The Design and Performance of Next-generation Real-time CORBA Middleware

Arvind Krishna, Raymond Klefstad
{krishnaa, klefstad}@uci.edu
EECS Department
UC Irvine
Irvine, CA 92697, USA

Douglas C. Schmidt
schmidt@dre.vanderbilt.edu
ISIS
Vanderbilt University
Nashville, TN 37203, USA

Angelo Corsaro
corsaro@cs.wustl.edu
CS Department
Washington University
St. Louis, MO 63130, USA

Chapter Overview

Distributed Real-time and Embedded (DRE) systems are becoming increasingly widespread and important. There are many types of DRE systems, but they have one thing in common: *the right answer delivered too late becomes the wrong answer.* Common DRE systems include telecommunication networks (*e.g.*, wireless phone services), process automation (*e.g.*, hot rolling mills), and defense applications (*e.g.*, avionics mission computing systems). The various aspects of DRE systems have the following challenging requirements.

- As *distributed systems*, DRE systems require capabilities to manage connections and message transfer between separate machines.
- As *real-time systems*, DRE systems require predictable and efficient control over end-to-end system resources.
- As *embedded systems*, DRE systems have weight, cost, and power constraints that limit their computing and memory resources. For example, embedded systems often cannot use conventional virtual memory, since software must fit on low-capacity storage media, such as EEPROM or NVRAM.

Designing DRE systems, that implement all the required capabilities, that are fast and reliable, and that use limited computing resources is hard; building them on time and within budget is even harder. In particular, DRE applications developers face the following challenges:

- **Tedious and error-prone development** — Accidental complexity proliferates, because many DRE applications are still developed using low-level languages, such as C and assembly languages.
- **Limited debugging tools** — Although debugging tools are improving, real-time and embedded systems are still hard to debug due to inherent complexities, such as concurrency and remote debugging.
- **Validation and tuning complexities** — It is hard to validate and tune key quality of service (QoS) properties, such as (1) pooling concurrency resources, (2) synchronizing concurrent operations, (3) enforcing sensor input and actuator output timing constraints, (4) allocating, scheduling, and assigning priorities to computing and communication resources end-to-end, and (5) managing memory.

Due to these challenges, developers of DRE applications have historically rediscovered core concepts and reinvented custom solutions that are tightly coupled to particular hardware and software platforms.

Over the past decade, distributed object computing (DOC) middleware frameworks, such as CORBA, COM+, and Java RMI have emerged to reduce the complexity of developing distributed applications. DOC middleware simplifies application development for distributed systems by off-loading many tedious and error-prone aspects of distributed computing from application developers to middleware developers. It has been used successfully in business and desktop systems where scalability, evolvability, and interoperability are essential for success.

Real-time CORBA is a rapidly maturing DOC middleware technology standardized by the OMG that can simplify many challenges for DRE applications, just as CORBA, COM+, and Java RMI have been used for business and desktop systems. The Real-time CORBA 1.0 specification was designed for applications with hard real-time requirements, such as avionics mission computers and process manufacturing controllers. Real-time CORBA 1.0 adds QoS control capabilities to improve DRE application predictability by bounding priority inversions, providing end-to-end priority enforcement, and end-to-end system resource management. In particular, the standard features defined by the Real-time CORBA 1.0 specification enable DRE applications to configure and control the following 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.

This chapter explains how the features in the Real-time CORBA 1.0 specification enables predictable and efficient control over process, communication, and memory resources. It also illustrates how application developers can apply the Real-time CORBA features to simplify the design of DRE systems. The material in this chapter is based on our extensive experience developing and applying *The ACE ORB* (TAO), which is an open-source implementation of Real-time CORBA 1.0 written in C++. TAO has been applied to hundreds of DRE applications in domains that include telecommunications, distributed interactive simulations, aerospace, process control, and defense.

Although the Real-time CORBA 1.0 specification was integrated into the OMG standard several years ago, it has not been adopted universally by DRE application developers, due in part to the following limitations

- **Steep learning curve**, caused largely by the complexity of the CORBA C++ mapping,
- **Run-time and memory footprint overhead**, which stem from monolithic ORB implementations that include all the code supporting the various core ORB services, such as connection and data transfer protocols, concurrency and synchronization management, request and operation demultiplexing, (de)marshaling, and error-handling,
- **Lack of support for dynamic scheduling**, which is needed to support applications with stringent soft real-time requirements, such as telecommunication call processing and streaming video.

This chapter presents the following contributions to the design and use of next-generation Real-time CORBA middleware that resolves the limitations outlined above:

1. It explains how the dynamic scheduling features in the Real-time CORBA 2.0 enable more flexible control over process, communication, and memory resources than is possible with Real-time CORBA 1.0.
2. It compares various software architectures for implementing Real-time CORBA, ranging from conventional monolithic ORB architectures to more flexible micro-ORB architectures.
3. It shows how design patterns can be applied to minimize the memory footprint of DRE middleware customized for various types of applications.
4. It describes how Real-time Java features can be applied to achieve low and bounded jitter for ORB operations and eliminate sources of priority inversion.

Our expertise is the result of experience developing and applying ZEN, which is an open-source Real-time CORBA ORB implemented using Real-time Java. ZEN is inspired by many of the patterns, techniques, and lessons learned in TAO. We describe how the challenge of implementing next-generation Real-time CORBA using Real-time Java can be decomposed into the following two levels:

- **Applying optimization principles to ensure predictability.** These optimizations are applied at the algorithmic and data structural level and are independent of the Java virtual machine (JVM). These strategies are applied at:
  - **Object Adapter layer** – Optimizations applied in this layer include predictable and scalable (1) request demultiplexing techniques, that ensure $O(1)$ look up time irrespective of the POA hierarchy, (2) object key processing techniques, and (3) servant lookups.
  - **ORB Core layer** – Optimizations applied in this layer include, (1) collocation optimizations, (2) buffer-allocation strategies and (3) asynchronous I/O using Java's `nio` package.
- **Applying Real-time features effectively within a Real-time CORBA ORB.** We illustrate how Real-Time Specification for Java (RTSJ) features can be integrated with components in Real-time CORBA ORBs (such as Real-time CORBA thread pool lanes) to span the following layers:
  1. I/O layer, *e.g.*, Acceptor-Connector and Reactor,
  2. ORB Core layer, *e.g.*, CDR streams and Buffer Allocators, and
  3. Object Adapter layer, *e.g.*, Thread Pools and the POA.

  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 to enable use of NoHeapRealTime (NHRT) threads. Proper use of NHRT threads for request dispatch ensures that user requests are not interrupted in an inappropriate time.

We use ZEN as a case-study of next-generation Real-time CORBA middleware to illustrate how, well designed ORBs enable developers to make choices between efficiency and flexibility. We also illustrate how to incorporate Real-time features (such as real-time threads and scoped memory) in an ORB without modifying the

CORBA specification and show patterns that can minimize the time/space overhead for DRE applications that do not require certain Real-time CORBA features.