Optimizing the ORB Core to Enhance Real-time CORBA Predictability and Performance

Arvind S. Krishna, Douglas C. Schmidt
{arvindk,schmidt}@dre.vanderbilt.edu
Electrical Engineering & Computer Science
Vanderbilt University

Krishna Raman, Raymond Klefstad
{kraman,klefstad}@uci.edu
Electrical Engineering & Computer Science
University of California, Irvine

This paper has been submitted to the 5th International Symposium on Distributed Objects and Application, Sicily, November 3-7, 2003

Abstract

Distributed real-time and embedded (DRE) applications possess stringent quality of service (QoS) requirements, such as low latency, bounded jitter, and high throughput. An increasing number of DRE applications are developed using QoS-enabled middleware, such as Real-time CORBA and the Real-time Specification for Java (RTSJ), to ensure predictable end-to-end QoS. Real-time CORBA is an open middleware standard that allows DRE applications to allocate, schedule, and control the QoS of CPU, memory, and networking resources. The RTSJ provides extensions to Java that enable it to be used as the basis for Real-time CORBA middleware and applications.

This paper provides the following contributions to the study of QoS-enabled middleware for DRE applications. First, we outline key Real-time CORBA implementation challenges within the ORB Core, focusing on efficient buffer allocation and collocation strategies. Second, we describe how these challenges have been addressed in ZEN, which is an implementation of Real-time CORBA that runs atop RTSJ platforms. Third, we describe how RTSJ features, such as scoped memory and no-heap real-time threads, can be applied in a real-time ORB Core to enhance the predictability of DRE applications using Real-time CORBA and the RTSJ. Our results show that carefully applied optimization strategies can enable RTSJ-based Real-time CORBA ORBs to achieve effective QoS support for a range of DRE applications.

1 Introduction

Motivation. Over the past decade, distributed computing middleware, such as CORBA [1], COM+ [2], Java RMI [3], and SOAP/.NET [4], have emerged to reduce the complexity of distributed systems. This type of middleware simplifies the development of distributed systems by off-loading the tedious and error-prone aspects of distributed computing from application developers to middleware developers. Distributed computing middleware has been used successfully in desktop and enterprise systems [5, 6] where scalability, evolvability, and interoperability are essential for success. In this context, middleware offers several benefits: hardware-, language-, and OS-independence, as well as open-source availability.

The benefits of middleware are also desirable for distributed real-time and embedded (DRE) systems [7]. Examples of DRE systems include telecommunication networks (*e.g.*, wireless phone services), tele-medicine (*e.g.*, robotic surgery), process automation (*e.g.*, hot rolling mills), and defense applications (*e.g.*, total ship computing environments). DRE systems possess stringent quality of service (QoS) constraints, such as bandwidth, latency, jitter, and dependability requirements.

The Real-time CORBA specification [8] was standardized by the OMG to support the QoS needs of a certain class DRE systems, *i.e.*, those that rely on fixed-priority scheduling. Real-time CORBA leverages features from the CORBA standard (such as the GIOP protocol) and the Messaging specification [9] (such as the QoS policy framework) to add QoS control capabilities to regular CORBA. These QoS capabilities help to improve DRE application predictability by bounding priority inversions and managing system resources end-to-end. Specifically, Real-time CORBA provides standard features that allow DRE applications to configure and control the following system resources:

- **Processor resources** via thread pools, priority mechanisms, intra-process mutexes, and a global scheduling service for real-time applications with fixed priorities,
- **Communication resources** via protocol properties and explicit bindings to server objects using priority bands and private connections, and
- **Memory resources** via buffering requests in queues and bounding the size of thread pools.

Optimizing Real-time CORBA Object Request Brokers. Underneath any CORBA middleware application is an Object

Request Broker (ORB), which allows clients to invoke operations on distributed objects without concern for object location, programming language, OS platform, communication protocols and interconnects, and hardware [10]. Figure 1 illustrates the key elements in the CORBA reference model [11] that collaborate to provide this degree of portability, interoperability, and transparency.[1] The heart of the CORBA reference model is the *ORB Core*, which is the element in standard CORBA that handles connection and memory management, data transfer, endpoint demultiplexing, and concurrency control for client and server applications [10]. When a client invokes an operation on a target object, the ORB Core delivers the request to the object and returns the response (if the operation is has two-way semantics).

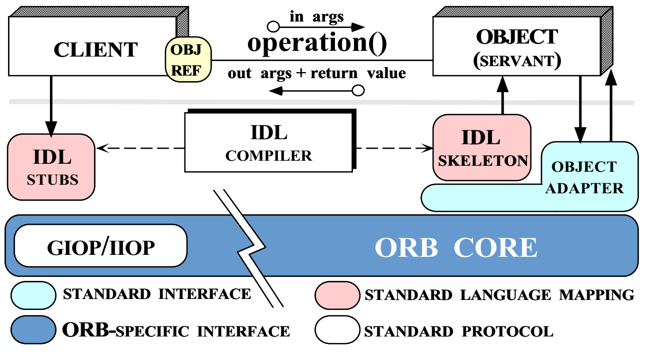

Figure 1: Key Elements in the CORBA Reference Model

Our prior work on CORBA has explored many dimensions of ORB design and performance, including scalable event processing; request demultiplexing; I/O subsystem and protocol integration; connection management, explicit binding, and real-time threading [12] architectures, asynchronous and synchronous concurrent request processing, and IDL stub/skeleton optimizations. This paper explores a previous unexamined dimension of ORB design: *optimizing the ORB Core to support real-time applications by increasing the predictability, performance, and scalability of ORBs developed using the Real-Time Specification for Java (RTSJ) [13].*

The vehicle used to showcase these optimizations is ZEN [14]. ZEN is an open-source[2] Real-time CORBA ORB designed using the micro-kernel architectural pattern [15] and implemented using Real-time Java. The design of ZEN was inspired by many of the patterns, techniques, and lessons learned when developing The ACE ORB (TAO). [16]. TAO is an open-source[3] Real-time CORBA ORB implemented using C++, with enhancements designed to ensure efficient, predictable, and scalable QoS behavior for high-performance and real-time applications.

TAO and ZEN are a rapidly maturing Real-time CORBA ORBs designed for applications with hard real-time requirements, such as avionics mission computing [16], as well as those with softer real-time requirements, such as telecommunication call processing and streaming video [17]. When combined with quality real-time operating systems [18], TAO and ZEN can meet both the QoS needs of DRE applications and the development benefits offered by middleware. Figure 2 illustrates ZEN's pluggable micro-kernel ORB architecture. Unlike monolithic ORBs, a micro-kernel ORB like ZEN factors out services whose behavior can vary based on (1) a user's choice for certain behavior and (2) the use of certain standard CORBA capabilities, such as object adapters, protocol transports, etc. In a micro-kernel ORB, these capabilities are moved out of the ORB "kernel," thereby reducing the footprint of the middleware and also increasing design flexibility.

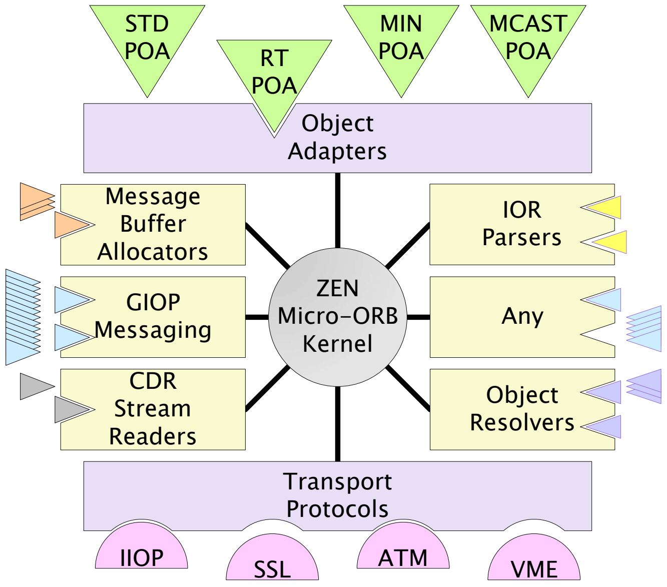

Figure 2: ZEN's Pluggable Micro-kernel ORB Architecture

Although the Real-time CORBA specification was integrated into the OMG standard in 1998 [19], it has not been adopted universally by DRE application developers. A key barrier to adoption arises from the steep learning curve caused by the complexity of the CORBA-C++ mapping [20, 21, 22]. To address this problem, the Java programming language has emerged as an attractive alternative. Since Java has less accidental complexity than C++, it is easier for application programmers to master it. Java also has other desirable language features, such as strong typing, dynamic class loading, reflection/introspection, and native support for concurrency and syn-

[1]This overview only focuses on the CORBA elements relevant to this paper. For a complete synopsis of CORBA's elements see [11].

[2]ZEN can be downloaded from `www.zen.uci.edu`.

[3]TAO can be downloaded from `deuce.doc.wustl.edu/Download.html`.

chronization.

Conventional Java runtime systems and middleware have historically been unsuitable for DRE applications, however, due to

1. The under-specified scheduling semantics of Java threads, which can lead to the most eligible thread not always being run.
2. The ability of the Java Garbage Collector (GC) to preempt any other Java thread, which can yield very long preemption latencies.

To address the above problems, the Real-time Java Experts Group has defined the RTSJ [13], which extends Java in several ways, including (1) new memory management models that allow access to physical memory and can be used in lieu of garbage collection and (2) stronger guarantees on thread semantics than in conventional Java. To have a predictable Java-based Real-time CORBA ORB, it is necessary to (1) apply optimizations to the ORB Core to ensure predictability and (2) apply RTSJ features effectively within the ORB Core.

Paper organization. To address the challenges outlined above, the remainder of this paper is organized as follows: Section 2 presents key challenges within the ORB Core layer, focusing on buffer management and object location techniques, and explains how these challenges have been addressed in ZEN; Section 3 describes the main problems that arose while designing ZEN using conventional Java implementations, analyzes the critical request/response code path within ZEN to identify sources for the application of RTSJ features, and illustrates how RTSJ features can be associated with key ORB components to enhance predictability; Section 4 summarizes how our work on ZEN relates to other research efforts; and Section 5 presents our concluding remarks and outlines our future work.

2 Optimizations Applied to ZEN's ORB Core

The ORB Core is the layer of a CORBA ORB implementation that is responsible for connection and memory management, data transfer, endpoint demultiplexing, and concurrency control. An ORB Core is also the minimal run-time layer associated with both a typical client and server. When a client invokes an operation on an object, the ORB Core is responsible for delivering the request to the server and returning the response, if any, to the client. For remote objects, the ORB Core, transfers requests using the General Internet Inter-ORB Protocol (GIOP) that runs atop many transport protocols, including TCP/IP.

Optimizing the ORB Core to support DRE applications poses several challenges to ORB implementors. This section outlines some of the key challenges present in this layer and describe the optimizations we have applied that ensure the predictability and efficiency required by DRE applications. These optimizations include minimizing memory management operations using efficient buffer management algorithms and transparently collocating clients and servants that are present in the same address space.

2.1 Buffer Management Optimizations

Context. The ORB Core uses memory buffers for storing GIOP messages both before sending and after receiving across a transport. These buffers are serially reusable, which means that only one thread can use a given buffer at any time, though after its completion the same buffer may be reused by another thread.

Problem. A naive Java implementation for ORB buffers would use operator `new` to allocate each one, thereby allowing the Java garbage collector to reclaim them for later use. Continued allocation/deallocation of these buffers would eventually lead to an invocation of the garbage collector, which is undesirable in DRE applications since it may incur unbounded jitter [23].

Solution $\rightarrow$ Buffer management optimizations. One solution to the problem of unbounded jitter is to pre-allocate buffer pools at ORB initialization time. A simple pool manager can allocate from the pool and return unneeded buffers to the pool after they are no longer needed. Application developers and/or system integrators can be given configuration control over buffer size and pool size. We can provide alternative dynamic storage management algorithms, such as first fit, random fit [24], and best fit [25].

Applying the solution in ZEN. ZEN supports the following buffer management strategies:

- **Linked List** strategy, in which a simple list is used to maintain all allocated buffers. The first fit algorithm is used to locate the most appropriate buffer. This strategy is suitable when buffer sizes are comparable. When buffer sizes vary, however, the search time considerably degrades as the list is not ordered. For example, the worst case behavior for this strategy is $O(n)$ when all buffers in the list are smaller than the required size.
- **Multi-level buckets** strategy in which, the buffers are divided into partitions *i.e.*, buckets based using a partition strategy that is typically a factor of the block size. To locate a buffer of given size, the most appropriate bucket is first determined, then the first fit strategy is used to return the most appropriate buffer. A default bucket is used for significantly large/small buffer sizes. This strategy is an improvement over the linked list scheme and has constant time lookup time for the non-default case. The default bucket has a behavior similar to the linked list scheme.

- **Buffer Pools** strategy in which the ORB maintains a pre-allocated pool of buffers of a fixed size. These individual buffers may be chained to hold larger messages and are written using gather-write I/O system calls. This strategy has a constant buffer lookup time, but incurs the overhead of managing multiple buffers. Moreover, earlier versions of Java (*i.e.*, up to JDK 1.3) did not provide gather-write facility requiring multiple I/O calls to read/write the buffer to the stream. ZEN provides this facility via Java's new I/O (`nio`) [26] package.

In ZEN, the abstract `ByteBufferManager` class manages the various buffer management schemes. The read/write helper methods defined on the buffer manager are used to marshal/demarshal GIOP messages, which are represented in ZEN as instances of `CDRInputStream` and `CDROutputStream` classes. ZEN provides two buffer manager implementations, shown in Figure 3 and explained below:

- The `VectoredByteBufferManager` implements the Buffer Pool optimization strategy explained above. This manager can only be supported with a JDK version 1.4 or later that provides gather-write I/O system calls.
- The `NonVectoredByteBufferManager` implements the multi-level bucket buffer management strategy.

The application developer can chooses either one of the buffer managers by setting the `zen.cdr.bufferManagerStrategy` property in the properties file. ZEN uses the Strategy pattern [27] to transparently plug in concrete buffer managers implementations. Moreover, this pluggable approach enables other buffer manager implementations to be provided, as long as they inherit from the `BufferManager` base class.

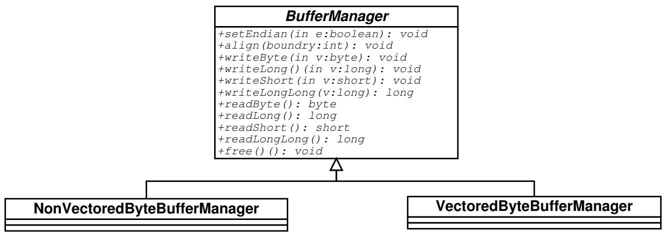
BufferManager
+setEndian(in e:boolean): void
+align(boundry:int): void
+writeByte(in v:byte): void
+writeLong()(in v:long): void
+writeShort(in v:short): void
+writeLongLong(v:long): long
+readByte(): byte
+readLong(): long
+readShort(): short
+readLongLong(): long
+free()(): void
NonVectoredByteBufferManager
VectoredByteBufferManager

Figure 3: ZEN Buffer Manager Class Diagram

Each of ZEN's concrete buffer manager classes are associated with a buffer allocator that controls how buffers are allocated and deallocated. The `ByteBufferAllocator` class shown in Figure 4 is the base class for all concrete buffer allocators. ZEN, provides the following concrete allocation schemes:

- `DynamicByteBufferAllocator`, where buffers are allocated/deallocated for each GIOP message sent/received.
- `CachedByteBufferAllocator`, where all the allocated buffers are cached and new buffers are created only if necessary. The `CacheNIOByteBufferAllocator` class deals with caching buffers in the Java `nio` package.
- `DynamicNIOByteBufferAllocator` deals with nio buffers, but buffers are allocated for each request.

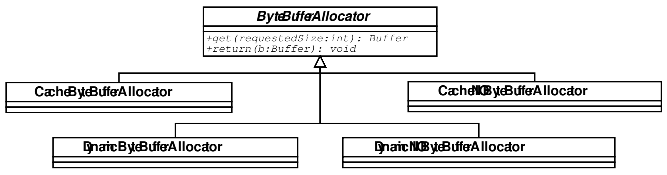
ByteBufferAllocator
+get(requestedSize:int): Buffer
+return(b:Buffer): void
CacheByteBufferAllocator
CacheNIOByteBufferAllocator
DynamicByteBufferAllocator
DynamicNIOByteBufferAllocator

Figure 4: ZEN Buffer Allocators Class Diagram

In ZEN, buffer allocators can be configured by setting the `zen.cdr.bufferAllocationStrategy` in the properties file. Similar to buffer managers, concrete buffer allocators implementations are plugged in using the Strategy pattern.

DRE applications often send small messages, whereas enterprise application may need to send larger messages. It is therefore important to configure the minimum block size (unit of allocation) to minimize fragmentation [28]. ZEN allows the end-user to configure the block size using the `zen.giop.messageBlock` property defined in the properties file. This variable should be set with a value corresponding to the message sizes that the system expects. The default value for this property is set to 1,024 bytes in ZEN. Combining buffer manager and allocators yields the different alternatives summarized in Table 1. As shown in the table, only certain combinations are possible.

	Vectored	**NonVectored**
cached	non-compatible	compatible
cached-nio	compatible	non-compatible
dynamic	non-compatible	compatible
dynamic-nio	compatible	non-compatible

Table 1: ZEN's Buffer Management Summary

Empirical results. We compared the performance of ZEN's buffer management optimizations. The following experiments were conducted:

- **Garbage collection analysis** compared the reduction in the number of garbage collection sweep by using caching with dynamic allocations. Our motivation was to observe if the buffer optimizations reduce garbage collection, and in turn increase predictability.
- **Throughput analysis** compared difference in throughput between `Vectored` and `NonVectored` buffer management strategies. Our motivation was to compare Buffer Pool and Multi-level bucket schemes in ZEN.

Since the RTSJ does not yet support java's nio package, these experiments were conducted using JDK 1.4.2. The testbed used was an Intel Pentium IV 1800 Mhz processor with 512 MB of main memory running Linux OS 2.4.21-0.11. We also used ZEN version 0.8 for these experiments.

- **Garbage collection analysis.** In this experiment the message sizes of the GIOP request sent by the client were increased by a factor of 2 starting with 1KB and continuing to 16KB. In each case, the number of GC executions at the server was measured for a total of 10,000 iterations. Figure 5 shows that buffer caching significantly reduces the number of GC sweeps. For both cases, GC executions increased with increasing buffer sizes. However, with caching the increase is gradual, whereas the increase for dynamic allocation is sharper. For example, in the case when message size is 16KB, the number of GC sweeps for dynamic case is greater by a factor ∼3, whereas for 1K it is greater only by a factor of ∼1.5. These results show that use of ZEN's buffer management algorithms reduce GC executions significantly.

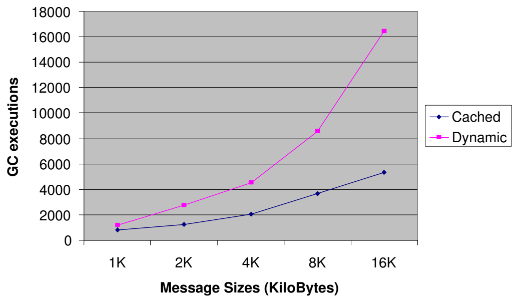

Figure 5: Cached v/s Dynamic GC analysis

- **Throughput analysis.** For this experiment throughput was defined as the number of events processed/sec at the server. In this experiment the message sizes of the GIOP request sent by the client was increased by a factor of 2, starting with 1KB and continuing to 16KB. In each case, throughput at the server was measured for a total of 10,000 iterations.

  As shown in Figure 6, for both strategies throughput decreases with increase in buffer size. The Vectored strategy, however, incurred greater overhead than the Non-Vectored strategy. This result was not expected since (1) the vectored strategy does not incur any data copying overhead as buffers are chained and (2) the Non-Vectored strategy incurs significant resizing overhead leading to greater data copying. Researchers [29] at University of Maryland also observed decreased throughput when using java's `nio` package. Their experiments showed that

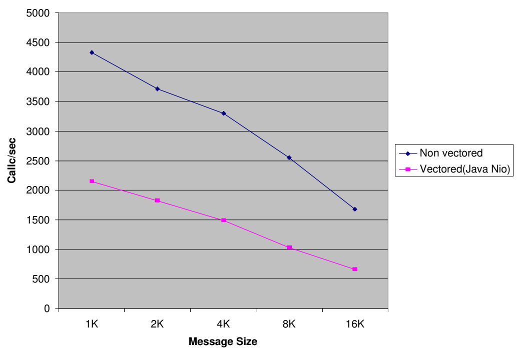

Figure 6: Vectored v/s Non-Vectored Throughput Analysis

  to speed up performance, nio's `ByteBuffers` should be converted into normal `byte[]` arrays for processing and should be used while writing to the network. We plan to implement this optimization in ZEN shortly.

2.2 Collocation Optimizations

Context. In addition to separating interfaces from implementations, a key strength of CORBA is its decoupling of servant implementations from how the servants are configured into server processes. CORBA is often used to communicate between remote objects. There are configurations, however, where a client and servant must be collocated in the same address space [30].

Problem When the client and server are collocated in the same address space, there should be no overhead from marshaling and demarshaling data or transmitting requests and replies over a "loop back" transport device. A naive implementation of CORBA would incur these overheads, thereby reducing performance and increasing jitter.

Solution → Collocation Optimization Clients can obtain an object reference in several ways, *e.g.*, from a CORBA Naming Service or Trading Service. Likewise, clients can use the `string_to_object` operation to convert a stringified Interoperable Object Reference (IOR) into an object reference. To ensure locality transparency, an ORB's collocation strategy must determine if the object is collocated within the same ORB and/or process. If so, it should optimize the request processing strategy by not incurring the overheads mentioned above.

Applying the solution in ZEN. ZEN supports the following two levels of collocation optimizations:

1. **Per-process collocation**, where the client and server ORBs are present in the same address space.
2. **Per-ORB collocation**, where the client and server ORBs are the same. This scheme is more fine-grained than per-process collocation, as the information relating to the tar-

get POA servant is directly available to the client. Moreover, this scheme also localizes the side-effects [30] of collocation, such as priority inversions [31] within a single ORB.

For these two levels of collocation, the following two strategies can be applied:

1. **Standard collocation** where the ZEN `Thru_POA` collocation strategy uses "collocation-safe stubs." As indicated by this strategy's name, all invocations go through the POA, *i.e.*, the steps for processing the request are the same as that of a remote request. This strategy ensures that all standard POA services (such as `POA_Current`) and various locks within the ORB Core and the POA are honored. `Thru_POA` is the default collocation strategy in ZEN.
2. **Direct collocation** where the collocation strategy forwards all requests directly to the servant, thereby bypassing the POA. Since the `Thru_POA` strategy adheres to all CORBA semantics for request processing, it incurs a considerable amount of overhead that may not be acceptable for DRE applications. In contrast, the `direct` strategy directly delivers a request to the servant, thereby avoiding marshaling overhead and context setup overhead (initializing current services in the POA and the ORB). This extension is not compliant with the CORBA specification, however, and is provided as an extension for DRE applications having stringent latency requirements.

There are four different collocation alternatives supported by ZEN, as shown in the Table 2. Irrespective of the combina-

	Thru_POA	**Direct**
Per-Process	collocation safe	suitable for DRE systems collocation unsafe
Per-ORB	collocation safe	suitable for DRE systems

Table 2: ZEN's Collocation Alternatives

tion used, the following three steps are involved in processing collocated requests:

1. Determining collocation. To determine if an object reference is collocated, ZEN's ORB Core maintains a *collocation table* that maps ORB endpoints to ORB object references. For IIOP, the endpoints are specified using `hostname` and `port number` tuples.

Multiple ORBs can reside in the same server process and each of these ORBs may support multiple transport endpoints. Rather than having one table per protocol, all endpoint structures in ZEN inherit from the `Address` class. By overriding the `hashCode()` and `equals()` methods for each type of endpoint, a single table can maintain information about all ORBs and their respective endpoints. Figure 7 shows the internal table structure of the collocation table managed in ZEN.

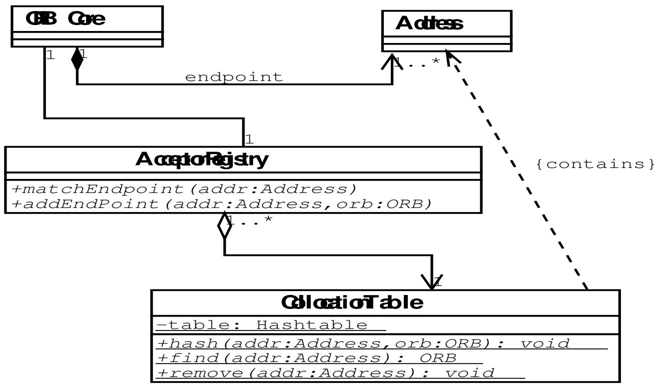
ORB Core
Address
1
1
endpoint
1..*
{contains}
1
AcceptorRegistry
+matchEndpoint(addr:Address)
+addEndPoint(addr:Address,orb:ORB)
1..*
1
CollocationTable
-table: Hashtable
+hash(addr:Address,orb:ORB): void
+find(addr:Address): ORB
+remove(addr:Address): void

Figure 7: ZEN's Collocation Tables: Static Structure

2. Resolving locality. Figure 8 shows how ZEN determines if an object is collocated. The client applica-

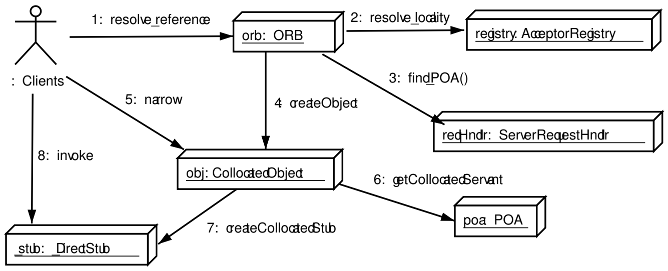

Figure 8: Finding a Collocated Object in ZEN

tion uses the ORB to resolve the reference obtained ((**1**)). The ORB consults its registry and resolves locality based on the level of locality configured in the client ORB ((**2**)). If local, the collocated POA is determined using the `ServerRequestHandler` ((**3**)). The ORB then creates a special collocated CORBA object ((**4**)). The client application narrows this generic object ((**5**)), which obtains the collocated servant from the POA ((**6**)). If a servant is found, a special `DirectStub` is created for servant-based operations **(7,8)**, otherwise the appropriate exception is raised.

3. Performing object invocations. ZEN has two strategies for performing object invocations after it resolves locality. These two schemes – `Thru_POA` and `Direct` collocation optimization – are discussed below:

- **Thru_POA collocation optimization.** This strategy uses a collocation safe stub to handle operation invocations on a collocated object. Invoking an operation via a collocation safe stub ensures the following checks are performed: (1) applicable client policies are used, (2) the server ORB (same/different than the client ORB) has not been shutdown, (3) the thread-safety of all ORB and POA operations, (4) the POA managing the servant still exists, (5) the POA Manager of this POA is queried to check if invocations are allowed, (6) the servant for the collocated object is still active, (7) the POA Current's context is set up for this upcall, and (8) all POA policies (*e.g.*, the *ThreadPolicy*, *LifespanPolicy*, and *ServantRetentionPolicy* are respected.

- **Direct collocation optimization.** To minimize the overhead of the standard collocation strategy describes above, it is possible to implement collocation to forward all requests directly to the servant class, thereby bypassing the POA. When implemented correctly, the performance of ZEN's `Direct` collocation strategy should be competitive to that of invoking a virtual method call on the servant. This strategy is not compliant with the CORBA standard, however, since: (1) the POA Current is not set up, (2) interceptors are not enabled, (3) the POA manager state is ignored, (4) not all POA policies are not considered (*e.g.*, the *ThreadPolicy* and *RequestProcessing* policies are circumvented, and (5) the ORB's status is not checked.

Empirical results. The performance of the collocation strategies described above was compared with that of no collocation, *i.e.*, where client and server communicate via the loopback device. The level of collocation was set to the the per-ORB strategy for this experiment. The measurements were performed on an Intel Pentium III 864 Mhz processor with 256 MB of main memory. For these experiments, ZEN version 0.8 was compiled using the GNU `gcj` compiler version 3.2.1 and executed using jRate [32] 0.3a on Linux 2.4.7-timesys-3.1.214 kernel.

Figure 9 shows the performance of the individual collocation strategies in ZEN. With no collocation, ZEN performs 1,675 call/sec. With the `Thru_POA` collocation optimization, the performance is greatly improved to about 43,000 calls/sec. The `Direct` collocation strategy gives the best performance of around 53,000 calls/sec. These metrics show that `Direct` collocation would be more suitable for real-time systems that require high throughput and low latency. The standard collocation strategy is still three times faster than the non-collocated request processing strategy.

3 Enhancing ZEN's ORB Core using the RTSJ

The OMG Real-time CORBA specification was adopted several years before the RTSJ was standardized. CORBA's Java mapping therefore does not use any RTSJ features, such as

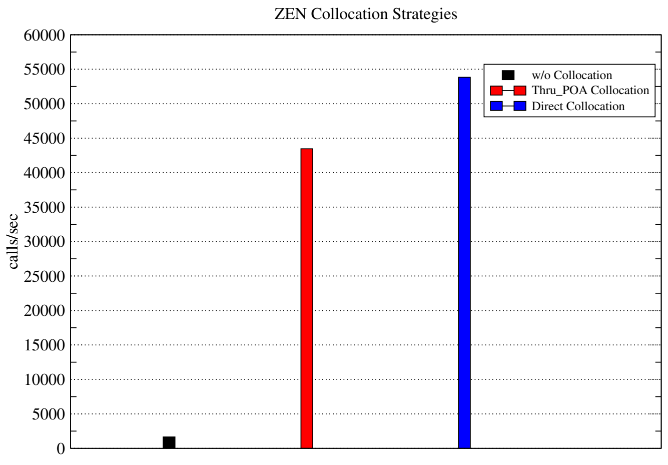

Figure 9: Performance of ZEN's Collocation Strategies

`NoHeapRealtimeThread` and `ScopedMemory`. To have a predictable Java-based Real-time CORBA ORB like ZEN, however, it is necessary to take advantage of RTSJ features to reduce interference with the GC and ensure predictability.

This section first identifies problems in the original design of ZEN, which was initially based on regular (*i.e.*, non-RTSJ) Java. We then analyze a typical end-to-end critical code path of a CORBA request within the original ZEN ORB, which was based on regular Java. Based on this analysis, we describe how we are enhancing the ZEN ORB Core to use RTSJ features, such as real-time threads and scoped memory, to improve its predictability.

3.1 Problems in the Original Design of ZEN

In the original architecture of ZEN [14], key ORB components that are involved in request/response processing (such as acceptors, connectors, transports, and thread pools) were originally allocated in the heap as shown in Figure 10 . This architecture suffered from the following problem: *the ORB allocates several temporary objects during the processing of a remote request/response*. This allocation can lead to *demand garbage collection*, *i.e.*, execution of the GC when the Java `new` operator cannot find enough memory. Execution of the GC can cause unbounded preemption latency to the thread processing the request. The situation is exacerbated if the request is critical (*i.e.*, highest priority), which can be catastrophic for certain types of safety- and mission-critical DRE applications. To eliminate priority inversions related to invocations of the garbage collector during a request upcall, it is essential that key ORB objects be allocated either within scoped or immortal memory. These objects would not cause demand garbage collection, thus minimizing the interference with the GC and enhancing the predictability of the ORB and DRE application.

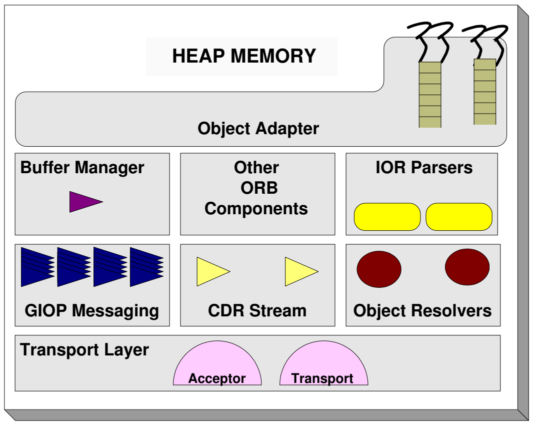

Figure 10: The Original ZEN ORB Core Architecture

Sidebar 1 provides an overview of the RTSJ thread & memory models.

3.2 Applying RTSJ Features to ZEN

RTSJ application goals. The goals for the application of RTSJ features include:

- **Minimizing interference with GC.** Garbage collection is generally considered to be unsuitable for DRE applications with stringent real-time requirements. Although there have been recent advances in GC algorithms [33, 34], the Java GC can preempt any thread in the system, leading to the thread incurring unacceptably long preemption latencies. A key goal of ZEN is therefore to avoid allocating critical ORB components in heap memory to reduce the number of GC sweeps.
- **Compliance with the CORBA specification.** To preserve compliance with the Real-time CORBA specification, the RTSJ features must be incorporated within ZEN's ORB Core without requiring any modifications to the Real-time CORBA specification. Options that require the end-user to be RTSJ-aware, such as associating scoped memory at the POA level, are provided as non-standard ZEN-specific options.
- **Interoperability with normal Java.** ZEN is designed to use intelligent strategies for component creation and extensibility [35] that allow configurability of real-time features (such as the number of static/dynamic threads, thread priorities, and buffer size) using properties and policies. These strategies use techniques, such as reflection [36, 37, 38] and aspects [39], to create real-time/vanilla Java components, thereby minimizing time/space overhead for applications that do not require real-time features.

Identification of Steps Our redesign of ZEN for Real-time CORBA began by identifying the participants associated with processing a request at both the client and server sides. For each participant identified, we associated the component with non-heap regions and resolved challenges arising from this association.

The first step needed to identify where to apply RTSJ features required the analysis of the end-to-end critical code path in ZEN. Figure 11 depicts the participants involved in servicing a CORBA two-way invocation. The discussion of the critical code path has been generalized using the Acceptor-Connector [40] pattern and thread-per-connection concurrency strategy. Sidebar 2 describes the various concurrency strate-

Sidebar 1: Overview of the RTSJ Thread & Memory models.

This sidebar briefly describes the RTSJ thread and memory models:

RTSJ memory model. The RTSJ has devised memory regions that bypass the GC, thereby giving application programmers control over the time at which memory is allocated and reclaimed. The memory regions introduced include:

- **Immortal memory.** The RTSJ introduces the concept of *immortal memory*, where allocated objects have the same lifetime as the Java Virtual Machine (JVM). The objects allocated in it are not subject to garbage collection.
- **Scoped Memory.** Like immortal memory, a scoped memory area is not garbage collected. Unlike immortal memory, however, the lifetime of a scoped memory region is not persistent. Instead, it is reference counted, indicating the number of active threads in that region. After the count drops to zero, the memory region is considered inactive and objects allocated in it are reclaimed.
- **Nested Scopes.** A real-time thread can make a scoped memory (m1) region its current allocation context by using the `enter()` method on the region. Subsequently, the thread may enter another scoped region (m2) making it its allocation context. The region m2 is an inner/nested scope of m1.

To maintain referential integrity, the JVM enforces certain rules for assignment in all memory regions, summarized in the table shown below:

Reference to:	**Heap**	**Immortal**	**Scoped**
Heap	Yes	Yes	**No**
Immortal	Yes	Yes	**No**
Scoped	Yes	Yes	**Inner Scope**
Local Variable	Yes	Yes	**Inner Scope**

Threading model. The RTSJ extends the Java threading model and introduces two new types of threads: `RealtimeThread` and `NoHeapRealtimeThread` (NHRT). NHRT threads are special real-time threads that do not "touch" the heap, *i.e.*, they cannot load or store a reference to an object in the heap. `NoHeapRealtimeThread` can therefore have execution eligibility higher than the garbage collector.

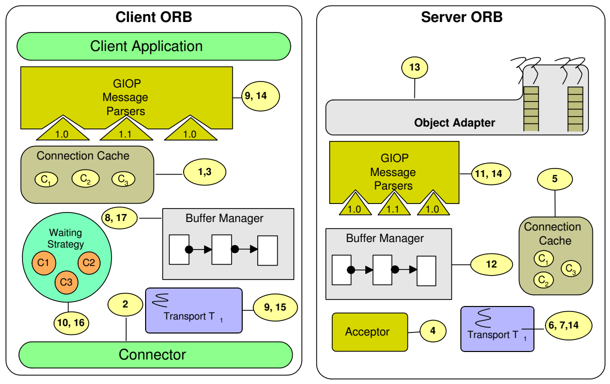

Figure 11: Tracing an Invocation Through the ZEN CORBA ORB

gies and patterns used by ZEN to process GIOP requests.

Sidebar 2: Concurrency Strategies and Patterns in ZEN

ZEN's architecture is based on many of the patterns described in [40]. Below we outline the key patterns in the ZEN ORB for request demultiplexing and dispatching:

- **Thread-per-connection.** In a server using this strategy, each concurrently connected clients is serviced by a dedicated thread. The thread completes a requested operation synchronously before servicing other requests. Hence to service multiple clients, the server spawns multiple threads. This is the default strategy used in ZEN.
- **Reactive synchronous.** GIOP request and reply handlers register with a Reactor [40] that uses a synchronous event demultiplexer to wait for data to arrive. When data arrives, the reactor is notified by the demultiplexer, which then dispatches the registered handler to service the request. In ZEN, reactive I/O support is present via java's `nio` package. Reactive I/O can be enabled in ZEN by turning on the *zen.asynch* option in the properties file.
- **Acceptor-Connector** is an initialization pattern that decouples the connection establishment between ORBs from the processing they perform after they are connected.
- **Half-Sync/Half-Async** is a pattern that decouples asynchronous and synchronous service processing in concurrent systems to simplify programming without unduly reducing performance. Synchronous service layer performs application processing services that run in separate threads. The Asynchronous service layer performs ORB-level processing of requests sent from clients.

We next describe the sequence of steps a client ORB performs to actively create a connection when a CORBA request is invoked by the application, *i.e.*, `result = object.operation (arg)`.

Connection management. We first describe how ZEN establishes a connection between a CORBA client and a server.

1. The client ORB's connection cache (`ConnectorRegistry` class in ZEN) is queried for an existing connection to the server, obtained from the object reference on which the operation is invoked.
2. If no previous connection exists, a separate connection handler is created (`Transport` class in ZEN) T_1 and the Connector connects to the server
3. This connection is added to the `ConnectorRegistry` since C_1 is bidirectional.

The activities of the server ORB for accepting a connection are described next:

4. An acceptor accepts the new incoming connection.
5. This connection C_1 is then added to the server's connection cache (`AcceptorRegistry` class in ZEN) as the server may send requests to the client.
6. A new connection handler T_1 is created to service requests.
7. The Transport's event loop waits for data events from the client.

Synchronous request/reply processing. The following are the steps involved when a client invokes a synchronous two-way request to the server.

8. The `BufferManager` class is queried to obtain a buffer to marshal the parameters in the operation invocation.
9. The appropriate GIOP Message Writer marshals the request and the Transport sends the request to the server.
10. The `WaitingStrategy` class associated with the transport waits for a reply from the server.

The server ORB performs the following activities to process the request.

11. The request header on connection C_1 is read to determine the size of the request.
12. A buffer of the corresponding size is obtained from the buffer manager to hold the request and the request data is read into the buffer.
13. The request is the demultiplexed to obtain the target POA, servant, and skeleton servicing the request. The upcall is dispatched to the servant after demarshaling the request.
14. The reply is marshaled using the corresponding GIOP message writer; Transport sends reply to the client.

The client ORB performs the following activities to process the reply from the server:

15. The Reader reads the reply from the server on the connection.

16. Using the request ID, the Waiting Strategy identifies the target Transport.
17. The parameters are then demarshaled and control is returned to the client application, which processes the reply.

Analyzing Request Processing Steps The request processing steps described above reveal the following characteristics:

- **Repetitive.** The steps involved with request/reply processing are repetitive, *i.e.*, carried out for every request. Steps 11-14 at the server side for request processing and steps 15-17 at the client side remain the same for each request from the client/server. Similarly, steps 1-3 are performed for every remote request sent to the server.
- **Independent & memoryless.** Steps required for processing request/response from two different client/server(s) are independent, *i.e.*, they do not share any context. Moreover, two requests from the same client do not share any context.
- **Ephemeral.** The objects created during the execution of these steps remain valid only for the duration of *one* cycle of request/response processing. ORBs therefore usually cache these resources to minimize resource management.
- **Thread bound.** Each of the steps are executed by a request processing thread. For example, steps 11-14 at the server side are executed by the transport and thread-pool threads.

The aforementioned characteristics of the steps lend themselves to the application of RTSJ features in the following manner:

- **Real-time Threads.** The thread-bound property of the steps enables components *e.g.*, acceptor-connector and transports to be associated with real-time threads. In particular, each of these components is designed based on an *logic* part, implemented as a Java class that implements the `Runnable` interface. This part is then associated with a scoped memory region and bound with the thread at creation time.
- **Scoped memory.** The ephemeral property of the steps enable the Upcall[4] objects to be associated with scoped memory regions. Sidebar 3 explains the various ways of creating upcall objects in scoped memory.

The repetitive, independence, and memoryless properties of the steps further shape how an ORB implementor can associate scoped memory. The *repetitive and memoryless* properties enable the request/response processing steps to be carried out within a scoped memory region[5], process the request and send the response to the client. The memory region is then exited[6] enabling all the objects created to be freed, thus minimizing the number of GC sweeps. This cycle is repeated for the next request. The *independence property* validates the above mechanism, allowing objects created during request processing to be freed before processing a subsequent request.

Sidebar 3: Object Creation in Scoped Memory

The RTSJ provides the following mechanisms for associating objects with scoped memory:

- **newInstance()**– using this method an object can be explicitly created in a scoped memory region. This method takes in the class constructor and the list of parameters arguments and uses reflection to instantiate the object.
- **new Thread()**– each real-time thread at creation time may be associated with a scoped memory region. This action leads to that region being used as the default allocation context. For *e.g.*, NHRT threads are associated with a scoped memory as they cannot "touch" the heap.
- **enter()**– using this method, a real-time thread can make a memory region its current allocation context. This method takes a runnable object as an argument. Objects created during the execution of the runnable's `run()` method are created in the current memory region and are finalized once the run method is completed.
- **executeInArea()**– this method has the effect of executing the runnable object's run method in the memory area on which the method was invoked. This method along with `newInstance` and `newArray` methods, allow applications to change the current execution context without actually changing the thread's scope stack.

Creation of scoped memory regions unlike heap/immortal memory requires the size of the memory region to be specified. However, the footprint required to process request/response is dynamic, *i.e.*, varies based on:

- **Request size.** The request size at the server depends on the size of the request sent by the client.
- **Options associated.** The footprint required during request processing depends on the options enabled.
- **Type of Request.** The request size directly depends on the type of GIOP request *e.g.*, a LOCATE_REQUEST message would be of a different size when compared to a normal request.

The most appropriate memory size would therefore have to be chosen during initialization time. One solution to this problem is to create the one huge chunk of memory. However,

[4]Upcall objects are *per request* objects that have the context necessary to process a remote request.

[5]Using the `enter()` method the memory region can be made the current allocation context.

[6]Exiting a memory region is implicit, done after the completion of the `run` method.

this solution is non scalable. Further, some JVMs may not be able to allocate huge chunk of scoped memory region. To address this problem, in ZEN we use *Nested Scopes* (explained in Sidebar 1) for every request/response demultiplexing phase, which is explained in Section 3.3.

3.3 Applying Scoped Memory within ZEN's ORB Core

To enhance predictability, we apply RTSJ features *e.g.*, scoped memory to ORB components along the critical request/response processing path. Moreover, to minimize the effect of pre-allocating memory regions, we use nested scope memory regions for each demultiplexing phase. Below, we explain the three broad phases of request processing, *i.e.*, at the server side and describe how we associate scoped memory with each of the three phases. Similar correlation exists at the client side.

1. I/O layer.

- **Steps.** This phase of demultiplexing corresponds to the steps 4-7 described in Section 3.2.
- **Participants.** The participants for this phase include, acceptors, connectors, and transports.
- **RTSJ application.** Each of these components are thread-bound components and are designed based on the *inner class* paradigm. This class derives from the `Runnable` interface and corresponds to the logic run by the thread. Instead of creating the entire component in scoped memory, we create the inner logic class in a scoped memory region, m$_I/O$. This logic class is associated with the thread at creation time. During ORB execution, multiple clients may connect to it, creating transports for every active client. Each of the transports will have a dedicated m$_{I/O}$ region. We collectively refer to these regions as a *space*.

2. ORB Core layer.

- **Steps.** This phase of demultiplexing corresponds to the Steps 11-12 in Section 3.2.
- **Participants.** GIOP Message parsers, Buffer Allocators and CDR Streams.
- **RTSJ application.** On receipt of new data events from the socket, the Transport reads the message header from the stream. Based on the size of the header, a `RequestMessage`[7] is created. After reading the request from the stream, the appropriate message parser is associated based on the type of the request. The message parser and the RequestMessage buffer are created in a nested memory region, m$_{ORB}$. The *ORB space* is a nested memory region. Based on RTSJ memory rules, references from the ORB to the I/O space are valid, *i.e.*, every m$_O RB$ scope may hold references to the corresponding m$_I/O$ region.

[7]This class encapsulates a buffer to hold the request.

3. POA Layer.

- **Steps.** This phase of demultiplexing corresponds to the steps 13-14 in Section 3.2.
- **Participants.** Upcall objects, and thread-pools
- **RTSJ application.** The message parser parses the request to find the target POA and servant. An `Upcall` object is created to hold all information necessary to perform the upcall on the skeleton. A worker thread in the thread-pool then performs the upcall. A `CDROutputStream` is created, to hold the response, which is then sent to the client. The `Upcall` objects and the output buffers are created in a nested scoped memory region m$_P OA$. The *POA space* is the innermost memory region. Again, references from POA to ORB or I/O space are valid.

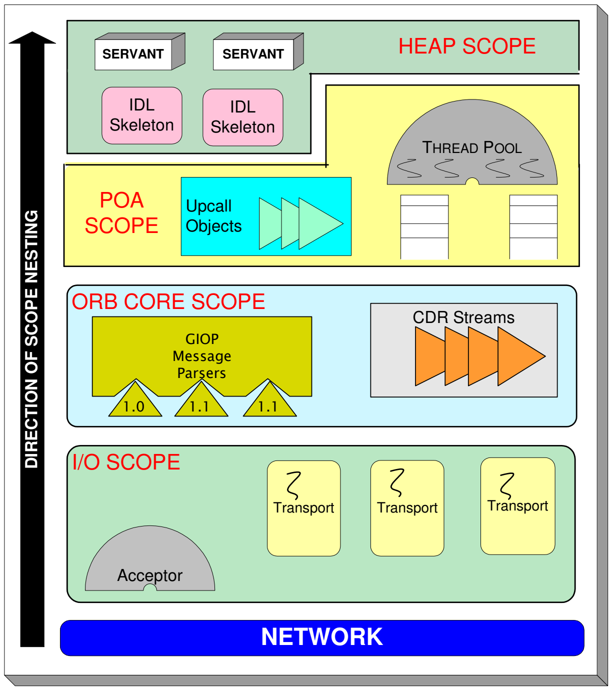

Figure 12: The Layered Architecture of ZEN

Figure 12 illustrates the layered architecture of ZEN's ORB Core. The figure also shows an *application layer*, associated with heap memory, where servants and IDL skeletons are created. Currently, the application of RTSJ features is within the ORB Core and does not require the application developer to be RTSJ aware. The architecture does not violate any of the RTSJ reference rules as (1) any of the ORB Core layers may hold references to the application layer and (2) a real-time thread can always allocate from Heap memory (enter it) without violating the single-parent rule.

Figure 13 (A) illustrates the nesting of scopes within the ZEN ORB Core. The I/O space is the outermost memory region while the POA layer is the innermost. Memory regions are entered from outer →inner, while references are maintained from inner → outer. On completion of a request, the memory regions are exited from innermost to outermost. All the objects thus created for request processing are finalized minimizing interference with the GC. Figure 13(B) depicts the scope stack structure of the request processing threads in ZEN.

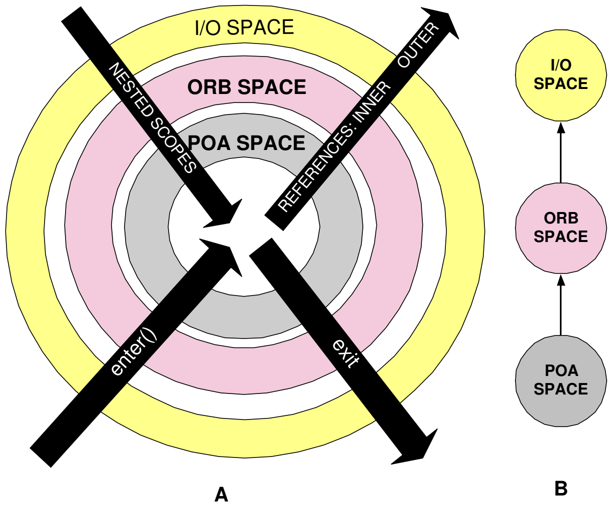

Figure 13: Scope Nesting in ZEN ORB

The ZEN architecture is compliant with the CORBA specification and is transparent to the application developer. In our architecture, however, a `NoHeapRealtimeThread` cannot be used for request processing as the application layer is heap allocated. The use of a `NoHeapRealtimeThread` is critical to enhancing the predictability of a Real-time CORBA ORB, which would require that the end-user be RTSJ aware. In ZEN, we plan to provide policies at the POA level that would determine the type of real-time thread to used for request processing. Thus an RTSJ aware end user can allocate servants in a memory region other than the heap and set the type of upcall processing thread to NHRT to enhance predictability.

4 Related Work

In recent years, a considerable amount of research has focused on enhancing the predictability of real-time middleware for DRE applications. In this section, we summarize key efforts related to our work on ZEN.

Real-time CORBA middleware research. Real-time CORBA 1.0 implementations are available from variety of suppliers including e*ORB from PrismTechnologies and ORBExpress from Object Interface Systems. Real-time CORBA has also been extensively studied in the research literature.

The *Time-triggered Message-triggered Objects* (TMO) project [41] 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.

The *ROFES* project [42] is a Real-time CORBA implementation for embedded systems. ROFES uses a microkernel-like architecture [42]. ROFES has been adapted to work with several different hard real-time networks, including SCI [43], CAN, ATM, and an ethernet-based time-triggered protocol [44].

The *URI* project [45] is a Real-time CORBA system developed at the US Navy Research and Development Laboratories (NRaD) and the University of Rhode Island (URI). The system supports expression and enforcement of dynamic end-to-end timing constraints through timed distributed method invocations (`TDMI`s) [46]

The *The ACE ORB TAO* [47] is a widely-used, open-source ORB compliant with most of the CORBA 3.0 specification [11]. TAO has been used in mission-critical DRE applications for over six years [48]. TAO supports the Real-time CORBA 1.0 specification and portions of Real-time CORBA 2.0.

RTSJ middleware research. RTSJ middleware is an emerging field of study. Researchers are focusing at RTSJ implementations, benchmarking efforts, and program compositional techniques.

The TimeSys corporation has developed the official RTSJ Reference Implementation (RI) [49], which is a fully compliant implementation of Java that implements all the mandatory features in the RTSJ. TimeSys has also released the commercial version, JTime, which is an integrated real-time JVM for embedded systems. In addition to supporting a real-time JVM, JTime also provides an ahead-of-time compilation model that can enhance RTSJ performance considerably.

The jRate *[32, 50]* project is an open-source RTSJ-based real-time Java implementation developed at Washington University, St. Louis. jRate extends the open-source GNU Compiler for Java (GCJ) run-time system [51] to provide an ahead-of-time compiled platform for RTSJ.

The *Real-Time Java for Embedded Systems* (RTJES) program [52] is working to mature and demonstrate real-time Java technology. A key objective of the RTJES program is to assess important real-time capabilities of real-time Java technology via a comprehensive benchmarking effort. This effort is examining the applicability of real-time Java within the context of real-time embedded system requirements derived from Boeing's Bold Stroke avionics mission computing architecture [53].

The researchers at the Washington University, St Louis are investigating automatic mechanisms [54] that enable existing Java programs to become storage-aware RTSJ programs. Their work centers on validating RTSJ storage rules using program traces and introducing storage mechanisms automatically and reversibly into Java code.

5 Concluding Remarks and Future Work

Distributed Real-time and Embedded (DRE) systems are growing in number and importance as software is increasingly used to automate and integrate information systems with physical systems. Over 99% of all microprocessors are now used for DRE systems [55] to control physical, chemical, or biological processes and devices in real time. In general, real-time middleware (1) off-loads the tedious and error-prone aspects of distributed computing from application developers to middleware developers, (2) provides standards that ultimately reduce development time, and (3) enhances extensibility for future application needs. In particular, Real-time CORBA has been used successfully in DRE systems, conveying the advantages of middleware to the unique and challenging requirements of DRE systems.

This paper presents the optimizations applied in ZEN to implement Real-time CORBA using Real-time Java. To achieve effective end-to-end real-time predictability within an ORB Core, the following two levels of optimizations must be considered:

1. Applying optimization principles to ensure predictability.
2. Applying RTSJ features effectively within an Real-time CORBA ORB Core.

As a part of the first level of optimizations, this paper illustrated the optimizations applied to the ZEN ORB Core, including its memory management and collocation schemes. Since these optimizations are applied at the algorithmic and data structural level, they are independent of the RTSJ implementation. For the second level optimization, the paper focused on a previously unexplored dimension in real-time middleware: *the integration of RTSJ features to support Real-time CORBA*. We showed how scoped memory and real-time threads can be associated within a real-time ORB Core without violating RTSJ rules, yet still remaining compatible with the CORBA specification.

Our ongoing research on ZEN is focusing on:

- **Completing the Real-time CORBA implementation** that resides atop a mature RTSJ layer,
- **Effective use of RTSJ features to implement Real-time CORBA**, such as using appropriate policies at the POA level to associate scoped memory with the CORBA POA,
- **Resolving challenges arising from use of Real-time Java features**, *e.g.*, the association of scoped memory with the POA may restrict servants from being registered with multiple POAs due to rules associated with scoped memory.
- **Enhance our benchmarking suite based on representative operational DRE applications**. Our first target platform is Boeing Bold Stroke [53], which is a framework for avionics mission computing applications. We are collaborating with researchers from Boeing and the AFRL *Real-Time Java for Embedded Systems (RTJES)* program [52] to define a comprehensive benchmarking suite [56].

References

[1] Object Management Group, *The Common Object Request Broker: Architecture and Specification, Revision 2.6*, Dec. 2001.

[2] J. P. Morgenthal, "Microsoft COM+ Will Challenge Application Server Market." www.microsoft.com/com/wpaper/complus-appserv.asp, 1999.

[3] A. Wollrath, R. Riggs, and J. Waldo, "A Distributed Object Model for the Java System," *USENIX Computing Systems*, vol. 9, November/December 1996.

[4] J. Snell and K. MacLeod, *Programming Web Applications with SOAP*. O'Reilly, 2001.

[5] M. Pohlmann and M. Schonefeld, "An evolutionary integration approach using dynamic corba in a typical banking environment." `http://www.corba.org/success.htm`, 2001.

[6] R. Zahavi and D. S. Linthicum, *Enterprise Application Integration with CORBA Component & Web-Based Solutions*. New York: John Wiley & Sons, 1999.

[7] D. C. Schmidt, "R&D Advances in Middleware for Distributed, Real-time, and Embedded Systems," *Communications of the ACM special issue on Middleware*, vol. 45, pp. 43–48, June 2002.

[8] Object Management Group, *Real-time CORBA Specification*, OMG Document formal/02-08-02 ed., Aug. 2002.

[9] Object Management Group, *CORBA Messaging Specification*. Object Management Group, OMG Document orbos/98-05-05 ed., May 1998.

[10] M. Henning and S. Vinoski, *Advanced CORBA Programming with C++*. Reading, MA: Addison-Wesley, 1999.

[11] Object Management Group, *The Common Object Request Broker: Architecture and Specification*, 3.0 ed., June 2002.

[12] I. Pyarali, D. C. Schmidt, and R. Cytron, "Techniques for Enhancing Real-time CORBA Quality of Service," *IEEE Proceedings Special Issue on Real-time Systems*, May 2003.

[13] Bollella, Gosling, Brosgol, Dibble, Furr, Hardin, and Turnbull, *The Real-Time Specification for Java*. Addison-Wesley, 2000.

[14] R. Klefstad, D. C. Schmidt, and C. O'Ryan, "The Design of a Real-time CORBA ORB using Real-time Java," in *Proceedings of the International Symposium on Object-Oriented Real-time Distributed Computing*, IEEE, Apr. 2002.

[15] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal, *Pattern-Oriented Software Architecture—A System of Patterns*. New York: Wiley & Sons, 1996.

[16] 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.

[17] D. C. Schmidt, V. Kachroo, Y. Krishnamurthy, and F. Kuhns, "Applying QoS-enabled Distributed Object Computing Middleware to Next-generation Distributed Applications," *IEEE Communications Magazine*, vol. 38, pp. 112–123, Oct. 2000.

[18] D. C. Schmidt, M. Deshpande, and C. O'Ryan, "Operating System Performance in Support of Real-time Middleware," in *Proceedings of the* 7^{th} *Workshop on Object-oriented Real-time Dependable Systems*, (San Diego, CA), IEEE, Jan. 2002.

[19] Object Management Group, *Real-time CORBA Joint Revised Submission*, OMG Document orbos/99-02-12 ed., Feb. 1999.

[20] D. C. Schmidt and S. Vinoski, "The History of the OMG C++ Mapping," *C/C++ Users Journal*, Nov. 2000.

[21] D. C. Schmidt and S. Vinoski, "Standard C++ and the OMG C++ Mapping," *C/C++ Users Journal*, Jan. 2001.

[22] I. ZeroC, "The Internet Communications Engine^{TM}." `www.zeroc.com/ice.html`, 2003.

[23] S. Grarup and J. Seligmann, "Incremental garbage collection," Tech. Rep. Student Thesis, Department of Computer Science, Aarhus University, Aug. 1993.

[24] S. Albers and M. Mitzenmacher, "Average case analyses of first fit and random fit bin packing," in 9^{th} *Annual ACM Symposium on Discrete Algorithms*, May 1998.

[25] E. Coffman, D. Johnson, P. Shor, and R. Weber, "Markov chains, computer proofs and average-case analysis of best fit bin packing," in *Proceedings of the* 25^{th} *Annual ACM Symposium on Theory of Computing*, (New York, USA), Aug. 1993.

[26] R. Hutchins, *Java NIO*. O'Reilly & Associates, 2002.

[27] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, MA: Addison-Wesley, 1995.

[28] M. S. Johnstone and P. R. Wilson, "Memory fragmentation problem: Solved?," 1977.

[29] W. Pugh and J. Spacco, "Mpjava: High-performance message passing in java using java.nio," in *MASLAP'03 Mid-Atlantic Student Worskshop on Programming Language and Systems*, Apr. 2003.

[30] N. Wang, D. C. Schmidt, and S. Vinoski, "Collocation Optimizations for CORBA," *C++ Report*, vol. 11, pp. 47–52, November/December 1999.

[31] 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.

[32] A. Corsaro and D. C. Schmidt, "The Design and Performance of the jRate Real-Time Java Implementation," in *On the Move to Meaningful Internet Systems 2002: CoopIS, DOA, and ODBASE* (R. Meersman and Z. Tari, eds.), (Berlin), pp. 900–921, Lecture Notes in Computer Science 2519, Springer Verlag, 2002.

[33] P. Cheng and G. Belloch, "A parallel, real-time garbage collector," in *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 125–136, 2001.

[34] D. F. Bacon, P. Cheng, and V. T. Rajan, "A real-time garbage collector with low overhead and consistent utilization," in *Proceedings of the 30th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pp. 285–298, ACM Press, 2003.

[35] A. Corsaro, D. C. Schmidt, R. Klefstad, and C. O'Ryan, "Virtual Component: a Design Pattern for Memory-Constrained Embedded Applications," in *Proceedings of the* 9^{th} *Annual Conference on the Pattern Languages of Programs*, (Monticello, Illinois), Sept. 2002.

[36] Gordon S. Blair and G. Coulson and P. Robin and M. Papathomas, "An Architecture for Next Generation Middleware," in *Proceedings of the IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing*, (London), pp. 191–206, Springer-Verlag, 1998.

[37] Fábio M. Costa and Gordon S. Blair, "A Reflective Architecture for Middleware: Design and Implementation," in *ECOOP'99, Workshop for PhD Students in Object Oriented Systems*, June 1999.

[38] F. Kon, F. Costa, G. Blair, and R. H. Campbell, "The Case for Reflective Middleware," *Communications ACM*, vol. 45, pp. 33–38, June 2002.

[39] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier, and J. Irwin, "Aspect-Oriented Programming," in *Proceedings of the 11th European Conference on Object-Oriented Programming*, June 1997.

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

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

[42] RWTH Aachen, "ROFES." http://www.rofes.de, 2002.

[43] M. P. S. Lankes and T. Bemmerl, "Design and Implementation of a SCI-based Real-Time CORBA," in *4th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC 2001)*, (Magdeburg, Germany), IEEE, May 2001.

[44] M. R. S. Lankes and A. Jabs, "A Time-Triggered Ethernet Protocol for Real-Time CORBA," in *5th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC 2002)*, (Washington, DC), IEEE, Apr. 2002.

[45] 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.

[46] 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.

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

[48] 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), pp. 184–199, ACM, Oct. 1997.

[49] TimeSys, "Real-Time Specification for Java Reference Implementation." `www.timesys.com/rtj`, 2001.

[50] A. Corsaro and D. C. Schmidt, "Evaluating Real-Time Java Features and Performance for Real-time Embedded Systems," in *Proceedings of the* 8^{th} *IEEE Real-Time Technology and Applications Symposium*, (San Jose), IEEE, Sept. 2002.

[51] GNU is Not Unix, "GCJ: The GNU Compiler for Java." `http://gcc.gnu.org/java`, 2002.

[52] Jason Lawson, "Real-Time Java for Embedded Systems (RTJES)." `http://www.opengroup.org/rtforum/jan2002/slides/java/lawson.pdf`, 2001.

[53] D. C. Sharp, "Reducing Avionics Software Cost Through Component Based Product Line Development," in *Proceedings of the 10th Annual Software Technology Conference*, Apr. 1998.

[54] M. Deters, N. Leidenfrost, and R. K. Cytron, “Translation of Java to Real-Time Java using aspects,” in *Proceedings of the International Workshop on Aspect-Oriented Programming and Separation of Concerns*, (Lancaster, United Kingdom), pp. 25–30, Aug. 2001. Proceedings published as Tech. Rep. CSEG/03/01 by the Computing Department, Lancaster University.

[55] Alan Burns and Andy Wellings, *Real-Time Systems and Programming Languages, 3rd Edition*. Addison Wesley Longmain, Mar. 2001.

[56] K. R. L. David C. Sharp, Edward Pla and R. J. H. II, “Evaluating real-time java for mission-critical large-scale embedded systems,” in *Proceedings of the* 9^{th} *IEEE Real-Time Technology and Applications Symposium* (G. Bollella, ed.), (Washington D.C), pp. 30–37, 2003.