Evaluating and Optimizing Thread Pool Strategies for Real-Time CORBA

Irfan Pyarali, Marina Spivak, and Ron Cytron
{irfan,marina,cytron}@cs.wustl.edu
Department of Computer Science
Washington University, St. Louis, MO 63130, USA

Douglas C. Schmidt
schmidt@uci.edu
Electrical and Computer Engineering Dept.
University of California, Irvine, CA 92697, USA

Abstract

Strict control over the scheduling and execution of processor resources is essential for many fixed-priority real-time applications. To facilitate this common requirement, the Real-Time CORBA (RT-CORBA) specification defines standard middleware features that support end-to-end predictability for operations in such applications. One of the most important features in RT-CORBA is thread pools, which allow application developers and end-users to configure and control processor resources.

This paper provides two contributions to the evaluation of techniques for improving the quality of implementation of RT-CORBA thread pools. First, we describe the key patterns underlying common strategies for implementing RT-CORBA thread pools. Second, we evaluate each thread pool strategy in terms of its consequences on (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 memory allocations, and (5) bounded and unbounded priority inversion incurred in each implementation. This paper also provides results that illustrate empirically how different thread pool implementation strategies perform in different ORB configurations.

1 Introduction

The maturation of the CORBA specification [1] and standards-based CORBA implementations has simplified the development of distributed systems with complex *functional* requirements. However, next-generation distributed real-time and embedded (DRE) systems, such as command and control systems, manufacturing process control systems, videoconferencing, large-scale distributed interactive simulations, and testbeam data acquisition systems, have complex *quality of service* (QoS) requirements, such as stringent bandwidth, latency, jitter, and dependability needs. Historically, DRE systems were not well served by middleware like CORBA due to its lack of QoS support.

The recent Real-time CORBA (RT-CORBA) 1.0 specification [2, 3] in the CORBA 2.4 standard is an important step towards defining standards-based, commercial-off-the-shelf (COTS) middleware that can deliver end-to-end QoS support at multiple levels in DRE systems. As shown in Figure 1, RT-CORBA ORB endsystems[1] define standard capabilities that support end-to-end predictability for operations in *fixed-priority* CORBA applications. RT-CORBA features allow applications to configure and control the following resources:

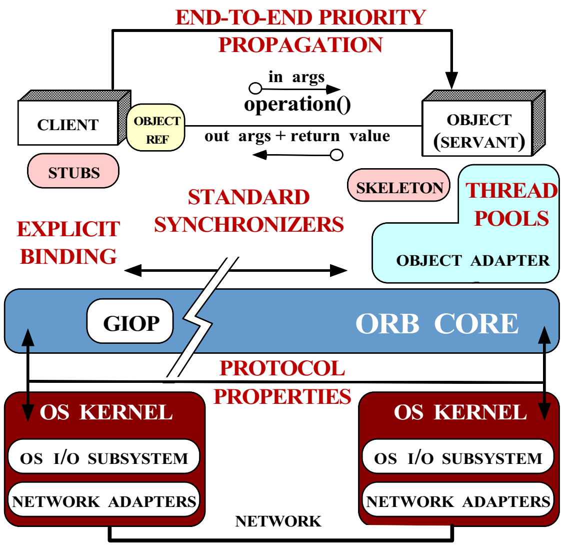

Figure 1: Standard Features in Real-Time CORBA ORB Endsystems

- *Processor resources* via thread pools, priority mechanisms, and intraprocess mutexes
- *Communication resources* via protocol properties and explicit bindings with non-multiplexed connections and

[1]An ORB endsystem consists of network interfaces, I/O subsystem and other OS mechanisms, and ORB middleware capabilities.

- *Memory resources* via buffering requests in queues and bounding the size of thread pools.

[3] presents an overview of the RT-CORBA features and [4] explains how communication resources are configured and controlled efficiently in TAO [5], which is our high-performance, real-time implementation of CORBA.

There are two general strategies for implementing RT-CORBA thread pools. The first strategy uses the *Half-Sync/Half-Async* pattern [6], 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 [6] to demultiplex I/O events into threads in a pool without requiring additional I/O threads. Each strategy is optimal 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 system may trade resource duplication to avoid any priority inversions by using the Leader/Followers pattern.

The remainder of this paper is organized as follows: Section 2 describes the key features in RT-CORBA thread pools; Section 3 illustrates how patterns can be applied to implement different RT-CORBA thread pool strategies; Section 4 provides empirical results that compare different thread pool implementation strategies; Section 5 compares our work on TAO's thread pools with related work; and Section 6 presents concluding remarks.

2 An Overview of RT-CORBA Thread Pools

Many real-time systems use multi-threading to

1. Distinguish between different types of service, such as high-priority vs. low-priority tasks [7]
2. Support thread preemption to prevent unbounded priority inversion and deadlock and
3. 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, the RT-CORBA specification defines a server *thread pool* model [8]. There are two types of thread pools in RT-CORBA:

- *Thread pool without lanes* – In this basic thread pool model all threads have the same assigned priority. This model is illustrated in Figure 2.

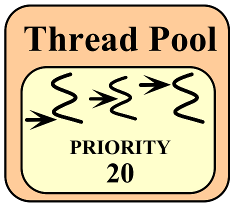

Figure 2: Thread Pool without Lanes

- *Thread pool with lanes* – In this more advanced model a pool consists of subsets of threads (called *lanes*) that are assigned different priorities. This model is illustrated in Figure 3.

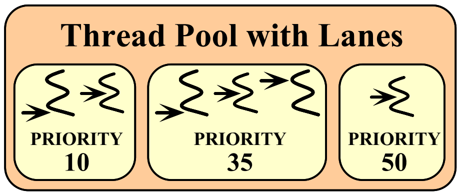

Figure 3: Thread Pool with Lanes

To create thread pools without and with lanes, developers of real-time applications can configure thread pools in an RT-CORBA server by using either the `create_threadpool` or `create_threadpool_with_lanes` methods, respectively, which are defined in the standard `RTORB` interface. Each thread pool is then associated with one or more POA via the `RTCORBA::ThreadPoolPolicy`. 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 4 illustrates the creation and association of thread pools in a server.

When created via the `create_threadpool*` methods outlined above, thread pools can be configured with the following properties:

- *Static threads*, which defines the number of pool threads pre-allocated at thread pool creation time.
- *Dynamic threads*, which 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 have 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, de-

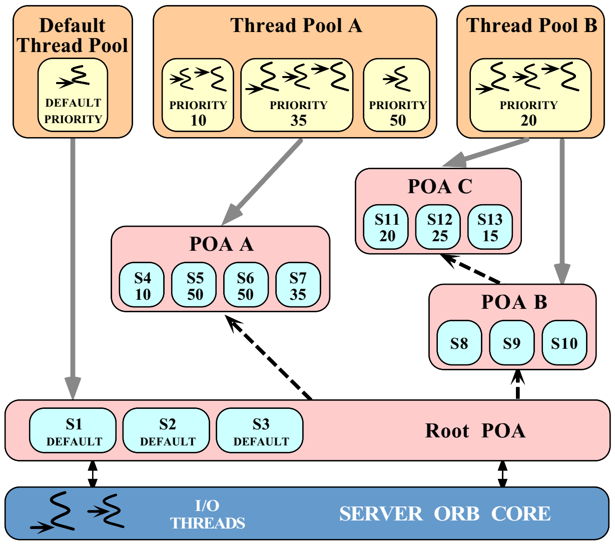

Figure 4: POA Thread Pools in Real-time CORBA

velopers can choose between dynamic and static threads to trade off (1) the jitter introduced by dynamic thread creation/destruction with (2) the wastefulness of under-utilized static threads.

- *Priority*, which defines the CORBA priority with which threads are created. There are two thread priority schemes used in RT-CORBA: native priority and CORBA priority. Native priority is the Real-Time Operating System (RTOS) specific thread priority representation. CORBA Priority, on the other hand, is a uniform representation used to overcome different RTOS specific thread priority representations. A priority mapping scheme is used to map between native and CORBA priorities and vice versa. The valid CORBA priority range is 0 to 32767.

  Depending on the *policies* configured in the ORB, this priority can be changed subsequently. Priority of threads in thread pools with lanes do not changes 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 4 illustrates this scenario. The 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 4 illustrates this scenario. The priority of the thread is restored after the client request has been processed.
- *Stack size*, which defines the bytes of stack size allocated for each thread.
- *Request buffering*, which 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, which indicates a temporary resource shortage. When a client receives this exception it can reissue the request at a later time. Figure 5 illustrates the thread pool request buffering feature.

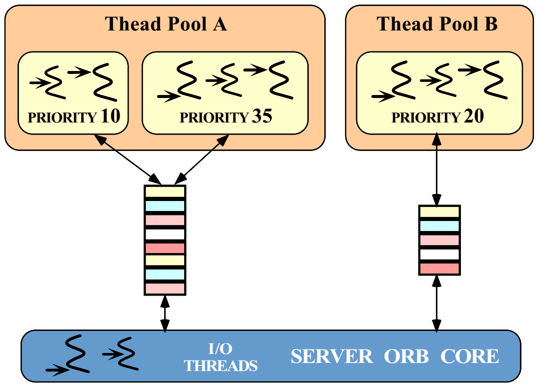

Figure 5: Buffering Requests in RT-CORBA Thread Pools

- *Thread borrowing*, which controls whether a lane with higher priority is allowed to "borrow" threads from a lane with lower priority when it exhausts its maximum number of threads (both static and dynamic) and requires an additional thread to service a new invocation. 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 once again to its previous value, and it is returned to the lower priority lane. Naturally, 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.

3 Alternative Patterns for Designing Optimal RT-CORBA Thread Pool Strategies

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 that 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. There are two general strategies for implementing RT-CORBA thread pools: *Half-Sync/Half-Async* and *Leader/Followers*. In this section, we use *patterns* to describe these two strategies in detail, outlining their structure, dynamics, implementation, and consequences for selecting optimal RT-CORBA thread pools for particular types of applications.[2] We focus on patterns in this paper to generalize the applicability of our work. Pattern descriptions help application developers and end-users understand the schedulability, scalability, and predictability consequences of a particular thread pool implementation used by their RT-CORBA ORB.

3.1 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.

3.1.1 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 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 [9] than synchronous threaded systems because the amount of information necessary to maintain program state can be reduced. In addition, asynchronous services can be mapped directly onto OS asynchrony mechanisms, such as WinNT I/O completion ports [10, 6]. 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.

[2]For completeness, this paper contains abbreviated descriptions of the Half-Sync/Half-Async and Leader/Followers patterns, focusing on the implementation of thread pools in RT-CORBA. A thorough discussion of these patterns appears in [6].

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 address 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.

3.1.2 Solution

An RT-CORBA ORB endsystem can be decomposed into two layers [11], synchronous and asynchronous; a queueing layer is introduced to mediate the communication between services in the asynchronous and synchronous layers.

3.1.3 Structure and Collaboration

The structure of the Half-Sync/Half-Async pattern is illustrated in Figure 6. This design follows the Layers pattern [11] 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. Demarshal the request
4. Perform 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.

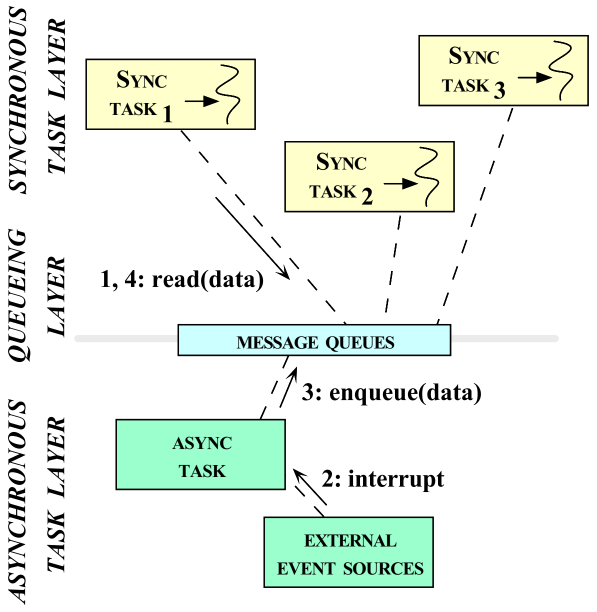

Figure 6: The Structure of Participants in the Half-Sync/Half-Async Pattern

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. Find 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 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 [11]. 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.

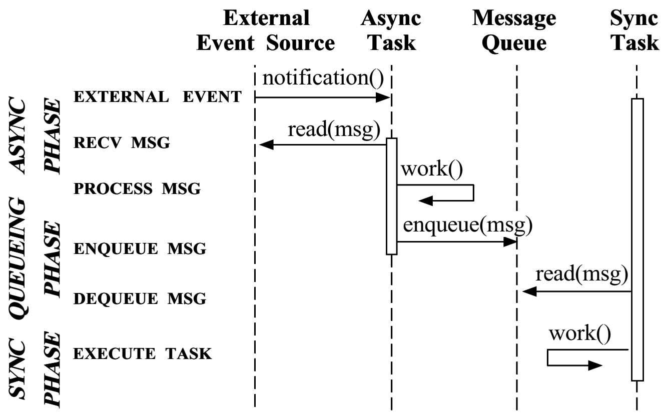

Figure 7: Collaboration between Layers in the Half-Sync/Half-Async Pattern

For an RT-CORBA server, common sources of external events include sensors, network interfaces, disk controllers, and end-user terminals.

Figure 7 illustrates these collaborations among participants in the Half-Sync/Half-Async pattern.

3.1.4 Implementation Synopsis

Figure 8 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 [6] is used to service 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 [6], 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.

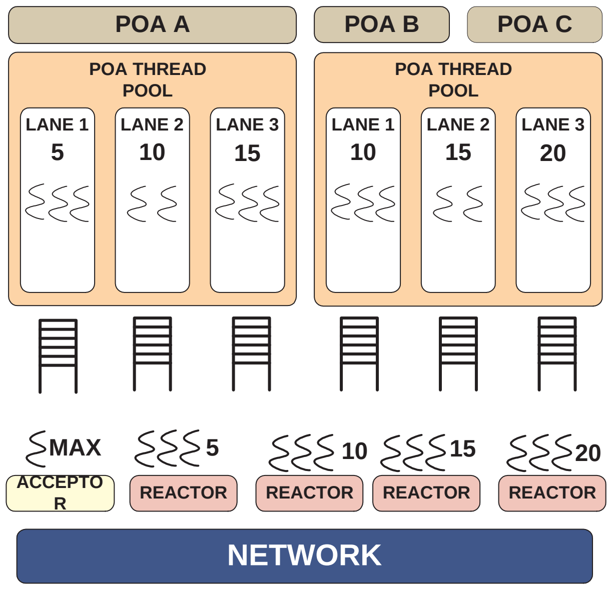

Figure 8: Implementing an RT-CORBA Thread Pool Using the Half-Sync/Half-Async Pattern

These threads block on a condition variable, waiting for requests to show up in their queue. After dequeueing the request, the target servant is found in the target POA, the request is demarshaled and application-level servant code is then executed.

3.1.5 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 phase. 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, 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 in a 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 [9].

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 and thread-specific storage (TSS) [6] cannot be used to optimize memory management for clients 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 1 summaries the evaluation for Half-Sync/Half-Async implementation of RT-CORBA thread pools.

Criteria	Evaluation
Feature Support	Good: supports request buffering and thread borrowing
Scalibility	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

Table 1: Evaluation of Half-Sync/Half-Async thread pools

3.2 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.

3.2.1 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 for 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 on 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.

3.2.2 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 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, which demultiplexes and dispatches the request to application-specific code in the processing thread. 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.

3.2.3 Structure and Collaboration

The key participants in the Leader/Followers pattern are shown in Figure 9 and are described below:

Handles and handle sets. Handles are 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. A handle set returns to its caller when a new request arrives from a client.

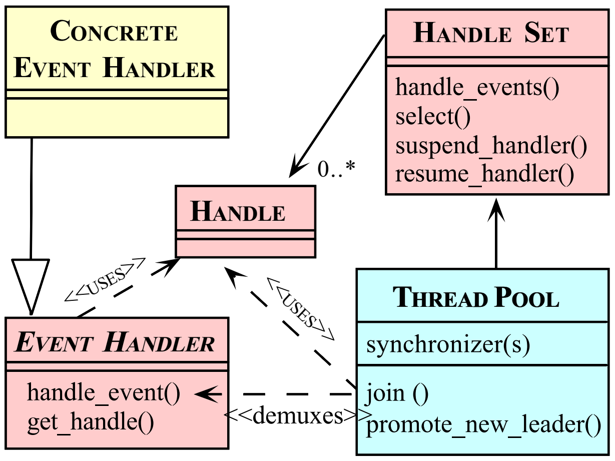

Figure 9: The Structure of Participants in the Leader/Followers Pattern

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
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 semaphore or condition variable, and implement a protocol for coordinating their transition between various roles. A thread's transitions between states is shown in Figure 10.

The collaborations in the Leader/Followers pattern are illustrated in Figure 11.

3.2.4 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 12, all client request processing (as described in Section 3.2.3) 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

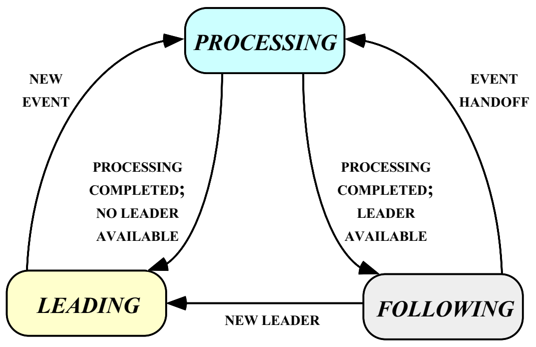
PROCESSING
NEW EVENT
EVENT HANDOFF
PROCESSING COMPLETED; NO LEADER AVAILABLE
PROCESSING COMPLETED; LEADER AVAILABLE
LEADING
NEW LEADER
FOLLOWING

Figure 10: A Thread's State Transitions in the Leader/Followers Pattern

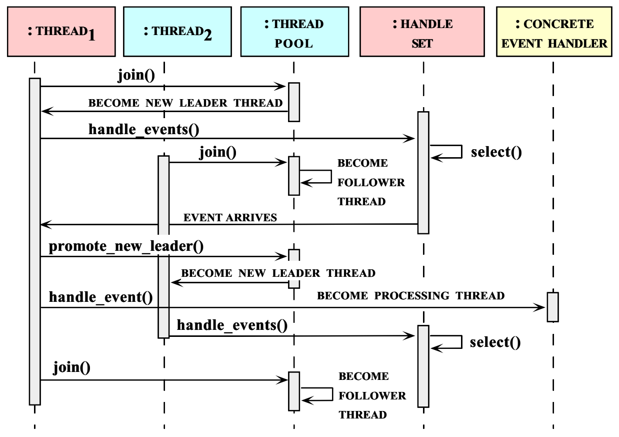

Figure 11: Collaboration Among Participants in the Leader/Followers Pattern

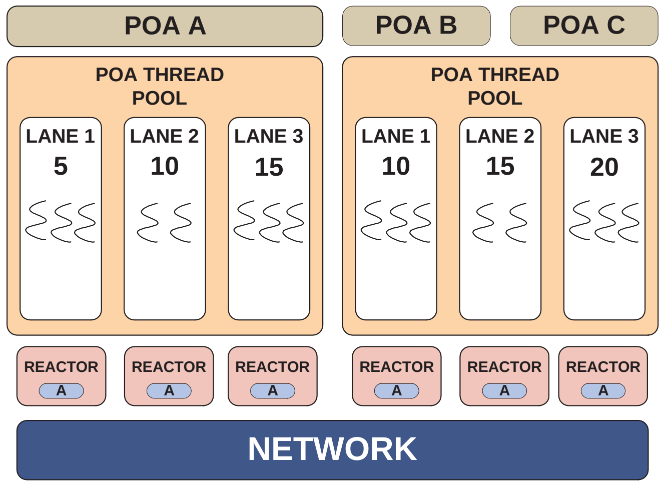

Figure 12: Implementing an RT-CORBA Thread Pool Using the Leader/Followers Pattern

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 application programmer will not have full control over all the thread resources.

3.2.5 Consequences

The Leader/Followers pattern provides several *benefits*:

Performance enhancements. Compared with the Half-Sync/Half-Async thread pool strategy described in Section 3.1, the Leader/Followers pattern can improve performance as follows:

- It enhances CPU 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 the thread-specific storage (TSS) [6] to allocate memory.
- It minimizes locking overhead by not exchanging data between threads, thereby reducing thread synchronization.
- It can minimize priority inversion because no extra queueing is introduced in the server. When combined with real-time I/O subsystems [12], the Leader/Followers thread pool implementation can reduce sources of non-determinism in server request processing significantly.
- 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 Half-Sync/Half-Async thread pool implementations.

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 [6].

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 2 summaries the evaluation for Leader/Followers implementation of RT-CORBA thread pools.

Criteria	Evaluation
Feature Support	Poor: not easy to support request buffering or thread borrowing
Scalibility	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 inversion

Table 2: Evaluation of Leader/Followers thread pools

4 Empirical results

Figure 13 compares the performance of the Half-Sync/Half-Async vs. the Leader/Followers thread pool implementations. These benchmarks were conducted using TAO version 1.0 on a quad-CPU 400 MHz Pentium II Xeon, with 1 GByte RAM, 512 Kb cache on each CPU, running Debian Linux release 2.2.5, and g++ version egcs-2.91.66. Our benchmarks

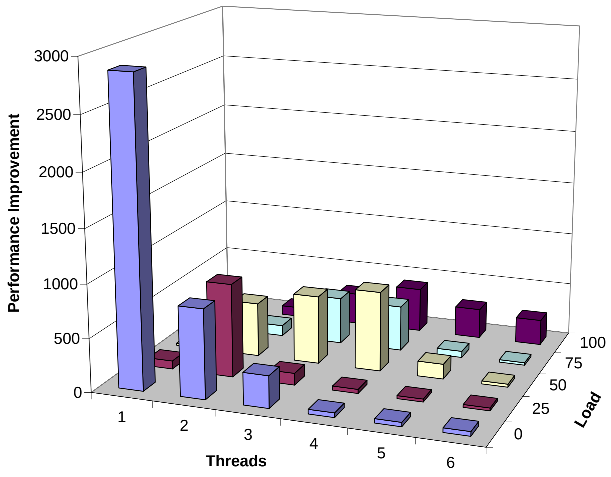

Figure 13: Performance of Half-Sync/Half-Async vs. the Leader/Followers thread pool implementations

measure the total time required by each concurrency strategy to process 100,000 CORBA request messages. We varied the number of threads and the amount of application-level processing performed for each request. The results in Figure 13 illustrate the percentage improvement in performance for the Leader/Followers thread pool strategy compared with the Half-Sync/Half-Async thread pool strategy.

As shown in the figure, the Leader/Followers strategy outperformed the Half-Sync/Half-Async approach for all combinations of threads and application workload. The largest improvement, ∼2,800%, occurred for a small number of threads and a small amount of work-per-request. As the number of threads and the amount of work-per-request increased the percentage improvement decreased to ∼8%. These results illustrate that the Half-Sync/Half-Async thread pool strategy incurs a higher amount of overhead for memory allocation, locking, and data movement than the Leader/Followers strategy.

Note that on a lightly loaded real-time system, using a small number of threads will generally yield better throughput than a higher number of threads. This difference stems from the higher context switching and locking overhead incurred by threading. As workloads increase, however, addition threads may help improve server throughput, particularly when the server runs on a multi-processor.

5 Related Work

Real-time middleware is an emerging field of study. An increasing number of research efforts are focusing on integrating QoS and real-time scheduling into distribution middleware, such as CORBA. Our previous work on TAO has examined many dimensions of ORB middleware design, including static [5] and dynamic [13] operation scheduling, event processing [7], I/O subsystem [12] and pluggable protocol [14] integration, synchronous [8] and asynchronous [15] ORB Core architectures, IDL compiler features [16] and optimizations [17], systematic benchmarking of multiple ORBs [18], patterns for ORB extensibility [6] and ORB performance [19]. 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) [20]. The system supports expression and enforcement of dynamic end-to-end timing constraints through timed distributed operation invocations (`TDMIs`) [21]. A difference between TAO and the URI approaches is that `TDMIs` express required 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 is developed at BBN Technologies [22]. 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 that allows the QuO run-time to adaptively trigger reconfiguration 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) [23]. The QuO middleware adds significant value to adaptive real-time ORBs such as TAO. We are currently collaborating [24] with the BBN QuO team to integrate the TAO and QuO middleware as part of the DARPA Quorum project [25].

UCI TMO. The Time-triggered Message-triggered Objects (TMO) project [26] 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 [7]. Where the TMO model creates new ORB services to provide its time-based invocation capabilities [27], TAO provides a subset of these capabilities by extending the standard CORBA COS 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.

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 paper describes the Half-Sync/Half-Async and the Leader/Followers strategies for implementing RT-CORBA thread pools. We evaluate these strategies using several different factors and present 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 used by their RT-CORBA ORB.

All the source code, documentation, examples, and tests for TAO and its RT-CORBA mechanisms are open-source and can be downloaded from `www.cs.wustl.edu/~schmidt/TAO.html`.

References

[1] Object Management Group, *The Common Object Request Broker: Architecture and Specification*, 2.4 ed., Oct. 2000.

[2] Object Management Group, *Realtime CORBA Joint Revised Submission*, OMG Document orbos/99-02-12 ed., March 1999.

[3] D. C. Schmidt and F. Kuhns, "An Overview of the Real-time CORBA Specification," *IEEE Computer Magazine, Special Issue on Object-oriented Real-time Computing*, June 2000.

[4] C. O'Ryan, D. C. Schmidt, F. Kuhns, M. Spivak, J. Parsons, I. Pyarali, and D. Levine, "Evaluating Policies and Mechanisms for Supporting Embedded, Real-Time Applications with CORBA 3.0," in *Proceedings of the* 6^{th} *IEEE Real-Time Technology and Applications Symposium*, (Washington DC), IEEE, May 2000.

[5] D. C. Schmidt, D. L. Levine, and S. Mungee, "The Design and Performance of Real-Time Object Request Brokers," *Computer Communications*, vol. 21, pp. 294–324, Apr. 1998.

[6] D. C. Schmidt, M. Stal, H. Rohnert, and F. Buschmann, *Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects, Volume 2*. New York, NY: Wiley & Sons, 2000.

[7] T. H. Harrison, D. L. Levine, and D. C. Schmidt, "The Design and Performance of a Real-time CORBA Event Service," in *Proceedings of OOPSLA '97*, (Atlanta, GA), ACM, October 1997.

[8] D. C. Schmidt, S. Mungee, S. Flores-Gaitan, and A. 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*, vol. 21, no. 2, 2001.

[9] D. C. Schmidt and T. Suda, "Measuring the Performance of Parallel Message-based Process Architectures," in *Proceedings of the Conference on Computer Communications (INFOCOM)*, (Boston, MA), pp. 624–633, IEEE, April 1995.

[10] J. Richter, *Advanced Windows, Third Edition*. Redmond, WA: Microsoft Press, 1997.

[11] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal, *Pattern-Oriented Software Architecture – A System of Patterns*. Wiley and Sons, 1996.

[12] F. Kuhns, D. C. Schmidt, C. O'Ryan, and D. Levine, "Supporting High-performance I/O in QoS-enabled ORB Middleware," *Cluster Computing: the Journal on Networks, Software, and Applications*, vol. 3, no. 3, 2000.

[13] C. D. Gill, D. L. Levine, and D. 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*, vol. 20, March 2001.

[14] C. O'Ryan, F. Kuhns, D. C. Schmidt, O. Othman, and J. 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, Apr. 2000.

[15] A. B. Arulanthu, C. O'Ryan, D. C. Schmidt, M. Kircher, and J. Parsons, "The Design and Performance of a Scalable ORB Architecture for CORBA Asynchronous Messaging," in *Proceedings of the Middleware 2000 Conference*, ACM/IFIP, Apr. 2000.

[16] A. B. Arulanthu, C. O'Ryan, D. C. Schmidt, and M. Kircher, "Applying C++, Patterns, and Components to Develop an IDL Compiler for CORBA AMI Callbacks," *C++ Report*, vol. 12, Mar. 2000.

[17] A. Gokhale and D. 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*, vol. 17, Sept. 1999.

[18] A. Gokhale and D. C. Schmidt, "Measuring the Performance of Communication Middleware on High-Speed Networks," in *Proceedings of SIGCOMM '96*, (Stanford, CA), pp. 306–317, ACM, August 1996.

[19] I. Pyarali, C. O'Ryan, D. C. Schmidt, N. Wang, V. Kachroo, and A. Gokhale, "Using Principle Patterns to Optimize Real-time ORBs," *Concurrency Magazine*, vol. 8, no. 1, 2000.

[20] V. F. Wolfe, L. C. DiPippo, R. Ginis, M. Squadrito, S. Wohlever, I. Zykh, and R. Johnston, "Real-Time CORBA," in *Proceedings of the Third IEEE Real-Time Technology and Applications Symposium*, (Montréal, Canada), June 1997.

[21] V. Fay-Wolfe, J. K. Black, B. Thuraisingham, and P. Krupp, "Real-time Method Invocations in Distributed Environments," Tech. Rep. 95-244, University of Rhode Island, Department of Computer Science and Statistics, 1995.

[22] J. A. Zinky, D. E. Bakken, and R. Schantz, "Architectural Support for Quality of Service for CORBA Objects," *Theory and Practice of Object Systems*, vol. 3, no. 1, 1997.

[23] G. Kiczales, "Aspect-Oriented Programming," in *Proceedings of the 11th European Conference on Object-Oriented Programming*, June 1997.

[24] 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)*, IEEE, April 2001.

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

[26] K. H. K. Kim, "Object Structures for Real-Time Systems and Simulators," *IEEE Computer*, pp. 62–70, Aug. 1997.

[27] K. Kim and E. Shokri, "Two CORBA Services Enabling TMO Network Programming," in *Fourth International Workshop on Object-Oriented, Real-Time Dependable Systems*, IEEE, January 1999.