RTZen: Highly Predictable, Real-time Java Middleware for Distributed and Embedded Systems ⋆ ⋆⋆

The original publication is available at www.springerlink.com

Krishna Raman, Yue Zhang, Mark Panahi, Juan A. Colmenares⋆⋆⋆, Raymond Klefstad, and Trevor Harmon

Department of Electrical Engineering and Computer Science
University of California, Irvine, CA 92697, USA
kraman, yuez, mpanahi, jcolmena, klefstad, tharmon@uci.edu

Abstract. Distributed real-time and embedded (DRE) applications possess stringent quality of service (QoS) requirements, such as predictability, latency, and throughput constraints. Real-Time CORBA, an open middleware standard, allows DRE applications to allocate, schedule, and control resources to ensure predictable end-to-end QoS. The Real-Time Specification for Java (RTSJ) has been developed to provide extensions to Java so that it can be used for real-time systems, in order to bring Java's advantages, such as portability and ease of use, to real-time applications.
In this paper, we describe RTZen, an implementation of a Real-Time CORBA Object Request Broker (ORB), designed to comply with the restrictions imposed by RTSJ. RTZen is designed to eliminate the unpredictability caused by garbage collection and improper support for thread scheduling through the use of appropriate data structures, threading models, and memory scopes. RTZen's architecture is also designed to hide the complexities of RTSJ related to distributed programming from the application developer. Empirical results show that RTZen is highly predictable and has acceptable performance. RTZen therefore demonstrates that Real-Time CORBA middleware implemented in real-time Java can meet stringent QoS requirements of DRE applications, while supporting safer, easier, cheaper, and faster development in real-time Java.

Keywords: RTSJ, Real-Time CORBA, Design Patterns, Middleware, DRE

1 Introduction

For as long as computers have been able to talk to one another, software engineers have struggled with the task of building distributed applications. Over the years, various technologies have been created to deal with the problem, culminating in the "golden age of networking" of the early 1980s, which saw the advent of remote procedure calls and the socket metaphor. More recently, object-oriented architectures such as CORBA have become popular for making computer communication easier to implement.

Traditionally, the overhead of CORBA-based middleware has limited its deployment to large enterprise-class servers and workstations. Developers of distributed, real-time,

⋆ This material is based upon work supported by the National Science Foundation under Grant No. 0410218, Boeing DARPA contract Z20402, and AFOSR grant F49620-00-1-0330.
⋆⋆ Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.
⋆⋆⋆ Also with the Applied Computing Institute, College of Engineering, University of Zulia

and embedded (DRE) systems, who must contend with far more limited resources, often seek lighter-weight alternatives, such as socket libraries, but these solutions are nearly as tedious and error-prone as they were following their invention a quarter-century ago.

In the last few years, however, research has shown that intelligent design and careful implementation of CORBA can produce middleware that meets the needs of today's DRE developers [1]. By bringing the CORBA model to the DRE domain, the low-level details of the network are abstracted away to the middleware layer, which shortens and simplifies the development cycle for distributed applications. Thus, DRE developers can enjoy the same benefits of CORBA that enterprise developers have enjoyed for many years, such as interoperability across varying hardware, languages, and operating systems.

CORBA middleware for DRE developers offers more benefits than just simplicity and portability. The recent Real-Time CORBA Specification [2] provides stringent quality of service (QoS) constraints on memory, performance, and dependability. CORBA middleware that conforms to this specification improves predictability by bounding priority inversions and managing system resources end-to-end. Such features are vital for DRE systems.

One key challenge in adopting CORBA, however, has been the steep learning curve for C++ middleware implementations, primarily due to the complexity of the CORBA-C++ mapping [3–5]. Simpler, easier-to-use languages, particularly Java, have been applied successfully to address this problem [6]. Java offers less "accidental complexity" than C++, a higher degree of portability, native support for concurrency and synchronization, a comprehensive class library, and other features that make it attractive to application developers.

In the DRE domain, however, Java middleware has previously been unable to offer the necessary QoS guarantees of predictability for two primary reasons: i) the under-specified scheduling semantics of Java threads can lead to the most eligible thread not always being run; and ii) the Java garbage collector can preempt any other Java thread, thus yielding unpredictably long preemption latencies.

The need to allocate or reclaim memory can potentially be a major source of unpredictability if such operations are allowed to occur *on demand* in unexpected circumstances (e.g., reallocating a buffer to handle a larger-than-expected amount of data, or having a garbage collector run to reclaim memory). To address this concern, the Real-Time Java Experts Group has defined the Real-Time Specification for Java [7]. RTSJ brings a simpler, more portable, and easier-to-use language to the world of DRE systems. It provides stronger guarantees on thread semantics than conventional Java and defines a new memory management model that allows allocation of objects not subject to garbage collection.

By using these newly-defined real-time Java features, CORBA middleware implemented in Java can provide the best of both worlds: a portable, developer-friendly language and the guarantee of predictability required by DRE systems. Implementing such middleware is not simply a feat of engineering, however. It remains to be seen, for instance, if the developer community will accept the strict scoped memory model of RTSJ, or whether ongoing research into real-time garbage collection will make such memory models obsolete.

Real-time systems are inherently more complex to develop and maintain than conventional systems. Thus, designing and implementing a software system as powerful as CORBA middleware, using the new RTSJ features for real-time memory management, is necessarily more complex than developing systems in conventional Java. However, RTSJ still retains many of Java's advantages compared to C++, such as superior portability and native thread support. Furthermore, RTSJ's memory model may be easier to manage than that of C++, which requires programmers to handle the memory management of

each individual object. RTSJ addresses this problem with the concept of *scoped memory*, allowing the system to reclaim the memory of multiple objects automatically. Maintaining entire blocks of memory as scopes can be less complex and error-prone than managing each object manually, as in C++.

Mapping Real-Time CORBA object lifetime models into this RTSJ memory model is a challenging task. The system must be designed carefully to ensure predictability through RTSJ features, while simultaneously complying with the Real-Time CORBA Specification, all the while shielding these complexities from the middleware user and maintaining Java's key advantage: ease of use.

In this paper, we show how we achieved these goals in designing and implementing the first open-source real-time Java, Real-Time CORBA middleware, which we call *RTZen*.[1] The largest known open-source RTSJ project, RTZen demonstrates that real-time Java and Real-Time CORBA are maturing into viable technologies for DRE system development. More importantly, our work proves that these specifications can be integrated into a single middleware architecture that combines the advantages of each. The result is a predictable, efficient, customizable, and embeddable RTSJ implementation of CORBA.

The remainder of this paper is organized as follows: Section 2 explains the RTSJ features used in RTZen to ensure predictability, with special focus on memory scoping; Section 3 describes the RTSJ-specific design patterns that we adopted in RTZen's implementation; Section 4 describes the architecture of RTZen; Section 5 presents empirical results that demonstrate RTZen's ability to accommodate real-time requirements; Section 6 describes related work; and in Section 7 we provide concluding remarks.

2 Overview of RTSJ

Java offers developers significant advantages, with features like object-oriented programming, platform independence, dynamic class loading, simplified memory management, exception handling, and run-time consistency checks. However, the Java VM mechanism that enables simplified memory management—the garbage collector—introduces challenges for real-time systems by potentially causing unbounded priority inversions, thus reducing predictability. To address this challenge, RTSJ reduces the need for garbage collection by introducing new types of memory regions and real-time threads.

2.1 RTSJ memory areas and switching

In addition to heap memory in standard Java, RTSJ introduces two new memory regions with restrictions aimed at making memory management more predictable. RTSJ specifies three memory regions: heap memory, immortal memory, and scoped memory. Each memory region has an associated life-span, and objects may be allocated within these regions by setting the allocation context before making allocations.

- **Heap memory** is the same as the original Java heap. Objects can be allocated in heap memory, and are alive until the last reference to them is removed, when the object becomes "garbage." Garbage objects may be collected automatically by the garbage collector. The running of a garbage collector is undesirable for real-time systems, because it may be invoked at a time which causes higher-priority tasks to be interrupted from accomplishing their time-critical task.[2] The lifespan of heap memory is the same as that of the JVM; i.e., objects created in heap memory can stay alive as long as the JVM exists or until they become garbage.

[1] Available at http://doc.ece.uci.edu

[2] This is assuming that real-time garbage collection is not used.

- **Immortal memory** is a fixed-sized area whose lifetime is the same as that of the JVM. Objects allocated in immortal memory, however, will never be garbage collected. Therefore, if not managed carefully, the memory in this region could easily become exhausted which will cause an `OutOfMemoryException`. As a consequence, this region must be used sparingly and managed carefully. In particular, memory allocations from the immortal region should generally occur at application initialization.
- **Scoped memory** is a memory region with a limited lifetime. The end of this lifetime occurs when there are no more threads executing in the region. Scoped memory is ideal for temporary allocations that follow the lifetimes of specific threads of control. The benefit of using scoped memory is that it is both allocated and reclaimed as a single (not necessarily contiguous) block,[3] which are predictable operations.

RTSJ also introduces two new thread types which can be used to execute in memory regions and are used to determine the lifetime of scoped regions. The most important feature of these new threads is that they are scheduled preemptively so that the highest priority thread is always running.

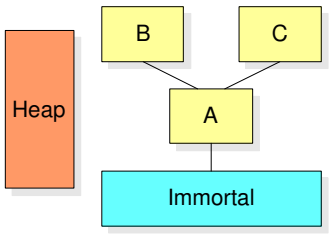
- `RealtimeThread`s (RTTs) are used to enter scoped, immortal, and heap regions. Also, memory located in the heap can be referenced from any other region, following the rules imposed by RTSJ (see Sect. 2.2).
- `NoHeapRealtimeThread`s (NHRTTs) are similarly used to enter scoped and immortal regions, but possess one important distinction: no heap access is allowed. According to normal memory access rules, any region can access the heap. However, if there is code executing in a NHRTT, that code cannot access the heap. The important consequence of this restriction is that NHRTTs can never be preempted by the garbage collector, whereas RTTs can. Therefore, NHRTTs should be used whenever possible to ensure predictability, even if heap memory will also be used in the application.

2.2 Nested Scopes

Scoped memory may be nested, producing a scoping structure called a scope stack. Since multiple memory areas can be entered from an existing memory area, this scope stack can form a tree-like structure. One key relationship is as follows: if region B is entered from region A, then A is considered the *parent* of B (see Fig. 1(a)). Certain rules govern memory access among scopes. Code within a given memory scope A can reference memory in another region B only if the lifetime of the memory in the region B is at least as long as that of the first region A. This lifetime can be guaranteed only if the requested object resides in an ancestor region (i.e., a parent or grandparent, etc.), immortal, or heap memory. A violation of this rule results in an `IllegalAssignmentError` or `IllegalAccessError`.

One important constraint is that a memory region can have only one parent, thereby preventing cycles in the scope stack. Consequently, a single scope cannot have two or more threads from different parent scopes enter it. If one thread takes a particular path to get to a memory region and forms a scoped memory hierarchy, a second thread will have to follow the same hierarchy to reach the same memory region, otherwise a `ScopedCycleException` is thrown. For example, if a thread enters scope B from A, then another thread that enters B must also be entered from A. An important implication of this restriction on scoping structure is that a given region cannot access memory residing in its "sibling" region. In the event that these two regions need to coordinate to perform

[3] While RTSJ supports both linear- and variable-time allocation of scoped memory regions, we strictly use the linear-time allocation mechanism in this work.



(a) Nested Scopes.

	to Heap	to Immortal	to A	to B	to C
from Heap	yes	yes	no	no	no
from Immortal	yes	yes	no	no	no
from A	yes	yes	yes	no	no
from B	yes	yes	yes	yes	no
from C	yes	yes	yes	no	yes

(b) Access rules for (a).

Fig. 1. RTSJ Access Rules.

some task, they will need to do so through memory stored in a common ancestor region. For example, in Fig. 1(a), scope C cannot access scope B. These regions can coordinate only via objects stored in A or immortal memory. Table 1(b) depicts the complete access rules among scopes in Fig. 1(a).[4]

The new memory regions introduced in RTSJ and described above provide memory that will not be managed by the garbage collector, but the restrictions imposed on these memory regions pose challenges for designing real-time middleware such as RTZen.

3 RTZen's Design Patterns

Traditional design patterns [8, 9] are used to simplify the development process of large software systems. Using design patterns leads to better modularity and maintainability of code. RTZen is based on such design patterns, especially those used in the development of networked and concurrent object-oriented middleware systems such as *Acceptor-Connector*, *Half-sync/Half-async* and *Interceptor*.

Design patterns have the potential to mitigate the complexity of RTSJ to a large degree. Consequently, some RTSJ design patterns have been proposed in the literature [10–12]. Also, additional RTSJ design patterns have been discovered in the course of developing RTZen, and the main goal of this section is to describe them.

3.1 Summary of Existing RTSJ Patterns

The patterns below alleviate some of the most common difficulties that an RTSJ programmer is likely to encounter. These difficulties mostly pertain to properly handling scoped memory hierarchies and obeying memory access rules.

Immortal Singleton. The Immortal Singleton pattern [12] is a simple adaptation of the classical *Singleton* pattern [8]. It allows the creation of a unique instance of a class from immortal memory, allowing it to be accessed from any memory area.

Wedge Thread. A Wedge Thread [10, 11] is used to prevent the premature reclamation of a scoped memory area by controlling its lifetime. It consists of a real-time thread that enters a scope and blocks, waiting for a signal to exit the area. Wedge threads should be used sparingly since they occupy system resources.

[4] Table 1(b) assumes that real-time threads are used. Note that if no-heap real-time threads are used, no references to the heap are permitted.

Memory Pool. The Memory Pool pattern [10] is a set of instances of a given class preallocated in a specific memory area (e.g., immortal memory). When an instance of this class is requested, an object is taken from the pool and when the instance is no longer needed, it is returned to the pool. Depending on the implementation, the pool size may vary (e.g., if the pool is empty, a new instance may be created and returned). In general, pooled objects must be mutable, so they can be reconfigured and reused.

Encapsulated Method. The Encapsulated Method pattern [11] allows the allocation of objects that represent intermediate results of an algorithm in a *temporary scope*. After the final result is obtained, the temporary scope is discarded, thereby avoiding unnecessary allocations in the original scope.

Multi-scoped Object. The Multi-scoped Object pattern allows transparent access of an object regardless of the originating region of the callee. This pattern ensures that the necessary steps are taken to guarantee that a given method is called from the correct scope by performing the proper memory scope traversals on behalf of the callee. Pizlo et al. [11] attempt to generalize the idea, but they cover only the case of a multi-scoped object performing allocations in its own scope from a child scope, among other simpler cases.

Memory Block. The Memory Block pattern [10] allows the pooling, via serialization, of objects of varying sizes in a byte array block allocated from immortal memory, thus allowing read and write access from any memory scope and any thread type. When an object is discarded, the memory block makes those bytes available for further use. This pattern can be used to communicate information between scopes and threads otherwise forbidden by RTSJ access rules. However, it has important disadvantages: i) it requires explicit memory management, and ii) (de)serialization incurs additional overhead.

3.2 New RTSJ Patterns

In developing one of the largest and most complex open-source RTSJ software projects, we have encountered more situations that warrant the use of four new design patterns.

Separation of Creation and Initialization.

Context. To use memory efficiently, RTSJ applications typically create some pools of recyclable objects, preallocated in specific memory areas such as immortal memory [10].

Problem. Creation of objects in another memory area requires the use of Java reflection. But reflection can become memory inefficient when creating objects with parameters because the parameters for the reflection call must be objects themselves.

Solution. To solve this issue, the Separation of Creation and Initialization pattern is used. It defines classes with the default constructor that creates uninitialized instances, as well as accessor methods that allow the modification of the object's internal state (i.e., the configuration) just before they are going to be used. RTZen uses this pattern to (de)marshal requests, as well as to create ORB and POA façades in memory pools.

Cross-scope Invocation.

Context. RTSJ programmers often encounter situations in which the calling object needs to invoke an operation on an object allocated in an different scope, such as in a sibling scope.

```
public class
     ExecuteInRunnable
   implements Runnable{
  private Runnable r;
  private MemoryArea a;
  public void init(
    Runnable r,MemoryArea a){
    this.r = r; this.a = a;
  }
  public void run(){
    try { a.enter(r);}
    catch(Throwable ex){...}
  }}
```

Fig. 2. The `ExecuteInRunnable` class.

```
MemoryArea parent;
ScopedMemory sibling;
Runnable logic;
...
ExecuteInRunnable eir =
  EIRPool.getEIR();
eir.init(logic, sibling);
...
try { parent.executeInArea(
      eir);}
catch (Throwable t) { ... }
finally { EIRPool.freeEIR(eir
      );}
...
```

Fig. 3. Using `ExecuteInRunnable`.

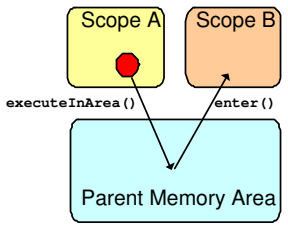


Fig. 4. Invocation between sibling scopes.

Problem. However, the memory access rules of RTSJ dictate that a given object can be accessed *directly* only if it is residing in the calling object's scope stack (an ancestor scope). Therefore, for *indirect* access to occur, elaborate memory traversal must be performed, in which the control thread must first jump to a scope that is a common ancestor of both objects, then enter the callee object's region (possibly traversing intermediate regions along the way), and finally invoke the operation.

Solution. By using the `ExecuteInRunnable` class (see Fig. 2), the Cross-scope Invocation pattern can simplify the indirect access process. If necessary, this `ExecuteInRunnable` class can be used repeatedly to perform such a memory traversal.

Figures 3 and 4 show the use of this pattern. Assume the simplest case in which B and C are sibling scopes and A is their parent memory region, with B being the current scope (Fig. 4). After being instantiated using the default constructor or obtained from a pool, the `ExecuteInRunnable` object is initialized within the sibling scope C and a `Runnable` object that contains the logic to be executed in B. Once the `executeInArea` method of the `MemoryArea` class is called by B, the `ExecuteInRunnable` object starts to run in A, making the current thread enter C and finally execute the logic provided in the `Runnable` object.

As is common in RTSJ programming, the allocation of arguments and returned values of the requested method require special care to avoid illegal access errors: arguments must be accessible from the callee scope, and returned values must be accessible from the caller scope. This requirement may add significant code complexity, but this complexity can be alleviated by the adoption of the Memory Pool and Memory Block patterns [10].

Immortal Exception.

Context. In RTSJ applications, exceptions may need to be thrown and handled in different memory areas.

Problem. However, in RTSJ, the propagation of exceptions is restricted by memory access rules. A given exception object must be handled in a memory area that can legally reference that exception. If not, a `ThrowBoundaryError` is returned and the original exception is lost.

RTSJ's memory area rules introduce accidental complexity into exception handling. The CORBA specification requires exceptions to be thrown in many scope regions. However, some of those exception objects cannot be handled in their local scopes, yet cannot be legally accessed from the region that can handle them either. For example, an exception raised in the *Thread Pool Scope* may need to be handled in *ORB Memory Scope*, but this access is prohibited by RTSJ memory access rules.

Corsaro et al. [12] proposed that exceptions can be initially handled in the local scope. With this approach, the notification of the exceptional condition is encapsulated in a status variable or object and then transferred to an outer scope, where the condition is finally handled, or propagated again to an outer scope. Although effective, this approach has the following drawbacks: 1) the code complexity is increased; 2) the exception propagation mechanism is tightly coupled with the system's memory structure; 3) the actual exceptional condition may not be reported correctly because of an inappropriate mapping between the exception type and the status variable or object (e.g., exceptions are commonly handled using general types); and 4) system performance may be affected since the exception must be re-instantiated several times as it is propagated from scope to scope.

Solution. Consequently, we have designed the Immortal Exception pattern, an efficient and flexible solution that allows exceptions to be handled independently of the memory area in which they are thrown, without violating RTSJ referencing rules. In this pattern, a factory class that creates exception objects of specified types resides in immortal memory. The Immortal Singleton pattern [12] is used to cache the exception objects in the factory so that they can be reused (i.e., re-thrown). Distinct families of exceptions, such as CORBA system exceptions and application exceptions, are organized into different factories.

This pattern offers important advantages and a minor disadvantage. Since all exceptions are allocated in immortal memory, they can be accessed from anywhere, thereby avoiding the boundary problem. This design is particularly useful when the system must handle a large number of exceptions, such as the 400 instances of CORBA system exceptions handled by RTZen. A limitation of this pattern, however, is that since exception objects are preallocated, no message that explains the cause of the run-time exception can be associated with the exception objects. However, good documentation can alleviate this inconvenience.

Immortal Façade.

Context. A consequence of RTSJ's scoping rules is that large RTSJ applications, such as RTZen, often have complex scoping structures.

Problem. Scoping structures introduce more development complexity to application users. In general, when objects in different scopes interact using method calls, the complexity of traversing the memory structure is exposed to both the caller object and callee object. Furthermore, the caller is typically tightly coupled with the system's memory structure, in particular with the callee object's locality. This exposed complexity makes development and system maintenance more difficult and therefore compromises one of RTZen's design goals.

Solution. To hide complexity from the application developer, as well as to minimize the dependencies of the caller object on the callee object's memory locality, we used the Immortal Façade pattern based on the Gang of Four's Façade design pattern [8]. The Immortal Façade consists of a *façade class* and an *implementation class*. The façade class acts as a surrogate for and typically implements the same interface as the actual implementation class. It encapsulates the logic that handles the cross-scope invocation. The façade objects need to be accessible from scopes of interest, so they are frequently allocated in immortal memory and managed by a pool. The *implementation class* implements the actual business logic behind the façade. An instance of it is allocated in a specific scoped memory.

In RTZen, two key patterns, Cross-scope Invocation and Immortal Façade, have been used to hide the complex scoping structures between callers and callees. One example of

the combined use of these two patterns is the ORB façade. RTZen maintains a pool of ORB façade objects in immortal memory. These façades do not implement any business logic. All the logic is contained in the ORB implementation object hosted in the ORB scope. Since the ORB façade is in immortal memory, the user can access it with ease and make invocations on it. The Cross-scope Invocation pattern is used when the invocation thread needs to laterally traverse scoped regions.

4 Architecture

This section explains the rationale behind the design of RTZen. First, we outline the goals for RTZen and the CORBA features influenced by the memory and thread constructs of RTSJ. Next, we describe the design of RTZen, emphasizing its scoped memory structure and illustrating the processing of an invocation on a remote object. Finally, we present an overview of RTZen's customization features.

4.1 RTZen Design Goals

The design of RTZen has been driven by the following requirements.

- **Predictability.** Real-time middleware must provide a high degree of predictability. As a result, a Real-Time CORBA implementation requires eliminating priority inversions and bounding the size of critical sections.
- **Specification Compliance.** An ORB must be compliant to the CORBA specification to ensure application portability across ORB implementations. However, proprietary features and optimizations should still be available if they prove to be advantageous in certain cases.
- **Performance.** Even though real-time applications tend to favor predictability over performance, it is the goal of RTZen not to compromise on this requirement. RTZen aims to provide both a predictable and high performance CORBA implementation.
- **Minimize User Complexity.** One of the key aspects of middleware is that it offloads the complexities of distributed programming from the application developer to the middleware developer. In the case of RTSJ middleware, complexities related to distributed programming brought on by the addition of memory and thread constructs are offloaded as well.
- **Efficient Use of Memory.** RTSJ memory constructs must be used efficiently. Allocations must be made in the context of memory scopes or managed carefully in pools or caches located in immortal memory. Memory leaks must be completely avoided to ensure continuous system operation. If possible, use of heap memory should be avoided to ensure that the garbage collector always remains idle.
- **Customizability.** Finally, middleware should be customizable and support minimization of footprint for embedded applications while maintaining all the advantages of using middleware.

Our earlier work with ZEN [13] focused on each of these goals except for the efficient use of memory, as RTSJ implementations have only recently become available. Maturing RTSJ implementations, such as jRate [14], have provided the real-time JVM layer necessary to ensure predictability and make the memory model of RTZen possible.

4.2 Mapping Real-Time CORBA to RTSJ

Primary features of RTZen are heavily influenced by the constraints imposed by the added memory and thread constructs of RTSJ. To understand the architecture of RTZen we must first examine them.

The feature that influences the architecture of RTZen the most is the CORBA requirement that an application developer must be able to control the lifetimes of various components, including ORB instances, POA instances, and CORBA objects. As a result of this requirement, each of these components is mapped onto a scoped memory region (Section 4.3). Furthermore, the CORBA specification defines the API that must be exposed to application programmers. Since RTZen will use scoped memory regions, the traversal of its internal scoped memory structure must not be exposed to the user.

The final issue is the selection of priorities of RTSJ threads. Recall that RTSJ introduces two new types of threads: `RealtimeThread` (RTT) and `NoHeapRealtimeThread` (NHRTT). The RTSJ platform was designed under the assumption that any NHRTT will possess a higher priority than any RTT, so that NHRTTs will never block for garbage collection [15]. If the application developer chooses to use both RTTs (to access heap memory) and NHRTTs, the priority mappings can ensure that NHRTTs are always mapped to higher priorities than are RTTs.

4.3 RTZen Design

To meet all of the goals and successfully implement the Real-Time CORBA specification, RTZen was designed with a unique memory hierarchy (Fig. 5). The main purpose of this hierarchy is to enable objects to be independently allocated and freed to follow the Real-Time CORBA specification. As a side effect, this design also allows for pluggable and customizable architecture that does not use the heap.

The idea of lifetime – the length of time for which an object is valid – is central to understand the rationale behind the design of RTZen. CORBA exposes to the application the ability to both create and destroy various CORBA components (e.g., ORBs and POAs). RTZen enables this by assigning memory scopes to these components. When the user creates one of these components, the associated memory scope is created, along with a wedge thread if required. Recall that wedge threads occupy system resources; therefore they are only used in scopes where there is not already an active thread keeping that scope alive. When the component is destroyed, the associated memory scope is freed by signaling all active threads in that region to terminate (including wedge threads).

RTZen is organized as a scoped hierarchy: Fig. 5 shows the memory layout of the RTZen components. Each component with a defined lifetime is allocated in its own scope and maintains its state within the scope. Moreover, some components have child scopes for dependent components with smaller lifetimes, thus creating a tree-like scoped memory structure.

In RTZen the application initially starts in immortal memory. The first application scope region is above the initial immortal region and holds references to the ORB façade and POA façade objects which are allocated from immortal memory and cached. The ORB and POA façades internally hold a reference to the ORB and POA scoped memory region respectively, not to the corresponding implementation object itself. In both cases the implementation object is the portal of the scope. Under the ORB scope, there are various other scoped regions for transport, acceptor, POAs, thread pools, and temporary request processing. Each region has at least one thread object inside to keep the region alive. Wedge threads keep the ORB and POA regions alive, whereas threads in the other regions perform an active role for request processing.

The scoped memory structure combined with object-oriented concepts like inheritance and polymorphism enables the development of customizable and adaptable systems [16, 13]. Each component can inherit its interface from a base class and implement different features. And since each component is maintained in an individual scoped region, it can be easily plugged in and out of the runtime memory structure of the program. RTZen's protocol and transportation framework is built using this technique. Thus

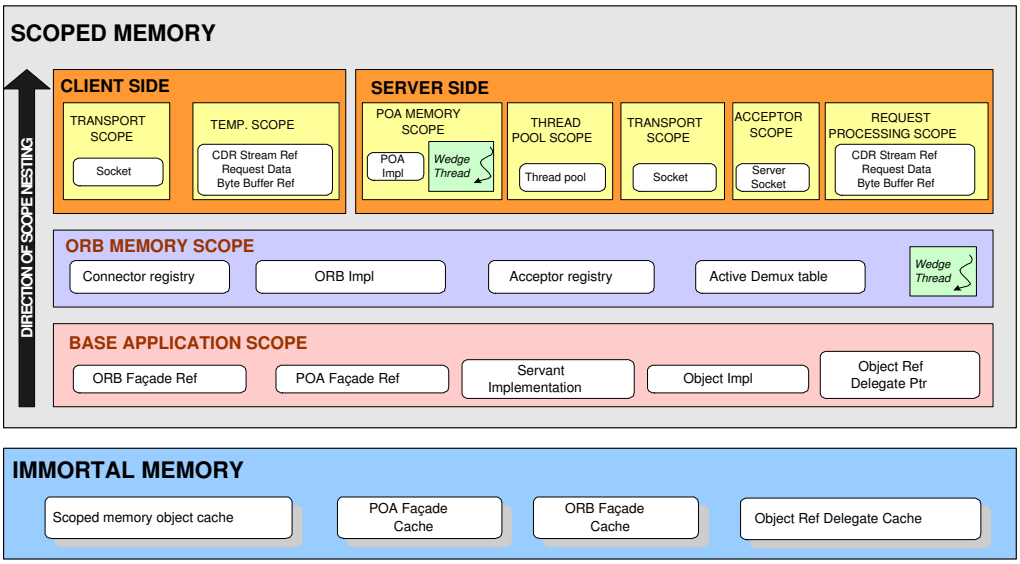


Fig. 5. Scoped Memory Structure of RTZen.

transports and protocols can be configured, added, and removed in a pluggable manner without affecting the other components of the ORB.

This scoped hierarchy also allows RTZen to avoid any heap allocation. However, since RTSJ scoped regions are not garbage collected, RTSJ developers have to be very careful about allocating and maintaining references to objects in these scoped regions. In RTZen, this issue has been resolved using memory pools and the immortal singleton pattern. Memory pools are used for any object that stores state and is simultaneously accessed by multiple request threads, while an immortal singleton is maintained for those objects which require only a global state and are accessed in a synchronous manner.

On the other hand, the scoped hierarchy introduces two accidental complexities into the design of RTZen. The first one is exception handling. Exceptions in RTSJ are not propagated beyond the scope in which they were thrown. However, the CORBA specification requires that the ORB throw exception in many locations. To solve this issue, RTZen uses a combination of local exception handling and the Immortal Exception pattern (Section 3). The second issue that may occur is creation of objects and references across scopes. RTSJ does allow creation of objects across scopes using reflection. However, if the constructor requires any arguments, then reflection causes wasteful allocation of memory for the arguments. To solve this issue, RTZen separates the creation and initialization of objects.

While allowing for more efficient memory usage and customizability, the scoped hierarchy described above potentially increases the complexity perceived by application developers – since it requires traversing the application and ORB internal scoped hierarchy to make invocations – if not for the use of two key patterns: cross-scope invocation and immortal façade (Section 3.2). One example of the combined use of these two patterns is the ORB façade. RTZen maintains a pool of ORB façade objects in immortal memory. These façades do not implement any business logic. All the logic is contained in the ORB implementation object hosted in the ORB scope. Since the ORB façade is in immortal memory, the user can access it with ease and make invocations on it. The cross-scope invocation pattern is used if this invocation's thread needs to laterally traverse scoped regions.

Along with using RTSJ scoped memory to enhance predictability, RTZen also ensures that priorities are maintained and respected throughout the ORB. To achieve this, RTZen is implemented with an endpoint-per-priority paradigm: for every distinct priority level, RTZen maintains a separate endpoint [17]. Each endpoint executes at the highest priority of requests that it may process. This ensures that i) high priority requests are not queued behind low priority requests, and ii) incoming requests are guaranteed that the thread reading the request data from the socket will run at an equal or higher priority.

RTZen also includes many of the performance and predictability enhancing techniques pioneered in ZEN [18–20] and TAO [21–24]. For example, RTZen's thread pool implements the Half-Sync/Half-Async pattern [9] to minimize complexity and allow high throughput, and the POA uses active-demux tables to allow $O(1)$ demultiplexing of server-side objects.

4.4 Sample Invocation using RTZen

This section traces through an invocation on the client and server side to illustrate the traversing of the scoped memory structure of RTZen during a remote method call. We assume that priorities are propagated with each request from the client to the server and that the server is using a thread-pool with lanes.

The server object is created on the remote end with the appropriate policies, and the corresponding Interoperable Object Reference (IOR) is generated. The IOR informs the client about the remote object's location and some supported policies. When the server object is registered on the server side, RTZen creates a separate endpoint for each supported request priority. This allows requests of varying priorities to be handled independently of each other. This information is also propagated to the client in the IOR.

After obtaining the IOR (e.g., from a Naming Service), the client application reads it and uses the client-side ORB to create a stub of the remote object. The stub acts as a placeholder for the remote object: local invocations made on the stub are translated to remote invocations on the server object by the ORB. RTZen creates the stub objects in the application scope so that the client application may invoke requests on them directly without having to traverse any scopes.

The invocation starts when the client application sets the priority of the request and invokes a method on the stub. Based on the priority, the stub locates the appropriate endpoint on the remote ORB to contact, sends the request message and then waits for the return value. Within the ORB, this translates to using the Cross-Scope Invocation pattern to jump to the ORB scope and then to the transport scope. At this point, the message is sent and the active thread jumps back to the ORB scope and then enters to a temporary scope where it waits for the reply.

After the request message is received by the server transport, the transport thread reads the request header to locate the POA that the remote object is registered with. Then the transport thread uses cross-scope invocation to jump from the transport scope to the POA scope where it locates the reference to the target remote object. At this point, the transport thread jumps to the thread pool region and locates a thread which supports the priority of the request. The request is passed to a thread from the thread pool, and the transport thread returns to its initial scope (i.e., the transport scope) and listens for more incoming requests (Half-Sync/Half-Async pattern [9]). The thread-pool thread now processing the request uses cross-scope invocation to jump to a temporary memory scope where the request is processed. At this point, the invocation is made on the actual remote object and once the invocation is complete, the thread jumps to the transport thread and sends back the reply message.

Finally, on the client side, the client transport thread receives the reply message and jumps to the temporary scope where the thread that made the request is waiting. The

client transport thread hands the reply back to the waiting thread which exits back to the client scope and returns from the invocation on the stub.

4.5 Customization Features

Over and above the Real-Time CORBA specification, RTZen also implements some additional features which allow for greater customizability. First, RTZen allows the server-side object to be hosted on thread pools which can be based on either RTTs or NHRTTs. This feature allows the application developer to choose the tradeoff between being able to use the heap or having a more predictable environment.

Second, RTZen includes the implementation of a pluggable transport and protocol framework [25, 13] that allows the application developer to plug in custom transport layers or protocols to the ORB. This is specially useful in embedded environments where standard TCP/GIOP functionality may be unnecessary or wasteful. Currently, RTZen includes a very compact version of GIOP with reduced functionality as well as a pluggable serial transport that enables the use of the serial port for CORBA invocations.

Third, RTZen also includes a set of Mock RTSJ classes[5] which enable it to run on standard (non-RTSJ) Java VMs. This feature also allows Java developers to use a standard Java VM to prototype RTSJ applications.

Finally, we have also developed ZEN-kit [26], a user-friendly graphical tool for customizing RTZen. ZEN-kit implements a customization strategy based on conditional compilation that takes advantage of the RTZen's modular architecture. Using this tool, developers can selectively include Core and Real-Time CORBA features into the ORB in order to meet specific requirements of DRE applications, in particular those related to memory footprint.

5 Empirical Results

5.1 Testing Environment

All experiments were run on 865 MHz Pentium III (Coppermine, 256KB Cache) processors with 512MB PC133 ECC SDRAM, for both server side and client side, connected via 10 Mbps Ethernet on a closed subnet. The operating system was TimeSys Linux GPL 4.1 based on the Linux kernel 2.4.21, which supports the Native POSIX Thread Library (NPTL) [27]. The non-real-time Java Virtual Machine (JVM) used for comparison was the Sun JDK 1.4 JVM. The real-time Java platform was jRate [14], a real-time Java ahead-of-time compiler.

5.2 Performance Measurements

For all tests, measurements were based on *steady state* observations, where the system is run until the transitory effects of cold starts are eliminated before collecting the measured observations.

Measuring typical performance. We used the median as a measure of typical performance because, as so often is true in real-time systems, distributions were typically highly skewed toward the minimum observation, with a large spike near the typical observation, and with a long, low-probability tail toward the maximum.

[5] Currently, the Mock RTSJ classes expose a reduced set of the RTSJ API and do not perform allocation of access checks.

Measuring worst-case performance. We used the maximum as an estimate of a system's "worst case." The worst case is an important measurement for real-time systems because real-time systems must be designed with the assumption that the system will always deliver the worst possible performance, even though designing to that assumption is wasteful since typical times are usually near the best case [15].

For these experiments, the observed maximum in a sample size of 10,000 observations was used to estimate the worst case for each message size. A sample size at least this large was necessary to observe a reasonable estimate for the maximum latency because the maximum values tended to be extremely low-probability events. The range of the observations ($maximum - minimum$), or jitter was also used as another measure of a system's predictability.

5.3 Typical Performance: Comparison of RTZen on jRate; TAO, JacORB on Sun JVM; and RTZen on Sun JVM

The test case used here has a single thread running on the client side, sending variable-size `octet` sequences to the server side. The size ranged from 32 bytes to 1024 bytes.

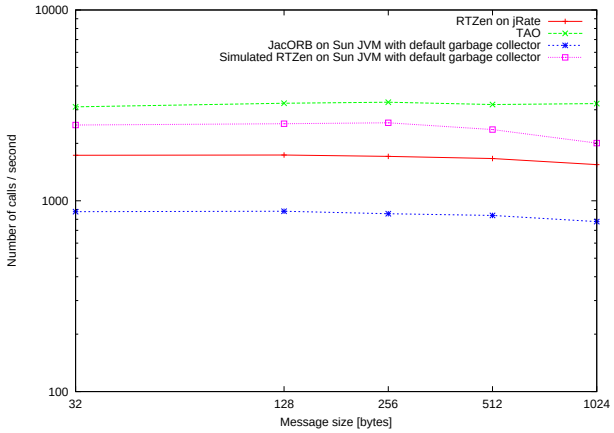


Fig. 6. Typical Performance: Comparison of RTZen on jRate; TAO, JacORB on Sun JVM; and RTZen on Sun JVM

Comparison of RTZen on Sun JVM to JacORB on Sun JVM. Java developers in non-real-time domains can afford to be careless about memory management because of the existence of the garbage collector. The process of memory housekeeping — allocating memory and cleaning it after it is used — creates overhead that can slow an application substantially. RTSJ developers, on the other hand, do not have the luxury of depending on a garbage collector for memory reuse, and must instead be more heedful of memory usage. Section 4 described the careful memory management design in RTZen. Along with the obvious effect of improved predictability, yet another consequence of careful memory management is improved performance. This would be shown by the fact that the typical performance of RTZen is better than JacORB's.

To measure this performance improvement, we compared RTZen with JacORB [6], a widely used Java-based ORB. Both ORBs were tested on the standard Sun non-real-time JVM detailed above. In this case, RTZen used its Mock RTSJ classes (Section 4.5), so

all scopes and immortal memory regions were therefore simulated as heap memory, and all allocations in those regions were subject to garbage collection.

The performance of JacORB was measured using the four types of garbage collectors (default, throughput, concurrent low pause, and incremental) supported by the JVM [28]. JacORB obtained its highest throughput with the throughput garbage collector, shown in Fig. 6. Note that, in the same conditions, RTZen significantly outperforms JacORB. Thus, the test shows the performance improvement gained from the extensive memory reuse (memory pools) and other performance enhancing techniques in RTZen (Section 4.3).

Comparison of RTZen on Sun JVM and RTZen on jRate. Figure 6 shows that RTZen on jRate performs about 30% slower than RTZen on Sun JVM. On the Sun JVM, RTZen uses the heap instead of the scoped memory and immortal memory regions; thus it does not incur any RTSJ scoped region traversal or access/allocation check penalties. In addition, jRate is not an optimizing compiler, so it generates unoptimized code; jRate also uses an open-source implementation of the Java API libraries which may not have been optimized. This measurement provides an approximate idea of the overhead introduced by RTSJ over normal Java.

Comparison of RTZen on jRate and TAO. We used TAO as our baseline measurements for RTZen performance. TAO was written in C/C++ and thus provides a good approximation of the highest performance possible by a Real-Time CORBA ORB. Figure 6 shows that RTZen is slower than TAO; however, considering the overhead of RTSJ and Java VMs discussed above, RTZen compares favorably to TAO.

5.4 Consistency: Comparison of RTZen on jRate to JacORB on Sun JVM

We next compared the round-trip latency jitter of RTZen and JacORB. JacORB was run on the Sun JVM with the default garbage collector, on which JacORB obtained its narrowest jitter; RTZen was run on jRate. Although the platforms were different, the measurements show the performance that can be expected from these ORBs on the platforms for which they were designed. Since performance was more or less equivalent across different message sizes, as shown in Fig. 6, we compared the two ORBs for a message size of 128 bytes.

Figure 7 shows the distribution of the round-trip latency values with the maximum and minimum bound indicated, as well as the circle to represent the median value. From Fig. 7 we can see RTZen is highly predictable compared to JacORB, with the jitter value of 90 μs and 9770 μs respectively; RTZen's maximum value is close to its median. Also, RTZen has not achieved this predictability by unduly degrading performance. Notably, RTZen's typical performance and predictability, as measured by the worst case observed, are within the range of time units typically used for distributed real-time systems (10 ms) [15]. These jitter values were expected and highlight the predictability gained by developing in RTSJ.

5.5 Typical Performance and Consistency: RTZen on jRate with variable message size

Figure 8 shows that RTZen is predictable across varying message sizes. RTZen performs within round-trip latency jitter of around 200 μs in all cases, which is better than the distributed real-time application requirements of 10 ms [15].

While satisfying the jitter requirement, RTZen's typical performance stays roughly constant even when message size increases. Throughput increases minimally (about

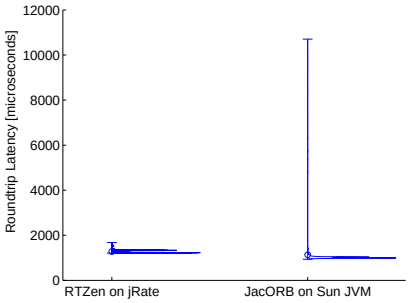


Fig. 7. Consistency: Comparison of RTZen on jRate and JacORB on Sun JVM

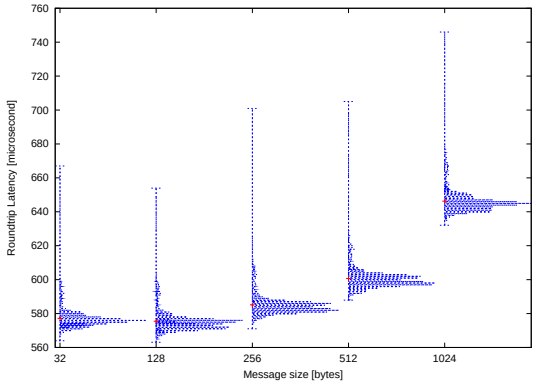


Fig. 8. Consistency: RTZen on jRate with variable message sizes.

20 μs) as the message size increases from 32 bytes to 1024 bytes. Once the message size exceeds the allocated buffer limit (1024 bytes), the round-trip latencies increase slightly (about 50 μs, about 8%). RTZen allows application developers to configure the message buffer size to customize performance and predictability as required.

5.6 Consistency: Comparison of RTZen on jRate and TAO

To compare the round-trip latency jitter of RTZen and TAO, we set up a test case running two client threads. The purpose of this experiment was to test the jitter bounds of both ORBs and to show that RTZen can be set up with NHRTTs that are not interrupted by the garbage collector. The first thread was run at the highest CORBA priority, while the second thread was run at the lowest CORBA priority. The low priority thread performed a long operation; the high priority thread performed a short action which would interrupt the lower priority thread. In RTZen, the high priority was a NHRTT, and the low priority thread was a RTT. The RTT was also set up to allocate data on the heap to generate some garbage data which would be reclaimed by the garbage collector.

Figure 9 shows a comparison of jitter measurements on the high priority thread with RTZen and TAO running. Although RTZen is still slower than TAO, the jitter of the high-priority task in RTZen is similar to TAO's. These performance and jitter measurements demonstrate RTZen's ability to accommodate real-time requirements.

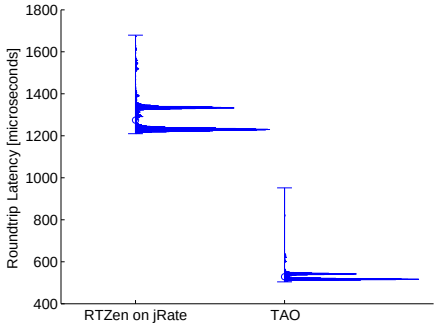


Fig. 9. Consistency: Comparison of RTZen on jRate and TAO

6 Related Work

During the last decade, a considerable amount of standardization [29] and research [30–34] work has been done on CORBA, and some results derived from this work have been incorporated in various ORBs available today, both commercial [35, 36] and open-source [37, 6, 38].

Additionally, significant efforts have been carried out to enhance the predictability and performance of CORBA and make it suitable for DRE systems. The research community has determined the strengths and limitations of CORBA as foundation for DRE systems [39, 40], and based on them, researchers have proposed i) software architecture designs [25, 23], ii) scheduling approaches and mechanisms [41–43], iii) techniques for improving quality of service [44, 24], iv) extensions for real-time network protocols [25, 45–47], v) the adaptation of CORBA services [48, 49], vi) techniques for tailoring CORBA ORBs to computational platforms under stringent resource constraints [50, 51, 13], and vii) modeling and verification methods [52]. Meanwhile, the Object Management Group has produced the Real-Time CORBA specifications [53, 17].

Several Real-Time CORBA implementations exist as of this writing. Perhaps the most well-known is TAO [21, 54], a popular open-source ORB compliant with most of the features and services defined in CORBA 3.x [55]. Built on top of TAO is CIAO [56], a CORBA Component Model (CCM) implementation for developing component-oriented DRE systems. ROFES [57] is a minimal memory footprint prototype of Real-Time CORBA. It has been adapted to work with several different hard real-time networks, including SCI [45], CAN, ATM, and an Ethernet-based time-triggered protocol [46]. Commercial Real-Time CORBA implementations are also available: OpenFusion e*ORB C Edition for Real-time [58] from PrismTech, ORBexpress RT [59] from Objective Interface Systems, and VisiBroker-RT [60] from Borland Software Corporation. Very recently, PrismTechnologies and Objective Interface Systems announced Real-Time CORBA compliant ORBs for RTSJ: OpenFusion RT for Java and ORBexpress RT for Java, respectively.

Java Remote Method Invocation (RMI) [61] is a mechanism for developing object-oriented distributed systems in Java, and there is some progress adapting RMI so that RTSJ supports timely invocation of remote objects [62]. Standard Java RMI has become more compatible with CORBA, in particular due to RMI/IIOP, a form of RMI that uses IIOP as the underlying protocol. RMI/IIOP holds promise to evolve into a bridge to RT-CORBA.

7 Conclusion

Memory management is a vital part of any RTSJ application. The RTZen architecture addresses the memory allocation and scoping issues related to implementing a Real-Time CORBA ORB using RTSJ. It provides a solid foundation for further research into implementations of Real-Time CORBA services and applications based on Java. Such research would incorporate RTSJ scheduling features into the RTZen scheduling service and provide support for custom configuration of RTZen to minimize its memory footprint for smaller embedded applications. Further research is also needed for adapting RTZen to Java virtual machines that support a real-time garbage collector.

In its current state, however, RTZen fulfills the essential goals of real-time distributed systems: predictability, specification compliance, high performance, minimal user complexity, customizability, and efficient use of memory. Our work proves that the RTSJ and Real-Time CORBA specifications can be integrated into a single middleware architecture that combines the advantages of each.

Acknowledgments

The authors thank Susan Anderson Klefstad for significant revision work and suggestions and Morgan Deters for timely jRate bug fixes. Juan A. Colmenares thanks the University of Zulia (LUZ) for supporting his participation in this research.

References

1. Schmidt, D.C.: R&D Advances in Middleware for Distributed, Real-time, and Embedded Systems. Communications of the ACM. Special Issue on Middleware **45** (2002) 43–48
2. Object Management Group: Real-time CORBA Specification. OMG Document formal/02-08-02 edn. (2002)
3. Schmidt, D.C., Vinoski, S.: The History of the OMG C++ Mapping. C/C++ Users Journal (2000)
4. Schmidt, D.C., Vinoski, S.: Standard C++ and the OMG C++ Mapping. C/C++ Users Journal (2001)
5. ZeroC, I.: The Internet Communications Engine^{TM}. www.zeroc.com/ice.html (2003)
6. Gerald Brose and André Spiegel and Reimo Tiedemann et al.: Jacorb. http://www.jacorb.org/ (2004)
7. Bollella, Gosling, Brosgol, Dibble, Furr, Hardin, Turnbull: The Real-Time Specification for Java. Addison-Wesley (2000)
8. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, Reading, MA (1995)
9. Schmidt, D.C., Stal, M., Rohnert, H., Buschmann, F.: Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects, Volume 2. Wiley & Sons, New York (2000)
10. Benowitz, E.G., Niessner, A.F.: A patterns catalog for RTSJ software designs. In: Lecture Notes in Computer Science. Volume 2889., OTM 2003 Workshops (2003) 497–507
11. Pizlo, F., Fox, J.M., Holmes, D., Vitek, J.: Real-time java scoped memory: Design patterns and semantics. In: 7th IEEE Int'l Symposium on Object-Oriented Real-Time Distributed Computing (ISORC 2004). (2004) 101–110
12. Corsaro, A., Santoro, C.: Design patterns for RTSJ application development. In: Lecture Notes in Computer Science. Volume 3292., OTM 2004 Workshops (2004) 394–405
13. Klefstad, R., Rao, S., Schmidt, D.C.: Design and Performance of a Dynamically Configurable, Messaging Protocols Framework for Real-time CORBA. In: Proceedings of the 36th Annual Hawaii Int'l Conference on System Sciences. (2003)
14. Corsaro, A., Schmidt, D.C.: The Design and Performance of the jRate Real-Time Java Implementation. In Meersman, R., Tari, Z., eds.: On the Move to Meaningful Internet Systems 2002: CoopIS, DOA, and ODBASE, Berlin, Lecture Notes in Computer Science 2519, Springer Verlag (2002) 900–921
15. Dibble, P.C.: Real-Time Java Platform Programming. Prentice Hall (2002)
16. Klefstad, R., Schmidt, D.C., O'Ryan, C.: Towards highly configurable real-time object request brokers. In: Proceedings of the 5th IEEE Int'l Symposium on Object-Oriented Real-Time Distributed Computing (ISORC 2002). (2002) 437–447
17. Object Management Group: RealTime-CORBA Specification, v 2.0. Object Management Group. OMG Document formal/03-11-01 edn. (2003)
18. Klefstad, R., Krishna, A.S., Schmidt, D.C.: Design and Performance of a Modular Portable Object Adapter for Distributed, Real-Time, and Embedded CORBA Applications. In: Proceedings of the 4th Int'l Symposium on Distributed Objects and Applications. (2002)
19. Krishna, A., Klefstad, R., Schmidt, D.C., Corsaro, A.: Towards predictable real-time Java object request brokers. In: Proceedings of the 9th IEEE Real-Time and Embedded Technology and Applications Symposium (RTTAS 2003). (2003) 49–56
20. Krishna, A., Schmidt, D.C., Klefstad, R.: Enhancing real-time CORBA via real-time java features. In: Proceedings of the 24th Int'l Conference on Distributed Computing Systems (ICDCS 2004). (2004) 66–73

21. Schmidt, D.C., Levine, D.L., Mungee, S.: The design of the TAO real-time object request broker. Computer Communications **21** (1998) 294–324
22. Gokhale, A., Schmidt, D.C.: Techniques for optimizing CORBA middleware for distributed embedded systems. In: Proceedings of the 18th Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM '99). Volume 2. (1999) 513–521
23. Pyarali, I., Spivak, M., Cytron, R., Schmidt, D.C.: Evaluating and optimizing thread pool strategies for real-time CORBA. In: Proceedings of the ACM SIGPLAN Workshop on Languages, Compilers and Tools for Embedded Systems (LCTES '01). (2001) 214–222
24. Pyarali, I., Schmidt, D.C., Cytron, R.: Techniques for Enhancing Real-time CORBA Quality of Service. Proceedings of the IEEE **91** (2003) 1070–1085
25. O'Ryan, C., Kuhns, F., Schmidt, D.C., Othman, O., Parsons, J.: The design and performance of a pluggable protocols framework for real-time distributed object computing middleware. In: IFIP/ACM Int'l Conference on Distributed Systems Platforms (Middleware '00). (2000) 372–395
26. Gorappa, S., Colmenares, J.A., Jafarpour, H., Klefstad, R.: Tool-based configuration of real-time corba middleware for embedded systems. In: Eighth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC'05), Los Alamitos, CA, USA, IEEE Computer Society (2005) 342–349
27. Corp., T.: TimeSys Linux GPL 4.1. www.timesys.com (2004)
28. Sun Microsystems, I.: Tuning garbage collection with the 1.4.2 java[tm] virtual machine. (2003)
29. Object Management Group: Catalog of OMG Specifications. http://www.omg.org/technology/documents/spec_catalog.htm (2005)
30. Gokhale, A., Schmidt, D.C.: Principles for Optimizing CORBA Internet Inter-ORB Protocol Performance. In: Proceedings of the 31st Annual Hawaii Int'l Conference on System Sciences. Volume 7. (1998) 376–385
31. Arulanthu, A.B., O'Ryan, C., Schmidt, D.C., Kircher, M., Parsons, J.: The Design and Performance of a Scable ORB Architecture for CORBA Asynchronous Messaging. In: Proceedings of the IFIP/ACM Int'l Conference on Distributed Systems Platforms (Middleware 2000). (2000) 208–230
32. Mishra, S., Shi, N.: Improving the Performance of Distributed CORBA Applications. In: Proceedings of the Int'l Parallel and Distributed Processing Symposium (IPDPS 2002). (2002) 36–41
33. Alberto Coen Porisini, Matteo Pradella, Matteo Rossi, Dino Mandrioli: A formal approach for designing CORBA-based applications. ACM Transaction on Software Engineering and Methodology **12** (2003) 107–151
34. Majumdar, S., Shen, E.K., Abdul-Fatah, I.: Performance of adaptive CORBA middleware. Journal of Parallel and Distributed Computing **64** (2004) 201–218
35. Borland Software Corporation: Borland Enterprise Server, VisiBroker Edition. http://www.borland.com/visibroker/ (2005)
36. IONA Technologies: Orbix 6.2. http://www.iona.com/products/orbix/ (2005)
37. McConnell, S., Pedersen, J., Evans, J.S., Kühne, L., Rumpf, M., Boyce, S., Wood, C.: Openorb community project. http://sourceforge.net/projects/openorb/ (2004)
38. Puder, A.: Mico: An open source corba implementation. IEEE Software **21** (2004) http://www.mico.org/.
39. Gokhale, A., Schmidt, D.C.: Evaluating CORBA latency and scalability over high-speed ATM networks. In: Proceedings of the 17th Int'l Conference on Distributed Computing Systems (ICDCS '97). (1997) 401–410
40. O'Ryan, C., Schmidt, D.C., Kuhns, F., Spivak, M., Parsons, J., Pyarali, I., Levine, D.L.: Evaluating policies and mechanisms for supporting embedded, real-time applications with CORBA 3.0. In: Proceedings of the 6th IEEE Real-Time Technology and Applications Symposium (RTAS 2000). (2000) 188–197
41. Gill, C.D., Levine, D.L., Schmidt, D.C.: The Design and Performance of a Real-Time CORBA Scheduling Service. Real-Time Systems **20** (2001)
42. Dipippo, L.C., Wolfe, V.F., Esibov, L., Cooper, G., Bethmangalkar, R., Johnston, R., Thuraisingham, B., Mauer, J.: Scheduling and priority mapping for static real-time middleware. Real-Time Systems **20** (2001) 155–182

43. Hao, T., Zhigang, L., Jinde, L.: An end-to-end scheduling approach for real-time CORBA. In: Proceedings of the 2002 IEEE Region 10 Conference on Computers, Communications, Control and Power Engineering (TENCON '02). Volume 1. (2002) 318–322
44. Zinky, J.A., Bakken, D.E., Schantz, R.: Architectural Support for Quality of Service for CORBA Objects. Theory and Practice of Object Systems **3** (1997) 1–20
45. Lankes, S., Pfeiffer, M., Bemmerl, T.: Design and Implementation of a SCI-based Real-Time CORBA. In: Proceedings of the 4th IEEE Int'l Symposium on Object-Oriented Real-Time Distributed Computing (ISORC 2001). (2001) 23–30
46. Lankes, S., Jabs, A., Reke, M.: A time-triggered ethernet protocol for real-time corba. In: Proceedings of the 5th IEEE Int'l Symposium on Object-Oriented Real-Time Distributed Computing (ISORC 2002). (2002) 215–222
47. Lankes, S., Jabs, A., Bemmerl, T.: Design and performance of a CAN-based connection-oriented protocol for Real-Time CORBA. Journal of Systems and Software **77** (2005) 37–45
48. Harrison, T.H., Levine, D.L., Schmidt, D.C.: The design and performance of a real-time CORBA event service. In: Proceedings of the 12th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA '97). (1997) 184–200
49. Hong, S., Kim, Y., Kweon, M., Min, D., Han, S.: Object-oriented real-time CORBA naming service on distributed environment. In: Proceedings of the 12th Int'l Conference on Information Networking (ICOIN-12). (1998) 637–640
50. Gokhale, A., Schmidt, D.C.: Optimizing a CORBA IIOP Protocol Engine for Minimal Footprint Multimedia Systems. Journal on Selected Areas in Communications - Special issue on Service Enabling Platforms for Networked Multimedia Systems **17** (1999)
51. Kim, K., Geon, G., Hong, S., Kim, S., Kim, T.: Resource-conscious customization of CORBA for CAN-based distributed embedded systems. In: Proceedings of the 3rd IEEE Int'l Symposium on Object-Oriented Real-Time Distributed Computing (ISORC 2000). (2000) 34–41
52. Rossi, M., Mandrioli, D.: A formal approach for modeling and verification of rtcorba-based applications. In: Proceedings of the 2004 ACM SIGSOFT Int'l Symposium on Software Testing and Analysis (ISSTA '04). (2004) 263–273
53. Object Management Group: Real-Time CORBA (Static Scheduling). 1.2 edn. (2005)
54. Schmidt, D.C.: TAO. Real-time CORBA with TAO (The ACE ORB). http://www.cs.wustl.edu/ schmidt/TAO.html (2004)
55. Object Management Group: Common Object Request Broker Architecture: Core Specification. 3.0.3 edn. (2004)
56. Schmidt, D.C.: CIAO. Real-time CCM with CIAO (Component Integrated ACE ORB). http://www.cs.wustl.edu/ schmidt/CIAO.html (2004)
57. RWTH Aachen: ROFES. http://www.rofes.de (2005)
58. PrismTech Corporation: OpenFusion e*ORB C Edition for Real-time. http://www.prismtechnologies.com (2005)
59. Objective Interface Systems, Inc.: ORBexpress RT. http://www.ois.com (2005)
60. Borland Software Corporation: VisiBroker-RT. http://www.borland.com/visibroker/ (2005)
61. Sun Microsystems Inc.: Java Remote Method Invocation (Java RMI). http://java.sun.com/products/jdk/rmi/ (2004)
62. Borg, A., Wellings, A.: A real-time RMI framework for the RTSJ. In: Proceedings of the 15th Euromicro Conference on Real Time Systems. (2003)