Towards Highly Configurable Real-time Object Request Brokers

Raymond Klefstad, Douglas C. Schmidt, and Carlos O'Ryan
{klefstad, schmidt, coryan}@doc.ece.uci.edu
Electrical and Computer Engineering Dept.
University of California, Irvine, CA 92697, USA*

This paper appeared in the Proceedings of IEEE International Symposium on Object-Oriented Real-time Distributed Computing (ISORC), Washington DC, April 29 – May 1, 2002.

Abstract

This paper discusses the software architecture of a Real-time CORBA object request broker (ORB) called ZEN, written in Real-time Java, which is designed to eliminate common sources of overhead and non-determinism in ORB implementations. We illustrate how ZEN can be configured to select the minimal set of components used by an application. Our experience with ZEN indicates that combining Real-time Java with Real-time CORBA is a major step forward towards simplifying the development and maintenance of distributed middleware and applications with stringent quality of service requirements.

Keywords: Distributed Real-time and Embedded Systems, Real-time CORBA, Real-time Java.

1 Introduction to Distributed, Real-time, Embedded Systems

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), tele-medicine (*e.g.*, remote surgery), manufacturing 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, are fast 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 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.

Because of these challenges, developers repeatedly rediscover core concepts and reinvent 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 [1], COM+ [2], Java RMI [3], and SOAP/.NET [4], have emerged to reduce the complexity of developing distributed applications. DOC middleware simplifies application development for distributed systems by off-loading the tedious and error-prone aspects of distributed computing from application developers to middleware developers. It has been used successfully in large-scale

*This work was funded in part by AFOSR grant F49620-00-1-0330, ATD, DARPA ITO, SAIC, and Siemens.

business systems where scalability, evolvability, and interoperability are essential for success.

Real-time CORBA [5] is a rapidly maturing DOC middleware technology standardized by the OMG that can simplify many challenges for DRE applications, just as CORBA has for large-scale business systems. Real-time CORBA is designed for applications with hard real-time requirements, such as avionics mission computing [6], as well as those with stringent soft real-time requirements, such as telecommunication call processing and streaming video [7].

This paper makes the following contributions to the design of Real-time CORBA middleware to address key challenges of developing DRE systems:

1. It describes the design of the ZEN ORB, an open-source Real-time CORBA ORB implemented in Real-time Java [8] to simplify the programming model for DRE applications. ZEN is inspired by many of the patterns, techniques, and lessons learned in The ACE ORB (TAO) [6], an open-source implementation of Real-time CORBA written in C++.
2. It explains how ZEN uses patterns [9, 10] to automatically minimize the memory footprint of DRE middleware customized for each application.
3. It compares ZEN's novel micro-ORB design and implementation with the design and implementation of traditional monolithic ORB architectures.

The remainder of this paper is organized as follows: Section 2 gives an overview of Real-time CORBA and Real-time Java; Section 3 describes how ZEN is designed to improve flexibility and minimize memory footprint; and Section 4 presents concluding remarks.

2 Overview of Real-time CORBA and Real-time Java

2.1 Real-time CORBA

CORBA is distribution middleware that provides run-time support to automate many distributed computing tasks, such as connection management, object (de)marshaling[1], object demultiplexing, language and OS independence, load balancing, fault-tolerance, and security. Real-time CORBA is integrated with the CORBA 2.5 specification [1] and adds QoS control capabilities to regular CORBA to

- Improve application predictability by bounding priority inversions and
- Manage system resources end-to-end.

[1]We use the term "(de)marshal" to mean marshal and/or demarshal.

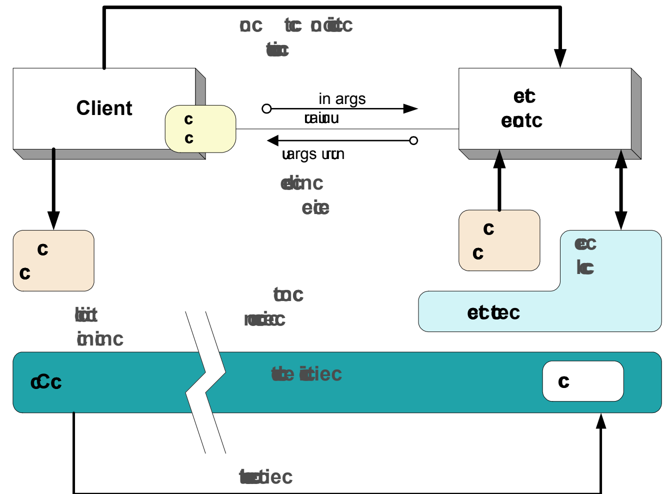

Figure 1: Real-time CORBA Features

Figure 1 illustrates the standard features that Real-time CORBA provides for DRE applications so that they can configure and control the following system resources:

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

The Real-time CORBA specification addresses some — though by no means all — important DRE application development challenges. Its primary focus is on "fixed-priority" real-time applications [11], where priorities are assigned statically to tasks, and the task with the highest priority always runs. In many new and planned DRE applications, however, static task prioritization is often not possible, since task workloads and their priorities are not known until run-time [12].

Although the Real-time CORBA specification was integrated into the OMG standard several years ago, it has not yet been adopted universally for DRE applications, due to its

- **Steep learning curve**, caused largely by the complexity of its C++ mapping, and
- **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.

2.2 Real-time Java

Most current implementations of Real-time CORBA are available only in C++ or Ada. Finding and retaining experienced developers trained in these languages is hard, however. The Java programming language is an attractive alternative for the following reasons:

- Java has a large and rapidly growing programmer base and is taught in many universities.
- Java is simpler than C++ or Ada; therefore, programmers experienced in those languages can learn it easily.
- Java has a powerful, portable standard library that can reduce programming time and costs.
- Java offloads many tedious and error-prone programming details from developers into the language run-time system.
- Java has desirable language features, such as strong typing, dynamic class loading, and reflection/introspection.
- Java defines portable support for concurrency and synchronization.
- Java's bytecode representation is more compact than native code, reducing the memory required for embedded systems.
- Many Java virtual machines (JVMs) support "lazy class-loading and linking," in which classes are loaded into memory and bound to an interface only upon first use.
- Java can make ORB and application development easier and faster, therefore, due to its simplicity and improved portability.

Conventional Java implementations are unsuitable for developing real-time systems, however, because they do not allow fine-grained control over memory management, nor do they enforce thread priorities with sufficient precision. To address these problems, the Real-time Java Experts Group has defined the Real-time Specification for Java (RTSJ) [8], which provides the following capabilities without modifying the Java programming language itself:

- New memory management models that can be used in lieu of garbage collection, which can cause significant non-determinism in real-time systems.
- Access to raw physical memory, which is required for many embedded systems.
- A higher resolution time granularity suitable for real-time systems.
- Stronger guarantees on thread semantics than regular Java: the highest priority runnable thread is always run.

The RTSJ therefore retains the advantages of regular Java, while improving language semantics and features for programming real-time systems. The RTSJ does not, however, include any facilities for distributed applications.[2]

[2]The JSR-50 effort [13] is attempting to define a Distributed Real-time Specification for Java.

3 An Overview of the ZEN Real-time ORB

ZEN is a Real-time CORBA ORB implemented using Real-time Java, thereby combining the benefits of these two standard technologies. This section presents the research challenges and goals addressed by the ZEN project, followed by an overview of the design process and architecture of the ZEN ORB.

3.1 ZEN Research Goals

Due to constraints on weight, power consumption, memory footprint, and performance, the development techniques for DRE application software have lagged behind those used for mainstream desktop and enterprise software. As a result, DRE applications are costly to develop, maintain, and evolve. Moreover, they are often so specialized that they cannot adapt readily to meet new functional or QoS requirements, hardware/software technology innovations, or market opportunities.

Programming DRE applications is hard also because QoS properties must be supported along with the application software and distributed computing middleware functionality. DRE applications have historically been custom-programmed to implement these QoS properties. Unfortunately, this tedious and error-prone manual development process has not adequately addressed the following challenges:

- **Isolating DRE application development from the details of multiple platforms and varying operational contexts**. Modern DRE applications must invest an ever-increasing proportion of functionality in software. Rapidly emerging technologies and flexibility required for diverse operational contexts force deployment of multiple versions of software on various platforms, while simultaneously preserving key properties, such as real-time response and end-to-end priority preservation.
- **Reducing total ownership costs**. Custom software development and evolution is labor-intensive and error-prone for complex DRE applications, such as fly-by-wire aircraft or autonomous vehicle systems, and can represent a substantial amount of total system acquisition and maintenance costs.
- **Sheltering the application development from obsolescence trends**. Incommensurate lifetimes between long-lived DRE applications (20 years or more) and commercial off-the-shelf (COTS) platforms and tools (2–5 years) lead to pervasive software obsolescence and multiply the total ownership costs by requiring periodic software re-development and COTS refresh.

While some aspects of these challenges have been addressed in developing mainstream distributed systems, relatively little has been done to meet these challenges for DRE systems. To address these challenges, therefore, the research goals of the ZEN project are as follows:

- Provide a full range of CORBA services for distributed systems, to meet the needs of a wide variety of application developers.
- Demonstrate the extent to which COTS languages, run-time systems, and hardware can meet the following QoS requirements:
  - Achieve low and bounded jitter for ORB operations
  - Eliminate sources of priority inversion
  - Allow applications to control Real-time Java features
  - Achieve low startup latency
- Reduce middleware footprint to enable memory-constrained embedded systems development.
- Achieve satisfactory level of throughput and scalability.
- Make the ORB easier for application developers to configure and maintain.
- Make the ORB easily extensible.
- Allow both *static* and *dynamic* configuration, to allow the application developer to choose a tradeoff between maximal efficiency and flexibility. One of the chief research challenges associated with supporting dynamic configuration is to minimize latency and to ensure satisfaction of end-to-end deadlines.

3.2 Overview of the ZEN Design Process

3.2.1 Generations of ORB Designs

Our work on ZEN has leveraged the lessons learned from our earlier efforts on TAO's design, implementation, optimization, and benchmarking. We have identified the following five generations of ORB designs, ranging from the first implementations to an ideal ORB for DRE systems.

1. **Static monolithic ORB**, in which all code is loaded in one executable, including the code for configuration variations. The original implementation of TAO [6] was designed this way, as are many other non-real-time CORBA ORBs. The advantage of this design is that it is efficient, it is relatively easy to code, and it can support all CORBA services. The obvious disadvantage is that a monolithic ORB implementation results in an excessive memory footprint, even if only a small subset of its features are used. Moreover, the footprint grows with each extension, such as adding support for a new protocol, and extensibility is hard.
2. **Monolithic ORB with compile-time configuration flags**, in which static mechanisms/tools, such as conditional compilation and smart static linkers, allow a variety of different configuration options. The second generation of TAO [14] was designed this way. Compared to a static, monolithic ORB, the advantage is a reduced footprint, since the preprocessor can eliminate unneeded code. The disadvantage, however, is that application and ORB developers must face scores of configuration options, and must select the appropriate ones to achieve particular footprint and performance needs. It is therefore much harder to code and maintain this type of ORB, due to the accidental complexities associated with conditional compilation [15].
3. **Dynamic micro-ORB**, in which only a small ORB kernel is loaded in memory, with various components linked and loaded dynamically on demand. Portions of the current third generation of TAO are designed this way (based on the Component Configurator [9] and Virtual Component [10] patterns), as is the initial implementation of ZEN presented in Section 3.3. 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. Dynamic configuration greatly reduces the myriad of configuration options facing application developers using second-generation ORBs. With dynamic configuration, application developers select only a few — rather than scores of — configuration options. The disadvantage is that dynamic linking on demand produces a potential source of jitter, which can be unacceptable for real-time systems. Moreover, dynamic linking may not be available or appropriate for some embedded systems.
4. **Dynamic reflective micro-ORB**, in which the ORB builds a configuration description for each application, based on information derived from the application's run-time execution history. This configuration description can be used to configure the ORB either adaptively or upon ORB initialization for future invocations of the application. The dynamicTAO [16] project and future versions of ZEN use this approach. The advantage of this design is that it can achieve a near-minimal footprint automatically. It can also eliminate jitter from on-demand class loading by pushing it into initialization. The disadvantage, again, is that dynamic linking may not be available or appropriate for some embedded systems.
5. **Static reflective micro-ORB**, in which the ORB uses the configuration description built by the dynamic reflective micro-ORB to generate the source code for a new custom ORB containing only the necessary or desired components. The LegORB [17] project and future versions

of ZEN use this approach. Its advantage is that it is fast, small, custom, and easy for application developers to use. Its disadvantage is that the reflective technology needed to perform automatic customization is still largely an open research issue.

Based on the taxonomy presented above, we are building ZEN in the following three stages:

1. We first design ZEN to reduce the footprint of a non-real-time Java ORB, using dynamic micro-ORB configuration corresponding to the third-generation ORB described above.
2. We next design ZEN to address real-time requirements, using dynamic reflective micro-ORB configuration corresponding to the fourth-generation ORB described above, and implementing Real-time CORBA features using Real-time Java.
3. Finally, we design ZEN to refine its own footprint and real-time performance, using static reflective micro-ORB configuration corresponding to the fifth-generation ORB described above. Here we use reflective configuration information and aspect-oriented programming (AOP) [18] techniques to generate statically a custom, small-footprint, real-time ORB.

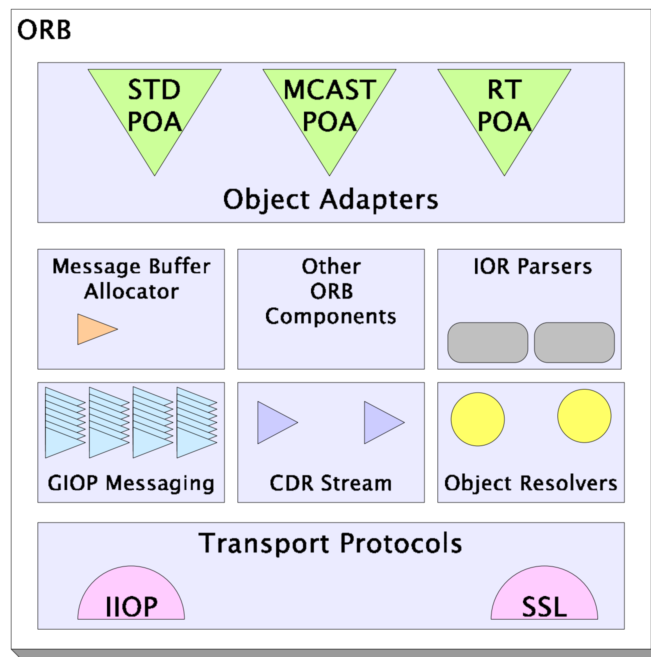

Figure 2: Monolithic ORB Architecture

The remainder of this section discusses the first stage of ZEN's design, since it is the most mature.

3.2.2 Micro-ORB versus Monolithic-ORB Designs

Our experience building TAO taught us the following lessons that we applied to the design of ZEN:

- Implementing a full-service, flexible, specification-compliant ORB can yield a monolithic ORB implementation with a large memory footprint, as shown in Figure 2.
- Basing the ORB architecture on patterns can resolve common design forces and separate concerns effectively [9]. For example, using a pluggable design framework based on TAO's pluggable protocol framework [19] can substantially reduce the middleware footprint.
- Achieving a small footprint is possible only if the architecture is initially designed to achieve it. It is much harder to reduce footprint in later stages of design.

As stated earlier, minimizing footprint is critical for memory-constrained DRE applications. Therefore, in the first stage of ZEN's design, we focused on minimizing its footprint. We generalized TAO's pluggable protocol to other modular services within the ORB, so that they need not be loaded until they are used. ZEN's micro-kernel architecture is also based on patterns that have been used to develop micro-kernel operating systems [20]. ZEN's therefore uses flexible, extensible *micro-ORB design*, rather than a monolithic-ORB design 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 on (1) a user's optional choice for certain behavior and (2) which standard CORBA features are actually used.
2. Move each core ORB service, such as the object adapter, protocol transport, and `Any` data type handling, out of the ORB and apply the Virtual Component pattern [10] to make each service pluggable dynamically.
3. Write concrete implementations of each abstract class and factories that create instances of them.
4. Extend each family of factories and concrete classes to support alternative features.
5. Minimize penalty for not using real-time features.
6. Optimize common use cases, while still ensuring that all operations are predictable, by bounding worst-case execution time.

3.3 ZEN's Pluggable ORB Architecture

ZEN's ORB architecture is based on the concept of *layered pluggability*, as shown in Figure 3. 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, protocol transports, object resolvers, IOR parsers, and `Any` handlers) out of the ORB to

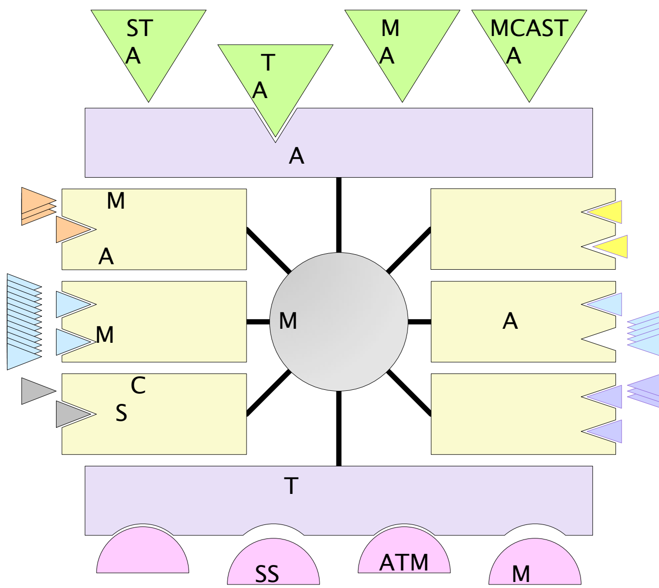

Figure 3: Micro-ORB Architecture of ZEN

reduce its memory footprint and increase its flexibility. We call the remaining portion of code the *ZEN kernel*.

Each ORB service itself is decomposed into smaller pluggable components that can be loaded into the ORB only when needed. This pluggable design makes ZEN a good research platform, because alternative implementations of various ORB components can be plugged in and profiled with standard benchmarks to determine their utility. Our future work will evaluate the performance of alternative implementations of core ORB services.

The remainder of this section describes how we designed the eight core ORB services identified as candidates for application of the Virtual Component pattern. For each core ORB service, we

1. Outline the key characteristics of the particular core ORB service to be factored out of the ORB
2. Discuss the problems encountered when implementing this service in a monolithic ORB and
3. Explain how our solution uses the Virtual Component pattern to reduce memory footprint.

3.3.1 Pluggable GIOP Message Handling

Context. The General Inter-ORB Protocol (GIOP) defines the standard messages that may be sent between CORBA-compliant ORBs. There are eight different types of GIOP messages, each with a unique format. Implementation of each GIOP message handler requires two methods, one to marshal and another to demarshal a particular type of message.

Three versions of GIOP messages (versions 1.0, 1.1, and 1.2) have been defined, with a new version being standardized by the OMG. Thus, there are 8 (different types) $\times$ 2 (marshal/demarshal methods) $\times$ 3 (different versions) = 48 methods required to handle each possible GIOP message for each possible version.[3]

The majority of client/server interactions are simple, requiring only a few of these methods. For example, a pure server will receive requests and send replies, and thus requires only a request reader and a reply writer. Conversely, a pure client will send requests and receive replies, requiring only a request writer and a reply reader. Peers typically use up to four methods to read and write both requests and replies. In addition, many applications use only one version of GIOP. However, clients, servers, and peers must be prepared to handle all 48 possible messages from various versions.

Problems with Monolithic-ORB Designs. Monolithic-ORBs contain code to handle all the possible GIOP messages and versions. Separate classes may be defined to handle each type of message, while `switch` statements inside each (de)marshal method may handle the various versions. This design has two major drawbacks:

1. It incurs non-trivial amounts of space overhead for all the methods, even if they are not used.
2. It is hard to modify the ORB to handle a new GIOP version because many class definitions must be modified, recompiled, and relinked.

Micro-ORB Design Solution in ZEN. GIOP message (de)marshaling is a rich source of footprint reduction in ZEN, as shown in Figure 4. Only a small number of the 48 possible

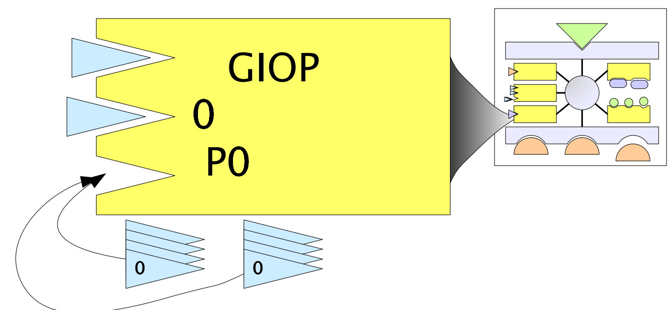

Figure 4: Pluggable GIOP Readers

(de)marshaling methods need to be loaded at a time. We have two different variants of the Virtual Component pattern:

1. *Fine-grain*, which uses a separate class for each of the 48 possible methods, loading only individual methods as needed, and caching them for faster use after the initial loading and

[3] Not all messages change with each version.

2. *Client/server pairing*, which groups complementary methods into a single class based on their likely use. The advantage of this approach is that the complementary method that will later be needed is already loaded.

Both designs use the same conventions in naming classes, enabling automatic construction from the particular use combination. For example, classes containing the methods for (de)marshaling are named by combining the message type, version number, and operation type (read or write). In particular, class `ReplyReader1_0` would contain one method to demarshal a GIOP 1.0 reply.

If a client sends a GIOP 1.0 request using the fine-grain model, only class `RequestWriter1_0` will be loaded to write the request. This approach is advantageous in situations requiring only one method, such as a one-way request, since only the `RequestWriter1_0()` method is loaded. In the more typical two-way invocation, however, the client must also load class `ReplyReader1_0` to demarshal the reply message, using its read method to process the message appropriately.

The client/server pairing model implementation handles the common two-way invocation by grouping methods that are often used together. Since servers typically read requests and write replies, we group the methods `RequestReader1_0()` and `ReplyWriter1_0()` according to GIOP version. Similarly, since clients typically write requests and read replies, these two methods are also grouped, by version. Thus, a pure client using one GIOP version needs to load only one class containing both client-oriented methods.

Modifying ZEN to support new versions of GIOP is straightforward. New classes are simply added (containing either one method for fine grain, or two methods for client/server pairing), following the naming conventions, for each message that changes. No existing classes require any changes.

CORBA applications that use either of these micro-ORB designs need not know beforehand that their behavior will be server-like or client-like, or what messages they will need. Instead, the necessary methods are loaded on demand depending on whether the program is a client, server, or peer. If the behavior of a particular program is known beforehand, however, the necessary classes may be pre-loaded at initialization time to eliminate any delays from lazy class loading.

3.3.2 Pluggable Object Adapters

Context. An object adapter maps client requests to the appropriate servant in a CORBA server. There are different types of object adapters, such as

1. The Standard Portable Object Adapter (POA), which offers a full interface of functionality to the application programmer and is targeted to general applications which are not real-time,
2. Minimum POA, which implements a smaller interface than the standard POA, intended to help reduce the memory footprint of an object adapter, and
3. Real-time POA, which adds methods to the standard POA to allow the application more control over threading and memory management. It must also ensure all operations are predictable.

In addition, new object adapters are being developed. We, in fact, are currently developing an object adapter specification to support filtering of multicast requests only to subscribed objects.

Problems with Monolithic-ORB Designs. An object adapter is necessary only in a server application. Monolithic-ORB designs contain an object adapter as part of the ORB, however, even for pure clients that do not require this functionality. Therefore, if multiple types of object adapters are supported, the code to handle each type may be loaded, whether used or not.

Micro-ORB Design Solution in ZEN. Using the Virtual Component pattern, ZEN loads only portions of an object adapter, only when object adapter services are needed, as shown in Figure 5. Pure clients have no object adapter, while pure servers load only the portions of the appropriate POA when needed. ZEN provides both a standard POA and a real-time POA. At most one of these object adapters is added into the ORB, only if the application plays the role of server. Application developers can choose which POA is used, RT or Standard, as part of the customization configuration.

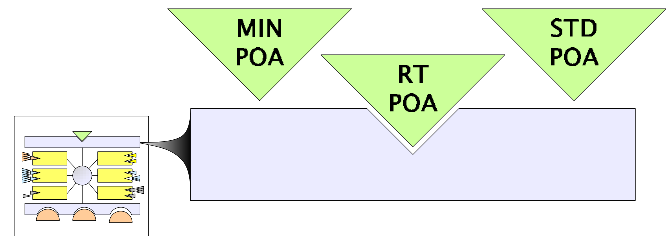

Figure 5: Pluggable Object Adapters

This pluggable design will also facilitate addition of new object adapters, such as the multicast object adapter, as they are standardized.

3.3.3 Pluggable Transport Protocols

Context. GIOP can run over many protocol transports, such as TCP/IP, shared memory, UNIX-domain sockets, and SSL. For a single protocol, there are roughly five different classes to implement it, each containing a few methods:

- Client-oriented classes, *e.g.*, connector, address, and transport, and

- Server-oriented classes, *e.g.*, acceptor, reactor, address, and transport.

There may therefore be 20 to 30 classes needed to handle the most common protocols.

Problems with Monolithic-ORB Designs. Some monolithic-ORB designs support all possible, or a large number of, protocols, with all code loaded and the particular methods necessary selected through `if` or `switch` statements. With this approach, the resident code is large and increases with each new protocol supported by the ORB.

An alternative monolithic-ORB design chooses only one protocol to support, typically TCP/IP. With this approach, the resident code is smaller, but the ORB then lacks the ability to handle other protocols, which is problematic for DRE applications that need to run over non-TCP/IP backplanes and real-time interconnects.

Micro-ORB Design Solution in ZEN. One micro-ORB approach, first implemented in TAO [19], reduces resident code by loading only the classes necessary for one protocol at a time. TAO loads all five classes for that protocol, however, regardless of whether it needs the client-oriented classes or the server-oriented classes.

The micro-ORB approach used in ZEN similarly allows one (or more) desired protocol(s) to be loaded, as shown in Figure 6. Only the required sub-classes are loaded dynamically

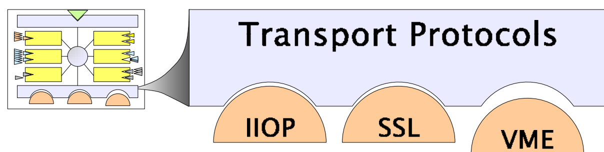

Figure 6: Pluggable Protocols

when the factory is loaded. For example, both clients and servers require the appropriate transport and address classes. Additionally, a pure server requires an acceptor and a reactor, while a pure client requires a connector.

3.3.4 Pluggable CDR Stream Reader/Writer

Context. CORBA ORBs must handle diverse endsystem instruction sets, where byte order may vary between clients and servers. CORBA defines Character Data Representation (CDR) input and output streams to allow (de)marshaling of multi-byte data objects to or from transport byte streams. A class `CDRInputStream` contains methods such as `readLong()`, `readShort()`, and `readByte()`. A `CDROutputStream` will contain methods such as `writeLong()`, `writeShort()`, and `writeByte()`. The input stream assembles the bytes properly according to a boolean flag contained within a message. Since JVMs represent all data in big-endian, all `CDROutputStreams` marshaled by a Java ORB, such as ZEN, will be in big-endian, even if the native hardware byte order is little-endian. However, since Java ORBs may need to interact with non-Java ORBs running on little-endian endsystems, they must be able to demarshal messages received from little-endian endsystems.

Problems with Monolithic-ORB Design. Monolithic ORBs define one class for `CDRInputStream` containing a method to handle each possible data type, *e.g.*, `readDouble()`, `readLong()`, `readShort()`, and `readByte()`. Each of these methods in turn uses an `if` statement to test the endian order of the stream being demarshaled, to assemble multi-byte entities, such as integers and floating points, into the correct order, as follows:

```
int readLong() {
  if (byteOrder == littleEndian)
    // return the combined four bytes
    // in little-endian format;
  else // byte order is big-endian
    // return the combined four bytes
    // in big-endian format;
}
```

Not only is this approach inconsistent with object-oriented programming techniques, but also it creates excessive footprint, since each method must handle both cases, although only one is needed at a time. Moreover, many applications, particularly Java-to-Java or those running on homogeneous hardware, will use the same byte order. In such cases, it is particularly undesirable to increase the memory footprint by having both byte-order versions in memory.

Micro-ORB Design Solution in ZEN. We applied the Virtual Component pattern to split the single `CDRInputStream` class into two derived classes: one that handles only big-endian messages, and another that handles only little-endian messages. Each method in the class for little-endian has the code in the `true` case, as follows:

```
int readLong() {
  // return the combined four bytes
  // in little-endian format;
}
```

Conversely, each method in the class for big-endian has just the code for the `false` case, as follows:

```
int readLong() {
  // return the combined four bytes
  // in big-endian format;
}
```

When a GIOP message is received, only the appropriate `CDRInputStream` matching the endian of the received message is loaded, if it is not already cached. When a big-endian

message is received, for instance, no little-endian conversion code will be loaded on either side of the communication, as shown in Figure 7. Instead of re-executing the conditional

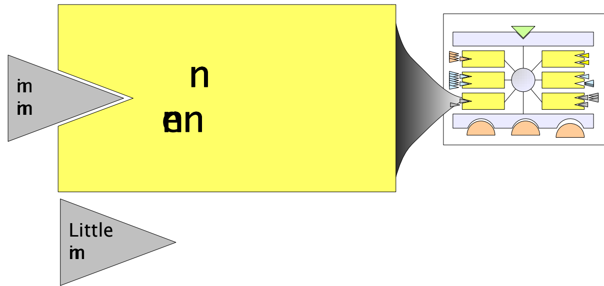

Figure 7: Pluggable CDR reader

statement each time a multi-byte primitive is read from a message, a single conditional is executed once, when the message is received, to determine the set of methods needed to demarshal the message.

3.3.5 Pluggable Any Handlers

Context. The `Any` data type is useful for generic services, because it can be used to hold any data type, such as arrays or primitives. `Anys` are used by many of the CORBA Object Services (COS) [21], such as the Trading Service, the Event Service, the Notification Service, and the Security Service. Each `Any` is preceded with a type code to allow interpretive manipulation of the values it contains. Many DRE applications do not require the `Any` data type, however, and methods to support `Anys` consume significant space.

Problems with Monolithic-ORB Designs. Monolithic ORBs include extensive code to support `Anys`. This code is extensive because it must include methods to read, to write, to (de)marshal, and to insert and extract objects from each of the primitives, structs, unions, arrays, user-defined types, and sequences of all possible `Anys`.

Micro-ORB Design Solution in ZEN. The methods to support `Anys` are good candidates for removal from the ORB kernel for further footprint savings. To minimize the default footprint of DRE applications, ZEN maintains only a minimal proxy object representing the `AnyReader` and `AnyWriter` objects in the ORB kernel, until an application tries to read or write an `Any`, as shown in Figure 8. At that time, the proxy loads the class for the appropriate specific methods needed. The same `Any` method is commonly reused, since arrays and sequences contain elements of the same type.

3.3.6 Pluggable IOR Parsers

Context. Interoperable ORB References (IORs) are CORBA's powerful object pointers. Unlike memory pointers,

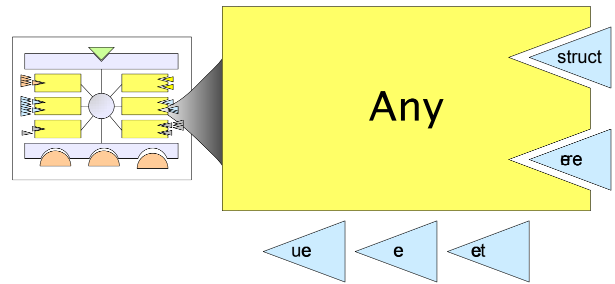

Figure 8: Pluggable Any

IORs can point to a specific servant (in the role of an object) on a remote host. CORBA supports a variety of formats for IORs, to allow object references to come from files, web pages, location services, or other sources. While the "`IOR:`" format is the most common, a CORBA-compliant ORB must support a variety of other IOR formats, such as "`FILE:`," "`HTTP:`," "`CORBALOC`," and "`FTP:`". IOR formats contain different information in different formats, and therefore require different methods for parsing and handling. As shown

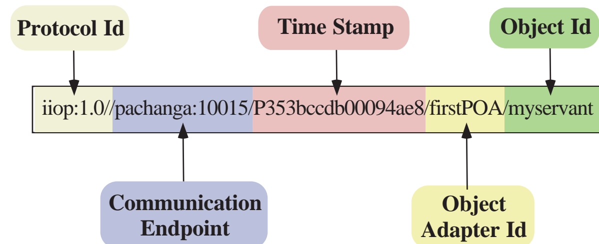

Figure 9: Example IOR

in Figure 9, the format of an "`IOR:`" IOR may contain information, such as protocol name, communication endpoint (*e.g.*, host and port), object adapter name, and object name, as well as other ORB-specific information to optimize servant lookups.

Problems with Monolithic-ORB Designs. In a monolithic-ORB design, matching the IOR format string, whether by table lookup or by cascaded `if` statements, requires that the code to parse and handle every format be loaded, even if it is not used.

Micro-ORB Design Solution in ZEN. ZEN uses the Virtual Component pattern to (1) define an interface that parses and handles IORs and (2) then derive separate class strategies to handle each specific IOR format, as shown in Figure 10. When a particular IOR format is encountered, ZEN loads only the class specialized to handle that format. Future repeated uses of that format will be faster, after the class has been loaded. This implementation saves a small amount of code size overhead and eliminates an extra test for each demarshaling operation.

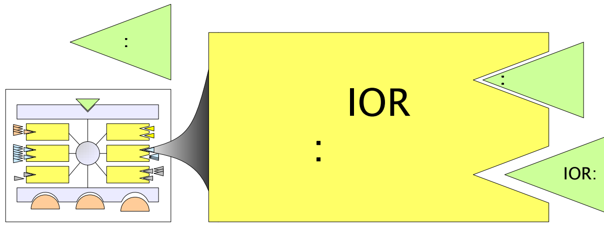

Figure 10: Pluggable IOR Parsers

3.3.7 Pluggable Object Resolvers

Context. CORBA applications use the `ORB::resolve_initial_references()` method to obtain references to ORB objects, such as the RootPOA, or to CORBA Service objects, such as the Naming Service or Event Channel. The number of objects that can be obtained through calls to `resolve_initial_references()` is large and increasing with each new version of the CORBA specification.

Problems with Monolithic-ORB Designs. Monolithic-ORB designs use a series of cascaded `if` statements in `resolve_initial_references()` to match the string name parameter to the code that handles each particular name value. This design is not easily extensible, because the ORB must be modified to handle each additional name value.

Micro-ORB Design Solution in ZEN. We apply the Virtual Component pattern to define pluggable object resolvers, as shown in Figure 11. At the core of this mechanism is an abstract base class with a factory method that takes a string and returns a reference to a CORBA Object. We then derive classes for each specific name value that may be resolved. ZEN uses a naming convention for the classes, so that the name being resolved matches the name of the class that can resolve it. For example, the object resolver for the name "RootPOA" is named class `RootPOAResolver`. Upon receiving a request to resolve an initial reference, the class for resolving that name value can be loaded as needed. This design enhances extensibility as new object names are added.

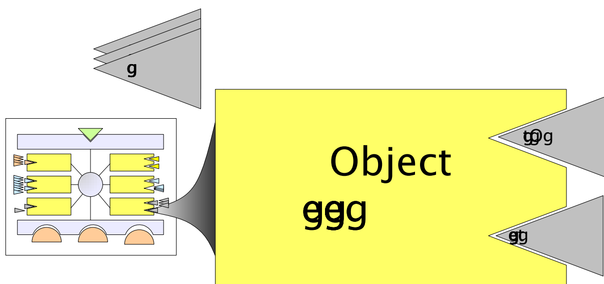

Figure 11: Pluggable Object Resolvers

3.3.8 Pluggable Message Buffer Allocators

Context. An ORB must provide a message buffer allocator to ensure efficient interprocess communication and to avoid unnecessary garbage collection. While general-purpose dynamic storage allocation algorithms are well understood [22], it is hard to determine which specific algorithm is optimal for allocating ORB message buffers for a particular DRE application. The optimal algorithm may vary, depending on an application's usage patterns, and is not intuitively obvious. Middleware, therefore, should allow the application developer the flexibility to choose different algorithms in different situations, based on their empirical performance.

Problems with Monolithic-ORB Designs. To provide the flexibility necessary to meet a wide range of DRE systems' real-time requirements, monolithic-ORB implementations must include all possible memory allocation algorithms, and not rely only on Java's built-in heap. Java's built-in heap, while simple to implement, allows the uncontrolled and unpredictable garbage collector to cause hard real-time deadlines to be missed. Furthermore, Real-time Java's garbage collector does not mandate predictable behavior and thus may allow unbounded priority inversion.

Micro-ORB Design Solution in ZEN. To support a variety of memory allocation algorithms, we use the Strategy pattern [23] to make the algorithms pluggable, as shown in Figure 12. We also use the Thread-Specific Storage pattern [9]

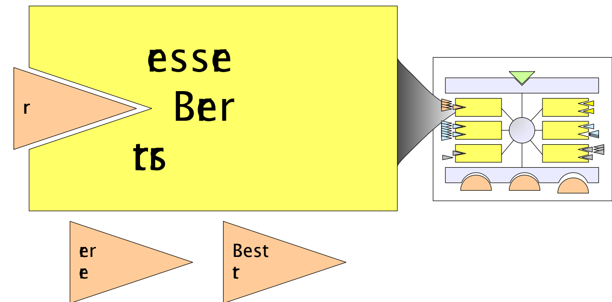

Figure 12: Pluggable Allocators

to allow each ORB to have its own memory management algorithm, thereby reducing the need for thread synchronization across ORB objects. We define a base class for buffer allocator that provides operations `new` and `delete`.

Middleware users may choose from a set of standard algorithms, such as *fast fit* or the *buddy system*, to implement `new` and `delete`. Only the classes implementing the algorithms chosen by the application developer need be loaded and plugged in for use. In future versions of ZEN, we will investigate varying the algorithm dynamically to improve performance, based upon online feedback and reflection.

4 Concluding Remarks

This paper describes the design of the ZEN ORB. The objectives of the ZEN project are to:

- Make development of distributed real-time embedded (DRE) systems easier, faster, more extensible, and more portable
- Reduce the footprint size of middleware for use in memory-constrained embedded systems.
- Provide an infrastructure for international DOC middleware R&D efforts by releasing ZEN in open-source form `http://www.zen.uci.edu`.

To achieve these goals, ZEN integrates the following COTS technologies:

- Java, which is relatively easy to learn and use correctly.
- Real-time Java, which alleviates drawbacks with Java when used to develop of real-time applications.
- CORBA, which is a widely adopted standard for developing distributed applications
- Real-time CORBA, which extends CORBA with key end-to-end QoS capabilities.

Acknowledgements

We would like to acknowledge the efforts of the Distributed Object Computing (DOC) research group members at UC Irvine who contributed to the implementation of ZEN: Mayur Deshpande, Arvind Krishna, Sumita Rao, Mark Panahi, Sean McCarthy, Ossama Othman, Jennifer Offtermatt, Krishna Raman, Prasad Mahendra, Emad Farraj, and Barry Nathan. We are also grateful to Angelo Corsaro, whose knowledge of Real-time Java and detailed critiques sharpened the focus of this paper. Finally, thanks to Susan Anderson for her suggestions that improved the presentation of this paper.

References

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

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

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

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

[5] Realtime Platform SIG, "Realtime CORBA," White Paper, Object Management Group, Dec. 1996. Editor: Judy McGoogan, Lucent Technologies.

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

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

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

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

[10] C. Gill, D. C. Schmidt, and R. Cytron, "Multi-Paradigm Scheduling for Distributed Real-Time Embedded Computing," *Submitted to the IEEE Proceedings Special Issue on Embedded Software*, Oct. 2002.

[11] C. D. Locke, "Software Architecture for Hard Real-Time Applications: Cyclic Executives vs. Fixed Priority Executives," *The Journal of Real-Time Systems*, vol. 4, pp. 37–53, 1992.

[12] C. D. Gill, D. L. Levine, and D. C. Schmidt, "The Design and Performance of a Real-Time CORBA Scheduling Service," *Real-Time Systems, The International Journal of Time-Critical Computing Systems, special issue on Real-Time Middleware*, vol. 20, March 2001.

[13] E. D. Jensen, "Distributed Real-Time Specification for Java." java.sun.com/aboutJava/communityprocess/ jsr/jsr_050_drt.html, 2000.

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

[15] J. Lakos, *Large-scale Software Development with C++*. Reading, Massachusetts: Addison-Wesley, 1995.

[16] F. Kon, M. Roman, P. Liu, J. Mao, T. Yamane, L. Magalhaes, and R. Campbell, "Monitoring, Security, and Dynamic Configuration with the dynamicTAO Reflective ORB," in *Proceedings of the Middleware 2000 Conference*, ACM/IFIP, Apr. 2000.

[17] M. Roman, M. D. Mickunas, F. Kon, and R. H. Campbell, "LegORB and Ubiquitous CORBA," in *Reflective Middleware Workshop*, ACM/IFIP, Apr. 2000.

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

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

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

[21] Object Management Group, *CORBAServices: Common Object Services Specification, Updated Edition*. Object Management Group, Dec. 1998.

[22] S. M. Donahue, M. P. Hampton, M. Deters, J. M. Nye, R. K. Cytron, and K. M. Kavi, "Storage allocation for real-time, embedded systems," in *Embedded Software: Proceedings of the First International Workshop* (T. A. Henzinger and C. M. Kirsch, eds.), pp. 131–147, Springer Verlag, 2001.

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