View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by Washington University St. Louis: Open Scholarship

Washington University in St. Louis

Washington University Open Scholarship

All Computer Science and Engineering Research

Computer Science and Engineering

Report Number: WUCSE-2002-28

2002-05-01

Patterns for Providing Real-Time Guarantees in DOC Middleware - Doctoral Dissertation, May 2002

Irfan Pyarali

The advent of open and widely adopted standards such as Common Object Request Broker Architecture (CORBA) [47] has simplified and standardized the development of distributed applications. For applications with real-time constraints, including avionics, manufacturing, and defense systems, these standards are evolving to include Quality-of-Service (QoS) specifications. Operating systems such as Real-time Linux [60] have responded with interfaces and algorithms to guarantee real-time response; similarly, languages such as Real-time Java [59] include mechanisms for specifying real-time properties for threads. However, the middleware upon which large distributed applications are based has not yet addressed end-to-end guarantees of QoS specifications. Unless this... **Read complete abstract on page 2.**

Follow this and additional works at: https://openscholarship.wustl.edu/cse_research

Recommended Citation

Pyarali, Irfan, "Patterns for Providing Real-Time Guarantees in DOC Middleware - Doctoral Dissertation, May 2002" Report Number: WUCSE-2002-28 (2002). *All Computer Science and Engineering Research.*
https://openscholarship.wustl.edu/cse_research/1145

Department of Computer Science & Engineering - Washington University in St. Louis
Campus Box 1045 - St. Louis, MO - 63130 - ph: (314) 935-6160.

This technical report is available at Washington University Open Scholarship: https://openscholarship.wustl.edu/cse_research/1145

Patterns for Providing Real-Time Guarantees in DOC Middleware - Doctoral Dissertation, May 2002

Irfan Pyarali

Complete Abstract:

The advent of open and widely adopted standards such as Common Object Request Broker Architecture (CORBA) [47] has simplified and standardized the development of distributed applications. For applications with real-time constraints, including avionics, manufacturing, and defense systems, these standards are evolving to include Quality-of-Service (QoS) specifications. Operating systems such as Real-time Linux [60] have responded with interfaces and algorithms to guarantee real-time response; similarly, languages such as Real-time Java [59] include mechanisms for specifying real-time properties for threads. However, the middleware upon which large distributed applications are based has not yet addressed end-to-end guarantees of QoS specifications. Unless this challenge can be met, developers must resort to ad hoc solutions that may not scale or migrate well among different platforms. This thesis provides two contributions to the study of real-time Distributed Object Computing (DOC) middleware. First, it identifies potential bottlenecks and problems with respect to guaranteeing real-time performance in contemporary middleware. Experimental results illustrate how these problems lead to incorrect real-time behavior in contemporary middleware platforms. Second, this thesis presents designs and techniques for providing real-time QoS guarantees in DOC middleware in the context of TAO [6], an open-source and widely adopted implementation of real-time CORBA. Architectural solutions presented here are coupled with empirical evaluations of end-to-end real-time behavior. Analysis of the problems, forces, solutions, and consequences are presented in terms of patterns and frame-works, so that solutions obtained for TAO can be appropriately applied to other real-time systems.

Department of Computer Science & Engineering

Washington University in St. Louis
School of Engineering & Applied Science

2002-28

Patterns for Providing Real-Time Guarantees in DOC Middleware - Doctoral Dissertation, May 2002

Authors: Pyarali, Irfan

May 1, 2002

Abstract: The advent of open and widely adopted standards such as Common Object Request Broker Architecture (CORBA) has simplified and standardized the development of distributed applications. For applications with real-time constraints, including avionics, manufacturing, and defense systems, these standards are evolving to include Quality-of-Service (QoS) specifications. Operating systems such as Real-time Linux have responded with interfaces and algorithms to guarantee real-time response; similarly, languages such as Real-time Java include mechanisms for specifying real-time properties for threads. However, the middleware upon which large distributed applications are based has not yet addressed end-to-end guarantees of QoS specifications. Unless this challenge can be met, developers must resort to ad hoc solutions that may not scale or migrate well among different platforms.

This thesis provides two contributions to the study of real-time Distributed Object Computing (DOC) middleware. First, it identifies

WASHINGTON UNIVERSITY

SEVER INSTITUTE OF TECHNOLOGY

DEPARTMENT OF COMPUTER SCIENCE

PATTERNS FOR PROVIDING REAL-TIME GUARANTEES IN DOC MIDDLEWARE

by

Irfan Pyarali

Prepared under the direction of Dr. Ron K. Cytron and Dr. Douglas C. Schmidt

---

A dissertation presented to the Sever Institute of
Washington University in partial fulfillment
of the requirements for the degree of

Doctor of Science

May, 2002

Saint Louis, Missouri

WASHINGTON UNIVERSITY
SEVER INSTITUTE OF TECHNOLOGY
DEPARTMENT OF COMPUTER SCIENCE

ABSTRACT

PATTERNS FOR PROVIDING REAL-TIME GUARANTEES IN DOC MIDDLEWARE

by Irfan Pyarali

ADVISORS: Dr. Ron K. Cytron and Dr. Douglas C. Schmidt

May, 2002
Saint Louis, Missouri

The advent of open and widely adopted standards such as Common Object Request Broker Architecture (CORBA) [47] has simplified and standardized the development of distributed applications. For applications with real-time constraints, including avionics, manufacturing, and defense systems, these standards are evolving to include Quality-of-Service (QoS) specifications. Operating systems such as Real-time Linux [60] have responded with interfaces and algorithms to guarantee real-time response; similarly, languages such as Real-time Java [59] include mechanisms for specifying real-time properties for threads. However, the middleware upon which large distributed applications are based has not yet addressed end-to-end guarantees of QoS specifications. Unless this challenge can be met, developers must resort to *ad hoc* solutions that may not scale or migrate well among different platforms.

This thesis provides two contributions to the study of real-time Distributed Object Computing (DOC) middleware. First, it identifies potential bottlenecks and problems with

respect to guaranteeing real-time performance in contemporary middleware. Experimental results illustrate how these problems lead to incorrect real-time behavior in contemporary middleware platforms.

Second, this thesis presents designs and techniques for providing real-time QoS guarantees in DOC middleware in the context of TAO [6], an open-source and widely adopted implementation of real-time CORBA. Architectural solutions presented here are coupled with empirical evaluations of end-to-end real-time behavior. Analysis of the problems, forces, solutions, and consequences are presented in terms of patterns and frameworks, so that solutions obtained for TAO can be appropriately applied to other real-time systems.

Contents

List of Tables

List of Figures

Acknowledgments

First and foremost, thanks to God Almighty for His help, blessings, and mercy which made this decade long journey possible. Thanks to my family for giving me unparalleled encouragement and love. Thanks to my advisors, professors, and mentors for their guidance and wisdom. Thanks to my friends and coworkers for their friendship and support. And thanks to Washington University for a decade of enlightenment.

Here are some people I would like to acknowledge by name:

Pyarali Family: Mohammad Hussain, Gulshan, Yasra, Shahid, Arif, and Asif.

Advisors and Mentors: Dr. Douglas C. Schmidt, Dr. Ron K. Cytron, Dr. David L. Levine, Dr. Christopher D. Gill, Dr. G. James Blaine, Dr. Douglas Niehaus, Dr. Robert B. Pless, Dr. Jeremy Buhler, Stephen M. Moore, Chris Cleeland, Fred Kuhns, Dr. Karlheinz Dorn, Dr. Ebrahim Moshiri, Dr. Richard Schantz, Dr. Joseph Loyall, Craig Rodrigues, Stephen T. Bachinsky, Dr. Russ Noseworthy, and Steve Huston.

Friends: Balachandran Natarajan, Carlos O'Ryan, Marina Spivak, Angelo Corsaro, Pradeep Gore, Jeff Parsons, James Hu, Tim Harrison, Nanbor Wang, Aniruddha Gokhale, Yamuna Krishnamurthy, Prashant Jain, Joe Hoffert, Sumedh Mungee, Michael Kircher, Ossama Othman, Darrell Brunsch, Vishal Kachroo, Kirthika Parameswaran, Sharath Cholleti, and Krishnakumar Balasubramanian.

Irfan Pyarali

Washington University in Saint Louis
May 2002

Chapter 1

Introduction

1.1 Thesis Scope and Related Work

Figure 1.1 shows an endsystem [66] that consists of network interfaces, operating system I/O subsystems, and middleware services. Real-time resources in these subsystems must be *vertically* (*i.e.*, network interface $\leftrightarrow$ application layer) and *horizontally* (*i.e.*, peer-to-peer) integrated and managed to ensure end-to-end predictable behavior for *activities*[1] that flow between clients and servers. These real-time resources are outlined below, starting from the lowest level abstraction and building up to higher level services and applications.

1. Communication infrastructure resource management: A real-time endsystem must leverage policies and mechanisms in the underlying communication infrastructure that support resource guarantees. This support can range from managing the choice of the connection used for a particular invocation to exploiting advanced QoS features, such as controlling the ATM virtual circuit cell pacing rate [10].

2. OS scheduling mechanisms: A real-time endsystem must exploit OS thread scheduling mechanisms to schedule application-level activities end-to-end.

3. Real-time Middleware: Middleware facilitates transparent communication between clients and servers. Real-time middleware must provide standard interfaces that allow specification of resource requirements, such as thread priorities and buffering constraints.

4. Real-time services and applications: Real-time middleware must preserve efficient, scalable, and predictable end-to-end behavior for higher-level services and application

[1] An activity represents the end-to-end flow of information between a client and its server that includes the request when it is in memory, within the transport, as well when it is being processed by one or more threads.

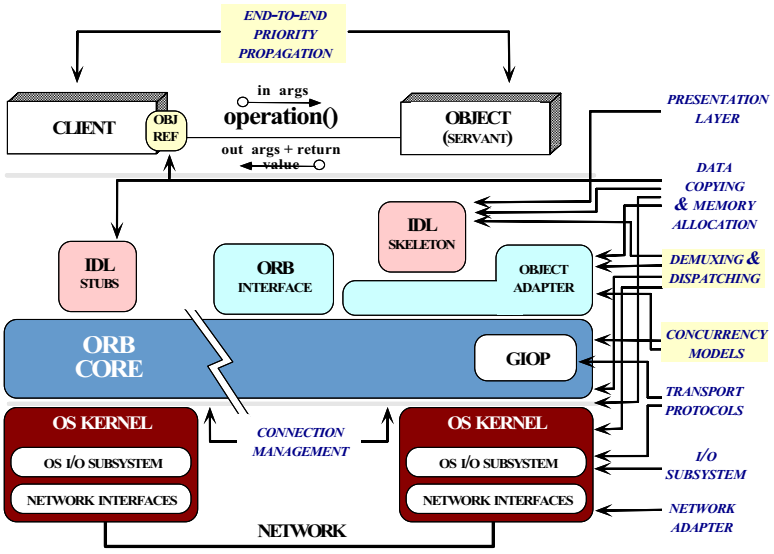


Figure 1.1: Thesis scope and related work

components. For example, a global scheduling service [66, 18] can be used to manage and schedule distributed resources.

This thesis focuses on the middleware layer and presents patterns for providing real-time, end-to-end timeliness guarantees. Figure 1.1 highlights the areas of this thesis, namely (1) end-to-end priority propagation, (2) dispatching and demultiplexing, and (3) concurrency models. Table 1.1 summarizes the challenges addresses and the contributions made by this research and Table 1.2 summarizes research done in related areas illustrated in Figure 1.1.

Our previous work has examined many dimensions of ORB middleware design, including static [66] and dynamic [18] operation scheduling, event processing [23], I/O subsystem [34] and pluggable protocol [48] integration, synchronous [68] and asynchronous [2] ORB Core architectures, IDL compiler features [1] and optimizations [22], systematic benchmarking of multiple ORBs [19], patterns for ORB extensibility [70], and ORB performance [54].

1.2 Thesis Organization

The rest of this thesis is organized as follows:

- Chapter 2 gives an overview of Real-time CORBA and describes the priority propagation models and thread pool features defined in the RT-CORBA specification.

Table 1.1: Summary of challenges addressed and contributions made by this research

Research Challenges	***Contributions***
Eliminate sources of unbounded priority inversion	1. Identify sources of unbounded priority inversion 2. Eliminate unbounded priority inversion by: - Using non-multiplexed resources where possible - Bounding priority inversion for shared resources
Bounding priority inversion in demultiplexing and dispatching	Document design patterns for predictable, scalable, efficient, flexible demultiplexing and dispatching
Concurrency architectures for RT applications	Document and evaluate design patterns for scalable and predictable concurrency architectures
COTS feasibility for RT applications	Empirical evaluation of end-to-end predictability of applications using RT-CORBA

Table 1.2: Summary of related research for providing predictable end-to-end behavior

Area	**Research**
Presentation Layer	Time/space tradeoffs of compiled vs. interpreted stubs [22].
Data Copying and Memory Allocations	Gather-write and scatter-read I/O calls to avoid excessive data copying and stack and TSS allocators to avoid heap allocations [55].
Transport Protocols	Principles for optimizing CORBA IIOP performance [21], and $a^I t^P m$, a synergistic combination of IP and ATM technologies to design a highly scalable gigabit IP router [50].
Connection Management	Non-multiplexed connection model to avoid priority inversion [67].
OS Scheduling	Empirical evaluation of context switching overhead and priority inversion overhead for several real-time and general purpose operating systems [36].
I/O Subsystems	RIO, an extensible and predictable I/O framework that can integrate seamlessly with real-time middleware [33].
Network Adapters	APIC, a high-performance ATM Port Interface Controller that supports efficient zero-copy buffer management by sharing request buffers across OS protection domains [9].

- Chapter 3 describes the demultiplexing layers in a CORBA server and shows how the unpredictability of naive demultiplexing schemes can lead to unbounded priority inversion. It then presents patterns for constant time demultiplexing that allows an ORB to provide real-time guarantees.
- Chapter 4 describes the challenges of dispatching in multi-threaded environments. Several dispatching patterns and their relative strengths and weaknesses are presented, including one that is ideal for real-time ORB because of its predictable behavior.
- Chapter 5 describes and evaluates patterns for implementing RT-CORBA thread pools. It explores issues of priority inversion, efficiency, and optimizations, in implementing thread pools.
- Chapter 6 traces the critical code path of a CORBA request and identifies potential predictability bottlenecks within the ORB. It then shows how the ORB can be redesigned to use non-multiplexed resources to eliminate these bottlenecks.
- Chapter 7 presents experiments that measure end-to-end predictability of the TAO ORB. Clients in these experiments feature threads of various priorities making rate monotonic invocations, along with best-effort threads trying to disrupt system predictability by stealing resources from threads of higher priorities. Servers feature thread pools with and without lanes.
- Chapter 8 summarizes the work presented in this thesis and suggests areas of future work.

1.3 Design Patterns

A *pattern* is a recurring solution to a standard problem within a particular context [16]. Patterns help researchers and developers communicate architectural knowledge, learn a new design paradigm or architectural style, and avoid traps and pitfalls that have been learned traditionally only through costly experience [7].

This thesis captures key design and performance characteristics of software components proven for their predictable, efficient, and scalable behavior in terms of patterns. Each pattern in this thesis resolves a particular set of forces, with varying consequences on performance, functionality, and flexibility. In general, simpler solutions result in better performance, but do not resolve all the forces that more complex solutions can handle. Application developers should not disregard simpler patterns, however. Instead, they should

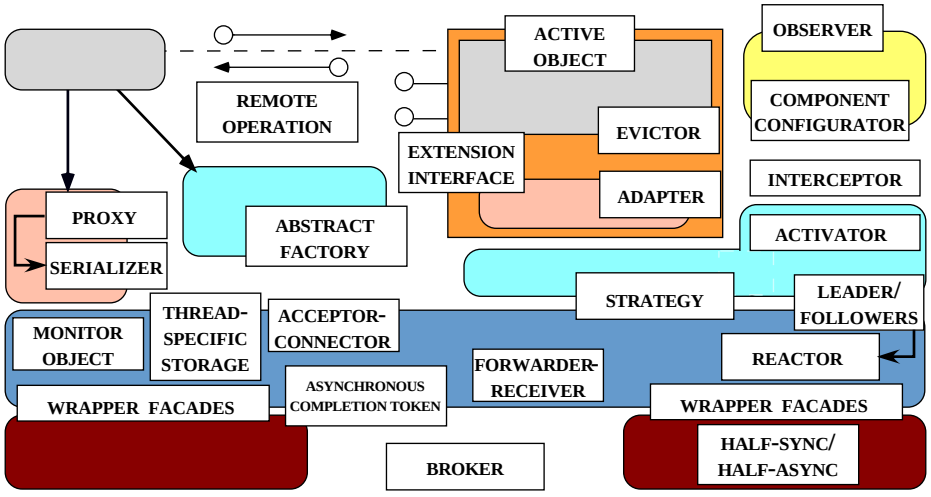


Figure 1.2: Patterns for real-time DOC middleware

apply the patterns that are most appropriate for the problem at hand, balancing the need to support advanced features with the performance and flexibility requirements of their applications.

Our long-term goal is to develop a “engineering handbook” of patterns for developing real-time distributed object computing (DOC) middleware as shown in Figure 1.2.

Chapter 2

Overview of Real-time CORBA

Abstract

This chapter provides an overview to the Common Object Request Broker Architecture (CORBA) [47] and describes the key components in the model. This chapter also introduces the Real-time CORBA specification [45] and illustrates the components and interfaces in the specification that can be used for propagating priorities end-to-end and for managing processor resources.

2.1 Introduction to CORBA

CORBA Object Request Brokers (ORBs) allow clients to invoke operations on distributed objects without concern for object location, programming language, OS platform, communication protocols and interconnects, and hardware [24]. Figure 2.1 illustrates the key components in the CORBA reference model [47] that collaborate to provide this degree of portability, interoperability, and transparency.[1] Each component in the CORBA reference model is outlined below:

Client: A client is a *role* that obtains references to objects and invokes operations on them to perform application tasks. A client has no knowledge of the implementation of the object but does know its logical structure according to its interface. It also doesn't know of the object's location - objects can be remote or collocated relative to the client. Ideally, a client can access a remote object just like a local object, *i.e.*, `object`→`operation(args)`.

[1] This overview only focuses on the CORBA components relevant to this thesis. For a complete synopsis of CORBA's components see [46].

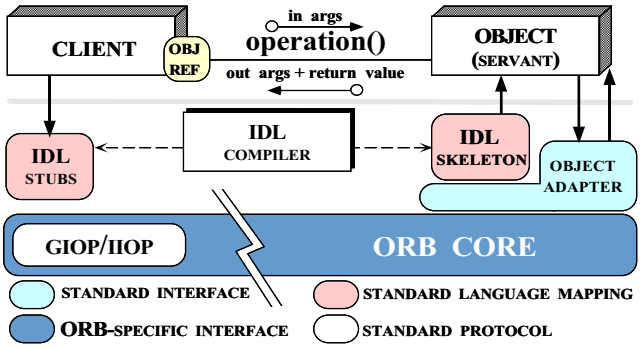


Figure 2.1: Key components in the CORBA 2.*x* reference model

Figure 2.1 shows how the underlying ORB components described below transmit remote operation requests transparently from client to object.

Object: In CORBA, an object is an instance of an OMG Interface Definition Language (IDL) interface. Each object is identified by an *object reference*, which associates one or more paths through which a client can access an object on a server. An *object ID* associates an object with its implementation, called a servant, and is unique within the scope of an Object Adapter. Over its lifetime, an object has one or more servants associated with it that implement its interface.

Servant: This component implements the operations defined by an OMG IDL interface. In object-oriented (OO) languages, such as C++ and Java, servants are implemented using one or more class instances. In non-OO languages, such as C, servants are typically implemented using functions and `structs`. A client never interacts with servants directly, but always through objects identified by object references.

ORB Core: When a client invokes an operation on an object, the ORB Core is responsible for delivering the request to the object and returning a response, if any, to the client. An ORB Core is implemented as a run-time library linked into client and server applications. For objects executing remotely, a CORBA-compliant ORB Core communicates via a version of the General Inter-ORB Protocol (GIOP), such as the Internet Inter-ORB Protocol (IIOP) that runs atop the TCP transport protocol. In addition, custom Environment-Specific Inter-ORB protocols (ESIOPs) can also be defined.

OMG IDL Stubs and Skeletons: IDL stubs and skeletons serve as a "glue" between the client and servants, respectively, and the ORB. Stubs implement the *Proxy* pattern [16] and marshal application parameters into a common message-level representation. Conversely,

skeletons implement the *Adapter* pattern [16] and demarshal the message-level representation back into typed parameters that are meaningful to an application.

IDL Compiler: An IDL compiler transforms OMG IDL definitions into stubs and skeletons that are generated automatically in an application programming language, such as C++ or Java. In addition to providing programming language transparency, IDL compilers eliminate common sources of network programming errors and provide opportunities for automated compiler optimizations [11].

Object Adapter: An Object Adapter is a composite component that associates servants with objects, creates object references, demultiplexes incoming requests to servants, and collaborates with the IDL skeleton to dispatch the appropriate operation upcall on a servant. Object Adapters enable ORBs to support various types of servants that possess similar requirements. This design results in a smaller and simpler ORB that can support a wide range of object granularities, lifetimes, policies, implementation styles, and other properties. Even though different types of Object Adapters may be used by an ORB, the only Object Adapter defined in the CORBA specification is the Portable Object Adapter (POA).

2.2 Overview of Real-time CORBA

Historically, CORBA has lacked features that allow applications to allocate, schedule, and control key CPU, memory, and networking resources necessary to ensure end-to-end quality of service. The Real-time CORBA (RT-CORBA) 1.0 specification [45] defines standard features shown in Figure 2.2 that support end-to-end predictability for operations in *fixed-priority* CORBA applications. RT-CORBA includes standard interfaces and QoS policies that allow applications to configure and control the following:

- *Processor resources* via thread pools, priority mechanisms, intra-process mutexes, and a global scheduling service;
- *Communication resources* via protocol properties and explicit bindings; and
- *Memory resources* via buffering requests in queues and bounding the size of thread pools.

Applications typically specify these real-time QoS policies along with other policies when they invoke standard ORB operations. For instance, when an object reference is created

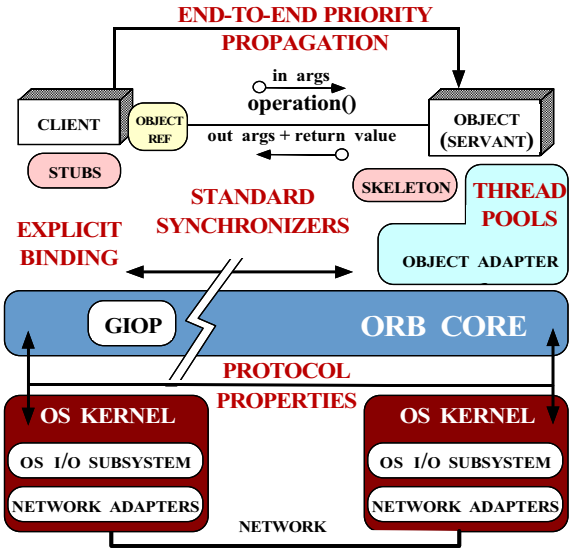


Figure 2.2: ORB endsystem features for RT-CORBA

using a QoS-enabled Object Adapter, the Object Adapter ensures that any server-side policies that affect client-side requests are embedded within a *tagged component*[2] in the object reference. This enables clients who invoke operations on such object references to honor the policies required by the target object.

Strict control over the scheduling and execution of processor resources is essential for many fixed-priority real-time applications. Therefore, RT-CORBA enables client and server applications to (1) determine the priority at which CORBA invocations will be processed and (2) allow servers to pre-define pools of threads to service incoming invocations.

It is important to recognize that RT-CORBA's priority mechanisms cannot overcome inherent sources of non-determinism. In particular, ORB middleware cannot imbue a non-real-time OS or communication infrastructure with completely deterministic behavior [36]. When used in the appropriate environment, however, certain RT-CORBA features help application developers and integrators configure heterogeneous systems to preserve priorities end-to-end, as described below.

2.3 Propagating Priorities with RT-CORBA

Conventional [46] CORBA ORBs provide no standard way for clients to indicate the relative priorities of their requests to ORB endsystems. This feature is necessary, however, to

[2]Tagged components are name/value pairs that can be used to export attributes, such as security or QoS values, from a server to its clients within object references [46].

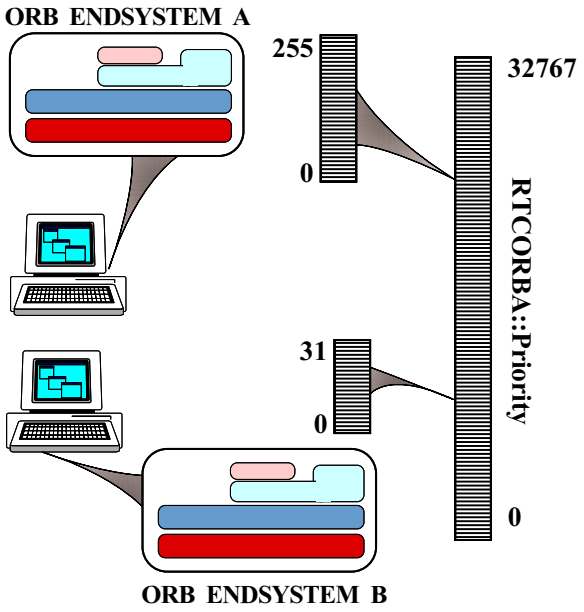


Figure 2.3: Mapping CORBA priorities to native priorities

reduce end-to-end priority inversion, as well as to bound latency and jitter for applications with deterministic real-time QoS requirements. Priority inversion is a scheduling hazard that occurs when a low priority thread or request blocks the execution of a higher priority thread or request. Therefore, RT-CORBA defines the following platform-independent mechanisms to specify the priority of operation invocations.

2.3.1 Priority Type System

RT-CORBA defines two types of priorities – *CORBA* and *native* – to handle OS heterogeneity. Each CORBA operation can be assigned a CORBA priority, which ranges in value between 0 and 32767, 0 being the minimum while 32767 is the maximum. Each ORB endsystem along an activity path can be customized to map CORBA priorities to native priorities, which may be unique on different endsystems. Figure 2.3 illustrates how CORBA priorities can be mapped onto two different native ORB endsystem priorities.

2.3.2 Priority Models

RT-CORBA defines a *PriorityModel* policy with two values, `SERVER_DECLARED` and `CLIENT_PROPAGATED`, as described below.

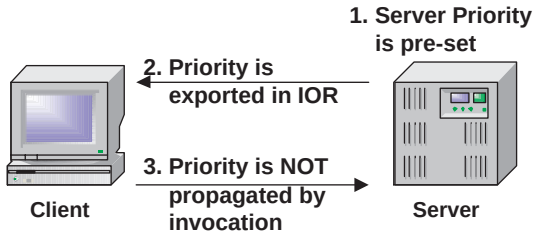


Figure 2.4: SERVER_DECLARED RT-CORBA priority model

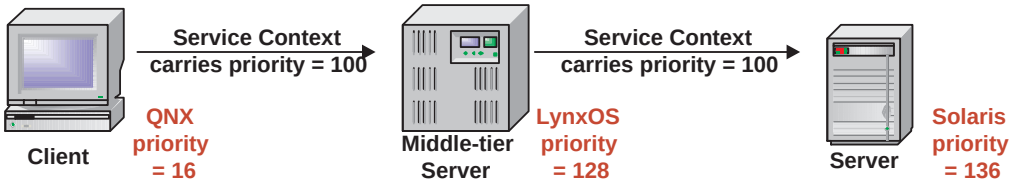


Figure 2.5: CLIENT_PROPAGATED RT-CORBA priority model

Server declared priorities: This model allows a server to dictate the priority at which an invocation made on a particular object will execute. In the server declared model, the priority is designated *a priori* by the server and is encoded into the object reference published to the client, as shown in Figure 2.4. Although the server declares the priority, the client is aware of the selected priority and can use this information to match its priority with the server selected priority to minimize end-to-end priority inversion.

Client propagated priorities: Although the server declared model is useful for certain real-time applications, it is not suited for all application use-cases. For instance, one way for a server to avoid priority inversion is to process incoming requests at a priority of the client thread [58]. The RT-CORBA client propagated model allows clients to declare invocation priorities that must be honored by servers. In this model, each invocation carries the CORBA priority of the operation in the service context list that is tunneled with its GIOP request. Each ORB endsystem along the activity path between the client and server maps this end-to-end CORBA priority to a native OS priority and processes the request at this priority. Moreover, if the client invokes a two-way operation, its CORBA priority will determine the priority of the reply.

Figure 2.5 depicts the case where an invocation from a client to a server goes through a middle-tier server. Each host has a different operating system with different native thread priority ranges. The CORBA priority of the client is propagated with the request. Each

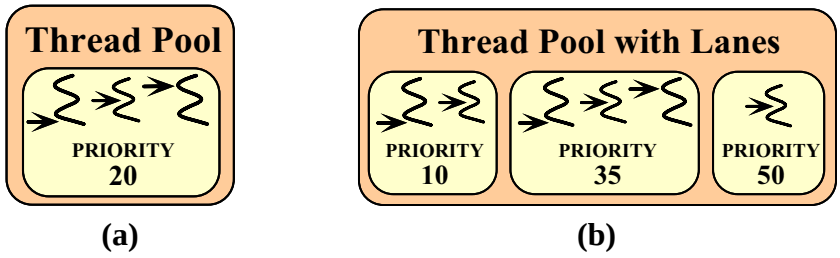


Figure 2.6: **(a)** Thread pool without lanes; **(b)** Thread pool with lanes

intervening server along the activity path maps the client's CORBA priority to a native priority that is appropriate for its host platform. For example, on LynxOS, the global CORBA priority of 100 can be mapped to a native OS priority of 128. Likewise, on Solaris, the same global CORBA priority can be mapped to a real-time thread with a priority of 136.

2.4 Managing Processor Resources with RT-CORBA Thread Pools

Many real-time systems use multi-threading to (a) distinguish between different types of service, such as high-priority vs. low-priority. tasks [23]; (b) support thread preemption to prevent unbounded priority inversion and deadlock; and (c) support complex object implementations that run for variable and/or long durations. To allow real-time ORB endsystems and applications to leverage these benefits of multi-threading, while controlling the amount of memory and processor resources they consume, RT-CORBA defines a server *thread pool* model [68]. There are two types of thread pools in RT-CORBA:

- *Thread pool without lanes* – All threads in this basic thread pool model have the same assigned priority. This model is illustrated in Figure 2.6 (a).
- *Thread pool with lanes* – Threads in this advanced thread pool model are divided into *lanes* that are assigned different priorities. This model is illustrated in Figure 2.6 (b).

Each thread pool is then associated with one or more POA(s). The threads in a pool perform processing of client requests targeted at its associated POA(s). While a thread pool can be associated with more than one POA, a POA can be associated with only one thread pool. Figure 2.7 illustrates the creation and association of thread pools in a server.
A thread pool is configured with the following properties:

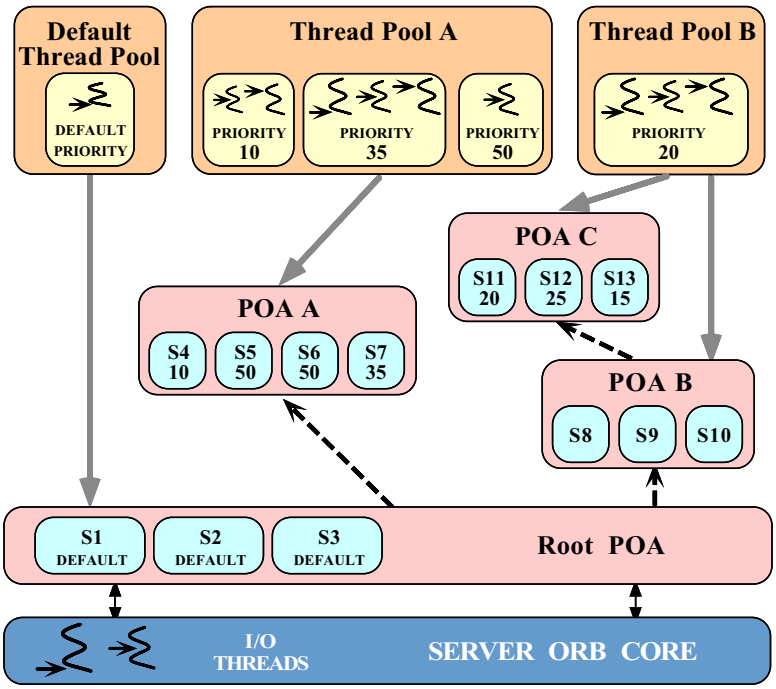


Figure 2.7: POAs and thread pools in RT-CORBA

- *Static threads* defines the number of threads in the thread pool pre-allocated at thread pool creation time.
- *Dynamic threads* defines the maximum number of threads that can be created on-demand. If a request arrives when all existing threads are busy, a new thread is created to handle the request if the number of dynamic threads in the pool has not exceeded the *dynamic* value specified by the user.

  The ability to configure the number of threads allows developers to bound the processing resources. Also, developers can choose between dynamic and static threads to trade off the jitter introduced by dynamic thread creation/destruction with the wastefulness of underutilized static threads.
- *Priority* defines the CORBA priority with which threads are created. Depending on the *policies* configured in the ORB, the priority of the threads can change subsequently. The priority of threads in thread pools with lanes does not change except when *thread borrowing* is used as described below. The priority of a thread in a thread pool without lanes is changed to match the priority of a client making the request. POA B serviced by thread pool B in Figure 2.7 illustrates this scenario. The

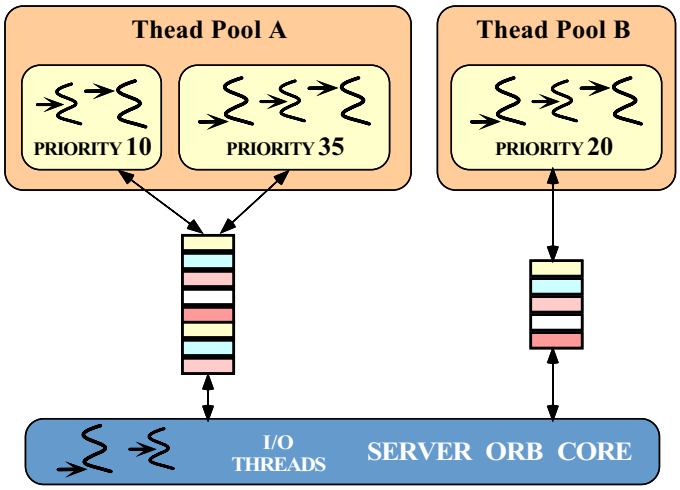


Figure 2.8: Buffering requests in RT-CORBA thread pools

priority of a thread in a thread pool without lanes is also changed to match the priority of the servant that uses this thread. POA C serviced by thread pool B in Figure 2.7 illustrates this scenario. The priority of the thread is restored after the client request has been processed.

- *Stack size* defines in bytes the size of stack space allocated for each thread.
- *Request buffering* bounds the maximum client request buffering resources used when all threads are busy, specified in number of bytes or requests. If a request arrives when all threads are busy and the buffering space is exhausted, the ORB should raise a TRANSIENT exception, that indicates a temporary resource shortage. When a client receives this exception it can reissue the request at a later time. Figure 2.8 illustrates the thread pool request buffering feature.
- *Thread borrowing* controls whether a thread lane is allowed to "borrow" threads from lower priority lanes when it exhausts its maximum number of threads (both static and dynamic) and requires an additional threads to service new requests. The borrowed thread has its priority raised to that of the lane that requires it. When the thread is no longer required, its priority is lowered back to its previous value, and it is returned to the lower priority lane. This property applies only to thread pools with lanes.

Static threads, *dynamic threads*, and *priority* are per-lane properties in thread pool with lanes model.

2.5 Concluding Remarks

The Real-time CORBA (RT-CORBA) 1.0 specification [45] offers solutions to resource management challenges facing researchers and developers of real-time systems, particularly when applications can be designed using fixed priority scheduling. However, for applications that execute under dynamic load conditions [18] and cannot determine the priorities of various operations *a priori* without significantly underutilizing various resources, the OMG is standardizing dynamic scheduling [43] techniques, such as deadline-based [78] or value-based [29] scheduling. This thesis focus primarily on systems that require fixed priority scheduling.

Chapter 3

Scalable, Predictable, and Efficient Request Demultiplexing

Abstract

Time spent by an ORB demultiplexing requests to servants can constitute a significant source of overhead for real-time applications [20]. This chapter describes how the demultiplexing strategies used impact the scalability and predictability of real-time ORBs. This chapter also illustrates how optimizations can enable constant time request demultiplexing in the average- and worst-case, regardless of organization of the POA hierarchy or number of POAs, servants, or operations configured in an ORB.

3.1 Introduction to Demultiplexing CORBA Requests

Demultiplexing requires routing requests through the layers of a server endsystem as shown in Figure 3.1. A CORBA request contains the identity of its object and operation. An object is identified by an object key, which is a sequence of octets. An operation is represented as a string. An ORB has to perform the following three layers of demultiplexing to deliver a CORBA request to the target object implementation, *i.e.*, servant, once the request has been read by the ORB I/O subsystem:

POA Layer: The first demultiplexing layer locates the POA where the target servant has been registered. POAs can be nested arbitrarily. Although nesting provides a useful way to organize policies and namespaces hierarchically, it complicates demultiplexing.

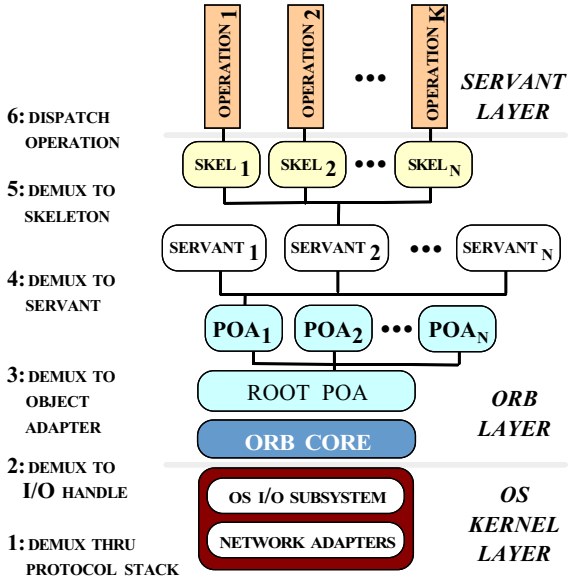


Figure 3.1: Demultiplexing layers in a CORBA server

Servant Layer: The object key identifies the target POA and the target servant. Once the target POA is found, the next demultiplexing layer finds the target servant.

Skeleton Layer: Once the target servant is found, the final demultiplexing layer finds the target skeleton. The operation name identifies the appropriate IDL skeleton that demarshals the request buffer into operation parameters and performs the upcall to code supplied by servant developers to implement the object's operation.

3.1.1 Organization of a Prototypical Server

Figure 3.2 shows the organization of a prototypical server. The POA layer has several hierarchically organized POAs, each providing a unique namespace for the servants registered with it. The first POA level in this example server represents television networks, namely *Fox* and *Disney*. The second POA level represents television shows hosted by these networks, namely, *Simpsons* and *King of the Hill* by the *Fox* network and *Winnie the Pooh* by the *Disney* network.

The servant layer represents the characters on the television shows. Since *Simpsons* has a large number of characters appearing on the show, an extra POA layer is added to further organize the characters into *Family* and *Townspeople*. Each of the leaf POAs has

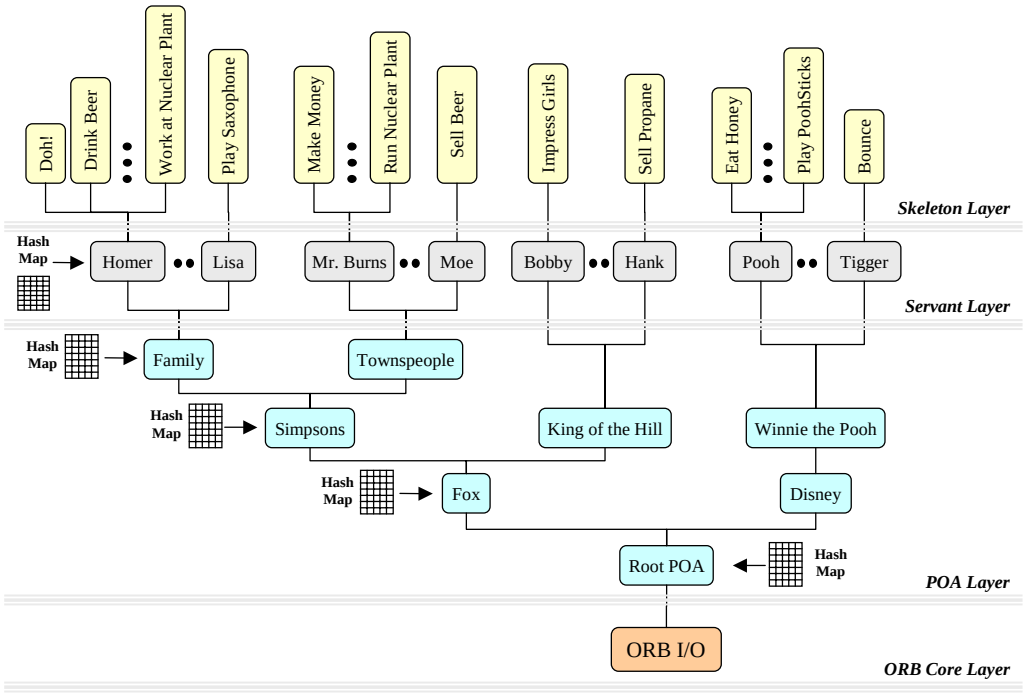


Figure 3.2: Organization of a prototypical server

several servants registered with it[1]. For example, *Homer* and *Lisa* are registered with the POA representing *Simpsons Family* members.

Finally, the skeleton layer represents favorite habits or actions by the characters in the television shows. This includes *Drinking Beer* and exclaiming *Doh!* by *Homer* and *Eating Honey* and *Playing PoohSticks* by *Winnie the Pooh*.

3.2 A Simple Demultiplexing Scheme

A simple way of demultiplexing a request would be to perform a dynamic hash lookup at each demultiplexing step. For example, to perform the *Doh!* operation on *Homer* who is part of the *Simpsons Family* on the *Fox* network, this demultiplexing scheme would:

1. Lookup *Fox* network POA on the *RootPOA*;
2. Lookup *Simpsons* television show POA on the *Fox* POA;
3. Lookup *Family* category POA on the *Simpsons* POA;

[1]It is not necessary that only the leaf POAs have servants register with them. Any POA, including the *RootPOA* can have servants registered with it.

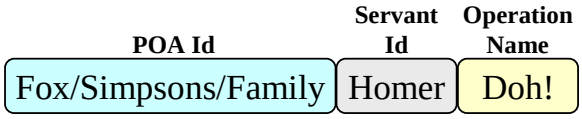


Figure 3.3: Simple identification of target object and operation

4. Lookup *Homer* character servant on the *Family* POA;
5. Lookup *Doh!* action skeleton on the *Homer* servant.

Figure 3.3 shows a request header that identifies the object and operation that should be invoked on the server. The object key, made up of the POA ID and the servant ID, contains the complete name of the target POA[2] and the name of the servant.

3.2.1 Shortcomings of Simple Demultiplexing Scheme

The simple demultiplexing scheme described in Section 3.2 is generally inappropriate for high-performance and real-time applications for the following reasons [74]:

Variable number of lookups required: The number of lookups required to reach the target POA depends on the organization of the POA hierarchy on the server. To reach the *Fox/Simpsons/Family* POA, three dynamic hash lookups were required, while two lookups were required to reach the *Disney/Winnie the Pooh* POA.

High worst-case time for dynamic hashing: Dynamic hashing provides $O(1)$ performance for the average case. However, due to the potential for collisions, its worst-case execution time is $O(n)$, where n is the number of entries in the map.

Server reorganization leads to new worst-case time: Even though dynamic hashing has a high worst-case execution time, the worst-case demultiplexing time can be calculated given the depth of the POA hierarchy, the number of POAs at each level of the hierarchy, the number of servants in each POA, and the number of operations for each servant. Unfortunately, that calculation is invalidated if the server is reorganized either to add or remove additional levels to the POA hierarchy or to change the number of POAs, servants, or operations. In addition, when calculating the worst-case demultiplexing time, the maximum number of the above parameters has to be considered, even though they may fluctuate considerably during the lifetime of the server. This may lead to highly underutilized systems.

[2] A complete POA name includes the name of the POA plus the names of all its ancestors.

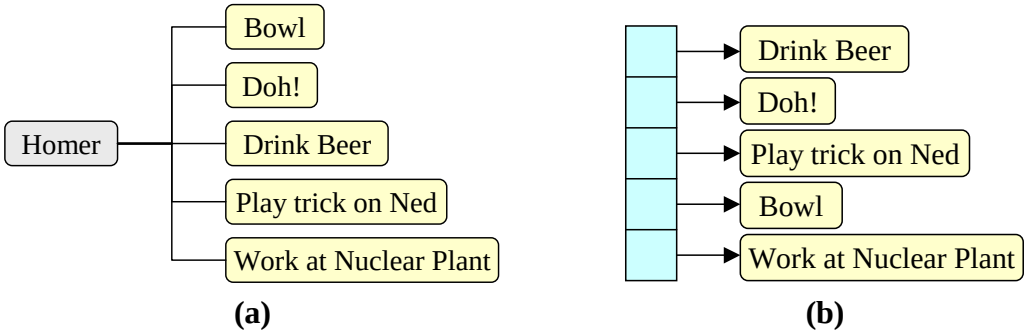


Figure 3.4: **(a)** Skeleton demultiplexing layer; **(b)** Perfect hashing used in the skeleton layer

3.3 Scalable, Predictable, and Efficient CORBA Request Demultiplexing

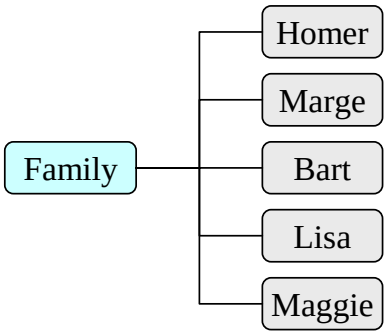
Optimizing a POA to support real-time applications requires the resolution of several design challenges. This section outlines these challenges and describes the patterns we applied to improve the predictability, performance, and scalability of each demultiplexing layer.

Skeleton demultiplexing: Figure 3.4 (a) provides a closer look at the demultiplexing occurring in the skeleton layer. The lookup key for this layer is the operation name defined by developers when declaring the IDL interface[3]. Since an IDL interface is defined *a priori*, all the operation names are known at compile time.

Given the above constraints, *perfect hashing* is suitable for demultiplexing in this layer. Perfect hashing improves on dynamic hashing by pre-computing a collision-free *perfect hash function* [62]. Figure 3.4 (b) illustrates a hash map used in this layer which executes in constant time because there are no collisions.

Servant demultiplexing: Figure 3.5 (a) provides a closer look at the demultiplexing occurring in the servant layer. Although the number and names of operations is known *a priori*, the number and names of servants are generally dynamic. However, it is possible to have a custom representation of the servant ID in the object key since the object key is ORB-specific and needs to be evaluated only by the ORB that created it.

[3]It is not possible to modify the operation name to include any additional indexing information without violating the GIOP protocol.



(a)

Index	Generation Count	Servant
0	5	Homer
1	2	Marge
2	1	Bart
3	10	Lisa
4	3	Maggie

(b)

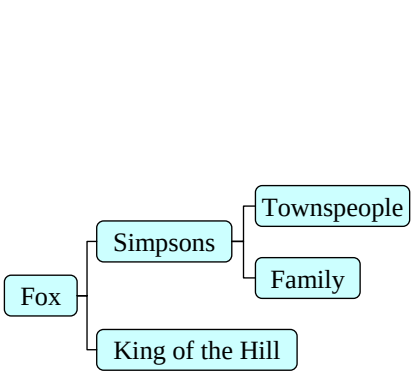
Figure 3.5: **(a)** Servant demultiplexing layer; **(b)** Active demultiplexing used in the servant layer

Given the above constraints, *active demultiplexing* [20] is suitable for demultiplexing in this layer. Active demultiplexing replaces the servant name with direct indexing information. Figure 3.5 (b) illustrates an active map used in this layer that executes in constant time because of direct indexing[4]. Active demultiplexing has low latency and is highly predictable. In contrast, dynamic hashing incurs higher constant overhead to compute the hash function. Moreover, unlike in active demultiplexing, the performance of dynamic hashing degrades gradually as the number of servants increases.

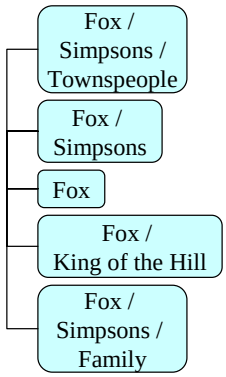
POA demultiplexing: Figure 3.6 (a) provides a closer look at the demultiplexing occurring in the POA layer. Similar to the servant layer, the number and names of POAs are generally dynamic. And similar to the servant ID, it is possible to have a custom representation of the POA ID in the object key. However, unlike the servant layer, POAs can be organized in a hierarchy.

To alleviate the problem of performing multiple lookups, one of each level of the POA hierarchy, the POA hierarchy is flattened as shown in Figure 3.6 (b). Even though the POA hierarchy is flattened for demultiplexing purposes, it is logically still the same: a POA can be asked for its parent or for the list of its children. Once flattened, active demultiplexing is suitable for demultiplexing in this layer. Figure 3.6 (c) illustrates an active map used in this layer that executes in constant time because of direct indexing. Active demultiplexing for the POA layer provides optimal predictability and scalability, just as it does when used for the servant layer.

[4]A *generation count* in the active map allows recycling of indexes as old servants are removed and new servants are added to the map.



(a)



(b)

Index	Generation Count	POA
0	5	Fox
1	2	Fox / Simpsons
2	1	Fox / Simpsons / Family
3	10	Fox / Simpsons / Townpeople
4	3	Fox / King of the Hill

(c)

Figure 3.6: **(a)** POA demultiplexing layer; **(b)** Flattened POA demultiplexing layer; **(c)** Active demultiplexing used in POA layer

3.3.1 Summary of Optimized Demultiplexing Strategies

Figure 3.7 summarizes the demultiplexing strategies most appropriate for real-time applications [23]. The POA hierarchy is flattened thus requiring only one active lookup. Active demultiplexing is used for the servant layer and perfect hashing is used for the skeleton layer.

Figure 3.8 shows the revised request header that identifies the object and operation that should be invoked on the server. The object key now contains active demultiplexing keys instead of the complete name of the POA and the name of the servant.

Table 3.1 [55] depicts the time in microseconds (μs) spent in each layer as a server processes a request on the quad-CPU 400 MHz Pentium II Xeon. The key observation is that the times presented in table are independent of the organization of the server. Therefore, unlike the case of the simple demultiplexing scheme presented in Section 3.2, a change in the POA hierarchy structure, or a change in the number of POAs, servants, or operations will not require a reevaluation of the worst-case execution time.

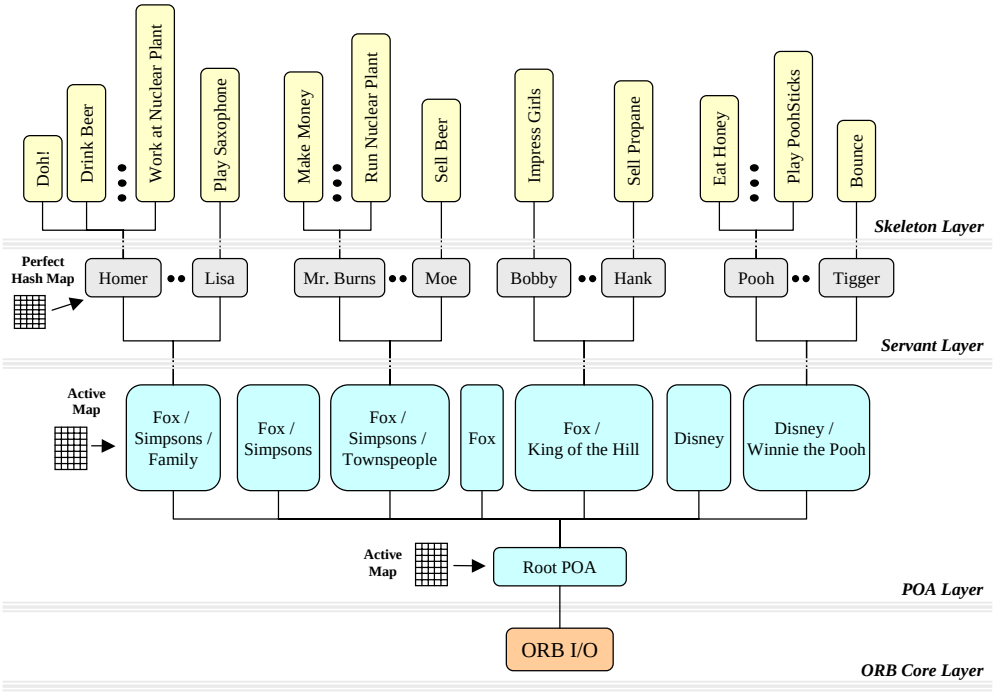


Figure 3.7: Optimized organization of a prototypical server

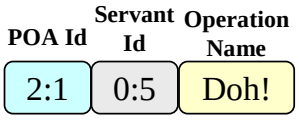


Figure 3.8: Optimized identification of target object and operation

Table 3.1: Time spent in each demultiplexing step

Demultiplexing Stage	*Absolute Time (µs)*
1. Parsing Object Key	2
2. POA demux	2
3. Servant demux	3
4. Operation demux	3
5. Parameter demarshal	Operation dependent
6. User upcall	Servant dependent
7. Return value marshal	Operation dependent

3.4 Related Work

Most I/O subsystems demultiplex messages through several layers of the protocol stack. Likewise, CORBA ORBs perform several extra levels of demultiplexing at the middleware layer to associate incoming client requests with the appropriate servant and operation. Related work on demultiplexing focuses largely on the lower layers of the protocol stack, *i.e.*, the transport layer and below, as opposed to the middleware layer. For instance, [74, 15, 9, 12] study demultiplexing issues in communication systems and show how layered demultiplexing is not suitable for applications that require real-time QoS guarantees.

Packet filters are a mechanism for efficiently demultiplexing incoming packets to application endpoints [41]. A number of schemes to implement fast and efficient packet filters are available. These include the BSD Packet Filter (BPF) [39], the Mach Packet Filter (MPF) [80], PathFinder [3], demultiplexing based on automatic parsing [28], and the Dynamic Packet Filter (DPF) [12].

3.5 Concluding Remarks

Demultiplexing in conventional CORBA implementations is typically inefficient and unpredictable. For instance, [19, 20] show that conventional ORBs spend $\sim$17% of the total server time processing demultiplexing requests. This chapter describes methodologies that demultiplex predictably while reducing average- and worst-case overhead. Constant time request demultiplexing *regardless* of organization of the POA hierarchy or number of POAs, servants, or operations, allows an ORB to provide uniform, scalable QoS guarantees to real-time applications.

Chapter 4

Patterns for Efficient, Predictable, Scalable, and Flexible Dispatching Components

Abstract

Dispatching components are responsible for delivering upcalls to one or more application objects when events or requests arrive in a system. Implementing efficient, predictable, and scalable dispatching components is challenging, and implementing them for multi-threaded systems is even more challenging. In particular, dispatching components must be prepared to (1) deliver the same upcall to multiple objects, (2) dispatch multiple requests simultaneously, (3) handle recursive requests originating from application-provided upcalls, (4) collaborate with applications to control object life-cycles, and (5) add and remove objects in dispatching tables while upcalls are in progress.

In our distributed object computing (DOC) middleware research, we have implemented many dispatching components that apply common solutions repeatedly to solve the challenges outlined above. Moreover, we have discovered that the forces constraining dispatching components often differ slightly, thereby requiring alternative solution strategies. This chapter presents a set of patterns that describe successful solutions appropriate for key dispatching challenges arising in various real-time DOC middleware and applications.

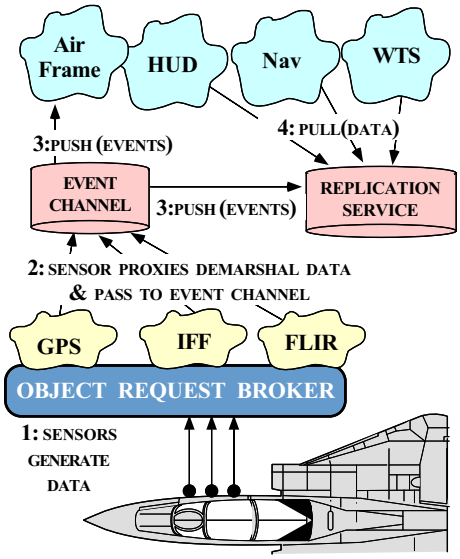


Figure 4.1: Multiple dispatching components in DOC middleware

4.1 Introduction to Dispatching

This chapter presents a family of related patterns that we have used to develop efficient, predictable, and scalable dispatching components in a variety of application domains, an example of which is shown in Figure 4.1. These domains include real-time avionics mission computing with strict periodic dead-line requirements [23] and distributed interactive simulations with high scalability requirements [49]. In addition, various dispatching-oriented framework components, such as Reactors [64], Proactors [52], Observers [16], and Model-View-Controllers [5] are implemented using these patterns.

This section summarizes the functionality and requirements of two common use-cases that illustrate the challenges associated with developing dispatching components. In the first use-case, events are dispatched to a single object, *e.g.*, through a CORBA ORB. The second use-case occurs when events are dispatched to multiple objects, *e.g.*, through an Event Channel [42].

Object Adapter dispatching components: The core responsibilities of a CORBA Object Adapter include (1) generating identifiers for objects that are exported to clients and (2) mapping subsequent client requests to the appropriate object implementations, that CORBA calls *servants*. Figure 4.2 illustrates the general structure and interactions of a CORBA Object Adapter. In addition to its core responsibilities, a CORBA Object Adapter must handle the following situations correctly, robustly, and efficiently:

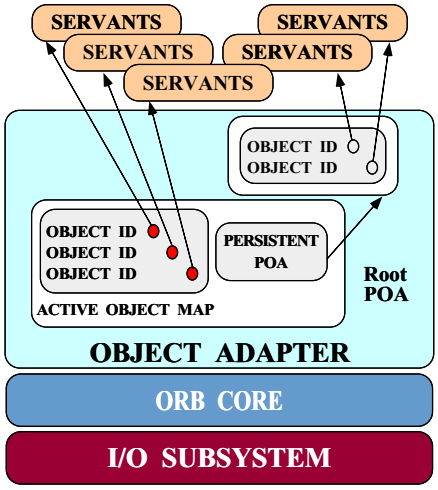


Figure 4.2: Object Adapter structure and interactions

- *Non-existent objects:* Clients may invoke requests on "stale" identifiers, *i.e.*, on objects that have been deactivated from the Object Adapter. In this case, the Object Adapter should not use the stale object because it may have been deleted by the application. Instead, it must propagate an appropriate exception back to the client.
- *Unusual object activation/deactivation:* Object Adapters are responsible for activating and deactivating objects on-demand. Moreover, server application objects can activate or deactivate other objects in response to client requests. An object can even deactivate itself while in its own upcall, *e.g.*, if the request is a "shut yourself down" message.
- *Multi-threading hazards:* Implementing an Object Adapter that works correctly and efficiently in a multi-threaded environment is challenging. For instance, there are many opportunities for deadlock, unduly reduced concurrency, and priority inversion that may arise from recursive calls to an Object Adapter while it is dispatching requests. Likewise, excessive synchronization overhead may arise from coarse-grained or imprecise locking performed on a dispatching table.

Event Channel dispatching components: The CORBA Event Service defines participants that provide a asynchronous and decoupled type of communication service that alleviates some restrictions [23] with the standard synchronous CORBA ORB operation invocation models. As shown in Figure 4.3 *suppliers* generate events and *consumers* process events received from suppliers. This figure also illustrates the *Event Channel*, which is

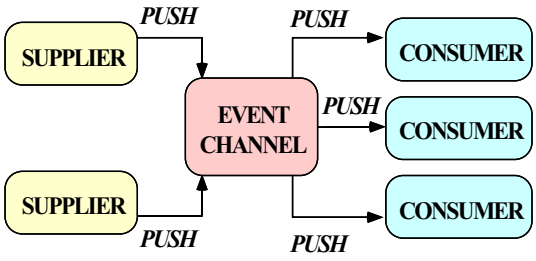


Figure 4.3: Participants in the COS Event Service architecture

a mediator [16] that dispatches events to consumers on behalf of suppliers. By using an Event Channel, a supplier can deliver events to one or more consumers without requiring any of these participants to know about each other explicitly.

Components that dispatch events to multiple objects must address a different set of challenges relative to components dispatching to a single object. These challenges include managing additions and removals to consumer subscriptions while dispatching to multiple consumers is in progress. Patterns that describe and address these challenges are presented in [53]. This chapter focuses mainly on components that dispatch to a single object.

Historically, a variety of *ad hoc* strategies have emerged to address the dispatching challenges outlined above. No one strategy is optimal for all application domains or use-cases, however. For instance, real-time implementations may impose too much overhead for high-performance, "best-effort" systems. Likewise, implementations tailored for multi-threading may impose excessive locking overhead for single-threaded reactive systems. In addition, strategies that support recursive access can incur excessive overhead if all upcalls are dispatched to separate threads or remote servers. Thus, what is required are strategies and methodologies that systematically capture the range of possible solutions that arise in the design space of dispatching components. One family of these strategies is described in the following section.

4.2 Patterns for Dispatching to a Single Object

Certain patterns, such as Strategized Locking [65] or Strategy [16] address some of the challenges associated with developing efficient, predictable, scalable, and flexible dispatching components. In other cases, however, the relationships and collaborations between dispatching components require more specialized solutions. Moreover, as noted in Section 4.1, no single pattern or strategy alone resolves all the forces faced by developers of

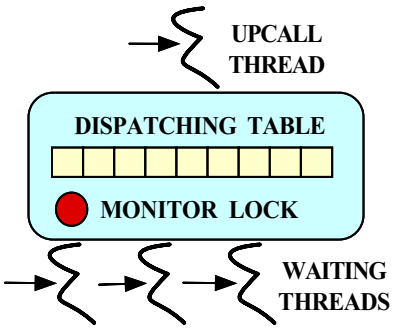


Figure 4.4: Serialized dispatching with a Monitor lock

complex dispatching components. Therefore, this section presents *patterns* that addresses the challenges for dispatching components outlined in Section 4.1. The initial patterns are relatively straightforward and are intended for less complex systems. The latter patterns are more intricate and address more complex requirements for efficiency, predictability, scalability, and flexibility.

4.2.1 Serialized Dispatching

Context: Dispatching components typically contain a collection of target objects that reside in one or more dispatching tables. These tables are used to select appropriate objects based upon identifiers contained in incoming requests.

Problem: Multi-threaded applications must serialize access to their dispatching table(s) to prevent data corruption.

Forces: Serialization mechanisms, such as mutexes or semaphores, should be used carefully to avoid excessive locking, priority inversion, and non-determinism. High-performance and real-time systems can maximize parallelism by minimizing serialization. However, application correctness cannot be sacrificed to improve performance, *e.g.*, a multi-threaded applications should be able to add and remove objects registered with the dispatching table efficiently during run-time without corrupting the dispatching table.

Solution: Serialize dispatching of requests by using the Monitor Object pattern [70] where a single monitor lock serializes access to the entire dispatching table, as shown in Figure 4.4. The monitor lock is held both while (1) searching the table to locate the object and (2) dispatching the appropriate operation call on the application-provided code.

In addition, the same monitor lock is used when inserting and removing entries from the table.

Consequences: A regular monitor lock is sufficient to achieve the level of serialization necessary for this dispatching component. Serialization overhead is minimal since only one set of acquire/release calls is made on the lock during an upcall. Thus, this design is appropriate when there is little or no contention for the dispatching table or when upcalls to application code are short-lived.

A simple protocol can control the life-cycle of objects registered with the dispatching component. For instance, an object cannot be destroyed while it is still registered in the dispatching table. Since the table's monitor lock is used for dispatching and modifying the table, other threads cannot delete an object that is in the midst of being dispatched.

However, this pattern may be inadequate for systems with stringent real-time requirements. In particular, the monitor lock is held during the execution of application code, which makes it hard for the dispatching component to predict how long it will take to release the monitor lock. Likewise, this pattern does not work well when there is significant contention for the dispatching table. For instance, if two requests arrive simultaneously for different target objects in the same dispatching table, only one of them can be dispatched at a time.

4.2.2 Serialized Dispatching with a Recursive Mutex

Context: Same as outlined in Section 4.2.1.

Problem: Monitor locks are not recursive on many OS platforms. When using non-recursive locks, attempts to query or modify the dispatch table while holding the lock will cause deadlock. Thus application upcall code cannot query or modify the dispatch table since it is called while the lock is held.

Forces: A monitor lock cannot be released before dispatching the application upcall because another thread could remove and destroy the object while it is still being dispatched.

Solution: Serialize dispatching of requests by using a *recursive* monitor lock [51]. A recursive lock allows the calling thread to re-acquire the lock if that thread already owns it. The structure of this solution is identical to the one shown in Figure 4.4, except that a recursive monitor lock is used in lieu of a non-recursive lock.

Consequences: As before, the monitor lock serializes concurrent access to avoid corruption of the dispatching table. Unlike the Serialized Dispatching pattern outlined in Section 4.2.1, however, application upcalls can modify the dispatching table or dispatch new upcalls.

Unfortunately, this solution does not resolve the concurrency and predictability problems since the monitor is held through the upcall. In particular, it is (1) still hard for the dispatching component to predict how long the monitor lock will be held and (2) the component does not allow multiple requests to be dispatched simultaneously. Moreover, recursive monitor locks are usually more expensive than their non-recursive counterparts [63].

4.2.3 Dispatching with a Readers/Writer Lock

Context: In complex DOC middleware and applications, requests often arrive simultaneously. Unless application upcalls are sharing resources that must be serialized, these operations should be dispatched and executed concurrently. Even if hardware support is not available for parallel execution, it may be possible to execute events and requests concurrently by overlapping CPU-intensive operations with I/O-intensive operations.

Problem: Serialized Dispatching patterns are inefficient for implementing concurrent dispatching upcalls since they do not distinguish between read and write operations, and thus serialize all operations on the dispatching table.

Forces: Although dispatching table modifications typically require exclusive access, dispatching operations does not. However, the dispatching component must ensure that the table is not modified while a thread is dispatching an upcall.

Solution: Use a readers/writer lock to serialize access to the dispatching table. The critical path, *i.e.*, looking up the target object and invoking an operation on it, does not modify the table. Therefore, a `read` lock will suffice for this path. Operations that modify the dispatching table, such as adding or removing objects from it, require exclusive access. Therefore, a `write` lock is required for these operations. Figure 4.5 illustrates the structure of this solution, where multiple reader threads can dispatch operations concurrently, whereas writer threads are serialized.

Consequences: Readers/writer locks allow multiple readers to access a shared resource simultaneously, while only allowing one writer to access the shared resource at a time. Thus, the solution described above allows multiple concurrent dispatch calls.

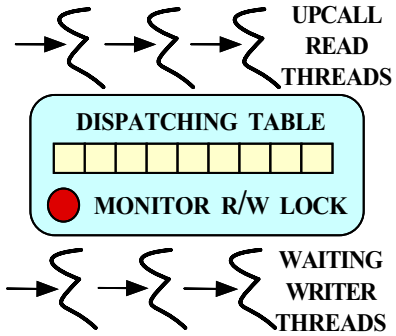


Figure 4.5: Dispatching with a Readers/Writer lock

Some DOC middlewares execute the upcall in a separate thread in the same process or on a remote object. Other middlewares execute the upcall in the same thread after releasing the `read` lock. Yet others don't allow upcalls to make changes to the dispatch table. Thus, this readers/writer locking pattern [51] can be applied to such systems without any risk of deadlocks. However, this solution is not applicable to systems that change the table from within an upcall while the `read` lock is held. This would require upgrading the readers/writer lock from a `read` lock to a `write` lock. Unfortunately, standard readers/writer lock implementations, such as Solaris/UI threads [13], do not support upgradable locks. Even when this support exists, if multiple threads require simultaneous upgrades, only one lock upgrade will succeed.

Note that applications using readers/writer locks become responsible for providing appropriate serialization of their data structures since they cannot rely on the dispatching component itself to serialize upcalls. As with recursive locks, the serialization overhead of readers/writer locks may be higher compared to regular locks [63] when little or no contention occurs on the dispatching table.

Implementors of this pattern must analyze their dispatching component carefully to identify operations that require only a `read` lock versus those that require a `write` lock. For example, the CORBA Object Adapter supports activation of objects within upcalls. Thus, when a dispatch lookup is initiated, the Object Adapter cannot be certain whether the upcall will modify the dispatching table. Note that gratuitously acquiring a `write` lock is self-defeating since it may impede concurrent access to the table unnecessarily.

Finally, this solution does not resolve the predictability problem. In particular, unbounded priority inversion may occur when high-priority writer threads are suspended waiting for low-priority reader threads to complete dispatching upcalls.

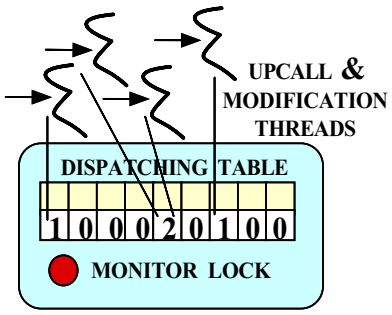


Figure 4.6: Dispatching with reference counted table entries

4.2.4 Reference Counting during Dispatch

Context: A multi-threaded system has stringent QoS requirements that demand predictable and efficient behavior from the dispatching component.

Problem: To be predictable, the system must eliminate all unbounded priority inversions. In addition, system efficiency should be maximized by reducing bounded priority inversions.

Forces: During an upcall, an application can invoke operations that modify the dispatching table. In addition, the dispatching component must be efficient and scalable, maximizing concurrency whenever possible.

Solution: Reference count the entries of the dispatching table during dispatch by using a single lock to serialize (1) changes to the reference count and (2) modifications to the table. As shown in Figure 4.6, the lock is acquired during the upcall, the appropriate entry is located, its reference count increased, and the lock is released before performing the application upcall. Once the upcall completes, the lock is re-acquired, the reference count on the entry is decremented, and the lock is released.

As long as the reference count on the entry remains greater than zero, the entry is not removed and the corresponding object is not destroyed. Concurrency hazards are avoided, therefore, because the reference count is always greater than zero while a thread is processing an upcall for that entry. If an object is removed from the dispatching table, its entry is only *logically* removed if outstanding upcalls are pending on it. The thread that brings the reference count to zero is responsible for deleting this entry from the table.

In programming languages, such as C and C++, that lack built-in garbage collection, the dispatching table must collaborate with the application to control the objects'

life-cycle. In this case, objects are usually reference counted[1]. For example, the reference count is usually incremented when the object is registered with the dispatching table and decremented when the object is removed from the dispatching table.

Consequences: This pattern supports multiple simultaneous upcalls since the lock is not held during the upcall. For the same reason, this model also supports recursive calls . An important benefit of this pattern is that the level of priority inversion does not depend on the duration of the upcall. In fact, priority inversion can be calculated as a function of the time needed to search the dispatching table. In Chapter 3, we have shown that bounded and very low search times can be achieved using techniques like active demultiplexing and perfect hashing. Implementations that use these techniques in conjunction with the serialization pattern described here can achieve predictable dispatching with bounded priority inversion.

A disadvantage of this pattern, however, is that it acquires and releases the lock *twice* per upcall. In practice, this usually does not exceed the cost of a single recursive monitor lock or a single readers/writer monitor lock [63]. This solution does, however, warrant extra care in the following special circumstances:

- *Accessing "logically deleted" objects* – A new request arrives for an object that has been logically but not physically removed from the dispatching table. Additional state can be used to record that this object has been removed and should therefore receive no new requests.
- *Reactivating "logically deleted" objects* – An implementation must handle the case where an object has been logically deleted and a client application requests a new object to be inserted for the same identifier as the logically deleted object. Typically, the new insertion must block until upcalls on the old object complete and the old object is physically removed from the dispatching table.

4.3 Concluding Remarks

This chapter described patterns for developing and selecting appropriate solutions to common problems encountered when developing efficient, scalable, predictable, and flexible dispatching components. Table 4.1 summarizes the different patterns for dispatching to a single object and compares their relative strengths and weaknesses. Real-time ORBs can use the reference counting dispatching pattern (described in Section 4.2.4) to bound priority inversion and hence provide timeliness guarantees to real-time applications.

[1]Note that this reference count is different from the per-entry reference count described above.

Table 4.1: Summary of dispatching to single object

Pattern	*Times lock acquired*	*Nested Upcalls*	*Priority Inversion*	*Appropriate when*
Serialized Dispatching	1	No	Unbounded	Little/no contention & short-lived upcalls
Recursive Lock	1	Yes	Unbounded	Same as above & nested upcalls
Readers/ Writer Lock	1	Limited	Unbounded	Concurrent upcalls
Reference Counting	2	Yes	Bounded	Predictable behavior

Chapter 5

Patterns for Implementing Thread Pools in Real-Time CORBA

Abstract

This chapter presents two contributions to improving the quality of implementation of RT-CORBA thread pools. First, the key patterns underlying common strategies for implementing RT-CORBA thread pools are described. Second, we evaluate each thread pool strategy in terms of its consequences for (1) feature support, such as request buffering and thread borrowing, (2) scalability in terms of endpoints and event demultiplexers required, (3) efficiency in terms of data movement, context switches, memory allocations, and synchronizations required, (4) optimizations in terms of stack and thread specific storage allocations, and (5) bounded and unbounded priority inversions incurred in each implementation. This chapter also provides results that illustrate empirically how different thread pool implementation strategies perform in different ORB configurations.

5.1 Introduction to Implementing Thread Pools

We present two general strategies for implementing RT-CORBA thread pools. The first strategy uses the *Half-Sync/Half-Async* pattern [70], where I/O thread(s) buffer the incoming requests in a queue and a different set of worker threads then process the requests. The second strategy uses the *Leader/Followers* pattern [69] to demultiplex I/O events into threads in a pool without requiring additional I/O threads. Each strategy is preferable for certain application domains, *e.g.*:

- Internet servers may use the Half-Sync/Half-Async pattern to improve scalability, at the expense of increased average- and worst-case latency.
- Telecom servers may tolerate some degree of priority inversion when using the Half-Sync/Half-Async pattern to support buffering and borrowing across different priority bands.
- Embedded avionics control systems may trade resource duplication for avoiding priority inversions by using the Leader/Followers pattern.

Although RT-CORBA defines a standard set of interfaces and policy types, it intentionally underspecifies many *quality of implementation* details, such as the ORB's memory management and connection management strategies. Though this approach maximizes the freedom of RT-CORBA ORB developers, it requires that application developers and end-users understand how an ORB is designed and how its design affects the schedulability, scalability, and predictability of their application.

The thread pool architecture is an essential dimension of an RT-CORBA ORB that also falls into the category of quality of implementation detail. In this section, we use *patterns* to describe these two strategies in detail, outlining their structure, dynamics, implementation, and consequences[1]. We focus on patterns in this chapter to generalize the applicability of our work.

5.2 Half-Sync/Half-Async

The Half-Sync/Half-Async architectural pattern decouples asynchronous and synchronous service processing in concurrent systems, to simplify programming without unduly reducing performance. The pattern introduces two intercommunicating layers, one for asynchronous and one for synchronous service processing.

5.2.1 Problem

Concurrent systems often contain a mixture of asynchronous and synchronous processing. For example, asynchronous events that an RT-CORBA server must react to include network messages and software signals. However, there are several components of an RT-CORBA

[1]For completeness, this chapter contains abbreviated descriptions of the Half-Sync/Half-Async and Leader/Followers patterns, focusing on the implementation of thread pools in RT-CORBA. Thorough discussion of these patterns appear in [70] and [69].

server that require synchronous processing, such as execution of application-specific servant code.

Synchronous programming is usually less complex compared to asynchronous programming because the thread of control can block awaiting the completion of operations. Blocking operations allow programs to maintain state information and execution history in their run-time activation record stack. If all tasks are processed synchronously within separate threads of control, however, thread management overhead can be excessive. Each thread contains resources that must be created, stored, retrieved, synchronized, and destroyed by a thread manager.

Conversely, asynchronous programming is generally more efficient. In particular, interrupt-driven asynchronous systems may incur less context switching overhead [71] than synchronous threaded systems. In addition, asynchronous services can be mapped directly onto OS asynchrony mechanisms, such as WinNT I/O completion ports [61, 70]. However, asynchronous programs are harder to develop, debug, and maintain. Asynchronous programs must manage additional data structures that contain state information and execution history, which must be saved and restored when a thread of control is preempted by an interrupt handler.

Two forces must therefore be resolved when specifying an RT-CORBA threading architecture that executes services both synchronously and asynchronously:

- The architecture should be designed so parts of the ORB that can benefit from the simplicity of synchronous processing need not address the complexities of asynchrony. Similarly, ORB services that must maximize performance should not need to suffer the inefficiencies of synchronous processing.

- The architecture should enable the synchronous and asynchronous processing services to communicate without complicating their programming model or unduly degrading their performance.

Although the need for both programming simplicity and high performance may seem contradictory, it is essential that both these forces be resolved in scalable RT-CORBA implementations.

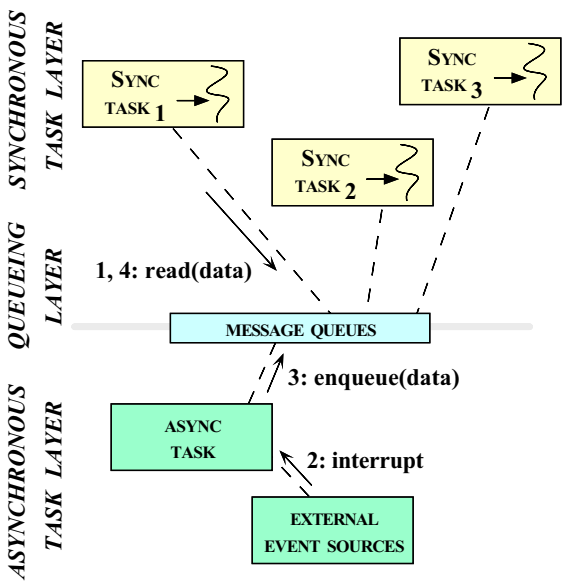


Figure 5.1: Structure of participants in the Half-Sync/Half-Async pattern

5.2.2 Solution

An RT-CORBA ORB endsystem can be decomposed into two layers [5], synchronous and asynchronous; a queueing layer is introduced to mediate the communication between services in the asynchronous and synchronous layers.

5.2.3 Structure and Collaboration

The structure of the Half-Sync/Half-Async pattern is illustrated in Figure 5.1 and the collaborations are illustrated in Figure 5.2. This design follows the Layers pattern [5] and includes the following participants:

Synchronous service layer: This layer performs high-level processing services. Services in the synchronous layer run in separate threads that can block while performing operations. In an RT-CORBA server, this layer:

1. Dequeues a request from the queueing layer;
2. Finds the target servant for the request;
3. Demarshals the request;

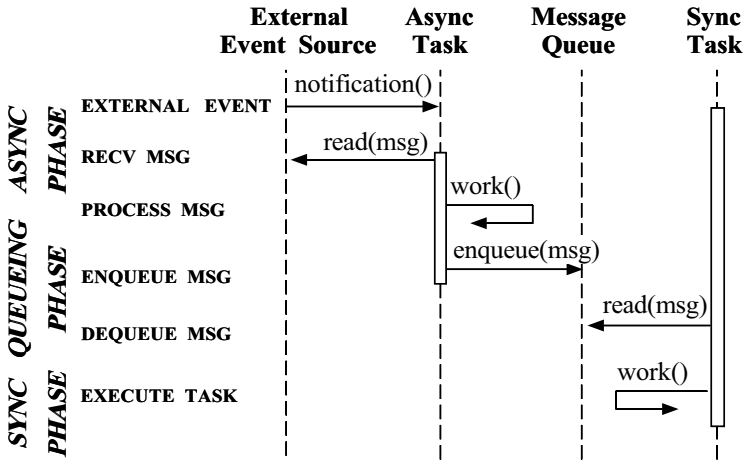


Figure 5.2: Collaboration between layers in the Half-Sync/Half-Async pattern

4. Performs upcalls into application-specific code by calling into the target servant registered in the POA by the application;

5. Marshals the reply (if any) to the client; and

6. Enqueues the reply (if any) in the queueing layer.

Asynchronous service layer: This layer performs lower-level processing services, which typically emanate from one or more external event sources. Services in the asynchronous layer cannot block while performing operations without unduly degrading the performance of other services. In an RT-CORBA server, this layer:

1. Reads the incoming request from the network;

2. Finds the target thread pool that will handle this request; and

3. Adds the request to the thread pool's queue that has the appropriate priority.

Queueing layer: This layer provides the mechanism for communicating between services in the synchronous and asynchronous layers. For example, messages containing data and control information are produced by asynchronous services, then buffered at the queueing layer for subsequent retrieval by synchronous services, and vice versa. The queueing layer is responsible for notifying services in one layer when messages are passed to them

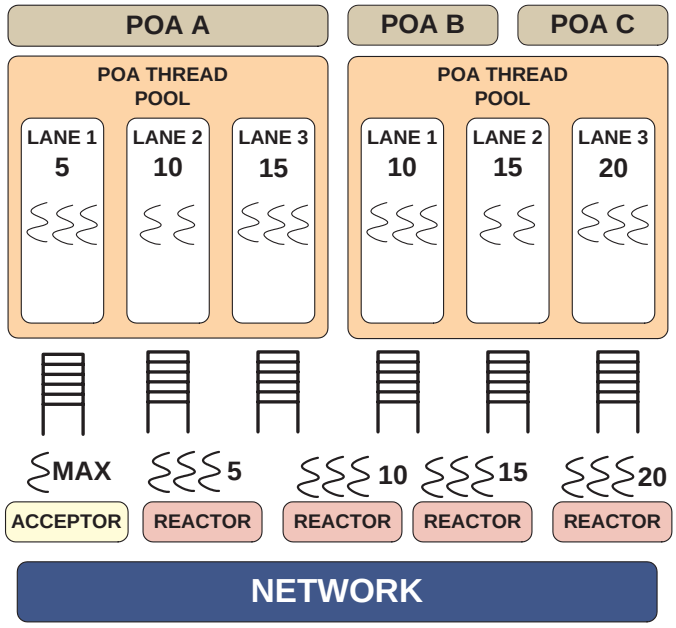


Figure 5.3: Implementing a RT-CORBA thread pool using the Half-Sync/Half-Async pattern

from the other layer. The queueing layer therefore enables the asynchronous and synchronous layers to interact in a "producer/consumer" manner, similar to the structure defined by the Pipes and Filters pattern [5]. For an RT-CORBA server, this layer queues incoming requests from and outgoing replies to clients.

External event sources: These sources generate events that are received and processed by the asynchronous service layer. For an RT-CORBA server, common sources of external events include sensors, network interfaces, disk controllers, and end-user terminals.

5.2.4 Implementation Synopsis

Figure 5.3 illustrates the architecture of a RT-CORBA ORB where thread pools are designed using the Half-Sync/Half-Async pattern. The asynchronous layer performs I/O processing, demultiplexing of incoming requests, and multiplexing of outgoing replies. It consists of the following components:

- *Acceptor* – An Acceptor [70] services connection requests from clients. The client establishes multiple connections to the server, one for every range of priorities that

will be used by the client when making requests. After a connection has been established, it is moved to the Reactor with the corresponding priority during the first request.

- *Reactors* – Each priority supported by the server has a corresponding Reactor [70], which is used to demultiplex and dispatch incoming client requests.
- *Threads* – The Acceptor is serviced by a thread running at an ORB-defined priority. Each Reactor is serviced by thread(s) at the appropriate priority.

To avoid priority inversion, the queueing layer consists of multiple queues, one for every thread pool lane. I/O threads read the incoming request, determine their target thread pool, and deposit the request into the right queue for processing. The synchronous layer consists of the threads in thread pool lanes. These threads block on a condition variable, waiting for requests to show up in their queue. After dequeuing the request, the target servant is found in the target POA, the request is demarshaled and application-level servant code is executed.

5.2.5 Consequences

The Half-Sync/Half-Async implementation of RT-CORBA thread pools has the following *benefits*:

- *Simplified programming:* The programming of the synchronous phase is simplified without degrading the performance of the asynchronous layer. Distributed systems based on RT-CORBA often have a larger quantity and variety of high-level processing services than lower-level services. Decoupling higher-level synchronous services from lower-level asynchronous processing services can therefore simplify ORB development because complex concurrency control, interrupt handling, and timing services can be localized within the asynchronous service layer. The asynchronous layer can also handle low-level details that are difficult to program robustly and can manage the interaction with hardware-specific components, such as DMA (Direct Memory Access), memory management, and network I/O.
- *Support for request buffering and thread borrowing:* Since a request remains in the queueing layer until a thread is available to service it, the queueing layer can be used to buffer requests by bursty clients. Thread borrowing can also be implemented relatively easily by buffering the request in a queue that has threads available to process the request.

- *Sharing of I/O resources:* ORB resources, such as reactors and acceptors, are per-priority resources in the I/O layer. Therefore, if a server is configured with many thread pools that have similar lane priorities, I/O layer resources are shared by these lanes.
- *Easier piece-by-piece integration into the ORB:* Ease of implementation and integration are important practical considerations in any project. Due to its layered structure, this approach is easier to design, implement, integrate, and test incrementally.

The Half-Sync/Half-Async implementation of RT-CORBA thread pools also has the following *liabilities*:

- *Data exchange overhead:* When exchanging data between the synchronous and asynchronous layers, the queueing layer can incur a significant performance overhead due to context switching, synchronization, cache coherency management, and data-copying overhead [71].
- *No memory management optimizations:* Since a request is handed off from an I/O thread in the asynchronous layer to a thread pool thread in the synchronous layer, stack memory and thread-specific storage (TSS) [70] cannot be used to optimize memory management for client requests. Instead, a shared memory pool must be used to allocate storage for the requests. Unfortunately, synchronization for this shared memory pool can lead to extra overhead. Moreover, if the memory pool is shared between threads of different priorities, it can lead to priority inversion.

Table 5.1 summarizes the evaluation for Half-Sync/Half-Async implementation of RT-CORBA thread pools.

5.3 Leader/Followers

The Leader/Followers architectural pattern provides an efficient concurrency strategy where multiple threads take turns sharing a set of event sources in order to detect, demultiplex, dispatch, and process service requests that occur on the event sources.

Table 5.1: Evaluation of Half-Sync/Half-Async thread pools

Criteria	*Evaluation*
Feature Support	***Good:*** supports request buffering and thread borrowing
Scalability	***Good:*** I/O layer resources shared
Efficiency	***Poor:*** high overhead for data movement, context switches, memory allocations, and synchronizations
Optimizations	***Poor:*** stack and TSS memory not supported
Priority Inversion	***Poor:*** some unbounded, many bounded

5.3.1 Problem

Mission-critical RT-CORBA servers often process a high volume of requests that arrive simultaneously. To process these requests efficiently, the following three forces must be resolved:

- Associating a thread with each connected client may be infeasible due to the scalability limitations of applications or the underlying OS and hardware platforms.
- Allocating memory dynamically for each request passed between multiple threads incurs significant overhead on conventional multiprocessor operating systems.
- Multiple threads that demultiplex events from a shared set of event sources must coordinate to prevent race conditions. Race conditions can occur if multiple threads try to access or modify certain types of event sources simultaneously.

5.3.2 Solution

A pool of threads is structured to share incoming client requests by taking turns demultiplexing the requests and synchronously dispatching the associated servant code that processes the request.

More specifically, this thread pool mechanism allows multiple threads to coordinate themselves and protect critical sections while detecting, demultiplexing, dispatching, and processing requests. In this mechanism, one thread at a time—the leader—waits for a request to arrive from the set of connected clients. Meanwhile, other threads—the

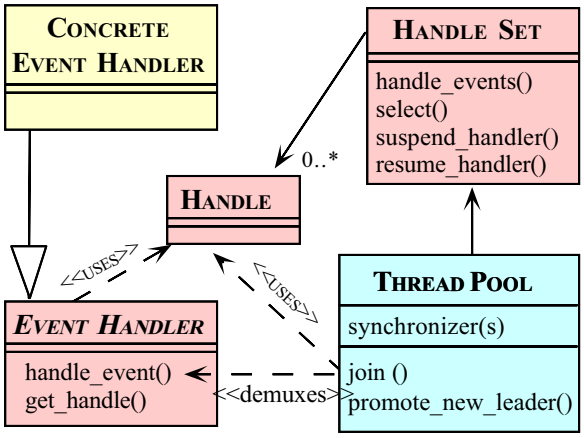


Figure 5.4: Structure of participants in the Leader/Followers pattern

followers—can queue up waiting their turn to become the leader. After the current leader thread detects a new client request, it first promotes a follower thread to become the new leader. It then plays the role of a processing thread, demultiplexing and dispatching the request to application-specific code. Multiple processing threads can handle requests concurrently while the current leader thread waits for new requests. After handling its request, a processing thread reverts to a follower role and waits to become the leader thread again.

5.3.3 Structure and Collaboration

The key participants in the Leader/Followers pattern are shown in Figure 5.4 and are described below:

Handles and handle sets: Handles identify operating systems objects, such as network connections, that indicate when new requests arrive from clients. A handle set is a collection of handles that can be used to wait for one or more clients to send requests.

Event Handlers: The ORB event handler dispatches the incoming request to the target servant. This process includes:

1. Reading the request from the network;
2. Finding the target servant for the request;
3. Demarshaling the request;
4. Performing the upcall into application-specific code by calling into the target servant registered in the POA by the application;

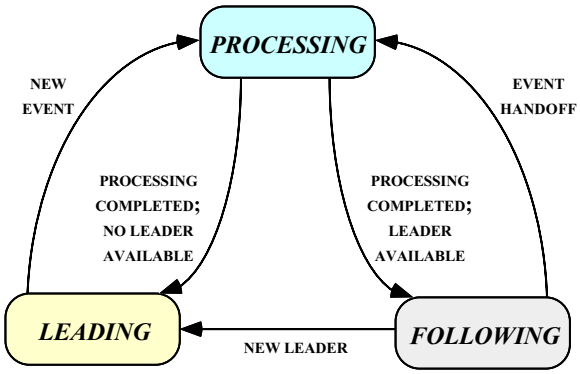


Figure 5.5: A thread's state transitions in the Leader/Followers pattern

5. Marshaling the reply (if any) to the client; and

6. Sending the reply (if any) back to the client.

Thread Pool: At the heart of the Leader/Followers pattern is a thread pool, which is a group of threads that share a synchronizer, such as a condition variable, and implement a protocol for coordinating their transition between various roles. A thread's state transitions are shown in Figure 5.5 and collaborations in the Leader/Followers pattern are shown in Figure 5.6.

5.3.4 Implementation Synopsis

In this design, each RT-CORBA thread pool lane has an integrated I/O layer, *i.e.*, there is one acceptor and one reactor for every lane. Clients connect to the acceptor endpoint with the desired priority and as shown in Figure 5.7, all client request processing (as described in Section 5.3.3) is performed by the thread of desired priority from very beginning. Thus, there are no context switches and priority inversions are minimized.

In addition, the ORB does not create any internal I/O threads. This allows application programmers full control over the number and properties of all the threads with the RT-CORBA thread pool APIs. In contrast, the Half-Sync/Half-Async implementation has I/O layer threads, so either a proprietary API must be added or the application programmer will not have full control over all the thread resources.

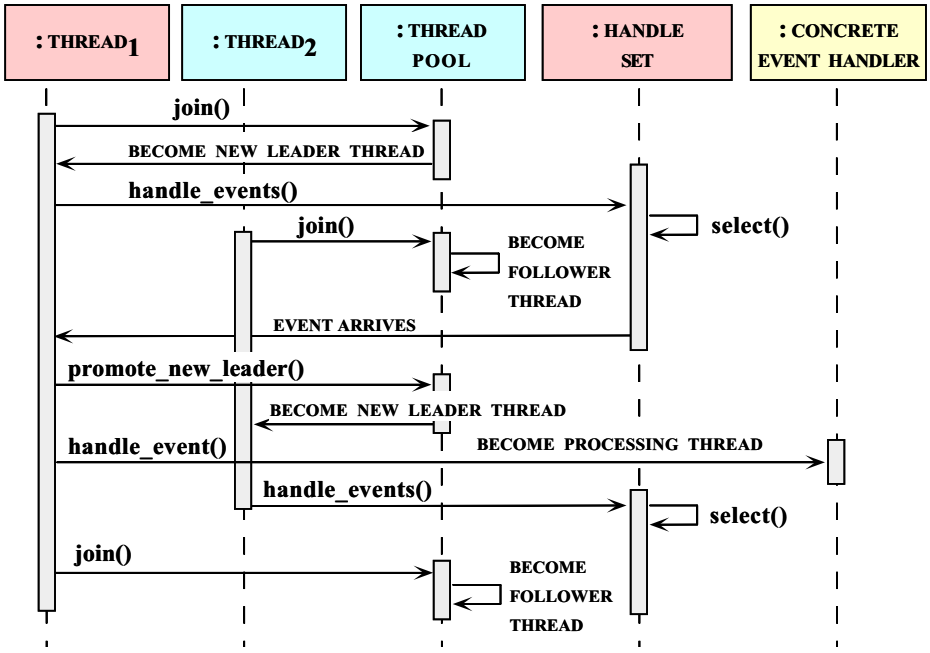


Figure 5.6: Collaboration in the Leader/Followers pattern

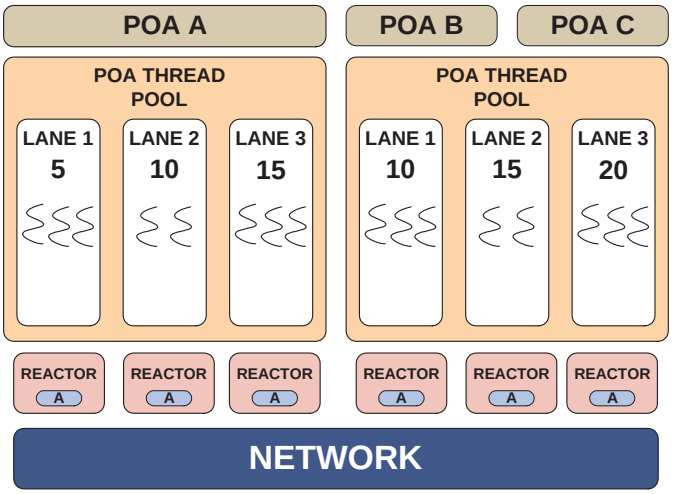


Figure 5.7: Implementing a RT-CORBA thread pool using the Leader/Followers pattern

5.3.5 Consequences

The Leader/Followers pattern provides several *benefits* and *improvements* compared with the Half-Sync/Half-Async thread pool strategy described in Section 5.2:

- *Memory optimizations:* It enhances cache affinity and eliminates the need for dynamic memory allocation and data buffer sharing between threads. For example, a processing thread can read the request into buffer space allocated on its run-time stack or by using thread-specific storage (TSS) [70].
- *Reduced synchronization:* It minimizes locking overhead by not exchanging data between threads, thereby reducing thread synchronization.
- *Reduced priority inversion:* It can reduce priority inversion because no extra queueing is introduced in the server. When combined with real-time I/O subsystems [34], the Leader/Followers thread pool implementation can reduce sources of non-determinism in server request processing significantly.
- *Improved dispatch latency:* It does not require a context switch to handle each request, reducing the request dispatching latency. Note that promoting a follower thread to fulfill the leader role does require a context switch. If two events arrive simultaneously, this increases the dispatching latency for the second event, but the performance is no worse than the Half-Sync/Half-Async thread pool implementation.

However, the Leader/Followers pattern has the following *liabilities*:

- *Implementation complexity:* The advanced variants of the Leader/Followers pattern are harder to implement than Half-Sync/Half-Async thread pools. A thorough discussion of these variants appears in [70].
- *Lack of flexibility:* The queueing layer in the Half-Sync/Half-Async thread pool implementation makes it easy to support features like request buffering and thread borrowing. In the Leader/Followers implementation, however, it is harder to implement these features because there is no explicit queue.

Table 5.2 summarizes the evaluation for Leader/Followers implementation of RT-CORBA thread pools.

Table 5.2: Evaluation of Leader/Followers thread pools

Criteria	*Evaluation*
Feature Support	***Poor:*** not easy to support request buffering or thread borrowing
Scalability	***Poor:*** I/O layer resources not shared
Efficiency	***Good:*** little or no overhead for data movement, memory allocations, or synchronizations
Optimizations	***Good:*** stack and TSS memory supported
Priority Inversion	***Good:*** little or no priority inversions

Table 5.3: Salient operations invoked by the Half-Sync/Half-Async and the Leader/Followers thread pool implementations

Operations	*Times called for HS/HA*	*Times called for LF*
malloc	2	0
free	2	0
enqueue	2	0
dequeue	2	0
signal	2	1
locks	8	2

5.4 Empirical Results

This section empirically compares the performance of the Half-Sync/Half-Async vs. the Leader/Followers thread pool implementation. Table 5.3 shows the salient operations invoked by the two thread pool implementations in the critical path of processing a request.

The Half-Sync/Half-Async thread pool implementation invokes `malloc` twice: (1) to create a buffer for the request, and (2) to create a buffer for the reply. Correspondingly, `free` is called twice to delete the dynamically allocated buffers. `enqueue` is called twice: (1) when the asynchronous layer queues the request for the synchronous layer, and (2) when the synchronous layer queues the reply for the asynchronous layer. Correspondingly,

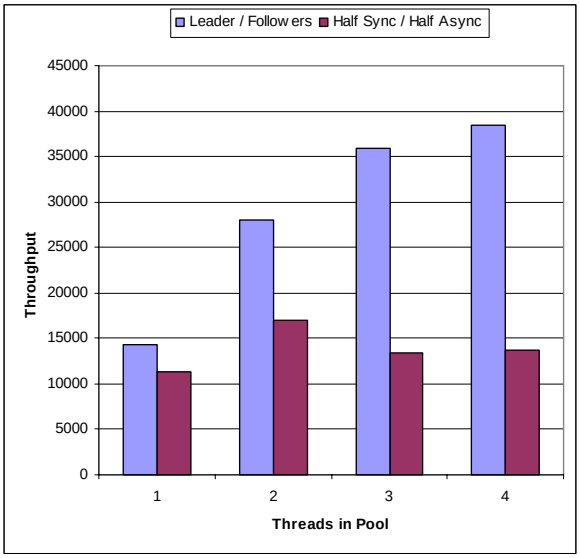


Figure 5.8: Performance of Half-Sync/Half-Async vs. the Leader/Followers thread pool implementations

`dequeue` is called twice to remove the request and reply from the queuing layer. `signal` is called twice, once after every `enqueue` operation to notify the receiving layer. Eight locks are held in the critical path, once for every `malloc` and `free` to synchronize the shared memory pool, and once of every `enqueue` and `dequeue` operation to synchronize the queuing layer.

Conversely, the Leader/Followers thread pool implementation creates the request and reply buffers utilizing space allocated on its run-time stack. Therefore, does not invoke `malloc` or `free`. In addition, since the request is completely processed by a single thread, it eliminates the queuing layer and the corresponding `enqueue` and `dequeue` operations. `signal` is called once to promote a follower to leader before processing the request. Two locks are held in the critical path: (1) to promote a new leader, and (2) to become a follower.

Figure 5.8 compares the relative performance of the two thread pool implementations as the number of threads in the pool increases. Not only does the Leader/Followers thread pool implementation outperform the Half-Sync/Half-Async implementation, it scales much better as the number of threads increases. This is because the Half-Sync/Half-Async thread pool implementation incurs higher overhead for memory allocation, locking, and data movement compared to the Leader/Followers implementation.

5.5 Related Work

In this section, we compare our work on TAO's RT-CORBA thread pools with related work on CORBA.

URI TDMI. Wolfe *et al.* developed a real-time CORBA system at the US Navy Research and Development Laboratories (NRaD) and the University of Rhode Island (URI) [79]. The system supports expression and enforcement of dynamic end-to-end timing constraints through timed distributed method invocations (`TDMI`s) [14]. A difference between TAO and the URI approach is that `TDMI`s are based on timing constraints, *e.g.*, deadlines relative to the current time, whereas TAO's threading strategies are based on the fixed-priority scheduling features defined in the RT-CORBA specification.

BBN QuO. The *Quality Objects* (QuO) distributed object middleware was developed at BBN Technologies [81]. QuO is based on CORBA and provides the following support for QoS-enabled applications:

- *Run-time performance tuning and configuration* through the specification of operating regions, behavior alternatives, and reconfiguration strategies. This allows the QuO run-time to trigger reconfiguration adaptively as system conditions change (represented by transitions between operating regions); and
- *Feedback* across software and distribution boundaries based on a control loop in which client applications and server objects request levels of service and are notified of changes in service.

The QuO model employs several *QoS definition languages* (QDLs) that describe the QoS characteristics of various objects, such as expected usage patterns, structural details of objects, and resource availability. QuO's QDLs are based on the separation of concerns advocated by Aspect-Oriented Programming (AoP) [30]. The QuO middleware adds significant value to adaptive real-time ORBs such as TAO. We are currently collaborating [37] with the BBN QuO team to integrate the TAO and QuO middleware as part of the DARPA Quorum project [8].

UCI TMO. The Time-triggered Message-triggered Objects (TMO) project [31] at the University of California, Irvine, supports the integrated design of distributed OO systems and real-time simulators of their operating environments. The TMO model provides structured timing semantics for distributed real-time object-oriented applications by extending conventional invocation semantics for object methods, *i.e.*, CORBA operations, to include

(1) invocation of time-triggered operations based on system times and (2) invocation and time bounded execution of conventional message-triggered operations.

TAO differs from TMO in that TAO provides a complete CORBA ORB, as well as CORBA ORB services and real-time extensions. Timer-based invocation capabilities are provided through TAO's Real-Time Event Service [23]. Where the TMO model creates new ORB services to provide its time-based invocation capabilities [32], TAO provides a subset of these capabilities by extending the standard CORBA Event Service. We believe TMO and TAO are complementary technologies because (1) TMO extends and generalizes TAO's existing time-based invocation capabilities and (2) TAO provides a configurable and dependable connection infrastructure needed by the TMO CNCM service. We are currently collaborating with the UCI TMO team to integrate the TAO and TMO middleware as part of the DARPA NEST project.

5.6 Concluding Remarks

Thread pools are an important RT-CORBA capability since they allow application developers and end-users to control and bound the amount of resources dedicated to concurrency and queueing. There are various strategies for implementing thread pools in the RT-CORBA. Since certain strategies are optimal for certain application domains, users of RT-CORBA middleware must understand the trade-offs between the different strategies.

This chapter described the Half-Sync/Half-Async and the Leader/Followers strategies for implementing RT-CORBA thread pools. We evaluated these strategies using several different factors and presented results that illustrate empirically how different thread pool implementation strategies perform in different ORB configurations. Our pattern-based descriptions are intended to help application developers and end-users understand the schedulability, scalability, and predictability consequences of a particular thread pool implementation in an RT-CORBA ORB.

Chapter 6

Real-time ORB Design

Abstract

This chapter takes a careful look at the end-to-end critical code path of a CORBA request and identifies sources of unbounded priority inversion within the ORB. It then shows how the ORB can be redesigned to use non-multiplexed resources to eliminate the sources of unbounded priority inversion. This chapter also examines how RT-CORBA adds several new challenges to collocation, endpoint selection, and memory management schemes and how these mechanisms can be redesigned to improve performance and predictability.

6.1 Tracing an Invocation

In order to identify sources of unbounded priority inversion within the ORB, this section traces the end-to-end critical code path of a CORBA request. The first part sets up a connection between the client and the server. The second part involves (a) the client sending a request to the server, (b) the server processing the request and sending a reply, and (c) the client processing the reply.

Client activities for creating a connection:

1. Query the connection cache for an existing connection to the server.
2. Use the Connector [70] to create a new connection if there are no previously cached connections to the server.
3. Add the newly established connection S to the connection cache.
4. Also add connection S to the Reactor [70] since S is bi-directional and the server may send requests to the client using S.

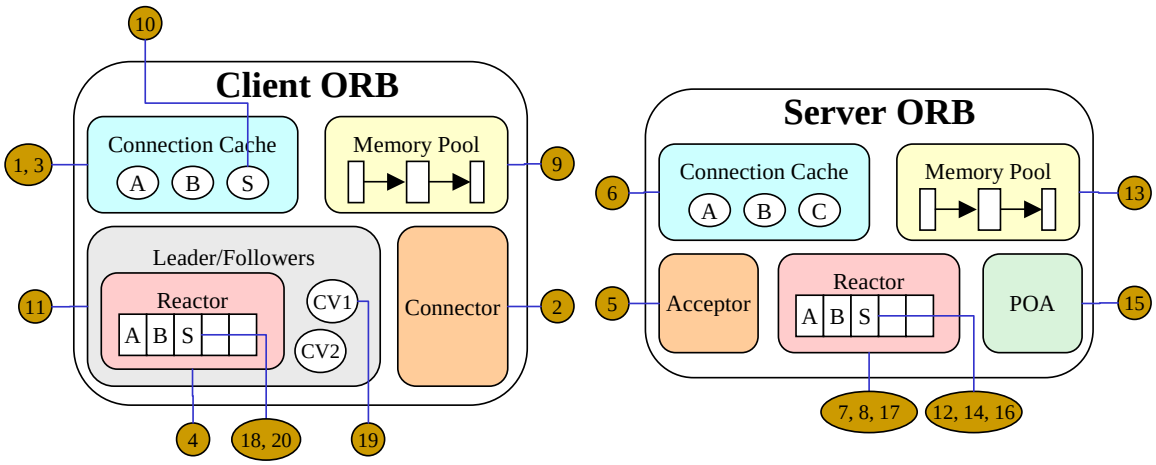


Figure 6.1: Tracing an invocation through the ORB

Server activities for accepting a connection:

5. Accept the new connection C from the client.
6. Add C to the connection cache since C is bi-directional and the server can use it to make requests to the client.
7. Also add connection C to the Reactor so that the server is notified when a request from the client arrives at the server.
8. Wait in the Reactor for new events.

Client activities for sending a request:

9. Allocate a buffer from the memory pool to marshal the invocation parameters.
10. Send the marshaled data to the server using connection S. Connection S is locked for the duration of the transfer.
11. Wait in the Leader/Follower [70] for a reply from the server. Assuming that a leader thread is already available[1], the client thread waits as a follower on a condition variable in the Leader/Follower.

[1]The leader thread may be a server thread waiting for incoming requests or another client thread waiting for its reply.

Server activities for processing a request:

12. Read the header of the request arriving on connection C to determine the size of the request.
13. Allocate a buffer from the memory pool for the request.
14. Read the request data into the buffer.
15. Demultiplex the request to find the target POA [56], servant, and skeleton. Dispatch an upcall to the servant after demarshaling the request parameters.
16. Send the reply (if any) to the client on connection C. Connection C is locked for the duration of the transfer.
17. Wait in the Reactor for new events.

Client activities for processing a reply:

18. Leader thread reads the reply from the server on connection S.
19. Leader thread hands off the reply to the follower thread after identifying that the reply belongs to the follower. This is done by signaling the condition variable used by the follower.
20. Follower thread demarshals the parameters and processes the reply.

6.2 Identifying Sources of Unbounded Priority Inversion

Predictable components and subsystems are essential in providing end-to-end QoS guarantees. This section identifies resources in the critical path (described in Section 6.1) that can cause unbounded priority inversion.

Connection Cache: In steps 10 and 16, the connection is locked for the duration of the data transfer. Mutual exclusion of the connection prevents multiple threads from writing to the connection simultaneously and hence avoids corruption of the request and reply data.

However, the time required to send the request data depends on availability of network resources and the size of the request. Unless the underlying network provides timeliness guarantees for data delivery, mutual exclusion of the connection can cause a higher priority thread to wait indefinitely, leading to unbounded priority inversion. If priority inheritance is not supported by the mutual exclusion mechanism used to lock the connection, priority inversion can be further exacerbated.

One approach to alleviate this problem would be to create a new connection to the peer ORB instead of waiting for existing connection to become available. However, creating a new connection can also take an indefinite amount of time.

Memory Pool: In steps 9 and 13, buffers are allocated for marshaling and demarshaling requests. Typically, the memory pool is locked while finding a buffer large enough to hold the processed data. Mutual exclusion of the memory pool prevents multiple threads from accessing the memory pool simultaneously and hence avoids corruption of the free and occupied buffer lists in the memory pool.

However, the time required to allocate a new buffer depends on pool fragmentation and the memory management algorithm. Unless the memory pool provides timeliness guarantees for buffer allocation and deletion, mutual exclusion of access to the memory pool can cause a higher priority thread to wait indefinitely, leading to unbounded priority inversion. As in the case of connections, if priority inheritance is not supported by the mutual exclusion mechanism that locks the memory pool, priority inversion can be further exacerbated.

One approach to alleviate this problem would be to use stack or thread-specific storage based memory pool, which do not need mutual exclusion since they are not shared among multiple threads. However, it may not be possible to use such memory pools if buffers need to be shared by multiple threads.

Leader/Followers: Assume that the leader thread is of low priority while the follower thread is of high priority. In steps 18 and 19, the leader thread handles the reply for the follower thread. During this time, the leader thread may be preempted by some other thread of medium priority before the reply is handed off to the follower.

This problem can be avoided if the leader thread can inherit the priority of the follower thread through the condition variable. Unfortunately, most condition variable implementations don’t support priority inheritance.

Reactor: There is no way to distinguish a high priority client request from one of lower priority at the Reactor level. This can also lead to unbounded priority inversion if the lower priority request is serviced before one of higher priority.

POA: In step 15, the POA dispatches the upcall after locating the target POA, servant, and skeleton. The time required to demultiplex the request may depend on the organization of the POA hierarchy and number of POAs, servants, or operations configured in the

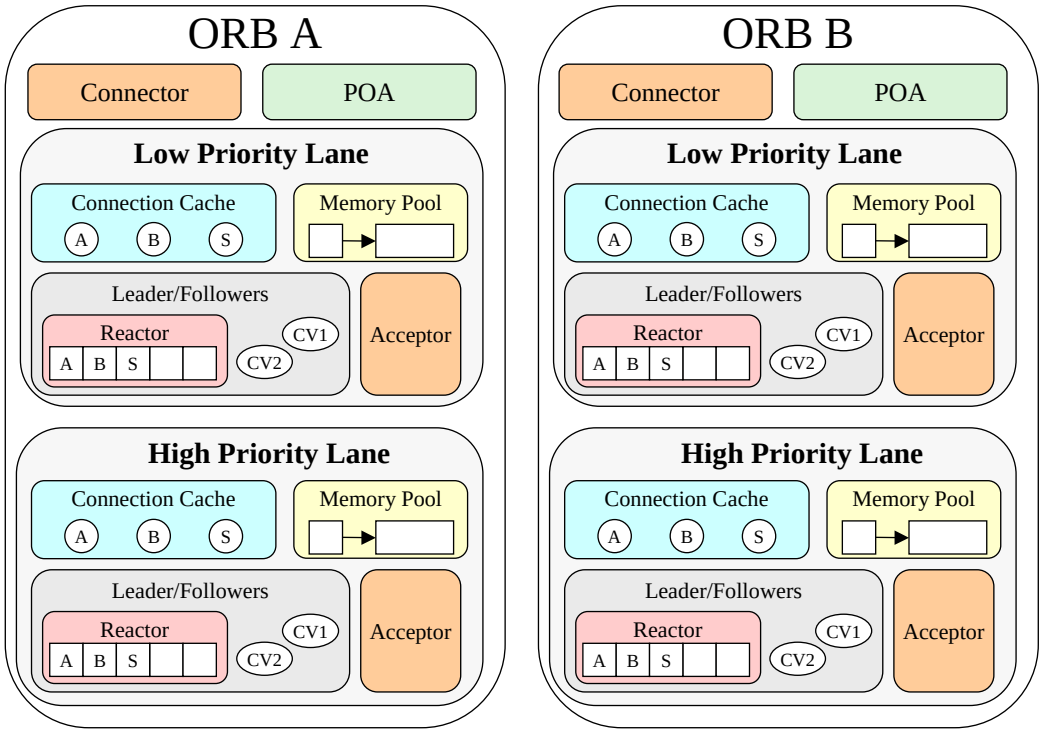


Figure 6.2: Real-time CORBA architecture

server. The time required to dispatch the request may depend on contention on POA dispatching table. Demultiplexing and dispatching in conventional CORBA implementations is typically inefficient and unpredictable [19, 20].

6.3 Eliminating Sources of Unbounded Priority Inversion

Figure 6.2 shows the redesigned ORB using non-multiplexed resources where each thread lane has it own set of resources, including its own (a) connection cache, (b) memory pool, (c) leader/followers manager that includes a reactor, and (d) acceptor. Components and factories that are not in the critical path or do not contribute to priority inversion, such as the connector, are shared by all lanes in the ORB. Conversely, the POA is a shared component that can be a potential source of non-determinism. Chapter 3 on demultiplexing describes methodologies that demultiplex predictably while reducing average- and worst-case overhead, *regardless* of organization of the POA hierarchy or number of POAs, servants, or operations. Also, Chapter 4 on efficient, scalable, and, predictable dispatching components

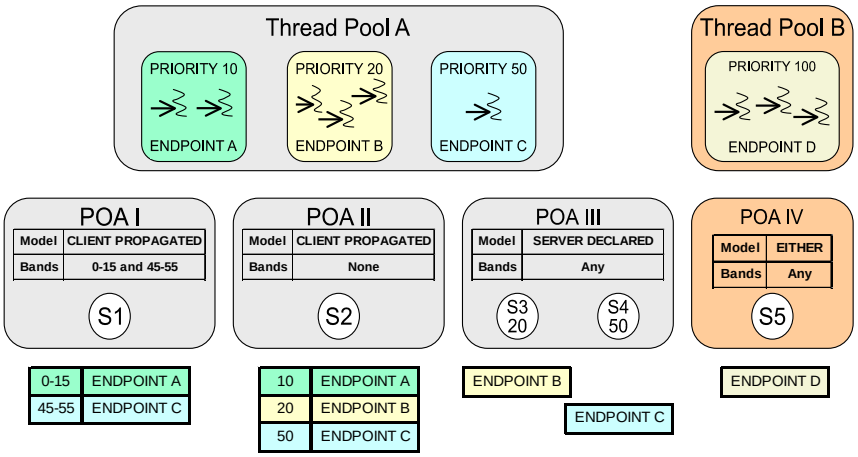


Figure 6.3: IOR creation and endpoint selection

shows how it is possible to bound priority inversion and hence provide timeliness guarantees to real-time applications.

Lane-specific resources are only shared by the threads in the lane. Since all the threads in a lane have the same priority, priority inversion is avoided. However, this technique is resource intensive. For example, assume that a thread in a high priority lane has established a connection to a particular server. If a thread in a low priority lane wants to communicate with the same server, it is not allowed to use the connection cached in the high priority lane.

Distributed systems typically deal with a handful of distinct priorities and therefore the above scheme is generally not a problem in practice. Even when hundreds of priorities are desired, priorities can be banded into ranges to ease resource requirements and schedulability analysis of the system.

6.4 Endpoint Selection

To propagate and preserve priorities, a client thread must communicate with a server thread of appropriate priority. With the multiplicity of acceptors on the server (as shown in Figure 6.2), care must be taken to ensure that the client thread selects an appropriate endpoint. This section describes the endpoints a server should publicize and how a client should select an appropriate endpoint. Figure 6.3 illustrates the range of possible endpoint publication and selection scenarios as described below:

Thread Pool Without Lanes: Servant 5 (S5) is registered in POA IV. POA IV is serviced by thread pool B, a thread pool without lanes. Thread pool B has one endpoint D[2]. When creating an IOR for S5, the server publicizes endpoint D.

Thread Pool With Lanes and SERVER_DECLARED Priority Model: Servants 3 and 4 (S3 and S4) are registered in POA III with priorities 20 and 50 respectively. POA III has the `SERVER_DECLARED` priority model and is serviced by thread pool A. Thread pool A has three lanes and each lane has its own endpoint. When creating an IOR for S3, the server only publicizes endpoint B since requests for this servant will only be processed by threads in the lane of priority 20. Similarly, when creating an IOR for S4, the server only publicizes endpoint C since requests for this servant will only be processed by threads in the lane of priority 50.

Thread Pool With Lanes and CLIENT_PROPAGATED Priority Model, but no Priority Bands: Servant 2 (S2) is registered in POA II. POA II has the `CLIENT_PROPAGATED` priority model and is serviced by thread pool A, but does not have priority bands. When creating an IOR for S2, the server has to publicize all three endpoints since it does not know *a priori* the priority of requests on S2. Depending on the priority of the client thread making an invocation on S2, one of the three endpoints will be selected. Note that the client is restricted to invocation priorities 10, 20, and 50 since the priorities of threads in lanes cannot change. Any other client invocation priority will cause an exception.

Thread Pool With Lanes, CLIENT_PROPAGATED Priority Model, and Priority Bands: Servant 1 (S1) is registered in POA I. POA I has the `CLIENT_PROPAGATED` priority model with two priority bands of 0–15 and 45–55, and is serviced by thread pool A. When creating an IOR for S1, the server publicizes endpoints A and C since the lane of priority 10 satisfies the 0–15 band and the lane of priority 50 satisfies the 45–55 band. Endpoint B need not be publicized in this case. Depending on the priority of the client thread making invocation on S1, one of the two endpoints will be selected. Note that the client is restricted to invocation priority ranges of 0–15 and 45–55.

[2] A thread pool without lanes can thought of as a thread pool with one lane, with the addition flexibility of allowing the threads to vary their priorities when processing requests.

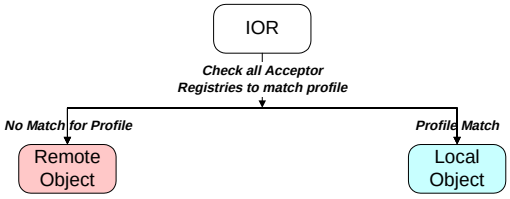


Figure 6.4: Non-RT CORBA collocation decision tree

6.5 Collocation Challenges in RT-CORBA

CORBA is primarily used to communicate between distributed applications, where clients use *remote* stubs to invoke operations on distributed servants. However, there are configurations where clients and servants are *collocated* in the same address space [72]. In such cases, there is no need to incur the overhead of transmitting requests/replies through a "loopback" transport device. In many collocated configurations, the overhead of data marshaling can also be avoided. In these configurations, clients use *collocated* stubs to invoke operations on servants.

To ensure location transparency, the client is unaware of which stub it is using – it is the ORB's responsibility to generate the correct stub for the client. As shown in Figure 6.4, if the ORB determines that the servant is collocated with the client, it generates a collocated stub; otherwise, it generates a remote stub.

RT-CORBA adds several new challenges to collocation. In RT-CORBA, if a servant is registered with a POA that is associated with a thread pool, then only threads belonging to that pool can execute upcalls on the servant. Therefore, collocated stubs, that use the client's thread of control to execute upcalls on servants, will not suffice for RT-CORBA. A new *cross pool/lane* stub is introduced that is somewhat similar to a collocated stub since it also avoids the transport and marshaling overhead incurred by a remote stub. However, a cross pool/lane stub does not use the client's thread to execute upcalls on the servant. The upcall is executed by a thread in the thread pool associated with the POA where the servant is registered.

RT-CORBA collocation decisions are further complicated by the use of lanes in the thread pool and by the priority model policy in effect. The following use cases illustrate various RT-CORBA collocation scenarios. The use cases are based on the server process illustrated in Figure 6.5, with several POA, thread pool, and priority model policy combinations.

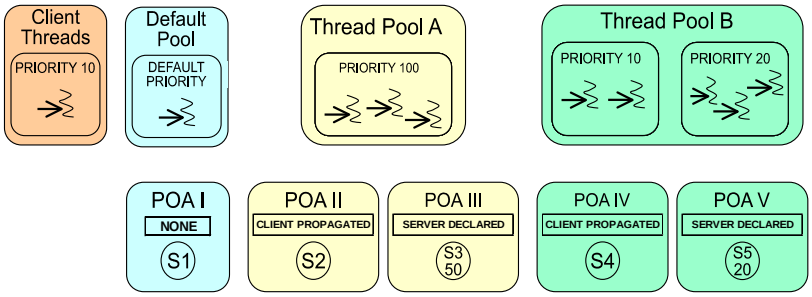


Figure 6.5: RT-CORBA collocation scenarios

Invocation on S1: Servant 1 (S1) is registered with POA I that does not have a priority model policy and is not associated with a thread pool, *i.e.*, it is associated with the default thread pool. In this case, there is no restriction on which thread can execute upcalls on S1. Therefore, all invocations on S1 are collocated, irrespective of which thread in the process makes the invocation.

Invocation on S2: Servant 2 (S2) is registered with POA II that has the `CLIENT_PROPAGATED` priority model policy and is associated with thread pool A. An invocation on S2 made by the thread in the default thread pool running at default priority is not collocated. This invocation is handled by a cross pool/lane stub and the upcall on S2 is executed by one of the threads in thread pool A. The priority of the upcall thread is changed to default priority for the duration of the upcall to match the priority of the client thread.

Invocation on S3: Servant 3 (S3) is registered with POA III that has the `SERVER_DECLARED` priority model policy and is associated with thread pool A. An invocation on S3 made by a thread in thread pool A running at priority 100 is collocated. The priority of the thread is changed to priority 50 for the duration of the upcall to match the server declared priority of S3.

Invocation on S4: Servant 4 (S4) is registered with POA IV that has the `CLIENT_PROPAGATED` priority model policy and is associated with thread pool B. An invocation on S4 made by a thread in thread pool B running at priority 20 is collocated. No modification needs to be made to the priority of the thread during the upcall.

Invocation on S5: Servant 5 (S5) is registered with POA V that has the `SERVER_DECLARED` priority model policy and is associated with thread pool B. An invocation on S5 made by a thread in thread pool B running at priority 10 is not collocated because it does not match the server declared priority of S5. This invocation is handled by a cross

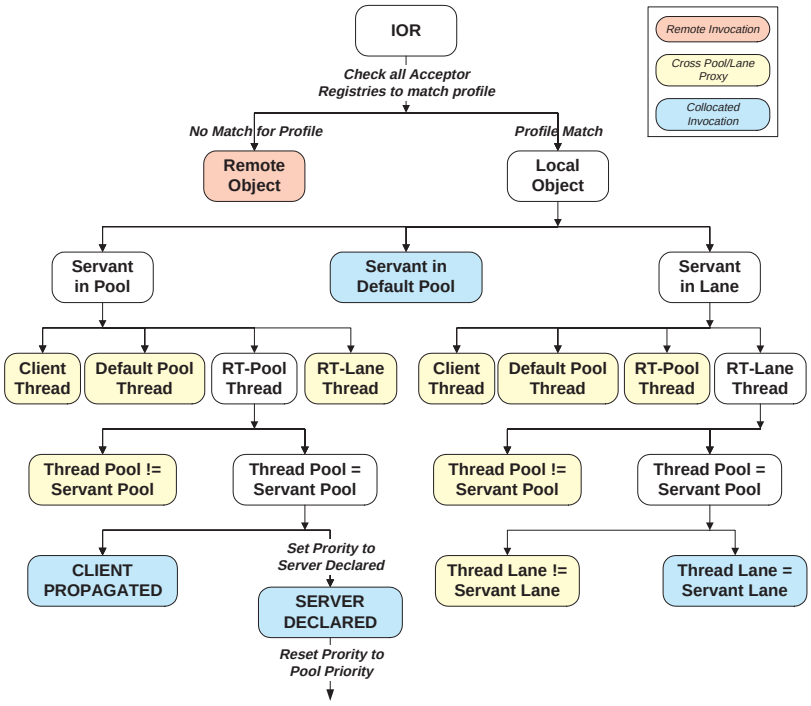


Figure 6.6: RT-CORBA collocation decision tree

pool/lane stub and the upcall on S5 is executed by one of the threads in thread pool B of priority 20.

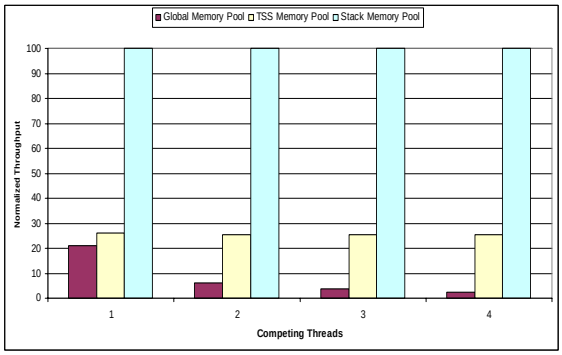
Figure 6.6 details the decision tree that is evaluated in RT-CORBA to determine if an invocation is collocated.

6.6 Memory Management Mechanisms to Improve Performance and Predictability

The memory management mechanisms used by an application can have significant affects on its performance and predictability. This section compares the behavior, properties, and performance of three memory management mechanisms: global memory pool, thread-specific storage (TSS) memory pool, and stack memory pool. Figure 6.7 (a) summarizes the behavior and properties of the three pools.

Behavior & Properties	*Global Memory Pool*	*TSS Memory Pool*	*Stack Memory Pool*
Malloc	1. Acquire lock 2. Manipulate pool 3. Release lock	1. TSS access 2. Manipulate pool	1. Stack allocation
Free	1. Acquire lock 2. Manipulate pool 3. Release lock	1. TSS access 2. Manipulate pool	Nothing
Share Memory	Yes	No	No
Contention	Yes	No	No
Size	Variable	Variable	Fixed

(a)



(b)

Figure 6.7: Comparing memory management schemes: **(a)** Salient operations; **(b)** Performance measurements

Global: A global memory pool is shared by all the threads in a process, and therefore requires synchronization in `malloc` and `free`. This pool can grow in size within the data segment size limits of the process.

Thread-specific storage: A TSS memory pool is specific to each thread but access to it requires a TSS lookup. Since it is not a shared resource, there is no synchronization required in `malloc` and `free`. This pool can also grow in size.

Stack: A stack memory pool is based on space allocated on the run-time stack of a thread. It is automatically reclaimed and cleaned up when the stack unwinds. Therefore, `free` for this pool is a no-op. This pool also does not require synchronization since it is not shared. One major drawback of this pool is that its size is determined at compile-time and cannot change during run-time. Therefore, it cannot be used for allocating buffers larger than the predetermined size of the pool.

Figure 6.7 (b) compares the relative performance of the three memory pools as the number of threads calling `malloc` and `free` increases. When the threads are using a global memory pool, the increased competition significantly affects the throughput achieved by each thread. However, when using TSS and stack memory pools, each thread has its own pool, and therefore increasing the number of threads does not affect competition for the pool nor does it affect each thread's performance.

Applications can get the best performance and predictability when using a stack memory pool compared to the other two pools. Therefore, if sharing is not required and allocation requests do not exceed the size of the stack memory pool, then the stack memory

pool should be used. Otherwise, the TSS memory pool should be used. When sharing is required, a global or shared memory pool is the only valid option.

6.6.1 Client-side Memory Management in the ORB

This section examines the design of the client-side memory management mechanism used in the ORB. This design is highly influenced by the results obtained in Section 6.6.

If the size of the marshaled user request data is less than the size of stack memory pool, the ORB marshals the user request data into a buffer allocated from the stack memory pool. Otherwise, it marshals the user request data into a buffer allocated from the TSS memory pool. Once marshaled, the ORB tries to send the request data to the server. If there is no network congestion, all the data is delivered to the server. If there is network congestion and only part of the data is delivered to the server, the remaining data is copied into a buffer allocated from the global memory pool. This allows any available thread to complete the delivery of the unsent data once the network congestion subsides at a later time.

This memory management design is well suited for predictable networks with little or no congestion. Network congestion forces an additional allocation and data copy to move the unsent data into a buffer allocated from the global memory pool. However, this is probably not a major performance issue since network congestion has already slowed down application progress and the extra allocation and data copy overhead will probably not be very significant.

6.7 Concluding Remarks

The Real-time CORBA specification [45] has introduced several novel concepts and requirements to the CORBA model. These new requirements for providing end-to-end predictability have added to the challenges faced by the ORB developers. This chapter described how to identify and eliminate sources of unbounded priority inversion in the critical code path of the ORB. This is primarily done using non-multiplexed resources where possible or by bounding priority inversion for shared resources. The chapter also illustrated how certain key ORB components, including collocation and memory management, can be redesigned to improve predictability and performance.

Chapter 7

Empirical Validation of End-to-End Real-time ORB Behavior

Abstract

This chapter presents an empirical analysis of end-to-end ORB behavior. First we illustrate incorrect real-time performance that is characteristic of contemporary middleware solutions that are unable to satisfy QoS requirements. We then perform the same experiments using TAO and show that true end-to-end predictability can be achieved if the underlying middleware (a) respects and propagates thread priorities, (b) avoids unbounded priority inversions, and (c) allows applications to configure and control processor, communication, and memory resources.

7.1 Introduction to Real-time Experiments

The experiments presented here illustrate real-time, deterministic, and predictable behavior of the ORB middleware. These experiments demonstrate end-to-end predictability by utilizing the ORB to propagate and preserve priorities, to exercise strict control over the management of resources, and to avoid unbounded priority inversions. End-to-end predictability of timeliness in a fixed priority CORBA system is defined as:

- Respecting thread priorities for resolving resource contention during the processing of CORBA invocations.
- Bounding the duration of thread priority inversions during end-to-end processing.
- Bounding operation invocations latencies.

In these experiments, different aspects and parameters of the test bed (described in Section 7.2) are varied to ensure that the above conditions are being met by the ORB middleware. Some experiments measure system reaction to increased workload, going from unloaded to an overloaded situation; others measure system reaction to increased best-effort work. The following experiments illustrate the real-time behavior of the ORB:

1. Increasing workload
2. Increasing invocation rate
3. Increasing client and server concurrency
4. Increasing workload in non-RT CORBA
5. Increasing workload in RT-CORBA with lanes
   Increasing priority $\rightarrow$ Increasing rate
6. Increasing workload in RT-CORBA with lanes
   Increasing priority $\rightarrow$ Decreasing rate
7. Increasing best-effort work in non-RT CORBA
8. Increasing best-effort work in RT-CORBA with lanes
9. Increasing workload in RT-CORBA without lanes

The first three experiments are basic non-real-time tests that illustrate the general characteristics of the test bed. All threads in these three experiments have default priority and are scheduled in the default scheduling class. Experiment 1 measures throughput as the workload increases. Experiment 2 measures deadlines made/missed as the target invocation rate goes beyond the system capacity. Experiment 3 measures throughput as both client and server concurrency increases while also adding additional CPU support.

The remaining six experiments have several client threads of different importance. The importance of each thread is mapped to its relative priority. Experiments 4 through 6 measure throughput with and without RT-CORBA as the workload increases. Experiments 7 and 8 measure throughput with and without RT-CORBA as best-effort work increases. Finally, Experiment 9 measures throughput with RT-CORBA thread pools without lanes as workload increases. Experiments 5, 6, and 8 use RT-CORBA thread pools with lanes.

All the RT-CORBA experiments exercise the `CLIENT_PROPAGATED` policy to preserve end-to-end priority. The priority of threads in a thread pool with lanes is fixed. However, the priority of threads in a thread pool without lanes is adjusted to match the priority of the client when processing the CORBA request. Once processing completes, the thread's priority is restored to the priority of the thread pool.

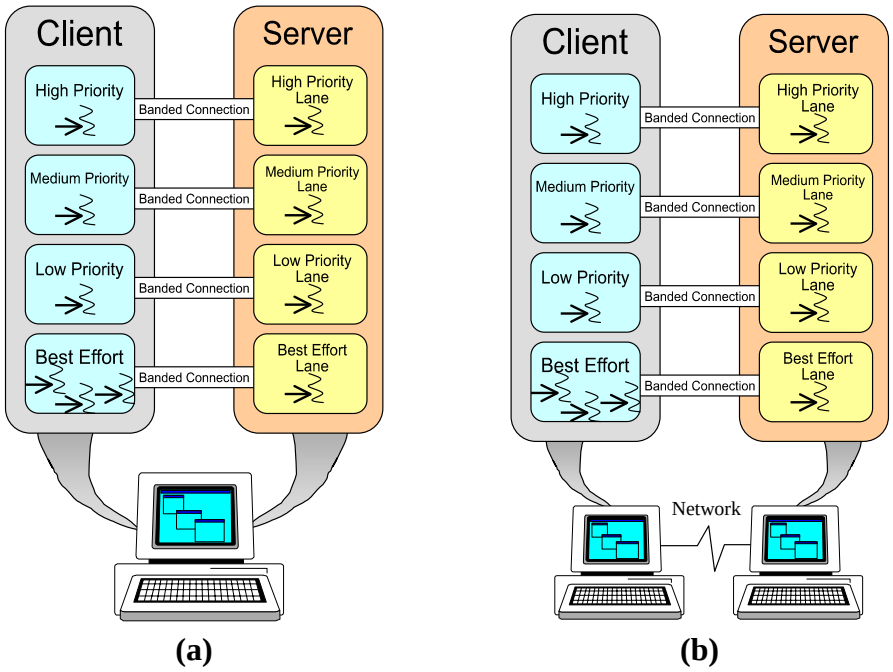


Figure 7.1: Test bed: **(a)** Client and server on same machine **(b)** Client and server distributed across different machines on a network

7.2 Description of Test Bed

Figure 7.1 illustrates and Table 7.1 describes the test bed used for these experiments. To make it easier to understand and interpret the performance results, most experiments have the client and server located on the same machine utilizing a single CPU. Explicit mention is made when the client and server are distributed across different machines on a network or when a second CPU is enabled.

Threads used in the real-time experiments are placed in the First-In-First-Out (FIFO) scheduling class and system scheduling scope. The FIFO scheduling class provides the most predictable behavior since threads scheduled to this policy will proceed to completion unless preempted by a higher priority thread.

CORBA priorities are linearly mapped to native OS priorities and vice versa – on Linux 2.4, `RTCORBA::maxPriority` of 32767 maps to the maximum priority in the FIFO scheduling class of 99, `RTCORBA::minPriority` of 0 maps to the minimum priority in the FIFO scheduling class of 1, and everything in between is evenly apportioned[1].

[1]Similar performance results were obtained when these experiments were conducted on Solaris 2.7. Similar behavior is also expected on other platforms with comparable real-time characteristics.

Table 7.1: Description of test bed

Name	hermes.doc.wustl.edu
Hardware Profile	
OS	Linux 2.4 (Redhat 7.1)
Processor (2)	Intel Pentium III 930 MHz
Memory	500 Megabytes
CPU Cache	256 KB
Threads Profile	
ORBSchedPolicy	SCHED_FIFO
ORBScopePolicy	SYSTEM
ORBPriorityMapping	linear
Priority Profile	
High Priority Lane	32767
Medium Priority Lane	21844
Low Priority Lane	10922
Best Effort Lane	0

The CORBA priority range is divided up evenly such that the high priority thread lane is assigned 32767, medium priority lane is assigned 21844, low priority lane is assigned 10922, and the best-effort thread lane is assigned 0.

7.2.1 Invocation

The invocation made by the client on the server is:

```
void method (in unsigned long work);
```

The `<work>` parameter specifies the amount of CPU intensive work the server will perform to service this invocation. Hence, the higher the value of `<work>`, the more the load on the server.

7.2.2 Rate-based Threads

Rate-based threads are identified by their frequency of invocation. An H Hertz thread tries to make H invocations to the server every second. The period P of a rate-based thread is the multiplicative inverse of its frequency. E is the time it takes for an invocation to complete and it depends on the `<work>` and the QoS it receives from the client endsystem, the network, and the server endsystem.

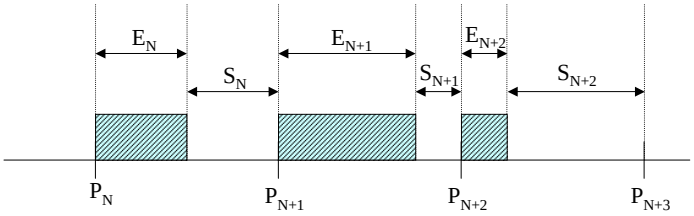


Figure 7.2: Invocation completes within its period

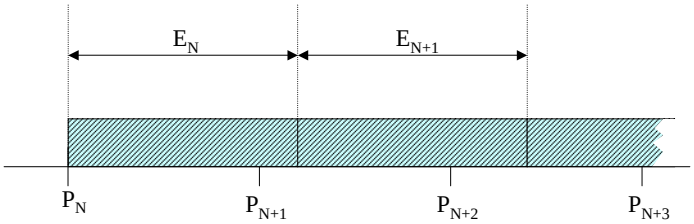


Figure 7.3: Invocation takes longer than its period

In presenting the performance results, an invocation is considered to have missed its deadline when it cannot be started within its period. No explicit consideration is given to the time when the invocation completes. However, these experiments can also be configured with more stringent requirements to consider an invocation to have missed its deadline when it cannot complete execution within its period. There are three scenarios of an invocation's execution with respect to its period:

Invocation completes within period (Figure 7.2): A rate-based thread sleeps for time S equal to $(P - E)$ before making its next invocation. No deadlines are missed in this case.

Invocation execution time exceeds period (Figure 7.3): The execution time of invocation N (E_N) is such that invocation $N + 1$ is invoked immediately since there is no time to sleep, *i.e.*, $(P - E) <= 0$. Note that since the invocations are twoway CORBA calls, a second invocation can only be made once the first one completes. Also notice that according to the above definition of a missed deadline, invocations N and $N + 1$ did not miss their deadlines. However, if the execution time is consistently greater than the period, then $(E - P)/E$ deadlines will be missed.

Invocation misses deadline (Figure 7.4): The execution time of invocation N (E_N) is such that invocation $N + 1$ could not be made during time P_{N+1} through P_{N+2}. In this case, invocation $N + 1$ missed its deadline and was not invoked.

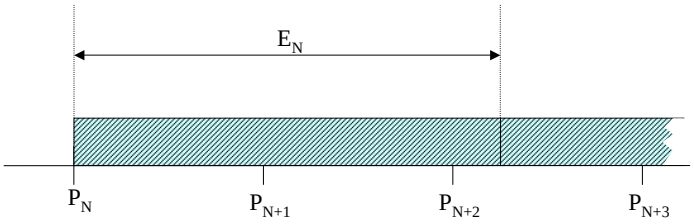


Figure 7.4: Invocation misses deadline

7.2.3 Continuous Threads

Continuous threads make continuous invocations on the server, *i.e.*, they do not pause between successive invocations.

7.3 Experiment Configurations and Performance Results

This section describes the configurations used and the performance results obtained for the various real-time CORBA experiments performed.

Experiment 1: Increasing Workload

This experiment measures the affect of increasing workload in the test bed. As shown in Figure 7.5 (a), the server has one thread handling incoming requests and the client has one continuous thread making invocations. Workload is increased by increasing the `<work>` parameter in the invocation and hence making the server do more work for every client request.

The performance graph in Figure 7.5 (b) plots the throughput achieved (invocations/second) as the workload increases. As the workload increases, the throughput decreases.

Experiment 2: Increasing Invocation Rate

This experiment measures the affect of increasing the target frequency for a rate-based thread beyond the capacity of the system. As shown in Figure 7.6 (a), the server has one thread handling incoming requests and the client has one rate-based thread making invocations. The frequency of the rate-based thread is increased, eventually exceeding the capacity of the system. Workload is kept constant in this experiment at 30. Correlating this

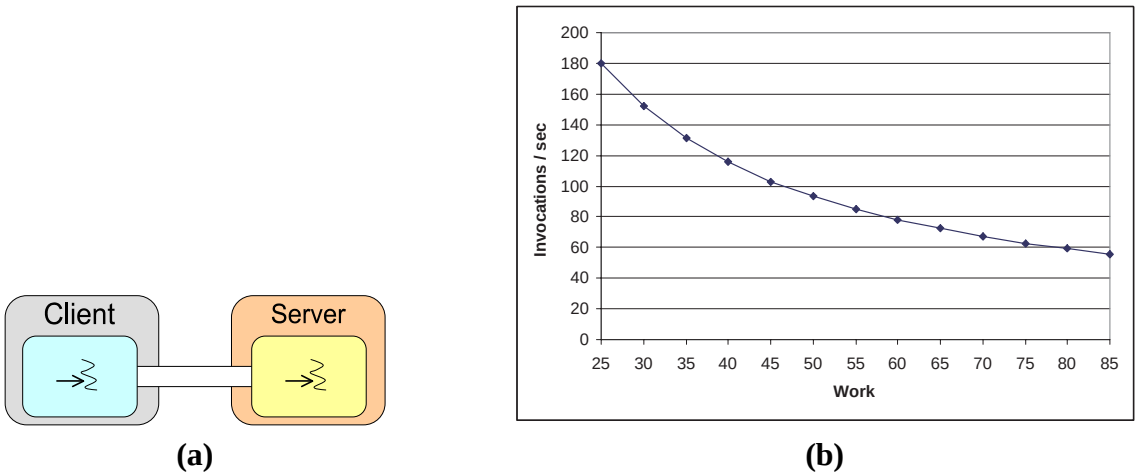


Figure 7.5: Increasing workload: **(a)** Configuration of test bed **(b)** Performance measurements

workload to the performance graph in Experiment 1 (Figure 7.5 (b)), note that the continuous thread was able to make about 150 invocations/second. Therefore, system capacity is estimated at 150 invocations/second with a workload of 30.

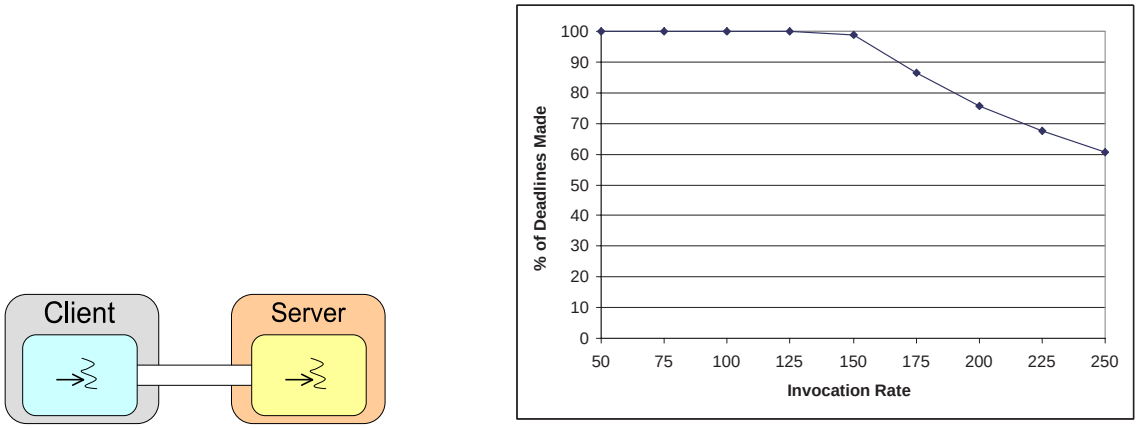
The performance graph in Figure 7.6 (b) plots the percentage of deadline made as the target frequency of the rate-based thread increases. Until about the point when the target frequency increases to 150 invocations/second, the rate-based thread is able to meet 100% of its deadlines. Once the target frequency goes beyond 150 invocations/second, the rate-based thread started missing deadlines. The number of deadlines missed increases with the increased target frequency.

Experiment 3: Increasing Client and Server Concurrency

This experiment measures the affect of increasing the concurrency of both the client and the server as shown in Figure 7.7 (a). The following three server configurations are used in this experiment:

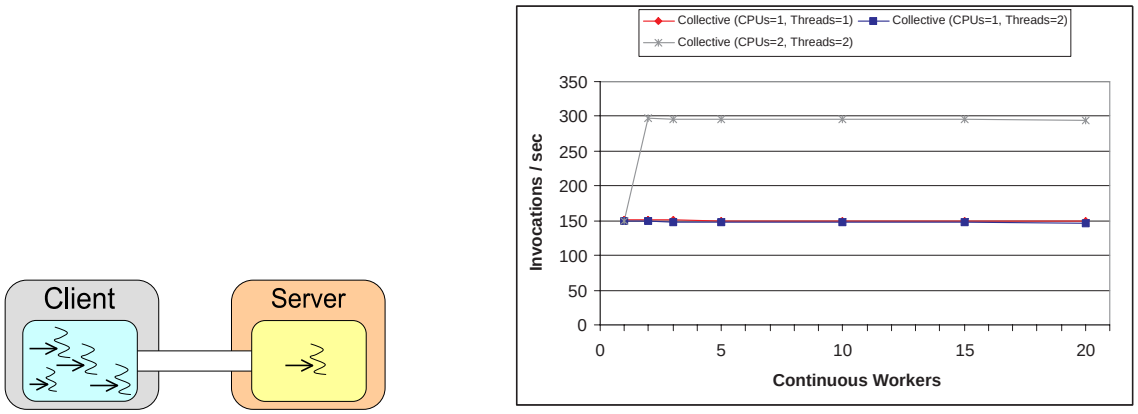
1. One thread to handle incoming requests; one CPU utilized.
2. Two threads to handle incoming requests; one CPU utilized.
3. Two threads to handle incoming requests; two CPUs utilized.

For each of the above server configurations, the number of client threads making continuous invocations is increased from 1 to 20. As in Experiment 2, workload is kept constant at 30.



(a) **(b)**

Figure 7.6: Increasing invocation rate: **(a)** Configuration of test bed **(b)** Performance measurements



(a) **(b)**

Figure 7.7: Increasing concurrency: **(a)** Configuration of test bed **(b)** Performance measurements

The performance graph in Figure 7.7 (b) plots the collective/cumulative throughput achieved for all the client threads as the number of client threads increases.

Configuration 1 (one server thread, one CPU): Increasing client concurrency did not affect collective throughput of the client threads: as the number of client threads increased, the throughput per thread decreased, but the collective throughput remained constant. This is because most of the time is spent on the server and increasing client concurrency does not affect the collective throughput for the client threads.

Configuration 2 (two server threads, one CPU): The results are almost identical to server configuration 1 (one server thread, one CPU). Increasing server concurrency without improving hardware support does not improve throughput when all the work is CPU bound. In fact, the throughput degrades slightly since the server now has to coordinate and synchronize the two threads on one CPU.

Configuration 3 (two server threads, two CPUs): Once the number of client threads reaches two, the collective throughput doubles. The second client thread is able to engage the second server thread, thus doubling the throughput[2]. Further increasing the number of client threads does not improve collective throughput since both server threads are already engaged.

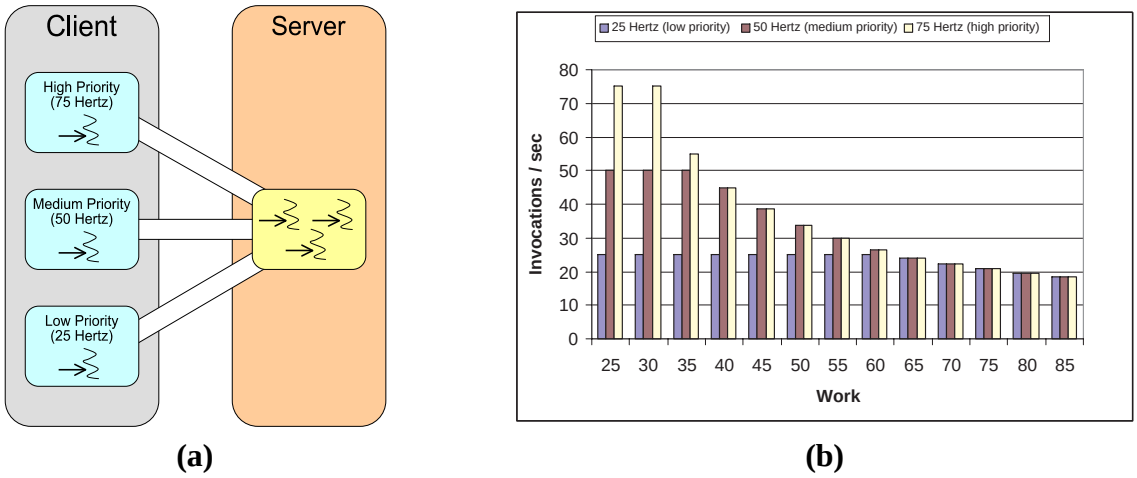
Experiment 4: Increasing Workload in Non-RT CORBA

This experiment measures the disruption caused by increasing workload in non-RT CORBA. As shown in Figure 7.8 (a), the server has three threads handling incoming requests[3]. The client has three rate-based threads of different importance – the high priority thread is at 75 Hertz, the medium priority thread is at 50 Hertz, and the low priority thread is at 25 Hertz.

The performance graph in Figure 7.8 (b) plots the throughput achieved for each of the three client threads as the workload increases. The combined capacity desired by the three client threads is 150 invocations/second (75 + 50 + 25). Correlating this desired throughput of 150 invocations/second to the performance graph in Experiment 1 (Figure 7.5 (b)), note that the continuous thread was able to achieve that throughput with a

[2]Some applications may not be able to double the throughput using this configuration if synchronization is required in the servant code while processing the CORBA requests.

[3]It is not necessary that there be three threads handling incoming requests on the server. As shown in Figure 7.7 (b), one thread is sufficient since increasing server concurrency without increasing CPU support does not affect throughput when the work is CPU bound.



(a) **(b)**

Figure 7.8: Increasing workload in non-RT CORBA: **(a)** Configuration of test bed **(b)** Performance measurements

workload of 30 or less. Therefore, for workloads of 25 and 30 in Figure 7.8 (b), each of the three client threads is able to achieve their desired frequency since the system capacity has not been exceeded.

However, once the workload is increased beyond 30, deadlines start to be missed. The expected behavior of a real-time system is to drop requests from client threads of lower priority before dropping requests from those of higher priority. Unfortunately, since all the three clients are treated equally by the server in non-RT CORBA, the first to be affected is the high priority 75 Hertz client thread, followed by the medium priority 50 Hertz client thread, and finally by the low priority 25 Hertz client thread. This behavior is unacceptable for a real-time system.

Experiment 5: Increasing Workload in RT-CORBA with Lanes: Increasing Priority $\rightarrow$ Increasing Rate

This experiment measures the disruption caused by increasing workload in RT-CORBA with lanes. As shown in Figure 7.9, the server has three thread lanes of high, medium, and low priorities handling incoming requests. Each lane has one thread. The client is identical to the client in Experiment 4.

The performance graph in Figure 7.10 (a) plots the throughput achieved for each of the three client threads as the workload increases. Note that for workloads of 25 and 30,

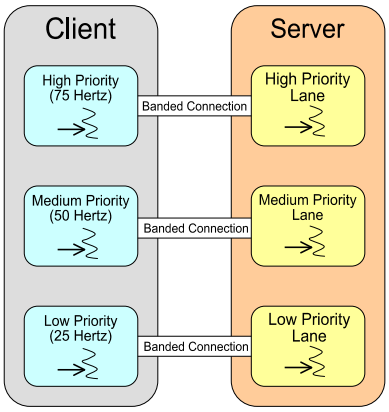


Figure 7.9: Increasing workload in RT-CORBA (increasing priority $\rightarrow$ increasing rate): Configuration of test bed

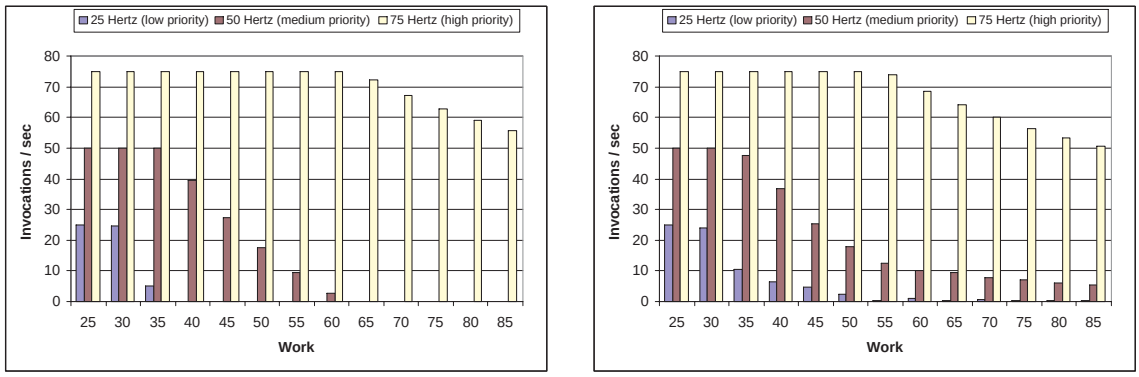
each of the three client threads is able to achieve its desired frequency since the system capacity has not been exceeded.

However, once the workload is increased beyond 30, deadlines start to be missed. Unlike in the non-RT CORBA experiment (Figure 7.8 (b)), the first to be affected is the low priority 25 Hertz client thread, followed by the medium priority 50 Hertz client thread, and finally by the high priority 75 Hertz client thread. This behavior is expected from a real-time system.

Figure 7.10 (b) shows the performance graph of the same experiment, except with the client and server on two different machines across a network. The results are similar to the result when the client and server are on the same machine (Figure 7.10 (a)). However, between the time the high priority server thread sent a reply to the high priority client thread and before it receives a new request from it, the server is free to process a request from a client thread of lower priority. Therefore, the medium priority 50 Hertz client thread is able to make some progress.

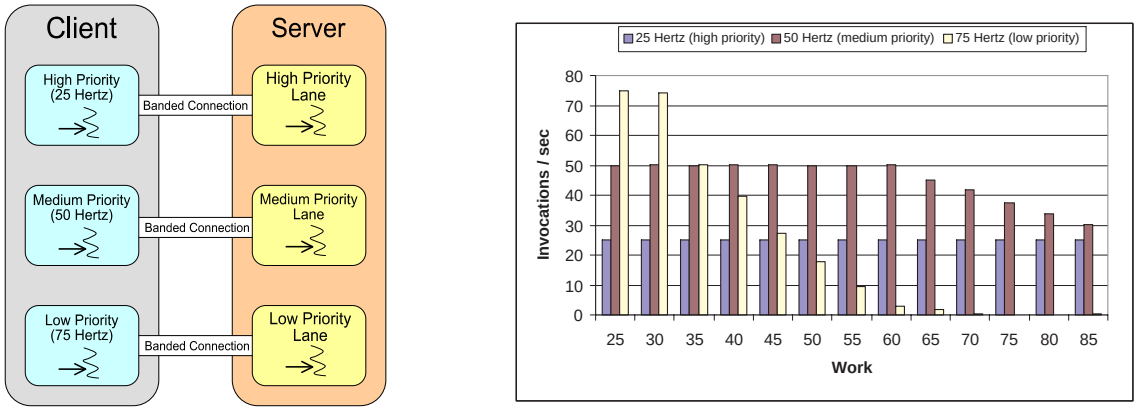
Experiment 6: Increasing Workload in RT-CORBA with Lanes: Increasing Priority $\rightarrow$ Decreasing Rate

This experiment is similar to Experiment 5 except that, as shown in Figure 7.11 (a), the high priority thread is at 25 Hertz, the medium priority thread is at 50 Hertz, and the low priority thread is at 75 Hertz.



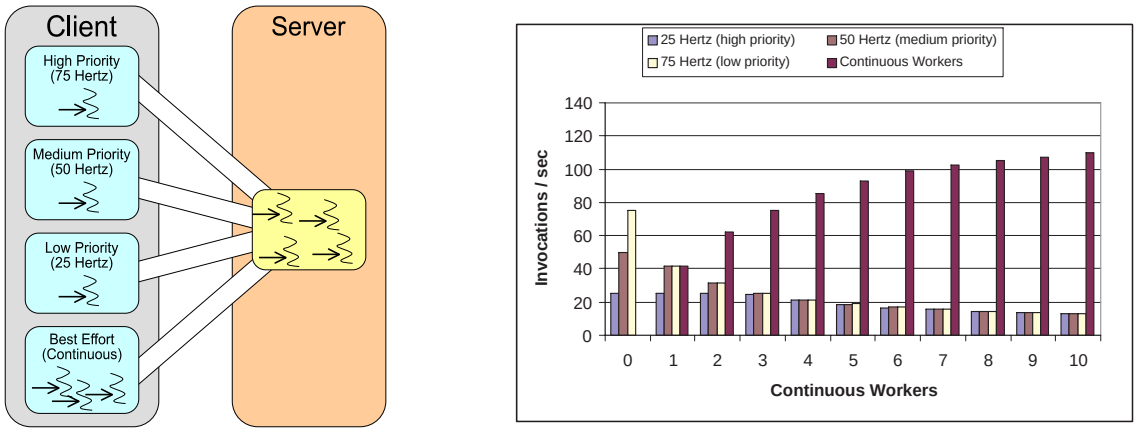
(a) **(b)**

Figure 7.10: Increasing workload in RT-CORBA (increasing priority → increasing rate): Performance measurements: **(a)** Client and server are on the same machine **(b)** Client and server are remote



(a) **(b)**

Figure 7.11: Increasing workload in RT-CORBA (increasing priority → decreasing rate): **(a)** Configuration of test bed **(b)** Performance measurements



(a) **(b)**

Figure 7.12: Increasing best-effort work in non-RT CORBA: **(a)** Configuration of test bed **(b)** Performance measurements

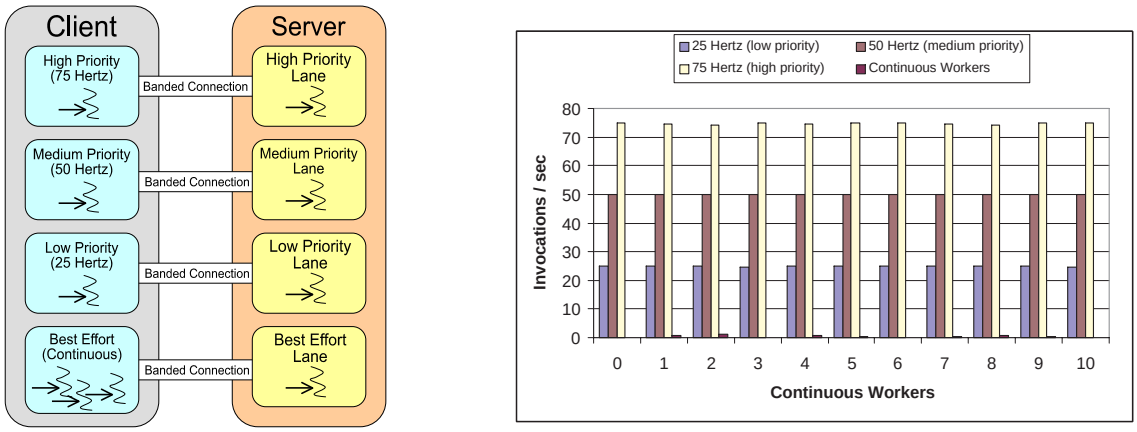
The performance graph in Figure 7.11 (b) shows that the low priority 75 Hertz client thread is affected first, followed by the medium priority 50 Hertz client thread. The high priority 25 Hertz client thread is unaffected since the system capacity never dropped below 25 invocations/second. This behavior is also expected from a real-time system.

Experiment 7: Increasing Best-effort Work in Non-RT CORBA

This experiment measures the disruption caused by increasing best-effort work in non-RT CORBA. As shown in Figure 7.12 (a), the server has four threads handling incoming requests[4]. The client has three rate-based threads of different priorities – the high priority thread is at 75 Hertz, the medium priority thread is at 50 Hertz, and the low priority thread is at 25 Hertz. The client also has a variable number of best-effort threads making continuous invocations. Workload is kept constant at 30, as the number of best-effort continuous client threads is increased from 0 through 10. Note that the system capacity is 150 invocation/second for a workload of 30; therefore, any progress made by the best-effort continuous client threads will cause the rate-based threads to miss deadlines.

The performance graph in Figure 7.12 (b) plots the throughput achieved for each of the three rate-based client threads and the collective throughput achieved by the all the best-effort continuous threads on the client as the number of best-effort continuous threads increases.

[4]For the same reasons noted in Experiment 4, one server thread is sufficient for handling incoming requests.



(a) **(b)**

Figure 7.13: Increasing best-effort work in RT-CORBA: **(a)** Configuration of test bed **(b)** Performance measurements: system running at capacity (work = 30); client and server are on the same machine

When there are no best-effort threads, the three rate-based threads are able to achieve their desired frequency. However, once best-effort threads are added to the client, the non-RT CORBA server treats all the client threads equally, and hence the collective throughput of the best-effort threads increases at the expense of the higher priority threads. This behavior is unacceptable for a real-time system.

Experiment 8: Increasing Best-effort Work in RT-CORBA with Lanes

This experiment measures the disruption caused by increasing best-effort work in RT-CORBA with lanes. As shown in Figure 7.13 (a), the server processes requests using four thread lanes of high, medium, low, and best-effort priorities. Each lane has one thread. The client is identical to the client in Experiment 7.

The performance graph in Figure 7.13 (b) shows that best-effort continuous threads are not able to affect the higher priority rate-based threads. This behavior is expected from a real-time system.

Figure 7.14 (a) shows the performance graph of the same experiment but with a slightly lower workload (`<work>=28`). Note that the slack produced by the lower workload is used by the best-effort threads. Also, increasing the number of best-effort threads does not lead to any increase in the collective throughput of the best-effort threads and the best-effort threads are not able to disrupt the higher priority rate-based threads. This behavior is expected from a real-time system.

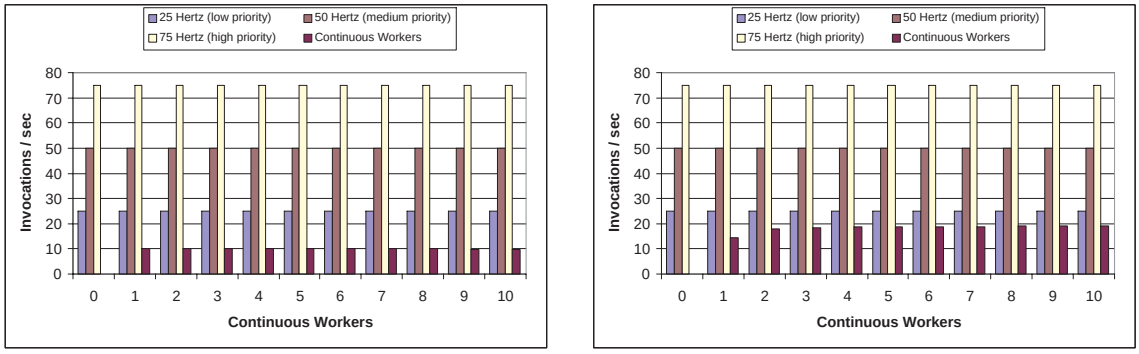


Figure 7.14: Increasing best-effort work in RT-CORBA: Performance measurements: system running slightly below capacity (work = 28): **(a)** Client and server are on the same machine **(b)** Client and server are remote

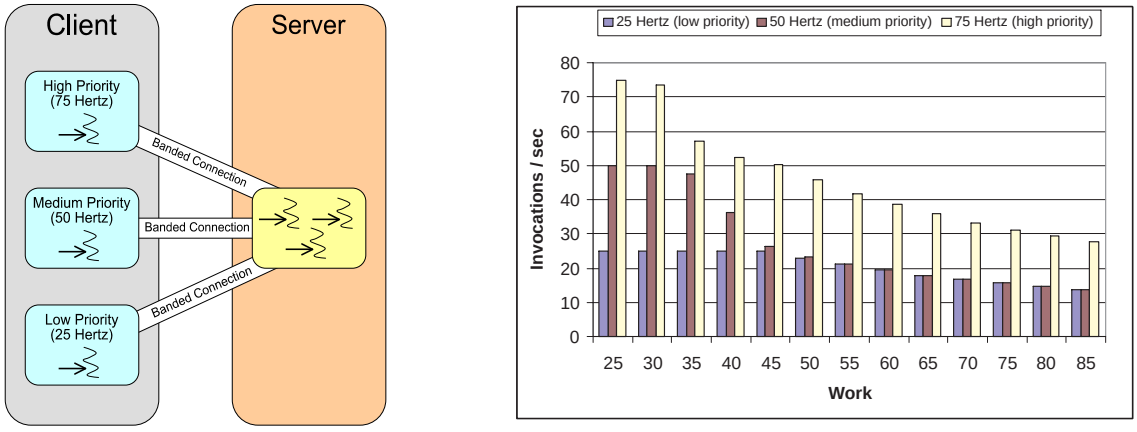
Figure 7.14 (b) shows the performance graph of the same experiment (`<work>=28`) but with the client and server on different machines across a network. Note that the slack available for the best-effort threads increases since all the client processing is now performed on the machine hosting the client, freeing up the server to do additional work.

Experiment 9: Increasing Workload in RT-CORBA without Lanes

This experiment measures the disruption caused by increasing workload in RT-CORBA without lanes. As shown in Figure 7.15 (a), the server has a thread pool of three threads handling incoming requests. The client is identical to the client in Experiment 4 and 5.

The performance graphs in Figures 7.15 (b), 7.16 (a), and 7.16 (b) plot the throughput achieved for each of the three client threads as the workload increases. The difference between the three graphs is the priority of the server thread pool: in Figure 7.15 (b) the thread pool runs at low priority, in Figure 7.16 (a) the thread pool priority runs at medium priority, and in Figure 7.16 (b) the thread pool runs at high priority.

Server thread pool priority = low (Figure 7.15 (b)): Assume that one of the server threads is processing a request from a client thread of low priority. During that time, a request arrives at the server from a higher priority client thread. Unfortunately, since the request processing thread has the same priority as the waiting thread, the waiting thread is not able to preempt the processing thread. The request from the higher priority client thread has to wait until the low priority client thread request has been completely processed.



(a) **(b)**

Figure 7.15: Increasing workload in RT-CORBA without lanes: **(a)** Configuration of test bed **(b)** Performance measurements: server thread pool priority = low

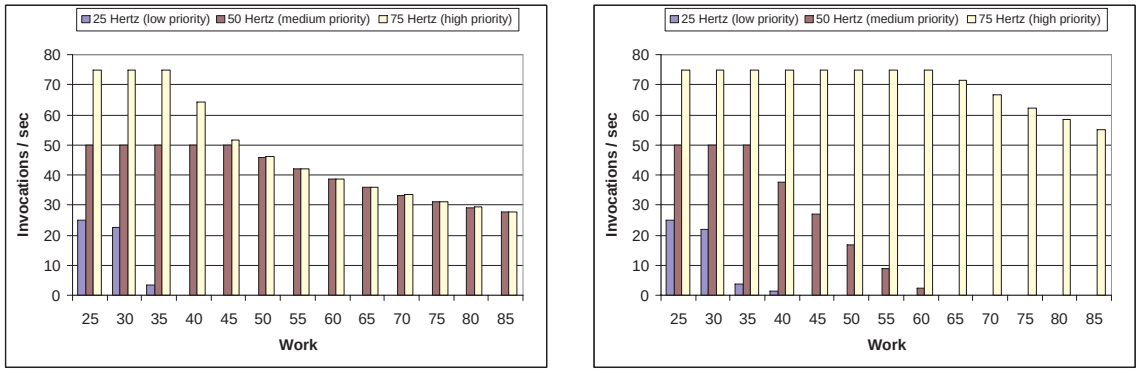
This leads to the three client threads being treated equally by the server. This behavior is unacceptable for a real-time system.

Server thread pool priority = medium (Figure 7.16 (a)): Assume that one of the server threads is processing a request from a client thread of medium priority. During this time, a request arrives at the server from the high priority client thread. Unfortunately, since the request processing thread has the same priority as the waiting thread, the waiting thread is not able to preempt the request processing thread. The request from the high priority client thread has to wait until the medium priority client thread request has been completely processed.

However, the same does not apply when a medium or high priority request arrives while a low priority request is being processed. Since the priority of the waiting thread is greater than that of the priority of the request processing thread, it is able to preempt the request processing thread and handle the higher priority request.

This behavior leads to the medium and high priority client threads being treated equally but are given preference over the low priority client thread. However, this behavior is also unacceptable for a real-time system.

Server thread pool priority = high (Figure 7.16 (b)): Assume that one of the server threads is processing a request from a client thread of low priority. During this time, a request arrives at the server from a higher priority client thread. Since the priority of the waiting thread is greater than the priority of the request processing thread, it is able to



(a) **(b)**

Figure 7.16: Increasing workload in RT-CORBA without lanes: Performance measurements: **(a)** server thread pool priority = medium **(b)** server thread pool priority = high

preempt the request processing thread and handle the higher priority request. Note that a high priority request can preempt both low and medium priority requests.

This behavior leads to the high priority client thread getting preference over the low and medium priority client threads and the medium priority client thread getting preference over the low priority client thread. This behavior is expected from a real-time system.

Notes on using RT-CORBA without lanes: The most desirable behavior is achieved in Figure 7.16 (b) where the server thread pool priority is equal to the highest request processing priority. A server using thread pools without lanes is more flexible than a server using thread pools with lanes since the former is able to adapt to any priority propagated by the client. A client invoking requests on a server using thread pools with lanes is restricted to priorities used in the lanes by the server[5].

However, a server without lanes can incur priority inversion. Consider the case where the server thread pool priority is high. Assume that one of the server threads is processing a request from a client thread of medium priority. During this time, a request arrives at the server from the low priority client thread. Since the priority of the waiting thread is greater than the priority of the request processing thread, it is able to preempt the request processing thread to read the incoming request. If the request includes a large amount of data, it can take a significant amount of time to read the request. Once this thread reads the request, it sets its priority to low to match the client propagated priority.

[5]Priority bands can be used map a range of client priorities to a server lane. However, the server lane priority is fixed and therefore, request processing priorities are limited to the priorities of the lanes.

The medium priority request processing thread is now able to preempt the low priority thread and resume processing.

This interruption caused by the low priority request leads to priority inversion for the medium priority thread. Depending on the number of waiting threads in the server thread pool and the time it takes to read the incoming request, the priority inversion can be significant or even unbounded.

7.4 Concluding Remarks

RT-CORBA ORBs must *vertically* and *horizontally* integrate and manage components to ensure end-to-end predictable behavior. The empirical analysis presented in this chapter illustrated that conventional non-RT ORBs are unsuitable for real-time systems that require timeliness guarantees. Conversely, the experimental results validate end-to-end real-time, deterministic behavior of the TAO RT-ORB. However, even when using a RT-ORB, careful consideration must be given to several other factors to ensure end-to-end system predictability and scalability, including (a) the configuration and structure of the client and server, (b) the use of lanes in thread pools, (c) the priorities assigned to the thread pools and lanes used in the server, and (d) the priority propagation and banding policies used. In general, even though thread pools with lanes are not as flexible as their counterparts without lanes, they provide the most predictable execution and do not exhibit unbounded priority inversions.

Chapter 8

Conclusions and Future Research Directions

This thesis focused on presenting designs and techniques for providing real-time QoS guarantees in DOC middleware. Historically, DOC middleware has lacked the ability to specify and enforce QoS and has failed to present real-time programming features to the user. However, the next generation of middleware must be QoS-enabled to support mission-critical distributed applications, including command and control systems, telecom, distributed interactive simulations, and financial services, that require stringent latency, determinism, and priority preservation support.

The work presented in this thesis is in the context of TAO [6], a high-quality, freely available, open-source CORBA-compliant middleware platform. TAO provides a complete implementation of the Real-time CORBA (RT-CORBA) 1.0 specification [45] and meets real-time requirements of fixed priority applications by (a) respecting and propagating thread priorities, (b) avoiding unbounded priority inversions, and (c) allowing applications to configure and control processor, communication, and memory resources. TAO has matured into a stable COTS middleware framework and is being used in many commercial projects, a representation of which is shown in Table 8.1.

Providing end-to-end QoS guarantees required careful engineering of the TAO's subsystems to ensure predictability, scalability, and performance. This work also focused on demultiplexing, dispatching, and concurrency mechanisms that are key components in the critical code path of the ORB. Analysis of the problems, forces, solutions, and consequences are presented in terms of patterns and frameworks, so that solutions obtained here can be appropriately applied to other real-time systems.

Table 8.1: TAO has been successfully used in a variety of domains

Organization	*Domain*
Boeing	Aircraft Mission Control Computer
SAIC	Distributed Interactive Simulation (HLA/RTI-NG)
ATD	Automated Stock Trading
Raytheon	Aircraft Carrier self-defense Systems
Cisco	High-performance network switch control
US Army	Joint Tactical Terminal
Contact Systems	Surface mounted “pick-and-place” machines
Turkish Navy	Shipboard Resource Management
Krones	Process Automation
Siemens	Hot Rolling Mill Control System
Lockheed-Martin	Shipboard Resource Management
CUSeeMe	Monitor H.323 Servers
Northrup-Grumman	Airborne Early Warning & Control
JPL/NASA	SOFIA Airborne Telescope, Cassini Space Probe
Marconi	Joint Tactical Radio System

Finally, the architectural solutions presented here are coupled with empirical evaluations of end-to-end real-time behavior. The evaluations show that real-time, deterministic, and predictable ORB behavior can be achieved by bounding the duration of thread priority inversions end-to-end and by bounding the latencies of operation invocations.

8.1 Future Research Directions

The Object Request Broker (ORB) is the foundation of Common Object Request Broker Architecture (CORBA) [47]. It enables objects to transparently make and receive requests and responses in a distributed environment. It provides the basis for building distributed applications and for interoperability between applications in hetero- and homogeneous environments. The ORB, however, needs to be combined with Object Services, *e.g.*, Naming, Trading, and Transaction Services, and Common Facilities, *e.g.* Secure Time, Internationalization, and Mobile Agent Facilities, to ensure meaningful and productive communication and to provide application semantic interoperability.

Similarly, RT-CORBA forms the basis of real-time distributed computing, and needs to be integrated with higher-level architectures, models, services, and tools to be meaningful and productive. The following integrations are of primary interest:

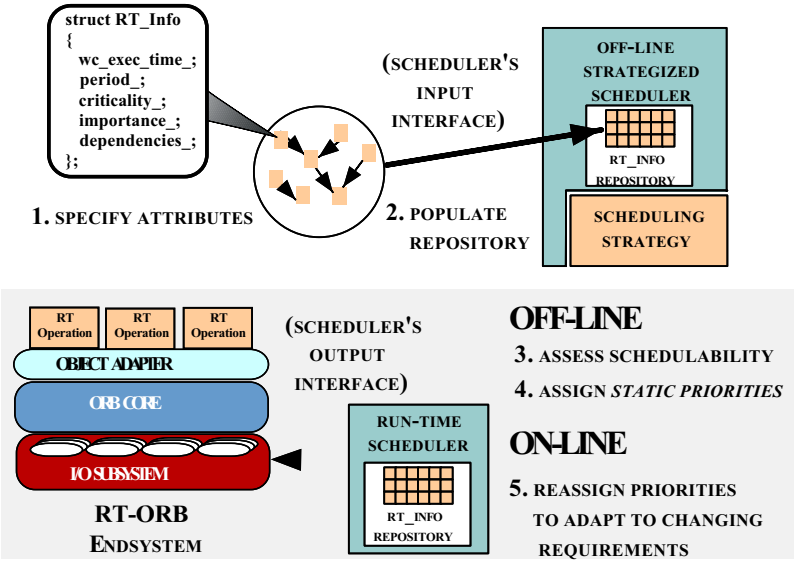


Figure 8.1: Dynamic scheduling

Scheduling Services and Resource Managers: RT-CORBA is well suited for applications using fixed priority scheduling. However, as shown in Figure 8.1, to ensure dynamic end-to-end management of priorities, it must be integrated with dynamic scheduling services [43], such as the Kokyu scheduling framework [17], and adaptive resource management systems, such as the RT-ARM [27] and the QuO [73] frameworks.

Network QoS: RT-CORBA does allow the selection and configuration of protocols used by the ORB. However, it needs to be integrated with network QoS protocols, such as RSVP and DiffServ, to provide true end-to-end predictability that includes the network. RSVP [57] is a network resource reservation protocol based on the Integrated Services (IntServ) model [26] and has good scaling and robustness properties. RSVP reservations are receiver-oriented and scale well for both unicast and multicast communication groups. Conversely, the Differentiated Services model (DiffServ) [25] allows network providers to allocate different levels of service to different users. This is accomplished by using the IP TOS (type of service) field according to a contracted service profile.

Higher-level Services: RT-CORBA also needs to be integrated with higher level services such as the Real-time Notification Service and the CORBA Component Model. The Real-time Notification Service [40] allows real-time systems to enjoy predictable and bounded behavior from the existing Notification Service [44] and would include priority ordering

and deadline scheduling of events. The CORBA Component Model [4] defines a standard configuration framework for packaging and deploying software components. It can be used to configure real-time properties into applications flexibly, transparently, and adaptively [77].

Modeling and Monitoring Tools: Integration of RT-CORBA with modeling and monitoring tools such as RapidRMA, TimeWiz, and TotalView, will make easier to analyze, develop, and use real-time applications. RapidRMA [76] is a Rate Monotonic Analysis (RMA) [35] visual modeling tool produced by Tri-Pacific. It allows real-time systems software developers to provides worst-case schedulability analysis and isolate and identify timing problems. TimeWiz [75] is a performance prediction tool produced by TimeSys. It offers architectural modeling, analysis and simulation for real-time systems. TotalView [38] is a multi-process, distributed debugger produced by LynuxWorks. It is fast and intuitive, making it easy to debug real-time applications running across multiple machines.

References

[1] Alexander B. Arulanthu, Carlos O'Ryan, Douglas C. Schmidt, and Michael Kircher. Applying C++, Patterns, and Components to Develop an IDL Compiler for CORBA AMI Callbacks. *C++ Report*, 12(3), March 2000.

[2] Alexander B. Arulanthu, Carlos O'Ryan, Douglas C. Schmidt, Michael Kircher, and Jeff Parsons. The Design and Performance of a Scalable ORB Architecture for CORBA Asynchronous Messaging. In *Proceedings of the Middleware 2000 Conference*. ACM/IFIP, April 2000.

[3] Mary L. Bailey, Burra Gopal, Prasenjit Sarkar, Michael A. Pagels, and Larry L. Peterson. Pathfinder: A pattern-based packet classifier. In *Proceedings of the* 1^{st} *Symposium on Operating System Design and Implementation*. USENIX Association, November 1994.

[4] BEA Systems, *et al*. *CORBA Component Model Joint Revised Submission*. Object Management Group, OMG Document orbos/99-07-01 edition, July 1999.

[5] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *Pattern-Oriented Software Architecture – A System of Patterns*. Wiley and Sons, New York, 1996.

[6] Center for Distributed Object Computing. The ACE ORB (TAO). www.cs.wustl.edu/∼schmidt/TAO.html, Washington University.

[7] James O. Coplien and Douglas C. Schmidt, editors. *Pattern Languages of Program Design*. Addison-Wesley, Reading, Massachusetts, 1995.

[8] DARPA. The Quorum Program. www.darpa.mil/ito/research/quorum/index.html, 1999.

[9] Zubin D. Dittia, Jerome R. Cox, Jr., and Guru M. Parulkar. Design of the APIC: A High Performance ATM Host-Network Interface Chip. In *IEEE INFOCOM '95*, pages 179–187, Boston, USA, April 1995. IEEE Computer Society Press.

[10] Zubin D. Dittia, Guru M. Parulkar, and Jerome R. Cox, Jr. The APIC Approach to High Performance Network Interface Design: Protected DMA and Other Techniques. In *Proceedings of INFOCOM '97*, pages 179–187, Kobe, Japan, April 1997. IEEE.

[11] Eric Eide, Kevin Frei, Bryan Ford, Jay Lepreau, and Gary Lindstrom. Flick: A Flexible, Optimizing IDL Compiler. In *Proceedings of ACM SIGPLAN '97 Conference on Programming Language Design and Implementation (PLDI)*, Las Vegas, NV, June 1997. ACM.

[12] Dawson R. Engler and M. Frans Kaashoek. DPF: Fast, Flexible Message Demultiplexing using Dynamic Code Generation. In *Proceedings of ACM SIGCOMM '96 Conference in Computer Communication Review*, pages 53–59, Stanford University, California, USA, August 1996. ACM Press.

[13] J.R. Eykholt, S.R. Kleiman, S. Barton, R. Faulkner, A Shivalingiah, M. Smith, D. Stein, J. Voll, M. Weeks, and D. Williams. Beyond Multiprocessing... Multithreading the SunOS Kernel. In *Proceedings of the Summer USENIX Conference*, San Antonio, Texas, June 1992.

[14] Victor Fay-Wolfe, John K. Black, Bhavanai Thuraisingham, and Peter Krupp. Real-time Method Invocations in Distributed Environments. Technical Report 95-244, University of Rhode Island, Department of Computer Science and Statistics, 1995.

[15] David C. Feldmeier. Multiplexing Issues in Communications System Design. In *Proceedings of the Symposium on Communications Architectures and Protocols (SIGCOMM)*, pages 209–219, Philadelphia, PA, September 1990. ACM.

[16] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, Massachusetts, 1995.

[17] Christopher D. Gill, Ron Cytron, and Douglas C. Schmidt. Middleware Scheduling Optimization Techniques for Distributed Real-Time and Embedded Systems. In *Proceedings of the 7^{th} Workshop on Object-oriented Real-time Dependable Systems*, San Diego, CA, January 2002. IEEE.

[18] Christopher D. Gill, David L. Levine, and Douglas C. Schmidt. The Design and Performance of a Real-Time CORBA Scheduling Service. *Real-Time Systems, The International Journal of Time-Critical Computing Systems, special issue on Real-Time Middleware*, 20(2), March 2001.

[19] Aniruddha Gokhale and Douglas C. Schmidt. Measuring the Performance of Communication Middleware on High-Speed Networks. In *Proceedings of SIGCOMM '96*, pages 306–317, Stanford, CA, August 1996. ACM.

[20] Aniruddha Gokhale and Douglas C. Schmidt. Measuring and Optimizing CORBA Latency and Scalability Over High-speed Networks. *Transactions on Computing*, 47(4), 1998.

[21] Aniruddha Gokhale and Douglas C. Schmidt. Principles for Optimizing CORBA Internet Inter-ORB Protocol Performance. In *Hawaiian International Conference on System Sciences*, January 1998.

[22] Aniruddha Gokhale and Douglas C. Schmidt. Optimizing a CORBA IIOP Protocol Engine for Minimal Footprint Multimedia Systems. *Journal on Selected Areas in Communications special issue on Service Enabling Platforms for Networked Multimedia Systems*, 17(9), September 1999.

[23] Timothy H. Harrison, David L. Levine, and Douglas C. Schmidt. The Design and Performance of a Real-time CORBA Event Service. In *Proceedings of OOPSLA '97*, pages 184–199, Atlanta, GA, October 1997. ACM.

[24] Michi Henning and Steve Vinoski. *Advanced CORBA Programming With C++*. Addison-Wesley, Reading, Massachusetts, 1999.

[25] Internet Engineering Task Force. Differentiated Services Working Group (diffserv) Charter. www.ietf.org/html.charters/diffserv-charter.html, 2000.

[26] Internet Engineering Task Force. Integrated Services Working Group (intserv) Charter. www.ietf.org/html.charters/intserv-charter.html, 2000.

[27] J. Huang and R. Jha and W. Heimerdinger and M. Muhammad and S. Lauzac and B. Kannikeswaran and K. Schwan and W. Zhao and R. Bettati. RT-ARM: A Real-Time Adaptive Resource Management System for Distributed Mission-Critical Applications. In *Workshop on Middleware for Distributed Real-Time Systems, RTSS-97*, San Francisco, California, 1997. IEEE.

[28] Mahesh Jayaram and Ron Cytron. Efficient Demultiplexing of Network Packets by Automatic Parsing. In *Proceedings of the Workshop on Compiler Support for System Software (WCSSS 96)*, University of Arizona, Tucson, AZ, February 1996.

[29] E. Douglas Jensen. Eliminating the Hard/Soft Real-Time Dichotomy. *Embedded Systems Programming*, 7(10), October 1994.

[30] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-Oriented Programming. In *Proceedings of the 11th European Conference on Object-Oriented Programming*, June 1997.

[31] K. H. (Kane) Kim. Object Structures for Real-Time Systems and Simulators. *IEEE Computer*, pages 62–70, August 1997.

[32] Kane Kim and Eltefaat Shokri. Two CORBA Services Enabling TMO Network Programming. In *Fourth International Workshop on Object-Oriented, Real-Time Dependable Systems*. IEEE, January 1999.

[33] Fred Kuhns, Douglas C. Schmidt, and David L. Levine. The Design and Performance of RIO – A Real-time I/O Subsystem for ORB Endsystems. In *Proceedings of the International Symposium on Distributed Objects and Applications (DOA'99)*, Edinburgh, Scotland, September 1999. OMG.

[34] Fred Kuhns, Douglas C. Schmidt, Carlos O'Ryan, and David Levine. Supporting High-performance I/O in QoS-enabled ORB Middleware. *Cluster Computing: the Journal on Networks, Software, and Applications*, 3(3), 2000.

[35] J. Lehoczky, L. Sha, and Y. Ding. The Rate Monotonic Scheduling Algorithm: Exact Characterization and Average Case Behavior. In *Proceedings of the 10th IEEE Real-Time Systems Symposium*, pages 166–171. IEEE Computer Society Press, 1989.

[36] David L. Levine, Douglas C. Schmidt, and Sergio Flores-Gaitan. An Empirical Evaluation of OS Support for Real-time CORBA Object Request Brokers. In *Proceedings of Multimedia Computing and Networking 2000 (MMCN00)*, San Jose, CA, January 2000. ACM.

[37] J. Loyall, J. Gossett, C. Gill, R. Schantz, J. Zinky, P. Pal, R. Shapiro, C. Rodrigues, M. Atighetchi, and D. Karr. Comparing and Contrasting Adaptive Middleware Support in Wide-Area and Embedded Distributed Object Applications. In *Proceedings*

of the 21st International Conference on Distributed Computing Systems (ICDCS-21), pages 625–634. IEEE, April 2001.

[38] LynuxWorks. TotalView. http://www.lynuxworks.com, 2002.

[39] Steven McCanne and Van Jacobson. The BSD Packet Filter: A New Architecture for User-level Packet Capture. In *Proceedings of the Winter USENIX Conference*, pages 259–270, San Diego, CA, January 1993.

[40] Middleware And Related Services (MARS) Platform Task Force. RT Notification Request For Proposals, June 2000.

[41] Jeffrey C. Mogul, Richard F. Rashid, and Michal J. Accetta. The Packet Filter: an Efficient Mechanism for User-level Network Code. In *Proceedings of the* 11^{th} *Symposium on Operating System Principles (SOSP)*, November 1987.

[42] Object Management Group. *CORBAServices: Common Object Services Specification, Revised Edition*, 95-3-31 edition, March 1995.

[43] Object Management Group. *Dynamic Scheduling*, OMG Document orbos/99-03-32 edition, March 1999.

[44] Object Management Group. *Notification Service Specification*, OMG Document telecom/99-07-01 edition, July 1999.

[45] Object Management Group. *Real-time CORBA Joint Revised Submission*, OMG Document orbos/99-02-12 edition, March 1999.

[46] Object Management Group. *The Common Object Request Broker: Architecture and Specification*, 2.4 edition, October 2000.

[47] Object Management Group. *The Common Object Request Broker: Architecture and Specification*, 2.6 edition, December 2001.

[48] Carlos O'Ryan, Fred Kuhns, Douglas C. Schmidt, Ossama Othman, and Jeff Parsons. The Design and Performance of a Pluggable Protocols Framework for Real-time Distributed Object Computing Middleware. In *Proceedings of the Middleware 2000 Conference*. ACM/IFIP, April 2000.

[49] Carlos O'Ryan, Douglas C. Schmidt, David Levine, and Russell Noseworthy. Applying a Scalable CORBA Events Service to Large-scale Distributed Interactive Simulations. In *Proceedings of the 5^{th} Workshop on Object-oriented Real-time Dependable Systems*, Montery, CA, November 1999. IEEE.

[50] Guru Parulkar, Douglas C. Schmidt, and Jonathan S. Turner. $\mathrm{a^{I}t^{P}m}$: a Strategy for Integrating IP with ATM. In *Proceedings of the Symposium on Communications Architectures and Protocols (SIGCOMM)*. ACM, September 1995.

[51] Paul E. McKinney. Selecting Locking Designs for Parallel Programs. In James O. Coplien, John Vlissides, and Norm Kerth, editors, *Pattern Languages of Program Design*. Addison-Wesley, Reading, Massachusetts, 1996.

[52] Irfan Pyarali, Timothy H. Harrison, Douglas C. Schmidt, and Thomas D. Jordan. Proactor – An Architectural Pattern for Demultiplexing and Dispatching Handlers for Asynchronous Events. In Brian Foote, Neil Harrison, and Hans Rohnert, editors, *Pattern Languages of Program Design*. Addison-Wesley, Reading, Massachusetts, 1999.

[53] Irfan Pyarali, Carlos O'Ryan, and Douglas C. Schmidt. A Pattern Language for Efficient, Predictable, Scalable, and Flexible Dispatching Mechanisms for Distributed Object Computing Middleware. In *Proceedings of the International Symposium on Object-Oriented Real-time Distributed Computing (ISORC)*, Newport Beach, CA, March 2000. IEEE/IFIP.

[54] Irfan Pyarali, Carlos O'Ryan, Douglas C. Schmidt, Nanbor Wang, Vishal Kachroo, and Aniruddha Gokhale. Using Principle Patterns to Optimize Real-time ORBs. *IEEE Concurrency Magazine*, 8(1), 2000.

[55] Irfan Pyarali, Carlos O'Ryan, Douglas C. Schmidt, Nanbor Wang, Vishal Kachroo, and Aniruddha Gokhale. Applying Optimization Patterns to the Design of Real-time ORBs. In *Proceedings of the 5^{th} Conference on Object-Oriented Technologies and Systems*, San Diego, CA, May 1999. USENIX.

[56] Irfan Pyarali and Douglas C. Schmidt. An Overview of the CORBA Portable Object Adapter. *ACM StandardView*, 6(1), March 1998.

[57] R. Braden et al. Resource ReSerVation Protocol (RSVP) Version 1 Functional Specification. *Network Working Group RFC 2205*, pages 1–112, Sep 1997.

[58] Ragunathan Rajkumar, Lui Sha, and John P. Lehoczky. Real-Time Synchronization Protocols for Multiprocessors. In *Proceedings of the Real-Time Systems Symposium*, pages 259–269, Huntsville, Alabama, December 1988.

[59] Real-time Java Experts Group. *Real-time Java Specification*.

[60] Real-time Linux. www.realtimelinux.org.

[61] Jeffrey Richter. *Advanced Windows, Third Edition*. Microsoft Press, Redmond, WA, 1997.

[62] Douglas C. Schmidt. GPERF: A Perfect Hash Function Generator. In *Proceedings of the* 2^{nd} *C++ Conference*, pages 87–102, San Francisco, California, April 1990. USENIX.

[63] Douglas C. Schmidt. An OO Encapsulation of Lightweight OS Concurrency Mechanisms in the ACE Toolkit. Technical Report WUCS-95-31, Washington University, St. Louis, September 1995.

[64] Douglas C. Schmidt. Reactor: An Object Behavioral Pattern for Concurrent Event Demultiplexing and Event Handler Dispatching. In James O. Coplien and Douglas C. Schmidt, editors, *Pattern Languages of Program Design*, pages 529–545. Addison-Wesley, Reading, Massachusetts, 1995.

[65] Douglas C. Schmidt. Strategized Locking, Thread-safe Interface, and Scoped Locking: Patterns and Idioms for Simplifying Multi-threaded C++ Components. *C++ Report*, 11(8), September 1999.

[66] Douglas C. Schmidt, David L. Levine, and Sumedh Mungee. The Design and Performance of Real-Time Object Request Brokers. *Computer Communications*, 21(4):294–324, April 1998.

[67] Douglas C. Schmidt, Sumedh Mungee, Sergio Flores-Gaitan, and Aniruddha Gokhale. Alleviating Priority Inversion and Non-determinism in Real-time CORBA ORB Core Architectures. In *Proceedings of the* 4^{th} *IEEE Real-Time Technology and Applications Symposium*, Denver, CO, June 1998. IEEE.

[68] Douglas C. Schmidt, Sumedh Mungee, Sergio Flores-Gaitan, and Aniruddha Gokhale. Software Architectures for Reducing Priority Inversion and Non-determinism in Real-time Object Request Brokers. *Journal of Real-time Systems*,

special issue on Real-time Computing in the Age of the Web and the Internet, 21(2), 2001.

[69] Douglas C. Schmidt, Carlos O'Ryan, Irfan Pyarali, Michael Kircher, and Frank Buschmann. Leader/Followers: A Design Pattern for Efficient Multi-threaded Event Demultiplexing and Dispatching. In *Proceedings of the* 6^{th} *Pattern Languages of Programming Conference*, Monticello, Illinois, August 2000.

[70] Douglas C. Schmidt, Michael Stal, Hans Rohnert, and Frank Buschmann. *Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects, Volume 2*. Wiley & Sons, New York, 2000.

[71] Douglas C. Schmidt and Tatsuya Suda. Measuring the Performance of Parallel Message-based Process Architectures. In *Proceedings of the Conference on Computer Communications (INFOCOM)*, pages 624–633, Boston, April 1995. IEEE.

[72] Douglas C. Schmidt and Steve Vinoski. Developing C++ Servant Classes Using the Portable Object Adapter. *C++ Report*, 10(5), June 1998.

[73] BBN Technologies. Quality Objects (QuO). www.dist-systems.bbn.com/papers.

[74] David L. Tennenhouse. Layered Multiplexing Considered Harmful. In *Proceedings of the* 1^{st} *International Workshop on High-Speed Networks*, May 1989.

[75] TimeSys. TimeWiz. http://www.timesys.com, 2002.

[76] Tri-Pacific. RapidRMA. http://www.tripac.com, 2002.

[77] Nanbor Wang, Douglas C. Schmidt, Kirthika Parameswaran, and Michael Kircher. Applying Reflective Middleware Techniques to Optimize a QoS-enabled CORBA Component Model Implementation. In *24th Computer Software and Applications Conference*, Taipei, Taiwan, October 2000. IEEE.

[78] Yu-Chung Wang and Kwei-Jay Lin. Implementing A General Real-Time Scheduling Framework in the RED-Linux Real-Time Kernel. In *IEEE Real-Time Systems Symposium*, pages 246–255. IEEE, December 1999.

[79] Victor Fay Wolfe, Lisa Cingiser DiPippo, Roman Ginis, Michael Squadrito, Steven Wohlever, Igor Zykh, and Russel Johnston. Real-Time CORBA. In *Proceedings of the Third IEEE Real-Time Technology and Applications Symposium*, Montréal, Canada, June 1997.

[80] M. Yuhara, B. Bershad, C. Maeda, and E. Moss. Efficient Packet Demultiplexing for Multiple Endpoints and Large Messages. In *Proceedings of the Winter Usenix Conference*, January 1994.

[81] John A. Zinky, David E. Bakken, and Richard Schantz. Architectural Support for Quality of Service for CORBA Objects. *Theory and Practice of Object Systems*, 3(1):1–20, 1997.

Vita

Irfan Pyarali

Date of Birth August 1, 1973

Place of Birth Karachi, Pakistan

Degrees B.S. Computer Science, May 1995,
M.S. Computer Science, May 1998,
from Washington University in Saint Louis.

Book Chapters

Irfan Pyarali, Tim Harrison, Douglas C. Schmidt, and Thomas Jordan, "Proactor: An Object Behavioral Pattern for Demultiplexing and Dispatching Handlers for Asynchronous Events", Pattern Languages of Program Design 4 (PLoPD 4), (Harrison, Foote, and Rohnert, eds.), Addison-Wesley, Reading, MA, 1999.

Irfan Pyarali, Timothy H. Harrison, and Douglas C. Schmidt, "Asynchronous Completion Token: An Object Behavioral Pattern for Efficient Asynchronous Event Handling", Pattern Languages of Program Design 3 (PLoPD 3), (Martin, Buschmann, and Riehl, eds.), Addison-Wesley, Reading, MA, 1997.

Journal Publications

Carlos O'Ryan, Douglas C. Schmidt, Fred Kuhns, Marina Spivak, Jeff Parsons Irfan Pyarali, and David L. Levine, "The Design and Performance of a Real-time CORBA ORB Endsystem", The Journal of Concurrency: Practice and Experience (Special Issue on Distributed Objects and Applications), Wiley and Sons, 2000.

Irfan Pyarali, Carlos O'Ryan, Douglas C. Schmidt, Nanbor Wang, Vishal Kachroo, and Aniruddha Gokhale, "Applying Optimization Principle Patterns to Real-time ORBs", IEEE Concurrency,

Object-Oriented Systems Track, edited by Murthy Devarakonda, Volume 8, Number 1, January/March 2000.

James Hu, Irfan Pyarali, and Douglas C. Schmidt, "The Object-Oriented Design and Performance of JAWS: A High-performance Web Server Optimized for High-speed Networks", The Parallel and Distributed Computing Practices journal, special issue on Distributed Object-Oriented Systems, edited by Maria Cobb, to appear in 2000.

Irfan Pyarali and Douglas C. Schmidt, "An Overview of the CORBA Portable Object Adapter", Special Issue on CORBA in the ACM StandardView magazine, Volume 6, Number 1, March 1998.

Irfan Pyarali, Douglas C. Schmidt, and Tim Harrison, "Design and Performance of an Object-Oriented Framework for High-Speed Electronic Medical Imaging", USENIX Computing Systems, Volume 9, Number 4, November/December, 1996.

Conference Publications

David A. Karr, Craig Rodrigues, Joseph P. Loyall, Richard E. Schantz, Yamuna Krishnamurthy, Irfan Pyarali, and Douglas C. Schmidt, "Application of the QuO Quality-of-Service Framework to a Distributed Video Application", proceedings of the 3rd International Symposium on Distributed Objects and Applications (DOA '01) in Rome, Italy, 18-20 September, 2001.

Irfan Pyarali, Marina Spivak, Douglas C. Schmidt, and Ron Cytron, "Optimizing Thread-Pool Strategies for Real-Time CORBA", proceedings of the ACM SIGPLAN Workshop on Optimization of Middleware and Distributed Systems (OM '01) in Snowbird, Utah, June 18, 2001.

Irfan Pyarali, Carlos O'Ryan, and Douglas C. Schmidt, "Patterns for Efficient, Predictable, Scalable, and Flexible Dispatching Components", proceedings of the 7th Pattern Languages of Programs Conference (PLoP '00) in Allerton Park, Illinois, August 2000.

Douglas C. Schmidt, Carlos O'Ryan, Irfan Pyarali, Michael Kircher, and Frank Buschmann, Leader/Followers: "A Design Pattern for Efficient Multi-threaded Event Demultiplexing and Dispatching", proceedings of the 7th Pattern Languages of Programs Conference (PLoP '00) in Allerton Park, Illinois, August 2000.

Carlos O'Ryan, Douglas C. Schmidt, Fred Kuhns, Marina Spivak, Jeff Parsons, Irfan Pyarali and David L. Levine, "Evaluating Policies and Mechanisms for Supporting Embedded, Real-Time Applications with CORBA 3.0", proceedings of the 6th IEEE Real-Time Technology and Applications Symposium (RTAS '00) in Washington D.C., May 31-June 2, 2000.

Irfan Pyarali, Carlos O'Ryan, and Douglas C. Schmidt, "A Pattern Language for Efficient, Predictable, Scalable, and Flexible Dispatching Mechanisms for Distributed Object Computing Middleware", proceedings of the IEEE/IFIP International Symposium on Object-Oriented Real-time Distributed Computing (ISORC '00) in Newport Beach, California, March 15-17, 2000.

Irfan Pyarali, Carlos O'Ryan, Douglas C. Schmidt, Nanbor Wang, Vishal Kachroo, and Aniruddha Gokhale, "Applying Optimization Patterns to Design Real-time ORBs", proceedings of the 5th USENIX Conference on Object-Oriented Technologies and Systems (COOTS '99) in San Diego, CA, May 3-7, 1999.

James Hu, Irfan Pyarali, and Douglas C. Schmidt, "Applying the Proactor Pattern to High-Performance Web Servers", proceedings of the 10th International Conference on Parallel and Distributed Computing and Systems (IASTED '98) in Las Vegas, Nevada, October 28-31, 1998.

James Hu, Irfan Pyarali, and Douglas C. Schmidt, "Measuring the Impact of Event Dispatching and Concurrency Models on Web Server Performance Over High-speed Networks", proceedings of the 2nd Global Internet Conference (held as part of GLOBECOM '97) in Phoenix, AZ, November 4-8, 1997.

Irfan Pyarali, Tim Harrison, Douglas C. Schmidt, and Thomas Jordan, “Proactor: An Object Behavioral Pattern for Demultiplexing and Dispatching Handlers for Asynchronous Events”, proceedings of the 4th Pattern Languages of Programs Conference (PLoP ’97) in Allerton Park, Illinois, September 1997.

Irfan Pyarali, Timothy H. Harrison, and Douglas C. Schmidt, “Asynchronous Completion Token: An Object Behavioral Pattern for Efficient Asynchronous Event Handling”, proceedings of the 3rd Pattern Languages of Programs Conference (PLoP ’96) in Allerton Park, Illinois, September 4-6, 1996.

Irfan Pyarali, Tim Harrison, and Douglas. C. Schmidt, “Design and Performance of an Object-Oriented Framework for High-Speed Electronic Medical Imaging”, proceedings of the 2nd USENIX Conference on Object-Oriented Technologies and Systems (COOTS ’96) in Toronto, Canada, June 18-22, 1996.

Irfan Pyarali, Timothy H. Harrison, and Douglas C. Schmidt, “An Object-Oriented Framework for High-Performance Electronic Medical Imaging”, proceedings of the Very High Resolution and Quality Imaging mini-conference at the Symposium on Electronic Imaging in the International Symposia Photonics West 1996, SPIE in San Jose, California USA, January 27 - February 2, 1996.

Workshop Publications

Irfan Pyarali, Marina Spivak, and Ron Cytron, “Evaluating Thread Pool Strategies For Real-time CORBA”, OMG’s Second Workshop On Real-Time And Embedded Distributed Object Computing in Herndon, VA, June 4-7, 2001.

Craig Rodrigues, David Karr, Yamuna Krishnamurthy, and Irfan Pyarali, “QoS Control Of Video Streams Using Quality Objects And The CORBA Audio/Video Service”, OMG’s Second Workshop On Real-Time And Embedded Distributed Object Computing in Herndon, VA, June 4-7, 2001.

Trade-Journal Publications

Douglas C. Schmidt and Irfan Pyarali, “Strategies for Implementing POSIX Condition Variables on Win32, C++ Report”, SIGS, Volume 10, Number 5, June, 1998.