Towards Predictable Real-time Java Object Request Brokers

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

Douglas C. Schmidt
Electrical Engineering & Computer Science
Vanderbilt University
douglas.c.schmidt@vanderbilt.edu

Angelo Corsaro
Computer Science
Washington University
corsaro@cs.wustl.edu

This paper was published in the Proceedings of the 9th Real-time/Embedded Technology and Applications Symposium.

Abstract

Distributed real-time and embedded (DRE) applications often possess stringent quality of service (QoS) requirements. Designing middleware for DRE applications poses several challenges to object request broker (ORB) developers. This paper provides the following contributions to the study of middleware for DRE applications. First, we outline the challenges present in one of the principal ORB components – the portable object adapter (POA) – focusing on predictable and scalable demultiplexing. Second, we describe how these challenges are addressed in ZEN, which is an implementation of Real-time CORBA that runs atop jRate, an ahead-of-time compiler that implements most of the Real-Time Specification for Java (RTSJ). Third, we qualitatively and quantitatively compare ZEN's demultiplexing strategies with those of other popular Java ORBs, including JacORB, Sun JDK ORB, and ORBacus. Our results show that ZEN and jRate incorporate the strategies necessary to enable predictability using standards-based middleware and also provide a baseline for what can be achieved by combining Real-time CORBA and RTSJ.

1. Introduction

Emerging trends in DRE middleware. Distributed, real-time, and embedded (DRE) applications and middleware have been traditionally developed in C and C++. Conventional Java middleware has historically been unsuitable for DRE applications due to the under-specified scheduling semantics of Java threads and the ability of the Java Garbage Collector (GC) to preempt any other Java thread, leading to unbounded preemption latency. To address these problems, the Real-time Java Experts Group has defined the RTSJ [1], which extends Java by providing new memory management models that allow access to physical memory and can be used in lieu of garbage collection, and stronger guarantees on thread semantics than in regular Java. The Real-time CORBA (RT-CORBA) specification [11] is rapidly maturing technology standardized by the OMG that can simplify many challenges present in developing middleware for DRE applications. It allows DRE applications to configure and control *processor resources* via thread pools and priority mechanisms, *communication resources* via protocol properties, and *memory resource* via request buffering and bounding size of thread pools.

Overview of ZEN. ZEN [10] is open-source RT-CORBA middleware inspired by many of the patterns, techniques, and lessons learned in The ACE ORB (TAO) [15]. In ZEN, the challenge of implementing RT-CORBA using RTSJ to achieve the predictability required for DRE applications is divided into two levels: (1) *Applying optimization principles at the algorithmic and data structure level to ensure predictability*. These optimizations are independent of the virtual machine. (2) *Applying RTSJ features effectively within an RT-CORBA ORB*. This paper focuses on optimizing a key element of the first level, *i.e.*, the CORBA Portable Object Adapter (POA). In addition, this paper shows that optimizations at only the first level are insufficient to achieve necessary real-time predictability; some real-time features of RTSJ must also be used.

Our earlier work on ZEN focused on the extensible component architecture of its POA [9]. This paper extends our earlier work on ZEN, as well as predictable demultiplexing techniques [13], by (1) adding real-time capabilities to the ZEN POA, (2) comparing the performance of ZEN's POA with other Java ORBs to show the predictability enabled by its demultiplexing strategies, and (3) illustrating how ZEN behaves when combined with jRate [3], an open-source ahead-of-time compiler that supports most of the RTSJ. jRate extends the open-source GNU Compiler for Java (GCJ) runtime system [4] to provide an ahead-of-time compiled platform. The Java and RTSJ services, such as garbage collection, real-time threads, and scheduling, are accessible via the GCJ and jRate runtime systems, respectively.

Paper organization. The remainder of this paper is organized as follows: Section 2 describes the RT-CORBA challenge of ensuring predictable and scalable demultiplex-

ing and explains optimizations applied in ZEN to addresses these challenges; Section 3 qualitatively and quantitatively compares the optimizations in ZEN with those in other popular Java ORBs; Section 4 compares our research on ZEN with related work; and Section 5 presents concluding remarks.

2. Optimizing Demultiplexing Predictability and Scalability

The POA is the CORBA component that enables server developers to manage object implementations (known as "servants") portably across ORB implementations [7]. Although the POA provides many powerful and flexible features, its richness complicates the request demultiplexing path through a real-time server ORB. Naive implementations of POA request demultiplexing can increase nondeterminism and priority inversions [13], which are problematic for DRE applications. This section therefore describes solutions to key challenges in optimizing POA implementations to support real-time features. We focus on achieving predictability in ZEN, through scalable and predictable demultiplexing strategies. Section 3 then quantitatively and qualitatively compares the approaches used in ZEN with those in other Java ORBs.

Context. A CORBA-compliant ORB endsystem must perform the demultiplexing steps shown in Figure 1 and described below:

- **(Steps 1-2) OS kernel demultiplexing.** The OS protocol stack demultiplexes the request from the network interface through the data link, network, and transport layers, and up to the user/kernel boundary. It is here that the data is passed to the ORB core running in a server process.
- **(Steps 3-4) ORB demultiplexing.** The ORB core uses the addressing information in an object key of the client request to locate the appropriate POA. Since a POA hierarchy may be nested, locating the POA that contains the servant can involve several demultiplexing steps. The target POA then uses the *object id*, which is part of the *object key*, to locate the associated servant.
- **(Steps 5-6) Operation demultiplexing.** The POA uses the operation name to locate the appropriate IDL skeleton, which (1) demarshals the request for the operation parameters, and (2) performs the upcall to the method implementing the object's operations.

Problem. Conventional ORBs spend a significant – and unpredictable – amount of the total server time demultiplexing requests [5]. Request demultiplexing becomes unpredictable because of these three server properties: 1) the POA hierarchy can have any number of levels, 2) an individual POA may have a large number of servants to manage, and 3) each servant may have a large number of operations

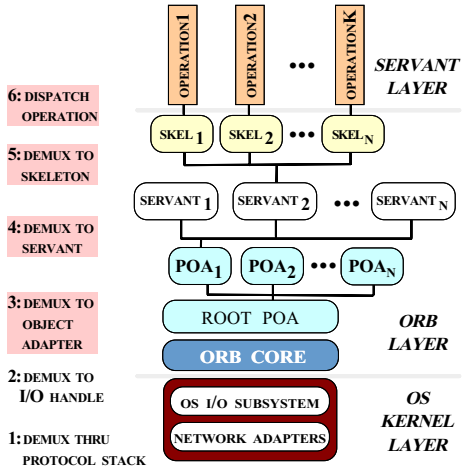

Figure 1. Demultiplexing Steps in CORBA Request Processing

as defined by their IDL interface. A naive demultiplexing implementation of the layered steps shown in Figure 1 may cause a variable number of iterations over the POA hierarchy to locate the POA, locate the servant, and then locate the appropriate method to perform the upcall. Since predictable and scalable demultiplexing is necessary for real-time applications that possess stringent QoS guarantees, the algorithm used to locate a specific method in a specific servant within a specific POA must be predictable and efficient.

An additional problem, in a layered demultiplexing implementation, is that servant-level QoS information is not available to the lower layers present in an ORB endsystem. An object adapter demultiplexing requests in FIFO order can inadvertently allow higher priority requests to wait while lower priority requests are serviced.

Solution → Active demultiplexing with perfect hashing. A variety of strategies can be applied to implement demultiplexing in an ORB, including linear search, binary search, perfect hashing [14], and active demultiplexing [5]. A comprehensive discussion of the implementation and measurements of each of these strategies used in TAO can be found in [6]. Building on these strategies, we outline below good solutions for ORB request demultiplexing, referencing the steps shown in Figure 1:

- **(Step 3) → Active demultiplexing.** The number of POAs in a CORBA server application can change with time, since POAs can be activated/deactivated dynamically. An efficient way to find the POA corresponding to a request is to apply an *active demultiplexing* strategy, which uses the POA ID stored in a request's object key to index directly into a table managed by the object adapter. The index itself is encoded into the object key as an *asynchronous completion token* (ACT) [16] and then passed to clients as part of

the object reference. When a server ORB receives a request from a client, the ORB uses the ACT encoded in the object key to access the corresponding POA in a single $O(1)$ table lookup.

• **(Step 4) → Active demultiplexing.** Servants within a POA can also be activated/deactivated dynamically. As with POA lookup, active demultiplexing can be used to locate a servant efficiently. In this case, the object id stored within the object key can be used as an ACT to index directly into a table managed by its enclosing POA. A POA servicing a client request uses the ACT to locate the servant in a single $O(1)$ table lookup.

• **(Step 5) → Perfect hashing.** Unlike POAs and servants, the names of operations in the IDL interface are known *a priori* and do not change dynamically. A perfect hash function [14] can therefore be computed by the IDL compiler. A demultiplexing strategy based on perfect hashing executes in constant time and space, regardless of the number of operations in an IDL interface.

ZEN implements the optimal active demultiplexing and perfect hashing strategies discussed above. Not only are these strategies optimal, they are also compliant with the CORBA specification. Section 3 presents a qualitative and quantitative comparison of ZEN's demultiplexing strategies with those of other popular Java ORBs.

3. Comparing ZEN's Demultiplexing Strategies with Other CORBA ORBs

This section quantitatively and qualitatively compares the approaches used in ZEN for the ORB-level demultiplexing (steps 3–5 in Figure 1) with those in other ORBs, including JacORB [2], the Sun JDK 1.4.1 ORB [17], and ORBacus [12]. Although each ORB supports a variety of options, we make the following assumptions for this analysis:

1. The child POAs have the same policies as the Root POA.
2. Portable interceptors are not considered in request processing.
3. The sequence of steps analyzed is for a remote client request, not a collocated request.
4. All POAs are created during initialization. The complexity of dynamic activations of POAs using adapter activators is not considered.
5. Servants are normal CORBA servants that inherit from `org.omg.PortableServer.Servant`. We do not consider DII and DSI.
6. No proprietary policies are used in the ORBs.

These assumptions are representative of the way in which DRE applications commonly apply ORB middleware.

In addition to the qualitative comparisons, we also present experimental results of quantitative comparisons of POA-related demultiplexing. In these experiments, a single-threaded client issues IDL operations at the fastest possible rate using a flooding model. For each client request, high-resolution timers used within the ORB measure the *dispatch time*, which is the request processing time starting when the appropriate POA was found until the the request was delivered to the servant. We present a two-fold predictability analysis using the results obtained as follows:

• **Inter-ORB analysis,** where we compare ZEN's demultiplexing schemes with those of the other three ORBs. These measurements were performed on a dual-CPU Intel Xeon 1,700 Mhz processor with 1GB of main memory. All tests were conducted on JVM version 1.4.0 running on Linux OS 2.4.18. ZEN version 0.8, JacORB version 1.4.1, the Sun ORB in J2SE^{TM} v1.4.1_01, and ORBacus version 4.1.2 were used for the experiments. For each condition, 5,000 samples were collected. To minimize the number of GC runs during ORB execution, the initial heap size of the Java Virtual Machine (JVM) was set to 600M using the `-Xms` option, and the maximum heap size was set to 800M using the `-Xmx` option.

• **jRate analysis,** where we compare the improvement in predictability by compiling ZEN using jRate [3], an ahead-of-time compiler that supports most of the RTSJ. These experiments show that combining jRate and ZEN's demultiplexing strategies can enhance the the predictability of middleware for DRE applications. Experiments were performed on an Intel Pentium III 864 Mhz processor with 256 MB of main memory. For these experiments, ZEN was compiled using the GNU `gcj` [4] compiler version 3.2.1 and executed using jRate 0.3a on Linux 2.4.7-timesys-3.1.214 kernel.

3.1. POA Demultiplexing Comparisons

Both JacORB and the Sun ORB use a *linear search* strategy, *i.e.*, iterate through each layer in the POA hierarchy to find the target POA. Each POA, in both the ORBs, is associated with a POA id, which is initialized to the complete POA path name starting from the RootPOA. This POA id is encoded in the object key portion of all CORBA objects. After receiving a client request, the POA id is demarshaled and parsed to obtain the names of the intermediate POAs between the Root and the target POA. A sequential traversal of the POA hierarchy starting at the Root POA leads to the target POA being found. Since JacORB and Sun ORB's POAs use a linear search algorithm, their demultiplexing incurs a worst case time bound of $O(n)$. This strategy is non-scalable and unpredictable for DRE applications when the depth of the POA hierarchy is not known *a priori*.

In ORBacus, every POA created is registered with two entities: 1) its POA Manager and 2) the POALocator class. Both the entities use *dynamic hashing, i.e.*, `java.util.Hashtable`, to associate the POA id with the corresponding POA reference. During object creation, the POA id is embedded in the object key portion of the CORBA object. On receipt of a client request, the POA id present in the request is demarshalled. The internal hashtable is consulted to locate the target POA. ORBacus's use of dynamic hashing provides $O(1)$ lookup time only when there are no collisions; performance degrades as the number of collisions increase. The number of collisions can be controlled to a certain extent by setting the `java.util.Hashtable load factor`, though this strategy also increases the amount of space used by the hashtable, which may not be appropriate for DRE applications where both predictability and memory footprint are important.

In ZEN, we use active demultiplexing. With active demultiplexing, the overhead of Java's hashtable and automatic hashtable resizing is eliminated. The `POAServerRequestHandler` class maintains an `ActiveDemuxPOATable`. Every POA created in ZEN registers with the `POAServerRequestHandler`. During this registration, the `ActiveDemuxPOATable.bind()` operation is invoked to create an entry in the table for the POA.

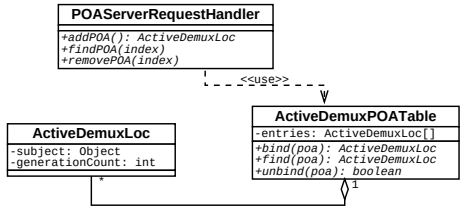

Figure 2. POA Demux: Static Structure

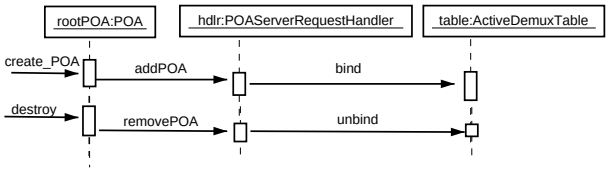

Figure 3. POA Demux: Sequence Diagram

Figure 2 shows the class diagram, and Figure 3 shows the sequence of steps leading to the creation of an entry for the POA in ZEN's demultiplexing table. When an object is externalized (*e.g.*, via the `ORB::object_to_string()` operation), the index into the POA demultiplexing table is added to the object key as a hint. Any request for this object then contains this index in its object key portion. The `POASeverRequestHandler` uses this information to locate the POA that services the request in a single $O(1)$ table lookup.

Empirical results: Inter-ORB analysis. For each of the four Java ORBs (ORBacus, JacORB, Sun, and ZEN), we measured the POA demultiplexing time as the depth of the POA hierarchy was varied. POA hierarchy depth was varied from 1 to 150 in increments of 25, for a total of 7 conditions. For each condition, the time to reach the leaf POA was measured. Four different dependent measures of demultiplexing time were analyzed: 1) latency for the average case; 2) latency for the 99% case; 3) latency for the worst case; and 4) dispersion (standard deviation). A sample size of 5,000 data points was used for each condition.

- **Average measures.** Figure 4 illustrates that ZEN's active demultiplexing is highly efficient; the average latency is $\sim 8\mu$secs. Furthermore, ZEN's performance is scalable; average latency remains invariant across all conditions of hierarchy levels. In contrast, the performance of dynamic

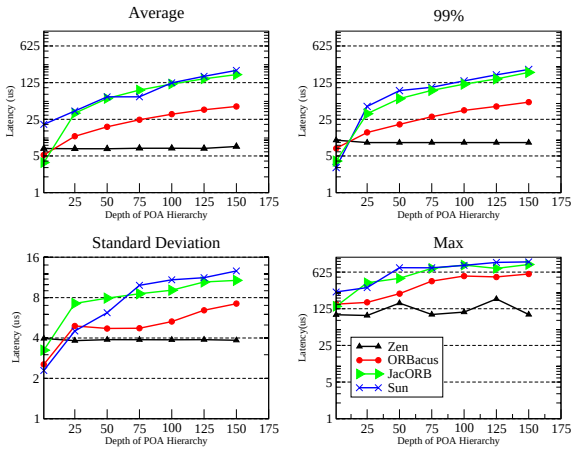

Figure 4. POA Demux Analysis

hashing (ORBacus) degrades rapidly as the nesting of the POA hierarchy increases and number of collisions in the hashtable increase. Moreover, linear search (Sun ORB and JacORB) degrades rapidly with increase in depth of POA hierarchy.

- **Dispersion measures.** ZEN's performance shows high predictability; the dispersion in ZEN is smaller than that of other ORBs. Furthermore, the dispersion for ZEN is independent of POA hierarchy depth. Conversely, the dispersions for ORBacus, Sun ORB, and JacORB increase with the depth of the POA hierarchy, indicating a reduction in predictability with the depth of hierarchy.

- **Worst Case Measures.** The 99% bounds for all ORBs follow a trend similar to that of their average cases. ZEN's bound is smaller than those of the other three ORBs and does not vary with the depth of POA hierarchy. Similarly, the worst case measures for ZEN are smaller than those of the other ORBs. In contrast to the 99% bound, which is close to the average case, the worst case measures are high. The high maximum (max) values correlate with

the GC in Java. Table 1 shows that the more the GC runs, the worse performance becomes (*e.g.*, Sun ORB's GC runs 54 times, whereas ZEN's GC runs 10 times). A second

ORB	GC runs	Avg Mem Collected/run	Avg Time/run
ZEN	10	609728K	2.67000ms
ORBacus	26	609728K	2.74590ms
JacORB	34	609728K	2.82470ms
Sun	54	609728K	2.16120ms

Table 1. GC Stats:POA Hierarchy depth=150

factor is the use of JDK 1.4, which does not support Real-time Java. Our analysis below shows how the use of jRate alleviates the high worst case values.

Empirical results: jRate Analysis. Figure 5 shows that the use of jRate increases the predictability of the demultiplexing strategy used in ZEN. The dispersion when using

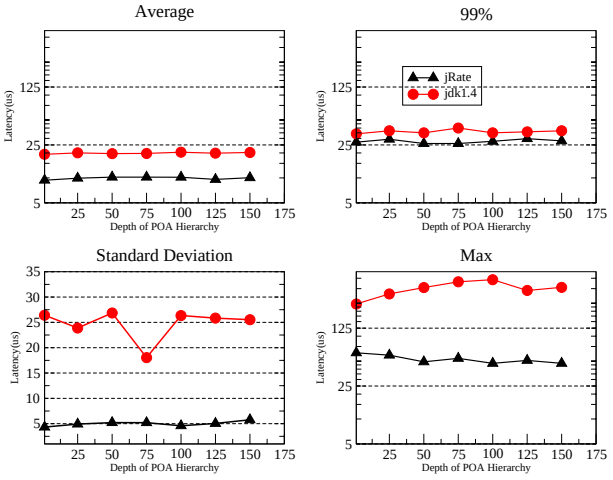

Figure 5. jRate POA Demux Analysis

jRate is smaller than when using JDK 1.4 by a factor of 5. Similarly, the worst case latencies with jRate are significantly smaller than with JDK 1.4; the worst case latency with jRate is $\sim$60μsecs, more than a factor of 7 better than $\sim$450 μsecs with JDK 1.4. Moreover, the average latency with jRate ($\sim$14μsecs) is less than with JDK 1.4 ($\sim$20μsecs).

3.2. Servant Demultiplexing Comparisons

ORBacus, JacORB, and Sun ORB use dynamic hashing for Step 4 of demultiplexing. Every POA in the ORBs is associated with an Active Object Map entity (AOM). The AOM maintains an internal Hashtable[1] of type `java.util.Hashtable` to register the association between the object id of the CORBA object being activated and its corresponding servant. All object references generated by the POA have the object id embedded within the object key. After receiving a client request, the object id embedded within the request is demarshaled. The AOM uses the object id within the client request to consult its internal hashtable to obtain the corresponding servant.

ZEN uses Active Demultiplexing for this stage of demultiplexing. ZEN's `RetainStrategy` (corresponding to the ACTIVE_OBJECT_MAP_ONLY value of the servant retention policy) maintains an `ActiveDemuxServantTable`, as shown in Figure 6. ZEN assigns every active object in the POA a location in this table. The POA `servant_to_reference()` and `id_to_reference()` operations assign the servant a slot in the demux table using the internal ZEN `bindServant()` method. The servant location in the table is added as an ACT to the object key corresponding to the externalized CORBA object. Any client requests for this object thus contain this index in the object key. By using the information identified by the ACT, the servant corresponding to the object key can be found in a single $O(1)$ table lookup. The POA's `deactivate_object()` operation invokes ZEN's internal `unbindServant()` method on the `ActiveDemuxServantTable`, which in turn recycles the demultiplexing slots in the table, as shown in Figure 7.

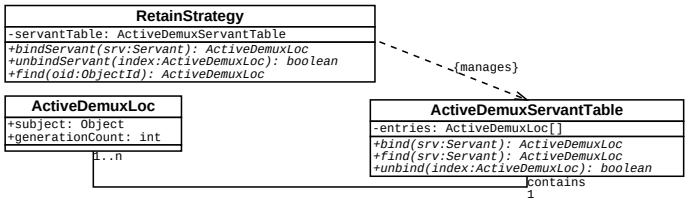

Figure 6. Servant Demux: Static Structure

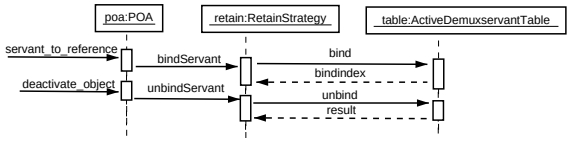

Figure 7. Servant Demux: Sequence Diagram

Empirical results: Inter-ORB analysis. For each of the four Java ORBs (ORBacus, JacORB, Sun, and ZEN), we measured the servant demultiplexing time as the number of active servants was varied. The number of active servants registered with the POA was varied from 1 to 1,500 in increments of 250, for a total of 7 conditions. For each condition, the time to reach the last servant registered and externalized was measured. The same four dependent measures used in the POA hierarchy tests were used here. A sample size of 5,000 data points was used for each condition.

[1] In JacORB, using the jacorb.hashtable_class property, the end user can specify the exact hashtable class to be used.

• **Average measures.** Figure 8 illustrates that ZEN's active demultiplexing is highly efficient; the average latency is ∼8 μsecs. Furthermore, ZEN's performance is independent of the number of active servants. In contrast, the per-

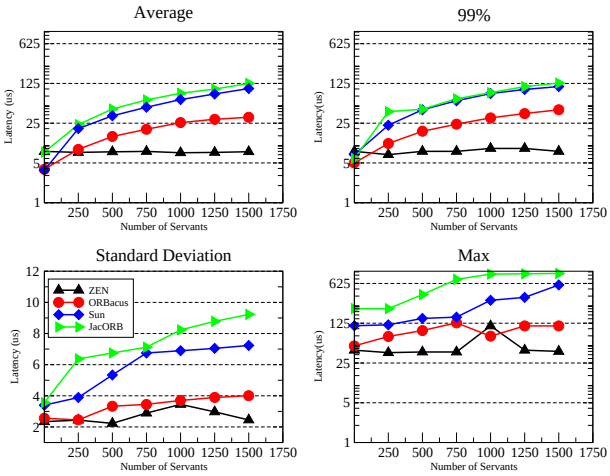

Figure 8. Servant Demux Analysis

formance of dynamic hashing used by ORBacus, JacORB, and Sun ORB degrades with the number of servants. Interestingly, the graph reveals that JacORB and ORBacus outperform ZEN for a small number of servants. The reason for this behavior is that after ZEN locates the target servant using the ACT in the object key, a check is necessary to 1) compare generation counts[2] in the object key and in the active demux table to determine if the target servant is the same, and 2) check if the target servant is still active. In the other ORBs, however, deactivation of servants causes entries to be removed from the AOM, thus avoiding these extra comparisons.

• **Dispersion measures.** Figure 8 shows that ZEN's dispersion is generally smaller than that of other ORBs. Further, ZEN's dispersion does not increase reliably with number of active servants. The Max panel reveals that for the condition of 1,000 servants, an anomalous worst case sample measurement at 112μsecs is responsible for the increase in dispersion. Dispersion is decreased in the other conditions, indicating that the increase did not stem from the increase in the number of servants. Conversely, the dispersions for ORBacus, Sun ORB, and JacORB increase with the number of active servants, indicating decreased predictability with increase in number of servants.

• **Worst Case Measures.** The 99% bound for ZEN is small, very close to its average latency. Unlike the other ORBs, ZEN's 99% bound does not vary with the number of active servants. ZEN's worst case latencies are high, however. Once again, similar to the POA demultiplexing results from Section 3.1, the more frequently the GC runs, the greater the worst case latencies become. (Table 2 shows that JacORB's GC runs 47 times, whereas ZEN's GC runs 12 times). The high worst case latencies are also due to Sun

ORB	GC runs	Avg Mem Collected/run	Avg Time/run
ZEN	12	609728K	2.4560ms
ORBacus	29	609728K	2.7459ms
JacORB	47	609728K	3.8247ms
Sun	37	609728K	2.5612ms

Table 2. GC Stats: Number of Servants=1,500

JDK 1.4, which does not guarantee real-time behavior. The use of jRate, discussed below, ameliorates this shortcoming.

Empirical results: jRate Analysis. The results presented in Figure 9 show a pattern similar to the POA demultiplexing results. The use of jRate not only ensures better pre-

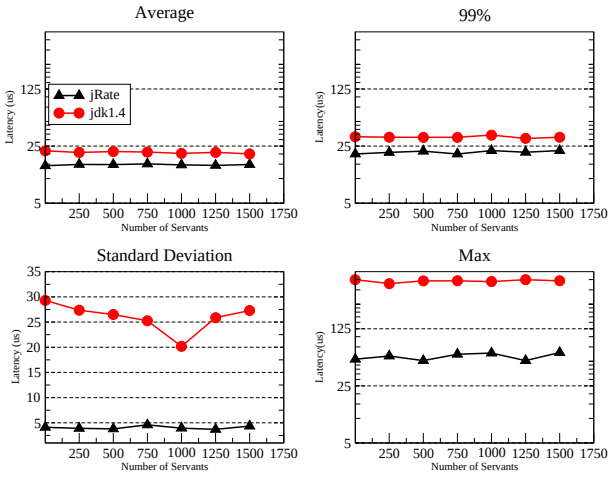

Figure 9. jRate Servant Demux Analysis

dictability but also leads to a lower latency for this stage of demultiplexing. The dispersion measures indicate that the use of jRate ensures a tighter bound than JDK 1.4; the dispersion with jRate is lower by a factor of 5, and does not vary with number of active servants. Moreover, the worst case results for jRate display a trend similar to the dispersion measures. The maximum worst case measurements for ZEN with JDK 1.4 (∼480μsecs) are greater than with jRate (∼65μsecs) by a factor of 7.

3.3. Operation Demultiplexing Comparisons

The ZEN IDL compiler invokes `jperf`, which is a Java implementation of the GNU `gperf` perfect hash function generator [14]. `jperf` runs as a child process to generate scalable, efficient, and predictable collision-free tokens for lookup methods based on the operation names defined in an IDL interface. After receiving a client request, the operation name is demarshaled and then used to perform the upcall.

[2]Generation counts help to recycle slots in the active demux table.

For each operation name in the IDL interface, JacORB and Sun's IDL compiler create a unique `java.lang.Integer` index and store the association in a `java.util.Hashtable`. After receiving a client request, the operation name is used as a key into the hashtable to obtain its index. Using a switch on the index, the code performing the upcall is reached. We were unable to run the `jidl` IDL compiler provided by ORBacus. Hence, it is not possible for us to discuss the demultiplexing strategy used. We have reported this as a bug.

Empirical results: Inter-ORB analysis. For each of the three Java ORBs (JacORB, Sun, and ZEN), we measured the operation demultiplexing time as the number of operations in an IDL interface was varied. Number of operations was varied from 1 to 50 in increments of 10, for a total of 6 conditions. The same four different dependent measures of demultiplexing time used previously were analyzed here as well. A sample size of 5,000 data points was used for each condition.

• **Average measures.** Figure 10 illustrates that ZEN's perfect hashing is highly efficient; the average latency is $\sim$1μsec, and does not vary with the the number of active servants. In contrast, both JacORB and Sun exhibit

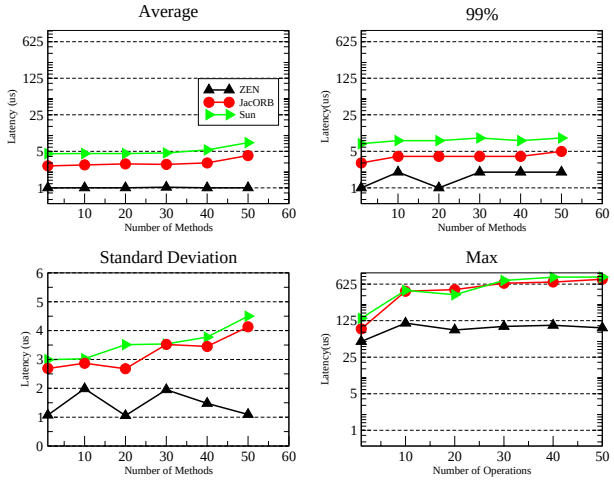

Figure 10. Operation Demux Analysis

slower performance. The dynamic hashing strategies used by JacORB and Sun ORB both incur a higher overhead to compute the hash function. Latency for both ORBs, however, is almost constant until the number of methods reaches 30, at which point the latency increases by 3μsec for each condition.

• **Dispersion measures.** Figure 10 illustrates that ZEN's predictability is better than that of the other ORBs; ZEN's dispersion is smaller than that of the other ORBs for all cases, and does not reliably vary with the number of methods. In contrast, the dispersions for the Sun ORB and JacORB are larger and increase with the number of operations, indicating decreased predictability as the number of operations increases.

• **Worst Case Measures.** ZEN's 99% bound and worst case measures are tighter compared to the other ORBs. As in previous tests, the worst case measures for all ORBs are high compared to the average and 99% bound. The GC analysis in Table 3 (with 50 methods) shows that the highest worst case latency (the Sun ORB) coincides with the maximum number of GC runs. In contrast, ZEN has the lowest worst case latency and fewest GC runs. The high

ORB	GC runs	Avg Mem collected/run	Avg Time/run
ZEN	5	609728K	1.156ms
JacORB	7	609728K	1.238ms
Sun	14	609728K	1.345ms

Table 3. GC Stats:Number of Methods=50

worst case latency indicates that ZEN has predictable demultiplexing most of the time but not all the time when run on JDK 1.4. The use of jRate, discussed below, addresses this issue.

Empirical results: jRate Analysis. Figure 11 shows that jRate improves the predictability of ZEN's perfect hashing strategy. Overall performance is faster, as shown by the av-

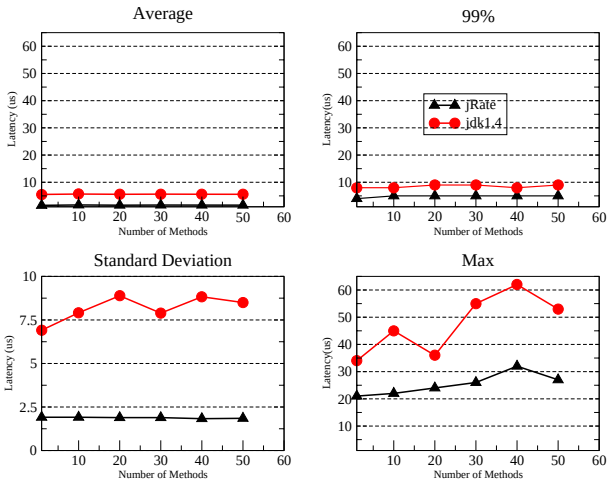

Figure 11. jRate Operation Demux Analysis

erage latencies which differ by a factor of four. The dispersion with jRate is tighter than with JDK 1.4, less by a factor of 5. The worst case measures for jRate show a trend similar to the dispersion measures; the maximum worst case latency with jRate is $\sim$30μsecs while that of JDK 1.4 is $\sim$60 μsecs.

4. Related Work

In recent years, a considerable amount of research has focused on enhancing the predictability of middleware for

DRE applications. We summarize key related efforts here.

TimeSys RTSJ Reference Implementation (RI). TimeSys has developed the official RTSJ Reference Implementation (RI) [18], a fully compliant implementation of Java that implements all the mandatory features in the RTSJ. As soon as the commercial TimeSys RTSJ implementation is available, we plan to port ZEN to it.

RTSJ benchmarking suites. RTJPerf [3] is an open-source RT-Java benchmarking suite that tests/benchmarks most of the RTSJ features critical to real-time embedded systems. The Real-Time Java for Embedded Systems (RTJES) program [8] 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. We are developing a similar benchmarking suite to test ZEN using RT-Java.

5. Concluding Remarks

Ensuring end-to-end middleware predictability is essential to support the QoS capabilities needed by DRE applications. This paper describes the optimizations applied in the CORBA object adapter layer to address key RT-CORBA scalability and predictability challenges. We show how the data structures and algorithms used in the ZEN RT-CORBA ORB are more predictable than those used in other Java ORBs, including JacORB, Sun JDK ORB, and ORBacus. By using a combination of active demultiplexing and perfect hashing, ZEN ensures efficient and scalable $O(1)$ worst-case lookup time for the ORB's demultiplexing stages. Our empirical results show that the strategies used in ZEN – in conjunction with an ahead-of-time compiled RTSJ implementation in jRate – help ensure predictability by bounding jitter and worst case performance for the demultiplexing stages.

The empirical results presented in this paper provide a baseline for what can be achieved using RTSJ middleware to implement RT-CORBA. To develop predictable Java-based middleware, RT-CORBA developers should focus on both (1) efficient data structures and algorithms and (2) using RTSJ features directly in the ORB. There is an important synergy between the two; optimizing just one of these aspects may not ensure the predictability required for DRE systems.

6. Acknowledgements

We would like to acknowledge the efforts of the other members of the Distributed Object Computing (DOC) research group at UC Irvine who are contributing to the design and implementation of ZEN. Special thanks to Carlos O'Ryan and Ossama Othman for contributing to ZEN's initial design.

References

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

[2] G. Brose. JacORB: Implementation and Design of a Java ORB. In *Proc. DAIS'97, IFIP WG 6.1 International Working Conference on Distributed Aplications and Interoperable Systems*, pages 143–154, Sept. 1997.

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

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

[5] A. Gokhale and D. C. Schmidt. Measuring and Optimizing CORBA Latency and Scalability Over High-speed Networks. *Transactions on Computing*, 47(4), 1998.

[6] A. Gokhale and D. C. Schmidt. Principles for Optimizing CORBA Internet Inter-ORB Protocol Performance. In *Hawaiian International Conference on System Sciences*, Jan. 1998.

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

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

[9] R. Klefstad, A. Krishna, and D. C. Schmidt. Design and Performance of a Modular Portable Object Adapter for Distributed, Real-Time, and Embedded CORBA Applications. In *Proceedings of the 4th International Symposium on Distributed Objects and Applications*, Irvine, CA, October/November 2002. OMG.

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

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

[12] I. Object Oriented Concepts. ORBacus. www.ooc.com/ob.

[13] 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*, 8(1), 2000.

[14] D. C. Schmidt. GPERF: A Perfect Hash Function Generator. *C++ Report*, 10(10), November/December 1998.

[15] D. C. Schmidt, D. L. Levine, and S. Mungee. The Design and Performance of Real-Time Object Request Brokers. *Computer Communications*, 21(4):294–324, Apr. 1998.

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

[17] I. Sun Micro Systems. Sun ORB. http://java.sun.com/.
[18] TimeSys. Real-Time Specification for Java Reference Implementation. `www.timesys.com/rtj`, 2001.