

Design and Performance of a Modular Portable Object Adapter for Distributed, Real-Time, and Embedded CORBA Applications

Raymond Klefstad, Arvind S. Krishna, and Douglas C. Schmidt

{klefstad, krishnaa, schmidt}@uci.edu

Electrical and Computer Engineering Dept.

University of California, Irvine, CA 92697, USA*

Abstract

ZEN is a CORBA ORB designed to support distributed, real-time, and embedded (DRE) applications that have stringent memory constraints. This paper discusses the design and performance of ZENs portable object adapter (POA) which is an important component in a CORBA object request broker (ORB). This paper makes the following three contributions to the study of middleware for memory-constrained DRE applications. First, it presents three alternative designs of the CORBA POA. Second, it explains how design patterns can be applied to improve the quality and performance of POA implementations. Finally, it presents empirical measurements based on the ZEN ORB showing how memory footprint can be reduced significantly while throughput is comparable to a conventional ORB implementation.

Keywords. Distributed Real-time and Embedded Systems, Real-time CORBA, Portable Object Adapter, 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), telemedicine (e.g., remote surgery), manufacturing process automation (e.g., hot rolling mills), and defense applications (e.g., avionics mission computing systems).

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 help reduce the complexity of developing distributed applications. DOC middleware simplifies application development for distributed systems by off-loading many tedious and error-prone aspects

of distributed computing from application developers to middleware developers. It has been used successfully in large-scale 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]. It can also handle applications with stringent soft real-time requirements, such as telecommunication call processing and streaming video [7].

ZEN [8] is an open-source Real-time CORBA object request broker (ORB) implemented in Real-time Java [9]. ZEN is inspired by many of the patterns, techniques, and lessons learned in The ACE ORB (TAO) [6], which is a widely-used, open-source implementation of Real-time CORBA written in C++. A key difference between the design of ZEN and that of earlier CORBA ORBs is its extensive application of the *Virtual Component* pattern [10]. This pattern helps reduce the memory footprint contributed by the middleware by factoring out optional or rarely-used functionality from a specific application of the middleware. Many earlier ORB designs were *monolithic* because they included code that supports all of the possible features, choices, and variants specified in the voluminous CORBA specification [1].

Early generations of distributed applications used text messages and message passing over TCP/IP sockets. One key difference between standard message passing and DOC middleware is that the latter allows programmers to exploit tools and techniques developed for stand-alone object-oriented programming. A substantial and significant component of an ORB supporting this style of object-oriented distributed computing is the *portable object adapter* (POA). One or more POAs reside in each server process and provide the following capabilities:

- Creates object references with the appropriate policies
- Activates and deactivates objects
- Etherealizes and incarnates object implementations (known as *servants* in CORBA terminology) and

*This work was funded in part by ATD, DARPA, SAIC, and Siemens.

- Demultiplexes requests sent by remote clients to the appropriate servants in the server.

It is important that an ORB's POA implementation be designed and optimized efficiently and predictably since conventional ORBs spend a significant amount of the total server time demultiplexing requests to servants [11, 12].

This paper makes the following contributions to the design of POAs for Real-time CORBA middleware:

1. It describes three alternative designs of the Portable Object Adapter (POA): *monolithic*, *coarse-grain*, and *fine-grain* architectures.
2. It explains how design patterns [13, 14] can be applied to improve the quality and performance of POA implementations.
3. It presents empirical measurements based on the ZEN ORB showing how memory footprint can be reduced significantly while throughput is comparable to a monolithic/conventional ORB implementation.

The remainder of this paper is organized as follows: Section 2 gives a brief overview of ZEN the Real-time CORBA ORB for Real-time Java; Section 3 explains the capabilities offered by portable object adapters (POAs); Section 4 describes the benefits of a highly-modular architecture and two alternative highly-modular architectures for a POA. Section 5 provides empirical results for the alternative POA designs and compares the performance of ZEN's POA against the performance of JacORB's POA; Section 6 compares our work on POAs with related work; and Section 7 presents concluding remarks and outlines future work.

2 An Overview of the ZEN Real-time ORB

ZEN [15] is a Real-time CORBA ORB implemented using Real-time Java [9], thereby combining the benefits of these two standard technologies. CORBA middleware has addressed many challenges posed by distribution for mainstream applications since the early 1990s. Until the past several years, however, relatively little has been done to meet the challenges specific to DRE systems. CORBA middleware supporting DRE systems must do the following:

- Provide a full range of CORBA services for distributed systems, to meet the needs of a wide variety of application developers.
- Meet a number of stringent QoS requirements, including achieving low and bounded jitter for ORB operations, eliminating sources of priority inversion, allowing applications access to Real-time programming language features, and minimizing 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 and configurable both *statically* and *dynamically*, thereby allowing application developers to trade off 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.

2.1 ZEN's Pluggable ORB Architecture

ZEN's ORB architecture is based on the concept of *layered pluggability*, where various components of the middleware may be "plugged" (included into) or "unplugged" (removed from) on an as-needed basis, thereby allowing flexible middleware configuration. Based on our earlier work with TAO, we identified these eight core ORB services (shown in Figure 1) as candidates to be factored out of the ORB by applying the Virtual Component pattern to reduce its memory footprint and increase its flexibility. We call the remaining portion of code the *ZEN kernel*.

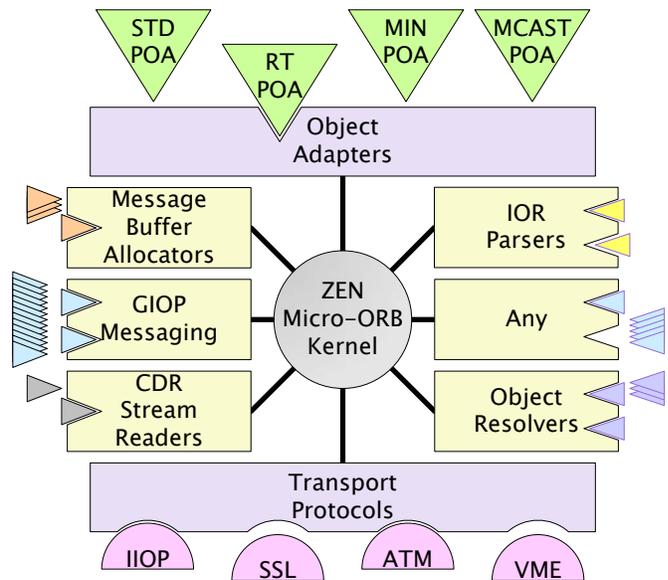


Figure 1: Micro-ORB Architecture of ZEN

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.

3 Portable Object Adapter Functionality and Architecture

The CORBA portable object adapter (POA) is a standard component that enables programmers to compose servants portably across ORB implementations. Unlike its woefully underspecified predecessor—the basic object adapter (BOA)—the POA specification is well designed and provides standardized APIs for the POA operations. For example, the POA specification standardizes the API for registering servants, which was unspecified by the BOA.

3.1 Summary of POA Functionality

The following paragraphs summarize the important functionality provided by the POA:

1. Generating object references. The POA is responsible for generating object references for the CORBA objects it maintains. These references contain addressing information that allows remote clients to invoke operations on each object in a distributed system. This information is provided to the POA by the ORB Core and underlying operating system transport.

2. Behavior governed by policies. The POA provides a extensible mechanism for associating policies with servants in a POA. The values for policies are specified when a POA is created. Currently there are seven standard policies for the POA: *thread*, *lifespan*, *object id uniqueness*, *object id assignment*, *servant retention*, *request processing*, and *implicit activation*.

3. Activation and deactivation of objects. The creation of object references stem from the creation of a CORBA object. Once created, an object can alternate between *activated* and *deactivated* states. An object can service requests only if it is activated. The `deactivate_object()` operation is used to deactivate an object. It is important to note that the lifetime of a CORBA object is different from that of the servant that implements it, as discussed next.

4. Incarnation and etherealization of servants. Servants implement CORBA object interfaces and can be registered with the POA implicitly or explicitly by application developers. To have requests delivered to it, however, an object must be *incarnated* by a servant. The POA can incarnate servants on demand. Etherealization of a servant breaks the association between it and its CORBA object.

5. Demultiplexing requests to servants. Client requests are sent as messages across the underlying OS transport. The POA demultiplexes these requests to the appropriate servants. The POA then invokes the appropriate operation on this servant.

6. SSI and DSI support. The POA allows programmers to construct skeletons that inherit from static skeleton interface (SSI) classes or a dynamic skeleton interface (DSI) class. Clients need not be aware that the request is serviced by a SSI or a DSI servant. Two CORBA objects supporting the same interface can be serviced, one by a DSI servant and one by a SSI servant. Further an object can be serviced by a DSI servant for some period of time, while being serviced by the IDL servant for the remaining time.

3.2 POA Structure and Dynamics

The ORB is an abstraction visible to both the client and server. The POA, in contrast, is only necessary in a process performing the server role, so clients do not require the services of a POA. In ZEN, the ORB and the POA interact through a well-defined `ServerRequestHandler` interface. This design prevents the tight coupling of the ORB and the POA. This interface is specific to ZEN since the OMG has not standardized the interface between the ORB and the POA. In ZEN, the POA is only one specific type of `ServerRequestHandler`. Variants, such as Real-time POA or Multicast POA, may handle requests and perform other POA activities, as well.

User-supplied servants are registered with the POA. Each client has an object reference, representing the remote servant, upon which it can invoke requests. When a request is made, it is passed as a message to the server. The POA then decides which servant the request corresponds to and invokes that operation on the appropriate servant. Figure 2 shows the POA architecture. The architecture shown in this figure is implied by the interface to, and specification of, the POA in the CORBA specification. As long as an implementation of this architecture meets the designated CORBA semantics, however, it need not follow any prescribed design.

Figure 2 shows a special POA (called the `RootPOA`) that is always available to an application through the ORB factory method `resolve_initial_references()`. Application developers can register servants with the root POA if the policies of the root POA specified in the POA specification are suitable for their applications.

A server application may establish multiple POAs to create a naming hierarchy (similar to the hierarchical directory structure found in an OS file system) that also allows setting of individual servant policies. For example, a server application might have two POAs:

- One supporting transient CORBA objects, whose lifetime can not exceed the POA in which it was activated and
- The other supporting persistent CORBA objects, whose lifetime can exceed that of its activating POA.

Child POAs are created by invoking the `create_POA()` factory method on a parent POA.

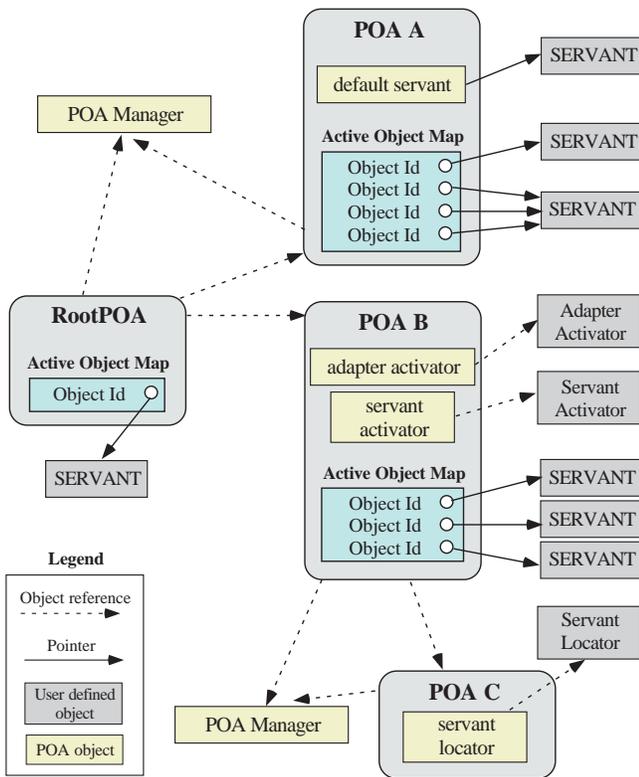


Figure 2: The POA Architecture

The server application in Figure 2 contains three other nested POAs: A, B, and C. POA A and B are children of the root POA; POA C is B’s child. Each POA has an active object map that associates object ids to servants. Other key components in a POA are discussed below.

Adapter Activator. An adapter activator can be associated with a POA by an application. The ORB will invoke an operation on an adapter activator when a request is received for a child POA that does not yet exist. The adapter activator can then decide whether or not to create the required POA on demand.

POA Manager. A POA manager encapsulates the processing state of one or more POAs. By invoking operations on a POA manager, server applications can cause requests for the associated POAs to be queued or discarded.

Servant Manager. A servant manager is a locality constrained servant that is provided by the application developer. The ORB uses a servant manager to activate and deactivate servants on demand. Servant managers are responsible for (1) managing the association of an object (as characterized by its Object Id value) with a particular servant and (2) for determining whether or not an object exists. There are two types of servant managers: *ServantActivator* and *ServantLocator*. The

type used in a particular situation depends on the policies in a POA, which are described in section /refPOAarch.

4 The Design of ZEN’s POA

This section details both the goals for ZEN’s POA and several alternative POA design that may achieve those goals. Each alternative design is implemented and the benchmark comparisons are presented in Section 5.

4.1 Goals of ZEN’s POA design

Our experience building TAO taught us that to achieve a small middleware footprint, feature subsetting must be planned early in the design stages since it is hard to reduce footprint after an implementation has become tightly coupled. Pluggability of optional components, ease of extension, and footprint reduction are primary design goals for ZEN and the POA. More specifically, we have the following goals for ZEN’s POA design:

- **Minimize footprint.** An important goal of ZEN’s POA design is to achieve a small memory footprint for the middleware suitable for DRE systems. Each application should only incorporate the sections of middleware code that it actually needs. By decomposing the POA into virtual components, the POA requires minimal memory for each application using ZEN.
- **Ease adaptation to new changes in the CORBA specification.** A pluggable, highly-modular POA design applies the core software engineering concept of *separation of concerns*. Each of the virtual components in ZEN’s POA encapsulates the implementation for a particular POA policy.
- **Facilitate addition of new custom POA policies.** ZEN is a research platform, so it is important to enable experiments with new algorithms, data structures, and capabilities.

4.2 Alternative POA Architectures

This section presents an overview of each of the three alternative POA design architectures we implemented, measured, and compared: *monolithic POA*, *coarse-grain POA*, and *fine-grain POA*.

4.2.1 Monolithic POA Architecture

In a monolithic POA architecture, the POA is a single large component that contains the semantics needed to implement

(1) policies in the OMG’s POA specification and (2) ORB-specific policies. The monolithic design can increase the footprint (both code and data size) of the POA considerably since the POA implements the behavior required by the entire set of policies, rather than a minimal subset.

Monolithic POA also cannot be easily extended as new policies are added to the CORBA POA specification. Moreover, monolithic POA implementations complicate the addition of ORB-specific policies. Monolithic POA implementations also suffer from inefficiency in terms of redundant checking required to determine the appropriate course of action based on POA policies.

For example, during most operations a monolithic POA needs to check for the presence/absence of policies to incorporate the necessary behavior dictated by the policies associated with the POA. This overhead would be incurred each time the operation was performed. For example, to process a client request, the POA would have to check for the request processing policy value associated with the POA, which would then dictate how the request is processed.

4.2.2 Coarse-grain POA Architecture

In a coarse-grain POA architecture, the POA is still a single large component, but we apply the Virtual Component pattern to the entire POA so it can be plugged-in or removed as shown in Figure 3. A coarse-grain POA architecture is useful for pure

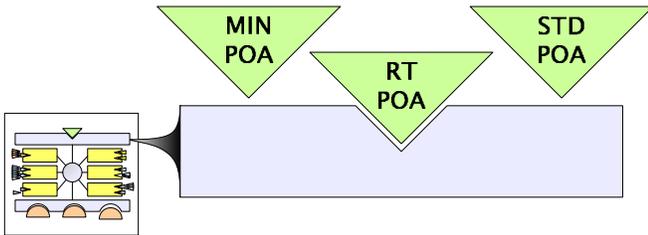


Figure 3: Pluggable Object Adapters

clients, which need no object adapter and can reduce their footprint by completely removing all POA methods. It also useful for pure servers, which can reduce their footprint and achieve custom functionality by loading the most appropriate POA (e.g., the `RootPOA` or a special group communication POA [16]) on demand. The coarse-grain pluggable POA design also simplifies the addition of new object adapters as they are standardized by the OMG.

4.2.3 Fine-grain POA Architecture

The coarse-grained POA architecture has been implemented in TAO [17] using the Component Configurator [13] pattern and

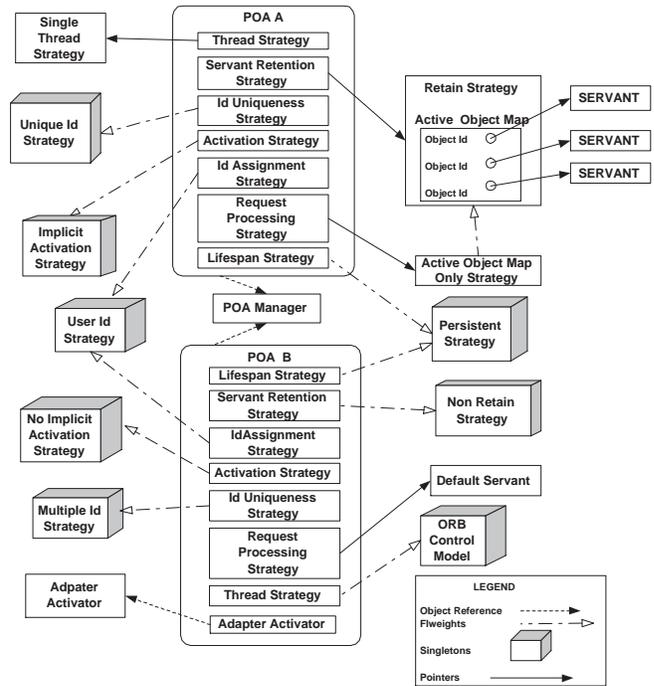


Figure 4: Fine-grain Architecture of the ZEN POA

dynamic link libraries (DLLs) to load each POA implementation variant. In ZEN, we have more aggressively applied the Virtual Component pattern to also support a fine-grain POA architecture. In this approach, instead of an all-or-nothing loading of the POA, individual components of the POA can be loaded as needed.

Based on our work with TAO [18], we observed it is possible to divide the POA into smaller pieces and make them virtual components. Such a fine-grain level of control can further reduce the footprint of a POA when it is needed by an application. We have found it useful to decompose the POA as dictated by the values of the POA policies. Each of the CORBA POA policies has a set of policy values that specify the behavior of the POA with that policy. By breaking down the policies according to their possible values, it is therefore possible to load only the pieces that the POA needs, based on the list of policies specified at POA creation time. For example, Figure 4 shows the fine-grain architecture of the ZEN POA, where each POA policy is factored out into a separate strategy class, as described next in Section 4.3.

4.3 The Design of ZEN’s Fine-grain POA Architecture

The remainder of this section describes how ZEN’s POA is decomposed into modular components in accordance with POA

policy values. We then provide an overview of each POA policy and explain how ZEN implements this policy in a highly modular manner using the virtual components of ZEN’s fine-grain POA architecture.

4.3.1 Primary POA Components

The four strategies described below are considered to be *primary components* in ZEN, *i.e.*, their behavior does not depend on other components. At POA creation time, these components are created first and hold the smallest amount of state. The following are the primary components of the POA.

ThreadStrategy component. This component implements the *Thread* policy, which is used to specify the threading model used in the POA. The POA can have one of the following threading models: *single thread*, *ORB controlled*, or *main thread*. If the POA is single-threaded, all the requests of the POA are processed sequentially. In a multi-threaded environment, all upcalls to the servant are invoked in a manner that is safe for multi-thread unaware code. In contrast, in the ORB-controlled model, multiple requests may be delivered simultaneously using multiple threads.

All requests to a main thread POA are processed sequentially in the thread that runs the `main()` function. All upcalls made by POAs with this policy to servants are made in a manner that is safe for thread-unaware code. If the environment has special requirements that some code must run on a distinguished main thread, servant upcalls will be processed on that thread.

Using the Strategy pattern [19], the semantics of implementing the *Thread* policy can be strategized into two alternatives: class `SingleThread` and class `ORBControlModel`. Each class encapsulates the state and the logic of implementing the behavior specified by the policy. In Zen, the main thread model strategy and the single thread strategy are equivalent. Figure 5 shows the class diagram for ZEN’s ThreadPolicyStrategy alternatives..

At POA creation time, a factory method `{init() }` in the base class `ThreadPolicyStrategy` creates the appropriate strategy instance based on the POA’s policy list. Since the `ORBControlModel` component does not maintain state specific to a POA, it is implemented using the Flyweight pattern [19]. This pattern uses sharing to support large numbers of fine-grain objects efficiently, which means there is one instance of the `ORBControlModel` object. Each POA with that value for the `ThreadPolicyStrategy` policy will have a reference to that single object, thereby reducing memory usage.

Prior to making the upcall on the servant, the POA uses the `ThreadPolicyStrategy`’s `enter()` method. If the `SingleThread` strategy is in place, this method acquires a mutex lock. After the upcall is performed, the `exit()`

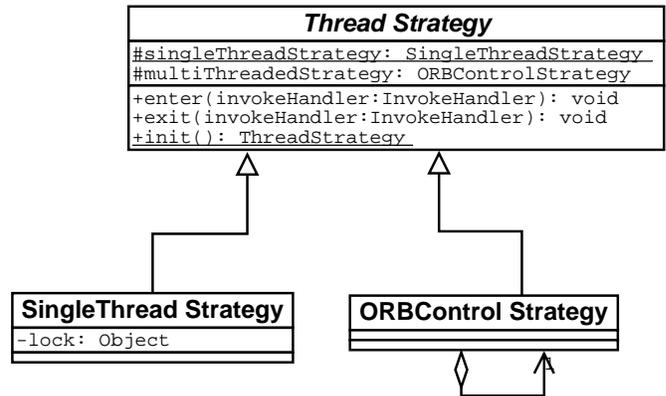


Figure 5: ZEN’s Thread Strategies

method releases the lock. This synchronization is not present in the `ORBControlModel` strategy.

LifespanStrategy component. This component implements the *Lifespan* policy, which is used to specify whether the CORBA object references created within a POA are persistent or transient. Persistent object references can outlive the process in which they are created. Unlike persistent object references, transient object references cannot outlive the POA instance in which they were first created. After the POA is deactivated, the use of object references generated from it will result in an `OBJECT_NOT_EXIST` exception.

The mechanism for implementing the POA’s *Lifespan* policy has been separated into ZEN’s `Persistent` and `Transient` strategies. Figure 6 shows the class diagram of the `LifespanStrategy` component.

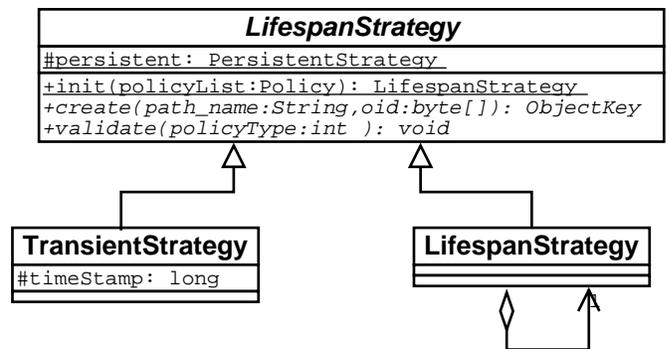


Figure 6: ZEN’s Lifespan Strategy

The responsibilities of the strategy include the creation of object ids for objects registered with the POA and validation of object keys contained in the client requests.

When asked to activate an object, the POA uses the `create()` method to generate an object id for the CORBA object. The object id generated depends on the concrete strategy loaded into the ORB. For example, the object id gener-

ated by a transient transient POA has a time stamp. When a client request is received, the `validate()` method of the `LifespanStrategy` determines whether it was this POA that generated the object id. If the POA is transient and the above is not true then a `OBJECT_NOT_EXIST` exception is returned to the client. In the persistent case, the adapter activator of the closest existing ancestor is used to create the POA automatically.

In ZEN, the persistent strategy does not maintain state specific to a POA, so it can be implemented as a flyweight `PersistentStrategy` object.

ActivationStrategy component. This component implements the `Activation` policy, which is used to specify whether implicit activation of servants is supported in the POA. If the implicit activation policy is active, it causes two things to happen when the servant method `{_this()}` is called:

1. The servant is registered with the POA and
2. The object reference for that servant is implicitly created.

Without this policy, the server must call either `activate_object()` or `activate_object_with_id()` to achieve this effect.

ZEN uses the `ActivationStrategy` shown in Figure 7, to implement the behavior required by the `ImplicitActivation` policy. The `validate()` method is invoked to check if

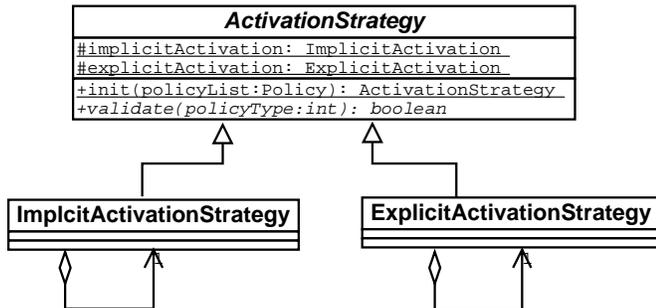


Figure 7: ZEN’s Activation Strategy

implicit activation is permitted, on this POA. Depending on the concrete strategy that is plugged into the ORB, the operation returns true or false. For example, the `servant_to_id()` and `servant_to_reference()` operations use the method to check if implicit activation is allowed.

Both concrete strategies, `ImplicitActivationStrategy` and `ExplicitActivationStrategy` do not maintain any state within them and hence are implemented as flyweights for space utilization.

4.3.2 Secondary POA Components

The secondary components in ZEN are strategies whose behavior depends on the values of primary strategies. These dependencies can lead to conflicts. When two policies cannot co-exist they are said to be in *conflict*. If the policy list specified at POA creation has conflicts, the strategies would also be in conflict. For example, if the `ImplicitActivation` policy value is `IMPLICIT_ACTIVATION`, the `IdAssignment` policy value cannot be `USER_ID`.

In ZEN, these conflicts are identified at strategy creation time (*i.e.* before processing client requests), and appropriate response can be taken (*e.g.*, raise an exception to the user, apply reflection to automatically select a non-conflicting set of policies, etc).

IdAssignmentStrategy component. This component implements the `IdAssignment` policy, which is used to specify whether object ids in the POA are generated by the application or by the POA. The possible object id assignment policy values are either `User-assigned` or `System-assigned`. Moreover, if the POA has both the system-id policy and persistent lifespan policy enabled, object ids generated must be unique across all instantiations of the same POA. If the POA has the `ImplicitActivation` policy, this policy’s value cannot be `USER_ID`. This subtle interaction between POA policy values is implicit, but must be enforced at POA creation time.

In ZEN, the `IdAssignmentStrategy` class models the behavior required by the `Id Assignment` policy. The interface of the `IdAssignmentStrategy` is shown in Figure 8. The `init()` factory method, that creates the concrete strat-

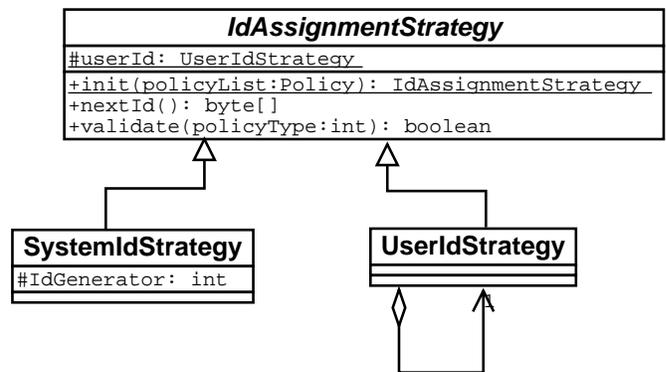


Figure 8: ZEN’s Id Assignment Strategy

egy also checks for conflicts and raises the `WRONG_POLICY` exception if necessary. The only responsibility of this strategy is to generate object ids for registering objects with the POA.

Under certain conditions, POA operations, such as `activate_object()` and `servant_to_id()`, can activate servant using POA generated object ids. The

{nextId()} method generates the new object id if the system id policy value is present in the POA. If the user id policy value is present, a WRONG_POLICY exception is raised. The semantics of incorporating the above behavior is present each of the concrete strategies.

The UserIdStrategy does not maintain any state specific to a POA, so it is designed as a flyweight.

IdUniquenessStrategy component. This component implements the *IdUniqueness* policy, which is used to specify if the servants activated in the POA must have unique object ids. If the policy value is unique id, servant activated by the POA support exactly one object Id. With the multiple id policy, servants activated by the POA may support multiple object Ids. The use of unique id policy value in conjunction with the *NonRetain* policy is meaningless. The OMG specification allows the ORB not to report an error if this combination is used, in ZEN this is considered to be in conflict and a WRONG_POLICY exception is raised.

The IdUniquenessStrategy enforces the behavior required by the policy. Figure 9 shows the class diagram and the concrete strategies that extend the IdUniquenessStrategy. The

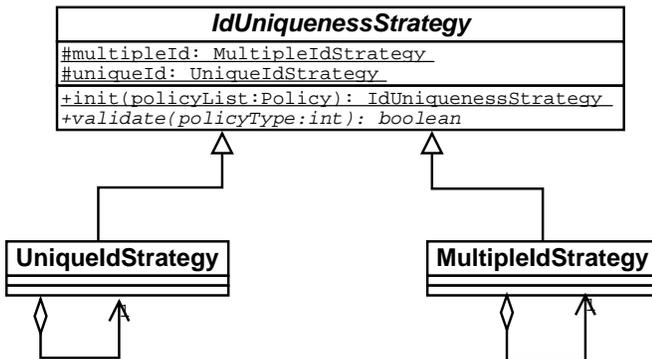


Figure 9: ZEN's Id-Uniqueness Strategy

{validate()} methods is used by the POA to check for the policy value associated with the POA. For example, {activate_object()} operation before activation of an already existing servant, calls the {validate()} method to check if re-registration is permitted. Both the concrete strategies do not maintain any state within them and hence are designed as flyweight references.

ServantRetentionStrategy component. This component implements the *Servant Retention* policy, which is used to specify if the POA retains the active servants in an active object map. This policy can either have retain or non-retain as the possible policy values. Some combinations of POA policies are not allowed. For example, the *ServantRetention* policy may have a value of NON_RETAIN and an *ImplicitActivation* policy may have a value of IMPLICIT_ACTIVATION, but

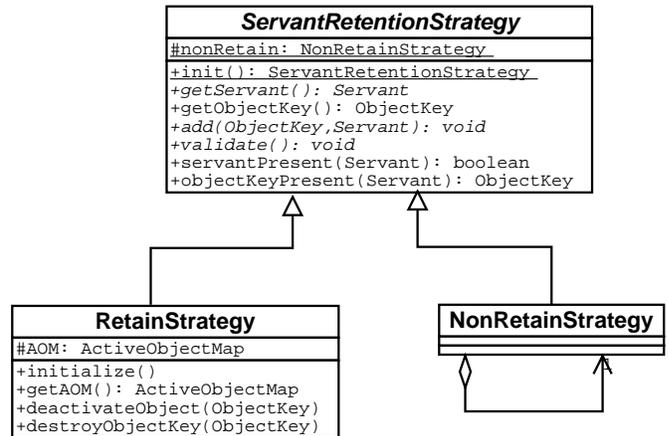


Figure 10: ZEN's Servant Retention Strategy

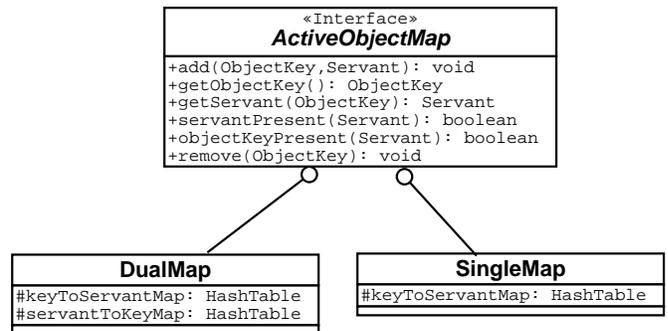


Figure 11: ZEN's Active Object Map Interface

they cannot have those values simultaneously since they conflict with one another. Again, these implicit and subtle issues must be enforced at POA creation time.

In ZEN, the *ServantRetentionStrategy* models the behavior required by the *ServantRetention* policy. Figure 10 shows the concrete strategies that extend the *ServantRetentionStrategy*.

The *ServantRetentionStrategy* maintains an active object map where the association between the CORBA object and the servant is maintained. If a POA has unique id and retain policies, there exists a one-to-one relationship between the object ids and servants and vice versa. In this case, operations *servant_to_id()* and *servant_to_reference()* support reverse lookups (e.g., given a pointer to a servant, return the object associated with it). To speed up these operations, ZEN uses a reverse map that maps servants to their object ids. Since this reverse map is only needed in certain cases, the active object map is further strategized into *SingleMap* and *DualMap*. Figure 11 shows the active object map interface.

The optimization describe above further reduces the foot-

print of the POA when a multiple id policy value is used. In the traditional approach, operations requiring lookups on the active object map would have to be preceded by guard conditions that check if the POA has the retain policy. In ZEN, depending on the concrete strategy in place, these either produce the desired behavior or raise the `WRONG_POLICY` exception.

The `NonRetainStrategy` encapsulates the mechanism of enforcing the non-retain policy. This strategy does not maintain any state specific to the POA and is implemented as a flyweight. All POA's having the non-retain policy have references to this flyweight.

RequestProcessingStrategy component. This component implements the *RequestProcessing* policy, which specifies how the POA should process requests. On receipt of a request, the POA based on the request processing policy value can do one of the following.

- **Consult the active object map only.** The POA using the object id searches the map for the associated Servant. It then uses that servant to process the request. If unsuccessful, an exception is returned to the client.
- **Use a default servant.** If the POA has the *Retain* policy and Step 1 is unsuccessful, then a default servant if present is used to service the request. If a default servant has not been associated or the POA does not have the policy an exception is returned to the client.
- **Use Servant Manager.** If the POA has the *UseServant-Manager* policy, the application supplied manager can be asked to incarnate/activate a servant for the object id. This servant is used by the POA to service the request. Depending on the *ServantRetention* policy, the servant manager can either be a servant activator or a servant locator.

The *RequestProcessing* policy is strategized along the three alternate courses of action mentioned above. Figure 12 shows the class diagram for the `RequestProcessingStrategy`. `ActiveObjectMapOnlyStrategy` encapsulates the logic of request dispatch if the active object map only policy is used. The POA uses the `{handleRequest() }` method of the base strategy strategy to service requests.

The `DefaultServantStrategy` is associated with the POA if the appropriate policy value is used. Depending on the servant retention policy value, this strategy either consults the active object map first for request dispatch, or uses the default servant. If the non-retain policy value is used the POA, the servant is directly used. In either of the cases, if no servant is associated with the POA, an exception is raised.

The `ServantManagerStrategy` is associated with the POA if the *Use Servant Manager* policy value is specified. Moreover, depending on the

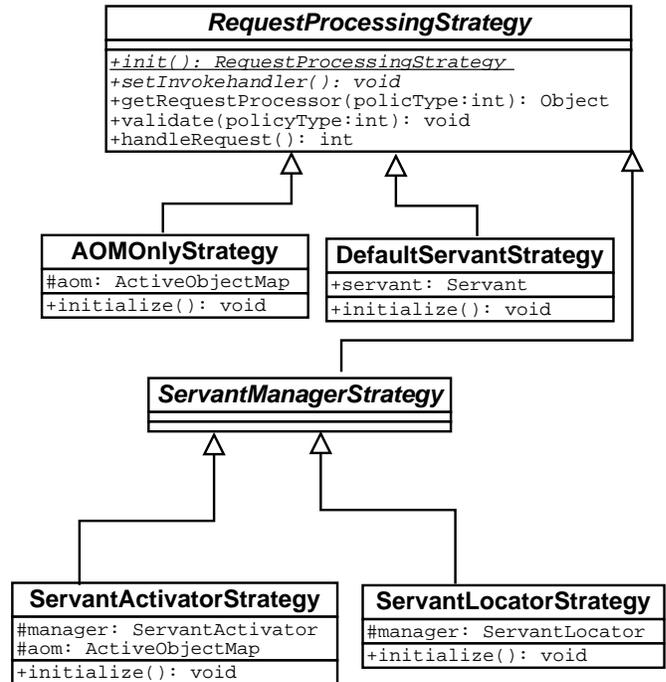


Figure 12: Request Processing Strategy

ServantRetention policy for the POA, this is strategized into a `ServantActivatorStrategy` or a `ServantLocatorStrategy`. Each of these concrete strategies have the semantics necessary for request dispatch.

In a traditional POA implementation, each time a POA receives a request it must check the value of the request processing policy. In ZEN, however, the semantics of request processing in each case is present in the concrete strategy for that policy, so the policy value need not be checked at all.

4.4 ZEN's POA Design Summary

Section 4.2 presented the alternative architectures that help reduce the footprint of ZEN's POA. Figure 4 shows the fine-grain architecture of the ZEN POA. POA A and B have the *UserId* and *Persistent* policy values present and consequently share the corresponding strategy components. Moreover, the `ServantRetentionStrategy` and `ActiveObjectMapOnlyStrategy` components of POA A share the active object map, as shown by the flyweight references in the figure. POA B illustrates the scenario in which the POA has the minimal footprint *i.e.*, maximal number of flyweight references.

A monolithically designed POA is a large body of code,

whereas the micro-POA in ZEN consists of several indivisible components associated at creation time. In terms of implementation ease, the monolithic design is easier to implement, whereas the micro-POA design requires subsetting and grouping of behavior of the POA into well-defined components. However, it is easier to add new policies and change semantics specified by individual policies in a micro-POA design than in a monolithic-POA design.

POAs of several ORBs, such as TAO [6] and JacORB [20], implement the monolithic design. In ZEN, we have embraced the micro-POA design since it provides the advantages mentioned in Section 4. Below, we summarize how our highly-modular pluggable designs have achieved the goals for ZEN’s POA outlined in Section 4.1:

- **Minimize footprint.** Application of the Virtual Component pattern to both the entire POA (coarse-grain) and to individual components of the POA (fine-grain) allows optimal selection of just the POA components required for each application. The footprint measurement presented in Section 5 demonstrate substantial footprint savings in a variety of usage scenarios.
- **Ease adaptation to new changes in the CORBA specification.** If a the behavior specified by a POA policy were to be changed, only the appropriate strategy would be affected. For example, if the semantics of the *ActiveObjectMapOnly* policy value were to change, only the specific class for handling that policy need to change. ZEN’s micro-POA design therefore focuses the point of change to one component.
- **Facilitate addition of ORB-specific policies.** The highly-modular design of ZEN eases the addition of new ORB-specific policy extensions for existing policies. For example, if we wish to add a new policy value, *e.g.*, *UseDatabaseManager*, for the *RequestProcessing* policy, we only need to derive a new class from the *RequestProcessingStrategy* that implements the behavior of this new policy. Instances of this new policy class can be plugged into the POA at creation time using factories. This addition would not require recompilation of the POA – only compilation of the classes for concrete strategy and for the concrete new policy.

ZEN’s highly-modular pluggable design has the additional benefit of improving the time needed to create a POA. A monolithic POA needs to check the request processing policy value with every request for request dispatch. In contrast, the conflicts among the policies are identified in ZEN while the strategies are being created. Based on the policy present in the POA, the appropriate request processing component can be plugged in. Thus, there is no need to first create the entire policy list and then iterate over the list repeatedly to identify

the conflicts among the policies. This capability reduces the creation time of the POA considerably, as shown in Figure 14.

5 Empirical Results

This section presents the results of both blackbox and whitebox benchmark measurements. These measurements were performed on a dual-CPU Intel Xeon 1,700 Mhz processor with 256 KB of main memory. The experiments compare the results obtained from ZEN version 0.8 alpha with that of JacORB [20] version 1.4 beta 4. All tests were conducted on JVM version 1.4.0 running on Linux OS 2.4.18.

5.1 Blackbox Experiments

Blackbox experiments do not instrument the software internals when evaluating the performance tests. In our case, each ORB was benchmarked end-to-end without knowledge of its internal structure. Moreover, the benchmarks used operations published by the ORB interfaces and did not modify or restructure the ORB internals.

To eliminate differences in the POA configurations, the following properties were set in both ZEN and JacORB:

1. Logging was turned off
2. POA monitoring was turned off for JacORB in the properties file.
3. The number of threads in the thread pool was set to 10
4. Maximum queue size was set to 100 and
5. No priority was set for the threads doing the request processing.

5.1.1 Root POA Metrics

Overview. As discussed in Section 3.2, the root POA is an integral part of every CORBA server and is always present, whether or not any other child POAs exist. A root POA suffices for many applications, unless the server needs to provide different QoS guarantees, such as object reference persistence. Thus, minimizing the footprint of the root POA is vital to minimizing server footprint.

This test measures the increase in footprint after the root POA has been associated with the ORB. The memory increase prior to and after the call to the `resolve_initial_references()` gives the foot print increase contributed by the root POA.

Results and analysis. Figure 13 illustrates that the footprint of the root POA in Zen is 61 kbytes, while that of JacORB is 180 kbytes. Thus, ZEN’s root POA is three times smaller than JacORB. In ZEN, the creation of the root POA results in initialization of all the base abstract strategies and the creation of the appropriate concrete strategies for the root POA policies.

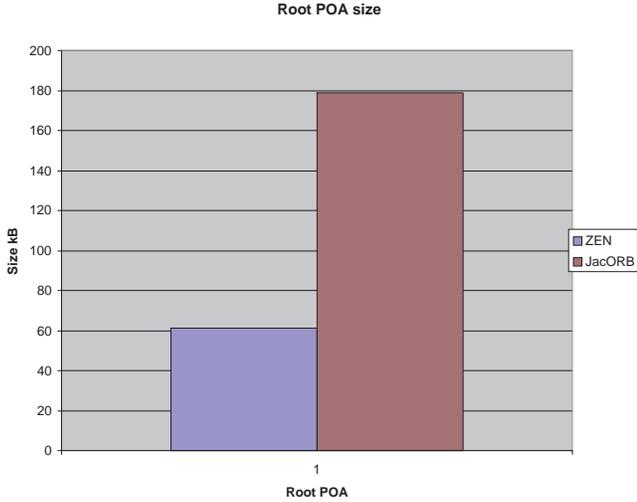


Figure 13: Root POA Footprint

The root POA maintains the maximum state among all POAs in ZEN. This small footprint bolsters the micro POA design in the ZEN. Since JacORB is designed monolithically, it suffers from a higher footprint overhead.

5.1.2 Child POA Creation Time Metrics

Overview. We expect the design of the ZEN POA to reduce the creation time of the POA since most of the concrete strategies are implemented as flyweights. In many applications, a POA could also be created as a side-effect of an upcall on the servant. In this scenario, a slow creation time for a child POA could decrease throughput and increase jitter.

This test measures the variation in creation time with the number of POAs created. The child POAs that are created have the same policies as the root POA. In ZEN, default policy values (which are those used by the root POA) require more memory and are more expensive to create than are non-default policy values, so this test exercises the worst-case scenario.

Results and analysis. Figure 14 shows that both ZEN and JacORB grow linearly with the increase in the number of child POAs. However, the rate of increase for ZEN is smaller than that of JacORB. From the samples, the average time for POA creation in ZEN is 0.17 millisecond while that of JacORB is 0.88 milliseconds. On average, therefore, JacORB is seven times slower than ZEN for child POA creation.

5.1.3 Child POA Footprint Metrics

Overview. A key design goal of the micro POA architecture is to minimize footprint for DRE systems. This paper would therefore be incomplete without the results for the footprint

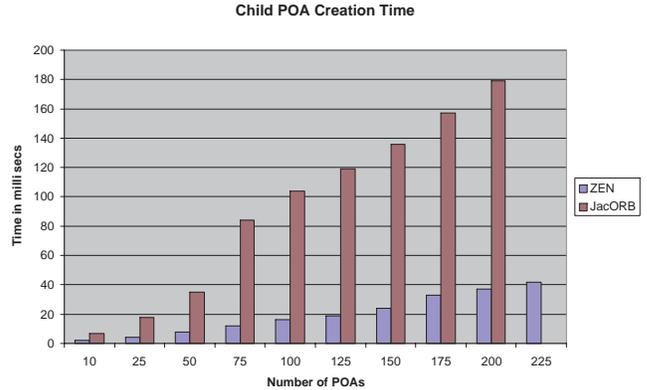


Figure 14: Child POA Creation Time

analysis for the child POAs. A CORBA server creates child POAs for the CORBA objects if the QoS parameters require persistent object reference, memory reduction (*e.g.*, associating multiple objects with a default servant), etc.

This test measures the variation of footprint with the number of POAs created. The increase in footprint prior to and after the call to the `create_POA()` method is measured in each of the case. Each of the POAs created have the following policy values: (1) *NonRetain* policy, (2) *ServantManager* policy, and (3) *UserIid* policy. This combination is chosen as since it minimizes the footprint for the ZEN POA.

Results and analysis. Figure 15 depicts the memory increase with the number of child POAs. Both JacORB and ZEN, grow linearly. Though ZEN seems to be constant, its rate of growth is simply very low. The average size of the child POAs in ZEN is 35 kbytes, while that of JacORB is around 300 kbytes. Thus, on average JacORB's POA is larger by a factor of 8.

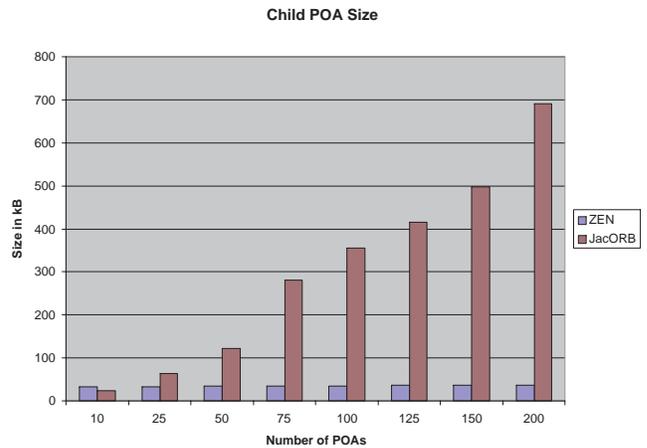


Figure 15: Child POA Footprint Results

There could be several factors contributing to the footprint increase than the POA per se. In the case of JacORB some of these factors contribute significantly towards increased footprint. For example, in JacORB for every POA created:

1. A request controller is associated with each POA. The Request controller is a thread pool that manages the request processing for that POA. In ZEN, the thread pool is associated with the ORB and not with each POA. For example, when 100 child POAs are created in JacORB, each POA has its own thread pool and specified channel capacity, adding significantly to the footprint of POA.
2. A POA monitor is initialized with every POA created even if monitoring is disabled in the properties file.

Even if a POA has only default set of policies, however, due to JacORB’s monolithic design it still has the semantics for implementing ORB-specific policies, such as bi-directional GIOP.

For the reasons mentioned above, the relative difference between the sizes of the POAs for ZEN and JacORB is large. Even with these differences, ZEN’s design is more scalable since the footprint increase is minimal with an increase in the number of POAs created.

5.1.4 Cost in Memory per Object Activation

Overview. One of the key functionality of the POA is to generate object references. Thus, the cost in memory per object activation provides an indication of footprint increase as servants are associated with the POA. In this test, a servant is activated multiple times. Each time the servant is activated the POA creates an association between the servant with its object id in the active object map. The increase in footprint prior to and after the activation of servants is measured for each case.

Results and analysis. Figure 16 illustrates that memory increases linearly with an increase in the number of activations. The average cost of memory per object activation is around

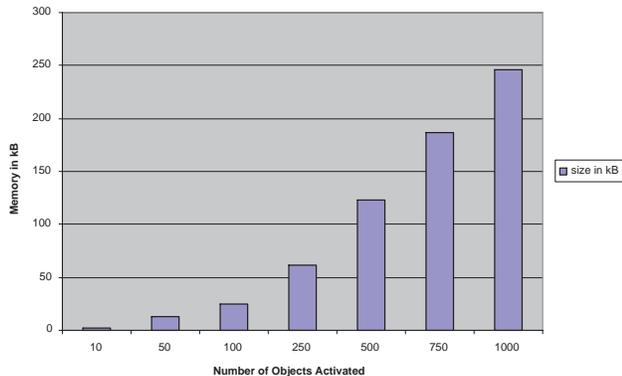


Figure 16: Cost in Memory per Object Activation

240 bytes. Some factors contributing to the increase include: the creation of object key for the servant, creating a hash map structure for servant, and registering the association between the object key and the hash map structure in the active object map.

At the writing of this paper, JacORB does not support the association of multiple objects with the same servant. Hence, it is not possible for us to determine the growth of memory with the number of object activated. We have reported this as a bug to the JacORB developer group.

5.2 Whitebox Experiments

Whitebox benchmarks are a performance evaluation technique where explicit knowledge of software internals is used to select the benchmark data. Unlike blackbox benchmarks, whitebox tests uses instrumentation of software internals to evaluate performance. Below, we present whitebox experiments on POA-related demultiplexing that were conducted using JacORB and ZEN.

A key function of a POA is demultiplexing of requests to servants. Demultiplexing in conventional CORBA implementations is typically inefficient and unpredictable. For instance, [11, 12] show that conventional ORBs spend 17% of the total server time processing demultiplexing requests. Constant time request demultiplexing regardless of organization of the POA hierarchy or number of POAs, servants, or operations, allows an ORB to provide uniform, scalable QoS guarantees to real-time applications.

In the whitebox experiments, a single-threaded client issued IDL operations at the fastest possible rate using a “flooding” model. Timers used internally within the ORB Core measure the *dispatch time* for each client request. Dispatch time was measured as the time taken for request processing from the time when the appropriate POA is found until the time the request was delivered to the servant. This definition of dispatch time eliminated the demarshaling overhead and measured the demultiplexing time required by the POA.

The average dispatch time was calculated by the experiments. We also measured the variation of the dispatch time with the depth of the POA hierarchy, breadth of the hierarchy, the number of objects registered with a POA. These metrics underscore the dependency of dispatch time on the aforementioned factors.

Below, we present the results from experiments that measure the three main demultiplexing steps that affect POA dispatch time.

5.2.1 POA Demultiplexing

Overview. The first step in the request delivery is to determine the POA that will service the request. As seen earlier,

the POA hierarchy can be arbitrarily deep and broad. Traditionally, ORBs performed a look up for each level of the POA hierarchy until the “leaf” POA is reached. This linear search strategy is expensive, however, and increases the demultiplexing time greatly. This test measures the variation in the POA demultiplexing time with the increase in the depth, breadth and number of objects of the POA hierarchy.

Results. Figure 17 shows the POA demultiplexing time analysis for ZEN. Figure 18 compares the effect of the depth

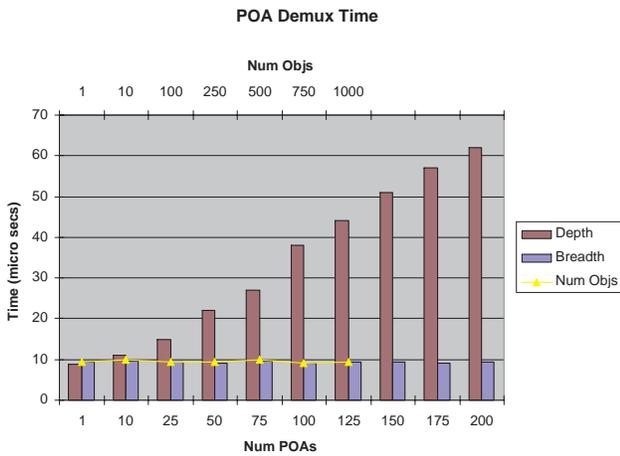


Figure 17: POA Demultiplexing Time Analysis for ZEN POA of the POA hierarchy on the demultiplexing times for ZEN and JacORB.

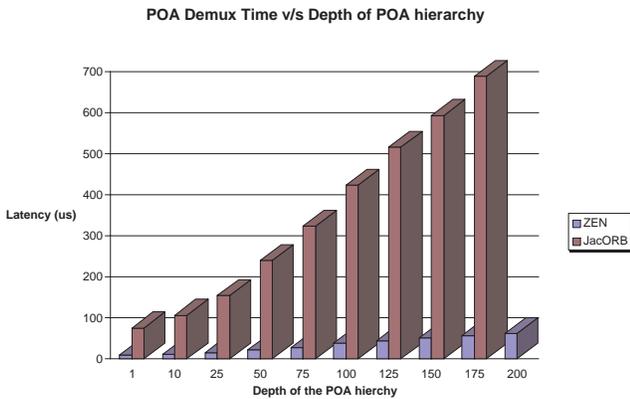


Figure 18: POA Demux Time v/s Depth of the POA Hierarchy

The latency of ZEN and JacORB both increase with the depth of the POA hierarchy. However, ZEN’s degradation is more graceful than JacORB’s. This behavior stems from the fact that ZEN flattens the POA hierarchy so the appropriate POA servicing the request can be determined in a single

lookup.¹ In contrast, JacORB incurs a lookup for ever depth of the POA hierarchy, leading to the steep degradation of its performance as the POA hierarchy deepens.

ZEN stores the complete POA path name (similar to the path name of a directory or URL), along with the POA reference, in a hash table.² The linear increase in ZEN is due to the increase in the length of the POA path name with the deepening of the POA hierarchy. This increase in path name contributes to comparison time needed for a successful lookup, though ZEN is still much faster than JacORB.

Figure 19 compares the variation in the demultiplexing time with the breadth of the POA hierarchy. This figure shows

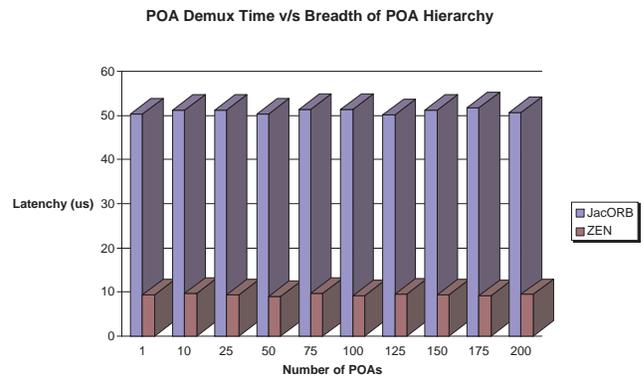


Figure 19: POA Demux Time v/s Breadth of POA Hierarchy

that the POA demultiplexing time remains constant for both ZEN and JacORB as the breadth and the number of objects in the POA hierarchy are increased. However, the latency in JacORB is much higher than in ZEN, which can be attributed to JacORB’s sequential traversal of the POA namespace.

To prevent the linear increase of demultiplexing time with the depth of the POA hierarchy, an *active demultiplexing* strategy (such as the one used in TAO [18]) should be used. Predictability is essential for real-time systems, so our first non-alpha release of ZEN will use active demultiplexing.

5.2.2 Servant Demultiplexing

Overview. Once the ORB Core demultiplexes a client request to the right POA, this POA demultiplexes the request to the corresponding servant. In this test, the variation of servant demultiplexing time is measured with the number of active objects in the POA.

¹If the lookup fails, the POA hierarchy is traversed sequentially and the required POAs are activated using the user-supplied adapter activator.

²We are currently using Hashtable provided by Java, *i.e.*, `java.util.Hashtable`.

Results. Figure 20 shows the variation of servant demultiplexing time with the increase in the number of servant in the active object map. In ZEN, we are currently using dynamic

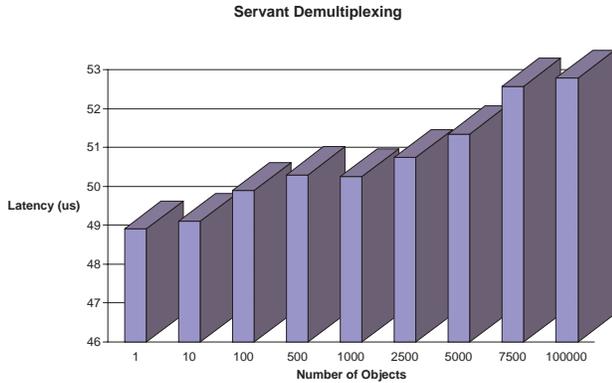


Figure 20: Servant Demultiplexing Time

hashing for this stage of demultiplexing (as before, we use the hash table provided by Java’s `java.util.Hashtable`). To locate the appropriate servant, the POA uses the object id part of the object key to look up the servant in the active object map, hence this stage is independent of the POA hierarchy. There are three steps in locating a servant:

1. Parsing the object key
2. Checking if the key is present and
3. Looking up the appropriate hash data structure that contains the servant.

ZEN’s servant demultiplexing implementation incurs a significant overhead during hash table lookup operations that contribute to its latency, which stems from the synchronized methods of Java’s `java.util.Hashtable` class. Some overhead is also necessary to compute the hash function, which uses the `hashCode()` method of Java’s `String` class. In addition, there is a gradual increase in the latency with the increase in the number of active objects in the POA. As stated above, in the first non-alpha release of ZEN we will also be implementing an active demultiplexing scheme for this stage of demultiplexing.

JacORB does not let the user associate multiple objects for the same servant, even with the *Multiple-Id* policy value for the POA. Hence we are unable to present its results.

5.2.3 Operation Demultiplexing

Overview. The final step at the Object Adapter layer involves demultiplexing a request to the appropriate skeleton. This skeleton then demarshals the request and dispatches the designated operation upcall in the servant. For real-time embedded systems, operation demultiplexing should be efficient, scalable, and predictable.

To prevent variations in operation dispatch time with number of methods, we use a Java-variant of GPERF [21] called JPERF, which is an open-source perfect hash function generator. JPERF automatically constructs perfect hash functions from a user-supplied list of keywords. Perfect hashing is predictable and efficient, and outperforms other search techniques, such as binary search and dynamic hashing. JacORB uses dynamic hashing for this stage (both ZEN and JacORB to use `java.util.Hashtable` class for our experiments).

Results. Figure 21 illustrates the variation in operation demultiplexing time as the number of methods increases. This

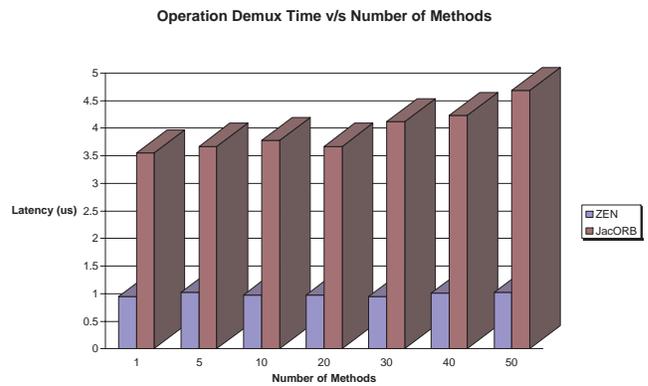


Figure 21: Operation Demultiplexing Time

figure shows that operation lookup is constant in ZEN and does not vary with the increase in the number of methods. Latency in ZEN is $\sim 1 \mu\text{sec}$, while that of JacORB is $\sim 4 \mu\text{secs}$. The higher latency of JacORB stems from its dynamic hashing overhead, which also increases the lookup time as the number of methods increase.

6 Related Work

TAO’s Portable Object Adapter. TAO is an open-source, high-performance real-time ORB written in C++. TAO had the first implementation of the POA specification [18]. The design of the POA is based on several design patterns, many of which have been adopted in ZEN. TAO also uses an optimized set of request processing strategies [22], e.g., active demultiplexing and perfect hashing. These strategies allow TAO’s POA to provide constant-time lookup of servants based on object keys and operation names contained in CORBA requests.

In ZEN, we have based our design on several optimizations used in the design of TAO’s POA, e.g., perfect hashing for $O(1)$ time servant lookups and flattening the POA hierarchy to prevent a lookup for every level in the POA hierarchy. Apart

from these, the main optimization present in ZEN is the fine-grain control of the components that are loaded. The extensive use of advanced strategies, such as the Virtual Component, Strategy, and Flyweight patterns, has helped reduce footprint of the ZEN POA. This design does not unduly compromise performance, as shown in Section 5.

JacORB. JacORB [23] is an open-source Java ORB developed at the University of Berlin. Like TAO, JacORB has been widely embraced in the industry. Likewise, JacORB has a monolithic POA design. JacORB also has a POA monitoring GUI that can be used to monitor the operations on the POA and request dispatch. JacORB does not implement some of the request demultiplexing techniques discussed earlier, *e.g.*, perfect hashing or flattening the POA hierarchy that help in bounding lookup times.

Reflective POAs. One other possible solution to fine-grain control over the components of the POA is to apply advanced meta-programming techniques, such as reflection [24, 25, 26], aspect-oriented programming [27], and model-integrated computing [28]. These techniques can be used to auto-generate most of the POA in such a way that only a minimal amount of space is used, while still supporting the standard CORBA APIs. Our future research will focus on exploring this alternative.

7 Concluding Remarks and Future Work

ZEN is a long-term research project with well-defined goals targeting distributed, real-time, and embedded (DRE) applications. We learned from our experience with TAO that a small memory footprint must be achieved during the initial design phase. We also learned that configuration must be automated as much as possible, to avoid placing an onerous burden on application developers.

The POA has been decomposed into a highly-modular, loosely coupled set of Virtual Components that may be loaded either on a fine-grain or a coarse-grain bases - depending on the application developers configuration options. This paper presents empirical results that measure the footprint and performance of three alternative POA designs. The conclusion is that considerable footprint savings can result from the fine-grain highly-modular design, without unduly reducing performance relative to existing Java ORBs.

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: Angelo Corsaro, Mayur Deshpande, Sean McCarthy, Carlos O’Ryan, Ossama Othman, Mark Panahi, Krishna Raman, and Sumita Rao.

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] D. C. Schmidt and F. Kuhns, “An Overview of the Real-time CORBA Specification,” *IEEE Computer Magazine, Special Issue on Object-oriented Real-time Computing*, vol. 33, June 2000.
- [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] 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.
- [9] Bollella, Gosling, Brosgol, Dibble, Furr, Hardin, and Turnbull, *The Real-Time Specification for Java*. Addison-Wesley, 2000.
- [10] A. Corsaro, D. C. Schmidt, R. Klefstad, and C. O’Ryan, “Virtual Component: a Design Pattern for Memory-Constrained Embedded Applications,” in *Submitted to the 9th Annual Conference on the Pattern Languages of Programs*, (Monticello, Illinois), Sept. 2002.
- [11] A. Gokhale and D. C. Schmidt, “Measuring the Performance of Communication Middleware on High-Speed Networks,” in *Proceedings of SIGCOMM ’96*, (Stanford, CA), pp. 306–317, ACM, August 1996.
- [12] A. Gokhale and D. C. Schmidt, “Measuring and Optimizing CORBA Latency and Scalability Over High-speed Networks,” *Transactions on Computing*, vol. 47, no. 4, 1998.
- [13] 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.
- [14] 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.
- [15] Center for Distributed Object Computing, “The ZEN ORB.” www.zen.uci.edu, University of California at Irvine.
- [16] S. Maffeis, “The Object Group Design Pattern,” in *Proceedings of the 1996 USENIX Conference on Object-Oriented Technologies*, (Toronto, Canada), USENIX, June 1996.

- [17] Center for Distributed Object Computing, “The ACE ORB (TAO).” www.cs.wustl.edu/~schmidt/TAO.html, Washington University.
- [18] I. Pyarali and D. C. Schmidt, “An Overview of the CORBA Portable Object Adapter,” *ACM StandardView*, vol. 6, Mar. 1998.
- [19] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, Massachusetts: Addison-Wesley, 1995.
- [20] G. Brose, N. Noffke, and S. Müller, “JacORB 1.4 Programming Guide.” http://jacorb.inf.fu-berlin.de/ftp/doc/programmingGuide_1.4.pdf, 2001.
- [21] D. C. Schmidt, “GPERF: A Perfect Hash Function Generator,” in *Proceedings of the 2nd C++ Conference*, (San Francisco, California), pp. 87–102, USENIX, April 1990.
- [22] I. Pyarali, C. O’Ryan, D. C. Schmidt, N. Wang, V. Kachroo, and A. Gokhale, “Applying Optimization Patterns to the Design of Real-time ORBs,” in *Proceedings of the 5th Conference on Object-Oriented Technologies and Systems*, (San Diego, CA), USENIX, May 1999.
- [23] G. Brose, “JacORB: Implementation and Design of a Java ORB,” Sept. 1997.
- [24] Gordon S. Blair and G. Coulson and P. Robin and M. Papathomas, “An Architecture for Next Generation Middleware,” in *Proceedings of the IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing*, (London), pp. 191–206, Springer-Verlag, 1998.
- [25] Fábio M. Costa and Gordon S. Blair, “A Reflective Architecture for Middleware: Design and Implementation,” in *ECOOP’99, Workshop for PhD Students in Object Oriented Systems*, June 1999.
- [26] F. Kon, F. Costa, G. Blair, and R. H. Campbell, “The Case for Reflective Middleware,” *Communications ACM*, vol. 45, pp. 33–38, June 2002.
- [27] 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.
- [28] J. Sztipanovits and G. Karsai, “Model-Integrated Computing,” *IEEE Computer*, vol. 30, pp. 110–112, April 1997.