Design and Performance of a Dynamically Configurable, Messaging Protocols Framework for Real-time CORBA

Raymond Klefstad, Sumita Rao, and Douglas C. Schmidt
{klefstad, srao, schmidt}@ece.uci.edu
Electrical and Computer Engineering Dept.
University of California, Irvine, CA 92697, USA*

Abstract

As distributed object computing (DOC) middleware standards and implementations evolve over time, they invariably must support a growing number of protocols to remain interoperable. Memory footprints of middleware implementations can become large if a concerted effort is not made to prevent this growth. Large middleware memory footprints are problematic for important classes of distributed, real-time, and embedded (DRE) applications that have stringent memory constraints. This paper makes two contributions to the design of middleware to address key challenges of developing DRE applications. First, it describes the design of dynamically configurable general inter-ORB protocols (GIOP). Second, it explains how we applied patterns and Java features to minimize and customize the memory footprint of messaging protocols used in DRE middleware.

Keywords: Distributed Systems, Real-time Systems, Embedded Systems, General Inter-ORB Protocol (GIOP), Real-time CORBA, Real-time Java.

1 Introduction

Over the past decade, distributed object computing (DOC) middleware frameworks, such as CORBA [1], COM+ [2], Java RMI [3], and SOAP/.NET [4], have emerged to reduce the complexity of developing distributed applications. DOC middleware simplifies application development for distributed systems by off-loading the tedious and error-prone aspects of distributed computing from application developers to middleware developers. It has been used successfully in large-scale 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 distributed, real-time, and embedded (DRE) applications, just as CORBA has for large-scale business systems. Real-time CORBA is designed for applications with hard real-time requirements, such as avionics mission computing [6], as well as those with stringent soft real-time requirements, such as telecommunication call processing and streaming video [7].

Programming DRE applications is hard because QoS properties must be supported along with the application software and distributed computing middleware functionality. DRE applications have historically been custom-programmed to implement these QoS properties. While some aspects of these challenges have been addressed in developing applications for mainstream distributed systems, relatively little has been done to meet these challenges for applications in DRE systems.

Interprocess communication (IPC) in CORBA-based applications involves the exchange of messages between ORB endsystems running on different types of machines using TCP/IP. The messaging protocol for CORBA is specified by General Inter-ORB Protocol (GIOP), which is a family of protocols that define standard message types and formats. The methods supporting the GIOP messaging protocol can be a significant contributor to memory footprint size [8]. A time and space efficient implementation of GIOP messaging protocols is therefore desirable to reduce footprint without unduly affecting operational throughput between ORB endsystems.

This paper discusses the design and performance of dynamically configurable GIOP messaging protocols for an open-source Real-time CORBA object request broker (ORB) called ZEN [9].[1]. ZEN is implemented using Real-time Java and is designed to eliminate common sources of overhead and non-determinism in ORB implementations. We illustrate how ZEN's pattern-oriented software architecture [10] systematically reduces memory footprint by allowing the selection of a minimal subset of ORB capabilities used by an application.

The remainder of this paper is organized as follows: Section 2 presents a brief overview of ZEN; Section 3 explains GIOP message handling, the challenges related to conforming to an evolving middleware specification, and how message handling is handled in ZEN; Section 3.4 describes how ZEN's GIOP messaging can be configured using four different strategies to improve flexibility and minimize memory foot-

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

[1] ZEN can be obtained from `http://www.zen.uci.edu`

print; Section 4 provides empirical measurements for ZEN's alternative messaging strategies; Section 5 compares our work on ZEN's GIOP framework with related work; and Section 6 presents concluding remarks and outlines our future directions.

2 An Overview of the ZEN Real-time ORB

ZEN [9] is a Real-time CORBA ORB implemented using Java, thereby combining the benefits of these two standard technologies. A detailed overview of ZEN appeared in [9]. This section therefore presents a brief summary of research goals addressed by the ZEN project and an overview of ZEN's architecture.

To address key challenges faced by developers of DRE applications, the research goals of the ZEN project are as follows:

- Provide a full range of CORBA services for distributed applications.
- Reduce middleware footprint to enable memory-constrained embedded applications development.
- Achieve satisfactory level of throughput and scalability.
- Make the ORB easier for application developers to configure and maintain.
- Support real-time QOS requirements, such as bounded jitter, elimination of priority inversion, and low startup latency.
- Make the ORB easy to extend to handle new requirements, such as new message types or new message versions.
- Allow both static and dynamic configuration, to allow the application developer to choose a tradeoff between maximal efficiency and flexibility.

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 allowing flexible middleware configuration. We call this design the *micro-ORB* design, whereas we call the traditional ORB design the *monolithic-ORB* design.

In a monolithic design for an ORB, all the code for various middleware features and their alternative implementations are tightly coupled within the ORB. Likewise, in a monolithic design for GIOP message handling, *all* the code for handling the message types and versions are structured in such a way that they must all be included for each application, whether they are used or not by any particular application. In contrast, in the micro-ORB design, each ORB service itself is decomposed into smaller components. As shown in Figure 1, the following components can be configured flexibly into the ZEN micro-ORB architecture:

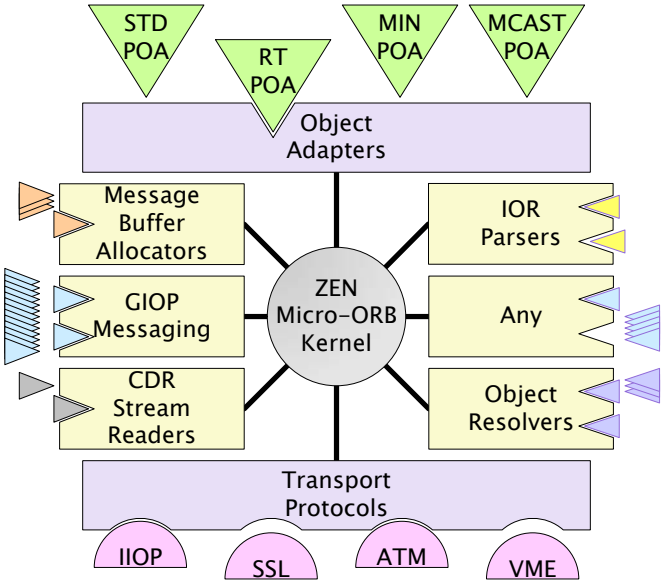

Figure 1: Micro-ORB Architecture of ZEN

1. **Object adapters**, which map incoming client requests to appropriate servants within a server.
2. **GIOP message handling**, which reads and writes standard messages to/from various protocol transports.
3. **Protocol transports**, such as TCP/IP, UDP or shared memory.
4. **Object resolvers**, which handle strings passed into the ORB's `resolve_initial_references()` factory method.
5. **IOR parsers**, which handle parsing of various formats of inter-operable ORB references, such as `"IOR:"`, `"FILE:"`, or `"HTTP:"`.
6. **Any data type handlers**, which allow the transfer of any possible CORBA data type to support generic services, such as the CORBA Naming and Event Services.
7. **Buffer allocators**, which allow application developers to choose from a variety of different dynamic storage allocation algorithms to suit their particular needs.
8. **Common Data Representation (CDR) Streams**, which allow the exchange of messages between CORBA ORBs that isolate applications from hardware variations related to data format and data sizes.

Based on our earlier work with The ACE ORB (TAO) [6], we identified these eight core ORB components as candidates to be factored out of the ORB to reduce its memory footprint and increase its flexibility. We call the remaining portion of code the *ZEN kernel*.

Each ORB component itself is decomposed into smaller pluggable components that can be loaded into the ORB on-demand when needed. We apply the *Virtual Component* pattern [11] throughout ZEN to decompose and factor most of the unused or rarely used components out of memory. This pattern provides an application-transparent way of loading and unloading components that implement middleware software functionality. ZEN's pluggable design makes it a useful research platform, because alternative implementations of various ORB components can be configured easily and profiled with standard benchmarks to determine their utility empirically.

3 GIOP Message Handling in ZEN

This section gives an overview of GIOP messaging and outlines the challenges of developing standard middleware that conforms to an evolving middleware specification.

3.1 An Overview of GIOP Messaging

The General Inter-ORB Protocol (GIOP) defines the standard format of messages that can be sent between clients and servers using CORBA-compliant ORBs. There are eight different types of GIOP messages, each with a unique format as shown in Figure 2. The GIOP message types include Request,

Message Type	Originator	Value	GIOP Versions
Request	Client	0	1.0, 1.1, 1.2
Request	Both	0	1.2 with BirDir GIOP in use
Reply	Server	1	1.0, 1.1, 1.2
Reply	Both	1	1.2 with BirDir GIOP in use
CancelRequest	Client	2	1.0, 1.1, 1.2
CancelRequest	Both	2	1.2 with BirDir GIOP in use
LocateRequest	Client	3	1.0, 1.1, 1.2
LocateRequest	Both	3	1.2 with BirDir GIOP in use
LocateReply	Server	4	1.0, 1.1, 1.2
LocateReply	Both	4	1.2 with BirDir GIOP in use
CloseConnection	Server	5	1.0, 1.1, 1.2
CloseConnection	Both	5	1.2
MessageError	Both	6	1.0, 1.1, 1.2
Fragment	Both	7	1.1,1.2

Figure 2: GIOP Message Types

Reply, LocateReply, LocateRequest, CloseConnection, MessageError, CancelRequest and Fragment. We briefly explain the purpose of each GIOP message type below:

- A client can invoke an operation on the server object (called a "servant") by sending it a Request message. This type of message contains all the information that is required to make a remote method invocation.
- A servant responds to a client's invocation by sending a Reply message containing the response to the Request.
- The LocateRequest and the LocateReply messages are sent to query the current location of a servant.
- The CloseConnection message is sent by the server to inform the client not to send any further Request messages on the connection since the connection will be closed.
- In case a GIOP message is received with a bad header, a GIOP MessageError message is sent to the Request initiator. This message can be sent either to a client or to a server.
- The Fragment message allows large messages to be divided into smaller messages having their "following fragment" bit in the message enabled. The last fragment in a series of message fragments has the "following bit" field disabled.

3.2 Problems with a Naive Implementation of GIOP Messaging

A full implementation of a GIOP message handler requires providing quite a few methods. For each message type, there are two methods: one method to *marshal* (encode) and another method to *demarshal* (decode) a message of that type. Four of the GIOP messages types vary in three versions of GIOP, specifically versions 1.0, 1.1, and 1.2. Thus, there are 4 (message types) $\times$ 2 (marshal/demarshal methods) $\times$ 3 (different versions) + 4 (message types with no change in version) $\times$ 2 (marshal/demarshal method) = 32 methods required to implement the full GIOP message protocol for each possible version.

Many client/server interactions are simple, requiring only a few of the 32 methods. For example, a pure server receives Request messages and sends Reply messages, and thus requires only a Request message reader and a Reply message writer. Conversely, a pure client sends Request messages and receives Reply messages, requiring only a Request writer and a Reply reader. Peers typically use up to four methods to read and write both Request and Reply messages. In addition, many applications limit their messages to only one version of GIOP. However, clients, servers, and peers must be prepared to handle all 8 possible message types from the various versions. There have been two new versions of GIOP messaging (version 1.1 and 1.2) since the inception of GIOP 1.0 in CORBA 2.2. It is likely that new types and versions of GIOP messages will be defined in the future.

A monolithic-ORB contains code to handle all the possible GIOP messages and versions. Each class for handling a message type defines both the marshal/demarshal method for that message type in the same class. A `switch` statement inside each (de)marshal method may be used to split into the code to handle each of the various versions. Although this is a common ORB design, it has two significant drawbacks:

1. **Large footprint**—It incurs non-trivial amounts of space overhead for all the methods even if some of those methods are not used.
2. **Poor extensibility**—It is hard to modify the ORB to handle a new GIOP version because many class definitions for the marshal and demarshal methods must be modified, recompiled, and re-linked.

In general, when features are added or changed in a monolithic-ORB, the changes are propagated to many parts of the code, which is time consuming and error-prone. As a middleware standard like CORBA continues to evolve, and monolithic-ORB implementations add new versions and methods, their memory footprint grows, making this design unsuitable for memory-constrained applications, such as embedded systems.

3.3 Micro-ORB Design Solutions in ZEN

The eight GIOP messages can be factored out of the ORB core footprint by applying the Virtual Component design pattern [11]. By applying this pattern to ZEN, we ensure that it provides a rich and configurable set of functionality, yet occupies main memory only for middleware components that are actually used.

The GIOP messaging module is typically implemented having the demarshal and the marshal methods for each version combined into one class per message type. These GIOP message marshaling and demarshaling methods for a particular message are not frequently used together, so eliminating unused methods can reduce ZEN's memory footprint, as shown in Figure 3. To factor unused methods out of the memory footprint, the 32 marshaling and demarshaling methods can be implemented using the Strategy design pattern [12]. The intent of this pattern is to define a family of algorithms, encapsulate each one, and make them interchangeable. The Strategy pattern lets the algorithm vary independently from the clients that use it.

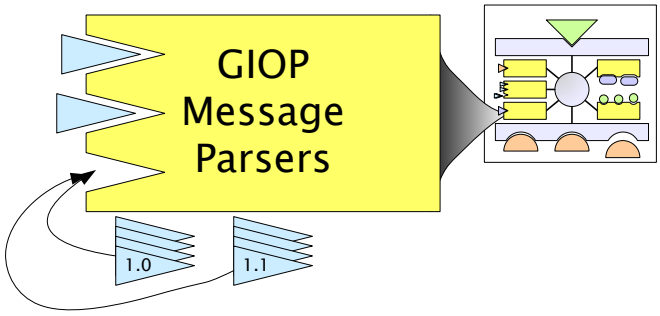

Figure 3: Pluggable GIOP Readers and Writers

In the GIOP messaging protocol, the format for each message type is different. In addition, there are several versions of each type of message, and the format and semantics can vary with the version. It is therefore beneficial to code each version-specific message reader and writer as a separate class so an instance can be plugged in based on the appropriate GIOP message version.

GIOP messages are always converted into CDR format before being sent over a transport. The CDR stream is an octet stream that isolates the application from variations in data formats in a heterogeneous network. GIOP messages are marshaled into a CDR input stream and demarshaled from a CDR output stream.

Each message type of a particular version has one method to marshal the message to a CDR Stream and another method to demarshal the message from a CDR Stream. The marshal and demarshal methods denote separate behaviors and hence the strategized versions can be further strategized into reader objects and writer objects. After strategizing the versions and the marshaler/demarshaler type, we have 32 object classes that offer a unique functionality based on their version, message type and marshaler/demarshaler type. If any one of the reader/writer classes are loaded into memory they offer the smallest footprint, since only the functionality that is needed at that instant by a message is loaded into memory.

3.4 Configuring GIOP Messaging Protocols in ZEN

We have identified four different design strategies for handling GIOP messaging protocols: *PluggableFineGrain*, *PluggableClientServer*, *PreloadedHandler*, and *NonPluggable*. The first three strategies apply the Virtual Component pattern to various degrees and the last is similar to a conventional monolithic design except that the classes implementing the messaging protocols are still decomposed into small units based on message type and version. We have implemented all four designs in ZEN and measured their performance for various use-case scenarios. Each design offers different behavior characteristics, and each is best suited for particular types of applications that are described below.

ZEN includes a framework that allows pluggins for any of the four different strategies, as shown in Figure 4. Our initial intent was to compare them for operation throughput, memory footprint, and real-time predictability to determine which was the best in general. Over time, however, we discovered that each had characteristics which made it applicable for a particular class of applications. Each of the following sections present an overview of one design strategy and the applicability of that strategy.

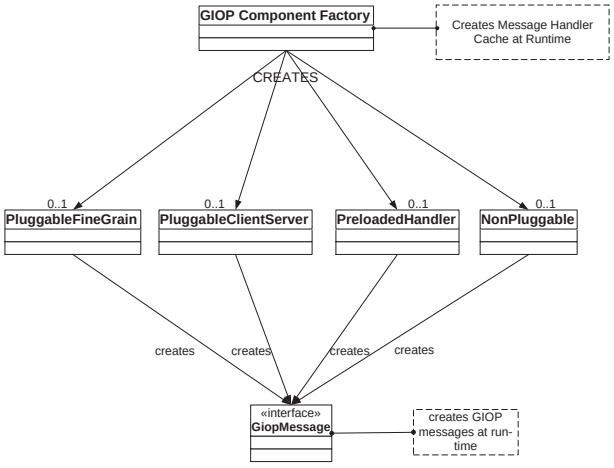

Figure 4: GIOP Component Factory

3.4.1 PluggableFineGrain Design

Strategy overview. PluggableFineGrain is the first alternative design of ZEN's configurable messaging framework. As shown in Figure 5, this design loads only one of either the marshaling or demarshaling method based on the GIOP version and message type.

Strategy design. A message factory is used to create and load marshaler/demarshaler classes dynamically. The message factory offers better configurability and extensibility since any change in the message type, version, or the marshaler/demarshaler type only requires a change in the message factory. In this strategy, only the message factory must be recompiled and re-linked after a new version is added or the name of a class file changes. No other classes need to be modified or recompiled and re-linked.

When a specific message reader/writer is needed, the class name is generated at run-time by concatenating the message name with the marshaler/demarshaler type and the message version, *e.g.*, `ReplyMessageReader10`. The decision to choose a marshaler or a demarshaler depends on whether the message must be written or read. The class matching that name is loaded into memory using a factory, then an instance is created.

We save a reference to the class in a cache (implemented by a hash table) to improve the average-case performance and predictability for subsequent class references.[2] If a class is referenced again, the class stored in the cache is used. To avoid re-generating the class names, we generate a unique key from the message type, version, and marshaler/demarshaler type to be used as a cache key index.

For example, the `ReplyMessageWriter10` class would be loaded to write a Reply message in GIOP version 1.0. In this design, any loaded class is cached for faster accessed if the same marshaler/demarshaler is used again. The messages are read/written into the CDR Stream by the marshal/demarshal methods. These methods are defined as separate classes based on the message type, marshaler/demarshaler type, and the GIOP version type.

The PluggableFineGrain design can be extended easily to handle new versions of GIOP. When a new version is added to GIOP, each message type can be added as new derived class from the appropriate base message type, but the name will be different to reflect the new version number. The marshaler and demarshaler classes can be created by concatenating the name of the message type, the marshaler/demarshaler type and the version number (which is contained in each message). After the classes are added with the defined syntax, the new classes have added to the messaging module and have to be compiled and linked. The rest of the ORB need not be modified or even recompiled.

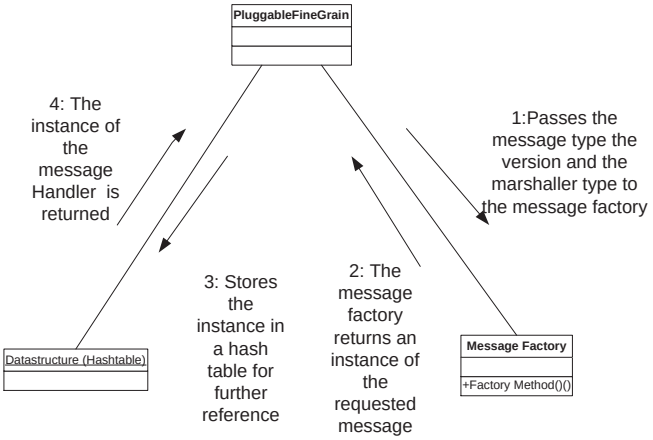

Figure 5: PluggableFineGrain Design Strategy

Strategy applicability. The PluggableFineGrain design strategy is well-suited for applications that invoke one-way operation calls. If users know their applications will require many one-way calls, then only the minimal functionality that is required by the application is loaded into memory. For example, if a client has subscribed to a notification service, the notification service sends the client notifications in the form of GIOP Reply messages. To read the messages, the client must load a reader of a Reply message type having a particular version number. Hence, only the functionality that is needed is

[2] We know beforehand the total number of message types and versions, so we can ensure the table size is sufficiently large to keep each cache look-up predictable.

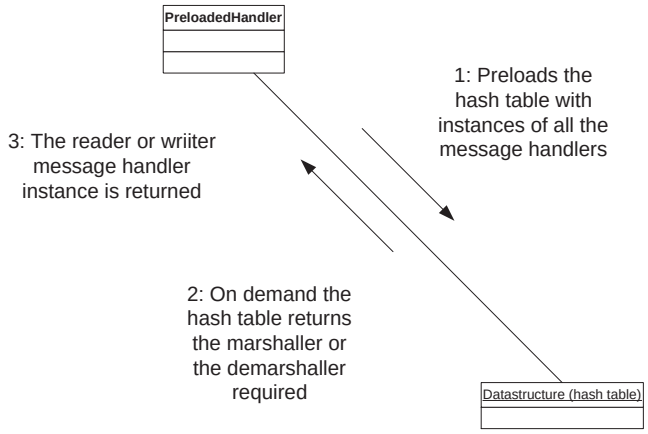

Figure 6: PreloadedHandler Design Strategy

loaded into memory. This leads to a small footprint because the writer is not loaded into memory.

This PluggableFineGrain design enables the use of CORBA for embedded applications. Monolithic ORBs tend to have a large memory footprint, which makes them unsuitable for memory-constrained embedded applications. The messaging framework in most monolithic ORBS have both the marshal and demarshal methods coded together for a message type. Multiple versions are handled by having multiple marshal and demarshal methods that are coded together in the same class dispatched by a `switch` or `if` statement. Unfortunately, this design is unsuitable for many embedded applications, because all the messaging functionality is tightly-coupled, and therefore, all is present in memory, even when not needed.

Consider an CORBA application where a server sends one-way requests of GIOP version 1.0 to a client. Using the PluggableFineGrain strategy, this server just needs to load a marshaler of Reply message of version 1.0 (`ReplyMessageWriter10`). Conversely, if the client only receives GIOP messages of version 1.1, then the demarshal Request message type of version 1.1 (`ReplyMessageReader11`) will be loaded into memory.

3.4.2 PreloadedHandler Design

Strategy overview. PreloadedHandler is the second alternative design of a configurable messaging framework shown in Figure 6. The PreloadedHandler loads instances of all the marshalers and demarshalers and stores them in the cache at initialization time, which is why we call it "pre-loading." The cache has an instance for a marshaler/demarshaler for every message type and every version. This design is different from the PluggableFineGrain design since all the instances are preloaded in the cache, whereas in the PluggableFineGrain model an instance of a reader or a writer had to be loaded and entered into the cache on-demand at run-time.

Strategy design Each message type, demarshaler, and version combination is used to construct a unique key, which is hard-coded in a `switch` statement. The key can be used as a reference to access the reader or the writer much faster than by hash table lookup. The pre-loaded instances and the unique key generation in a `switch` statement ensures that a minimum amount of time is spent to create the key and access the instance for the marshaler/demarshaler class in the cache. Each time an instance of a reader or a writer is required, it is returned from the cache.

In the PluggableFineGrain implementation, the class instance based on the message type, version, and marshaler/demarshaler type is created along with the unique key and is stored in the cache. In the PreloadedHandler method, the cumulative time to create an instance of the class, a key and the time to load it into the cache is not needed. To add a new version of GIOP in the PreloadedHandler design, concrete classes implementing the new version must be derived and implemented. Then instances of each of these classes must be created and load into the cache at initialization time.

Strategy applicability. The PreloadedHandler design is a beneficial for applications that demand predictability. Instances of all the readers and writers are stored in the cache before the first use, so there is no extra latency on first access. In addition, this design strategy conserves memory because at most one class instance is required.

The PreloadedHandler design is more efficient than the PluggableFineGrain design since instances are pre-loaded and stored in the cache. This design is useful for real-time and embedded applications that require both predictability and small footprint.

With the PreloadedHandler design, if a client makes a one-way Request method call, it directly accesses `RequestMessageWriter10` which is pre-loaded in the cache. As the performance results show in Section /refoperationThroughput, however, the Request message is loaded faster than when loaded in the PluggableFineGrain. The only required marshaler/demarshaler class on the server is `RequestMessageReader10`.

The PreloadedHandler design provides the user to choose between an extensible small footprint model (PluggableFineGrain) and model with a similar footprint and a characteristic of being faster and predictable (PreloadedHandler).

3.4.3 Client/Server-Pairing Design

Strategy overview. PluggableClientServer is a third alternative design of a configurable messaging framework (shown in

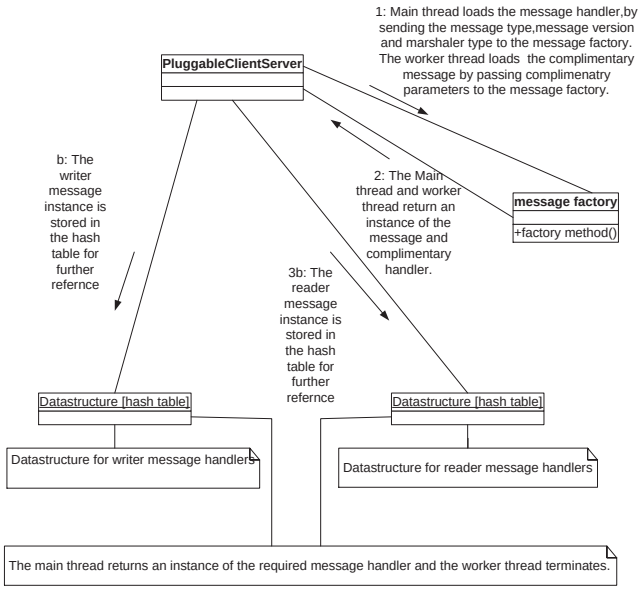

Figure 7: PluggableClientServer Design Strategy

Figure 7). In this design, complementary methods are grouped together based on likely usage patterns. For example, Request messages and Reply messages are complimentary to each other. If a Request message is sent by a client, then most probably a Reply message will be expected back and the Reply reader will be needed. With this design, the Request writer and the Reply reader are loaded together at the same time.

The advantage of this design strategy is that the complementary method needed later is already loaded. The client/server-pairing design is advantageous for implementations that handle request-reply two-way invocation operation calls. The PluggableClientServer design groups the marshaling and demarshaling methods that will be used together in a two-way call. When the message has to be read or written, it sends a request for the corresponding marshaler or demarshaler. If the marshaler (writer) is not present in the writer cache or the demarshaler (reader) is not present in the reader cache, then the marshaler/demarshaler is loaded into the corresponding cache.

Strategy design. When a Request message is to be sent by a client, the main thread loads the class to marshal Request messages of the appropriate GIOP version. At the same time, a worker thread work loads the complementary class to demarshal Reply messages of the matching version because a reply is expected for two-way calls. After the worker thread has finished loading the complementary class, it enters it into the reader cache and terminates. When the actual Reply message arrives, the Reply message demarshaler is already in the reader cache. The Reply message can then be demarshaled without waiting for the class to be loaded. Similar behavior occurs when a server receives a Request: both the `RequestMessageReader` and `ReplyMessageWriter` are loaded in parallel.

Servers typically read Requests and write Replies, so we group together the classes `RequestMessageReader` and `ReplyMessageWriter` according to version. Similarly clients typically write Request messages and read Reply messages, so we group the two classes `RequestMessageWriter` and `ReplyMessageReader` of the same version. Thus, a pure client using one GIOP version need only load the classes containing both client-oriented methods.

As shown in Figure 10, this method is faster than the PluggableFineGrain implementation. In nearly the same amount of time required to load one of the message readers or writers in a PluggableFineGrain design, both a reader and a writer can be loaded in a PluggableClientServer design. Although at first this seemed like a better implementation than the PluggableFineGrain implementation, it fails in applications with limited resources. This design is therefore unsuitable for applications that are single threaded or cannot afford to dedicate a thread to the messaging system due to limited resources.

A new version can be added by creating classes of the readers and the writers based on the message type, marshaler/demarshaler type and version. The classes have to be added to the messaging module and can be used without changes made to any other part of the existing messaging module code. This design is extensible and pluggable.

Strategy applicability. The PluggableClientServer design is beneficial for two-way operation calls where a Request message is sent and it is known that a Reply message will be arrive soon. This design offers faster execution of reading/writing a message. It can also reduce footprint since only the functionality that is needed is loaded into memory. The message reader/writer pair loaded by the parallel thread offers eager loading of a functionality that will be needed by the process. Hence, the eager loading leads to the faster reading and writing of messages into the CDR Stream.

To illustrate this design, we give another example. A client sends a two-way Request message to a server and therefore expects to receive a Reply message. The main thread of the client loads the Request marshal message of type 1.0 in the writer cache. Simultaneously, a pre-loaded parallel worker thread loads into the reader table the anticipated method, which, in our example, is demarshaler of Reply of version 1.0. When the Reply message is received from the server, the anticipated method for reading this Reply is already loaded into the cache. Normally, the client would be blocked waiting for the reply, then it could load the required method, so the loading is hap-

pening in parallel with the remote execution of the method thus no time wasted to load the class when the Reply message is received.

3.4.4 Nonpluggable Design

Strategy overview. NonPluggable is the fourth design alternative for the ZEN GIOP messaging framework. In this design, the instances of readers and writers are hard coded at the point of reading and writing the message. It is therefore similar to the conventional monolithic design, so we added it to compare the performance of this strategy against the three more modular strategies.

Strategy design. Unlike the three earlier GIOP design strategies, this strategy is not pluggable. Instead, the code for handling each type and version of message is tightly-coupled. However, it is faster than the other designs because no time is spent loading classes, building unique keys, or in performing cache table look-ups. As shown in Figure 10, the NonPluggable design is the fastest implementation of all the designs we presented in the previous sections. The Nonpluggable design is beneficial for users who need high throughput. This design is not suitable for applications that demand predictability, however, because the readers and writers have instances hard-coded at the point of use, thereby preventing the ORB from storing and reusing an instance of a message reader/writer. Each time a message requiring the same message reader is encountered a new instance of the reader is created.

This design is not exactly the same as a monolithic design, since it still uses the separated readers/writers defined in the earlier strategies. This decoupling can allow JVMs to selectively load and unload classes as needed, which is not the case in the usual monolithic designs. However, the NonPluggable strategy gives the user the choice to use the monolithic design for faster applications with a tradeoff of predictability and extensibility. In our paper we have used the NonPluggable design to compare the operational throughput results of the other designs with the NonPluggable design.

Strategy applicability. The NonPluggable design strategy provides high throughput and provides a small footprint. Since this design strategy is not easily extensible, however, adding new GIOP versions requires modifying each message-type class to detect and handle the new version. Moreover, this strategy is not dynamically adaptable.

3.5 Evaluating ZEN's GIOP Protocol Framework

Below, we compare and contrast the GIOP messaging design strategies to illustrate the tradeoffs between them.

3.5.1 Ease of Messaging Extension via Pluggability

Modifying ZEN's messaging framework to support new versions of GIOP is straightforward, provided one of the pluggable designs is used (PluggableClientServer or PluggableFineGrain). If PreloadedHandlers are used, instances of the new classes that have been created must be added to the cache along with unique keys. The keys can be defined for them and added in the `switch` statement.

For the Pluggable designs (PluggableFineGrain and PluggableClientServer), new classes can be added based on the message type, the version and the marshaler type. The classes must be created in the particular syntax for the message factory to return the right instance of the reader or the writer. Besides taking care of creating new classes with syntax defined names, none of the other files need to be changed, recompiled, or relinked. The ease of extension in these designs allows changes to be made easily while ensuring that a small footprint is maintained. As the messaging module grows the design of the pluggable and PreloadedHandler models ensures a clean code and backward compatibility with previous message versions.

3.5.2 Trading Off Messaging Flexibility for Predictability

CORBA applications using a flexible messaging framework that use any of the pluggable designs, need not know beforehand what type of messages they will send and/or receive. For example, they could be server-like or client-like or both and the middleware messaging framework will adapt to suit their needs. Instead, only the necessary methods are loaded into memory on demand depending on whether the program is a client, a server, or a peer. If the behavior of a particular program is known beforehand the necessary classes may be preloaded at initialization time to eliminate any delays from lazy class loading.

ZEN's messaging framework allows a minimal CORBA subset to be installed in the server or client ORB. Hence, enabling the design to be suitable for embedded applications. Since the behavior is predefined on the client or server ORB based on the application, the whole system is predictable which is essential for real-time applications.

4 Empirical Results

This section compares the throughput and memory footprint of the various GIOP design strategies.

4.1 Footprint Measurements

Overview The footprint measurements presented here were obtained from a client-server program that transmits the "null" operation, *i.e.*, an operation with no parameters. The machine on which the measurements have been taken is a Pentium III dual-CPU 930.976 MHZ machine with a cache size of 256 kb and 513 Mb of RAM. The machine has a swap memory of 996 Mb. The experiments were conducted using JVM version 1.2.2 running on Linux OS 2.4.16. We used ZEN version Alpha release 0.8. These test were run in loopback mode to isolate the performance of the ORB, rather than the network drivers.

Results and analysis. Table 1 presents the footprint measurements. These measurements show that the memory footprint of each of our four alternative GIOP designs has a much smaller footprint than the monolithic design used in JacORB. Note also that JacORB only implements one version of GIOP, whereas ZEN implements all three of the current specified versions.

Memory / Virtual components	Data size (Pure Client) [kilobytes]	Data size (pure server) [kilobytes]	Code size + data size + stack size (pure Client) [kilobytes]	Code size + data size + stack size (pure Server) [kilobytes]
Pluggable Fine Grain (message handler cache)	51.688	30.632	53540	53786
Pluggable Client Server (Message handler cache)	49.560	30.664	53543	53833
PreLoaded MessageHandler (Message Handler cache)	48.600	30.632	53554	53950
Nonpluggable (Message Handler cache)	47.800	30.632	53555	53798
Jacorb (Vesrion 1.3.3)	119.928	42.600	56312	61508

Table 1: Memory Footprint Measurements

The NonPluggable strategy, has the smallest footprint since we use the highly-decomposed classes for readers/writers based on the version number identified for the pluggable strategies. The JVM can therefore selectively loads classes when the first instance is created. This design strategy also does not have the data space overhead of maintaining the caches. The NonPluggable version does not use any caches, since the instances are hard-coded at the point of demand. The other designs do consume more data space, however, since they all use hash tables for the caches.

Figures 8 and 9 also show that the data size of the server is small for most of the strategized versions, except for the PluggableClientServer strategy. This is due to the extra storage

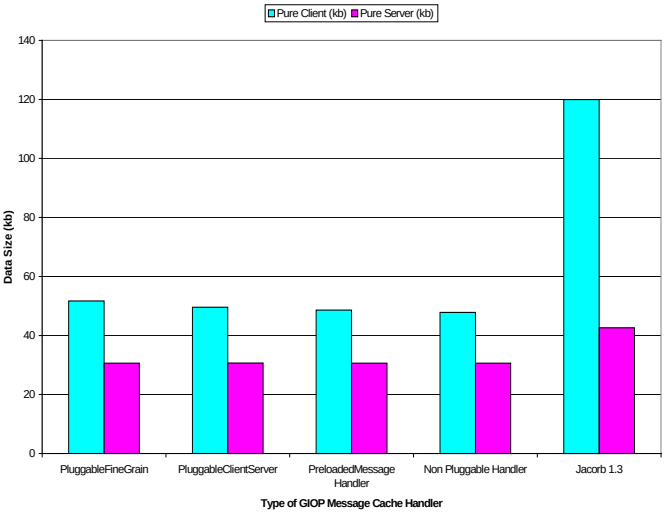

Figure 8: Data Size Comparisons from Table 1

required for the spawned worker thread to execute in parallel with the main thread. The PreloadedHandler design strategy includes initializing the hash table cache and creating instances of all the possible message types at initialization time. As a result, the stack utilization of this design is larger.

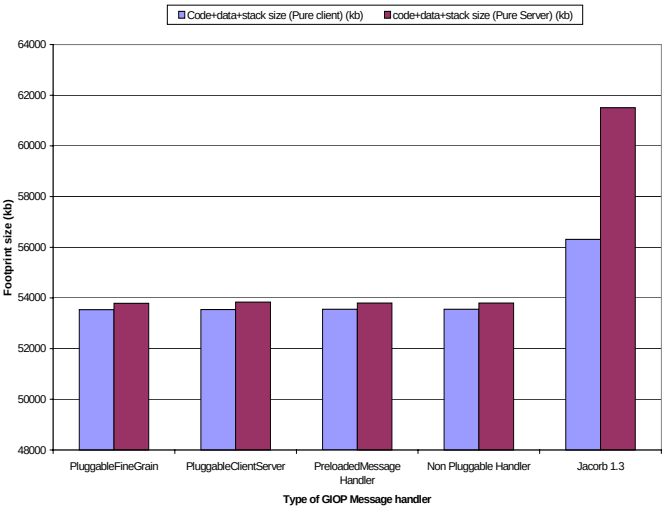

Figure 9: Footprint Comparisons from Table 1

4.2 Operation Throughput Measurements

Overview. Operational throughput is compared for the different GIOP design strategies by making "null" invocations

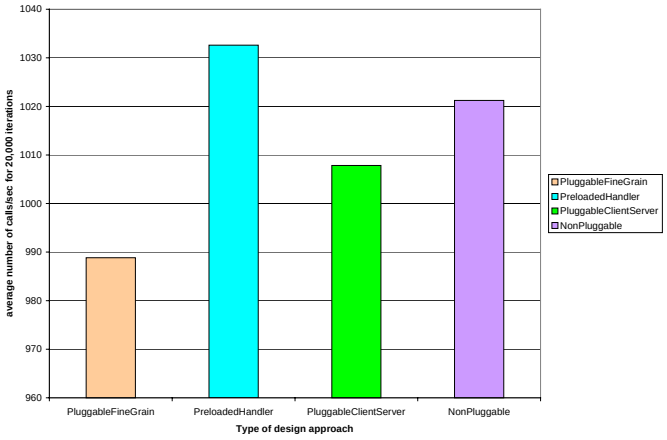

Figure 10: Operation Throughput for All GIOP Design Strategies

for a specific number of iterations. The number of iterations were increased gradually to calculate the average operational throughput for each design strategy. However, we only show the measurements for 20,000 two-way loopback operation calls using the same configuration described in Section 4.1.

Results and Analysis. Figure 10 shows the operation throughput measurements taken for each of the designs presented above. As shown in Figure 10, the PreloadedHandler design has the highest operational throughput of all the GIOP design strategies. However, this design has the largest memory footprint.

As we can see from the Figure 10, the PluggableClientServer design yields a faster implementation than does the PluggableFineGrain design. This can be explained as in the PluggableFineGrain design for making two-way invocations, the main thread is blocked waiting for the Reply to return. When the reply returns the appropriate Reply reader has to be loaded. Whereas in the PluggableClientServer implementation the worker thread loads the appropriate reader to read the Reply message that the main thread is blocked waiting for the reply. Hence, the latency to load the message handler is saved in the PluggableClientServer design strategy.

4.3 Interpretation of the Empirical Results

Our measurements show that each of the highly-modular GIOP messaging design strategies have a much smaller footprint than the monolithic design because only the needed classes are loaded into memory. Each of the design strategies has advantages for certain applications depending on the need to optimize memory footprint, extensibility, or operational throughput.

For example, the Nonpluggable and PreloadedHandler designs excelled at operational throughput, but neither one extends easily to handle new message types and new message versions. The PluggableClientServer and PluggableFineGrain designs performed nearly as well as the other two, but both are fully extensible. Moreover, PluggableClientServer is best suited for multi-threaded applications since it requires a worker thread to load the complementary class in parallel. PluggableFineGrain is best suited for single-threaded applications since it does not need any extra threads.

Providing a flexible messaging framework allows selection of the specific implementation based on the application programmers needs. We intend to explore the use of reflection [13] to automatically select the appropriate messaging protocol based on dynamic feedback in the near future.

5 Related Work

Conventional DOC middleware has historically been too slow, too unpredictable, and too large to meet the requirements of many types of DRE applications. The ZEN project is aimed at alleviating these problems. The design of ZEN's pluggable GIOP messaging framework is influenced by prior research on the design and optimization of protocol frameworks for communications. This section outlines this research and compares it with our work on ZEN.

Configurable communication frameworks: The *x*-kernel [14], Conduit+ [15], System V STREAMS [16], ADAPTIVE [17], and F-CSS [18] are all configurable communication frameworks that provide a protocol backplane consisting of standard, reusable services that support network protocol development and experimentation. These frameworks support flexible composition of modular protocol processing components, such as connection-oriented and connectionless message delivery and routing, based on uniform interfaces. TAO's pluggable protocols framework focuses on implementing and/or adapting to transport protocols beneath a higher-level middleware API, *i.e.*, the standard CORBA programming API.

The frameworks for communication subsystems listed above focus on implementing various protocol layers beneath relatively low-level programming APIs, such as the Socket API. In contrast, ZEN's pluggable GIOP messaging framework focuses on selecting a subset of the many standard CORBA GIOP message readers/writers. Therefore, existing communication frameworks can provide building blocks for ZEN's pluggable GIOP messaging framework.

Patterns-based communication frameworks: An increasing number of communication frameworks are being designed and documented using patterns [19, 15]. In particular, Conduit+ [15] is an OO framework for configuring network pro-

tocol software to support ATM signaling. Key portions of the Conduit+ protocol framework, *e.g.*, demultiplexing, connection management, and message buffering, were designed using patterns like Strategy, Visitor, and Composite [12]. Likewise, the concurrency, connection management, and demultiplexing components in ZEN's ORB Core and Object Adapter also have been explicitly designed using patterns like Virtual Component [11], Acceptor-Connector, and Active Object [19].

CORBA pluggable protocol frameworks: The architecture of ZEN's pluggable GIOP messaging framework is based on ideas learned from the pluggable protocol frameworks used in TAO [8]. TAO's GIOP messaging is implemented in the monolithic design and therefore each message type marshaler/demarshaler consumes memory footprint whether or not they are needed and used by an application. By extensive application of the Virtual Component pattern, we improved the design of GIOP messaging in ZEN to allow flexible adaptation to any one of four alternative implementations. Each alternative implementation, in turn, is dynamically and flexibly configurable so to include only the minimal amount of code necessary to provide the necessary GIOP messaging functionality required by each application.

6 Concluding Remarks

This paper describes the design of ZEN's GIOP messaging framework, which allows selection of one of four different design implementations. We measured and compared them for footprint requirements and operational throughput. Our results indicate that a high-degree of decomposition yields the best footprint gains, as shown by the fact that all four have a smaller footprint than JacORB, which does not even implement all the versions of GIOP.

We learned from our experience with TAO that small memory footprints must be designed for during the initial design phase. We also learned that configuration should be automated as much as possible, to avoid placing an onerous burden on application developers. Our future work will focus on using reflection [13] to provide feedback to the middleware to allow automatic static optimization of the middleware to customize ZEN for each application developer's needs.

References

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

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

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

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

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

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

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

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

[9] C. O. Raymond Klefstad, Douglas C. Schmidt, "Towards Highly Configurable Real-time Object Request Brokers," in *Proceedings of the International Symposium on Object-Oriented Real-time Distributed Computing (ISORC)*, (Newport Beach, CA), IEEE/IFIP, Mar. 2002.

[10] 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.

[11] 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 9^{th} Annual Conference on the Pattern Languages of Programs*, (Monticello, Illinois), Sept. 2002.

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

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

[14] N. C. Hutchinson and L. L. Peterson, "The *x*-kernel: An Architecture for Implementing Network Protocols," *IEEE Transactions on Software Engineering*, vol. 17, pp. 64–76, January 1991.

[15] H. Hueni, R. Johnson, and R. Engel, "A Framework for Network Protocol Software," in *Proceedings of OOPSLA '95*, (Austin, Texas), ACM, October 1995.

[16] D. Ritchie, "A Stream Input–Output System," *AT&T Bell Labs Technical Journal*, vol. 63, pp. 311–324, Oct. 1984.

[17] D. C. Schmidt, D. F. Box, and T. Suda, "ADAPTIVE: A Dynamically Assembled Protocol Transformation, Integration, and eValuation Environment," *Journal of Concurrency: Practice and Experience*, vol. 5, pp. 269–286, June 1993.

[18] M. Zitterbart, B. Stiller, and A. Tantawy, "A Model for High-Performance Communication Subsystems," *IEEE Journal on Selected Areas in Communication*, vol. 11, pp. 507–519, May 1993.

[19] D. C. Schmidt and C. Cleeland, "Applying Patterns to Develop Extensible ORB Middleware," *IEEE Communications Magazine*, vol. 37, April 1999.