systems have emerged as the foundation of today’s enterprise computing.
In the course of distributed systems evolution, one of the most significant advancements was the introduction of middlewares such as RPC, CORBA, D/COM, J2EE and .net. A middleware consists primarily of a broker, which is a piece of software that is attached to each component and resides as a middle tier between the component’s application logic and the communication stack, hence the name "middleware".

The basic contribution of a middleware is that it provides a unified communication mechanism with clearly-defined properties for component interaction. Furthermore, a middleware simplifies the common client-server communication paradigm, where a client component sends a request to a server component who it turn responds with a reply\(^1\). Using a middleware, such an interaction is abstracted as a function or method invocation from one component to another, regardless of their relative locations. Thus, the physical distribution of the system becomes nearly-transparent to the application logic. Middlewares also allow a formal definition of the interface to a server component to be created separately from the server’s actual implementation, using a high-level language such as CORBA’s IDL. The server is committed to support the interface whereas each client component is bound to access the server only through the provided interface. This further facilitates component isolation. In a larger scale, distributed applications become a hierarchy of services built on top of each other in an increasing abstraction level. Contemporary object-oriented middlewares include, in addition to the broker itself, a set of standard basic services for object creation/deletion, object location, transactions, synchronization, etc.

Failures remain a constant problem to be dealt with in distributed services, especially since failure probability grows with the number of components in increasingly complex hierarchies of implementations. Furthermore, service disruptions cripple the operations of all the clients that depend on the service. High service availability is crucial for mission-critical applications, such as airplane control, nuclear power station management, hospital patient monitoring, etc. Thus, a great deal of research was dedicated to rendering services highly-available by making them fault-tolerant, that is, capable of withstanding failures of certain types and scopes.

The common techniques for overcoming service failures are based on redundancy, either of space or of time. Spatial redundancy is having more resources than required by the application, e.g., replication of components and/or communication routes. Temporal redundancy is doing additional work to avoid or to overcome failures, e.g., retrying requests after receiving no response or recovering a component after failure. The active replication technique primarily utilizes spatial redundancy to make a service fault-tolerant by routing client requests to multiple server component replicas. On the other

---

\(^1\) The titles of “client” and “server” are only roles that serve one session of communication and do not impose any particular functions.
hand, passive replication capitalizes on temporal redundancy by having clients wait until the server that handles their requests is recovered from failure or replaced by a backup.

The growing use of middlewares resulted in a natural need of generic methods for adding fault-tolerance capabilities to the middleware. This is the research question from which our work began. Prior work can be generally categorized into three approaches towards solving this problem. The integration approach modifies the broker by adding replication support, e.g., atomic broadcast for supporting active replication. The interception approach places an additional software element outside the server component to intercept client-server communication and redirect it to an external replication mechanism. The service approach constructs a generic middleware replication service to be used as a building block for a fault-tolerant application. Each approach has its own typical advantages and disadvantages. The integration approach is transparent to the application but generates proprietary brokers that have limited interoperability with standard brokers. The interception approach is transparent both to the application and the broker but the interception code itself is highly platform-dependent since it resides outside the broker. The service approach yields the most portable solution but is not transparent to the application, thereby requiring adaptation for existing applications. More on this can be found in Chapter 2.

In the beginning of 2000, the core CORBA standard was extended to include a specification called FT-CORBA [90] for constructing fault-tolerant CORBA applications. The FT-CORBA standard is designed as a comprehensive solution that unifies most of the existing techniques for replicating distributed objects, both clients and servers, while allowing various levels of application transparency and control. However, there are several fundamental requirements of this standard whose benefits we examined in our work.

### 1.2 Research Overview

Our work began, as mentioned, with the question of what is the best technique to add fault-tolerance to an existing middleware. Our test middleware is CORBA[87], for being mature, platform-independent, and specifically for being the test middleware of choice in many related works in our field. Our first result is FTS [39, 45, 46], a CORBA Fault-Tolerance Service, which works through active replication of CORBA objects. The architecture behind FTS is simple, symmetric, highly portable yet nearly completely transparent to applications. Also, FTS provides good performance and is capable of withstanding a wide range of benign faults. FTS is written in Java, deployed currently on Linux and uses the Ensemble [1, 46] group communication (GC) toolkit to provide
the atomic broadcast (ABCAST) service needed for disseminating client requests to all
server replicas.

Conceptually, FTS operates by having each client communicate with a single server
replica. The server replica is responsible for propagating client requests to the rest
of the replicas in its cluster using the ABCAST service of the underlying GC toolkit,
and eventually responding to the client after executing the request. Server failures and
disconnections are handled on the client side using a redirection sub-service to let clients
re-establish contact with a non-failed server. On the server side, failures are handled
using the membership service of the GC toolkit. The architecture of FTS embeds all this
functionality in near-transparent fashion into the standard CORBA operation model.

FTS embodies a unique approach to our research question, called the object-adaptor
approach, which aims to combine the qualities of the preceding approaches while trying
to avoid the associated pitfalls. In principle, the object-adaptor approach is an extension
of the interception approach, in the sense that it intercepts not only the client-server
communication, but also the interaction between the server objects and the broker (the
CORBA ORB). Still, FTS does it without losing portability, since it utilizes the pre-
viously unused CORBA’s standard Portable Object Adaptor (POA) as an extended
interceptor. Next, the core logic of FTS operates above the ORB level, similar to a
standard service. However, FTS remains nearly transparent to the service application
by disguising itself as a standard object adaptor, which is the CORBA part that medi-
ates between the ORB and the objects. Last, for reasons of performance and simplicity,
FTS integrates its own code with an underlying GC toolkit that provides the required
ABCAST service. However, FTS operates outside the ORB without modifying it or its
interaction with clients, in the sense that each client still communicates with a single
FTS server, contrary to classic integration approach examples.

Additionally, FTS features a number of interesting properties, such as context-based
replication, that is, replicating whole groups of objects sharing a common context, as
a means to reduce per-object replication overhead. Also, FTS is capable of handling
network partitions using a primary-component model [74]. As an added value, we use
FTS to constructively criticize FT-CORBA, by examining its various limitations and
suggesting alternatives demonstrated through FTS.

Next, we present a comprehensive performance test of FTS. We focused on typical
FTS operation, that is, serving client requests, and on measuring the resulting over-
head. This test produces a number of conclusions. First, we justify the need in typical
active replication systems to select a possibly weak consistency model that matches the
requirements of the application, rather that defaulting to the linearizability requirement
of FT-CORBA. We implement a weaker sequential consistency semantics using the local
read [8] algorithm, which atomic-broadcasts only requests tagged as updates, and the
standard linearizable consistency using atomic broadcast of all requests. Next, we identify a general scalability limitation of services operating above the ORB and within its threading model (including FTS). The ORB threading model assigns a thread to handle a request until the request is done processing. As a result, the thread cannot be re-used even if the request is in a waiting status, thereby decreasing processor utilization when using thread pools. Last, we show that increasing server cluster size does not necessarily improve performance, as a subsequent result of the previous limitation. Thus, cluster size should be matched with the client load to remain cost-effective.

Another aspect of FTS performance we study is batching multiple requests into a single ABCAST message as a simple means to increase the ABCAST throughput, and consequently, that of FTS. The concept of increasing ABCAST throughput by batching (or packing) multiple messages into a single protocol packet has already been proven by previous works both empirically [41] and analytically [17, 79]. However, in FTS, we approach this subject from a different aspect. First, unlike [41], we assume no knowledge of the underlying ABCAST protocol for reasons of portability. Second, and more important, we observe the need to adapt the batching threshold with the request rate in order to maintain high throughput under varying rate conditions. Consequently, we provide two batching mechanisms that provide inherent adaptivity and show their advantage compared to other fixed mechanisms.

Last, we present a simple method for reducing the memory footprint of FTS and of FT-CORBA-compliant servers. One of the largest memory-consuming components of each FTS server is the request/reply cache, whose purpose is to guarantee at-most-once request execution semantics by returning a stored reply rather than re-executing a request that was already issued before (a retry). Such a cache is purged using time-based expiration and thus grows with both expiration time and the server throughput. We suggest augmenting client requests with selective acknowledgments (SACKs) of requests a client received replies for. Additionally, we propose a method of SACK processing at the server side that matches FTS architecture. Such an implementation is also suitable to FT-CORBA-compliant passive-replication constructs. We show the positive impact of our SACK implementation, as well as a trade-off of using it.

1.3 Thesis Layout

The rest of this thesis is organized as following. This chapter is appended with three more sections that provide further relevant technical background. Section 1.4 in this Chapter presents a more detailed description of CORBA, which is required for understanding how FTS works. On a similar note, Section 1.5 provides a general overview of middleware fault-tolerance and replication techniques, followed by Section 1.6 that
Figure 1.1: CORBA IOR Structure

CORBA IOR מבנה IOR

presents an overview of Group Communication (GC). Next, Chapter 2 presents a survey of works and techniques related to all the subjects covered in the thesis. Chapters 3 and 4 present a more detailed description of FTS from both a structural and an operational aspects. Next, Chapter 5 presents the performance test of FTS, including the test environment and results. The adaptive batching study follows in Chapter 6. Chapter 7 covers the proposed SACK implementation and its impact on reducing FTS server memory. Last, Chapter 8 summarizes the thesis and suggests directions for future work.

1.4 CORBA

The Common Object Request Broker Architecture (CORBA) [86, 87] is a middleware standard introduced by the Object Management Group (OMG) [44]. Simply stated, CORBA allows client applications to communicate with server objects regardless of the objects’ location, operating environment or implementation. The communication is in a client-server model, where the client holds a reference to a remote server object in order to invoke that object’s services. The reference is called an Interoperable Object Reference (IOR) and consists of a group of tagged profiles. Each profile provides data for reaching the target object through the communication protocols supported by that profile. Protocol information inside a profile is stored in tagged components, which are name-value pairs. The structure of an IOR is displayed in Figure 1.1.

In CORBA, the Object Request Broker (ORB) is the middleware agent that mediates

---

2For a more comprehensive technical report of FTS see [45].
the communication between a client and a server object. In order for a client to be able to access a remote object, it uses a stub - an object wrapper containing the remote object’s IOR. Using the stub, a client can invoke methods and/or properties on the remote object in a transparent fashion, i.e., as if that object existed in the client’s own address space. The ORB intercepts the invocation at the client’s stub and is responsible for relaying the client’s request to the server for execution, as well as for returning the request’s results back to the client, once the server had finished processing the request. Request and reply data fields, such as parameters, are marshaled (serialized) by the ORB into a unified format called Canonical Data Representation (CDR) at each side before being sent, and are unmarshaled (de-serialized) at the receiving end. On the server side, the ORB communicates with the servant (server object implementation) through an Object Adaptor that the server object is connected to. In order to be able to process CORBA invocations, the servant extends a special base class called a skeleton. The CORBA client-server interaction is summarized in Figure 1.2.

The communication between remote clients and servers is through a high-level protocol called General Inter-ORB Protocol (GIOP). This protocol has several mappings to lower level transport protocols, of which the most common is its TCP/IP mapping called Internet Inter-ORB Protocol (IIOP). The server object’s interface, i.e., methods and properties, through which the client can communicate with the object, is defined in a platform-independent language called Interface Definition Language (IDL). Both the client-side stub and the server-side skeleton are automatically and independently generated from the IDL specification, using the mapping of CORBA to the specific pro-
gramming language being used at each side. Alternatively, a client may invoke a server object using the Dynamic Invocation Interface (DII). The DII is a generic API for constructing an invocation on-the-fly using only the IOR of the target object and not the IDL-generated stub. Similarly, on the server side, a servant may extend the Dynamic Skeleton Interface (DSI) base class instead of an IDL-generated skeleton. This way, a DSI servant is capable of processing any client invocation as a generic ServerRequest parameter.

The ORB itself supports only a minimal mechanism of remote object invocation, as described above. Other functionalities, some of which are also part of the CORBA standard, are implemented as CORBA services. A CORBA service is a set of CORBA objects that can be invoked in a standard fashion. However, CORBA service objects are not related to any specific server application, but rather are generic building blocks, used for constructing more complex service applications. Common CORBA services include, for example, the naming service [93], which provides a global directory that maps searchable names to object references. This service enables clients to obtain references to objects using a globally known name. Another popular service is the life cycle [92], which deals with dynamic object creation, destruction, migration and copying. The persistent state service (PSS) [97] defines object state serialization and storage mechanisms through interaction with back-end database systems. The concurrency control service (CCS) [85] coordinates concurrent client access to shared resources. The object transaction service (OTS) [94] provides transaction-processing semantics to sequences of object invocations. The CORBA services specification [88, 89] lists many more services.

One of the most effective extensions to the CORBA standard has been the Portable Interceptors (PI) specification [96]. This specification enables a standard mechanism for intercepting the client-server communication in either the client or server side at various processing points, such as sending or arrival of both requests and replies, depending on the side of the invocation the interceptor is located at. Interceptors are especially equipped to read and write service contexts, which are data records that may be piggybacked upon requests and replies. Service contexts can be handled by PIs transparently, i.e., without application awareness, and can thus assist in constructing transparent control mechanisms that do not interfere with application execution, such as collecting invocation statistics, associating requests with transactions, etc. CORBA PIs can also redirect requests to alternate target objects by throwing a specific LOCATION_FORWARD exception which aborts request processing and returns to the client that issued the request, causing it to re-issue the request at the alternate target.

On the downside, CORBA forbids PIs from modifying basic request data (operation, parameters, result/exceptions). Specifically, in the CORBA/Java mapping [91], which is the current environment of FTS, PIs are denied any access to request data because of Java security limitations. Generally, PIs have full r/w access mainly to service contexts,
which are data records that are attached to request and replies and are meant to be used by services and extensions but remain transparent to the application. Despite those limitations, the PI specification makes it possible to implement many required functions in a portable fashion, as we have demonstrated in a preliminary work [38]. FTS implementation relies heavily on this specification.

1.5 Middleware Fault-Tolerance

Fault-tolerance (FT), in general, consists of fault detection and fault handling [16]. Fault detection is the task of sensing the occurrence of events defined as faults, such as communication failures, software exceptions, abnormal sensor readings, etc. Once detected, event notifications are passed from the fault detector to the fault handler, which is in charge of responding to the fault events. The fault handler may take various actions, such as reconfigure the system to adjust to the new status, attempt recovery of failed parts, note the event in a journal, etc. The combined operation of the fault detector and the fault handler is supposed to enable a system to continue functioning as a whole despite faults.

Our research focuses on introducing fault-tolerance into distributed object-oriented software systems. Such a system consists of several independent server processes. Each server process is logically composed of objects. The objects are responsible for serving client processes. By executing client requests and returning the execution results to the clients. Client request execution are assumed to modify an object’s state in a deterministic fashion\(^3\). Thus, a service object resembles a state machine model [57, 109].

Being independent, each server process can experience faults separately. Objects in a server process may also fail independently of each other and of the process itself, although this is uncommon in real-life environments. Additionally, network links may fail, disrupting communication among the servers and between servers and clients.

Detecting failures based on middleware facilities alone is typically very limited. In CORBA, for example, this boils down to detecting only client-server invocation failures [87, 90]. Thus, the solution to adding fault-tolerance to a middleware must be capable of detecting additional failures that impose changes in the service configuration, namely server/object failures and loss of connectivity between servers. Many solutions [31, 72, 63, 84, 90, 64] add a complex logic to address all aspects of fault detection above the middleware for CORBA portability. However, in FTS, as in, e.g., Eternal [77], we chose to defer most of the fault detection (except CORBA’s provisions),

\(^3\)Actually, FTS allows execution of non-deterministic requests, provided that they do not change the state of the object, i.e., queries. See Chapter 3.
to a dedicated off-the-shelf product such as a GC toolkit (see Section 1.6). This yields a simpler design and separates specific middleware-related issues such as object-level fault-tolerance from generic fault-tolerance issues handled by the GC.

### 1.5.1 Replication Techniques

With both the GC and (to a lesser degree) the middleware providing fault detection, fault handling remains as the primary concern of FTS. Generally, the key concept behind fault handling is that of redundancy. That is, during normal operation (no failures), the system exploits more resources than minimally required for the service to function so that when failures do occur, a failed component may be replaced or recovered using these additional resources. Technically, we distinguish between spatial redundancy and temporal redundancy. Spatial redundancy refers to using excess resources to store and access service state information. This includes, for example, using multiple servers and multiple communication media between clients and servers. Temporal redundancy, on the other hand, means that more work is performed than merely sending a client request once, executing it and returning a reply. For example, a client may re-issue its request more than once, if it failed to get a response in the first invocation. Alternatively, a client request “execution” may involve recovering a failed server.

The most common approach to middleware fault-tolerance is that of replication [16]. That is, the server is instantiated more than once either over space or time. Replication techniques diverge as to how to maintain state consistency between the server’s replicas, so that clients experience a virtually continuous service. Active replication is a common replication technique that embodies spatial redundancy by maintaining multiple concurrent replicas of a server. Each client request is concurrently executed at all the servers, with the replies being condensed/suppressed prior to returning to the client. This requires that all the server replicas remain consistent despite handling client requests independently. Replica consistency is typically solved using an ABCAST transport service (e.g., using GC) to deliver client requests to all replicas.

Passive replication is the complementary technique that focuses on temporal redundancy. A client request is executed in a single server designated as a primary. The primary’s state modification resulting from executing the client request is propagated to a backup storage or server before a reply is sent to the client. Consequently, when the primary server fails, an alternate server replica is either re-instantiated from stored data or promoted from a backup state. Passive replication variants include cold passive replication [90], also known as checkpointing [30, 103] or cold backup, where a snapshot of the primary object’s state is committed to a stable storage log and retrieved upon recovery. Alternatively, a full checkpoint is generated only once every pre-defined period
and smaller incremental updates are recorded between checkpoints. *Warm* passive replication [90], on the other hand, propagates state information to running backup servers and may maintain only a small log of updates since the last checkpoint on stable storage. Once a backup is promoted to a primary status, it updates its state using the log. Propagating state updates to multiple backups from possibly multiple primaries may require using an ABCAST service, as in active replication.

Active and passive replication variants are comparable in numerous aspects [36]. From the aspect of implementation complexity, simple state backup is considered the simplest, with complexity increasing when using warm backups and up to full active replication. The reason for increased complexity is the need to propagate data to multiple independent servers while maintaining that all servers process the same set of messages in the same order. Active replication further adds the need to coordinate data from multiple origins (clients).

Another important advantage of using stable storage is the ability to survive complete site failures. Active and warm passive replication depend on at least part of the replication group to survive, since the state is partially stored in the volatile server’s memory. For that reason, almost every commercial fault-tolerant application is equipped with a backup storage solution, regardless of the replication method in use.

Resource utilization is another important aspect to consider when selecting a replication technique, since it directly affects service scalability. Active replication is typically considered more wasteful than cold passive replication in terms of CPU and network usage since it uses more processors to do essentially the same amount of work as that of a single primary server, with the added overhead of network-based coordination. However, cold passive replication may consume far greater storage resources due to its continuous state storage⁴. Warm passive replication typically consumes less CPU than active replication and less storage than cold passive replication but may require more network bandwidth depending on how state updates are propagated.

When responsiveness and speed of recovery is of the essence, active replication typically has an advantage over passive replication. The added operational overhead of cold passive replication involves working with relatively slow local storage systems, storing data that may be large depending on the object’s state and update policy. Warm passive replication stores less but still mobilizes state information. In contrast, active replication propagates only client requests, which are typically much smaller than state information. Also, active replication allows a client to recover from a communication failure either immediately (simply less replies to choose from) or using a very fast *fail-over* operation, where a client switches from a failed connection to an alternate one. In

⁴Passive replication techniques tend to purge storage logs of old entries (“log rotation”) to recycle storage space
passive replication on the other hand, clients have to wait for a server to resume, and
the recovery operation may be lengthy. In this case, warm passive replication is a little
faster than cold passive since it requires only applying the last updates instead of fully
restoring all the object’s state and then applying updates.

Last, there is the determinism of request execution to consider. In active replication,
when all requests are delivered to all the servers, requests must be executed exactly the
same way for all object replicas to remain consistent. In passive replication techniques,
on the other hand, a single server executes a request and commits the modified state.
Thus, a request may execute non-deterministically. However, this advantage dissolves
in case of a replicated object invoking another object. In this case, should the primary
server, as a result of a non-deterministic decision, invoke other objects (for example,
when using time-based load-balancing) and fail, a substitute backup may not repeat the
same decision and thus possibly violate system-wide invariants.

Since FTS was designed with performance in mind, it features mainly active repli-
cation. However, its flexible architecture can and is planned to be adapted to explicitly
support passive replication as well, as is customary with many fully-mature FT solutions.

1.6 Group Communication

In many applications in distributed systems messages must be disseminated to multiple
destinations with certain guarantees of reliability and message ordering. Consider, for
example, an e-mail service that relays electronic correspondence among multiple users.
The service typically has to ensure that every user eventually receives all messages that
are sent (reliability) and that each reply message is delivered to a user only after the
message to which it replies (causal order).

Specifically, as mentioned in Section 1.5, the service objects in FTS are assumed to
follow the popular deterministic state-machine model. Thus, objects can be actively-
replicated by having all service objects receive the same set of client requests (atomicity)
and in the same order (total order). The atomic broadcast (ABCcast) transport ser-
vice, which combines atomicity and total-ordered delivery, is frequently used in active
replication.

A group communication (GC) system [13, 16, 23] (or a GC toolkit) is a set of generic
services that facilitates the construction of a data transport that provides various guar-
antees. Such a transport operates within a well-defined group of processes. The partic-
ipating processes are called group members. Each member is notified by the GC toolkit
of arriving messages from the transport as well as changes in the group membership (see
below). Also, members can actively send messages through the transport, join and leave
the group, and fail. All of a group’s operations are usually performed and managed by group members without requiring an additional centralist entity.

A fundamental service of GC toolkits is the membership service. This service provides a consistent status of the current group’s membership, also known as the group’s view, to all group members. The membership service encapsulates a failure-detection mechanism that is typically distributed among the group members. Information from the failure-detectors is merged together with explicit member join/leave operations to generate a single coherent sequence of view change events that are delivered to all remaining group members, making them install new views of the group. Typically, installing a new view also involves electing one of the members as a coordinator of the new view. The coordinator’s identity is agreed upon as part of the new view and can be used in various distributed algorithms that require a leader.

This results in a fail-stop failure model [109, 110], where member failures are masked as leave operations. In this model, members that become excluded from a group following a view-change, are banned from communicating with the members still in the group by means of group transports, even if those excluded members have not actually failed but were only suspected of being failed.

The main service offered by GC toolkits is a group transport service that can be configured to have various properties. A transport can be reliable, that is, guarantee delivery of a sent message at all non-failed members, or safe/uniform, meaning that each delivered message is delivered at all non-failed members. A transport can also have message delivery ordering properties such as FIFO delivery that maintain sender order, causal delivery that maintains Lamport’s happen-before relation [57], or total-order delivery where messages are delivered in the same order at all the members. GC toolkits commonly utilize the network hardware’s broadcast and multicast capabilities to provide efficient implementations for the transport service.

GC transport services are often combined with the membership service, in the sense that the properties of a transport, e.g., regarding reliability and order, are guaranteed separately during the period of each view. The virtual synchrony (VS) model implemented in most GC toolkits ensures that messages sent to group members in one view are atomically delivered (all-or-none) provided that the members remain in the following view as well. By combining this model with total-order of client request delivery, consistent member servers indeed become “virtually” synchronous during a view, since they all process the same client requests in the same order, but most likely are never synchronized simultaneously in real-time.

Most modern GC toolkits allow network-wide GC services even when a process group splits into multiple components (or “partitions”) due to actual loss of connectivity (network partitions) or false failure suspicions. Accordingly, many GC toolkits offer a pri-
*many component* [23] service that distinguishes a single concurrent component as primary component. GC-based replication mechanisms (such as FTS) can use this distinction to restrict state progress to the primary component to avoid divergence of state between independent components. Furthermore, each new primary component is guaranteed to share some of its members with the previous primary component, so that state can be continuously propagated between consequent primary components.

Many GC toolkits further include specific services that are useful for maintaining state-consistency between replicas in GC-based replication mechanisms. The *state-transfer* service allows transfer of application state data from a process that was a member of the previous view to a new process that has joined in the new view (or a recovering process that has re-joined the group), in order to have all replicas in the same partition remain consistent following a view change. Also, various forms of *safe delivery* notifications [23, 74] let the application know which messages are delivered at all the members of the view, not only those that install the next view together (stronger than virtual synchrony). Such a service is especially useful to replication mechanisms, since it allows consistent progress of replicated state in the sense that each modification that is declared safe is guaranteed to affect each replica.

Notable GC toolkits include, among others, Isis [15, 13, 16], Horus [117, 118], Ensemble [1, 46], Totem [75, 73], and Transis [4, 28]. FTS generalizes on the GC operational concepts through a generic *GC Interface* (GCI) (see Section 3.2.2), and is therefore capable of being ported to various toolkits. The current implementation of FTS uses Ensemble as the underlying GC toolkit for implementing active replication of CORBA objects.
Chapter 2

Related Work

2.1 Adding Replication to CORBA

The problem of adding replication to middlewares has been researched extensively, and most of the works focus on CORBA. This section describes principal approaches at this problem based on [35] and [36].

2.1.1 The Integration Approach

In this approach, an existing ORB is modified and integrated with replication support code. A classic example is adding group communication code to support active replication, as described in Figure 2.1. As a result, the ORB can hide a group of object replicas behind a single client stub by using GC reliable multicast to send IIOP messages to the entire group of replicas. In essence, service replication can be made entirely transparent to both clients and servers, with the exception of state-transfer capability that needs to be added to the service objects. Another benefit of this method is that the replication implementation has direct access to both the inner workings of the ORB and the low-level communication stack, so it can achieve top performance. However, this method requires intimate knowledge of specific ORB internals, thus limiting the solution’s portability over different ORBs. Furthermore, the resulting integrated ORB is also proprietary and its CORBA compliance may suffer. Thus, clients not using a modified ORB might either break the inter- replica consistency, by accessing only a single replica instead of the entire group, or be banned access to the service.

Orbix+Isis [50] is the earliest example of this approach, which, true to its name, integrates the once popular Orbix [49] ORB with the Isis [13, 106] GC toolkit.
provides both active and passive replication. Furthermore, it allows failure-detection at object granularity. To support this, as well as state-transfer, service object implementations are required to inherit from a specific ActiveReplica base class. Application control of client-server interaction for choosing replication style, load-balancing, etc. is provided by overriding the default CORBA stub with an Orbix smart proxy.

Electra [60] is yet another classic example. Electra originally integrated the Horus [118, 117] GC toolkit into the ORB, thus allowing active replication of service objects with various degrees of transparency. However, Electra’s architecture can be easily adapted to other GC toolkits using a common virtual machine interface that is mapped to the GC toolkit using an adaptor object mediation layer. A similar property is featured in the GCI layer of FTS that allows FTS to integrate with a GC toolkit (see Section 3.2.2). Contrary to Electra, FTS does not modify the ORB or even depends on proprietary ORB code.

Maestro [14, 2, 119], another classic integration example, is basically an IIOP dispatch layer that is implemented on top of the Ensemble [1] GC toolkit. Active replication is provided through Maestro’s replicated updates request manager. For each replicated object, Maestro constructs a compound CORBA objects reference (IOR) containing references of all the object’s replicas, which clients can use for fail-over\(^1\), similar to FT-CORBA’s IOGR [90]. FTS, in comparison, also combines client fail-over with active replication to maintain the standard CORBA client-server interaction model and decrease the span of group management. However, FTS clients use only standard IORs pointing to a single replica. Fail-over is provided using a special redirection sub-service.

The Adaptive Quality of service Availability (AQuA) [26] framework capitalizes on Maestro to provide support for object replication. AQuA’s primary contribution is dynamic support for the application’s quality of service (QoS) requirements. These requirements are defined as specialized high-level quality objects (QoO) and are relayed to AQuA’s Proteus dependability manager, which actually handles the object replication, similar to the Replication Manager component of FT-CORBA [90]. Client-server interaction is channeled through AQuA’s proprietary Maestro-based gateways integrated into ORBs in both clients and servers for purposes of replication and monitoring. AQuA allows both active and passive replication, as well as detection and handling of object value faults (using majority voting) and response time faults beyond the common crash failures.

Recently, the OMG [44] has been considering extending CORBA with an Open Communication Interface (OCI) [101] specification, which would decouple the ORB from the underlying communication protocol (e.g., IIOP), thereby allowing dynamic selection of

\(^1\)A client fail-over operation is for the client to redirect its current request to an alternate server following suspicion of the current server’s failure.
alternate protocols as plug-ins, and as a result, better portability and interoperability between replication-integrated ORBs. Using OCI has already been demonstrated in recent works such as FTGCS [61] and PluggableFT [122] using the ORBacus [52] ORB that supports this mechanism.

2.1.2 The Interception Approach

In this approach, client-server CORBA communication is intercepted outside the client and server ORBs by a separate interceptor component and re-routed through a replication mechanism that propagates requests to server replicas and also collects and suppresses replies returned to clients. This approach is transparent both to the ORB and to the CORBA application and features good performance by using efficient low-level replication mechanisms that avoid the ORB operation overhead, such as a GC toolkit for active replication (see Figure 2.2). However, the interception mechanism itself is usually not portable, even when using CORBA PIs, since replicating client invocations requires more control over request/reply data than is allowed by the PI specification and is usually achieved using extra non-portable code.

Eternal [77, 80, 81, 78] is the first and most familiar example for an interception-based
system. Eternal intercepts the system calls of both client and server ORBs using UNIX’s /proc file-system interception and through library interposing [77] - replacing dynamic libraries with stubs at loading time. The resulting interceptor has very limited portability since it depends both on the operating system (UNIX in this case) and knowledge of each ORB’s ins and outs, although it does not change the ORB.

With the emergence of CORBA’s portable interceptors (PI) specification [96], it was made possible to write portable interception code, which FTS indeed exploits for replication. However, as already mentioned, a CORBA interceptor may not access the request data when using CORBA in security-aware languages such as Java [91]. Thus, a CORBA PI cannot be used for actually relaying the request content to a replication mechanism. FTS compensates for this limitation by extending the server-side interception mechanism using CORBA’s portable object adaptor (POA) and the DSI mechanism [87], as is explained in Section 3.2.1. In the client side, FTS may need to extend the interceptor with a transparent proxy to generate unique client request ids, as explained in Section 2.1.2.

Other notable interception examples that use CORBA PIs are Open EDEN [37, 43] and DAISY [11]. Open EDEN supports both active and passive replication of generic objects. To overcome the access limitation of CORBA interceptors in Java, Open EDEN implements the server-side code in C++. Another CORBA PI limitation of being unable to suppress unwanted requests/replies is handled in Open EDEN by means of gateways (see Section 2.1.2). The second example, DAISY, uses CORBA PIs to implement a simple passive replication and uses Java serialization for state-transfer between a primary and a backup. The PI is not used in DAISY for replicating the request, just relaying its serialized form.

Using Gateways For Interception

Another interception alternative is that of gateways (or proxies). A gateway is a stand-alone CORBA application located between the client and the server. A gateway mediates client-server interaction, acting as a server for the client and as a client for the server. Contrary to an interceptor attached to an ORB, a gateway has no need for application transparency and as a result can be implemented in a much simpler and more straightforward fashion. Also, being a full-fledged CORBA application, a gateway suffers from no message access limitations such as those of the CORBA PIs. On the negative side, a gateway is another component that needs to be made fault-tolerant to avoid becoming a single point of failure. Also, gateways add considerable overhead of request processing that may even exceed that of a low-level interceptor due to the added two-way request processing (both as a server and a client) and possible congestion of client-server traffic.
in the gateway.

[32] suggests construction of generic actively-replicated gateways that mediate between clients with fail-over capabilities, such as FT-CORBA clients, and multiple instances of un-replicated servers. Each gateway, upon receiving a request from a client, invokes the request at all the servers and returns a unified reply to the client. Server consistency is maintained by having the gateways totally-order client requests among themselves before invoking the servers. Such a technique would greatly simplify adding fault-tolerance to existing CORBA services since it removes the need for consistency maintenance and coordination from the servers, thereby eliminating the added complexity. However, this technique is limited to rather simple applications, such as read-only/stateless services or services with idempotent update operations. The reason is that if a gateway invoking an update fails or the gateway-server links fail, it is impossible to tell which of the servers has already processed the update, hence the need for recoverable updates. Last, this technique is not suitable for passive replication since this would require state-transfer abilities from the application servers.

Open EDEN [37, 43] may be considered a slightly different version of [32] that allows both active and passive replication, as well as working with generic objects. Compared to [32], Open EDEN also uses fail-over-capable clients with replicated gateways, but also attaches portable interceptors to the servers. Also, in Open EDEN the gateways only handle unreliable multicast of requests to the servers, whereas the server-side interceptors add the reliable request dissemination and ordering.

Gateways are also used in Eternal [82] and in the FT-CORBA [90] standard, but in a concept that is opposite to [32], that is, to connect standard clients with no FT capabilities to FT-enabled servers. The gateway is needed in this case in order to bridge over protocol differences. The client interacts with the gateway using standard IIOP, whereas the gateway interacts with the servers either using IIOP augmented with FT-specific extensions (special object references and service contexts) or using an entirely different protocol, such as a proprietary ABCAST protocol. Thus, the gateway does not ensure fault-tolerant service to a non-FT client, but rather the basic ability of the client to use the service. FT-enabled clients do not need the gateways.

FTS does not require stand-alone gateways (see Section 2.1.4), since the FTS server itself takes care of bridging between IIOP and the internal ABCAST protocol, yielding a much simpler architecture. However, in the absence of an authority that generates unique request ids for FT-clients, as is the case with legacy ORBs that do not support FT-CORBA, FTS attaches a portable transparent proxy object [38] to each FT-client in order to generate the request id².

²This is due to another CORBA portable interceptor limitation that forbids the client-side interceptor from assigning any identifier to a request that can also be assigned to its consequent retries.
Reflection

Exploiting reflective capabilities of the middleware and/or service object implementation language presents an interesting alternative to replication implementation that bears much similarity to the classic interception approach. Reflection enables generic run-time exploration and control of a program's components, such as objects, classes, threads, etc. Specifically, it may allow attaching a replication mechanism to a service implementation by means of hooking control code to various events of interest, e.g., method invocation/return, object creation/deletion, etc. This concept has been generalized in the recent Aspect-Oriented Programming (AOP) [55] technique that allows generic transparent integration of modifications into an unchanged code-base through the use of hooks.

FT-MOP [54] is an example of using reflection to implement replication of CORBA objects. In FT-MOP, each service object is assigned a control object that is called at the hooked events to perform replication-related tasks, such as transferring a state in passive replication following a method invocation. FT-MOP is implemented in C++ that provides very little reflection support. Thus, it uses a specialized compiler to insert the needed hooks into the program at compile-time. FT-MOP further uses that compiler to analyze the structure of objects and generate automatic code for state transfer.

Programming language based-reflection seems much more powerful and flexible than mere interception. However, any specific language dependency is considered proprietary when used with a language-independent middleware such as CORBA. On the other hand, CORBA’s own reflection mechanisms, such as the dynamic request processing facilities and portable interceptors, are extensively used in many generic replication mechanism, including FTS.

2.1.3 The Service Approach

In this approach, the entire replication mechanism is implemented as a CORBA service [88]. Figure 2.3 shows a conceptual implementation of a CORBA active replication service. The service approach yields a CORBA-compliant and, as a result, a highly portable and interoperable implementation, which can be used as a building block for applications and higher-level services. A typical active replication service involves the construction of a group communication service. Passive replication, on the other hand, is customarily implemented in a straightforward fashion.

The most prominent drawbacks of this approach are lack of transparency and high performance overhead. Transparency is limited at most to the client side where the service function of fail-over, multicast etc. may be encapsulated inside a transparent
Figure 2.2: The Interception Approach

גשת היירוט

proxy or a client-side interceptor. On the server side, however, explicit interfaces must be defined between the replication service and the higher-level application. Furthermore, replication services incur significant performance overhead [33, 70] as a result of service objects communicating using the ORB, compared to, e.g., low-level TCP or UDP. Also, the legacy CORBA communication model permits only reliable point-to-point communication, which is inefficient for implementing multicast operations, as required for active replication.

The service approach is explored in numerous works, for its portability and fitness to the CORBA model. The Object Group Service (OGS) [31, 33, 34] is a classic example of a CORBA object replication service. OGS consists of a hierarchical set of CORBA sub-services that is analogous to a typical GC toolkit, including services for unordered reliable multicast, messaging, consensus, monitoring (failure detection) and membership. Total ordering of client requests is achieved through a sequence of consensus sessions, similar to the technique that was first demonstrated by [18] and [59]. At the top level, OGS provides a unified support for both active and passive replication styles. Specifically, passive replication [27] is implemented by using the membership service to elect a primary, which can be rotated if suspected by the client. The primary sends the updated object state to the backups and then all the participants agree on the update before committing it, to ensure they all see the same update sequence.
OGS was later followed by many similar examples, such as NewTop [72], OFS [63] and Timewheel [70]. NewTop, specifically, is a GC service that allows partitionable operation. However, NewTop only guarantees that each partition has continued GC services, without enforcing global service consistency, e.g., by enforcing a single primary partition.

Another well-known service approach example is that of the Distributed Object-Oriented Reliable Service (DOORS) [84, 83]. Its main contribution is as a reference implementation of much of the (then emerging) FT-CORBA [90] standard. DOORS focuses on passive replication and does not construct or use a group communication system. Similar to FT-CORBA, it uses a separately-replicated hierarchy of management and failure-detection component for identifying and handling both object- and host-failures. DOORS clients perform fail-over using a list of alternate object references embedded in FT-CORBA’s IOR group reference. However, outside the scope of FT-CORBA, DOORS clients handle complete service group change by falling back to the persistent replication management asking for updated references of service objects. Following the FT-CORBA standard, using DOORS requires extended FT-CORBA-compliant ORBs such as TAO [108] that can work with IORs.

The Interoperable Replication Logic (IRL) [67] is another notable service implementation of FT-CORBA active replication but without using a GC service. IRL provides active replication in a portable fashion that is compatible with legacy ORBs using standard IORS and allows ORB heterogeneity, similar to FTS. However, unlike FTS, IRL is fully visible to applications and implements ABCAST through a sequence of unicast invocations (instead of a more efficient low-level multicast). Furthermore, IRL detects service object failures through invocation time-outs, which might be inaccurate depending on request execution time and server load.

Future CORBA group communication services are expected to provide better performance without losing portability with the recent introduction of CORBA’s Multicast Inter-Orb Protocol (MIOP) [100] messaging standard. This new standard defines an unreliable multicast transport of one-way invocations, that is, invocations that require no response from the server. This new protocol is mapped to efficient low-level multicast primitives such as IP multicast, and can be used as an initial building-block for, e.g., a high-level ABCAST implementation. Furthermore, the OMG has already issued an initial RFP (Request For Proposal) for a higher-level abstraction of a Reliable Ordered MIOP (ROMIOP) [98], which is currently undergoing revision. This abstraction can serve as a standard future basis for object replication that requires much less additional logic compared to current implementations.
2.1.4 FTS: The Object-Adaptor Approach

FTS embodies a new approach named the Object-Adaptor approach (see Figure 2.4). In this approach, replication is embedded in a mechanism that completely envelops the server ORB. Similar to standard interception, this mechanism controls the interaction between the ORB and the OS using CORBA PIs. However, unlike any previous system, FTS also controls the interaction between the ORB and the service objects, by creating a new type of an object adaptor\(^3\) called the *Group Object Adaptor* (GOA). Thus, our approach is so named.

The GOA is a single component that encapsulates all the mechanisms that are required for managing active replication. FTS clients communicate with only a single server. Thus, the GOA’s basic function is to intercept the client requests and propagate them to other GOAs that manage replicas of the same object using an underlying GC toolkit. However, there are many sub-functions built into the GOA as part of its operation, including caching replies, performing state-transfer, providing various request execution models, monitoring objects, etc. More information about the GOA can be found in Chapter 3.

\(^3\)An object adaptor is a CORBA component that mediates the communication between the ORB and its supported objects.
The object-adaptor approach is, in essence, a merge of qualities of the previous approaches. By controlling ORB interaction, FTS may be considered an extension of the interception approach. However, the interception is not entirely outside (or below) the ORB, as is the case with classic interception-based implementations. Furthermore, FTS uses only standard CORBA components without modifying the ORB, thus remaining completely portable, contrary to other interception implementations.

FTS neither fits the integration approach. Indeed, FTS integrates a GC toolkit to maintain consistency efficiently through its ABCAST service. However, only FTS code is integrated and the ORB is unmodified. Additionally, FTS maintains a degree of portability over GC toolkits, by attaching to the GC toolkit indirectly through a common GCI interface. Thus, despite the integration, FTS remains portable.

Last, most of the logic of FTS is encapsulated in the object adaptor that operates above the ORB, in the application level. Also, other than the GC toolkit, FTS builds upon standard unmodified CORBA facilities. These make FTS a close relative of the service approach. In fact, some of the conclusions of FTS’s performance test (Chapter 5) reveal that it suffers from limitations that are common to CORBA services in general. However, contrary to common CORBA services, FTS can be made nearly transparent even to the objects of the application service that uses it since it’s functionality is hidden in the black box of the GOA, whose interface closely resembles that of the standard POA, thus “embedding” FTS inside the CORBA layer. This considerably eases the porting of non-replicated services into FTS, compared to explicit interfacing with a replication service. Actually, the degree of exposure of FTS can be as small as that of the integration or interception approaches - the application is required to provide only state-transfer and
monitoring capabilities at the object level\textsuperscript{4}.

Aquarius [22] is a more recent CORBA active replication service that followed FTS to use the Object-Adaptor approach. Aquarius directly implements the Paxos [58, 59] ABCAST protocol where client proxies (gateways) function as initiators of requests and servers as acceptors. Hence, actual request ordering is done at the proxy level. Each server has a Quorum Object Adaptor (QOA) that performs the acceptor role of Paxos prior to actually invoking the objects. The client proxies are servers designed to increase overall scalability by reducing the number of Paxos initiators (instead of clients themselves), from which a leader needs to be elected in order to make progress\textsuperscript{5}. The proxies are stateless since they can learn their state through Paxos. Thus, they can be restarted or switched for fault-tolerance. References to replicas of the same object are encapsulated in a single FT-CORBA [90] group reference.

The resulting architecture of Aquarius seems more scalable than that of FTS using GC since it does not require servers to communicate at all and relieves them of message ordering and reliability overheads. However, GC toolkits provide greater flexibility in selecting protocol guarantees. Thus, FTS, for example, can choose to use only a fast and simple FIFO transport when allowing only a single server out of a whole cluster to serve clients. Furthermore, for portability, Aquarius uses CORBA invocations instead of more efficient low-level communication primitives offered by GC toolkits. Also, unlike FTS, Aquarius does not allow non-participating clients, i.e., clients that are not aware of proxies, to access the replicated service since doing so would harm replica consistency. Last, in order to bypass the scalability limitations imposed by common CORBA threading models (see Chapter 5), Aquarius modifies the proxy ORB to implement its own threading model that allows the same server thread to serve multiple client requests without having to finish serving one request before serving another. FTS, on the other hand, avoids ORB modifications, and aims to achieve the same scalability effect using a more portable technique, as discussed in Chapter 5.

2.2 The Fault-Tolerant CORBA (FT-CORBA) Specification

As part of its ongoing effort to create comprehensive standards for various aspects of distributed computing, the OMG, in January 2000, incorporated into the CORBA stan-

\textsuperscript{4}Even state-transfer may not be required in future versions of FTS if service objects are defined as CORBA value-types, which are objects whose state can be automatically read and written using standard CORBA facilities.

\textsuperscript{5}Aquarius uses a probabilistic back-off protocol to select a leader. The back-off protocol assumes a known bound on the time it takes to order a request.
A generic fault-tolerance specification called FT-CORBA [90]. This specification is currently implemented in several projects including DOORS [84, 83], IRL [67] and Eternal [80, 81, 78].

As can be realized from its name, the FT-CORBA specification defines more than a mere CORBA service. It includes a wide set of interfaces, policies and CORBA enhancements. However, FT-CORBA still draws its main strength from the service approach, i.e., most of the mechanisms depicted in the specification are CORBA-based and CORBA-compliant. Another approach, which FT-CORBA utilizes, is the interception approach, in the sense that requests and replies are intercepted, mainly for the purposes of redirection and logging. Last, although not explicitly promoted by the specification, FT-CORBA actually builds on augmented (or enhanced) ORBs in order to function.

The basic method used for achieving fault-tolerance in FT-CORBA is object replication. FT-CORBA defines fault-tolerance domains (ft-domains), where object replication is provided within specific administrative and/or application boundaries. In each domain there exists a single fault-tolerance infrastructure that consists of a hierarchy of fault-detectors and fault-notifiers, as well as of a management object called Replication Manager, as described in Figure 2.5. All the management and failure-detection objects are replicated themselves to avoid becoming a single point of failure.

Each object implements a Monitorable interface that allows it to be monitored by
the failure detectors, and a **Checkpointable** interface for state-transfer. When a service object is suspected of being failed by the failure-detection mechanism, the Replication Manager is notified and takes proper action, such as starting a new replica and updating its state, so that the replication group is dynamically stabilized. In the case of passive replication, corrective action may also include making one of the replicas the new primary. Other than handling failures, the Replication Manager is also responsible for creating and deleting replicated objects. To that end, it uses its global knowledge of each object’s replication group as well as lists of replica factories on different hosts provided by the application.

FT-CORBA allows the application to configure the replication method of each object by passing selected properties and policies to the Replication Manager. These include the **replication style** (stateless, active w/o voting, cold/warm passive), list of available hosts/factories, group size limits (minimum/maximum), and frequency of checkpointing and monitoring. Also, the application can specify its degree of involvement in creating/deleting replicas in **membership style** and in providing various replication aspects of monitoring, logging, recovery, etc. as **consistency style**.

Object groups managed by the same infrastructure are included in the domain. Hosts, however, may belong to multiple domains. This allows applications to span multiple domains, and have several fault-tolerance mechanisms handle their entire set of replicated objects. Thus, applications can be made scalable.

FT-CORBA maintains that object replication be transparent to clients at the application level. This means that a client communicates with an object but is unaware whether the object is replicated or not. This transparency is enabled through the introduction of two new enhancements to the CORBA standard, and particularly to the ORB core: **Iogr** and the **redirection** mechanism.

An **Interoperable Object Group Reference** (IGR) is an extension of the CORBA IOR. According to the pre-FT-CORBA standard [86], the IOR profiles are supposed to be independent references of the same object implementation that are used separately by different CORBA protocols, such as IIOP, except for a special type of profiles (TAG_MULTIPLE_COMPONENTS) that can be shared by multiple protocols. IGR has the same structure as IOR, but incurs additional semantics to the structure. In an IGR, there may be several IIOP profiles, each pointing to a different replica of the target object.

If the target object is passively-replicated, then one of the IIOP profiles is tagged as the **primary** profile, i.e., the one which is supposed to be accessed first in order to perform a request on the target object. Also, there exists one shared profile that contains information common to the entire group of replicas, such as group identifier, group version (see ahead), etc.
A client needs to be aware of the current membership (or view) of its target object group in order to perform invocation and fail-over correctly. Thus, the IOGR held at the client should match the actual membership of the server group. A change in the group’s membership might cause the client’s IOGR to become stale, i.e., inconsistent with the current membership, possibly making the client direct its next request to failed servers although there are working servers ready to process the client’s request. In order to handle stale IOGRs, a versioning mechanism is used: each group’s membership is assigned a version number, which is updated as the membership changes. When a client accesses a server, the membership version number of the client’s IOGR is attached to the request in the form of a CORBA service context [87, 90]. The server ORB compares its known version of membership with that of the client’s request. If the server’s version is newer, indicating a stale client IOGR, it replies to the client with a LOCATION_FORWARD exception [87], which, when processed by the client, automatically updates the client’s IOGR to the new version6.

In order to enjoy the benefits of a fault-tolerant service according to FT-CORBA, a client has to be mounted on top of an enhanced ORB that supports FT-CORBA extensions, among which using IOGRs. In case of passive replication, the FT-enhanced client ORB is expected to direct the first invocation of the request to the target replica designated as primary. If communication with that server fails, according to a set of well-defined fail-over conditions [90], the ORB should try to automatically redirect the request to another replica by using another profile from the IOGR. Once all the profiles have been tried and failed, an exception is returned to the client. In the case of an actively-replicated object, the FT-enhanced client ORB is responsible for delivering its request to all the replicas of the object concurrently and consistently, which is typically achieved through the use of an ABCAST service, such as that provided using a GC toolkit. To accommodate legacy or partially enhanced client ORBs, FT-CORBA permits the use of gateways to bridge between standard IIOP and the group protocol. Gateways can be made fault-tolerant using the same fail-over principle of passive replication.

The FT-CORBA specification requires that each ORB be equipped with logging and recovery mechanisms. The logging mechanism records both the state of service objects (checkpoints) as well as incoming client requests. The recorded log is used for multiple purposes. When the service is passively-replicated, the recovery mechanism uses the log (last checkpoint + following requests of the same object) to promote a backup server to primary status following primary suspicion. When the service is actively-replicated, the log is used for state-transfer when adding new members to the object group. The log is also used for detecting multiple invocations of the same request, in

---

6If the request’s group version is higher than that of the server, the specification only requires that the server retrieve the group’s most up-to-date version. Although the client’s request is assumed to be authentic, view change of the server group is triggered by a separate failure-detection mechanism.
which case subsequent invocations other than the first are responded with a recorded reply to maintain CORBA’s at-most-once semantics. The log is purged of old useless data to restrict its growth. All checkpoints of the same object other than the last are deleted. Client requests/replies are removed if they occur before the last checkpoint of the target object and a timeout that ensures the client will not re-invoke them has expired.

For replication consistency, FT-CORBA requires that the infrastructure maintains strong replica consistency, which means that an object’s replicas’ states should be kept identical even in the presence of faults. For active replication, replicas are required to be consistent at the end of each method invocation, when the client has received replies from all the replicas. For passive replication, replica consistency is required at the end of each state-transfer from the primary to the backups. This definition is a bit vague since it does not define, for example, whether clients and servers are supposed to be actually synchronized in real-time upon completion of request invocation or state-transfer.

Furthermore, FT-CORBA requires for active replication that each group member process every client request in the same order. For passive replication, only one member (the primary) is required to execute the request. Combining either of these requirements with strong replica consistency (even ignoring real-time synchronization) and FT-CORBA’s requirement for deterministic request execution (see below), we have that FT-CORBA permits only linearizable consistency of replicated service objects.\footnote{According to FT-CORBA, strong replica consistency implies strong group membership, meaning that each request is done executing either before or after handling membership change. When using GC, this is typically ensured by using safe delivery of requests.}

The FT-CORBA specification also defines several limitations of its methods. First, as the reader could have already observed, clients need FT-enhanced ORBs to enjoy the benefits of FT-CORBA-based fault-tolerance. Second, FT-CORBA requires using a homogeneous infrastructure based on the same vendor’s ORBs inside the same ft-domain to ensure interoperability, especially since it permits non-standard augmentations such as group communication integration etc. Third, FT-CORBA assumes that all service objects feature deterministic behavior for maintaining strong replica consistency. Fourth, FT-CORBA does not handle network partitions, mainly since it allows the replication control to exist separately from the data. Fifth, FT-CORBA does not handle generic Byzantine (or commission) failures, other than notify reply inconsistency to the application. Last, FT-CORBA does not handle correlated faults, that is, faults that result from design or programming bugs. Obviously, this results from FT-CORBA relying on homogeneous implementations.

As the reader can see, the specification of FT-CORBA is rather large and complex, and so is comparing it to FTS. In Chapter 3 we detail such a comparison after presenting
the relevant parts of the FTS architecture. As one of the goals of this work, we seek to improve this specification, both for CORBA and for middlewares in general.

2.3 Request Batching/Packing

An early classic example of batching (or packing) large amounts of information into a single packet can be found in the Nagle algorithm of TCP [76], where it is used as a means to improve a protocol’s throughput by reducing its framing overhead, i.e., the ratio between its header size and payload size. More specific evidence of the advantages of packing together multiple messages into a single packet to significantly increase the throughput of an ABCAST protocol can be found in [41]. This work measured throughput and per-message latency of popular ABCAST protocols including the rotating-token (Rottoken) used by the Totem GC toolkit [75, 73] and the dynamic-sequencer (Dynseq) used by Amoeba [53]. Additionally, [41] also tested three variants of these two protocols, which were augmented with message-packing mechanisms.

The Rottoken protocol in its classic form allows a process to order and send its pending messages only when it receives a token containing the ordering data that is rotated between the processes in a logical ring order, after which it passes on the token, piggybacked with the last pending messages or explicitly if no messages were pending. The first packing variant of Rottoken (Denoted Rotfc) that tested in [41] simply packs the pending messages together upon token reception and piggybacks the token. Another semi-packing variant of Rottoken is request-token (Reqtoken) where the token passes upon request only between the actively-sending parties. Reqtoken packs only messages pending from before the token reception, so (typically few) additional messages will be sent unpacked until the token is passed on. Last, the Dynseq protocol orders messages by having the senders forward them to a sequencer process. Its packing variant (Dysfc) consisted of simple time-based accumulation of messages at the sender in a 1-msec window.

Tests of the protocols in [41] using bursty clients (single sender and all senders) and very short (1-byte) messages show a dramatic increase, by several orders of magnitude, of the throughput of the packing variants compared to all the non-packing protocols. The throughput increase is coupled with a significant decrease in average message latency, despite the delay imposed by the buffering of messages. These effects were attributed to the amortization of per-message overhead of headers and interrupt processing over multiple messages, as well as to reduced network contention due to the decreased number of on-the-wire messages.

Further support of the performance improvement of Totem using message batching is provided in [79]. In this work, Totem is augmented with a simple count-based mes-
sage batching. That is, messages are added to a buffer until the accumulated buffer size surpasses a threshold, at which point the buffer is sent as a single packet. The resulting mechanism is both analyzed using stochastic tools and empirically tested, proving the increased throughput. Furthermore, this work checks performance under varying message size, concluding that the throughput gain of batching decreases when batching larger messages, up to a no-batching level, i.e., as if no batching is applied.

The count-based batching method is also explored in [25], combined with the on-demand protocol that resembles the Reqtoken protocol. The tests involved scenarios of a single and multiple senders with varying degrees of overlapping bursts and a fixed message size of 17 bytes, resulting in a 3- to 5-fold increase in throughput.

A recent paper [17] presents a general claim that increasing the batching factor, i.e., the number of messages-per-packet, contributes to increase the throughput of reliable multicast protocols, regardless of the batching method being used. The rationale behind this claim is that as networks bandwidth grows significantly larger (Gbit/sec), the bottlenecks of message transmission shift from the network to the end-points processors, i.e., the senders and the receivers. Message batching reduces processing time at the end-points through overhead amortization, thereby increasing throughput. Using probabilistic analysis with conservative restraints of loss probability, the claim is proved for two simple types of unordered reliable multicast, based on positive and negative acknowledgments. Further empirical study of both protocols shows that the throughput indeed improves up to an order of magnitude as the batching factor grows.

One key factor that is missing from [17] is the message arrival rate. In other words, the model that is used for analyzing the throughput assumes that no time is wasted on waiting for arrivals of new messages to add to the current batch. As we show in our work, the latency imposed by the batching can actually lead to a much smaller increase of throughput, especially when the message arrival rate is low. The reason is, of course, the time overhead that is the sum of all the inter-batching delays together with the delay from the batching of the last arriving message to the batch send event. Thus, in order to achieve a large throughput increase, batching must be tuned or adapted according to the message rate.

A first-time notion of the need for adaptivity in batching mechanisms is presented in [10], using the Spread [5] GC toolkit and count-based batching. This work is focused on reaching and maintaining maximal throughput in a dynamic environment. First, an observation is made that throughput varies when using a constant batching threshold due to changes resulting both from porting the system to various platforms and from runtime fluctuations of message rate and size. Next, this work presents a simple rule of improving the throughput by constantly measuring the throughput and modifying the threshold accordingly. The result is close to peak throughput but has a tendency
to oscillate due to temporal throughput measurement inaccuracy. Furthermore, the rule performs little modification steps and is likely to lock on to a local maximum of throughput rather than to advance to the globally-maximal throughput.

Last, the impact of batching is demonstrated in [9] when applied to a different context of implementing content-based publish/subscribe communications on top of structured overlay networks. In general terms, this paper suggests extending the distributed hash table (DHT) function of the overlay to map both event-publications and event-subscriptions to partially-overlapping sets of overlay nodes so that for each subscription $\sigma$ and event $e$ that matches the subscription, there is at least one node $r$ (designated as a rendezvous node) that will match $e$ to $\sigma$ and forward a notification by multicast to all the listed subscribers. By performing time-based batching of event notifications for the same subscription, the average number of hops required for event notifications is significantly reduced.

Following the works above, we also add request batching abilities to FTS in order to increase its throughput. Because of portability considerations, FTS batching was made independent of the underlying GC component, assuming no knowledge of the GC protocol specifics, contrary to, e.g., the Rotfc protocol in [41]. Furthermore, similar to [10], our work tackles the subject of batching adaptivity, although from a different angle. We argue that good throughput can be maintained over a wide range of message rates by varying the batching threshold with the rate, in order to ensure that batching latency does not grow without actually adding messages. Therefore, applying naive time-based and count-based methods, as demonstrated by [17, 25, 41, 79], can be costly in terms of batching latency, e.g., when the message rate is too low w.r.t. the time or count thresholds. Our work also provides two variants of protocol-independent batching algorithms that have inherent adaptivity to message rate in the sense that they modify their behavior according to the message rate without requiring explicit measurements, as opposed to [10]. A single parameter that we call the doorway length controls the dynamic behavior of these algorithms in a wide range of message arrival rates and provide throughput that is close to or even better than that of the classic methods.

### 2.4 FTS Request Cache and Selective Acknowledgements

Caches are employed in numerous types of applications. Typical examples can be found in areas of memory caching [115], web caching [121], file-system caching [102], database caching [112] and middleware object caching [20]. All of these cache implementations share a common quality of an information proxy. That is, the cache is used to store
a copy of remote information (e.g., data, instructions, objects) in a location that is closer to the component that uses it than the source of the information, in order to improve access latency and/or throughput. In that sense, a cache may be considered an optimization that replaces direct access to the remote data, since it still maintains similar guarantees. Continued access to information (liveness) is ensured at the cache by techniques for retrieving remote data that are missing from the cache, e.g., cache miss and pre-fetching. Additionally, cache contents are kept correct w.r.t. the origin (safety) by having the cache guarantee a certain consistency condition, such as linearizability or sequential consistency [20].

In contrast, the request cache in FTS is not an optimization at all. Rather, it is used to guarantee a safety condition of at-most-once semantics of request execution at the server. In other words, the cache ensures that the same request is not executed in an FTS server more than once, even if the client that sent it originally re-invokes the request due to not receiving the reply in the first time, e.g., following a temporal communication disconnection. In such cases, at-most-once semantics are maintained by having the client receive its reply from the cache instead of by re-executing the request, once the request is done executing for the first time.

A subsequent second difference between the request cache of FTS and typical caches is how the cache size can be limited. In typical caches the cache size can be arbitrarily limited by using various policies, e.g., LRU or LFU to enforce removing items when the cache becomes over-full. These policies do not harm the availability of the information since removed items can be retrieved back to the cache from the origin, e.g., when needed. In FTS, however, an update request that is removed from the cache without first ensuring that the client will not re-invokes it, may result in the request being re-executed, thereby violating the at-most-once safety condition. Thus, all update requests must be placed in the cache. A completed request can be removed only after ensuring that the client will not require the reply again. This may cause the cache to grow indefinitely depending on the cluster throughput and client load, instead of up to a fixed size. Alternatively, in order to set a fixed limit to the cache size in FTS, execution of newly arriving requests should be delayed/aborted when the cache exceeds its limit.

As one can see from the text above, much more resemblance can be found between the FTS cache and the send-buffer management algorithm of reliable transport protocols, such as the sliding window of TCP [113]. Both the request cache and the send-buffer basically guarantee that the respective other side, namely the client in FTS and the other peer of the TCP connection, eventually either receives the stored data or fails/disconnects. However, the FTS cache guarantees this property at a higher layer than transport, where a client may re-open a connection more than once in order to receive a reply to the same request. Thus, despite the typical transport layer of CORBA being also TCP, the cache cannot rely on the guarantees of a single connection to de-
liver the reply. Furthermore, since the FTS cache operates above the ORB level, it has no notion of successful delivery of reply data to the client due to CORBA limitations\(^8\). Consequently, it has to rely either on timing assumptions or on application-level acknowledgments in order to be able to safely discard the reply. TCP, on the other hand, uses timing assumptions only to detect whether the opposite peer has failed/disconnected and all of its packets are removed from the network. In contrast, TCP discards data from the send-buffer only upon receiving acknowledgments of successful delivery.

There are several other well-known but non-CORBA distributed services that employ a server-side time-based request cache for the purpose of ensuring at-most-once request execution semantics, most notably the Remote Procedure Call (RPC) [114] and the RPC-based Network File Service NFS [102] services. However, to the best of our knowledge, no published works consider the scalability problem that could be imposed by such a cache and how to reduce such a problem using additional client acknowledgments.

The logging mechanism of FT-CORBA [90] operates quite similarly to the request cache of FTS. Both mechanisms store completed requests and use a time-out to discard it. Thus, applying our suggested improvement of adding SACK can contribute to reduce the log size of compatible implementations. Our specific implementation of SACKs in FTS where the receiving server propagates received SACKs to its peers using ABCAST to ensure causality (SACKs after matching requests) and to make SACKs effective at all the cache replicas, would also suit the warm-passive replication mode of FT-CORBA where a single primary updates the state of its backups. Active replication in FT-CORBA is less relevant since it assumes that a client receives its replies from all the servers using an ABCAST transport, in which case the log may not be needed for caching replies. One specific variant allows the client to communicate with the servers via a gateway, in which case the gateway should cache replies, and then SACK support can be applied to gateway replication.

However, not all FT-CORBA-compliant implementations use time-based logging. In Eternal [77, 80, 78], for example, a request/reply pair is removed from the log only when the next request from the same client arrives. This is because Eternal assumes a certain behavior model for clients in which a client may issue at most one request at a time, sending another request only after receiving a reply to the first one. Thus, the arrival of a new request is an implicit acknowledgment of receiving reply for the previous request, which can now be safely discarded. This results in a cache that is only as large as the number of clients, but severely limits client behavior\(^9\). Contrary to this, FTS assumes no

---

\(^8\)A future modification of FTS (see Section 8.3.3), may notify a server of a successfully-delivered reply to allow for immediate removal of the matching request from the cache.

\(^9\)In fact, without an backup evacuation timeout, the Eternal cache can overflow when many clients interact for short periods with the same service and then cease to. However, we found no documented evidence of such a timeout.
limitations to client behavior that may result in request dependency. An FTS client may invoke requests asynchronously or be multi-threaded. Consequently, requests can only be discarded based on explicit acknowledgments or time-outs. On the other hand, using time-outs impose stronger synchronization requirements in FTS (and in FT-CORBA) compared to Eternal, in the sense that there need to be known upper bounds on the times of request execution, client retry and client-server communication.

In order to reduce the cache size, FTS now employs explicit selective acknowledgments (SACKs), added to client invocation as CORBA service contexts. The standard ordered acknowledgments (ACKs) used by any sliding-window or go-back-n transport protocols such as TCP are much more efficient in the sense that a single ACK can acknowledge an entire sequence of requests. However, using such acknowledgments require that request ids be ordered, which is not assumed by FTS, primarily because FTS is designed to make use of FT-CORBA’s request IDs if available, which guarantee only request id uniqueness but not order.

SACK TCP [68] is a standard extension of basic TCP that also supports SACKs in addition to the standard ACKs. The added SACKs allow better bandwidth utilization by allowing the receiver to inform the server that it has received non-continuous segments of the data stream. As a result, only the missing segments are re-transmitted, as opposed to re-transmitting the entire range that starts with the first missing segment in the stream, which is required when using ACKs. Additionally, the sending buffer can discard the acknowledged requests, and the freed space can be used to accept more data to send. In FTS, SACKs serve a similar purpose of speeding up client request removal from the cache, prior to the time-out, thereby reducing the cache size.
Chapter 3

Introducing FTS: A Fault-Tolerance Service

In this chapter we outline FTS, our CORBA Fault-Tolerance Service. FTS is the tool through which we implement and test our ideas of introducing fault-tolerance into middlewares, efficiently and without breaking CORBA compliance. FTS, in its current development status, provides automatically-managed active replication of server objects using an underlying GC system. FTS is highly portable and interoperable. It is currently implemented in Java and deployed on Linux, but depends on neither. Furthermore, the architectural concepts behind FTS are not CORBA-specific and can be ported to other middlewares just as well, as we explain in Chapter 8.

Our introduction of FTS in this chapter begins with the design motivation, i.e., a schematic list of qualities we strove to achieve in FTS in our work. Next in this chapter we present a description of FTS and its primary components. Chapter 4, which follows next, completes a full description of FTS by presenting the main algorithms and protocols involved in FTS operation, as well as discussions on some related aspects. As previously noted, a more detailed technical description of FTS can be found in [45].

3.1 Design Guidelines

3.1.1 Fault-Tolerance and Survivability

As a fundamental design goal, FTS is designed to provide fault-tolerance. In its current design, FTS handles benign failures, i.e., server crashes and communication failures, as well as value faults, i.e., object failures that generate erratic replies. Furthermore,
deploying FTS on a heterogeneous mixture of clients and servers residing on different platforms, and detecting value faults, results in a simplified form of N-version programming [66] that can reduce the chance of failures resulting from widely-spread software bugs (correlated failures).

Specifically, FTS can handle network partitions, whether these are real partitions or virtual partitions caused by false failure detection [16]. In particular, it is known that when the network is loaded, virtual partitions do occur. In order to avoid split brain behavior, FTS enforces a primary component model of operation [23]. This guarantees that updates only occur in one partition, but isolated partitions remain alive until they can rejoin the other replicas.

The architecture of FTS is completely decentralized and symmetric, guaranteeing no single point of failure. FTS guarantees that as long as at least one server of the cluster\(^1\) that served clients is alive, the objects FTS replicates will remain alive and virtually consistent\(^2\).

FTS further preserves service accessibility to clients through redirection. Existing client-server interactions are not disrupted unless one of the parties (client, server or communication) actually fails or disconnects from a primary component. Also, if a client tries to rebind to an object and there are no network partitions, the client will eventually get a reference to a live server. In the presence of network partitions, rebinding is restricted to at most one partition.

FTS, by design, is almost entirely self-reliant, so that it can be used to replicate any other basic middleware service without dependency loops. Other than its cluster, an FTS server depends only upon a local CORBA interface repository (IR) [87], which is a simple local database attached to each server, and the LMR sub-service (see Section 4.6), which is an internal part of FTS.

### 3.1.2 Portability and Interoperability

Portability is a key objective of FTS design. As a generic replication solution, FTS aims not to depend on any particular platform that combines a programming language, an operating system, an ORB and a GC toolkit. To that end, FTS design features strict CORBA compliance and a minimal dependency on common GC properties. On

\(^1\)There are two distinct group-related terms in FTS. The first is a GC process group that is used for replication and the second is a group of objects that are replicated together using the same process group. For semantic clarity, we call the first group a cluster and the second a replication context group.

\(^2\)FTS can also survive complete service failures by logging objects’ state to stable storage, at the cost of lower performance.
a higher level, the architecture of FTS can also be ported to other middlewares as well, as discussed in Section 8.3.1.

FTS design further strives for interoperability, meaning that FTS clients and servers can inter-mix with non-replicated CORBA components without additional mediation. Specifically, non-FTS clients can freely access FTS servers without risking replica consistency, although they lack the benefits of a fault-tolerant service through fail-over. Moreover, FTS is capable of replicating the same object between different ORBs, platforms and programming languages, as long as the same GC toolkit is used\(^3\).

As a third related property, FTS features application transparency. That is, FTS can conceal much of its functionality behind a standard interface, so that the application code is mostly unaware of the replication. This facilitates porting non-replicated applications into FTS, as it minimizes the cost of design and code modifications.

Note that application transparency is not always required. Many complex fault-tolerant applications may require the opposite, that is to be specifically involved in various aspects of replication, such as failure-detection, deployment decisions, etc. In those cases, FTS defines additional interfaces that can be invoked by the application in order to receive failure/view-change notifications, control failure-detection, and so forth. Originally, FTS began with a minimalistic approach of automatic active replication, and this direction of increased application involvement is under constant development.

Another FTS limitation of application transparency pertains to low-level services, such as transactions, security and life-cycle (object operations of creation, destruction, copy and move). While these services seemingly address unrelated aspects of distributed applications, they all require the cooperation of FTS since it controls access to objects and request execution scheduling. Currently, FTS provides only support for dynamic creation and destruction of objects, and support for other services is planned for future versions, as detailed in Chapter 8.

### 3.1.3 Performance and Scalability

FTS design aims for good performance and scalability through a number of specific goals. First, it promotes low invocation overhead, compared to a non-replicated object. To this end, FTS features efficient propagation of data to replicas (using a dedicated GC ABCAST transport) and support for sequential consistency (in addition to the default linearizable consistency) that enables localized computation of queries at each replica.

\(^3\)Using a middleware-based GC implementation, such as OGS [31] or NewTop [72] would make FTS completely portable, depending solely upon the middleware in use
Also within the context of performance and responsiveness, FTS design pursues *quick recovery* of service from failures. The basic strategy is active replication that allows clients to quickly rebind to an alternate object replica during fail-over\(^4\). On top of that, FTS tries to reduce the number of view-changes that require a state-transfer round so that the service can be resumed swiftly following a view-change.

Another related important goal in FTS design is *scalability*, which is manifested in efficient resource utilization. By replicating multiple objects together using the same cluster (see Figure 3.1), the per-object resource overhead is reduced by a factor of the number of objects. Thus, FTS can scale better in the number of replicated objects. Similarly, distributing the client load over all the servers in a replication cluster reduces per-server work-loads of managing client-server TCP connections [71] and propagating client requests to other servers. Furthermore, local query processing utilizes server multiplicity in clusters to the point that it achieves speedups compared to non-replicated services (see Chapter 5).

Last, the additional results of adaptive batching and selective acknowledgments (SACKs) also contribute to scalability. Adaptive batching increases network and GC utilization and consequently boosts FTS throughput. The SACK mechanism reduces overall memory consumption in FTS servers, for increased memory utilization.

\(^{4}\)Recall that due to the interoperability requirement, FTS does not permit broadcast operations in client-server communication.
3.1.4 Simplicity

Simplicity of layout and configuration is important for a large software system such as FTS, since it makes it easier to understand and maintain. FTS features a simple structure that is decentralized and symmetric. Furthermore, FTS architecture consists of only four major components, each with a clear role assigned to it (see Figure 3.2). Last, for easy administration, FTS operation can be completely controlled from a couple of external configuration files that are copied to each client and server.

Application interaction in FTS is also simple, resulting from FTS transparency. At the basic level, no additional interaction is required compared to the non-replicated case, i.e., register the application objects at the broker. More advanced replication involvement requires additional interfaces, all of which are presented to the application as if they are provided by the broker itself.

3.2 FTS: Top-Level Description

A top-level view of the FTS infrastructure is presented in Figure 3.2. Each of the server application’s objects belongs to a replication context group. Each such group is identified by a unique context assigned by the application designer. All the objects in the same replication context are replicated together, i.e., their replicas are part of their context group’s replicas. Each replica of the context group is managed by a single Group Object Adaptor (GOA), which encapsulates all functions related to routing client requests, managing the replication of its registered objects and maintaining replica consistency. Each server can participate in replication of multiple contexts, by installing their respective GOAs and replicas. FTS imposes no restrictions on how to divide the objects of a specific application into replication contexts. However, it is possible that a service administrator would add meaning to such a division, which would match actual deployment of the servers, e.g., sub-trees of a naming service directory.

In FTS, a client communicates through standard CORBA with a single object replica. The GOA of the replica’s context is responsible for propagating the request to the other replicas of the same object as following. If the invoked method is defined as a query, then it is executed locally and the reply is returned to the client, just as in a standard CORBA server. On the other hand, if the method invoked is defined as an update, i.e., may change the object’s state, then the GOA captures the request and sends it through the underlying GC subsystem’s ABCAST transport to the rest of the replicas. Update requests are executed at all the replicas only after being delivered from the ABCAST transport, thereby ensuring state consistency with the replica that was accessed by the client. In addition, update requests are cached at each server along with their replies.
(see Section 4.8), so that a client re-issuing the same request will receive a cached reply instead of causing re-execution of the same request.

When a client’s communication with a server fails, the FTS client request interceptor detects this failure, and locates, using the LMR sub-service (see Section 4.6), an alternate server to be invoked. The interceptor then transparently redirects the request to the new server. The process may repeat, following consequent failures, until either the request completes successfully or a pre-configured timeout for client requests expires, in which case an exception is returned to the client.

The GOA is integrated with the underlying GC system through a generic Group Communication Interface (GCI), which enables the GOA to use most existing GC toolkits. Through the GCI, the GOA is aware of the view of the replication cluster. When the view changes, the GOAs in the cluster are notified, and, consequently, perform a state transfer, during which the replication context group’s state, including the replicated objects’ states and cached requests, is passed to the new servers joining the cluster (see Section 4.7). Also, network partition may cause a cluster to split, in which case one surviving component may continue to service clients (see Section 4.6).

We now provide more detailed descriptions of the key components of FTS architecture: the GOA, the GC subsystem interface, the interceptors and the interface with object replicas. Each component’s description contains references to the specific FTS activities the component is involved in, which appear in the Chapter 4.

### 3.2.1 Group Object Adaptor

The GOA is the core of the FTS infrastructure. It is a nexus that connects the application objects, the middleware broker (i.e., the CORBA ORB) and the GC subsystem. Its functions include:
1. The GOA is an object adaptor. Thus, once created, it is used for registering, (de)activating and managing information about CORBA objects. Also, as an object adaptor, it is responsible for injecting the FTOID component (Section 4.2) into the IORs of registered objects. As an object adaptor, the GOA also serves as a single access point of the application code to FTS functions. The basic interface of the GOA resembles that of the standard CORBA POA, in order to simplify porting non-replicated applications into FTS. However, the GOA also includes extended API for dynamic object management (see Section 4.9) and increased application involvement, such as joining and leaving the GC cluster, monitoring GC cluster view and suspecting members.

2. The GOA also serves as a high-level invocation interceptor. By intercepting the link between the ORB and the application objects, it has full access to request contents and uses it to relay the request contents to the GC ABCAST transport. Furthermore, the GOA can also efficiently re-route client requests, that is, direct a request to an object that is different than the request’s target\(^5\). The GOA uses the re-routing capability for all of its operations. Request caching (see Section 4.8) is first enforced by having requests re-directed to a special cache servant instead of the original target when a copy of the request is found in the cache. Then, during request processing (see Section 4.4), updates are re-routed to a special multicast servant that invokes the ABCAST transport prior to execution, whereas queries are directed to their original targets (pending age synchronization).

3. The GOA serves as a manager for a single replica of the replication context. As such, it manages the membership of the server in a GC cluster for the GOA’s replication context. This means that the GOA responds to all GC events of view-change and message delivery, handles state-transfer (see Section 4.7) and executes optional high-level functions of fault-detection and safe delivery (see Section 4.3). Furthermore, the GOA can be appointed as a coordinator of a GC cluster view, in which case it also conducts the service acquisition and the implied LMR communication (see Section 4.6).

4. The GOA functions as a generic factory of objects. This ability is derived from FTS’s support for dynamic creation of objects, as described in Section 4.9. During state-transfer, when the GOA has the consumer role during state-transfer (see Section 4.7), it initiates both creation and destruction of objects.

Currently, there is no CORBA standard for building proprietary object adaptors in a portable fashion. Still, the GOA is portably constructed over any CORBA-compliant

\(^5\)Recall (from Section 1.4) that the standard CORBA PI can also re-route requests, but at a much higher cost of returning to the client and re-issuing the request again.
ORB in the following manner. The GOA wraps POA methods and relays them to real POAs that are encapsulated inside it (see below). This gives the GOA the ability to intercept downcalls from application objects to the POA. In addition, the GOA needs to intercept upcalls, that is, request invocations that are relayed from the ORB through the GOA’s internal POAs to application objects. To accomplish that, the GOA configures the internal POA to direct the requests that go through it to application objects by using a custom CORBA-compliant servant locator that is provided by the GOA. This makes the GOA capable of controlling request routing, as discussed above. Also, updates that need to be replicated are directed by the servant locator to a generic DSI servant. The DSI mechanism provides the servant full access to the request contents, so the request can be replicated. Thus, the entire construction of the GOA uses only standard CORBA facilities. Specifically, the flexible configuration of the standard CORBA POA is what gives the GOA the ability to intercept and re-route requests.

Note, however, that the object-adaptor-based construction does not limit the implementation of FTS architecture only to CORBA. Alternate middlewares, such as .NET, offer a seemingly much simpler implementation, where the entire logic of the interception and replication can be put inside standard .NET interceptors. This is further discussed in Chapter 8.
Figure 3.3 contains a schematic description of the structure of the GOA. The main components of the GOA are listed below. The interested reader may find a more detailed description of these components and others in [45].

**Outer POA:** This is the POA that is used for interaction with clients, as explained above. The object references that are published to clients are registered on this POA. Also, this POA is also used for inserting the FTOID component into the published IORs (see Section 4.2). The outer POA has no object table that maps requests to objects, but uses the outer servant locator (see below) of the containing GOA. During view change, the outer POA holds incoming client requests, and resumes service afterwards.

**Outer Servant Locator:** This component is associated with the outer POA and controls the routing of requests to servants, thus enabling request replication and caching through special-purpose servants, as previously discussed.

**Inner POA:** This is a default-configuration permanently-active POA. Application objects are registered in this POA as well as in the outer POA, but the IORs generated at this POA are only accessible from within the GOA. The inner POA’s IORs are used for executing updates that are delivered from the ABCAST transport, so that the target object replicas are invoked for updates in the same way as if they were unreplicated.

**Replica Manager:** This component coordinates all the functionality of the GOA that is related with replication management. It interfaces with the GCI and handles view-change, including state-transfer. Additionally, it is involved in update processing (delivering ABCAST updates), fault-detection and safe delivery notifications.

**Group Manager:** A special component inside the replica manager that is used only when the GOA is selected as a coordinator. In such a case, the group manager is used for registering information for all the members of the view and monitoring the progress of the view-change protocol, including age computation, service acquisition and state-transfer.

**Quorum Client:** The quorum client is responsible for interacting with LMR servers during service acquisition (see Section 4.6), when the GOA functions as a view coordinator. It accesses LMR servers in order to obtain a quorum of leases. Then, it automatically refreshes every lease it acquires until it is ordered to abort by the replica manager.

**Invocation Logic:** The invocation logic is consulted with by the outer servant locator in order to decide whether an incoming client request is either an update or
a query. The invocation logic reaches a decision by checking the request’s operation signature using configuration-based tagging information and CORBA IDL definitions of the application (obtained from the interface repository).

**Servant Registry:** The servant registry is used for managing the servants (application objects) registered with the GOA. Specifically, it manages the age information that is associated with each object. The servant registry is also responsible for reading and writing the state of the replication context during state-transfer, as described in Section 4.7.

**Cache:** This is the request/reply cache that is used for maintaining at-most-once execution semantics, as described in Section 4.8. Aside from being a store of request objects, the cache also contains a special DSI servant that returns cached replies when an incoming request is found in the cache.

**Multicast Servant:** A DSI servant used for sending requests in ABCAST messages and then waiting for them to be delivered back (optionally also notified as safe) and complete execution, after which it sends the generated reply back to the client (see Section 4.4).

**Dispatcher:** The dispatcher sends updates that are delivered from the ABCAST transport for execution at their respective target servants, following the consistency granularity level specified in FTS configuration (see Section 4.4).

### 3.2.2 Interfacing With GC

The *Group Communication Interface* (GCI) mediates between the GOA and the GC toolkit, thus enabling the GOA use of generic GC services. FTS relies on properties of GC which are common to many available toolkits (according to [23]). The FTS design aims to rely upon as few properties as possible to increase its portability over GC toolkits. Specifically, several of the properties discussed below are optional, and can be replaced by FTS’s own alternate solutions.

The GCI operation is bi-directional. As a wrapper of the GC subsystem, it relays commands handed down from the GOA to the GC subsystem, such as joining/leaving a cluster and sending messages. As a wrapper of a client of the GC services, it relays GC events such as end of view, new view (view change) and delivery of messages to the upper GOA level that implements a GCI.*Callbacks* interface for this purpose. Additionally, the GCI layer includes abstractions for messages (GCI.*Message*) and membership identity (GCI.*Member*). A full description of the API of the GCI layer is provided in [15].

A basic property that FTS requires of the GC toolkit is the ordering of all message-related GC operations, both upwards and downwards, into a sequence of views that are
indicated by view change events. That is, every message send and receive occur within the context of a view. Installed views further include agreed ordering of members (ranks), out of which a coordinator can be determined. Also, each new view notification from the GC is preceded by an end-of-view (block [23]) notification indicating the forthcoming end of the current view. The GOA uses the end-of-view notification to hold non-executing request processing, which is required for achieving coherence (see Section 4.7).

On top of views, the virtual synchrony [23, 16] property guarantees a correlation between the sequence of views installed at each server and the history of changes that its replica reflects. FTS exploits this correlation to correctly compare the degree of state update between replicas (see Section 4.7). Virtual synchrony also implies self-delivery of ABCAST updates, which is required for executing updates received from clients in consistency with the global order. View synchrony [16] (also known as precise membership [23]), which is also implied from virtual synchrony, requires that the membership at each GC component accurately reflects member reachability changes that results from network partitions, member crash-failures, etc.

For replica consistency, the GC is required to provide an ABCAST transport. This transport is needed for the request processing protocol of FTS (see Section 4.4). Optionally, the GC may also provide a reliable unicast (point-to-point) transport as a lighter form of communication that can be used for sending information from members to the coordinator during the view-change protocol (see [45]). In the lack of such a facility, FTS can use either a simple TCP/IP connection, a CORBA invocation or the ABCAST transport, for the price of possibly reduced efficiency.

Safe delivery notifications are also used by FTS in order to commit the effect of updates upon which replies are returned to clients (see Section 4.4). However, GC support for safe delivery is not mandatory. FTS can provide safe notifications by itself by having each GOA broadcast acknowledgments of updates that were either delivered or executed. When a GOA receives notifications for a request from all the members of the view, it tags the request as safe. Acknowledgment of delivered updates makes for faster safe delivery notifications than acknowledgments of executed updates. On the other hand, an execution acknowledgment can be accompanied with the actual reply, in which case voting can be conducted to detect value-faults (see Section 4.3).

Currently, GCI is defined in Java. It has a back-end implementation for the Ensemble GC toolkit [46, 1], version 2.01. However, since the GCI’s model of operation is common to GC toolkits in general, we believe it can be ported to other toolkits just as well.
3.2.3 FTS Interceptors

FTS employs standard CORBA Portable Interceptors (PIs) [96] for several tasks that can be handled in a portable and application-transparent fashion in CORBA only by components of this type. The first task is exchanging FTS service contexts between clients and servers. These service contexts contain the request id (see Section 4.8), the age values for the request processing protocol (see Section 4.4) and selective acknowledgments (SACKs) used for efficient cache size reduction (see Chapter 7).

The second task is redirecting clients upon fail-over. This task is handled by the client-side PI. When an invocation results in a fail-over condition (see Section 4.3), the client PI is notified of the matching exception. After verifying it is indeed a fail-over, the client PI obtains an alternate object reference by contacting a quorum of LMRs, as described in Section 4.4. Then, the PI causes the client to redirect its invocation at the alternate reference by throwing a CORBA LOCATION_FORWARD exception.

A third optional task, also handled by the client PI, is the installation and operation of a transparent generic (DSI) proxy at the client side, as suggested in [38]. This task basically requires that all the client requests are directed at the proxy, which performs some functions and then forwards the requests to their original respective target objects. Functions performed by the proxy include portable generation of a unique request id, if such an id is not already provided by the ORB (see Section 4.8). Another function which is planned to be added soon is de-coupling invocations and replies for improved server performance, as suggested in Chapter 5.

FTS interceptors restrict their operation only to requests targeted at FTS objects, by checking the FTOID component of the IOR (see Section 4.8) before attempting any task.

3.2.4 Object Replica

In order for an application object to be replicated in FTS, each of its replicas must be able to perform certain functions, which are defined in the interface StatefulObject, detailed in [45]. The first and foremost of these functions are the methods of set_state() and get_state() for setting and retrieving the object replica’s state, which are required for state transfer (see Section 4.7). The rest of the methods that an object replica needs to implement are optional, depending on the application’s requirements of FTS.

When an object can be dynamically deleted (see Section 4.9), each of its replicas needs to implement the notify_st_unregistration() method, which is used by the GOA to request each replica to remove itself, either implicitly during state-transfer or explicitly upon automatic deletion requested by the application.
When object-level fault-detection is used in FTS (see Section 4.3), an object replica may need to implement the `is_alive()` method. The method implementation can be a watchdog-style liveness check for threads inside the replica’s implementation, and/or self-diagnostic code that verifies continued correctness of the object operation, such as CRC check, invariant calculation, etc.
Chapter 4

Operational View of FTS

In this chapter we review principles, algorithms and protocols used by FTS for its operation. This chapter complements Chapter 3 as it describes interactions between FTS component. In addition, this chapter provides discussions of various aspects of FTS operation.

4.1 Replication Model

FTS features context-based replication of application objects. This means that replication is applied to non-overlapping groups of server objects, rather than replicating each object separately. Each group is named by a unique context and is formed by creating a GOA with the desired context as its name. Server objects are assigned to a context by registering at that context’s GOA. Finally, the GOA forms a GC endpoint and joins the context’s GC cluster on behalf of the server as part of activating the replicated objects. The allocation of replication contexts and the division of objects to replication context groups is handled by the application designer for maximum flexibility.

Every server participating in the replication of a specific context holds replicas of all the objects that are contained in that context’s group, as described in Figure 3.1. By having replicas of different objects of the same service share resources such as replication management, group memberships and multicast and state transfer operations, the per-object replication resources, in terms of memory, CPU and messages, are reduced by a factor of the number of objects supported by the group. A further discussion of the benefits of context-based replication is presented in Section 4.10.1.

Replication context groups are heterogeneous in nature, i.e., objects of different types can share the same group. Also, a single FTS-based server process can replicate multi-
ple contexts of the same service, in accordance with the CORBA principle that allows multiple object-adoptors to be installed on the same ORB. On a broader scale, a single server machine can support multiple server applications. A server can join/leave the GC cluster that replicates a context dynamically (by calling the respective GOA methods) or crash-fail. However, a context's GC cluster is assumed to never be left without members all at once. At least one server is supposed to survive at one network partition until other servers may join and synchronize with it in order to guarantee continuous existence of the replicated objects. There are higher-level monitoring facilities currently being developed within the FTS project in order to ensure a minimum cluster size for each context by launching additional servers and having them join the context's GC cluster in a network partition.

4.2 Associating Objects and Replicas

FTS utilizes the standard CORBA object reference, namely the IOR, for client-server interaction without any structural or semantic changes. Accordingly, FTS requires that a client communicates with only one replica of an object, rather than the entire cluster. However, in order to take advantage of object replication, i.e., to enable the fail-over of clients from one replica to another, there needs to be some sort of identification that binds together replicas of the same object. FTS implements this common identification through the Fault-Tolerant Object Identifier (FTOID), which is a component (a record) of information that is embedded in the IOR of each replica. All the IOIs of the replicas of the same object have the same FTOID component.

The FTOID of each replicated object consists of three string fields. The first is a globally-unique service name, which identifies an instance of a replicated application inside a deployment domain of FTS. The second is a service-unique context name, identifying the replication context to which the object belongs. The combination of the first and second field is also used as the name of the GOA at which the replicas are registered. The third field is the object id, which further identifies the object inside the replication context. The object id is a standard CORBA identifier that is assigned to an object whenever it gets registered in a POA. FTS generates object ids by itself and assigns them to objects using a standard registration technique, rather than let each ORB automatically generate object ids on its own. The reason is that different ORBs (even different instances of the same product) might automatically generate different object ids, for example when joining the service at different times. The state of the generator used for creating object ids is also part of the replication state that is passed to new members during state-transfer, in order to ensure identical generation of new object ids.
The FTOID is placed inside a replica’s IOR using the \textit{IOR Interceptor} facility of the CORBA PI specification [96]. Each new IOR, when being formed by registering a CORBA object at a standard CORBA POA, can have custom components embedded in it using an IOR interceptor that is associated with the POA. The POA and the IOR interceptor are both encapsulated inside the GOA of FTS. The technicalities involved in adding the FTOID component to a replica’s IOR are explained in [45].

The FTOID serves as a high-level persistent reference for a replicated object. This means that when a client request arrives at an FTS server, the request is eventually directed to the target replica hosted at the server based only upon the FTOID value. The standard IOR data, including server, port and object key is used only for directing the request to the server and GOA.

When a client PI detects a fail-over condition, it turns to LMRs for redirection (see Section 4.6). Each LMR extracts the FTOID component from the failed IOR. Using the service and context fields, it locates the view that is associated with the context and constructs an alternate IOR using an alternate server’s address and the FTOID. The alternate IOR is then returned to the client so that it can resume being served.

\subsection{4.3 Fault-Detection}

In order to detect a wide rage of faults, FTS relies on several mechanisms, including the GC subsystem, the middleware broker (i.e., CORBA ORB) and FTS logic itself. The GC subsystem is used to detect changes in the service configuration (see Section 4.6) including server crash-failures, disconnections, join and leave operations. All of these faults are translated into view change events that are relayed to FTS through the GCI layer.

The CORBA \textit{General Inter-ORB Protocol} (GIOP) [87] defines a high-level abstraction of a reliable point-to-point communication transport. Among others, it defines several exceptions that are raised at the client side (the method invoker) to indicate a possible failure in reaching or communicating with the target object on the server side (the invokee). Each received exception includes a completion status of “yes”, “no” and “maybe” to indicate that the request did manage to execute, did not execute or could have executed, respectively. According to these, FT-CORBA [90] defines a set of \textit{fail-over conditions}, which are every combination of a failure exception with a completion status that is not “yes”. Under each of these conditions, a client may consider its request possibly not-executed and thus retry it\footnote{The “maybe” completion status is permitted since server-side caching or logging is supposed to prevent re-execution of requests to maintain at-most-once semantics.}. FTS relies on these conditions for
clients to perform fail-over between servers and for both servers and clients to detect communication failures with LMR servers.

Each GOA, being a member of a GC cluster on behalf of the server process in which it resides, provides a default process-level failure detection through the GC membership service. FTS allows finer failure detection at object granularity through high-level (above-ORB) operations. First, objects that can crash-fail independently of the process, such as 3rd-tier database components, can have their failures detected earlier than the next client request by having the GOA periodically invoke an is alive() method of the object. Second, FTS can run a voting mechanism [31, 77] on methods tagged as updates in order to detect value faults, which are Byzantine failures that are detectable through erratic replies and are restricted to the boundaries of an object or a replication context (in case of inter-dependent objects). The update request is executed at all the servers, which then propagate their replies back to the server that needs to return the reply to the client. That server delays the reply until it collects all the replies (in fact, this is implemented as a variant of safe delivery. See Section 3.2.2). FTS assumes that less than half the replicas of any given object/context may value-fail, so the majority of the replies should always be identical. The replica whose replies are different from those of the majority, as well as those replicas who fail the is alive() check, are reported as suspected to the GC subsystem which in turn excludes them from the next view. By default each suspected server eventually tries to re-join the cluster, which would result in resetting the failed context state during state-transfer.

4.4 Replica Consistency

FTS provides support for multiple modes of consistency between replicas that can be matched to application requirements. These modes differ in the type of consistency provided (linearizable or sequential) and the granularity at which consistency is provided (context or object).

In order to guarantee these consistency modes, FTS requires that the GC provides services of an ABCAST transport that preserves FIFO (and thus causality), and a safe delivery of messages. Following [23], the assumed message service model is that each message is delivered back from the ABCAST service, followed by a later notification that this message is safe, meaning it was delivered at all the members of the current view of the GOA’s GC cluster.

FTS supports a unified protocol for providing all modes of consistency. This protocol relies on a configuration-based tagging of all the service interface methods as either queries or updates. A query is a method that computes a result without modifying the
4.4.1 Protocol Description

The following Figures present a pseudo-code description of the request processing protocol in both clients and servers of FTS. Figure 4.1 describes data associated with each request and object/context at both the clients and servers. Figure 4.2 describes the client part and Figure 4.3 describes the server part. Note that request processing at the server is multi-threaded, so when an event handler code goes into waiting as a result of invoking a lock or waiting on a flag, other events may still occur. Also, note that reading and writing data fields of a request occurs either at the client or at the server, and pertain to the stored copy of the request at that respective location. Last, note that the same protocol is applied at both object- and context-level of granularity. The difference is that at the object granularity each object has its own record of data both at the clients and the servers. At context granularity, all objects of the same context hold pointers to a single common record of associated data.

4.4.2 Properties of the Protocol

This section provides a characterization of the consistency properties of the algorithm. For the purpose of this section, the reader may assume that there are no cascading invocations, that is, every object, when executing a request from a client, does not further invoke other objects. For handling cascading invocations, see Section 4.5.

Furthermore, the request processing protocol is augmented with safe delivery notifications that can be ignored for the sake of clarity, i.e., the reader may simply assume that no view changes take place and that updates and queries do not wait for safe notifications. Safe delivery (in conjunction with state-transfer) can be considered as an orthogonal extension that adds the durability of modifications in the presence of failures and disconnections (which trigger view-changes). That is to say, it guarantees that if a client receives a reply that acknowledges the execution of its request at a server, then the state of the service in the end of the request execution is committed forever\(^2\). This is achieved by ensuring uniform delivery, that is, that all the replicas of the target object’s context also deliver and execute that request. Thus, even if only some of the replicas later get to serve clients in another view (see Section 4.6), the state that they carry and

\(^2\)In combination with Section 4.6, the durability of a safe update depends upon at least one server surviving from the current primary component to the next.
Server data:

For each object/context:

- last_up: Request // points to the last update request applied at this object/context (initially NULL)
- lock: Read-Write Lock // synchronize between queries and updates
- age: Counter // counts updates at this object/context (initially 0)

Client data:

For each target object/context:

- age: Counter // last-seen target age according to replies (initially 0)

Per-request data:

- safe(*): Boolean // is this request safe (for updates)
- last_up(*): Request // points to last update applied to target object/context before this request (for queries)
- req_age: Counter // last-seen target age by client when sent
- rep_age: Counter // age of the object/context when reply is made - this field is attached to reply

(*) used only at the server in the cached copy of the request

Figure 4.1: Request Processing Protocol - Data Definitions

פרוטוקול עיבוד בקשות - המרחב נתונים

Client operation:

Upon issuing a new request r at target object/context o:

\[ r.\text{req\_age} := o.\text{age} \]

Upon receiving a reply for request r at target object/context o:

\[ o.\text{age} := r.\text{rep\_age} \]

Figure 4.2: Request Processing Protocol - Client Part

פרוטוקול עיבוד בקשות - רכיב הלקוח

56
Server operation:

After every view change (optional state-transfer):

For each object/context o:

- o.last_up := NULL

For each request r:

- r.safe := true
  
- resume and return every request that is still waiting for r.safe
  
- Abort all remaining waiting requests with fail-over exception (causes client to retry)

Upon receiving a new update request u from a client:

- store u in cache
- send u through ABCAST transport

Upon delivering update u from ABCAST:

- store u in cache (if not there already)
- send u to dispatcher for execution

When delivered safe notification for update u:

- u.safe := true

When update u is dispatched for execution at target object/context o:

- o.lock; write_lock()
- execute u
- o.last_up := u
- o.age++
- u.rep_age = o.age
- o.lock; unlock()
  
- if this server received u from client then
  
- wait until u.safe
  
- return u's reply to client

Upon receiving a new query request q from a client with target object/context o:

- wait until q.rep_age ≤ o.age
- o.lock; read_lock()
- execute q
- q.rep_age := o.age
- q.last_up := o.last_up
- o.lock; unlock()
  
- if q.last_up ≠ NULL then
  
- wait until q.last_up.safe
  
- return q's reply to client

Figure 4.3: Request Processing Protocol - Server Part

פרוטוקול עיבוד פלטוטות - רכיב השרת
that will eventually be presented to clients through replies to requests, will reflect the impact of that request.

By default, when no configuration-based tagging is applied, all methods are tagged as updates. Assuming that requests are dispatched for execution one after the other at the same order they are delivered at each GOA, then the mechanism provides linearizable consistency of entire replication contexts, regardless of all the additional age-based mechanism. This is achieved by totally-ordering all the requests applied at all the replicas of a context [8].

The same protocol in FTS (without request tagging) can also provide linearizable consistency at object granularity, rather than at context granularity. This is achieved by changing the dispatcher of the GOA [45], which dispatches updates for execution after being delivered back from the ABCAST service. The default dispatcher is the sequential dispatcher, which dispatches a new update for execution only after its predecessor has completed executing, and at the same order the updates were delivered from the ABCAST transport. This ensures the context-granularity linearizable consistency. An alternate object-sequential dispatcher can be used instead, which allows concurrent dispatching of updates designated at different target objects, but still sequential execution of requests at the same target object, in accordance with the delivery order. Thus, object-granularity consistency is guaranteed.

Linearizable consistency is known to be local [47]. This means that by guaranteeing linearizable consistency at one granularity, it is also guaranteed at the alternate granularity. For this reason, context-granularity linearizable consistency may seem redundant and harmful, since it only reduces request execution parallelism and thus impedes performance. However, note that in order to guarantee linearizable consistency at object granularity, application objects are required to be independent of each other, that is, unable to influence each other’s request execution through shared state or message-passing. Inter-object dependency can occur, for example, by having two objects indirectly share application data. In case of inter-dependent objects, FTS can still guarantee linearizable consistency for such objects by replicating them together in the same context using context-granularity linearizable consistency, thus totally-ordering the requests to all the objects together.

When request tagging is applied, the same request processing protocol provides sequential consistency either of separate objects or of entire replication contexts, depending on the pair of <dispatcher, age-granularity> being used. A summary of all the consistency modes is provided in Table 4.1.

The tagging-based request processing protocol is essentially a variant of the classic local read algorithm as presented, e.g., in [8]. The FTS version of this algorithm adds adaptations for operating on top of a GC cluster, which means handling view change,
<table>
<thead>
<tr>
<th>Granularity</th>
<th>Linearizable Consistency</th>
<th>Sequential Consistency</th>
</tr>
</thead>
<tbody>
<tr>
<td>Object</td>
<td>Object-Sequential Dispatcher</td>
<td>Object-Sequential Dispatcher</td>
</tr>
<tr>
<td></td>
<td>No tagging</td>
<td>Per-object Age</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Tagging</td>
</tr>
<tr>
<td>Context</td>
<td>Sequential Dispatcher</td>
<td>Sequential Dispatcher</td>
</tr>
<tr>
<td></td>
<td>No Tagging</td>
<td>Per-context Age</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Tagging</td>
</tr>
</tbody>
</table>

Table 4.1: Summary of Consistency Modes

 пояשו פקביות

state-transfer and safe delivery.

We now provide a more accurate classification of the consistency guaranteed by FTS when using request tagging. According to [21, 116], the following properties hold for sequential consistency:

**Update Propagation:** requires that each update to an object’s state is eventually received by all the replicas of that object.

**Read Your Writes:** requires that the effects of every update made by a client are visible to all subsequent queries of that client to the updated object.

**Monotonic Reads:** requires that the effects of every update seen by a client query are visible to all subsequent queries of this client (unless overwritten by later updates).

**Monotonic Writes (FIFO):** requires that all consequent updates requested by the same client are applied in the same order as they were issued.

**Writes Follow Reads:** requires that the updates whose effects are seen by a client query are applied before all subsequent updates requested by that client.

**Total Ordering:** requires that all the updates are applied in the same order in all the replicas.

The request processing protocol, when with request tagging, guarantees all the above properties, thus ensuring sequential consistency. The update propagation and total ordering properties are guaranteed by the ABCAST service. The monotonic writes property is guaranteed by having each write performed before its matching reply is returned to the client (synchronous operation). Furthermore, the read/write lock and the ABCAST protocol also guarantee the writes-follows-reads property, since each query
reply reflects a state that results from a sequence of updates that have already been delivered and executed. Thus, any consequent update from the same client would only be delivered and applied after those earlier updates.

The last two remaining properties of read-your-writes and monotonic reads are enforced by the age mechanism that makes each query execute only after the target object/context is at least as updated as the client has last seen it. Each object/context at the server has an age attached that indicates the number of updates applied to it. This age is advanced by one after each update that is executed at the object/context. Accordingly, each client PI stores, for each target object/context, the last-seen age of that object/context, updated by values that are attached to each reply of either a query or an update. Each new query issued by a client has the last-seen age value attached to it and waits until the target object/context reaches that age before executing.

The consistency provided by the FTS algorithm when using request tagging is still strictly weaker than linearizable consistency. Consider a simple example of two clients accessing each a separate replica of the same object. The first client performs an update and completes. The second client then performs a query that returns the state of the object as it was before the first client’s update. This is possible since the query was executed at the second replica before the update from the first client was delivered to it.

The best way to describe the consistency provided by the FTS algorithm is as a restrictive variant of sequential consistency. The restriction results from the fact that not all sequentially-consistent executions are permitted by this algorithm. Specifically, sequential consistency permits that if two clients issue updates back-to-back, i.e., the second client sends its update only after the first one is finished with his update, then the actual global order of update execution at the replicas can be the opposite of the issue order. This is not allowed in FTS where a return of an update indicates it was already executed.

When using request tagging, the consistency provided by FTS is not local, since it is weaker than linearizable consistency but no weaker than sequential consistency [120]. Thus, sequential consistency at granularity larger than that of a single object is currently provided only by using context-granularity3. This provides further justification for context-granularity consistency even if the objects inside the same context are independent of each other.

The sequential consistency modes allow that queries be non-deterministic, since they are executed in only one replica. However, updates are still expected to be deterministic. Specifically, FTS assumes that objects execute updates deterministically. This is left as a responsibility of the application. For example, an application object that is internally

---

3Future FTS design can incorporate extended mechanisms in order to guarantee sequential consistency at arbitrary granularities. See Chapter 8.
multi-threaded can have non-deterministic method executions, when no proper thread synchronization and control is applied. Other systems, such as Eternal [78] prefer to take control of the entire threading mechanism through OS-embedded hooks, for the price of solution portability.

Last, note that non-FTS clients, i.e., clients that do not have the FTS PI component that executes the client part of the request processing protocol, may still enjoy the same consistency mode as FTS clients when accessing a single replica, only without the fail-over capability. Updates issued by non-FTS clients undergo ABCAST and execution just as those of FTS clients, thereby not affecting the replica consistency as presented to FTS clients. Queries of non-FTS clients, on the other hand, have no age indications attached and are therefore executed as soon as possible\textsuperscript{4}. When working with one replica, this causes no problem, since the replica itself (either object or context) guarantees the read-your-write and monotonic-reads. Problems may arise at a non-FTS client that holds references to replicas of multiple inter-dependent objects that reside on different servers and belong to the same sequentially-consistent context. By controlling which object references are provided to non-FTS clients, such problems can be avoided.

4.5 Cascading Invocations

Handling cascading invocations is a necessity for any replication infrastructure since it allows supporting the classic n-tier architectures which are common to many modern applications. There are two issues to be dealt with when allowing replicated objects to further invoke other replicated objects during request processing in FTS. The first is how to implement the required communication between replicated contexts, and the second is what consistency can be guaranteed at the application level.

4.5.1 Inter-Context Communication

The issue of inter-context communication is fairly simple to implement in FTS, by adding each server the client-side FTS PI and optional transparent proxy for generating request IDs (see Section 4.8). The method for generating request ids is different, though, from that of a client-only component. Each replica needs to generate the same request id for the same cascading invocation in the sequence that is generated during processing the same client request, so that redundant invocations from multiple replicas can be suppressed at the invoked context to maintain at-most-once semantics (see Section 4.8). Still, the generated request id needs to be different for each cascading invocation, and

\textsuperscript{4}pending synchronization with the read/write lock of the object
different from that of the client request id. Therefore, the following scheme is employed. Each request arriving at the GOA of a context is assigned a counter, starting at 0. Each new cascading invocation advances the counter by 1, and the outgoing request carries the request id of the incoming request appended with the value of the counter.

Note that the current design for inter-context communication is inefficient in terms of amount of messages involved, which can be as worse as $O(m \cdot n)$ when the invoker object has $m$ replicas and the invitee has $n$ replicas. Still, in failure-free operations this would typically involve only $O(m + n)$ messages since the invitee’s published reference points to only one replica and redundant invocations are suppressed before generating broadcasts among the invitee’s replicas. Furthermore, future design modifications (see Chapter 8) are planned to improve this aspect considerably.

When a context-granularity consistency mode is applied, FTS employs an optimization for cascading invocations between objects of the same context, by treating them as application-level point-to-point invocations. This means that these requests are ignored by the request processing protocol, and are sent instead directly to the target servant. Consequently, no communication is generated as each replica of the invoker object interacts locally with the matching replica of the invitee object.

### 4.5.2 Global Consistency

When all the requests are tagged as updates (default when no explicit tagging is applied), then FTS guarantees global linearizable consistency, based on the locality property that requires guaranteeing this consistency at only one level of invocation.

Global sequential consistency, however, is currently not provided by FTS in a general sense, since sequential consistency is not local and maintaining it globally requires additional mechanisms that are currently reserved for future versions, as explained in Chapter 8. However, using the intra-context invocation optimization explained in the previous section, FTS can guarantee sequential consistency at context granularity while allowing cascading invocations inside the same context. For example, in order to achieve a sequentially-consistent 3-tier implementation, both the 2nd- and 3rd-tier objects would have to be replicated together in the same context using context-granularity sequential consistency.

Last, a note on non-determinism when used with cascading invocations. Client queries, being unable to modify state of objects, can be non-deterministic and involve cascading queries. The client eventually receives only one reply and thus is unaffected by possible inconsistent replies. However, if the non-determinism of a query execution can influence the sequence of cascaded invocations, then duplicate request suppression in the cache would be less effective, resulting in queries being unnecessarily executed more
than once. Also, all invocations that cascade from an object processing an update must be deterministic, so that no replica receives a reply that is different from one received by another replica. The future extension of the communication mechanism will also allow replicated objects to issue non-deterministic queries, as discussed in Chapter 8.

4.6 Dynamic Service Configuration

The FTS service configuration, i.e., the definition of the current GC cluster of servers that provide replication for a given replication context inside an application, is subject to change due to many factors. First of all, servers may crash-fail unexpectedly. Furthermore, the cluster may split due to network partitions or false failure suspicions at the GC level. Last, administrative operations that make servers join/leave a GC cluster explicitly change the service configuration. Thus, the general outlook of each GC cluster, assuming all above possible faults, is as a set of one or more disjoint components, i.e., sub-clusters that belong to the same replication context. FTS relies on the GC subsystem to detect the above events and consequently update the component layout. Still, there needs to be a mechanism that allows the service to guarantee the selected consistency mode despite the disjoint operation in multiple components.

Additionally, these same factors can also cause client-server communication to fail. Recall that each client knows only of a single replica of an object at each given time, in accordance with the standard CORBA communication model. As a result, a client may need information as to which server (replica) to fail-over to upon communication disruption, so that it can continue using the service.

FTS architecture contains an additional sub-service called Lease Manager / Redirector (LMR) whose purpose is to resolve both issues of replica consistency and client fail-over in a dynamically-changing service configuration. The general concept behind the LMR sub-service, as illustrated in Figure 4.4, is a combination of a global lock and a global directory, where the term "global" means that the relevant LMR functionality may be invoked from clients and servers in multiple network partitions. The global lock is used for enforcing a single primary component [23] to be elected each time the former primary component’s view changes. Also, each new primary component’s servers must be at least as updated (in terms of service state) as those of its predecessor. The global directory registers each primary component’s view and uses this information to redirect clients during fail-over.

In reality there is, of course, no single common LMR component that can be accessed from multiple disconnected network partitions. The LMR service essentially consists of a set of simple independent LMR servers. Still, the LMR service is fault-tolerant and
self-reliant. Each LMR server commits any update made to its state to persistent storage (e.g., a hard-disk) before acknowledging it. Thus, it can be easily restarted using any available watch-dog solution. Still, the LMR service, as a larger entity, further tolerates failures of one or more LMR servers. This is achieved by defining a quorum system [42] over the set of deployed LMR servers, known to both clients and servers. Each server or client that accesses the service, needs to get replies from a quorum of LMR servers before proceeding.

Each LMR server is a very simple CORBA server that exports two interfaces as described in Figure 4.5. The LeaseManager interface is accessed by servers whereas the Redirector interface is used by clients. As a LeaseManager, the LMR allows to obtain lock(), refresh (relock()) and abort(unlock()) a mutually-exclusive lease for a given context. Also, a lease holder can update view information that is used for redirecting clients. As a Redirector, the LMR receives an IOR of an object replica that a client fails to connect with and replies with an IOR of an alternate replica of the same object, which should be working, according to the current knowledge of the LMR.

To ensure propagation of service state from one primary component to the next, the LMR sub-service makes use of Primary View Identifiers (PVIDs). A PVID contains a Primary View Counter (PVC), which is a monotonically-increasing natural number that
Per-context data:

minPVC: Counter // indicates minimal PVC value required for a lease, initially 0
view: View // view of last registered primary component, initially NULL
lease: Lease // currently held lease – unique value for each lease create/refresh (NULL when not leased)

As LeaseManager:

Lease lock(context, interval, pvid)
if context.lease ≠ NULL and context.lease has not expired then
    throw LockedContext exception
    if pvid.pvc < context.minPVC then
        throw LowPVID exception
    context.lease := new Lease(interval)
    return context.lease

Lease relock(context, interval, lease)
if lease = NULL or lease ≠ context.lease then
    throw NotOwner exception
    if context.lease has expired then
        throw LeaseExpired exception
    context.lease := new Lease(interval)
    return context.lease

void unlock(context, lease)
if lease = NULL or lease ≠ context.lease then
    throw NotOwner exception
    context.lease := NULL
    return

void update(context, lease, pvid, view)
if lease = NULL or lease ≠ context.lease then
    throw NotOwner exception
    if context.lease has expired() then
        throw LeaseExpired exception
    context.MinPVC := pvid.pvc
    context.view := view
    return

As Redirector:

<IOR,pvid> redirect(IOR) // offer an alternate IOR for a failed one with pvid as version of view
    extract context from failed IOR (using the FTOID component)
    if context.MinPVC = 0 then
        return <NULL, 0> // no view information for redirection
    select alternate IOR from view information (may apply redirection policy e.g. load-balancing)
    return <alternate IOR, MinPVC>

Figure 4.5: LMR Server Operation

פונקציה שרת
is uniquely assigned to each new primary component and is known to all the servers that participate in that primary component. Intuitively, the PVID represents how up-to-date (concerning the service state) a primary component is.

Each LMR stores for each context a Minimum PVC (minPVC), which identifies the PVC of the last primary component that was granted a lease by this LMR. Each component, when requesting a lease in order to become primary, must prove that it is at least as up-to-date as the last primary component by providing the PVID with highest PVC known to any of its servers. If the PVC of that PVID is not lower than the context’s minPVC, a lease can be granted (pending mutual exclusion, of course). When the primary component is established, it updates the minimum PVC value of the LMR, thus guaranteeing that any future consequent primary component must include a member of the current primary partition and thus propagate the service state to it through state-transfer (see Section 4.7).

Figure 4.6 describes the server-part of the protocol, whose purpose is to establish a new primary component. This part is called service acquisition in FTS since the established primary component gets a mutually-exclusive right to serve clients. According to this protocol, the coordinator of each component first collects PVID information from all of the members in order to determine the PVID with the highest PVC, then uses it to invoke all the LMRs in order to acquire a quorum of leases. If the coordinator succeeds, i.e., receives a quorum of positive acknowledgments, then it proceeds to perform state-transfer. After the state-transfer, all the member servers compute the same new PVID, whose PVC is that of the previously-computed maximal PVID plus one, and acknowledge it to the coordinator, which in turn writes it along with the new view information to all the LMRs that provided the leases, and then signals all servers to begin processing client requests. Once the writing is acknowledged by the LMRs, the leasing is aborted, since it is not needed anymore. Any other concurrent component that would try to obtain a lease from a quorum of LMR servers would certainly fail with a LowPVID exception, since any quorum of LMR servers would contain a MinPVC requirement that is at least as high as the PVC of the just-committed new PVID, which is unknown to any server outside the new primary component

The coordinator will not proceed beyond requesting leases in one of three possibilities. The first is when the coordinator receives a negative reply of LowPVID from an LMR, indicating that this component cannot become sufficiently up-to-date in order to

---

5It may seem as if the service acquisition protocol implicitly assumes that all the servers of a dissolved primary component must cease to serve clients before any of them establish a new primary component, since such a scenario breaks mutual exclusion. However, note that the request processing protocol (Section 4.4) will not allow any server in the old (dissolved) primary component to acknowledge any new updates (or queries that reflect these updates), since none of these updates would be safe. Hence, replica consistency is guaranteed at all times.
Component Coordinator Operation – Service Acquisition:

Upon view-change/time-out:

- Compute max_pvid as the PVID with maximum PVC known to any server in the component’s view.
- Invoke lock(context, interval, max_pvid) for the component’s replication context on all LMRs.
- If received LowPVID exception then:
  - wait until next view-change to retry.
- If received a quorum of leases then:
  - initiate state-transfer.
    - after state-transfer, each member computes same new pvid with pvid,pvc := max_pvid,pvc + 1 and acknowledges the coordinator.
    - invoke update(context, lease, pvid, view) at all leasing LMRs.
    - invoke unlock(context, lease) at all leasing LMRs.
    - signal all members to open for service.
- otherwise (received only LMR communication errors and/or LockedContext exceptions):
  - wait for time-out to retry.

Upon lease renewal (while holding leases):

- renew leases by invoking relock(context, interval, lease) at all leasing LMRs.

Figure 4.6: Service Acquisition at Component Coordinator

רכישת שירות באמצעות מחסן רכיב
Client Redirection:

Upon fail-over for IOR o:

invoke redirect(s) at all LMRs
if received a quorum of replies <o’, pvid> then
    redirect to o’ with highest pvid,pvc
Otherwise (communication errors with LMRs)
    wait time-out and retry LMRs

Figure 4.7: Client Redirection

serve clients. In this case, the service acquisition will only be attempted again when the view changes. The second abort occurs when the coordinator receives a negative reply of LockedContext, meaning that this context lease is locked. In this case, service acquisition needs to be restarted after a time-out that is as long as a lease, since the context lease might be a left-over of a dissolved component. The third abort occurs when the coordinator does not receive replies from enough LMRs to form a quorum, in which case proceeding might break mutual exclusion. A similar abort occurs when the coordinator renews its acquired leases, since one or more LMRs might become inaccessible. As in the second case, service acquisition is restarted after a time-out to try and contact alternate LMRs.

The client part of the protocol, as described in Figure 4.7, is much simpler than the server part. Whenever a client (actually, the client PI) detects a fail-over condition, it concurrently invokes all (or just many) LMRs, asking for redirection to a working server based on the current object reference which is provided as a parameter. A response from each LMR includes the PVID which identifies how up-to-date that LMR's knowledge. Once the client receives a quorum of responses, it redirects to a server according to the reply of the LMR with the PVID that has the highest PVC. If the client does not receive a quorum of replies (i.e., instead it detects communication failures), it retries the LMRs after a time-out. The client may eventually abort retrying after a predefined count of retries and return an exception to the client.

4.6.1 LMR Protocol Properties

FTS combines PVID, quorums and leases to enforce safety conditions that there exists at most one primary component at any given moment and that each new primary component contains at least one server from its predecessor. By applying state-transfer before updating the minPVC at the LMR servers, each server of a primary component is
guaranteed to be at least as updated as the servers in the previous primary component, so that the service state presented to clients does not regress. Hence, the service state survives as long as at least one server of the last primary component survives. Server-side progress, i.e., establishment of a new primary component after the current one dissolves, is guaranteed provided that at least one surviving server is able to communicate with a quorum of LMR servers\(^6\). Continued access to the service is guaranteed for clients that have access to a quorum of LMRs and to a primary component’s server. Note, however, that LMR access is only for redirection. Clients that have a valid reference to a server in a primary component can simply continue working without accessing the LMRs. Furthermore, a client can alternatively invoke as many LMRs as it can (if it can’t reach a full quorum) and try to invoke the returned reference with the highest PVID. Although this does not guarantee that the invoked server would be active, it may increase the chance of successful redirection for a client with too-limited access to LMRs.

Last, for resource conservation, the LMR sub-service is designed to be used globally for the same domain of deployment, in the sense that a single instance of this service supports multiple replication contexts of FTS services. Also, being updated with server view information, LMRs can apply various policies for client redirection such as load-balancing.

### 4.7 Age and State-Transfer

Each time the view of a component changes, new servers may have joined the component. The term “new” does not necessarily indicate servers that were recently instantiated, but servers that were unable to participate in the previous view of this component due to disconnections. If the component has served clients as primary in its previous view, then the state of its surviving members may have updated as a result of executing client requests, as opposed to the state of the newly-joined servers. In order to become as up-to-date as the “veteran” servers, the new servers receive a record of the state of the entire replication context. This state consists of the updated states of object replicas, indications of object replicas that were dynamically added to or removed from the replication context, cache contents and the PVID of the veteran server that provided the state.

In order to actually transfer state between servants (application objects), each servant must implement an interface for getting and setting state. Further details on the requirements of application objects can be found in Section 3.2.4 and in [45].

\(^6\)When LMRs might become accessible from multiple components, locking policies need to be applied in order to avoid deadlocks, such as ordered locking [111]. This guarantees that out of multiple candidate components that are sufficiently up-to-date, one will succeed.
Prior to beginning state-transfer, FTS assigns roles to all the members of the view. One server, which is no less up-to-date than all the other members of the new view, serves as a provider of the state. Servers that are less up-to-date than the provider take the role of consumers and should process the transferred state and update their objects accordingly. Last, servers that are as up-to-date as the provider take a role of none, that is, do not process the delivered state information. Additionally, if all the servers are equally up-to-date (such as in a view change that follows a failure of a server), then there is no need to perform state-transfer at all.

In order to compare the update degree of two servers, FTS defines an age that consists of the following fields: a primary view id (PVID), a consequent view id (CVID) and an update count (UC). The PVID itself consists of two fields: a Primary View Count (PVC) and a View ID (VID). The CVID and VID are view identifiers that are generated by the GC subsystem for each new view. The UC is the total count of all the updates performed on all the objects in the server’s replica. The PVC is a monotonically-increasing value that is uniquely assigned to each new primary partition, as described in Section 4.6.

The rationale behind the age definition is as following. Each new view installed by the GC subsystem has a unique view id assigned to it by the GC subsystem. Also, every primary view has a higher unique PVC value assigned to it. Since a server can have its state updated only during a primary view, then a server age with a high PVC value represents a state more up-to-date than an age with a low PVC value. Still, this is not enough. The virtual synchrony property of the GC guarantees that two servers deliver the same set of messages in the same primary view if they also install the same consequent view. Thus, having the same pair <PVC, CVID> in the age indicates that two servers have the same update degree.

Note that two servers whose ages have the same PVC but different CVID (view split) must transfer state from one to the other when installing a new primary view together. This is because one of the servers may have delivered and executed updates that were not delivered at the other server during the end of the previous common primary view. These updates were not acknowledged to the clients that issued them (since they were not safe) but they did change the service state of one server, making the states of both servers inconsistent.

The UC field of the age record indicates how many updates does a server’s state reflect. This value is used for selecting a provider from several candidates who all have PVIDs with the highest PVC value in a view, but not necessarily the same CVID, and thus possibly not the same UC, due to possibly different amounts of additional unsafe updates, as explained above. In this case, the selected provider is the candidate with the highest UC. The reason for this is as following. Every state-transfer during view-change also transfers the request cache contents (see Section 4.7). Every request that is
waiting for a safe notification in order to return to the client is aborted with a fail-over exception after view change if the update it depends on is not in the cache anymore following state-transfer, meaning it was declared unsafe, as shown in Figure 4.3. Thus, these requests are likely to be retried by their respective clients. On the other hand, every update that is in the transferred cache contents is declared as safe after the view-change and requests that wait for it can return to their clients. Also, the higher the UC value of a candidate’s age, the more requests that candidate’s cache will contain. Therefore, by transferring the cache of the candidate with the highest UC, there would likely be less client re-invocations as a result of view-change, compared to selecting any other candidate.

To summarize the role selection stage, a candidate for provider is a server whose state age has the highest PVC value. Out of several possible candidates, the one with the highest UC becomes the provider. If several candidates have the same highest UC value, the one with the lowest rank is selected. Every other server with the same <PVC, CVID> as the provider takes a role of none, and the rest are consumers.

FTS also maintains an age for each object in a replication context, in addition to the global age of the server w.r.t. the replication of the context. This age is used in the request processing protocol, as explained in Section 4.4. However, in practice, an object’s true age resembles the age record as explained above rather than the simplified version that was presented in the request processing protocol. The number of updates applied at the object is stored in the UC field. The PVID contains the last primary view in which an object was updated, and the CVID contains the consequent view of that PVID. By using per-object age, state-transfer can be made more efficient when setting the state of objects at the consumers, since an object with the same <PVC, CVID> as its matching transferred state does not need to set state - it is already updated. This is especially useful for servers who have become disconnected during transient network partitions and then re-joined a primary component.

Note that the global age of an entire replication context may become higher than that of the most up-to-date object. This can happen when no updates occur in the lifetime of at least one view of a primary component. The PVC of the global age is advanced at the beginning of each new primary view, whereas the age of an object is advanced only when updates are applied. Still, the “artificial” advancement of the global PVC is required since it takes place in the beginning of a view in which updates may eventually occur, so the members of the new primary component must be distinguished from other servers a-priori7.

Updating the CVID part of a global server age or the age of an object is not possible

---

7In the future design modification we consider for the LMR sub-service where primary components are decided by themselves (see Chapter 8), a global age will not be required.
during the current view, since the next view is not installed yet. Therefore, each updated object is only tagged as *dirty* and the CVID field of its age is cleared. In the beginning of each new view, the view id of the new view is inserted as the CVID value for all dirty objects as well as global age.

In the beginning of each new view, the replication context needs to achieve a state of *coherence*, where there are no requests being executed on the objects. This is required for computing the UC value of both objects and global ages, as well as for later reading and writing object state during state-transfer. In order to reach coherence, processing of client requests that have not begun executing yet is suspended when the GCI signals that the view is about to change. After the new view is notified from the GCI, the server waits for the currently-executing requests to terminate, after which coherence is achieved and age calculation and state-transfer may commence.

### 4.8 Request Identification and Caching

FTS provides a transparent fail-over mechanism for clients that can guarantee a reliable execution of each client request\(^8\). However, if a client detects a failure and fails-over to another replica after its update request has already arrived at a replica, the update might get executed more than once at each replica before the client gets one reply. The re-execution may occur perhaps even in a non-consecutive sequence, so even retries of idempotent updates might have an effect different than that of a single execution. As a result, an inconsistency might appear between the actual state of the object and its execution history as it appears to clients. To prevent this from happening in non-replicated applications, the CORBA standard includes a safety condition termed *at-most-once* execution semantics [87] stating that every request a client receives a reply for is executed no more than once at the server. Each CORBA-compliant ORB guarantees this safety condition but only within the boundaries of a single continuous communication session between a client and a single server.

FTS also needs to uphold the at-most-once safety condition as part of maintaining service state consistency towards clients. However, in order to get a reply for one request, an FTS client may communicate with more than one server, or perhaps establish more than one session with the same server, due to the break-ups of invocations that actually trigger the fail-over mechanism. In order to guarantee at-most-once semantics under these conditions, FTS employs a *request cache* into which every request is stored (along with its matching reply, which is added later) at the first time it is delivered either

---

\(^8\)provided that the client is able to communicate with a primary component for a long-enough time for the request to finish executing
from the client connection or from the ABCAST transport. This cache is replicated at every FTS GOA. Whenever a client issues a request to a server, the request is first checked against the cache to see if it is a retry of a previous request. If so, the reply returned to the client is that of the previous request once it is done executing.

In order to identify a request and its following retries, each request is assigned a globally-unique request id that is subsequently attached to its retries. Generating such an id is outside the scope of the legacy (pre-FT) CORBA standard. Hence, FTS has a choice of generators depending on the version of the ORB. If the ORB supports the newer FT-CORBA standard, then it also supports the generation of a unique request id in the form of the `FTRequestServiceContext` record [90]. When running over legacy ORBs, FTS can generate the unique request ids by itself, using the FTS client PI and either a transparent proxy (for complete portability) or a little-intrusive but faster ORB hack, as discussed in [45].

In order to prevent the cache from growing indefinitely with the continued usage of the cluster, every request is held in the cache for a predefined grace period, after which the client is assumed not to retry that request. Then, the request is purged by a dedicated daemon thread. In Chapter 7 we further discuss this method of evacuating expired requests and suggest a more efficient improvement in the form of Selective Acknowledgments that allow removing requests prior to the time-out.

The request cache is also part of the state of a replication context that is exchanged between servers during state-transfer. This is because servers joining a GC cluster might later receive re-invocations regarding requests that may have already completed execution prior to the state-transfer. These servers must be able to return a cached reply so that at-most once request execution semantics are not violated, so they also receive the cache contents of the server that provided the updated state. See Section 4.7.

### 4.9 Dynamic Object Management

FTS supports dynamic creation and deletion of replicated objects, as a fundamental requirement for replicating modern applications that often require these abilities. In order to dynamically generate new objects, local application object factories need to be registered at the GOA for each supported object type. These factories need to generate only object implementations, i.e., servants, upon a request from the GOA, which later automatically handles the CORBA registration and activation of the servant as a CORBA object. The reason that factories must be registered at the GOA is that after a client explicitly invokes a method that creates a new object, a GOA may need to implicitly create that object at consumer servers during state-transfer (see Section 4.7).
Consequently, FTS offers two alternate methods for handling object creation at the application level during client request processing. The application may either generate the servant by itself and then register it at a GOA of its choice, or simply ask the GOA to create and register the servant. The first method is longer than the second but is also more compatible with the standard POA. Also, it lets the application use a wide variety of custom constructors to create the object, instead of the default constructor that is used by the factory registered at the GOA. Both methods, however, allow the created object to be associated with a replication context that is different from that of the creating object, by locating/creating the GOA that handles that context and delegating the creation/registration to that GOA.

Client requests that involve object creation and deletion must be tagged as updates. Although invoking such a request may not change the state of the creating or destroying object, it does change the state of the replication context as a whole, so it must be reflected at all the copies of the replication context.

In order to implicitly destroy objects during state-transfer, each servant must implement an interface through which the GOA can notify it to self-destruct, after first de-activating it at the CORBA level (see Section 3.2.4). As a result, similar to the object creation support, an application can either remove a servant by itself after asking the GOA to de-activate it, or by asking the GOA to handle the entire removal.

Last, note that factory registration, as well as automatic creation and deletion of objects, are unique additions to the GOA and are not part of the standard POA interface. Beyond being an object adaptor, but as a single entity that manages all aspects of replication in FTS, the GOA needs to incorporate these additional capabilities (see Section 3.2.1).

4.10 Discussion

4.10.1 Replication Granularity

The granularity of replication, that is, the “size” of the basic state unit to which replication is applied, depends on the replication infrastructure designer’s choice. Most contemporary middleware replication solutions (see Section 2.1) tend to promote object-based replication, meaning that each object must be replicated independently of other objects, including failure-detection, consistency, state-transfer, fail-over, etc. An alternate extreme approach, promoted by classic fault-tolerance research, is process-based replication. In this case, the replicated state consists of the entire memory image of a process, and replication operations apply to the process as a whole. Each approach has its benefits, as discussed below.
There are several important arguments in favor of object-base replication. In some applications, objects and threads may crash-fail individually, without bringing down the rest of the process. In these cases, monitoring and replicating individual objects seems logical and even possibly efficient, since it allows partial recovery of the application state during failures. Also, when the state of an application process consists of a set of objects (which is typical for middleware applications), then state-transfer, for example, is made object-aware in order to efficiently save and restore the state of the entire application. Object-awareness becomes crucial also when the objects in the replicated process are inter-connected using middleware references, so the inter-object relationship need also become part of the state. Furthermore, in cases where there is no accessible global registry of all the objects inside an application process, it may not be possible to read and write the state of all the objects together. Hence, per-object replication is essential. Last, object-based replication is consistent with the common object-oriented paradigm employed in design and implementation of middleware services.

On the other hand, in many cases it may be better to set replication granularity at a larger scale than a single object, up to the the entire server process. For starters, process-based replication can be more resource-efficient than object-based replication. For example, proactive fault-detection (periodical checks) is more efficient at the process level than at the object level, since it saves many additional checks. Network utilization in the inter-server transport can be increased by channeling more client requests to the same transport, which results from sharing the same transport over multiple objects, so all the clients of these objects eventually propagate their requests through the shared transport.

When a process state contains objects that invoke each other during request processing, replicating the process as a whole (despite the need for object-aware state-transfer) may prove more efficient and simpler to implement than replicating its objects individually. The reason is that correct and efficient interaction between individually-replicated objects is a non-trivial problem. Consider, for example, two actively-replicated objects \( A \) and \( B \), where \( A \) needs to invoke a method on \( B \). A default solution of each replica of \( A \) invoking each replica of \( B \) is both incorrect, since the method may be executed more than once at each replica of \( B \), and inefficient, since it results in at least \( O(m \cdot n) \) invocation messages. Many works address this issue, such as [48], where a specially-designed protocol for efficient group-to-group communication is developed, or NewTop [72], which creates a unifying group to communicate between two groups. However, if both interacting objects are replicated together, then the interaction is reduced to a set of local invocations from each replica of \( A \) to the matching replica of \( B \), with no network communication at all\(^9\).

\(^9\)This form of communication is implemented in FTS. See Section 4.5.
As a last argument for process-based replication, note that in many applications, objects have no threads of their own, but are rather memory images inside the address space of the containing process. In these applications, threads that handle client requests are spawned and managed by the server process, without relation to any object’s status. As a result, objects cannot crash-fail individually in such applications, and value-faults can be detected by voting upon replies. Thus, there is no point in proactive object-based failure-detection.

FTS does not favor one approach over the other. Instead, it offers the entire spectrum of options, using the concept of context-based replication. Each replication context, which is the basic unit of replication, contains any number of objects starting at one. Context-based replication can be as fine as object-based replication, by placing each object in a different context, or as coarse as process-based replication, by putting all the objects of a process inside a single context. Fault-detection, state-transfer and fail-over are always provided at context granularity. However, fault-detection and consistency can be further refined to focus on each object inside a context, as described in Sections 4.3 and 4.4, respectively. Thus, objects can be effectively be independent of each other while still residing in a common replication context for resource-sharing efficiency. The choice of associating objects with contexts and of configuring each context’s replication operations is left to the application designer and administrator for maximum flexibility.

4.10.2 Selecting Consistency Modes

As shown in Section 4.4, FTS supports multiple modes of consistency that differ in type (linearizable or sequential consistency) and granularity (an entire replication context or a single object). In this section we discuss considerations for selecting a proper consistency mode depending on the application.

First, there is the aspect of granularity\(^\text{10}\). As already explained in Section 4.4, context-granularity is required for maintaining the required consistency type in FTS when objects are inter-dependent. On a broader scale, there are several other points to consider. As already mentioned, context-granularity imposes total-ordering of updates and age-delay of queries for all the objects in the same context together. Obviously, this can reduce performance compared to object-granularity. However, recall (from Section 4.5) that cascading invocations inside contexts when using context-granularity can be more efficient in terms of communication than the standard context-to-context or object-to-object invocations.

\(^{10}\)In this particular discussion we specifically address the subject of consistency granularity in FTS, as opposed to Section 4.10.1 where we discuss granularity of replication in general.
Another benefit of using context-granularity in general is avoiding deadlocks that can occur as a result of cascading invocations between replicated objects. Consider two objects, $A$ and $B$, where each performs an update on the other during request processing. A deadlock can occur if a request from client $C_A$, while executing at object $A$, waits for a request from client $C_B$ that executes at object $B$ and is pending since it is waiting for the request from $C_A$ to complete. Ignoring the poor application design this case suggests, the deadlock would not have happened when if both $A$ and $B$ were replicated together in the same context using context-granularity consistency.

Context-granularity can also be used as a substitute for explicit locking in order to achieve transaction-like serializability [47] of operation sequences. By putting all the objects participating in sequences of operations in the same context under context-granularity consistency and by using cascading invocations, a client request can engage multiple objects as a single atomic operation. Consider, for example, a banking application where two account-manager objects in the same context, while processing client requests, modify the same internal account balance object through operations of withdrawals and deposits, where each operation is a read-modify-write sequence. Using object-granularity may produce a linearizable run, but the effect of concurrently running two operations of withdrawal or deposit may result in loss of one of the operation's effect. Using context granularity forces these operations to be carried out one after the other so no effect is lost.

The second aspect to consider is the consistency type. As already indicated in Section 4.5, linearizable consistency can be guaranteed application-wide due to its locality property. Sequential consistency, on the other hand, is not local, so even if only some of the replicated contexts are sequentially-consistent, the global application consistency may not even be sequential.

On the other hand, sequential consistency can lead to better performance. Totally-ordering both updates and queries may delay a query unnecessarily. Consider, for example, client $C_A$ that issues an update to replica $A$ of an object and client $C_B$ that issues a query to replica $B$ of the same object. Suppose that client $C_A$'s update was already executed before client $C_B$'s query reached the server. If the replicated object employs linearizable replica consistency, then client $C_B$'s query would be delayed at replica $B$ until client $C_A$'s update was done executing in this replica as well. If, alternatively, sequential consistency is employed, then the query of client $C_B$ could be carried out immediately, quite possibly shortening the average request latency. In FTS, fast query processing is an advantage of sequential consistency, which is manifested especially in services such as information repositories, where common usage involves many queries but relatively few updates (see Chapter 5).

Note, however, that linearizable consistency might be required by application needs,
e.g., when a client holds more than one reference to the same FTS-replicated object. In such a case, a client might fail if an update issued through one object reference is not reflected in a later query for the same object that is issued through the alternate reference. This can happen if each reference points to a different replica of the same object.

Last, note that safe-delivery is optional in FTS and can be dropped for objects in which durability of modifications that result from updates can be guaranteed by other means. As an example, consider a replicated object that is a proxy of a 3rd-tier persistent storage layer. If each update of this proxy results in committing the result into the database (which is common to all the replicas of the proxy), then there is no need for safe delivery.

4.10.3 Rationale of Handling Dynamic Configuration

As indicated in Section 4.6, the primary goal behind the LMR service was to handle dynamic changes in the GC clusters that result in multiple concurrent components of the same cluster. On the server side, state consistency towards clients needs to be maintained despite changes. On the client side, there needed to be a way to provide clients with alternate object references in case of fail-over. FTS addresses the problem of dynamic changes through a single architectural module that provides solutions for all related aspects.

Currently, FTS uses the primary component technique for global consistency, since it is application independent. If multiple concurrent components were allowed to perform updates on the same object/context, then FTS would have to resort to application-dependent state-merger when components re-unite. Still, a future version of FTS may permit queries in non-primary components since it is still compatible with sequential consistency and thus seems practical for applications such as distributed information systems.

FTS employs a quorum-system for determining a mutually-exclusive primary component since components may not be able to communicate with each other due to network partitions. A general quorum-system further adds better accessibility and more resilience to individual LMR server failures compared to a single server. The definition of the quorum-system is left to the application designer or administrator for maximum flexibility.

The leases employed in the interaction between coordinators and LMR servers are required for establishing mutual exclusion between components when some LMR servers can be accessed from more than one component. This can happen when a GC cluster splits due to false suspicions or network routing errors. Another important reason for
employing leases is to reduce the number of state-transfers in non-primary components, since a state-transfer can be a highly network- and/or CPU-intensive operation.

Using leases to reduce the number of state-transfers is explained as following. FTS design guarantees that the state of a replication context survives without regression provided that at least one server of the current primary component, i.e., the component whose PVC is the minPVC value in a quorum of LMRs, survives to the next primary component. In order to uphold this guarantee, a coordinator must verify that all the servers of the current primary component are sufficiently updated before committing the PVC value to the LMRs. Therefore, state-transfer must be conducted before updating the LMRs. However, there is no need for a component to perform state-transfer if it does not get to become primary. Therefore, by acquiring leases first, a component verifies that it will become primary (has sufficient PVID and is mutually exclusive) unless the configuration changes, and can thus initiate state-transfer. This is a useful quality in transition periods when servers are added and removed, since being held in a long state-transfer operation with its state not fully defined might delay an FTS server from responding quickly to a view change and starting to participate in establishing a new primary component.

Client redirection is guaranteed to point to the new primary component using the same principle of shared register emulation [7] with the LMR servers posing as the register. This register is essentially MRSW (multi-reader single-writer) since writing occurs in mutual-exclusion due to leasing. Therefore, writing is executed in 2 message rounds (request+reply). Still, we use only 2 message rounds also for reading, i.e., without the additional “write-back” rounds. This is because each client redirects independently of other clients. Therefore, there is no need to guarantee read-monotonicity between clients, that is, that each client must see a new primary component information during redirection if another client already saw it in a previous redirection. Instead, the design guarantees only read-after-write, meaning that once a primary component’s coordinator finishes updating the LMRs, every client redirection would point to a server in the primary component.

4.10.4 Replicated Object Reference Semantics

FTS makes no change in the standard CORBA IOR structure. However, IOR semantics are slightly modified to accommodate for the existence of object replicas. Specifically, two (or more) replicas of the same CORBA object are conceptually equivalent from the application point of view. In order to maintain application transparency, they should appear as such to application code, and in particular their corresponding IORs should be considered equivalent by the standard IOR comparison method, namely the
is Equivalent() method of the Object interface, which is implemented by all CORBA objects. This problem is handled differently at the client side and the server side, as discussed below.

At the server side, the application is aware of object replication, albeit in a minor degree. The GOA provides an explicit is Equivalent() method for an extended comparison of IORs, which considers group relevance by comparing the FTOID component of the IOR. Using this API, we make sure that IORs of replicas of the same object will always appear equivalent.

At the client side things are more complicated. Recall that FTS transparently redirects client requests to alternate servers. At the CORBA level, this is accomplished by having the client PI raise a LOCATION FORWARD exception with the alternate reference. This exception "reprograms" the object reference to effectively point to the alienate location but virtually remain the same. Thus, when comparing two IORs of the same object, one obtained before redirection and one after, they will appear equivalent.

On the general case, however, the is Equivalent() method of each CORBA object is beyond FTS control. According to the CORBA standard, this method, both at the client stub and server implementation, is either automatically generated or implemented by the ORB itself. For portability, FTS does not interfere with either implementations. Thus, when using legacy (pre-FT) ORBs, calling is Equivalent() method on the client side would return false when comparing references of different replicas of the same object.

Note, however, that even this does not generally violate CORBA reference semantics. CORBA supports a model of weak reference semantics [36]. This means that there can be two different references that point to the same object, especially since a CORBA reference is generated independently of the servant (object) itself, by registering it at an object-adaptor. Thus, the proper interpretation of a false return value when comparing two references is that the references are distinct, but they still might point at the same object. Therefore, the reference semantics provided by FTS are not weaker than the default CORBA reference semantics.

Additionally, the client-side reference comparison problem can be completely eliminated in ORBs that support FT-CORBA [90]. According to the FT-CORBA specification, compliant ORBs must compare object references while considering group relevance, i.e., that the references may point to replicas of the same object. This is accomplished by having the is Equivalent() method check the TAG_FT_GROUP component of the references, which marks them as belonging to a group according to FT-CORBA. The TAG_FT_GROUP resembles the FTOID component of FTS in the sense that it contains specific fields that globally and uniquely identify the replicated object. By a simple mapping of FTOID contents to the TAG_FT_GROUP contents, FTS can also generate the
TAG_FTGROUP and embed it in references, thereby making the is_equivalent() method aware of FTS replicas.
Chapter 5

FTS Performance Evaluation

This chapter describes how FTS performance was measured and some interesting conclusions that arise from this test regarding efficient implementations of middleware services. The performance test focuses on request processing during normal operation, i.e., without server failure. We use short requests that take very little time to execute so that the test results are primarily affected by FTS’s own processing limitations. The test simulates multiple clients accessing an FTS cluster and spans over multiple client loads and cluster sizes to estimate FTS scalability.

5.1 Test Environment

The platform on which FTS was tested is an IBM JS20 Blade Center with 28 blades (machines). Each blade has two PowerPC 970 processors running at 2.2GHz with 4GByte RAM. The blades are divided into two 14-node pools, each internally connected with a Gbit Ethernet switch, and both switches are connected to each other. Additionally, the cluster includes an Intel X346 server with dual Pentium 4 Xeon 3.0GHz (hyper-threaded) with 2GByte RAM, used also as an NFS server for the common file system on which FTS is installed.

The blades and the server run Suse Linux Enterprise Server 9 (2.6.5 kernel) service pack 3. All FTS components run on IBM JVM v1.4.2 service pack 2 for PPC and Intel platforms, respectively. The ORB used is ORBacus for Java [52] v4.1.3. Ensemble [1] v2.01 is used for group communication.

FTS is deployed as following. Clients are running in blade pool 1 and servers in blade pool 2, so that all clients are equally distant from all the servers. Each client blade runs 250 clients sharing a common ORB. Each server blade runs a single server and a local
Ensemble daemon. A single LMR is installed on the Intel server for the purpose of constructing the server cluster. Also, a single ORBacus interface repository daemon runs on the Intel server. Last, the Intel server runs a test manager, whose purpose is to conduct a single test of clients vs. servers and an executive that generates sequences of tests, passes each test to the test manager and collects the results.

5.2 Test Configuration

Each client runs as a separate thread that performs a sequence of simple synchronous invocations on a separate server object. The interface of the test server object is called Hello0bj and consists of two methods called set() and get(). The set() method accepts a single string parameter and stores its value in the object, while the get() method returns the value of the stored string. Accordingly, the set() method is tagged as an update whereas the get() method is tagged as a query. We used 15-character string parameters in method invocations.

FTS servers are configured to use the object-granularity consistency (object-sequential dispatcher + object age) and simple adaptive batching, without safe delivery\(^1\). Each server has a 500-thread pool for handling requests arriving from clients and a 20-thread pool for executing client requests in the dispatcher after being delivered back from the ABCAST transport. This allocation policy is explained below when the results are analyzed in Section 5.5.

5.3 Test Procedure

There are 3 types of tests: a write test, a read test and a mixed test. During a write test, each client invokes only updates on its server object. In the read test, only queries are used. In the mixed test, each client randomly chooses either update or query with a uniform probability of 0.25 for choosing updates. The purpose of the mixed test is to simulate a possible usage of, e.g., a distributed information system where most of the invocations are queries.

For each test type, we measure performance in multiple test cases. Each test case is defined by a specific client load (number of clients) vs. cluster size (number of servers). In each test case, the client load is equally divided over all the servers in the cluster.

\(^1\)This test focuses on measuring the bare overhead of FTS components at peak performance, so we omitted explicit delays of a sequential dispatcher and safe delivery.
For each test case, results are averaged over 4 runs. Each run consists of 2 phases of warm-up and test. During warm-up, each client performs a sequence of 500 requests. After finishing warm-up, a client notifies the test manager but continues sending batches of 100 “tail” requests, to maintain load on the servers. After the last client is done warming up, the test phase begins. Each client sends additional 500 requests as a test, notifies the test manager upon completion, and continues to send “tail” requests, just as in the warm-up phase. Once the last client completes the test, the test phase ends. The test manager notifies all clients and servers to shut down and write the results. Measurements are collected only during the test phase, assuming that the servers are at a constant peak load during that time.

5.3.1 Measurements

For invocation latency, we used test cases of 1 client vs. cluster sizes of 1,2,4,6,8 servers, and measured invocation latency at the client as round-trip time from request to reply during the test phase. We also compare the resulting latency to the non-replicated case, which is the same application server but not using FTS (actually, simply replacing the GOA with the standard POA).

In order to measure cluster throughput we put each cluster size of 1,2,4,6,8 servers under gradually increasing client loads of 500,750,1000,1500,2000 and 2500 clients. In each test case we measure throughput at each server by dividing the count of completed requests during the test phase by the length of the test phase. The total cluster throughput is the sum of the individual server throughputs. As in the latency test, we compare the results with the non-replicated server.

5.4 Summary of Results

Figure 5.1 is a graph of invocation latency as a function of cluster size for queries and updates. Note the overlapping boxes in the bottom left of the graph whose heights indicate the non-replicated invocation latency for both a query and an update (approx. 1.9 msec).

Figures 5.2 and 5.3 show cluster throughput during the write test as a function of cluster size and client load, respectively. Similarly, Figures 5.4 and 5.5 plot cluster throughput as a function of cluster size and client load during the read test. Last, Figures 5.6 and 5.7 plot cluster throughput during the mixed test.

Throughput comparison with a non-replicated server is presented in Figures 5.8, 5.9 and 5.10 for write, read and mixed tests, respectively. As in the latency test, the non-
Figure 5.1: Invocation Latency Test Results
Horizonttal:_CLUSTER x Time
Vertikal: Invocation Time (msec)

Figure 5.2: Write Test: Throughput vs. Cluster Size
Horizonttal: Cluster Size
Vertikal: Throughput (req/sec)

85
Figure 5.3: Write Test: Throughput vs. Client Load

מבחן כתיבה: תועות ביחס ל놈ה הלקוחות

Figure 5.4: Read Test: Throughput vs. Cluster Size

מבחן קריאה: תועות ביחס למיקום השירותים
Figure 5.5: Read Test: Throughput vs. Client Load

מבחן קריאה: תפוקה ביחס ל UInt והלות

Figure 5.6: Mixed Test: Throughput vs. Cluster Size

מבחן מעורב: תפוקה ביחס לגודלמקבץ השרתים
replicated server uses the same code as the replicated server, only replacing the GOA with a standard POA.

5.5 Analysis

5.5.1 Invocation Latency

Figure 5.1 shows that query invocation latency is about 2.5 times slower than when not using FTS. The reason for the added latency is the extra dynamic processing of the request in both the server-side interceptor and the outer POA of the GOA (see Chapter 3). The interceptor verifies that the target object is replicated (through the FTOID component of the IOR) and processes FTS service context. The outer POA checks that the request is a query by checking its signature in its configuration database.

The invocation latency of an update is higher by an order of magnitude than that of a query. Obviously, the added delay is linked to the passing of the request through the ABCAST transport before executing it. Examining the work involved in this lists several factors. First, there is the processing of the request through a dynamic (DSI) servant that serializes it and sends it through the GCI to Ensemble. In its current
Figure 5.8: Comparison of FTS write throughput with non-replicated server

Figure 5.9: Comparison of FTS read throughput with non-replicated server
version, Ensemble interfaces with Java applications (including FTS) only as a daemon process that communicates through the loopback network interface. Thus, communication with Ensemble requires at least 3 more operations of copying the serialized request in each direction (marshaling to network format, copying between address spaces and un-marshaling). Next, there is the cost of running the ABCAST protocol itself and accessing the network. Last, after the request is delivered back from Ensemble, it is de-serialized and only then executed.

According to the profiling information of the write test in Table 5.1, the serialization, de-serialization and execution are negligible compared to the round-trip through the GC, which is mostly responsible for the added 90% of the delay. Note, however, that these results reflect the fact that we are using very short requests that execute quickly. More complex requests and/or longer-executing requests should make the GC-specific overhead the minor part.

Examining the behavior of the TOTEM ABCAST protocol, which is used by Ensemble in our test configuration, reveals the following. This protocol operates by ordering the servers in a logical ring and passing a token along that ring. Each server that wishes to broadcast a message has to be holding the token, which carries the sequence number of the last message that was broadcast. Upon receiving the token, a server can broadcast its pending messages ordered according to the token’s sequence number, and then
<table>
<thead>
<tr>
<th>Cluster Size</th>
<th>1</th>
<th>2</th>
<th>4</th>
<th>6</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>Server RTT</td>
<td>43.3</td>
<td>44.2</td>
<td>43.8</td>
<td>43.8</td>
<td>43.7</td>
</tr>
<tr>
<td>Serialization</td>
<td>0.15</td>
<td>0.19</td>
<td>0.19</td>
<td>0.15</td>
<td>0.14</td>
</tr>
<tr>
<td>De-serialization</td>
<td>0.19</td>
<td>0.19</td>
<td>0.13</td>
<td>0.19</td>
<td>0.17</td>
</tr>
<tr>
<td>Execution</td>
<td>0.89</td>
<td>0.88</td>
<td>0.78</td>
<td>0.77</td>
<td>0.77</td>
</tr>
<tr>
<td>GC RTT</td>
<td>39.2</td>
<td>39.6</td>
<td>39.4</td>
<td>39.3</td>
<td>39.6</td>
</tr>
<tr>
<td>$t_{pr} = (\text{Server RTT}) - (\text{GC RTT})$</td>
<td>4.1</td>
<td>4.6</td>
<td>4.4</td>
<td>4.5</td>
<td>4.1</td>
</tr>
</tbody>
</table>

Table 5.1: Write invocation latency profiling (msec)

פורמיול של תמסוג תמסוגים (מיליסניון)

update the token accordingly. The token is later either piggy-backed on the last message from the currently holding server or explicitly broadcast (if there were no messages to send) and picked up by the next server in ring order.

Note that from this behavior, one would expect that the bigger the cluster, the longer the delay, since a server has to wait on average for a half a ring’s worth of delay before being able to transmit. However, in the latency test (Figure 5.1) only one server actually broadcasts requests, while others only relay the token as a very small message. The network latency of the blade center, following a simple ping test, is close to 0.1 msec, so the entire delay imposed by the token relay would amount to less than 1 msec, which helps to explain why the overall latency stays nearly fixed with cluster growth.

The significance of the flatness of the invocation latency graph is best explained when comparing Figure 5.1 with similar Figures that were presented for a pure replication service implementation such as OGS [31]. Comparing the data itself yields little meaning since OGS was tested using its C++ version running on ORBs different than ORBacus and using a different hardware platform. However, the latency graphs’ behaviors present a difference of interest. The OGS latency graph shows a near-linear latency growth with cluster size, which is attributed to the repeated cost of serialization/de-serialization and sending a point-to-point message for each additional participant. In FTS, on the other hand, invocation latency (of FTS, disregarding ABCAST specifics) does not grow with cluster size since it uses more efficient broadcast primitives and performs serialization/de-serialization of a request only once. This added scalability provides a clear justification to the approach taken in FTS to delegate ABCAST implementation to efficient dedicated GC services.

Eternal [77], which is a classic example of a low-level interception solution, does not provide a similar latency graph whose behavior can be compared to that of FTS. Still, [77] does indicate a very low additional overhead (no more than 10%) compared to a non-replicated server, when using a 3-fold replication. As with OGS, there are

91
many differences that already makes any comparison ineffective (C++ vs. java, direct linking vs. daemon connection, different ORBs and platforms). However, there is one fundamental difference in favor of Eternal (or any low-level solution), which is direct r/w access to the low-level packets that are exchanged between the client and the server. This removes the need for any explicit serialization/de-serialization stages, which would significantly benefit FTS throughput, as is further discussed in Section 8.2.1.

5.5.2 Cluster Throughput

Figures 5.4 and 5.5 show that FTS query throughput improves with cluster size and client load. Furthermore, Figure 5.9 indicates that FTS eventually provides a speedup compared to a non-replicated server at query processing. This is quite expected considering that query processing is handled independently at each server so adding servers contributes directly to the cluster processing capacity. Note that a 2-server cluster is still slower than a single non-replicated server, but speedup begins at clusters of 4 servers. This is consistent with the latency slowdown (1:2.5) of FTS from the previous Section. Note that our measurement system uses fixed groups of clients that perform synchronous invocations. Also, query execution is mostly CPU-bound (disregarding the rather minor client-server message latency). This means that two concurrently-executing queries take together the same time as if they were executing one after the other\(^2\). Consequently, longer client invocation latency translates to smaller throughput, at about the reverse ratio.

Update processing, on the other hand, shows a throughput ratio of 1:3 to 1:10 compared to the non-replicated case, as is shown in Figure 5.2. This is much higher than what could be expected by the latency ratio (approx. 1:25), which can be explained as following. Unlike query processing, which is CPU-bound, update processing involves a long period of waiting at the ACAST transport stage. This waiting time is exploited for processing additional requests, thus yielding much better throughput. Also, the default adaptive batching contributes to further increasing the throughput, as explained in Chapter 6.

Figures 5.2 and 5.3 show that update processing improves with client load, but degrades considerably with cluster size. From its peak performance in a cluster of 2 it degrades by more than a factor of 2/3 in a cluster of 8. To understand this phenomenon, we analyze the work involved in update processing. Each update, after being received from the client, is serialized into a byte-buffer, then undergoes ACAST until being

\(^2\text{Recall that our test platform is dual-processor, so it may execute queries faster than a single processor. However, the speedup explanation holds since the un-replicated version of the test application runs on the same hardware as well.}\)
Table 5.2: Write latency profiling for 1500 clients (msec)

<table>
<thead>
<tr>
<th>Cluster Size</th>
<th>1</th>
<th>2</th>
<th>4</th>
<th>6</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>Server RTT</td>
<td>1010.35</td>
<td>809.0</td>
<td>1183.7</td>
<td>1864.04</td>
<td>2445.68</td>
</tr>
<tr>
<td>$t_w = GCRTT$</td>
<td>56.56</td>
<td>98.08</td>
<td>917.58</td>
<td>1733.3</td>
<td>2346.95</td>
</tr>
<tr>
<td>$t_{pr}$</td>
<td>13.8</td>
<td>21.32</td>
<td>208.54</td>
<td>393.9</td>
<td>572.42</td>
</tr>
<tr>
<td>Clients per Server</td>
<td>1500</td>
<td>750</td>
<td>375</td>
<td>250</td>
<td>187.5</td>
</tr>
</tbody>
</table>

Recall that the same client load is equally divided on a growing number of servers. Thus, the throughput of the serialization stage, which is executed independently at each server, only improves with cluster size. De-serialization and execution are performed for every request from each client at each of the servers. Thus, the amount of work that is imposed on each server in these stages does not change with cluster size. Therefore, the key stage that is responsible for the throughput degradation is the interaction of FTS with the ABCAST transport.

Table 5.2 shows profiling information taken in the write throughput test for the 1500 clients test cases, which reveals the following picture. The round-trip time (RTT) of a GC message grows considerably with cluster size, and becomes the most dominant part (close to 95%) of the total server RTT. Furthermore, by comparing this data with the data from Table 5.1, we can see that the time a request spends waiting (name it $t_w$) is much higher than the total time (both before and after ABCAST) a request uses the processor (name it $t_{pr}$). In fact, for most test cases with cluster sizes of 4 and up, the ratio $\frac{t_w}{t_{pr}}$ is larger than the number of clients allocated per-server. This means that during one request’s waiting time, even if all the requests from the other clients would use the processor, there would still be times where all the requests are in waiting states and the processor(s) become idle\(^3\). Even at the border cases (4-server cluster), where the $\frac{t_w}{t_{pr}}$ ratio is not higher than the number of clients per-server but is not low compared to it, idle gaps begin to form simply because not all client requests arrive together or close to each other. Note, however, that this throughput degradation is not a limitation of FTS, but rather an effect of the measurement system that equally divides a fixed client load over a growing number of servers, and of clients using synchronous invocations.

The growth of the GC RTT seems to be mainly attributed to the choice of protocols

\(^3\)Several FTS daemon threads, such as the cache thread and SACK handling thread can utilize a small part of the idle processor time, thus contribute to overall performance. See Chapter 7 for details.
in use. Totem, for example, plays a major role in RTT growth in this part of the test. The reason is that the token now gets delayed at each server until that server broadcasts all its stored messages. The growing cluster size also reduces the request batching effect (see Chapter 6) of the adaptive batching mechanism, which groups multiple requests into a single ABCAST message to reduce ABCAST per-message overhead. As the same client load gets divided over more servers, more messages are generated for the same load, with larger framing and protocol overhead. We intend to further explore this problem in the future to isolate the exact causes in the GC subsystem for the rapid RTT growth.

A different limitation, this time of FTS itself, appears when examining test cases in which more clients are allocated per-server but throughput does not improve by much. Consider, for example, the cases of 1500 and 2500 clients vs. a cluster of 4 servers. Even though there are 66% more clients per-server in the case of 2500 clients compared to the 1500 clients case, the throughput is only a little higher. The reason is that the number of concurrent requests that can be processed independently at the same FTS server is also limited by the thread pool size. The thread pool becomes exhausted at 500 threads, so although in the case of 2500 vs. 4 there are 750 clients allocated to each server, only 500 of them can actually be served concurrently, which is not much higher than the 350 clients per-server in the 1500 vs. 4 case.

Note that the thread pool limitation is not unique only to FTS, but rather a general limitation that is imposed on middleware services using a thread pool. Most common threading models, such as thread-per-request and thread-pool, make a fixed association between a request and a thread, which ends only when the request is done processing. Thus, a thread may not be able to switch to handle a different request even if its currently assigned request is pending for a long period (as is the case with FTS). Consequently, a thread pool might become exhausted if all of the pool’s threads are assigned to waiting requests, leaving the processors idle. Alternatively, one may consider dynamically changing the thread pool size, which may not be allowed by the broker (such is the case with ORBacus). Using a thread-per-request instead of a thread pool generates a high overhead of thread creation and destruction and may also lead to excessive resource usage and server instability. In FTS we used a very large thread pool (500 threads)\(^4\) in order to minimize the effect of thread pool exhaustion.

Equation 5.1 presents a unified formalization of the constraints that lead to throughput degradation of FTS in our measurements, and Figure 5.11 is a graphical representation of the problem. As mentioned before, \(t_w\) stands for the waiting time imposed by the GC ABCAST RTT, and \(t_{pr}\) denotes the total processor time out of a request processing

\(^4\)We experienced JVM instability problems with thread pools larger than 500 threads, so we set it as a limit.
cycle. $c$ and $s$ respectively indicate the total number of clients and servers in the test case and $P$ marks the size of the thread pool used for serving client requests. Note that under a more realistic usage with longer-executing CPU-bound requests, the ratio of $\frac{t_w}{t_{pr}}$ would be smaller (due to larger $t_{pr}$) and thus the overall throughput would be closer to that of a non-replicated server.

\[
\frac{t_w}{t_{pr}} > \min\left(\frac{c}{s}, P\right)
\] (5.1)

One last factor that decreases throughput and grows with cluster size is the CPU requirements of the GC subsystem (Ensemble in this case). As the cluster grows, a larger fraction $(1 - \frac{1}{s})$ of the requests that are executed at each server are received from remote servers, simply due to the fact that the same client load is divided over more servers. Delivering remote requests typically consumes more CPU than delivering back local requests, since it involves more processing at the lower layers of the GC stack. The growing CPU usage comes at the expense of local client request processing, thus slowing it down. Additionally, this factor might also contribute to the growing delay of the TOTEM token at each server, thus further increasing GC RTT.

Last, we consider the mixed case throughput. As Figures 5.10 and 5.7 show, the cluster throughput nearly doubles (compared to the write test) when using a mixture of 25% updates and 75% queries, with the increase clearly attributed to the inclusion of queries. However, Figure 5.6 shows that the throughput degradation of updates with increase in cluster size still dominates the mixed test results, despite updates being
a minority of the requests. The reason for this is the large difference in throughput between updates and queries.

5.6 Conclusions

There are several conclusions to be drawn out of the performance test. The first is that separating requests into updates and queries and processing queries locally yields a dramatic improvement in performance: 10-fold improvement in invocation latency and 5-fold in throughput. A distributed information system with a relatively small fraction of updates executed by the provider, and a large majority of queries issued by the public, could double its throughput by providing different processing, as is demonstrated by the mixed test. In a more general sense, weaker consistency models, such as the sequential consistency provided by the update/query scheme, should be pursued (depending on application requirements) for the sake of performance, rather than restricting the implementation to linearizable consistency, as required by FT-CORBA.

Another important conclusion deals with work allocation for each server. The concept of equally sharing any fixed client load over a growing number of servers has been shown to be a negative factor that decreases throughput. Thus, a good strategy would be to adjust cluster size to client load, that is, use large clusters only if the number of concurrent clients is expected to be large as well. Alternatively, in FTS, the number of servers that actually serve clients in the cluster can be reduced (e.g., using the LMR service). Figure 5.12 shows a cross-reference test-case where 2000 clients are accessing a growing cluster of 2-8 servers out of which only 2 servers actively serve clients. As the reader can see, the throughput degradation is nearly gone, which is expected due to much-increased wait-time utilization and much smaller GC RTT. Note also that the amount of remote messages at each of the active servers is small (only half the total messages), leading to reduced resource requirements from the GC subsystem and thus better throughput.

On a more general note, service redundancy may not necessarily lead to increased throughput if request processing includes long wait-states and the number of concurrent clients is bounded. Consider, for example, gateways that mediate between different middleware domains (e.g., .net and CORBA). Each gateway operation basically involves a short operation of translation, then sending the request and waiting for reply, then translating back the reply. Thus, if the delay between request and reply is long, adding gateways to serve the same amount of clients would only reduce each gateway’s throughput but not change the invocation latency. This suggests another aspect when considering cost-effectiveness of hardware investments.
Figure 5.12: Write test: 2000 clients vs. 2, 4, 6, 8 servers with only 2 active servers

Our last notable observation is that the threading model limitation is a general limitation that affects FTS throughput due to the fact that FTS operates at the service level, i.e., above the broker, and is therefore subject to its imposed threading model limitations. Note that lower-level interception and integration solutions may not suffer from such a problem at all since at those levels the replication mechanism can construct and utilize an efficient threading model of its own, where there is no strong coupling of a request and a thread. In Section 8.2.1 we further discuss this issue as part of middleware support for portable replication in general. Also, in Section 8.3.3 we propose a future improvement for FTS that would aid in improving FTS throughput without efficient threading support.
Chapter 6

Adaptive Batching

In this chapter we present and study Adaptive Batching, a protocol-independent efficient scheme for batching (or packing) multiple requests into a single message.

6.1 Characterizing The Problem

FTS employs an underlying group communication component that features an atomic broadcast (ABCAST) service for relaying update requests received from clients by each server to its cluster peers. Many previous works [10, 17, 41, 79] show that batching (or packing) multiple messages into a single transport frame is an effective technique for reducing per-message processing and transmission overhead, resulting in much-improved throughput\(^1\).

In FTS, this gives motivation to use batching of multiple requests into a single ABCAST message as a simple means to increase its throughput. Following FTS design guidelines for portability, the batching mechanism is required to be independent of the underlying GC protocol. One such protocol-independent method is \textit{time-based} batching [25, 41], where requests arriving inside a predefined time-window are batched together. Another method is \textit{count-based} batching [17, 25, 79] that batches arriving requests up to a predefined threshold.

Batching multiple requests into a single message requires that requests be buffered until the actual batch send event, which may not be imminent. Obviously, the longer

\(^1\)The method commonly used for estimating throughput is to measure the time it takes for the ABCAST mechanism to deliver a predefined or a measured amount of messages. Thus, batching improves ABCAST throughput by making it deliver more messages in less time. Alternatively, batching can be considered as a means to improve the maximum message arrival rate that the improved ABCAST can handle without losing messages.
the time interval between sending of batches, the larger the batches are expected to become since more requests are likely to be added to the batch. As [17] shows, batching reduces per-request latency (compared to no-batching) by amortizing processing and transmission overhead latencies generated, e.g., by packet framing, routing and delivery. As a result, the throughput of the system may increase considerably.

However, as noted in Section 2.3, the analysis of [17] does not consider the request arrival rate. Delaying the sending of buffered requests adds a time overhead of waiting intervals between consecutive request arrivals and between the last arrival and the batch send. This overhead is manifested in a factor we call cast latency, which is the average waiting time between the request arrival and the send of the batch that contains it. Thus, increased batching intervals would result in increasing cast latency. At a low request arrival rate or between bursts [25, 10], this can result in a smaller throughput increase compared to e.g., those predicted in [17], or even a decrease of throughput. The reason is that in such cases, the additional delay does not increase the size of the batch at all, or by too little, compared to the added time. This results in a dynamic trade-off between cast latency and batch size: how long should a batch be delayed so that large batches can be created without the cast latency growing too large as well, resulting in large throughput increases.

6.1.1 The Need For Adaptivity

In FTS servers, under real-life conditions, client requests arrive often in a non-uniform pattern, assuming, e.g., bursty behavior [25]. Consequently, selecting when to send a batch should be dynamic in order to maintain good throughput. Simple time-based or count-based batching, where a batch send event is set to occur after every fixed time window or after the batch reaches a fixed threshold, are too rigid. As mentioned earlier, cast latency might become excessive compared to the request rate in either methods, especially when the request rate is low. Setting the time or size thresholds too small to avoid excess delay would yield low throughput when the request rate grows as a result of minimizing the batching effect. Hence, such methods require tuning when being applied within real-world services with varying request rate. Such tuning can be done manually by an administrator or automatically, for example using a feedback loop as suggested in [10].

Furthermore, count-based batching might not even guarantee progress, e.g., in cases where the number of independent request arrivals is bounded. This requires augmenting the basic count-based mechanism with a backup timeout to trigger premature batch send. Such a timeout would have to be large enough to ensure that the threshold can be achieved whenever possible. Thus, the problem of tuning to avoid excessive delay
remains, with the added cost of tuning the backup timeout as well as the size threshold\(^2\).

Some batching mechanisms [41] assume additional knowledge of the underlying protocol in order to choose when to send a batch so as not to incur additional latency. This is not the case in FTS, however, due to the protocol independence assumption. Thus, we seek a batching mechanism that essentially adapts its behavior to the changing request rate without external intervention. Furthermore, this mechanism should be protocol-independent and, preferably, it should require minimal configuration.

### 6.1.2 Model of Operation

FTS servers, as typical CORBA applications, handle multiple concurrent client requests through multi-threading. Each request is assigned a single thread that handles it until completion. Consequently, threads carrying requests arrive at the batching mechanism concurrently and independently. Each thread invokes the `cast()` method that adds the thread’s request to the current batch. At some point later, an event defined by the batching mechanism triggers the sending of the current batch and then resets it for new requests.

### 6.2 Adaptive Batching

The `adaptive`\(^3\) batching (AB) mechanism, as shown in Figure 6.1, operates as following: each thread executes a common code path before invoking `cast()`. A part of this code path ending at the send message primitive is set as a `doorway`\(^4\). The entry code in the beginning of the doorway is used for registering the thread executing it as being inside the doorway, e.g., by increments the doorway’s counter. Similarly, the exit code un-registers the executing thread, e.g., by decrementing the doorway’s counter. AB utilizes a doorway as following; the doorway entry code is inserted at some point that precedes

---

\(^2\)One could argue that good throughput over a large range of request rates can be achieved by combining both time and count limits, for example using a fixed large count-based threshold along with a short time-out. For low request rates, the timeout would ensure short delays, whereas for high request rates the large request count would ensure large batches. However, this would result in a mechanism whose performance requires a large amount of testing and tuning and that is highly implementation- and even deployment-dependent.

\(^3\)The term “adaptive” has a different meaning when applied to distributed algorithms in general, such as adaptive mutual exclusion [6]. We chose to use this term, though, for the literal meaning of the word.

\(^4\)The term "doorway" is used here in the same meaning as, e.g., in Lamport’s Bakery algorithm [56], as it operates on the same concept of a code segment that registers the threads inside it.
the `cast()` invocation. The `cast()` method itself serves as the exit code. If a thread executing `cast()`, after un-registering itself and adding its request to the current batch, finds that no other thread is in the doorway, then it triggers the batch send event.

The intuition behind AB is that multiple requests should be included in the same batch only if they arrive “close together” in the `cast()` primitive, in the sense that the threads carrying these requests closely follow each other in when invoking `cast()`. The doorway serves as a natural proximity sensor: the time required for a thread to cross it is a threshold. Should no other threads enter the doorway by that time, the batch is complete and sent, since the next threads’ requests are too far behind to join the requests in the current batch.

The length of the doorway can be used to control the sensitivity of the mechanism: longer doorways cause long batches to begin forming at lower client loads than shorter doorways, since it becomes more likely that another thread would enter the doorway before the current thread exits it. Note, however, that the adaptive nature of the mechanism remains for every doorway length, as is later demonstrated.
6.3 Timed-Adaptive Batching

AB suffers from several shortcomings. First, there is the problem of finding and setting a common code-path to serve as a doorway. The absence of a long-enough common code-path can be circumvented by deploying multiple copies of the same entry/exit mechanisms along different code-paths. However, implementing AB still requires careful code analysis (to ensure that threads actually pass through the entry/exit codes) and code modification. Next, the length of the AB doorway is set at coarse instruction granularity, making it hard to clearly evaluate the length since it is both code- and platform-dependent. Also, possible doorway length is bounded by available code-path lengths. Last, implementing AB clearly requires a multi-threaded processing model, as the doorway works by detecting advancements of multiple independent execution contexts.

All the above problems are overcome by another AB variant named timed adaptive batching (TAB). The key difference between AB and TAB is that TAB employs a timer to serve as a doorway instead of a code-path. The timer is reset each time a thread invokes `cast()`. If the timer reaches a timeout, which is analogous in AB to a thread exiting the doorway with no other thread inside, the batch is sent.

A disadvantage of TAB compared to AB is that AB allows an immediate send of batch after the last thread of the current batch arrives with no other thread in the doorway. In TAB, on the other hand, the timeout still has to elapse after the last thread, thereby forcing a minimum non-zero additional latency even in the case of a single request in a batch. However, the TAB timer allows to define the doorway length in time units regardless of the implementation. Last, unlike AB, a timer-based implementation can also be used in a single-threaded event-based environment.

6.4 Performance Evaluation

To test the properties of the adaptive batching variants, we used FTS, deployed in the same testing environment as described in Chapter 5. To simulate varying client request rate, we used varying amounts of client loads, ranging from 100 clients to 2500 clients. The server cluster consists of 4 servers.

Following [41, 79], we sought short requests, where the impact of batching should be most explicit. However, since FTS operates at the CORBA application level, we cannot control the exact byte-size of a serialized request buffer. Also, note that the serialized buffer contains additional FTS data such as the request id. So, as a short invocation, we used the `set_greeting()` method of the target Hello object with a string
Figure 6.2: Performance Evaluation: Throughput vs. Client Load
בידיקה ביצועים: תפוקה ביחס ל עומס לומות

Figure 6.3: Performance Evaluation: Cast Latency vs. Client Load
בידיקה ביצועים: השהית קובים ביחס ל עומס לומות
Figure 6.4: Performance Evaluation: Batch Length vs. Client Load

בידיקת ביצועים: גודל אוטומטי ביחס לעומס לקוחות

A parameter of 15 characters. Such an update request is very small in CORBA terms but still translates to a buffer of 280 bytes. As we later show, batching still proves highly effective for requests of this size.

As for batching mechanisms, we used 5 different types. The first two are the classic methods mentioned before, namely Time-based Batching (TB) and Count-based Batching (CB). Next are the adaptive variants, code-oriented Adaptive Batching (AB) and Timed Adaptive Batching (TAB). Last, we also tested the case of No Batching (NB) as a base reference.

Figure 6.2 shows a comparison of the throughput of FTS resulting from applying the various batching mechanisms, each with multiple values of parameters. Figure 6.3 compares the average per-request cast latency imposed by the batching mechanisms. Figure 6.4 shows the resulting average batch length for each of the mechanisms. Each mechanism’s graph is marked by the corresponding acronym followed by the value of its parameter. For example: ”TB-30” marks a test of the TB mechanism with a time window size of 30 msec. Note that in each Figure, the graphs of the adaptive batching mechanisms are on the right pane, whereas the rest of the graphs are on the left pane, in order to avoid a crowded view.

The TB mechanism batches all requests accumulated over a predefined time window size. It is implemented by having a separate timer-thread wake up after each window elapses and send the currently accumulated batch. We tested a wide range of window sizes and present a few of the more characteristic exemplars, large window sizes vs. small window sizes.
Th CB mechanism batches requests up to a predefined threshold. It is implemented simply by keeping count of the requests in the current batch and sending the batch after the last request is added. The backup timeout is implemented using a time-thread that is reset when the first request is added. For each of the CB thresholds demonstrated, we calibrated the backup timeout to be the lowest value we found that guarantees that the resulting batch size is very close (5% diversion) to the threshold at most of the client loads. As with the TB mechanism, we present the effects of both high and low counting thresholds.

The code-based AB variant was tested with doorway lengths that were determined according to the structure of FTS code. All the threads that handle requests in FTS execute the same code-path prior to invoking cast(). This code-path turns out to be quite short in terms of execution time. Thus, we defined only two lengths of doorway. The short doorway is implemented by putting the entry code just before the exit code, i.e., upon the invocation of the cast() method. The long doorway simply wraps the entire code-path by placing the entry code at its earliest beginning, which is the point where the client request is first determined to be an update by analyzing its signature.

Contrary to AB, the TAB variant’s doorway length is defined in milliseconds. Note, however, that in the following graphs, the values used for testing are quite small, e.g., compared to the TB window size. This is because the doorway length determines a maximum time-distance between the arrivals of two consecutive requests, whereas the TB window size sets the entire interval required for the batch.

6.5 Discussion

Figure 6.2 clearly demonstrates the aforementioned rigidity of the classic CB mechanism. For low client loads, using a low threshold (10 request per batch) for CB yields throughput that is good in the sense that it is close to the maximum we were able to obtain for that load using any of the mechanisms and parameters. However, under high client load, throughput drops considerably. An opposite effect is seen when using high thresholds (100 requests per batch). The fixed threshold behavior is presented in Figure 6.4, showing the expected near-constant average batch length in all client loads except the very smallest. The average cast latency for CB tends to start high and decrease with the growing client load. This should be expected since as the client load grows more CB batches are sent because of reaching the threshold, rather than because of the timeout.

A similar rigid behavior is also seen in TB graphs. TB with a 100 msec window behaves similar to CB with a threshold of 100. Also, TB with a 30 msec window shows good throughput (same as e.g., TAB-5) in small client load but its throughput is
definitely not the best at higher client loads. Also, TB batch length (Figure 6.4) grows with the client load. This is not surprising considering that the higher the client load, the more requests are likely to arrive during the same time interval.

Specifically, note that TB, when using small windows, seems to yield better throughput under high client loads than CB with small thresholds. The explanation for this lies in the implementation. The code inside \texttt{cast()} that adds a single thread's request to the current batch is defined as a mutually-exclusive critical section. The timer-thread has to lock this section to prevent new requests from being added until the current batch is sent and reset. Under high load, the timer-thread scheduling is often delayed because of the large number of concurrent threads. As a result, more requests are added to the batch than there should have been allowed for the given window size\textsuperscript{5}. The support for this explanation is given by Figure 6.3 showing that the average TB cast latency under high load is often higher than the window size. If the implementation did not suffer from delayed closure of batches, the cast latency should have remained very close to the window size.

Both AB and TAB show the expected adaptive behavior, as can be seen in Figures 6.3 and 6.4. Both batch length and average cast latency start low and grow with the client load. However, Figure 6.4 indicates the code-dependency limitation of AB. Profiling shows that the longest code-path doorway that could be defined is still very short in terms of computation time (less than 1 millisecond). Thus, the average batch length is small compared to the other mechanisms under the same load. TAB, on the other hand, as being able to work with much longer doorways, obtains much better throughput and truly proves the advantages of adaptivity, since it constantly maintains near-maximal throughput at all client loads.

TAB's remarkable ability to maintain high (or even peak) throughput at all loads compared to the non-adaptive mechanisms can be explained through Figure 6.5, which plots the ratio of batch length over cast latency as a function of client load. Note that in this Figure, the behavior of each batching mechanism's graph shows great similarity to the respective throughput graph of the same mechanism in Figure 6.2. This can be expected by considering the cast latency as an indication of the total batch accumulation time, in the sense that a batch that takes a long time to construct, will most likely yield a high cast latency for its requests, and vice-versa. Therefore, a high value of the batch length to cast latency ratio will likely indicate that more requests are batched together in shorter time intervals, which results in a higher throughput.

As shown in Figure 6.5, all TAB graphs maintain a narrow range of high ratio at

\textsuperscript{5}To reduce this effect we added code that lets the request threads themselves read the clock and close the batch if past window size. However, the effect remained in a lesser degree, most likely because of clock inaccuracy problems.
all loads, whereas the non-adaptive graphs tend to oscillate over a wide range of values, and specifically achieve low ratios at either low loads (for high thresholds, e.g., CB-100 and TB-100) or at high loads (for low thresholds, e.g., CB-10 and TB-30). The TAB graphs behavior results directly from TAB’s tendency to batch together only requests that arrive close to each other, thereby achieving a high ratio value. In comparison, non-adaptive mechanisms tend to either include unnecessary delay in low loads or split bursts of close arrivals at high loads, which results in a low ratio. Therefore, Figure 6.5 presents adaptive behavior as a major contributor to obtaining good throughput under various client loads (or dynamic request arrival rates), as demonstrated in the TAB mechanism.

Note that for TAB, batching incurs a fixed additional time overhead to the cast latency whose length is that of the doorway, as a result of waiting for no additional requests to arrive before sending the batch. Thus, by specifying excessively long doorways for TAB, it begins to lose its efficiency at low client loads, similar to TB and CB. A good example for this is presented in Figure 6.2 where the throughput of TAB with a doorway of 30 m/sec is lower than with shorter doorways (5 and 10 m/sec). AB does not suffer from this problem since it incurs no time overhead beyond the inclusion of the last request in the current batch.
6.6 Conclusions

In this Chapter we presented adaptive mechanisms for batching messages (requests) together prior to sending them through an ABCAST transport layer. Earlier works either focus on the benefits of batching in general or try to maximize batching throughput in dynamic environments by constant measurement and calibration. Our work, on the other hand, is based on the observation that one of the key factors for obtaining good throughput under changing message rate is how well the batching is suited to the message rate so that it does not incur excessive batching latency. Consequently, we defined two adaptive batching mechanisms, code-based AB and time-based TAB. We showed the effectiveness of adaptivity by comparing the throughput of these two mechanisms with the classic time-based and count-based mechanisms, under a wide range of request arrival rates. Furthermore, the behavior of both adaptive mechanisms is controlled by a single parameter, the doorway length. Yet, adaptivity and the resulting good throughput are maintained for a large range of doorway lengths, making the mechanisms relatively easy to configure.

Both AB and TAB can also be combined with other batching mechanisms to achieve additional qualities. For example, in a model where the number of independent message arrivals is unbounded, neither AB nor TAB guarantee progress. That is, a buffered message might never get transmitted inside a batch. In more common scenarios, neither of the adaptive variants provides predictable batching latency, so neither can provide bounded overall transmission latency even if the transport latency is bounded. Both issues stem from the fact that the batching latency depends on the temporal amount of closely following request arrivals. Progress can be guaranteed by adding a timeout or a threshold limitation, albeit this re-introduces rigidity into the algorithms, especially during high message rates. Specifically, a timeout would also guarantee predictable batching latency.

Last, batching in general, and specifically adaptive batching, can also be applied in many contexts that include a transport medium with a high per-transmission cost, in order to increase the transport throughput. For example, disk access is an expensive operation, which is why operating systems commonly cache disk contents. In case of a write-through cache accessed by a varying number of writers, applying adaptive batching of write operations for the same file or memory page may increase the write throughput. For another example, consider a web cache that serves multiple users who are writing to the same wiki [62] site. The per-write cost in this case originates from constructing a TCP connection for each HTTP update. Again, adaptive batching can help to efficiently pack multiple updates in the same TCP connection.
Chapter 7

Reducing FTS Server Memory Consumption

This chapter presents a case of reducing the size of the memory footprint of FTS servers by using selective acknowledgments to accelerate removal of executed requests from the cache.

7.1 Motivation

The request/reply cache (or request log) is a fundamental component of server replication [12, 90, 77] and also of other distributed services [114, 102] that need to maintain at-most-once request execution safety. It stores the client requests received by the cluster. When a client issues a request to a server, the server checks the request ID at the cache to determine whether it has already processed that request. If so, the server returns the matching stored reply instead of re-processing the request. This ensures that a server processes the same request at most once. In FTS, the cache is used only for messages tagged as updates, since (re-)executing queries does not modify the service state.

The cache is also one of the most memory-consuming elements of an FTS server. Every request received either from clients or from the ABCAST transport are stored in the cache in every processing stage. In comparison, most of the other internal data structures of FTS contain only references to requests stored in the cache, and each structure references only requests that are in one particular stage. Excessive cache growth might eventually impede performance due to memory swapping and, in Java, dynamic memory management overhead. Furthermore, once memory is exhausted, servers might
crash and service might be disrupted\textsuperscript{1}. Hence, slimming down the cache may increase both FTS performance and scalability\textsuperscript{2}.

In FTS, clients may perform multiple independent invocations on a server object using different threads or asynchronous invocations, just as in standard CORBA. As a result, a new request arriving from the same client cannot be assumed to implicitly acknowledge receiving replies for any previous requests that have finished processing. Furthermore, the CORBA invocation model does not provide FTS with any indications of a successful delivery of replies to clients. Thus, requests are removed from the cache based on a globally known time-limit after which the request is assumed not to be re-issued\textsuperscript{3}. Accordingly, a dedicated cache daemon thread purges the cache periodically from requests whose time-out has expired.

Several ad-hoc solutions might be considered for reducing the cache size. Shortening the timeout could be effective but is limited to a point below which at-most-once semantics can not be guaranteed. Also, adding purging cycles might remove requests with greater time-out accuracy but would also require more CPU, which is likely to degrade performance without considerably reducing the cache since the timeout remains the limiting factor. Hence, we considered adding information to the purging process to make it possible to remove stored requests from the cache even before the timeout without violating safety conditions of at-most-once and at-least once request execution.

### 7.2 Implementing SACK in FTS

#### 7.2.1 Description

We propose using selective acknowledgments (SACKs) as an effective means to accelerate request removal from the cache. On the client side, SACKs are transparently handled by the FTS PI, which collects request IDs of replies and attaches them as SACKs to subsequent invocations that target the same object the replies were received from. This also ensures that SACKs of multiple independent invocations, such as by multiple client threads or using asynchronous invocations, are also collected and sent in a subsequent request from the same client to the same object. Thus, even if a client thread interacts with one service object and then with another, the SACKs that were collected from its invocations of the first object may still be sent along with a different thread’s invocation.

\textsuperscript{1}In fact, the motivation for working on this problem began after noticing that FTS servers crash-fail under high loads due to excessive memory requirements.

\textsuperscript{2}Of course, it is possible to store some of the cache’s contents on disk in order to save memory, but this entails a significant latency.

\textsuperscript{3}To make this mechanism more flexible, FTS allows defining per-operation time-outs.
FTS also allows to accumulate SACKs according to the target replication context group (which may contain multiple objects) instead of only the target object in order to further increase SACK coverage. That is, if a thread invokes an object inside a particular context, the request carries SACKs of requests invoked by any thread at the same client that previously invoked an object in the same context.

On the server side, SACKs are handled inside a SackHandler object in each GOA. This object holds two SACK stores called arrived and cast. The arrived store contains SACKs that arrived from clients and are pending ABCAST. The cast store contains SACKs that were delivered back from ABCAST and need to be processed. The SackHandler object is operated by a dedicated daemon thread that periodically (once per second) collects and sends the arrived SACKs in batches, and then processes cast SACKs by locating their matching requests in the cache and tagging them as removable. The actual removal of a request takes place only after the request is done executing. After being processed, the cast SACKs are discarded. Further implementation details can be found in [45].

7.2.2 Rationale

In many common distributed applications, such as a database service, a directory service, etc., a client interacts with a service object using a sequence of invocations rather than a single invocation. Thus, subsequent invocations can be used by the client for acknowledging successfully-received replies of previous invocations. Consequently, on the server side, acknowledged requests can be discarded from the cache. This concept resembles the technique used by TCP [113] and SACK TCP [68] to remove acknowledged segments from the sending window. However, unlike TCP or SACK TCP, FTS cannot use the more space-efficient sequence ACKs, only selective acknowledgments (SACKs). The reason is that FTS is designed to take advantage of third-party mechanisms for generating unique request IDs, wherever such mechanisms already exist in the client ORB, such as FT-CORBA’s FTRequestServiceContext [90]. Such mechanisms guarantee request ID uniqueness but do not impose any order or other relation between IDs, which is required for regular ACKs. In other words, acknowledging one request ID can not be used to acknowledge additional request IDs that might have preceded it.

Sending SACKs through the ABCAST transport is justified as following. FTS does not assume clock synchronization between a client and any server, so the client PI does not expunge collected SACKs that have become invalid due to expired time-out at the servers. Therefore, a SACK that arrives at a server may refer to a request that is not in the cache, either since that request has not been delivered yet or since it has already been removed from the cache. With the server being unable to determine which case,
the SACK is simply discarded with no additional effect. However, in order to ensure that a SACK is not discarded before the request it refers to is delivered to the server, SACKs are not processed upon reception from a client, but rather are sent over the intra-cluster ABCAST transport and used only when delivered back from it. Since the SACK, by definition, acknowledges a request that has already been delivered in at least one server, the total-ordering property of ABCAST ensures that the SACK is delivered after the matching request in all the servers in the cluster. The ABCAST propagation of SACKs further ensures that all the servers receive the SACKs and use them to shrink their caches.

Note, however, that this method of SACK processing also results in an overhead in the form of ABCAST messages that are not used for relaying requests among the servers. This overhead is both in bandwidth and in message processing (sending and receiving messages), and is added to the processing overhead of the SACK daemon thread. The SACK message processing overhead is aggressively reduced by batching multiple SACKs together into SACK ABCAST messages, using simple time-based batching with a window size of 1 second\(^4\). This delay is still much smaller than the request expiration limit which is set to 10 seconds.

From an operational correctness point of view, SACKs are merely an optimization to the existing timeout-based purging mechanism. If a request has not been processed by a server yet, it may be tagged as removable by a SACK but it will not be removed from the cache until is is done processing. Also, clients send SACKs only after receiving a correct reply so a request removed through a SACK is guaranteed not to be needed again. Last, SACKs that refer to non-existing requests are simply discarded with no other effect. Therefore, these operating rules do not violate the underlying safety conditions of each request being processed at least once as well as at most once by each server.

7.3 Performance Evaluation

We used FTS to test the performance impacts of the SACK implementation. FTS is deployed in the same testing environment as described in Chapter 5. We tested SACK under varying amounts of client loads, ranging from 500 clients to 2500 clients. The server cluster consists of 4 servers. We used short requests, where the overhead of ABCAST bandwidth should be close to maximum. The invocations are of the set_greeting() method of the target Hello object with a string parameter of 15 characters.

\(^4\)Actually, in our tests, SACK batches is also limited to 5000 SACKs each due to Ensemble daemon settings.
Figure 7.1: Performance Evaluation: Cache Size vs. Client Load

We conducted two simple sets of tests, one with SACKs and one without SACKs. The measurements that we used to compare the results are FTS throughput, memory usage and overhead estimates. Memory usage data consists of JVM allocated memory and cache size (counted in request records). This data is averaged over a sequence of read-outs performed once every two seconds during the test phase. SACK overhead is estimated as SACK percentage of total traffic. Thus, SACK bandwidth overhead is estimated in byte percentage and SACK message processing overhead is estimated in message percentage. The processing overhead of the SACK thread is harder to evaluate, so we use the throughput measurement as a global indicator for the entire processing overhead.

Figure 7.1 compares the cache sizes between runs with and without SACKs. Figure 7.2 shows a similar comparison of total server memory consumption. Note that the scale of 7.2 is in percents of the JVM memory allocation limit. The top allocation limit is the same in all runs (1GB) but is not relevant and therefore omitted. Next, Figure 7.3 shows FTS throughput with and without SACKs. Last, Table 7.1 shows the SACK overhead of ABCAST traffic in bytes and messages.
Figure 7.2: Performance Evaluation: Server Memory vs. Client Load

Figure 7.3: Performance Evaluation: Cluster Throughput vs. Client Load
<table>
<thead>
<tr>
<th>Client Load</th>
<th>Message Overhead (%)</th>
<th>Byte Overhead (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>500</td>
<td>1.31</td>
<td>32.9</td>
</tr>
<tr>
<td>750</td>
<td>1.23</td>
<td>32.9</td>
</tr>
<tr>
<td>1000</td>
<td>1.29</td>
<td>32.9</td>
</tr>
<tr>
<td>1500</td>
<td>1.25</td>
<td>32.9</td>
</tr>
<tr>
<td>2000</td>
<td>1.11</td>
<td>32.9</td>
</tr>
<tr>
<td>2500</td>
<td>0.93</td>
<td>32.9</td>
</tr>
</tbody>
</table>

Table 7.1: Performance Evaluation: SACK Overhead vs. Client Load

7.4 Results Analysis

The results for the cache size, as shown in Figure 7.1 show that applying SACKs reduces the cache size by at least a factor of 2. In loads of 750 and 1000 clients the cache size is reduces by a factor of 3, which means that SACKs are indeed very effective.

An additional evidence of the positive impact of using SACKs is presented in Figure 7.3 that shows an improvement of about 20% in cluster throughput in high client loads. This means that despite the added processing and network overhead imposed by SACKs, the overall contribution improves not only server scalability but also performance. The throughput improvement can be mostly attributed to two factors. First, SACKs are handled by a daemon thread, i.e., a thread that does not serve client requests. This may actually increase processor utilization by running in the idle periods that are the result of client-serving threads waiting for ABCAST delivery, as explained in Chapter 5. Second, the accelerated out-of-order removal of requests from the cache decreases the work of the cache purging thread, thus decreasing its contention with client threads that results from cache access synchronization and CPU utilization.

In the more general case (i.e., not necessarily FTS), any change to throughput following SACK usage may depend on the cache and SACK implementation. However, it seems reasonable to expect that applying SACKs would cause at worst a small decrease of throughput (due to the additional SACK processing), with each stored request being accessed and removed only once, either because of a SACK or because of a time-out. On the other hand, SACKs may also cause a positive throughput improvement through reduced cache contention and reduced memory swapping.

When comparing Figures 7.3 and 7.1, there are a couple of observations to be made. First, one can easily verify that the size of the cache when not using SACKs indeed matches the product of the cluster throughput and request expiration timeout (10 seconds). Second, the largest reductions of cache size occur at loads in which the cluster
throughput is higher with and without SACKs compared to other loads. This can be explained as following. Each request occupies cache space as soon as it arrives at the server from the client/ABCAST transport, and stays there until being removed. Later, a SACK that matches the request arrives from the same client as that of the request. The higher the throughput, the lower the latency until that client sends the SACK attached to the next request. Thus, the time each request gets to stay in the cache is reduced with throughput increase.

Figure 7.2 clearly shows how applying SACKs reduces the overall memory consumption of the server by at least 50% for the client loads below 1000, and at all loads in no less than 40%. Also, following the previous observations, note that the larger memory reductions occur in the same client loads that exhibit higher throughputs with and without SACKs. This can be explained as following. The SackHandler causes removal of requests at the same rate as it accepts and processes SACKs, which matches the throughput. Therefore, the SackHandler on average contains as many SACKs as there are requests in the cache. Since each SACK consumes much less memory than the request it removes (the size of only the request ID vs. the size of the entire request object) and the cache shrinks at least by half (Figure 7.1, the overall effect is substantial memory reduction. Another observation arising from Figure 7.2 is the considerable weight of the request/reply cache in the total server memory consumption, which is due to our test objects consuming little memory.

Last, we consider the message and bandwidth (byte) overhead as presented in Table 7.1. As this Table shows, the bandwidth overhead is fixed at approximately 33%. This is actually expected, since the number of SACKs received by each server is roughly the same as the number of requests. Thus, the ABCAST byte overhead roughly matches the ratio between the average size of a serialized SACK (approx. 90 bytes) and the average size of a serialized request (about 280 bytes). Note that since we explicitly used short requests in the test, the measured byte overhead is close to the maximum and will become much smaller when using typical requests with many more parameters and complex data types. The message overhead, on the other hand, grows smaller at higher loads. This can be understood when considering that SACKs are time-batched (see Chapter 6), so at higher client loads more SACKs get packed in less batches, yielding lower SACK message overhead. This also reduces the total processing overhead involved in SACKs, which can explain why the throughput improvement grows with client load as shown by Figure 7.3.
7.5 Conclusions

In this chapter we showed the applicability of SACKs as an effective means to reduce the memory consumption of the request cache in an FTS server, and, consequently, that of the entire server. This work is based on an underlying observation that the request cache essentially resembles a sending window in sliding window protocols such as TCP, instead of a regular cache. Thus, it can use similar means, such as SACKs, for evacuating stored data.

After designing and testing the SACK implementation in FTS, we realize that SACKs are indeed a worthwhile modification. In addition to being a safe optimization, SACKs’ incurred overhead is outweighed by their contribution to both server scalability and performance. Furthermore, the message and bandwidth overheads associated with SACKs are expected to grow smaller at higher client loads and with more typical usage that results in larger requests.

The proposed method of propagating SACKs to different cache replicas is a derivative of FTS architecture, where each server is responsible for relaying information received from its clients to other servers. Thus, it cannot be applied as-is to, e.g., FT-CORBA-compliant active replication, where a client communicates with all the servers through a proprietary protocol that guarantees request atomicity. Still, it is applicable to the warm replication style, where each primary server propagates client data/state updates to the backup servers. As a general concept, SACKs should be attempted as a means to improve scalability in various middleware implementations such as RPC [114] implementations that maintain at-most-once semantics by caching requests and using time-outs to purge the cache. SACKs may also be especially applicable to modern 3-tier applications, in which most of the state is stored in the 3rd layer (the database) making the reply cache of the 2nd layer a dominant memory consumer.
Chapter 8

Summary and Future Directions

In this chapter we conclude the outcome of this work. In the first part, we present a short summary of the results. Next, we discuss two lessons acquired through this work. The first lesson refers to what provisions should a middleware support for portable replication. The second lesson is a comparison between FTS and FT-CORBA, which is currently, to the best of our knowledge, the only standard middleware specification that deals with fault-tolerance. Last, in the future directions section we list a few key possibilities out of the numerous directions at which this work may evolve further.

8.1 Summary of Results

In this thesis we have presented FTS, a fault-tolerance service. FTS embodies our proposed architecture for adding fault-tolerance capabilities to middleware services. Contrary to previous works, FTS does not try to improve any particular classic approach at this problem, but rather merge together advantages of all approaches in a single design. Consequently, FTS offers a unique combination of properties that includes portability and interoperability of a service, transparency and generality of an interceptor and the efficient group communication of an integrated component. In addition, the minimally-intrusive nature of FTS offers an uncommon ability to completely intermix its clients and servers in a standard middleware deployment, leaving replication as an added aspect that is nearly-orthogonal to the application.

Designing and building FTS was and remains a difficult task, as it is a living and growing project. While trying to maintain an external impression of a simple black-box that handles all aspects of replication, the GOA of FTS is highly complicated internally, combining functionalities of failure-detection, request processing and service
configuration management. Still, the motivation that stands behind this architecture is that although fault-tolerance must be addressed within the scope of the middleware, it should be easier to engage for simpler applications, up to near-transparency.

Apart from its architecture, FTS addresses issues that are usually not covered by parallel works of middleware fault-tolerance. The first is support for multiple consistency modes. This allows consistency that is both weaker than the typical linearizable consistency and at larger granularity than that of a single object. The weaker sequential-like consistency provides for better performance than linearizable consistency, as indicated in Chapter 5. Larger granularity support, on the other hand, extends the supported programming model to match that of the standard un-replicated middleware in the sense that it allows application designers to define fault-tolerant middleware objects that are partially inter-dependent, of even different facades of the same implementation.

The second uncommon issue addressed by FTS is maintaining client-server communication in the presence of changes to service configuration. While many GC-like implementations provide for consistency between replicas under changing configurations, we have found no other works that provide an integrated fault-tolerant solution for client-server connectivity in the presence of both network partitions and complete changes of service layout. FTS provides a unified solution to both server consistency and client-server connectivity in the form of the LMR sub-service. This decouples client and server administration and provides great flexibility to service administrators in deploying large-scale services across WANs. Although a standard naming (or a directory) service may seem as a simple solution for the redirection problem\(^1\), tight coupling is required between the redirection solution and the actual service deployment in order to ensure client redirection to a primary component within a network partition.

A careful test of FTS performance provides several insights that pertain both to FTS, to service-level replication and to services in general. First, it justifies our assumption that weaker-than-linearizable consistency provides considerably better performance. This makes the implementation of FTS, which accelerates queries over updates, especially relevant for distributed fault-tolerant information services, such as yellow pages and stock-exchange consoles, which require only sequential consistency [120]. Second, it refutes the notion that adding more servers to a cluster that serves the same amount of clients would increase performance, since for services that include long wait periods, latency may not improve by parallelizing request service. Last, it points to an inherent throughput limitation that results from constructing the main FTS logic above the broker, as do all middleware services. The tight coupling of requests and threads limits throughput when the thread pool becomes exhausted. We expand more on the subject of middleware support for replication in Section 8.2.1.

\(^1\)as is suggested for example in OGS [31]
Our last two contributions have a more general application range than FTS, although each one’s effectiveness was proven through it. First, we introduced two new adaptive protocols that deal with batching of multiple high-level messages into a single low-level packet. The key properties that distinguish these protocols are that they are communication-independent, i.e., do not assume any specific knowledge of the low-level communication protocol, and that they inherently adapt the batching threshold to the arrival rate of messages so as to gather large batches in small time-windows. Consequently, these protocols (specifically, time-based adaptive batching) yield excellent throughput in a wide range of message arrival rates compared to more naive implementations with fixed thresholds. In principle, these protocols can be applied as a booster layer for communication protocols that incur a high per-message overhead, such as AB-CAST protocols. However, as shown in Chapter 6, they can also be applied for other uses, such as efficient utilization of WAN TCP connections.

The last contribution deals with efficiently reducing the size of the request/reply cache. This type of cache is common not only to FTS, but to any distributed service that wishes to maintain at-most-once semantics when the same request may be re-issued in more than one connection/session, such as RPC, NFS and others. Typically, the evacuation policy of such a cache is time-based, since the client has no obligation to further contact the server after having received a reply, so the server has to rely on timing assumptions in order to remove stored replies. We observe that such a cache essentially resembles the send buffer of reliable point-to-point communication protocols such as TCP. Therefore, by having a client acknowledge (using SACKs) having received previous replies during consequent invocations at a server, replies can be evacuated much earlier than the safety time-out, thereby dramatically shrinking the effective size of the reply cache. The cache size reduction can further diminish general server memory consumption in cases of large replies and/or heavy client load combined with a relatively small application memory footprint. Such a setting is typical for 3-tier applications where most of the service state is stored in a database in the 3rd layer.

8.2 Acquired Lessons

8.2.1 Middleware Support for Replication

During our work on FTS, we have identified several issues in FTS design that require middleware support. One of these issues, namely the portable interception, is mandatory, but support for the other issues is essential for an efficient implementation. More important, these issues are relevant to portable (non-intrusive) replication design in general and to other types of services as well. In this section, we briefly discuss each issue.
and indicate the respective CORBA provisions.

Most of the presented issues stem from the need to adjust CORBA to work with efficient low-level reliable multi-point communication mechanisms, such as GC ABCAST protocols, instead of inefficient multi-point-to-point communication [33]. No mainstream middleware, including CORBA, currently includes multi-point communication in its specification. CORBA is in the process of adopting new multi-point semantics in the specifications of MIOP [100] and ROMIOP [98], but neither of these specifications has matured yet.

**Portable Interception**

At the top of the list stands portable interception, that is, interception ability provided by the broker itself. This capability is required for a replication mechanism to read, write and redirect client/server communication, at the granularity of a single request/reply. Reading a request is required for copying its contents to the replication mechanism. Writing (more precisely, generating) a request is required for triggering processing of requests when they are delivered from non-standard transports. Last, redirection is required for fail-over operations as well as for re-routing requests inside the server, e.g., for the purpose of caching.

The CORBA Portable Interceptors provide very little for efficient replication. Reading request contents is not allowed in certain language mappings including Java, so it cannot be used in a generic solution that is language-independent, such as FTS. Generating requests is completely forbidden in any language mapping, as the interceptors are only notified of requests and replies instead of initiating them. Last, PI redirection is highly inefficient since it requires an exception to be returned all the way back to the client and then a new request to be re-issued at the alternate target.

In FTS, we managed to bypass those limitations inside the GOA by using DSI to read a request contents and DII to generate a new request at the server after it being delivered from the ABCAST transport. The object-adaptor is used for efficient routing of requests inside the server. While the DSI/DII and object-adaptor mechanisms are not intended for interception purposes as the PIs, they in fact provide most of the missing functionality. The portable interceptors themselves are used mainly for client fail-over\(^2\).

Although a stand-alone gateway can provide similar functionality to portable interception, it can crash independently of the client and server and thus needs to be replicated by itself. Moreover, as shown in [32], gateways are not enough and still require some sort of interception to modify the behavior at the server side. A transparent

\(^2\)The service context exchange is discussed in the following Sections.
application proxy attached to clients and/or servers (as suggested in [38]) can also fulfill this demand without separate replication. Still, it is less efficient than a middleware interceptor since each request/reply undergoes additional marshaling and unmarshaling cycles in order to be processed by a proxy, as it operates at the application level\(^3\). Another alternative is modifying the stub/skeleton generation process, such as the “smart proxies” in Orbix+Isis [50], which provides efficient interception points, but reduces portability of the solution, and thus is unsuitable for FTS.

Another requirement of interceptors that is found to be missing in CORBA PIs is the definition of a threading model that is related to interception [37, 43]. The CORBA PI specification does not specify which thread operates the interceptor at each interception point, so it is left to each vendor’s decision. Thus, a vendor might allocate only one dedicated thread to operate the interceptor at all points. As a result, blocking one request at the interceptor level (until it is delivered back from the ABCAST transport, for example) will block all other incoming client requests, thereby eliminating multi-threading and increasing the chance of deadlocks.

**Access To Internal Information**

Although replication is implemented above the broker, it could benefit considerably from access to low-level information inside the broker. In order to put CORBA requests and replies through the ABCAST transport, which is not CORBA-aware, a replication mechanism needs to marshal (or serialize) the sent data at the sender and un-marshal (de-serialize) it at each receiver. If a replication mechanism has access to a request in its serialized network format, then it can simply copy the serialized format to all replicas and have the replicas then de-serialize it as new requests. This is how integration-based and interception-based replication solutions work, which gives them a performance advantage over portable interception (for the price of portability).

In CORBA, there are no standard facilities for accessing low-level CORBA messages at all. Thus, FTS has to rely on application-level facilities of DSI and DII (see Section 8.2.1) to read and generate messages. Consequently, FTS incurs additional redundant cycles of serialization and de-serialization before and after each request gets to execute inside an application object. Also, as it turns out, the CORBA facilities for portable (de)serialization, which FTS uses for transferring requests/replies into and out of ABCAST messages, are poorly implemented even in popular ORBs such as ORBacus [52]\(^4\). During the construction of FTS, we realized that by replacing the standard

\(^3\)Note that this efficiency problem also affects additional applications, such as inter-domain bridges.

\(^4\)The reason for this is probably that portable serialization is under-tested since is of little use for typical CORBA applications.
CORBA serialization with a specific CORBA/Java serialization facility, which is a little faster, we achieved a significant improvement of both request latency and server throughput at high client loads. Crossing this with profiling information, we realized that the large improvement resulted from request threads that not only processed faster, but also spent much less time waiting for other threads. Therefore, we can safely assume that a complete removal of explicit serialization and de-serialization would yield an even more dramatic performance gains.

The CORBA OCI [101] specification, which is not yet officially adopted but seems to be gaining wide acceptance, can solve the problem of access to requests at the low-level message format. Generally, the OCI specification allows portable plug-in of non-standard protocols into the CORBA connection model. As such, it provides interception points that have unlimited access to low-level CORBA messaging. Thus, a possible OCI-based implementation of FTS can simply intercept the arrival of a message containing a request to the server and copy its buffer to the ABCAST transport, much like Eternal [77] does. On the ABCAST delivery side, the OCI interceptor simply copies the received buffer and signals the ORB for a new request. Using the OCI, marshaling and unmarshaling of requests and replies are performed by the ORB, efficiently and only once per request and reply at each server.

Another aspect of of low-level access is the ability to set any possible internal ORB-state that relates to each object during state-transfer. According to our current findings, there is no need for such a mechanism in FTS over CORBA (see Section 8.2.2 for a detailed discussion), but it could be possible for other middlewares that the broker maintains a private internal state for each object, which may affect the broker behavior. In this case, the replication mechanism must gain non-standard read/write access to this internal state so that replica consistency is maintained.

Threading Model

As noted in Section 5.6, the popular threading model of a thread-pool becomes a performance bottleneck when its size cannot be adjusted dynamically to meet increasing client loads. The problem arises from the tight coupling of requests and threads, i.e., that a thread cannot be used to handle another request while its currently-assigned request is not done executing. As a result, services in which requests undergo long wait periods during execution achieve low throughput and high request latency from forced idle processor periods, despite having pending requests. This problem, therefore, affects not only FTS but also, for example, persistent-storage services that block requests due to long disk read/write operations. Therefore, performance of services that include long wait-periods would greatly benefit from middlewares that provide dynamic control of the thread pool size or, even better, a method for de-coupling threads and requests.
The CORBA standard is vague w.r.t. threading. On one hand, it does not specify which threading models should be provided by ORBs or how a thread-pool size should be controlled. Thus, vendors are free to provide their own implementations. On the other hand, CORBA does not define any standard facilities that would allow to decouple a thread from its currently assigned request. As a result, any portable service implementation such as FTS must assume tight coupling of requests to threads and no control over thread pool size.

In middlewares that are integrated into the platform, such as Java (J2EE) and .NET, it is easy to implement a server threading model where the same thread may service multiple requests, since the platform controls thread scheduling and is aware when a thread goes to a waiting state, and can thus be assigned to a different request. CORBA, being a platform-independent middleware, does not provide any facilities to implement such mechanisms. Several ORB versions, such as ORBacus [52] for C++ and TAO [108] offer a proprietary threading library that is used by the server application. This library essentially intercepts thread operation and notifies the ORB when a thread can be re-assigned to another request, either since it is going to a wait-state or by explicit notification. Another cross-platform middleware called ICE [51] provides API through which server threads can explicitly ask to be assigned to a different request.

Last, note that a proposed design improvement in FTS, as appears in Section 8.3.3, aims at solving the thread pool bottleneck regardless of the threading model in use, by decoupling client requests and replies.

Unique Request ID

FTS, as well as any other replication solution, must uniquely identify each request (and its consequent retries) in order to guarantee at-most-once request execution semantics (see Section 4.8). In principle, FTS does not depend on middleware support for this. As seen in Section 3.2.3, pre-FT-CORBA does not support unique identification of requests at a context that exceeds a single communication session between the same client and server. To compensate, FTS can generate unique request IDs on its own and associate them to requests using a transparent client-side proxy as explained in [38]. However, as already pointed out, a dynamic-processing proxy incurs additional overhead by adding cycles of marshaling and unmarshaling. On the other hand, when using FT-CORBA-compliant ORBs, FTS can use the FtRequestServiceContext [90] generated by the ORB as a unique request ID without requiring a proxy.

---

5This overhead may be reduced in ORBs that implement standard CORBA optimization for colocated invocations [87], where a client and a server objects share the same ORB.
Service Contexts

As described in Section 3.2.3, FTS needs to associate additional data with each request and reply exchanged between clients and servers. The additional data are used for the request ID (Section 4.8), for implementing sequential consistency (Section 4.4), and for storing SACKs (Chapter 7). FTS uses CORBA’s service context facility for defining and manipulating data associated with requests.

Middleware support for service contexts is not mandatory. In the absence of service contexts in a given middleware, service context-like support can be provided for slightly reduced transparency by adding a context structure as an input/output parameter to each method of a replicated object interface. This parameter would be handled entirely by the replication mechanism and will not be relayed to the application servants. It may even be possible to avoid an explicit additional parameter by having the replication mechanism transparently invoke a transparent proxy upon receiving a request/reply in order to retrieve the associated context from the opposite peer.

Indeed, FTS utilizes service contexts for purposes that may not be common to portable replication mechanisms in general. However, service contexts are useful for countless other applications as a means to transparently extend the semantics of a request. For example, a distributed transaction service may associate a transaction identifier with a request to indicate that it is a part of a larger transaction. Also, a security service may attach credentials to a request in order to restrict or allow its execution. The concept of service contexts is embedded in many modern client-server protocols such as SOAP [24].

8.2.2 Comparison With FT-CORBA

Originally, FTS was designed to provide fault-tolerance for CORBA applications. Since the FT-CORBA [90] specification is the official CORBA standard in this field, as well as the only middleware specification we know of that deals with fault-tolerance, a comparison between FTS and FT-CORBA is in order. We present this comparison also in the hope that some of it may contribute to improving and extending the FT-CORBA specification in the future, since we believe FT-CORBA is a step in the right direction towards integrating fault-tolerance aspects into middleware standards.

FTS does not adhere to the FT-CORBA specification, but rather represents an independent approach at the same problem of providing middleware systems with fault-tolerance capabilities. In the following sub-sections we compare FTS and FT-CORBA from aspects of replication style (operations and configuration), implications of replication on object reference and solution architecture. We conclude by explaining how FTS
overcomes several of FT-CORBA’s limitations.

Replication Style

FT-CORBA was defined from scratch as a comprehensive specification that encapsulates all techniques of replication, including stateless, passive, active and voting (for value faults). Furthermore, it allows a broad scale of application involvement and control, ranging from no involvement (infrastructure handles all replication aspects) to full integration, where the application itself may implement parts of the replication mechanism, such as failure-detection, state-transfer, etc. In contrast, FTS is a growing project that started with the most difficult implementation issues, namely automatic active replication of server objects, and constantly expands towards new territories, as discussed in Chapter 8.

On a more fundamental level, FT-CORBA requires object-based replication, in accordance with the general notion of CORBA, but allows sharing replication resources in the background without specifying how. In FTS, we observe that there are also strong practical arguments in favor of coarser-granularity replication and thus provide a more flexible context-based replication (see Section 4.10.1), which gives the application designer control over replication granularity and the ability to specify which objects are replicated together. This is especially useful in a typical scenario when the same pool of servers is used for replicating many (or all) objects.

FT-CORBA further requires that replicated objects provide linearizable consistency to clients. However, as we show in Chapter 5, the common technique of utilizing GC-supported ABCAST transports in order to achieve this consistency (which is also suggested in the FT-CORBA specification) can be costly in terms of performance. FTS, on the other hand, also supports sequential consistency that allows for better performance by changing the actual order in which operations from different clients are executed (when compared to the observed order). This difference in consistency is undetectable by independent clients that hold at most one reference for the same object, which is quite a common situation in practice.

Replicated Object Reference

FT-CORBA defines the Interoperable Object Group Reference (IOGR) as a means for a client to refer to a group of replicas. The IOGR is a backward-compatible extension of the standard CORBA IOR where each of the reference’s profiles can point to a different replica of the same object. In addition, the reference contains a component called TAG_FT_GROUP that identifies the replicated object as a whole. According to
FT-CORBA, all clients of actively-replicated servers must use enhanced ORBs capable of handling IOGRs\(^6\). Obviously, this limits interoperability, which is why FT-CORBA defines gateways to allow legacy clients to communicate with FT-CORBA servers.

In contrast, FTS utilizes the legacy IOR with no semantical changes. The only modification is adding the FTOID component that associates the referenced replica with the object it replicates (see Section 4.2). The FTOID component contains a global identification of the replicated object, similar to the \texttt{TAG\_FT\_GROUP} of FT-CORBA. However, adding and processing the FTOID does not require modifying the ORB, since it can be done using the \textit{IOR Interceptor} facility of the CORBA PI specification [96] or even through the core CORBA standard itself that provides a portable and accurate specification of the IOR structure.

Because of not modifying the ORB, using an IOR may present a partial semantic limitation of replica equivalence, as discussed in Section 4.10.4). On the other hand, using standard IOR promotes interoperability, by allowing non-replicated clients to communicate directly with FTS servers without risking service consistency. Furthermore, it allows server-side FTS to be deployed upon legacy ORBs, unlike FT-CORBA.

Another implication of using IOGR (compared to IOR) is the need to keep each client updated w.r.t. the current membership of the group of replicas. To this end, FT-CORBA defines the reference versioning mechanism, where a server is responsible for updating the client’s reference by throwing a \texttt{LOCATION\_FORWARD} exception with an updated IOGR whenever the membership changes. This has two drawbacks. First, when the view of a servers changes (servers join/leave/crash), all pending client invocations at servers, even those that did not fail, are aborted with exceptions and need to be retried. In contrast, FTS client invocations are only delayed during view-change with some requests aborted because of unsafe updates or the servers becoming inactive in non-primary components. Second, if all the servers in the cluster are replaced, the client is left with no alternative but to abort operation. FTS clients, on the other hand, always eventually obtain a new reference from the LMR sub-service during fail-over, provided that they have sufficient access to it (see Section 4.6)\(^7\).

One possible advantage of FT-CORBA’s IOGR over IOR in FTS is that client redirection following a fail-over is faster since an alternate server is determined locally at the client without initiating a distributed protocol. As a counter-advantage, note that LMR servers can enforce server-side redirection policies, such as load-balancing or network proximity-based redirection.

\(^6\)Passively-replicated servers can be accessed directly from legacy ORBs, as the clients are redirected to the primary member using \texttt{LOCATION\_FORWARD} exceptions.

\(^7\)In fact, in a much later paper [36], the proposers of FT-CORBA suggest implementing a secondary client redirection, similar to LMR, in order to address the problem of complete change of a server cluster.
IOGRs also require additional coordination when creating new objects [36]. After each replica is created and registered with its own IOR, the references are forwarded to a central Replication Manager, which composes the finalized IOGR that is returned to the client. In contrast, FTS does not require central coordination to create a new object, since each factory creates, registers a new replica and constructs a new reference on its own.

Last, on a more abstract level, we strongly believe that unlike a group communication service, where a client explicitly communicates with a group of peers, in a fault-tolerance service a client communicates with only one peer (although a fault-tolerant one). Thus, the replication configuration is a concern of the replicated object alone and should not be disclosed to the client through the object reference.

Architecture

The FT-CORBA architecture (see Figure 2.5) defines a hierarchy where the object replicas on all hosts are governed by central fault-detection and replication management mechanisms that are also replicated to avoid a single point of failure. In comparison, FTS architecture consists of symmetric servers where the replication infrastructure is tightly coupled with the object replicas.

Compared to the symmetric structure of FTS, the hierarchy of FT-CORBA suffers from two major drawbacks. First, an additional fault-tolerance solution may be needed for the infrastructure itself, aside from the replicated objects. Some FT-CORBA-compliant systems such as IRL [67] resemble FTS by coupling parts of the infrastructure with the objects, whereas others, such as DOORS [84], incorporate additional solutions such as passive replication of the replication managers using two instances that monitor each other. The second architecture drawback concerns network partitions, where a separation of the infrastructure and the object replicas would make them inaccessible to clients, even if the clients are still capable of communicating with the replicas.

The observant reader may notice that the self-reliant LMR sub-service may cause problems with FTS that are similar to those of FT-CORBA’s hierarchic architecture. However, there are differences. First, the LMR service is a quorum-system of passive, simple and easily-restartable servers, compared to the active complex functionality that is reserved for the replication manager of FT-CORBA. Second, LMR servers are needed only upon view-change and during fail-over. Thus, unlike FT-CORBA, an FTS client that holds a reference to an active server in an established primary component will be able to continue being served even if neither parties can access LMR servers. Last, the proposed modification in Section 8.3.3 completely removes the need for the LMR sub-service and guarantees that once a quorum of FTS servers is gathered inside the same
network partition, it will be able to both establish a primary component and provide client redirection inside the partition without any external dependencies.

As a second comparable aspect, FT-CORBA allows using proprietary protocols, such as GC-based multicast protocols or even multiple concurrent IIOP invocations, for client-server communication, especially when the server is actively-replicated. Similar to the previously-discussed IOGR problem, this further reduces interoperability and requires the use of gateways to allow interaction with legacy ORBs. FTS, by encapsulating the proprietary ABCAST functionality inside the server, allows greater interoperability.

As a final architectural comparison, FT-CORBA defines \textit{fault-tolerance domains} (ft-domains) inside which multiple objects are replicated using virtually the same replication infrastructure. FTS defines replication contexts for the same purpose. The main difference between the two solutions is that. The same FT-CORBA infrastructure can replicate different objects that reside on remote hosts, whereas replication contexts can only replicate together objects that reside on the same host. On the other hand, ft-domains have several shortcomings. The first is the separation of infrastructure and objects, which was already discussed. Second, FT-CORBA requires that non-replicated objects would access an ft-domain through gateways, for the reasons already indicated of protocol and reference incompatibility. This makes an ft-domain a physical deployment entity as a set of servers isolated from the rest of the non-replicated servers through gateways. In contrast, FTS servers (and the supporting LMR servers) merge into a general middleware deployment environment with replication remaining as an orthogonal aspect, thus giving greater deployment flexibility to the service administrator.

**Breaching FT-CORBA Limitations**

In accordance with the previous sections, FTS provides at least partial solutions to some of the documented limitations of FT-CORBA, as listed below.

**Homogeneity** FT-CORBA requires that all the ORBs within the same ft-domain will be of the same vendor. There are two documented reasons for this. The first [90] is proprietary extensions that vendors are permitted to introduce into their ORBs for client-server interaction. FTS, by limiting itself to legacy client-server interaction, does not suffer from this problem. The second reason [36] is the need for identical ORB behavior under identical conditions. Ideally, an ORB should be a stateless black-box with a uniform external behavior regardless of its vendor. As a result, the state that needs replicating includes that of the application object being replicated and its matching registry at the infrastructure, such as the age of objects in FTS. In reality, however, there could be object-related information that is hidden inside the ORB and can affect its behavior w.r.t. the replicated object.
Therefore, the problem is not only of using same-vendor ORBs, but also of being able to transfer ORB-level state between ORBs, to which the standard makes no provisions. Still, the only documented example of ORB divergence problem we have been able to locate comes from Eternal [77], where the problem is that of a replicated client, whose replicas generate different unique request ids in cascading invocations because their ORBs have different internal state for the same object. However, FTS is capable of portably generating unique request ids on its own, which would eliminate the problem. Still, this aspect should be carefully looked into when porting FTS to other middlewares as well.

**Determinism** In order to fulfill FT-CORBA’s *strong replica consistency* requirement, all request executions are required to be deterministic. FTS, on the other hand, adds a distinction between queries and updates, and thus allows queries to be non-deterministic. Furthermore, a proposed FTS extension (see Chapter 8) will also allow non-deterministic updates under certain conditions. This relaxes the limitations put upon application designers. Note, however, that some caution is still in order. For example, querying a server for a random number should be ok, but asking for a server’s time of day without proper coordination might produce decreasing time-readings over a sequence of queries if a client fail-over occurs.

**Network Partitions** As previously indicated, FT-CORBA is incapable of handling network partitions, for its hierarchic architecture (see Section 8.2.2). Additionally, a complete change of the server cluster may result in clients being unable to access the service despite having network reachability. FTS, in contrast, replicates the infrastructure together with the objects and introduces the LMR sub-service for handling any changes in the server cluster configuration (see Section 4.6). This gives FTS services some (although not full) ability to keep serving clients even in the presence of network partitions.

**Correlated Faults** FT-CORBA is especially vulnerable to correlated faults, i.e., widespread software bugs, due to its requirement of homogeneous implementations. FTS, on the other hand, can be deployed across multiple ORBs, platforms and programming languages, provided that the GC toolkit in use can also be ported and that the application state can be portably transferred. The GC portability requirement is met by many popular GC toolkits such as Ensemble [1] and Spread [5] and also by middleware-based GC toolkits such as OGS [31]. The latter requirement of portable state-transfer is application dependent but can be helped with by recording state using CORBA facilities of value-types and records for portable transfer of generic data constructs. If all requirements are met, then a replication deployment over multiple environments, together with voting (see Section 4.3), can form a simple variant of *N-version programming* [66], which may increase its resilience to correlated faults.
8.3 Future Directions

8.3.1 Porting to Different Middlewares

The only mandatory condition out of the ones identified in Section 8.2.1 is the interception mechanism, since all other conditions can be provided for by FTS itself. This strengthens our belief that the architecture of FTS is portable to other middlewares. In principle, the entire FTS logic can be visualized as a wrapper that encapsulates the middleware broker from every direction. The lower-level interception points of incoming and outgoing requests and replies are covered by the required interception mechanism, whereas the upper-level interception points of interaction between the application objects and the broker is covered by a wrapper that emulated standard broker behavior towards the objects (e.g. the standard POA of CORBA). Therefore, the replication concept itself suffers from no apparent portability limitations.

On a separate note, CORBA is currently fading out of the commercial market. It is being replaced by more recent middlewares of Java/J2EE [69] and .NET [19]. This provides strong motivation for us to consider a proof-of-concept of porting FTS to at least one of these middlewares.

Both .NET and Java effectively encompass two levels of distributed application brokering. The basic level, called .NET remoting for .NET and Remote Method Invocation (RMI) for Java, is used for applications where both the client and server are developed using the same middleware. This level offers a distributed object model similar to that of CORBA.

Implementing FTS using .NET remoting seems not only feasible, but potentially more efficient than its CORBA version. .NET remoting offers a wide range of interceptors with full access to request contents, both before and after marshaling and unmarshaling. Specifically, .NET remoting supports installing custom channel objects, which allow seamless plug-in of proprietary protocols, similar to the CORBA OCI specification [101].

Java RMI, on the other hand, has offered no interception mechanisms before its recently-introduced version of Java EE 5. However, the J2EE specification defines a more recent variant of RMI that uses CORBA IIOP as its communication protocol, by translating RMI invocations into standard CORBA invocations between ORBs embedded in the Java Virtual Machines (JVMs). This suggests a strategy of positioning FTS at the CORBA level to achieve transparent replication of RMI objects. Still, the automatic mapping of Java RMI objects to CORBA may have to be modified since FTS is not completely transparent to server objects, in the sense that FTS requires them to register to an object adaptor that is different from the standard POA.
At the higher enterprise level, .NET and J2EE offer component-based architectures called Enterprise Services and Enterprise JavaBeans (EJB), respectively. A component is essentially a class that instantiates objects upon client request or upon a predefined server model. In most cases, the objects are used only to manipulate state rather than to store it, since the application state is typically stored in a 3rd-layer database. Such a setting also provides a foundation for distributed transactions involving the application objects. The component-based architectures are primarily intended for constructing web services, that is, services that can be deployed and accessed regardless of the middleware used at either the client or server sides. As such, both .NET and J2EE components can communicate through a middleware-independent SOAP [24] protocol, in addition to the middleware-specific protocol.

In accordance with the persistent state storage, both architectures offer fault-tolerance solutions that are based on passive replication. In order to offer active replication, FTS would have to be adapted to handle database replication, which implies support for transactions and locking (see Section 8.3.2). Also, FTS operation at the enterprise level can be different than at the basic middleware level, since at both J2EE and .NET, SOAP processing provides specific high-level interception points that can be exploited to improve efficiency [19, 69].

8.3.2 Extending FTS

As noted in Section 8.2.2, FTS currently supports only a small subset of fault-tolerance solutions that focus on active replication with a small degree of optional application involvement. In this section we list complementary directions of evolution that would make FTS a more comprehensive fault-tolerance solution.

Passive Replication and Efficient Cascading Invocations

Passive replication, as discussed in Section 1.5.1, offers several noteworthy advantages over active replication, which result from processing a request by only one server instead of the entire cluster, followed by transferring the resulting state-update to the rest of the members. The first advantage is resource efficiency, provided that the cost of request processing is smaller than state propagation. Second, a single execution of a request can be non-deterministic, unlike multiple independent executions of the same request.

A third advantage of passive replication, which is of special importance to FTS, is the ability to handle cascading invocations efficiently. In a failure-free run, only the server that executes the request issues cascading invocations. Thus, if a request execution at an object with \( m \) replicas involves \( k \) cascading invocations at objects with \( n_1 \ldots n_k \) replicas
respectively, then the resulting message complexity using passive replication would be $O(m + k + \sum_{i=1}^{k} n_i)$. The addition of $O(m)$ messages is due to the propagation of state in the end of the request execution. In comparison, a “good run” of active replication in FTS where all the replicas of the invoker address the same replica of the invokee would be $O(k \cdot m + \sum_{i=1}^{k} n_i)$. Furthermore, the worst-case scenario of active replication, when each replica of the invoker contacts a different replica of the invokee, would yield a far worse message complexity of $O(k \cdot m + m \cdot \sum_{i=1}^{k} n_i)$. Therefore, passive replication leads to more message-efficient cascading invocations.

Consequently, we would like to incorporate passive replication into FTS, but without changing its basic method of operation for active replication. A hybrid architecture that streamlines both active and passive replication has been demonstrated in other works. OGS [31], for example, unifies active and passive replication by having its clients reliable-broadcast their requests to all the servers in the replication cluster regardless of the replication method. On the server side, both active and passive replication are implemented on top of a consensus service, which is used to order requests in case the active replication’s sequencer or the passive replication’s primary fail, in a manner that resembles a view-change in other GC systems. The resulting architecture further enables allowing passive replication and active replication for different methods of the same object. This seems a useful property since different methods of the same object may differ as to the relation between the amount of computation required and the amount of state that needs to be transferred following the computation.

An interesting research direction would be to construct a conceptually similar hybrid architecture in FTS. However, unlike OGS, the solution would have to be based on common properties of GC, as specified in the GCI component of FTS, namely virtual synchrony and group transports. Note that the Delta-4 semi-active replication [104] is, in fact, an existing GC-based passive replication solution that simply assigns the coordinator to handle execution of all requests issued at a replicated component and then propagate the updates to the rest of the cluster. However, unlike Delta-4, we aim to reach a solution that allows dynamic mixing of active and passive replication styles of request executions for the same context/object. Furthermore, regardless of the coordinator, we would like to assign multiple concurrent primaries for replicated contexts with object-granularity consistency. Each primary would be assigned to a different object in the same context, so that request execution for the same context can better utilize the proliferation of processors in the cluster.

Another interesting research direction that could follow is to develop a scheme for dynamically selecting a subset of members as primaries to execute the same (deterministic) request in order to explore a trade-off between speed of response and resource efficiency. By allowing more than one primary to execute a request, the client is likely to receive a reply from the fastest primary, for the price of wasting more resources. The selection
scheme can also be combined with a comparative information about the “processing strength” of servers, based on platform properties and processor load indications, which could be exchanged between servers.

Last, note that the message complexity of cascading invocation can also improve for active replication by assigning only one replica to issue each cascading invocation and then relay the reply to all the other replicas. That replica can be selected using a local deterministic rule, such as a hash function that maps the request id to a member rank. As a result, the worst-case scenario of a failure-free operation where each invoker replica contacting a different invoked replica will be avoided. Thus, the failure-free message complexity of cascading invocations will not exceed $O(k \cdot m + \sum_{i=1}^{k} n_i)$, following the definitions above.

Replica Consistency and Transactions

Supporting multiple types of consistency in FTS gives the application designer the choice of using a consistency that is weaker than linearizable consistency for the gain of better performance, in the sense that requests can be processed and replied-to faster, as demonstrated in Chapter 5. Multiple granularity levels further assist in guaranteeing consistency despite partial dependency between objects, thereby providing a more comprehensive programming model. These results provide strong motivation to further explore mechanisms for defining and guaranteeing consistency models.

Other than linearizable consistency, FTS currently supports only sequential consistency, and only at context granularity and below. Since sequential consistency is not local, it is not guaranteed at the entire application as a whole. Following CASCADE [120] and [116], the request processing protocol can be extended to support age vectors that map each object’s FTOID to its last-seen age. The basic principle is that every method invocation propagates back to the invoker a unified vector of all the ages of objects that were involved, including through cascading invocations. This vector is then used to maintain the read-your-writes and monotonic-reads properties by attaching it to consequent invocations from the same client and using it to delay request execution until the matching object reaches the specified age (recall that the other properties are guaranteed by the causality-preserving total order of updates at each object). Also, optimizations can be applied to reduce the additional bandwidth overhead of age vector transport (see [120]).

On a broader scale, it would be interesting to replace the request processing protocol of FTS with a more generic consistency framework as demonstrated in CASCADE. This would allow supporting weaker (but still useful) consistency models for the gain of performance. For example, consider a yellow pages application [120], in which each
client updates only its own information record but can inquire information about other clients’ records. An implementation of such a service requires only a FIFO of updates instead of total order, which also reduces the GC requirements.

A general consistency framework in FTS would have provide additional properties compared to CASCADE. First, it would provide fault-tolerant active replication of objects, which is not supported by CASCADE. Second, it would handle partitioned operation with automatic reconstruction of a set of replicas provided by the integrated GC component. Last, it can be configured to operate over WANs instead of LANs by selecting a WAN-oriented GC toolkit for update propagation such as Spread [5].

On a more abstract level, maintaining fault-tolerance consists largely of guaranteeing replica consistency. On the other hand, in the presence of failures, guaranteeing replica consistency requires fault-detection and fault-handling. Therefore, adding a flexible consistency framework to FTS seems a logical step towards a fully-customizable comprehensive solution to both problems.

As a short-term consistency-related goal, it may be interesting to explore the possibility of reducing GC dependency by moving the total-order to FTS, so that GC is required to provide only efficient broadcast and virtual synchrony. On one hand, the higher-level implementation of total-order may be less efficient in FTS than in the GC. On the other hand, aside from reduced GC dependency, the combined solution will allow independent delivery of concurrent requests that target different objects in the same context. In comparison, recall that in the current design of FTS, a single ABCAST transport delivers all the requests that target same context, regardless of the consistency granularity being used.

Another aspect of consistency that needs to be extended in FTS is support for serializability of requests, namely transactions. At first glance, transactions may seem as an unrelated aspect that should be encapsulated inside a higher abstraction in the form of a transaction service such as CORBA OTS [95]. Still, FTS needs to provide basic support for transactions since it controls the scheduling of requests in replicated objects. On top of this basic support, further research may pursue a construction of a full-blown fault-tolerant transaction service that defines the semantics of a transaction, supports nested transactions, etc.

Currently, durability of the effect of a request execution, which is required for transactions, is guaranteed using safe delivery. However, additional required properties of request sequence execution atomicity and isolation can only be provided if all the replicated objects reside in the same context using context-granularity consistency. In order to guarantee atomicity and isolation in arbitrary granularity, an explicit locking mechanism would have to be introduced. As noted in [120], in order to support fail-prone clients, such a locking mechanism would most likely be based on leasing to allow progress.
despite client failure. FTS may also expose its state-transfer capabilities to the application level in order to implement transaction roll-back.

Security

Security is a crucial need for distributed applications that serve human users. However, security policies cannot be applied to FTS applications without explicit FTS support, since FTS controls all communication to and from replicated objects. Additional challenge lies in establishing the required support for security with minimum reduction of FTS portability, interoperability and performance.

Note that in order to achieve total application security, support must also be provided at all the layers underlying FTS. Specifically, no additional interception should be permitted at the operating system / virtual machine / middleware broker / GC subsystem levels. Accessing the FTS infrastructure through either of these layers can bypass any security provision, forcing FTS to resort to overcoming Byzantine (arbitrary) failures in order to at least survive damage to part of the replicas.

Based on the comprehensive CORBA security service specification [99], we identify key security features that need to be supported at the level of object replication in FTS, as listed below.

**Authentication:** each client and server entity (application / object) needs to be authenticated to verify who it claims to be. This requires FTS to utilize digital signatures and certification authorities when admitting clients and server replicas into a replication domain.

**Authorization and Access Control:** this pertains to deciding whether a client entity may invoke certain methods of an object and/or whether a server may participate in replicating an object. To this end, access-control policies may be specified at various granularities (application/context/object), and the FTS interceptors will be used to enforce these policies using the authenticated identity of the entity in question.

**Security Auditing:** in order to allow accurate post-attack analysis, all inter-object communication, both between clients and servers as well as between server replicas, must be logged, with each record specifying the authenticated identity of the communicating parties.

**Security of communication:** to ensure privacy of communication between all parties, FTS interceptors and GC endpoints may have to encrypt communication, using a suitable scheme of private / public key cryptography.
Non-repudiation: this feature requires that every operation can be traced back to its causes, i.e., not only its originator but also the factors that triggered a given operation at its originator. Common techniques for achieving non-repudiation include digital signing of requests / replies (for non-repudiation of the originator), inclusion of identifiers of causing messages in the signed content and using a trusted third-party non-repudiation service to generate undeniable evidence of an operation [99].

Additional functionalities, including administration (interface, semantics of global security policies) and user identification, can be implemented by a higher-level security service. Also, support for inter- replica authentication, digital signatures and encryption can be provided by the GC subsystem by using toolkits such as Ensemble’s security extensions [107].

One last security aspect to consider is the previously-mentioned survivability of a consistent service state in case hackers do manage to infiltrate into servers and make part of the replicas Byzantine, i.e., behave arbitrarily instead of according to their predefined specifications. This type of failures also pertains to cases where a hardware failure such as memory corruption causes few of the replicas to behave inconsistently with their specification.

If Byzantine failures can be limited to the application objects, then FTS in its current design can tolerate failures at less than half of the replicas using voting, as described in Section 4.3. However, containing damage inside one layer requires access-control inside the application server, which depends on the platform. Java and .NET provide security managers for this purpose, at the cost of performance. Also, operating system facilities, such as Windows NT security [111] or Security-Enhanced Linux (SELinux) [3] can provide fine- grained access-controlo based on hardware support.

Generally, however, Byzantine failures are hard to overcome. For starters, FTS can employ a Byzantine-tolerant GC toolkit, such as JazzEnsemble [29] or Rampart [105] in order to guarantee the strong semantics of view synchrony, virtual synchrony and ABCAST transport for correct servers. However, FTS logic, which controls the GC subsystem, can also Byzantine-fail. For example, it can return corrupted replies to clients, or not reply at all. Also, it can inject replays of authentic client requests to the ABCAST transport in order to break at-most-once semantics.

Therefore, FTS architecture would undergo extreme changes to be able to handle Byzantine failures. A most likely design would resemble the solution of Eternal [77], only in a portable implementation. The key strategy is rule out dependency upon any single channel of information, since such a channel can fail and corrupt or lose information. Therefore, clients would directly contact all servers and filter replies using majority
voting. If the client-server communication is still in standard middleware point-to-point invocations, then servers would also agree on client requests to guarantee atomicity, order and filtering out of non-identical requests. In short, the solution would likely be not only much more complex than the present design, but also less interoperable and scalable.

8.3.3 Proposed Design Modifications

In this section we review several proposed short-term modifications of primary FTS components, aimed at improving qualities of FTS without extending its model of operation. While these modifications involve mostly technical development, some of them also point out a few interesting research directions.

Merging Dynamic Configuration Management

Currently, changes in service configuration are handled in FTS both at the client and server sides using the LMR sub-service. However, this design has a few shortcoming. First, the core FTS service is not completely self-reliant, in the sense that establishing client-server communication depends on a third party that may not reside in the same network partition as the clients and servers. Second, the leasing mechanism requires clock synchronization between FTS servers and LMR servers.

Thus, we propose new design, in which a quorum-system will be defined over the FTS servers themselves. Each component can decide on its own whether it contains a quorum and can thus become a primary component. State propagation would still be maintained since each two primary component views intersect through the quorums they contain. The client redirection functionality would be provided by selecting one quorum of FTS servers and activating a redirector in each of that quorum’s servers\(^8\). When a cluster of FTS servers forms a primary component, it executes the view-change protocol including age computation and state-transfer and opens for service. The redirectors that co-exist with FTS servers in the same component are automatically updated with the view information (there must be at least one redirector in each primary component due to quorum intersection). Once the redirectors are updated, each client that can reach a primary component’s server is guaranteed to obtain a reference to an accessible object by contacting the quorum of redirectors.

Compared to the current design, the new design considerably simplifies service acquisition on the server side, since each server can locally determine whether it belongs

\(^8\)This solution still improves over [36] in the sense that it does not depend on a single fixed redirector that also needs to survive on its own.
to a primary component already during view-change. Also, redirection information is automatically updated. The removal of the LMR interaction also discards the requirement for clock synchronization. The client side also changes a little, in the sense that it contacts a fixed list of redirectors and attempts redirection according to the reply with the highest PVC, instead of contacting all redirectors and waiting for a quorum of replies before redirection. From the aspect of client progress, however, the future design provides the same quorum-based guarantees, only without the need for a third-party of LMR servers. That is, a client that resides in the same same network partition with a primary component is guaranteed to be served.

The only expected limitation of the new design in comparison with the current one is the increased complexity of handling addition and removal of FTS servers. In the current design, FTS servers can be added and removed freely, since the primary component and redirection issues were decided using a fixed infrastructure of LMR servers. In the new design, however, adding and removing servers change the quorum-system that is constructed on top of the set of servers. Consequently, a protocol would have to be developed that will introduce the knowledge of a new quorum-system into an existing one, in the spirit of works such as RAMBO [65].

Decoupling Requests and Replies

As Chapter 5 concluded, current FTS design suffers from a performance bottleneck that results from the tight coupling of requests and threads in popular threading models provided by ORBs (as well as other middleware brokers). Specifically, when using a limited thread pool to serve requests that exhibit long wait periods during execution, the thread pool becomes exhausted at high loads, so the processor becomes idle with all the currently handled requests waiting while new requests, which could use the processor but cannot be assigned to a thread, are pending.

Aquarius [22] is another CORBA active replication work where the threading model problem was observed. The solution that was chosen was to modify the ORB code to allow explicit de-coupling of thread and request. In FTS, however, we strive not to
violate middleware portability. Therefore, the solution that we plan to implement in FTS is as described in Figure 8.1, and is based on de-coupling request submission and execution. Each FTS client would access the service through a transparent proxy\(^9\). Using DSI, this proxy would convert each client request into a \texttt{Request} structure that would be passed as a parameter to a special \texttt{submit()} method of the server. The \texttt{submit()} would return as soon as the request is \texttt{submitted}, which at the server would occur as soon as the request is serialized and passed to the \texttt{ABCAS} transport. When the request is delivered back from \texttt{ABCAS} and executed, the server will invoke a \texttt{reply()} callback at the client proxy, which would then pass the reply back to the client. This way, the server threads can switch requests as soon as the request is passed to the \texttt{ABCAS} transport, instead of waiting until it is done executing. Also, this solution does not depend on the threading model being used.

There are a couple of additional minor optimizations that can be applied within this modification. First, the query/update tagging can be done at the client proxy, as well as the serialization (marshaling) of updates. This can take processing work from a server and distribute it over many clients, thereby harnessing more processing power to improve performance. Note, however, that more bandwidth may be required to transfer the serialized request and the tagging information. Second, by using a synchronous invocation for \texttt{reply()}, the server can receive a notification of successful delivery of a reply to the client. This notification can be used to remove the request from all the caches, instead of a dedicated SACK mechanism. On the downside, notifications still need to be broadcast to all the servers in order to update all the request caches. Also, the synchronous \texttt{reply()} invocation generates more bi-directional network traffic than a single-directional reply.

\(^9\)A proxy solution already exists in FTS in order to generate globally-unique request ids at the clients that use legacy (pre-FT-CORBA) ORBs
Bibliography


145


[85] OMG. Concurrency Service specification, v1.0. formal/00-06-14.

[86] OMG. CORBA/IIOP Specification 2.3.1.

[87] OMG. CORBA/IIOP Specification 2.4.2. formal/01-02-33.


[89] OMG. CORBAservices: Common Object Services Specification. formal/98-12-09.

[90] OMG. Fault Tolerant CORBA specification, v1.0. ptc/00-04-04.
[91] OMG. IDL to Java Language Mapping Specification 1.2. formal/2002-08-05.
[93] OMG. Naming Service specification, revised. formal/01-02-65.
[95] OMG. Object transaction service, v1.4. formal/03-09-02.
[96] OMG. Portable Interceptors. ptc/01-03-04.
[97] OMG. PSS 2.0 Revised Submission. orbos/99-07-07.
[98] OMG. Reliable ordered multicast inter-orb protocol. realtime/02-11-28.
[99] OMG. Security service, v1.8. formal/02-03-11.
[100] OMG. Unreliable multicast inter-orb protocol. ptc/01-11-08.
[101] OMG/AT&T. Internetworking between corba and intelligent network systems. telecom/98-06-03.

149


[121] D. Wessels. Squid web proxy cache.

ארוכitates לשליחות חסינות-קסיני- novità

חיבור על מחקר

לשם מילוי חקיק של הדרישות לקבלת תואר
דוקטור לפילוסופיה

ארז חדר

הנהלת הטכניון – מכון טכנולוגי לישראל
ניסן ה'תשס"ז' – יוני 2006
המחק עשה בחינתה את רועי פרידמן בפקולטה למדעי המחשב.

אני מבקש להודות לרועי פרידמן על הנחיה המלאת השראה והעוצמה של ההכדורים וה открываות שלו בכלפיו במקלדת התמודדות עם קשייה קסתיו.

תודה לפרופ’ חגי עטייה ול’רעיית קידר על סיועםToggleButton ו:^(טרים ו�� מיצורים של תקוה, לה öffفع על afterward מתכלה מתייצבים מועלים ונספים. תודה

לפרופ’/ת יני על הנוחות והנושאים של התاحتجנות.

תודה לכל קרגבר על התמיכה ב’huiי שנקראים הקדום/ים בברית.

תודה לירדנה קול למנוע על העזרה הרחב וה𝗴ימל היייל בעניין אדריכלות.

כמentreprise חנה ובר ייביס בידות, תודה לעצקה מוחלפת נפשם נפשם אירת ילדי המ👌יקן אווד

ואלא שوترו של שומרון על שתות החיה ובר עונה מתוח קשיא מחירי, ואצטבים להורות מאיים מוכלים תר.

להםinki והאני עם תמי והשכלתי מרסים רימש. כל הМИני הכותב אוית, סקף רכ איהי מוגע שלב ה.

אני מודה לכרך י’שו איהי וברбит נטר. لكل אסבל של משמר המדה, لكلו וה’של המעציה להשכלאה

גבוהו הלטיני על התמיכת המכפים המירב במקלדת השולמות.
תוכן עניינים

תקציר  3
מבוא  1
. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
ה회עה עלработה  1.1
סקירת המחקר  1.2
מבחנה ההרה  1.3
CORBA  1.4
תס eiusונת לתוך התחום  1.5
טכניקות שכפול  1.5.1
תקווה קרבה  1.6

חסינות לתקלות בתווכה  1.5

tekfora 2.2

סקירה ספרותית  2
. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
הוספת טכניקות שכפול ל  2.1
גישתה ההרכבה  2.1.1
בישה היירוט  2.1.2
בישה שירות  2.1.3
בישה מתאם האובייקטים  2.1.4

מפרט  2.2

ckeditor 3.2

מטמון הבקשות ואישורים סלקטיביים  2.3

מטמון הבקשות ואישורים סלקטיביים  2.4

8
 услуги התכולה

3.1 עמידות למתקלת והזדמנות

3.1.1 ביצועי יוכלו התוכלמות

3.1.2 פשטות מבנה והישומים

3.1.3 FTS של הנחלות

3.1.4 מתאם קבוצת אי-בטיחות

3.2.1 התממשות עם תקשורות קבוצת

3.2.2 FTS של BUILD

3.2.3 עקביות מבנה

3.2.4 התוכן של מבני

4.1 מודל התוכן

4.2 קיור ביני זמנים

4.3 תהליך התוכלמות

4.4 עקביות ביני זמנים

4.4.1 תיאור הפרוטוקול

4.4.2 תורת הפרוטוקול

4.5 בקשות פנים ביני אי-berraית

4.5.1 תקף ביני הקשרים של כלכל

4.5.2 קישוטים ביני החסרות

4.6 תוצרת שירות דינמית

4.6.1 תורת הפרוטוקול

4.7 זיהוי ומיטר בקשות

4.8 גוון זרימה של אי-berraית

4.9 גוון

4.10.1 גוון ייחודי של הכלכל
ב váחית אופט החמנתו ותקומ
שיקול בטיפול בקונמיורזיציה זנמית
סנטטיקה של מוביע לארוביקט מזוכפל

82 בדיקה בצוועים של
FTS

שיקולים בטיפולבקונפיגורציה דינמית
סמנטיקה שלמצביעלאובייקטמשוכפל

5 בדיקה בצוועים של

5.1 סיבית הבית
5.2 תצורת הבית
5.3 שיטת הבית
5.3.1/dir
5.4 סיכום התצאות
5.5 ניוון התצאות
5.5.1 זמני ביצוע בקשת
5.5.2 תפקות השירות
5.6 מסכמת

6 קיבוץ סתברלי

6.1 אופי העיבא
6.1.1 הצוור בסטנגל
6.1.2 מודל פועל
6.2 קיבוץ סתברלי
6.3 קיבוץ סתברלי מתוחנה
6.4 בדיקת ביצועים
6.5 דיו
6.6 מסכמת

7 המתקף צווחי הקורז ובשדות

7.1 מוטיבציה
7.2 תרשים הengage
7.2.1 תיאור
7.2.2 תבניות
7.3 בדיקת ביצועים
7.4 ניווט התוצאות
7.5 מסקנות
8 סיכום וכיוונים עתידיים
8.1 סיכום תוצאת המחקר
8.2 לקחים שנרכשו במשולב המחקר
8.2.1 ת-caption המחובר בشدد
8.2.2 השוואה עם FT-CORBA
8.3 כלים עדכניים
8.3.1 ניר למש.PathVariable FTS
8.3.2 קורректה FTS
8.3.3 שיפור תכני מציים

תקציר בעברית
## רשימה אזורים

<table>
<thead>
<tr>
<th>מספר</th>
<th>תema</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>בנויה של CORBA IOR (1.1)</td>
</tr>
<tr>
<td>9</td>
<td>תקשורת ל- klient ב-CORBA (1.2)</td>
</tr>
<tr>
<td>19</td>
<td>unist הרכבה (2.1)</td>
</tr>
<tr>
<td>23</td>
<td>unist הייורט (2.2)</td>
</tr>
<tr>
<td>25</td>
<td>unist השירות (2.3)</td>
</tr>
<tr>
<td>26</td>
<td>unist משותם האובייקטים (2.4)</td>
</tr>
<tr>
<td>28</td>
<td>אריכיטקטורה FT-CORBA (2.5)</td>
</tr>
<tr>
<td>41</td>
<td>ишוף משאיב שול יין אובייקטים (3.1)</td>
</tr>
<tr>
<td>43</td>
<td>תенной-על של FTS (3.2)</td>
</tr>
<tr>
<td>45</td>
<td>מכתב-ה (3.3)</td>
</tr>
<tr>
<td>56</td>
<td>פרוטוקול עיבוד בקשה - תדרות תונים (4.1)</td>
</tr>
<tr>
<td>56</td>
<td>פרוטוקול עיבוד בקשה - כלב הלוחות (4.2)</td>
</tr>
<tr>
<td>57</td>
<td>פרוטוקול עיבוד בקשה - כריב השירות (4.3)</td>
</tr>
<tr>
<td>64</td>
<td>墀-שירות LMR (4.4)</td>
</tr>
<tr>
<td>65</td>
<td>سفرת שרת LMR (4.5)</td>
</tr>
<tr>
<td>67</td>
<td>רכש שרות אחר מתחם כריב (4.6)</td>
</tr>
<tr>
<td>68</td>
<td>تكنוגט ל-קוח (4.7)</td>
</tr>
<tr>
<td>85</td>
<td>otropic בדיקת זן תוגה מבצעים בקשת (5.1)</td>
</tr>
<tr>
<td>85</td>
<td>מבוך תוגה: תפקוד בותל ל- bordel מבצעים השורות (5.2)</td>
</tr>
<tr>
<td>86</td>
<td>מבוך תוגה: תפקוד בותל ל- bordel מבצעים (5.3)</td>
</tr>
</tbody>
</table>

ה
רשימת טבלאות

<table>
<thead>
<tr>
<th>מספר</th>
<th>סימון</th>
</tr>
</thead>
<tbody>
<tr>
<td>59</td>
<td>סיכום מטרי עLabouratorio</td>
</tr>
<tr>
<td>91</td>
<td>פרופיל של זמנים תכנון (מיליסנויות)</td>
</tr>
<tr>
<td>93</td>
<td>פרופיל של זמנים תכנון ב��ונים של 1500 الاثنين</td>
</tr>
<tr>
<td>115</td>
<td>בדיקת יציבות: משורות SACK ביהס ליוועס הל rekohot</td>
</tr>
</tbody>
</table>

bfd

2
תקציר

מערכת תכנון מובילה את הנושא באזף של רכיבי תכנון שמושולים לחיזיור. לפועל ופגיע לכל שואו
בלילה-תלי זה בוה. התוכנית בין רכיבי התוכנה מתבצעת באמצעות העברת העבירה והודעה דרך ודאי תקשורת
משתמשי ואימותים קראו את קולה של מ トラックון. רלו היותי התוכן המוסבר משותף ושתי
ממד כל החברות של תכנית תורנייה בכל של מערכות התוכנות לשתי המובילים לתחרות בין שני חותם ב
רכיבי התוכנות כמו גם התקנות אפקטיביות בחידוק mơ הפרד וליום הב使いב.

עם זאת, מערכות מובלות יוצרות עTransportה של האפשרויות הממסור נע שרורות על צומת שגרה ולהשלים
שלים מרחב ייחודי לאמענה. פרס של העיבוד עד-פיי ממס圓ים ממקורות המערביים שעצה זה ביבשת
מקורות. ברורה הזנוף למשת slaמה פלטפורמה ישירה ב למצוא הזרע המרוכב ממ�藏תמקים קניי יולימ
במקומ ממסוב ושארית מצור ביריחות לפני התוכנה בחרת או להגדיל את התוכנה בחרת ליציגט (scale up).
כמכם, האפשרות של החברות שלו שייכים לגופים אחרים, אונס שמסרי אונס塑胶 קניי יולימ
בריכים מואכים לביוסים שרגים, מת עקמרא ברכיבי מסרי גוני והלולג והלולג משטרת וו
לגריך גונל מלי. י疗程 בשארית הנהלנים כל מערכות התוכנה ומחרת התוכנה ומחרת התוכנה, בברית בכר של מנהיג חיים בחודש
המאוהט החשיח לדר המעודכתי למסווע תכנית גולדות, בברית בכר שמהדר帝 רכיב התוכנה

(Enterprise computing)

ארוגני

(neutral) במחקל האביוולוציה של מערכות מבוזרות והופיעה בחיה של תכנית תמיכה שitesse מוערقم תחנה
וכן, RPC, CORBA ו PG של האביוולוציה של מערכות מבוזרות והופיעה בחיה של תכנית תמיכה שitesse מוערقم תחנה
(enterprise computing)

(2006-2006)
בה הרב כיוון שלולית בשכבה שלהם יריית ומפרק תirasיה. מסתיימה תיריה את מעלה התirasיה תחת
הפשתה (abstraction) של כיוון לפורקציה או להℇות מרבייה זהל לאל תילו במעל תבכום 한יה.
ככ שכתיבים דפי מעבר שאמו מטעם ברמת היישום. כמך. כו. עבון איסר-לחק. תיריה ומפשירות
הגדה פורמלית של ממיש התirasיה niên בירח התיריה ביו בפרספקטיבי כו. שטיח שיל
בנער מ痧ו מישור של התיריה ע зло. התירו מחיית לה akşam בنظم הזה כו. שטיח התיריה משמר של
陑 התיריה לירה ואר דר המישור המוקר. ברגרזיו זו מפורפק המישה של ריכי התיריה מחירים
אפת התים מישור לשכבה. כבק-מידה גודל גודל. היישום המברך חוף להဧיטות ישיריות הבניין
זה מכיל זה מבחרה עליה של הפשתה.

ביה עקרונית ומיתרה במעורוב מתוורות医疗卫生 משורות אחרים לתחותר על תיריה. שטיבורית עולה על
הזהל מספר הרכיבים העצמאית של תיריה וה㈮ה המשולב במעורוב. בפרט, במעל תילו-לחק.
השעון של שירות משמעה פיעה בRequestParam הליכו התיריה ו. הסכנות הלפיטות השיריות של
לمراות (redundancy) (fault-tolerant) (replication) הממשה על חיות: ריכי חותית ותילות התיריה.
חייתוNESS מת النقدיות במעורוב:风云 חותית של הירח לישון דער בקיפת תושובה/וא הطائرית התיריה של
ריכי חותית אחר תיריה. סכינת השכפולampil (active replication) מצרלח חיות מיקום הנברד
imore חותירת מככ שיאיות יבשת חותית לטש תיריה לשכפול שטח והוא שטח השכפול
( passive replication) מפריע חיות בקכ שיא שיר חותית המטשל בקיפת הירח וליריה.
תקיפה. השיכוף כייקע דע ליאור הח協會ות מחילקה.

בעקבות הגיאוגרבי שעימה מועד הפרה במעורוב מבוטה תיריה במסלול 해יר גורק צעף בפרותרון בינין
המשלבים עידות הליכי תיריה על תיריה הקים. והי שיאם התира נשיאת ממנה התילה עבדה
וז. מתחים קדומים שובضع בזיעה תייג לשלוח ישות בסיים. גישה התורבה
משנה את מנוגנ התיריה הבסיסית של התיריה בק שיא מוסיפהATALOG של缛 התיריה
integration) interception) לתמייה שלכל מילוי. גישה יירה (atomic broadcast)
(ט kone התיריה בראש ריכי זייגי המימייא את התיריה הבסיסית של התיריה בק מתח את
( service) ב곤 שיריה שיכפל כללי יישום מבורט מעל תיריה. שעלו רון
זיו האיסום עידות תיריה. לכל אחד יש יירה ותיריה, שדרוג: גישה התורבה שקומפת ליישום אבל
In their common techniques with different limitations, they do not have a standardized response. The method of their response is copied from their implementation, and they are limited. The available methods for the implementation of their techniques are not sufficient to implement them. For this reason, they require additional work to adapt them. As such, a method limited in its implementation is often needed.

The request for 1999 for the CORBA standard for the implementation of the techniques, incorporated in the FT-CORBA standard, is not possible to achieve. The methods that are copied from their implementations, and the tools they use, are often limited. The available methods for the implementation of their techniques are not sufficient to implement them. The work is then needed to adapt them. As such, a method limited in its implementation is often needed.

In the first chapter, the methods that are copied from their implementations, and the tools they use, are often limited. The methods that are copied from their implementations, and the tools they use, are often limited. The available methods for the implementation of their techniques are not sufficient to implement them. The work is then needed to adapt them. As such, a method limited in its implementation is often needed.
In the algorithms, examples of this are implemented through the use of user activities, and the parallel behavior in this context is executed in the CORBA service model. The service model is designed to improve the service's performance in parallel, as it is shared between different networks. The CORBA service model, when used in parallel, can improve the performance of the service in this context, as it is shared between different networks.

In addition, the examples are implemented through methods that can be used to batch requests, which can be used to improve the service model's performance in parallel. The FTS method, implemented in parallel, can improve the service model's performance in parallel, as it is shared between different networks.

The FTS method, implemented in parallel, can improve the service model's performance in parallel, as it is shared between different networks. The CORBA service model, when used in parallel, can improve the service model's performance in parallel, as it is shared between different networks.
הסתי了一批. נקבה מידה רוחית, מסתה של תהליך ל meilleurs יישום חוויתית מתנה זה שימש בניו. בניה, נראים של מהנדס של יירוט, פעולות בעלות זה המתח אורות, חומרים, כאן של.NET, מדלא המחקר שית תרומת נספה. יישום בעלת תを与え בתוכנית, מ italiane. ר.0.NET, מתוך המחקר שית תרומת נספה. האחתによות טכניקת לקיון ידע בויל וקישורה וגריג ובלעלת התקדשות מגדיר פ-ודעה, כי שאן מתאימים עזר קיון התפתחות האוספיים. טכניקת ויברקליל הת知らない ליוויים של ספילי בליל אפיי דימה קנן קיון חינוב תודס. הרהמה השתייה היא טכניקת לקיון מתן שלט מתן מתקשוב באימוץ ר.0.NET-1 RPC

. נקן