Middleware for Communications

Contents

Preface

1

Real-time CORBA Middleware

Arvind S. Krishna
Douglas C. Schmidt
Dept. of Electrical Engineering and Computer Science
Vanderbilt University
Nashville, TN 37235, USA

Raymond Klefstad
Dept. of Electrical Engineering and Computer Science
University of California, Irvine
Irvine, CA 92697, USA

Angelo Corsaro
Dept. of Computer Science and Engineering
Washington University
St. Louis, MO 63130, USA

1.1 Introduction

Middleware trends. Over the past decade, distributed computing middleware, such as CORBA (Obj 2002c), COM+ (Morgenthal 1999), Java RMI (Wollrath et al. 1996), and SOAP/.NET (Snell and MacLeod 2001), has emerged to reduce the complexity of developing 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

Middleware for Communications. Edited by Qusay H. Mahmoud

used successfully in desktop and enterprise systems (Zahavi and Linthicum 1999) where scalability, evolvability, and interoperability are essential for success. In this context, middleware offers the considerable benefits of hardware-, language-, and OS-independence, as well as open-source availability in some cases.

The benefits of middleware are also desirable for development of distributed, real-time, and embedded (DRE) systems. Due to their multiple constraints across different dimensions of performance, DRE systems are harder to develop, maintain, and evolve than mainstream desktop and enterprise software. In addition to exhibiting many of the same needs as desktop and enterprise systems, DRE systems impose stringent quality of service (QoS) constraints. For example, real-time performance imposes strict constraints upon bandwidth, latency, and dependability. Moreover, many embedded devices must operate under memory, processor, and power limitations.

As DRE systems become more pervasive they are also becoming more diverse in their needs and priorities. 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). The additional difficulties faced in developing these systems intensifies the need for middleware to off-load time-consuming and error-prone aspects of real-time and embedded computing, as well as to eliminate the need for continual reinvention of custom solutions.

The Real-time CORBA specification (Obj 2002b) was standardized by the OMG to support the QoS needs of DRE systems. Real-time CORBA is a rapidly maturing middleware technology designed for applications with hard real-time requirements, such as avionics mission computing (Schmidt et al. 1998b), as well as those with softer real-time requirements, such as telecommunication call processing and streaming video (Schmidt et al. 2000a). When combined with a quality real-time operating system foundation, well-tuned implementations of Real-time CORBA can meet the end-to-end QoS needs of DRE systems, while also offering the significant development benefits of reusable middleware Douglas C. Schmidt and Frank Buschmann (2003).

Contributions of this chapter. Although Real-time CORBA offers substantial benefits – and the Real-time CORBA 1.0 specification was integrated into the OMG standard several years ago – it has not been universally adopted by DRE application developers, partly due to the following limitations:

- **Lack of customization** (or the difficulty of customization), where customization is needed to enable Real-time CORBA Object Request Brokers (ORBs) to be used in diverse domains,
- **Memory footprint overhead**, stemming largely 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, and
- **Steep learning curve**, caused largely by the complexity of the CORBA C++ mapping.

This chapter therefore presents the following contributions to the design and use of Real-time CORBA middleware implementations that address the challenges outlined above:

1. It describes how highly-optimized ORB designs can improve the performance and predictability of DRE systems.

2. It shows how well-designed Real-time CORBA middleware architectures can simultaneously minimize footprint and facilitate customization of ORBs to support various classes of applications.
3. It shows that Java and Real-time Java (Bollella et al. 2000) features can be applied to an ORB to simultaneously increase ease of use and predictability for Java-based DRE applications.

The material in this chapter is based on our experience developing The ACE ORB (TAO) Schmidt et al. (1998b) and ZEN (Klefstad et al. 2002). TAO is an open-source[1] Real-time CORBA ORB for use with C++, with enhancements designed to ensure efficient, predictable, and scalable QoS behavior for high-performance and real-time applications. ZEN is an open-source[2] Real-time CORBA ORB for use with Java and Real-time Java, designed to minimize footprint and maximize ease of use. ZEN is inspired by many of the patterns, techniques, and lessons learned from developing TAO. This chapter presents TAO and ZEN as case studies of Real-time CORBA middleware to illustrate how ORBs can enable developers to control the tradeoffs between efficiency, predictability, and flexibility needed by DRE systems.

Chapter organization. The remainder of this book chapter is organized as follows: Section 1.2 provides an overview of Distributed Real-time and Embedded (DRE) systems, focusing on the challenges involved in developing these systems; Section 1.3 discusses how Real-time CORBA can be used to ensure end-to-end predictability required for DRE systems; Section 1.4 illustrates the motivation, design, successes, and limitation of TAO; Section 1.5 details the goals, technologies, design, and successes of ZEN; Section 1.6 summarizes how our work relates to other research efforts on real-time middleware; and Section 1.7 presents concluding remarks.

1.2 DRE System Technology Challenges

DRE systems are generally harder to develop, maintain, and evolve than mainstream desktop and enterprise software since DRE systems have stringent constraints on weight, power consumption, memory footprint, and performance. This section describes the requirements and challenges present in this domain, both for contemporary DRE systems and for future DRE systems.

1.2.1 Challenges of Today's DRE Systems

Some of the most challenging problems facing software developers are those associated with producing software for real-time and embedded systems in which computer processors control physical, chemical, or biological processes or devices. Examples of such systems include airplanes, automobiles, CD players, cellular phones, nuclear reactors, oil refineries, and patient monitors. In most of these real-time and embedded systems, *the right answer delivered*

[1]TAO can be downloaded from `http://deuce.doc.wustl.edu/Download.html`.
[2]ZEN can be downloaded from `http://www.zen.uci.edu`.

too late becomes the wrong answer, *i.e.*, achieving real-time performance end-to-end is essential. In addition, some embedded devices have limited memory (*e.g.*, 64-512 KB) available for the platform and applications.

The three primary characteristics of DRE systems pose the following requirements for their development:

- 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 end-to-end control over 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 electrically erasable programmable read-only memory (EEPROM) or non-volatile random access memory (NVRAM).

1.2.2 Challenges of Future DRE Systems

As hard as today's DRE systems are to develop, DRE systems of the future will be even more challenging. Many of today's real-time and embedded systems are relatively small-scale, but the trend is toward significantly increased functionality and complexity. In particular, real-time and embedded systems are increasingly being connected via wireless and wireline networks to create *distributed* real-time and embedded systems, such as total ship computing environments, tele-immersion environments, fly-by-wire air vehicles, and area/theater ballistic missile defense. These DRE systems include many interdependent levels, such as network/bus interconnects, many coordinated local and remote endsystems, and multiple layers of software.

Some of the key attributes of future DRE systems can be characterized as follows:

- Multiple quality of service (QoS) properties, such as predictable latency/jitter, throughput guarantees, scalability, dependability, and security, must be satisfied simultaneously and often in real time,
- Different levels of service will occur under different system configurations, environmental conditions, and costs, and must be handled judiciously by the system infrastructure and applications,
- The levels of service in one dimension must be coordinated with and/or traded off against the levels of service in other dimensions to achieve the intended application and overall mission results, and
- The need for autonomous and time-critical application behavior requires flexible system infrastructure components that can adapt robustly to dynamic changes in mission requirements and environmental conditions.

All of these attributes are interwoven and highly volatile in DRE systems, due to the dynamic interplay among the many interconnected parts.

DRE applications are increasingly combined to form large-scale distributed systems that are joined together by the Internet and intranets. These systems can further be combined with other distributed systems to create "systems of systems." Example of these large-scale DRE systems of systems include *Just-in-time manufacturing inventory control systems* that schedule the delivery of supplies to improve efficiency. Figure 1.1 illustrates how the combination of individual manufacturing information systems is fundamental to achieve the efficiencies of modern "just-in-time" manufacturing supply chains. Information from engineering systems

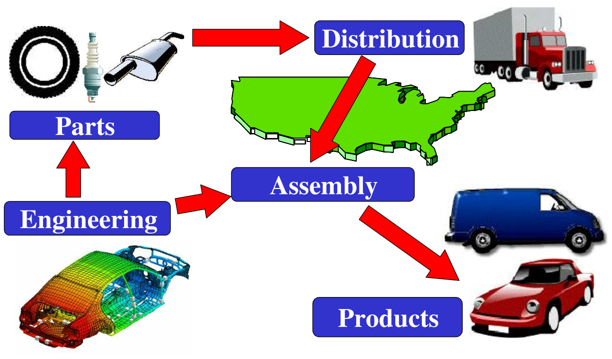

Figure 1.1 Characteristics of Manufacturing System of Systems

is used to design parts, assemblies, and complete products. Parts manufacturing suppliers must keep pace with (1) engineering requirements upstream in the supply chain and (2) distribution constraints and assembly requirements downstream. Distribution must be managed precisely to avoid parts shortages while keeping local inventories low. Assembly factories must achieve high throughput, while making sure the output matches product demand at the sales and service end of the supply chain. Throughout this process, information gathered at each stage of the supply chain must be integrated seamlessly into the control processes of other stages in the chain.

1.2.3 Limitations with Conventional DRE System Development

Designing DRE systems that implement all their required capabilities, are efficient, predictable, and reliable, and 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 DRE applications are often still developed using low-level languages, such as C and assembly language.
- **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, asynchrony, and remote debugging.
- **Validation and tuning complexities** — It is hard to validate and tune key 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 systems have historically tended to rediscover core concepts and reinvent custom solutions that were tightly coupled to particular hardware and software platforms. The continual rediscovery and reinvention associated with this software development process has kept the costs of engineering and evolving DRE systems too high for too long. Improving the quality and quantity of systematically reusable software via middleware is essential to resolving this problem Douglas C. Schmidt and Frank Buschmann (2003).

1.3 Overview of Real-time CORBA

To address the challenges for DRE systems described in Section 1.2, the OMG has standardized the Real-time CORBA specification. Version 1.0 of this specification (Obj 2002b) defines standard features that support end-to-end predictability for operations in *statically scheduled and provisioned* CORBA applications. Version 2.0 of this specification (Obj 2001) defines mechanisms for *dynamically scheduled and provisioned* CORBA applications. This section first presents an overview of the CORBA reference model and its key components and then describes how versions 1.0 and 2.0 of the Real-time CORBA specification add QoS capabilities to CORBA.

1.3.1 Overview of CORBA

CORBA Object Request Brokers (ORBs) allow clients to invoke operations on distributed objects without concern for object location, programming language, OS platform, communication protocols and interconnects, and hardware (Henning and Vinoski 1999). Figure 1.2

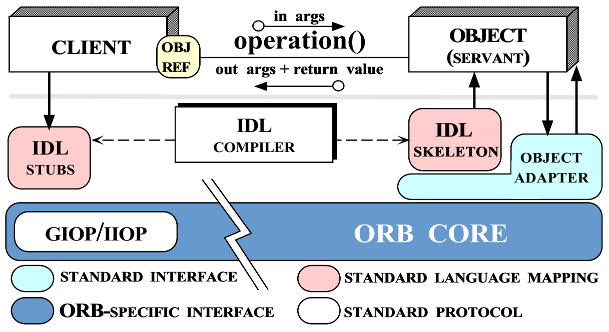

Figure 1.2 Key Components in the CORBA Reference Model

illustrates the following key components in the CORBA reference model (Obj 2002c) that collaborate to provide this degree of portability, interoperability, and transparency:

Client. A client is a *role* that obtains references to objects and invokes operations on them to perform application tasks. A client has no knowledge of the implementation of the object but does know the operations defined via its interface.

ORB Core. An 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. 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 and many embedded systems interconnects.

Object. In CORBA, an object is an instance of an OMG Interface Definition Language (IDL) interface. Each object is identified by an *object reference*, which associates one or more paths through which a client can access an object on a server. Over its lifetime, an object can be associated with one or more servants that implement its interface.

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

OMG IDL Stubs and Skeletons. IDL stubs and skeletons serve as a "glue" between the client and servants, respectively, and the ORB. Stubs implement the *Proxy* pattern (Buschmann et al. 1996) and marshal application parameters into a common message-level representation. Conversely, skeletons implement the *Adapter* pattern (Gamma et al. 1995) and demarshal the message-level representation back into typed parameters that are meaningful to an application.

Object Adapter. An Object Adapter is a composite component that associates servants with objects, creates object references, demultiplexes incoming requests to servants, and collaborates with the IDL skeleton to dispatch the appropriate operation upcall on a servant. Object Adapters enable ORBs to support various types of servants that possess similar requirements. Even though different types of Object Adapters may be used by an ORB, the only Object Adapter defined in the CORBA specification is the Portable Object Adapter (POA) Pyarali and Schmidt (1998).

1.3.2 Overview of Real-time CORBA 1.0

The Real-time CORBA 1.0 specification is targeted for *statically scheduled and provisioned* DRE systems, where the knowledge of applications that will run on the system and/or the priorities at which they execute are known *a priori*. This specification leverages features from the CORBA standard (such as the GIOP protocol) and the Messaging specification (Object Management Group 1998) (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. Figure 1.3 illustrates the standard features that Real-time CORBA provides to DRE applications to enable them 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. To enforce strict control over scheduling and execution of processor resources, Real-time CORBA

1.0 specification enables client and server applications to (1) determine the priority at which CORBA invocations will be processed, (2) allow servers to pre-define pools of threads, (3) bound the priority of ORB threads, and (4) ensure that intra-process thread synchronizers have consistent semantics in order to minimize priority inversion.

- **Communication resources** via protocol properties and explicit bindings to server objects using priority bands and private connections. An Real-time CORBA endsystem must leverage policies and mechanisms in the underlying communication infrastructure that support resource guarantees. This support can range from (1) managing the choice of the connection used for a particular invocation to (2) exploiting advanced QoS features, such as controlling the ATM virtual circuit cell rate.
- **Memory resources** via buffering requests in queues and bounding the size of thread pools. Many DRE systems use multi-threading to (1) distinguish between different types of service, such as high-priority vs. low-priority tasks (Harrison et al. 1997) and (2) support thread preemption to prevent unbounded priority inversion. Real-time CORBA, specification defines a standard *thread pool* model (Schmidt et al. 2001) to pre-allocate pools of threads and to set certain thread attributes, such as default priority levels. Thread pools are useful for real-time ORB endsystems and applications that want to leverage the benefits of multi-threading, while bounding the amount of memory resources, such as stack space, they consume. Moreover, thread pools can be optionally configured to buffer or not buffer requests, which provides further control over memory usage.

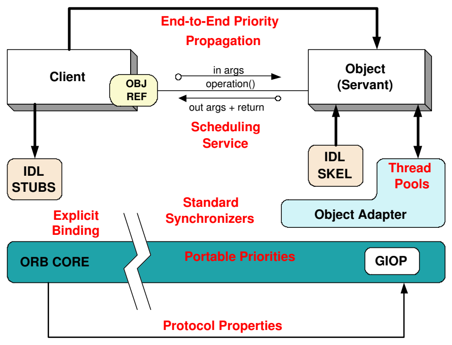

Figure 1.3 Real-time CORBA 1.0 Features

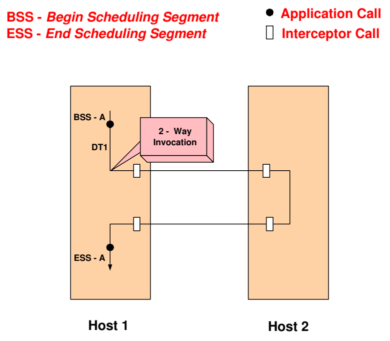

Figure 1.4 Distributable Thread Abstraction

1.3.3 Overview of Real-time CORBA 2.0

The Real-time CORBA 2.0 specification is targeted for *dynamically scheduled and provisioned* DRE systems, where the knowledge of applications that will run on the system and/or the priorities at which they execute is *not* known *a priori*. Real-time CORBA 2.0 extends version 1.0 by providing interfaces and mechanisms to plug in dynamic schedulers and interact with them. It allows applications to specify and use the scheduling disciplines and parameters that most accurately define and describe their execution and resource requirements, *e.g.*, this specification supports the following capabilities:

- It allows application developers to associate any scheduling discipline, *e.g.*, Earliest Deadline First or Maximum Urgency First, with the scheduler.
- It enables the scheduling parameters to be associated with the chosen discipline at any time.
- It provides a set of ORB/Scheduler interfaces that support the development of portable (*i.e.*, ORB-independent) schedulers.

The Real-time CORBA 2.0 specification defines the following two key capabilities that provide significant enhancements compared with version 1.0:

- **Distributable thread.** Dynamic DRE systems require more information than just priority, *i.e.*, fixed-priority propagation is not sufficient to ensure end-to-end timeliness requirements. Instead, an abstraction is required that identifies a schedulable entity and associates with it the appropriate scheduling parameters. The Real-time CORBA 2.0 specification defines a *distributable thread* abstraction that enables a thread to execute operations on objects without regard for physical node boundaries. This abstraction is the loci of execution that spans multiple nodes and scheduling segments. Figure 1.4 shows a distributable thread $DT1$ that spans two nodes Host 1 and Host 2 as a part of a two-way invocation on a remote object.
- **Scheduling service architecture.** To facilitate the association of various scheduling disciplines with a dynamic scheduler, the Real-time CORBA 2.0 specification defines a Scheduling Service interface that provides mechanisms for plugging in different schedulers. An application passes its scheduling requirements to the scheduler via these interfaces. Similarly, the ORB also interacts with the scheduler at specific scheduling points to dispatch and share scheduling information across nodes.

As discussed above, Real-time CORBA versions 1.0 and 2.0 provides mechanisms to meet the needs of a wide variety of DRE applications that possess different constraints and priorities. These differing needs, however, must be met through appropriate designs and implementations of ORBs. In particular, DRE system designers need configurable ORBs to provide a flexible range of choices to meet the needs of their particular system's functional and QoS requirements. Research and implementation experience gained while developing Real-time CORBA ORBs have yielded insights, which in turn have led to the evolution of a palette of ORB design alternatives to meet those specialized needs. Sections 1.4 and 1.5 illustrate various ways in which ORBs can offer differing advantages, providing the flexibility of different alternatives for developers of DRE applications.

1.4 TAO: C++ based Real-time CORBA middleware

1.4.1 Motivation

Traditional tools and techniques used to develop DRE software are often so specialized that they cannot adapt readily to meet new functional or QoS requirements, hardware/software technology innovations, or emerging market opportunities. Standard commercial-off-the-shelf (COTS) middleware could therefore be implemented and deployed in an efficient and dependable manner, to provide advantages of COTS middleware to DRE applications. Until recently, however, it was not feasible to develop mission-critical DRE systems using standard COTS middleware due to its inability to control key QoS properties, such as predictable latency, jitter, and throughput; scalability; dependability; and security.

Over the past decade, researchers at Washington University, St.Louis, University of California, Irvine, and Vanderbilt University have worked with hundreds of developers and companies from around the world to overcome the problems outlined above. The results has been an open-source standard CORBA Object Request Broker (ORB) called The ACE ORB (TAO) (Schmidt et al. 1998b). The remainder of this section describes TAO and summarizes its current status and impact.

1.4.2 TAO Architecture and Capabilities

TAO is a C++ ORB that is compliant with most of the features and services defined in the CORBA 3.0 specification (Obj 2002c), as well as the Real-time CORBA specification (Obj 2002b). The latest release of TAO contains the following components shown in Figure 1.5 and outlined below:

1.4.2.1 IDL Compiler

TAO's IDL compiler (Gokhale and Schmidt 1999) is based on the enhanced version of the freely available SunSoft IDL compiler. The latest CORBA 3.0 IDL-to-C++ mapping has been implemented, including the latest CORBA Component Model and POA features. The IDL compiler also supports the Object-by-Value specification and the CORBA Messaging specification, as well as asynchronous method invocations. In addition, TAO's IDL compiler can generate stubs/skeletons that can support either native C++ exceptions or the more portable `CORBA::Environment` approach. Finally, TAO's IDL compiler generates code for smart proxies that allow 3rd party applications to "plug" features into clients and portable interceptors that implement the Interceptor pattern (Schmidt et al. 2000b).

1.4.2.2 Inter-ORB Protocol Engine

TAO contains a highly optimized (Gokhale and Schmidt 1998) protocol engine that implements the CORBA 3.0 General/Internet Inter-ORB Protocol (GIOP/IIOP), version 1.0, 1.1, and 1.2. TAO can therefore interoperate seamlessly with other ORBs that use the standard IIOP protocol. TAO's protocol engine supports both the static and dynamic CORBA programming models, *i.e.*, the SII/SSI and DII/DSI, respectively. TAO also supports Dynamic Anys, which facilitate incremental demarshaling. In addition, TAO supports a pluggable protocols framework (O'Ryan et al. 2000) that enables GIOP messages to be exchanged over non-TCP transports, including shared memory, UDP unicast, UDP multicast, UNIX-domain sockets, Secure Sockets (SSL), and VME backplanes. TAO's pluggable protocols framework is important for DRE applications that require more stringent QoS protocol properties than TCP/IP provides.

1.4.2.3 ORB Core

TAO's ORB Core provides an efficient, scalable, and predictable (Schmidt et al. 2001) two-way, one-way, and reliable one-way synchronous and asynchronous communication infras-

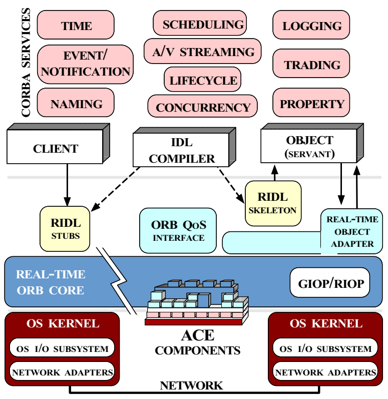

Figure 1.5: TAO ORB End System Architecture

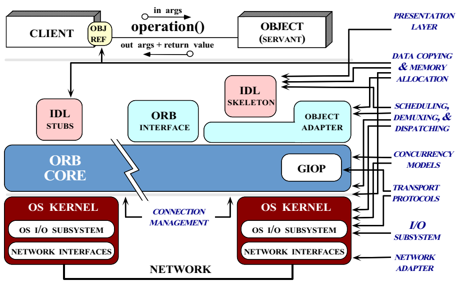

Figure 1.6 Optimizations in TAO ORB

tructure for high-performance and real-time applications. It provides the following concurrency models (Schmidt et al. 1998a): (1) reactive, (2) thread-per-connection, (3) thread pool (including the Real-time CORBA thread pool API), and (4) reactor-per-thread-priority (which is optimized for deterministic real-time systems). TAO's ORB Core is based on patterns and frameworks in ACE (Schmidt and Huston 2002a,b), which is a widely used object-oriented toolkit containing frameworks and components that implement key patterns (Schmidt et al. 2000b) for high-performance and real-time networked systems. The key patterns and ACE frameworks used in TAO include the Acceptor and Connector, Reactor, Half-Sync/Half-Async and Component/Service Configurator.

1.4.2.4 Portable Object Adapter

TAO's implementation of the CORBA Portable Object Adapter (POA) (Pyarali and Schmidt 1998) is designed using patterns that provide an extensible and highly optimized set of request demultiplexing strategies (Pyarali et al. 1999), such as perfect hashing and active demultiplexing, for objects identified with either persistent or transient object references. These strategies allow TAO's POA to provide constant-time lookup of servants based on object keys and operation names contained in CORBA requests.

1.4.2.5 Implementation and Interface Repositories

TAO's Implementation Repository automatically launches servers in response to client requests. TAO also includes an Interface Repository that provides clients and servers with run-time information about IDL interfaces and CORBA requests.

In addition, TAO provides many of the standard CORBA services, including Audio/Video Streaming Service (Mungee et al. 1999), Concurrency Service, Telecom Logging Service, Naming Service, Notification Service, Property Service, Time Service, and Trading Service. TAO also provides non-standard services that are targeted for various types of DRE application domains, including a Load Balancing service (Othman et al. 2001), a Real-time Event Service (Schmidt and O'Ryan 2002), and a Real-time Scheduling Service (Gill et al. 2001).

1.4.3 TAO Successes

The TAO project has been active since 1997. The ACE project, which provides the reusable frameworks and components upon which TAO is based, has been active since 1991. During this time, results from the ACE and TAO projects have had a significant impact on middleware researchers and practitioners, as described below:

1.4.3.1 Research Innovations

For the past decade, research on TAO has focused on optimizing the efficiency and predictability of the ORB to meet end-to-end application QoS requirements by vertically integrating middleware with OS I/O subsystems, communication protocols, and network interfaces. TAO is designed carefully using architectural, design, and optimization patterns (Schmidt et al. 2000b) that substantially improve the efficiency, predictability, and scalability of DRE systems. The optimization related research contribution of the TAO project illustrated in Figure 1.4.2.2 includes the following:

- An ORB Core that supports deterministic real-time concurrency and dispatching strategies. TAO's ORB Core concurrency models are designed to minimize context switching, synchronization, dynamic memory allocation, and data movement.
- Active demultiplexing and perfect hashing optimizations that associate client requests with target objects in constant time, regardless of the number of objects, operations, or nested POAs.
- A highly-optimized CORBA IIOP protocol engine and an IDL compiler that generates compiled stubs and skeletons. TAO's IDL compiler also implements many of the optimizations pioneered by the Flick IDL compiler (Eide et al. 1997).
- TAO can be configured to use a non-multiplexed connection model, which avoids priority inversion and behaves predictably when used with multi-rate real-time applications.
- TAO's pluggable protocols allow the support of real-time I/O subsystems (Kuhns et al. 1999) designed to minimize priority inversion interrupt overhead over high-speed ATM networks and real-time interconnects, such as VME.
- TAO's Real-time Event Service and static and dynamic Scheduling Services integrate the capabilities of TAO ORB described above. These services form the basis for next-generation real-time applications for many research and commercial projects, including ones at Boeing, Cisco, Lockheed Martin, Raytheon, Siemens, and SAIC.

1.4.3.2 Technology Transitions

Now that TAO has matured, thousands of companies around the world have used it in a wide range of domains, including aerospace, defense, telecom and datacom, medical engineering, financial services, and distributed interactive simulations. In addition, a number of companies began to support it commercially. Open-source commercial support, documentation, training, and consulting for TAO is available from PrismTech and OCI. OCI also maintains the TAO FAQ and anonymous CVS server. iCMG has developed its K2 Component Server based on the CORBA Component Model (CCM) specs (BEA Systems, et al. 1999)and TAO. The K2 Component Server is a server-side infrastructure to develop and deploy CORBA Components written in CORBA 3.0 IDL. It is based on OMG's CCM that includes a Component Model, Container Model, Packaging and Deployment, Component Implementation Framework and Inter-working with EJB 1.1.

1.5 ZEN: Real-Time Specification for Java (RTSJ) based Real-time CORBA middleware

1.5.1 Motivation

Although there have been many successful deployments of Real-time CORBA (such as those outlined in Section 1.4.3), Real-time CORBA middleware has suffered to some extent from the following limitations:

- **Lack of feature subsetting** – Early implementations of CORBA middleware incurred significant footprint overhead due to ORB designs that were implemented as a large body of monolithic code, which stymies feature subsetting and makes it hard to minimize middleware footprint.
- **Inadequate support for extensibility** – Distributed systems not only require a full range of CORBA services but also need middleware to be adaptable, *i.e.*, meet the needs of wide variety of applications developers. Current Real-time CORBA middleware designs are not designed with the aim of applicability in various domains.
- **Increased complexity** – A key barrier to the adoption of Real-time CORBA middleware arises from steep learning curve caused by the complexity of the CORBA C++ mapping (Schmidt and Vinoski 2000; ZeroC 2003). Real-time CORBA middleware should therefore be designed using programming languages that shield application developers from type errors, memory management, real-time scheduling enforcement,and steep learning curves.

Custom software development and evolution is labor-intensive and error-prone for complex DRE applications. Middleware design should therefore be simultaneously extensible, provide feature subsetting, and be easy to use, thereby minimizing the the total system acquisition and maintenance costs.

In recent years, the Java programming language has emerged as an attractive alternative for developing middleware. Java is easier to learn and program, with less inherent and accidental complexity than C++. There is also a large and growing community of Java programmers, since many schools have adopted Java as a teaching language. Java also has other desirable language features, such as strong typing, dynamic class loading, introspection, and language-level support for concurrency and synchronization. Implementation in Java could therefore provide an easy-to-use Real-time CORBA middleware tool, shielding application developers from the steep learning curve due to type errors, memory management, and real-time scheduling enforcement.

Conventional Java run-time systems and middleware have historically been unsuitable for DRE systems, 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 unpredictably long preemption latencies.

To address these problems, the Real-time Java Experts Group has defined the RTSJ (Bollella et al. 2000), 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. Real-time Java offers middleware developers a viable means of producing middleware with a simpler programming model that still offers control over memory and threading necessary for acceptable predictability.

1.5.2 ZEN Architecture and Capabilities

The ZEN ORB developed at the University of California, Irvine, leverages the lessons learned from our earlier efforts on TAO's design, implementation, optimization, and benchmarking. ZEN is a Real-time CORBA ORB implemented using Java and Real-time Java (Bollella et al. 2000), which simplifies the programming model for DRE applications. To address the challenges specific to DRE systems, the ZEN project has the following research goals:

- Provide an ORB which increases ease of use by leveraging the advantages of Java.
- Reduce middleware footprint to facilitate memory-constrained embedded systems development, yet provide a full range of CORBA services for distributed systems.
- Demonstrate the extent to which COTS languages, run-time systems, and hardware can meet the following QoS performance requirements: (1) achieve low and bounded jitter for ORB operations; (2) eliminate sources of priority inversion; and (3) allow applications to control Real-time Java features.

Our experience developing TAO taught us that achieving a small memory footprint is only possible if the architecture is designed to support this goal initially. Implementing a full-service, flexible, specification-compliant ORB with a monolithic ORB design can yield a large memory footprint, as shown in Figure 1.7. ZEN has therefore been designed using a micro-ORB architecture. Sidebar 1 discusses the advantages and disadvantages of the monolithic- and micro-ORB architectures shown in Figures 1.7 and 1.8.

In ZEN, we generalized TAO's pluggable protocol framework to other modular services within the ORB so that they need not be loaded until they are used. ZEN's flexible and extensible *micro-ORB design* (rather than monolithic-ORB design) is used for all CORBA services. In particular, we applied the following design process systematically:

1. Identify each core ORB service whose behavior may vary. Variation can depend upon (1) a user's optional choice for certain behavior and (2) which standard CORBA features are actually used.
2. Move each core ORB service out of the ORB and apply the Virtual Component pattern (Corsaro et al. 2002) to make each service pluggable dynamically.
3. Write concrete implementations of each abstract class and factories that create instances of them.

ZEN's ORB architecture is based on the concept of *layered pluggability*, as shown in Figure 1.8. Based on our earlier work with TAO, we factored eight core ORB services (object adapters, message buffer allocators, GIOP message handling, CDR Stream readers/writers,

Sidebar 1: Monolithic v/s Micro-ORB Architectures

In a *monolithic ORB architecture*, the ORB is a single component that includes all the code including those of configuration variations. This component is loaded as one executable. The advantage of a monolithic design is that its run-time performance is often efficient, it is relatively easy to code, and it can support all CORBA services. The main disadvantage, however, is that a monolithic ORB implementation can incur excessive memory footprint and therefore must rely on OS virtual memory, even if only a small subset of its features are used.

Basing an ORB architecture on patterns (Gamma et al. 1995; Schmidt et al. 2000b) can help to resolve common design forces and separate concerns effectively. For instance, the pluggable design framework can lead to a micro-ORB architecture that substantially reduces middleware footprint and increases flexibility. In a *Micro-ORB architecture* only a small ORB kernel is loaded in memory, with various components linked and loaded dynamically on demand. The advantage of this design is the significant reduction in footprint and the increase in extensibility. In particular, independent ORB components can be configured dynamically to meet the needs of different applications. The disadvantage is that dynamic linking on-demand incurs jitter, which may be undesirable for many DRE systems.

protocol transports, object resolvers, IOR parsers, and `Any` handlers) out of the ORB to reduce its memory footprint and increase its flexibility. We call the remaining portion of code the *ZEN kernel*. Moreover, each ORB service itself is decomposed into smaller pluggable components that are loaded into the ORB only when needed.

In addition to ZEN's goals of ease of use and small footprint, an additional goal is to support the predictability requirements of DRE systems, such as end-to-end priority preservation, upper bounds on latency and jitter, and bandwidth guarantees.

1.5.3 ZEN Successes

The ZEN project has been active since 1999. In a short span of time, ZEN is impacting middleware researchers and practitioners, as described below:

1.5.3.1 Research Innovations

Similar to TAO, research on ZEN has focused on optimizing the efficiency and predictability of the ORB to meet end-to-end application QoS requirements. In particular research on ZEN focuses on achieving predictability by applying optimization principle patterns and the RTSJ to Real-time CORBA as discussed below.

Applying optimization principle patterns to ZEN. At the heart of ZEN are optimization principle patterns (Pyarali et al. 1999) that improve the predictability as required by DRE applications. These optimizations are applied at the algorithmic and data structural level and are independent of the Java virtual machine (JVM). In ZEN, these strategies are applied at the following levels:

- **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 (Krishna et al. 2003),

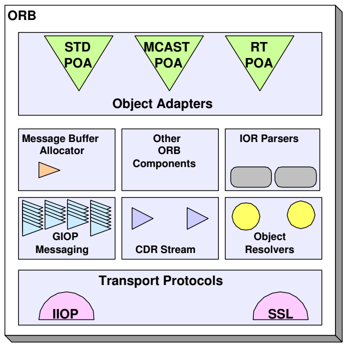

Figure 1.7 Monolithic ORB Architecture

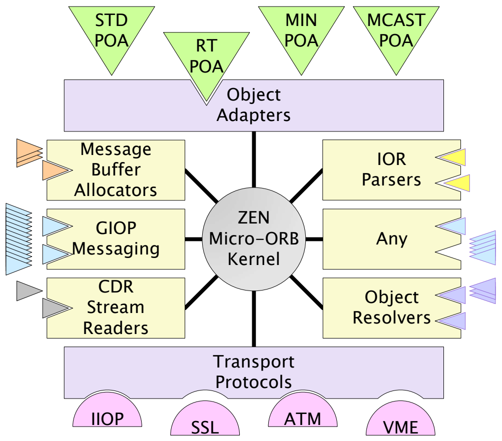

Figure 1.8 Micro-ORB Architecture

2. **Object key processing techniques** that optimize the layout and processing of object keys based on POA policies (Krishna et al. 2004),
3. **Thread-pooling techniques** that control CPU resources by bounding the number of threads created by the middleware, and
4. **Servant lookup techniques** that ensure predictable-servant to-object-id operations in the POA.

- **ORB Core layer** – Optimizations applied in this layer include the following:
  1. **Collocation optimizations**, that minimize the marshaling/de-marshaling overhead based on object/ORB location,
  2. **Buffer-allocation strategies**, that optimize the allocation and caching of buffers to minimize garbage collection, and
  3. **Reactive I/O** using Java's `nio` (Hutchins 2002) package that allows asynchronous communication.

Applying RTSJ features to ZEN. The OMG Real-time CORBA specification was adopted several years before the RTSJ was standardized. Real-time CORBA's Java mapping therefore does not use any RTSJ features. 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 improve predictability. In these optimizations, RTSJ features are directly associated within key ORB core components to enhance predictability accrued from the optimization principle patterns.

Our goal for apply RTSJ features to ZEN is to (1) comply with the Real-time CORBA specification and (2) be transparent to developers of DRE applications. Our predictability-enhancing improvements of ZEN begin by identifying the participants associated with processing a request at both the client and server sides. For each participant identified, we associate the component with non-heap regions and resolve challenges arising from this association. RTSJ features can be utilized (Krishna et al. 2004) to implement mechanisms in a

Real-time CORBA ORB, such as priority-banded thread pool lanes, to improve ORB predictability. These mechanisms can be implemented in each of the following design layers within ZEN:

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.

After the key ORB core components are allocated within scoped and immortal memory and `RealtimeThreads` are used for request/response processing, predictability will improve. Currently ZEN has been ported (Krishna et al. 2003) to both ahead of time compiled RTSJ platforms such as jRate (Corsaro and Schmidt 2002) and interpreted platforms like OVM (OVM/Consortium 2002).

1.5.3.2 Technology Transitions

The ZEN project is part of the DARPA PCES (Office n.d.) program, which provides language technology to safely and productively program and evolve cross-cutting aspects to support DRE middleware and "plug & play" avionics systems. Although ZEN's development is still in its early stages, ZEN is being used to support several other research projects. The distributed automated target recognition (ATR) project (Dudgen and Lacoss 1993) developed at MIT, uses ZEN to transmit images of identified targets to another vehicle over wireless ethernet. ZEN supports the FACET (Hunleth and Cytron 2002) real-time event channel (RTEC) implemented in RTSJ, whose design is based on that of the TAO RTEC. ZEN is also being extended to support the CORBA component model (CCM), which is designed to support DRE system development by composing well-defined components. The Cadena project (Hatcliff et al. 2003), provides an integrated environment for building and modeling CCM system. ZEN is currently being integrated with Cadena to model CCM implementations using a combination of the ZEN ORB, ZEN CCM, and FACET.

1.6 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 TAO and ZEN.

QoS middleware R&D. An increasing number of efforts are focusing on end-to-end QoS properties by integrating QoS management into standards-based middleware.

URI (Wolfe et al. 1997) 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) (Fay-Wolfe et al. 1995).

ROFES (RWTH Aachen 2002) is a Real-time CORBA implementation for embedded systems. ROFES uses a microkernel-like architecture (RWTH Aachen 2002). ROFES has been adapted to work with several different hard real-time networks, including SCI (S. Lankes and Bemmerl 2001), CAN, ATM, and an ethernet-based time-triggered protocol (S. Lankes and Jabs 2002).

The *Quality Objects* (QuO) distributed object middleware, developed at BBN Technologies (Zinky et al. 1997), is based on CORBA and provides the following support for agile wide area based applications: the run-time performance tuning and configuration through the specification of *Quality of Service (QoS) regions*, and reconfiguration strategies that allow the QuO run-time to trigger reconfiguration adaptively as system conditions change.

The *Time-triggered Message-triggered Objects* (TMO) project (Kim 1997) 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 *Kokyu* project, at Washington University St. Louis, provides a multi-paradigm strategized scheduling framework. Kokyu has been implemented within TAO's Real-Time Event Service. Kokyu enables the the configuration and empirical evaluation of multiple scheduling paradigms, including static (*e.g.*, Rate Monotonic (RMS) (Liu and Layland 1973)), dynamic (*e.g.*, earliest deadline first (EDF) (Liu and Layland 1973)) and hybrid (*e.g.*, maximum urgency first (MUF) (Stewart and Khosla 1992)) scheduling strategies.

The *Component Integrated ACE ORB* (Wang et al. 2003a) project is an implementation of the CORBA Component Model (CCM) (Obj 2002a) specification. CIAO provides a component-oriented paradigm to DRE system developers by abstracting DRE-critical systemic aspects, such as QoS requirements and real-time policies, as installable/configurable units supported by the CIAO component framework (Wang et al. 2003b).

The *Model Driven Architecture* (MDA) (Gokhale et al. 2003) adopted by the OMG is a software development paradigm that applies domain-specific modeling languages systematically to engineer DRE systems. MDA tools are being combined with QoS-enabled component middleware to address multiple QoS requirements of DRE systems in real-time (Trask 2000).

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) (TimeSys 2001), 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 *(Corsaro and Schmidt 2002)* 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 (GNU is Not Unix 2002) to provide an ahead-of-time compiled platform for RTSJ.

The *Real-Time Java for Embedded Systems* (RTJES) program (Jason Lawson 2001) 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 (Sharp 1998).

Researchers at Washington University in St. Louis are investigating automatic mechanisms (Deters et al. 2001) 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.

1.7 Concluding Remarks

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 (Alan Burns and Andy Wellings 2001) 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) defines standards that help to reduce total ownership costs of complex software systems, and (3) enhances extensibility for future application needs. In particular, Real-time CORBA has been used successfully in a broad spectrum of DRE systems, conveying the benefits of middleware to the challenging requirements of applications in domains ranging from telecommunications to aerospace, defense, and process control.

As Real-time CORBA middleware has evolved to meet the needs of DRE systems, middleware developers are designing ORB implementations that offer unique advantages and allow DRE systems developers to control the tradeoffs between different development concerns. Universal adoption of early versions of Real-time CORBA middleware has been hampered to some extent, however, by the steep learning curve of the CORBA C++ mapping and the difficulty of obtaining skilled C++ developers. Java and Real-time Java have emerged as an attractive alternative, since they are simpler to learn and have less inherent and accidental complexity.

The next generation of Real-time CORBA middleware, such as the ZEN ORB described in this chapter, is designed to reduce the difficulties of earlier middleware by combining Real-time Java with Real-time CORBA. The result is an easy-to-use, extensible, flexible, and standards-based middleware with an appropriate footprint and QoS to meet the needs of many DRE systems. As illustrated by the TAO and ZEN case studies described in this chapter, future generations of middleware will continue to evolve to meet the changing and demanding needs of DRE systems.

Bibliography

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

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

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

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

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

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

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

Douglas C. Schmidt and Frank Buschmann 2003 Patterns, Frameworks, and Middleware: Their Synergistic Relationships *International Conference on Software Engineering (ICSE)* IEEE/ACM, Portland, Oregon.

Dudgen DE and Lacoss RT 1993 An overview of automatic target recognition.. *MIT Lincon Laboratory Journal* **6**, 3–10.

Eide E, Frei K, Ford B, Lepreau J and Lindstrom G 1997 Flick: A Flexible, Optimizing IDL Compiler *Proceedings of ACM SIGPLAN '97 Conference on Programming Language Design and Implementation (PLDI)* ACM, Las Vegas, NV.

Fay-Wolfe V, Black JK, Thuraisingham B and Krupp P 1995 Real-time Method Invocations in Distributed Environments. Technical Report 95-244, University of Rhode Island, Department of Computer Science and Statistics.

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

Gill CD, Levine DL and Schmidt DC 2001 The Design and Performance of a Real-Time CORBA Scheduling Service. *Real-Time Systems, The International Journal of Time-Critical Computing Systems, special issue on Real-Time Middleware*.

GNU is Not Unix 2002 GCJ: The GNU Compiler for Java `http://gcc.gnu.org/java`.

Gokhale A and Schmidt DC 1998 Principles for Optimizing CORBA Internet Inter-ORB Protocol Performance *Hawaiian International Conference on System Sciences*.

Gokhale A and Schmidt DC 1999 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*.

Gokhale A, Schmidt DC, Natarajan B, Gray J and Wang N 2003 Model Driven Middleware In *Middleware for Communications* (ed. Mahmoud Q) Wiley and Sons New York.

Harrison TH, Levine DL and Schmidt DC 1997 The Design and Performance of a Real-time CORBA Event Service *Proceedings of OOPSLA '97*, pp. 184–199 ACM, Atlanta, GA.

Hatcliff J, Deng W, Dwyer M, Jung G and Prasad V 2003 Cadena: An Integrated Development, Analysis, and Verification Environment for Component-based Systems *Proceedings of the International Conference on Software Engineering*, Portland, OR.

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

Hunleth F and Cytron RK 2002 Footprint and feature management using aspect-oriented programming techniques *Proceedings of the joint conference on Languages, compilers and tools for embedded systems*, pp. 38–45. ACM Press.

Hutchins R 2002 *Java NIO*. O'Reilly & Associates.

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

Kim KHK 1997 Object Structures for Real-Time Systems and Simulators. *IEEE Computer* pp. 62–70.

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

Krishna AS, Schmidt DC and Klefstad R 2004 Enhancing real-time corba via real-time java *Submitted to the 24th International Conference on Distributed Computing Systems (ICDCS)* IEEE, Tokyo, Japan.

Krishna AS, Schmidt DC, Klefstad R and Corsaro A 2003 Towards predictable real-time Java object request brokers *Proceedings of the 9th Real-time/Embedded Technology and Applications Symposium (RTAS)* IEEE, Washington, DC.

Kuhns F, Schmidt DC and Levine DL 1999 The Design and Performance of a Real-time I/O Subsystem *Proceedings of the* 5^{th} *IEEE Real-Time Technology and Applications Symposium*, pp. 154–163 IEEE, Vancouver, British Columbia, Canada.

Liu C and Layland J 1973 Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment. *JACM* **20**(1), 46–61.

Morgenthal JP 1999 Microsoft COM+ Will Challenge Application Server Market www.microsoft.com/com/wpaper/complus-appserv.asp.

Mungee S, Surendran N and Schmidt DC 1999 The Design and Performance of a CORBA Audio/Video Streaming Service *Proceedings of the Hawaiian International Conference on System Sciences*.

Obj 2001 *Dynamic Scheduling Real-Time CORBA 2.0 Joint Final Submission* OMG Document orbos/2001-06-09 edn.

Obj 2002a *CORBA Components* OMG Document formal/2002-06-65 edn.

Obj 2002b *Real-time CORBA Specification* OMG Document formal/02-08-02 edn.

Obj 2002c *The Common Object Request Broker: Architecture and Specification* 3.0.2 edn.

Object Management Group 1998 *CORBA Messaging Specification* OMG Document orbos/98-05-05 edn Object Management Group.

Office DIE n.d. Program Composition for Embedded Systems (PCES) `www.darpa.mil/ixo/`.

O'Ryan C, Kuhns F, Schmidt DC, Othman O and Parsons J 2000 The Design and Performance of a Pluggable Protocols Framework for Real-time Distributed Object Computing Middleware *Proceedings of the Middleware 2000 Conference* ACM/IFIP.

Othman O, O'Ryan C and Schmidt DC 2001 Designing an Adaptive CORBA Load Balancing Service Using TAO. *IEEE Distributed Systems Online*.

OVM/Consortium 2002 OVM An Open RTSJ Compliant JVM `http://www.ovmj.org/`.

Pyarali I and Schmidt DC 1998 An Overview of the CORBA Portable Object Adapter. *ACM StandardView*.

Pyarali I, O'Ryan C, Schmidt DC, Wang N, Kachroo V and Gokhale A 1999 Applying Optimization Patterns to the Design of Real-time ORBs *Proceedings of the* 5^{th} *Conference on Object-Oriented Technologies and Systems*, pp. 145–159 USENIX, San Diego, CA.

RWTH Aachen 2002 ROFES http://www.rofes.de.

S. Lankes MP and Bemmerl T 2001 Design and Implementation of a SCI-based Real-Time CORBA *4th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC 2001)* IEEE, Magdeburg, Germany.

S. Lankes MR and Jabs A 2002 A Time-Triggered Ethernet Protocol for Real-Time CORBA *5th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC 2002)* IEEE, Washington, DC.

Schmidt DC and Huston SD 2002a *C++ Network Programming, Volume 1: Mastering Complexity with ACE and Patterns*. Addison-Wesley, Boston.

Schmidt DC and Huston SD 2002b *C++ Network Programming, Volume 2: Systematic Reuse with ACE and Frameworks*. Addison-Wesley, Reading, Massachusetts.

Schmidt DC and O'Ryan C 2002 Patterns and Performance of Real-time Publisher/Subscriber Architectures. *Journal of Systems and Software, Special Issue on Software Architecture - Engineering Quality Attributes*.

Schmidt DC and Vinoski S 2000 The History of the OMG C++ Mapping. *C/C++ Users Journal*.

Schmidt DC, Kachroo V, Krishnamurthy Y and Kuhns F 2000a Applying QoS-enabled Distributed Object Computing Middleware to Next-generation Distributed Applications. *IEEE Communications Magazine* **38**(10), 112–123.

Schmidt DC, Kuhns F, Bector R and Levine DL 1998a The Design and Performance of an I/O Subsystem for Real-time ORB Endsystem Middleware. *submitted to the International Journal of Time-Critical Computing Systems, special issue on Real-Time Middleware*.

Schmidt DC, Levine DL and Mungee S 1998b The Design and Performance of Real-Time Object Request Brokers. *Computer Communications* **21**(4), 294–324.

Schmidt DC, Mungee S, Flores-Gaitan S and Gokhale A 2001 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*.

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

Sharp DC 1998 Reducing Avionics Software Cost Through Component Based Product Line Development *Proceedings of the 10th Annual Software Technology Conference*.

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

Stewart DB and Khosla PK 1992 Real-Time Scheduling of Sensor-Based Control Systems In *Real-Time Programming* (ed. Halang W and Ramamritham K) Pergamon Press Tarrytown, NY.

TimeSys 2001 Real-Time Specification for Java Reference Implementation `www.timesys.com/rtj`.

Trask B 2000 A Case Study on the Application of CORBA Products and Concepts to an Actual Real-Time Embedded System *OMG's First Workshop On Real-Time & Embedded Distributed Object Computing* Object Management Group, Washington, D.C.

Wang N, Schmidt DC, Gokhale A, Gill CD, Natarajan B, Rodrigues C, Loyall JP and Schantz RE 2003a Total Quality of Service Provisioning in Middleware and Applications. *The Journal of Microprocessors and Microsystems* **27**(2), 45–54.

Wang N, Schmidt DC, Gokhale A, Rodrigues C, Natarajan B, Loyall JP, Schantz RE and Gill CD 2003b QoS-enabled Middleware In *Middleware for Communications* (ed. Mahmoud Q) Wiley and Sons New York.

Wolfe VF, DiPippo LC, Ginis R, Squadrito M, Wohlever S, Zykh I and Johnston R 1997 Real-Time CORBA *Proceedings of the Third IEEE Real-Time Technology and Applications Symposium*, Montréal, Canada.

Wollrath A, Riggs R and Waldo J 1996 A Distributed Object Model for the Java System. *USENIX Computing Systems*.

Zahavi R and Linthicum DS 1999 *Enterprise Application Integration with CORBA Component & Web-Based Solutions*. John Wiley & Sons, New York.

ZeroC I 2003 The Internet Communications Engine^{TM} `www.zeroc.com/ice.html`.

Zinky JA, Bakken DE and Schantz R 1997 Architectural Support for Quality of Service for CORBA Objects. *Theory and Practice of Object Systems* **3**(1), 1–20.