Applying Patterns to Design a High-performance, Real-time Pluggable Protocols Framework for OO Communication Middleware

Carlos O'Ryan, Fred Kuhns, Douglas C. Schmidt, and Jeff Parsons
{coryan,fredk,schmidt,parsons}@cs.wustl.edu
Department of Computer Science, Washington University
St. Louis, MO 63130, USA *

Abstract

To be an effective platform for performance-sensitive real-time and embedded applications, off-the-shelf OO middleware like CORBA, DCOM, and Java RMI must preserve communication-layer quality of service (QoS) properties to applications end-to-end. However, conventional OO middleware interoperability protocols, such as CORBA's GIOP/IIOP or DCOM's MS-RPC, are not well suited for applications that cannot tolerate the message footprint size, latency, and jitter associated with general-purpose messaging and transport protocols. It is essential, therefore, to develop standard pluggable protocols frameworks that allow custom messaging and transport protocols to be configured flexibly and used transparently by applications.

This paper provides three contributions to research on pluggable protocols frameworks for performance-sensitive OO communication middleware. First, we outline the key design challenges faced by pluggable protocols developers. Second, we describe how TAO, our high-performance, real-time CORBA-compliant ORB, addresses these challenges by applying design patterns in its pluggable protocols framework. Third, we present the results of benchmarks that pinpoint the impact of TAO's pattern-oriented OO design on it end-to-end efficiency, predictability, and scalability.

Our results demonstrate how applying optimizations and patterns to OO communication middleware can yield highly flexible/reusable designs and highly efficient/predictable implementations. In particular, TAO's middleware overhead is only ∼110 μsecs using a commercial, off-the-self 200 Mhz embedded system CPU, interconnect, and OS. These results illustrate that (1) OO communication middleware performance is largely an implementation detail and (2) the next-generation of optimized, standards-based middleware can replace ad hoc and proprietary solutions.

Subject areas: Frameworks; Design Patterns; Distributed and Real-Time Systems

*This work was supported in part by Boeing, DARPA contract 9701516, GDIS, NSF grant NCR-9628218, Nortel, Siemens, and Sprint.

1 Introduction

Current trends and limitations: Three trends are shaping the future of communication application and system development. First, there is a movement away from *programming* applications from scratch, using low-level protocols and operating system APIs, to *integrating* applications, using reusable components [1], such as ActiveX, Java Beans, and CORBA components. Second, there is great demand for *OO communication middleware* that provides remote method invocation and/or message-oriented middleware to simplify application component collaboration [2]. Third, there are increasing efforts to define *standard* OO communication middleware, such as CORBA [3], that permits applications to interwork seamlessly throughout *heterogeneous* networks and endsystems.

Standard OO communication middleware is now available off-the-shelf that allows clients to invoke operations on distributed components without concern for component location, programming language, OS platform, communication protocols and interconnects, or hardware [4]. However, off-the-shelf OO communication middleware generally lacks (1) support for QoS specification and enforcement, (2) integration with high-speed networking technology, and (3) performance, predictability, and scalability optimizations [5]. These omissions have limited the rate at which performance-sensitive applications, such as video-on-demand, teleconferencing, and avionics mission computing, have been developed to leverage advances in OO communication middleware.

Overcoming OO communication middleware limitations with pluggable protocols: To address the shortcomings of OO communication middleware described above, we have developed *The ACE ORB* (TAO) [5]. TAO is open-source,[1] standards-based, high-performance, real-time ORB endsystem OO communication middleware that supports applications with deterministic and statistical QoS requirements, as well as "best-effort" requirements. TAO is the first ORB to support end-to-end QoS guarantees over ATM/IP networks [6, 7].

[1] TAO is available at `www.cs.wustl.edu/~schmidt/TAO.html`.

We have used TAO to research many dimensions of high-performance and real-time ORB endsystems, including static [5] and dynamic [8] scheduling, request demultiplexing [9], event processing [10], ORB Core connection and concurrency architectures [11], IDL compiler stub/skeleton optimizations [12], systematic benchmarking of multiple ORBs [13], I/O subsystem integration [7], and patterns for ORB extensibility [14]. This paper focuses on a previously unexamined dimension in the high-performance and real-time ORB endsystem design space: *the patterns used to develop a pluggable protocols framework* that supports high-speed protocols and networks, real-time embedded system interconnects, and standard TCP/IP protocols over the Internet.

At the heart of TAO's pluggable protocols framework is its patterns-oriented OO design [15], which decouples TAO's ORB messaging and transport interfaces from its transport-specific protocol components. This design allows custom ORB messaging and transport protocols to be configured flexibly and used transparently by CORBA applications. For example, if ORBs communicate over a high-speed networking protocol like ATM AAL5, then simpler, optimized ORB messaging and transport protocols can be configured to eliminate unnecessary features and overhead of the standard CORBA General Inter-ORB Protocol (GIOP) and Internet Inter-ORB Protocol (IIOP). Likewise, TAO's pluggable protocols framework makes it straightforward to support customized embedded system interconnects, such as CompactPCI or VME, under standard CORBA inter-ORB protocols like GIOP.

For OO researchers and practitioners, the results in this paper are important, since they demonstrate concretely that the ability of standards-based OO communication middleware to support high-performance, real-time systems is largely an implementation issue, rather than an inherent liability. For instance, TAO's end-to-end latency overhead is only $\sim$110 μsecs using commercial off-the-self 200 Mhz PowerPCs, a 320 Mbps VMEbus, and VxWorks, which is as fast, or faster, than many *ad hoc*, proprietary solutions. These results motivate the use of well-tuned, standards-based OO communication middleware, even for real-time embedded applications with very stringent QoS requirements. The paper also explores how patterns were applied to resolve key design challenges. Our pattern-oriented OO design can be extended to other pluggable protocol frameworks, either in standard middleware or in distributed applications using proprietary middleware.

Paper organization: The remainder of this paper is organized as follows: Section 2 outlines the CORBA protocol interoperability architecture; Section 3 motivates the need for a CORBA pluggable protocols framework; Section 4 describes TAO's pluggable protocols framework, focusing on the patterns that guide its software architecture and resolve key design challenges; Section 5 illustrates the performance characteristics of TAO's pluggable protocols framework; Section 6 compares TAO with related work; and Section 7 presents concluding remarks.

2 Overview of the CORBA Protocol Interoperability Architecture

The CORBA specification [3] defines an architecture for ORB interoperability. Although a complete description of the model is beyond the scope of this paper, this section outlines the parts that are relevant to our present topic, *i.e.*, object addressing and inter-ORB protocols.

Object addressing synopsis: To identify objects, CORBA defines a generic format called the Interoperable Object Reference (IOR). An object reference identifies one object and associates one or more paths through which that object can be accessed. Each path references one server location that implements the object and an opaque identifier valid on that particular server.

Different object references may represent the same object, *e.g.*, if a server is re-started on a new port or migrated to another host. Likewise, if a server has multiple network interfaces connecting it to distinct networks, there may be multiple network addresses. Thus, multiple server locations can be referenced by one IOR.

References to server locations are called *profiles*, which provide an opaque, protocol-specific representation of an object location. Profiles can be used to annotate the server location with QoS information, such as the priority of the thread serving each endpoint or alternative addresses for fault-tolerance.

Protocol model synopsis: CORBA Inter-ORB Protocols (IOP)s define interoperability between ORB endsystems. IOPs provide data representation formats and ORB messaging protocol specifications that can be mapped onto standard and/or customized transport protocols. Regardless of the choice of ORB messaging or transport protocol, however, the standard CORBA programming model is exposed to the application developers. Figure 1 shows the relationships between these various components and layers.

In the CORBA protocol interoperability architecture, the standard General Inter-ORB Protocol (GIOP) is defined by the CORBA specification [3]. In addition, CORBA defines a transport-specific mapping of GIOP onto the TCP/IP protocol suite called the Internet Inter-ORB Protocol (IIOP). ORBs must support IIOP to be "interoperability compliant." Other mappings of GIOP onto different transport protocols are allowed by the specification, as are different inter-ORB protocols, known as Environment Specific Inter-ORB Protocols (ESIOP)s.

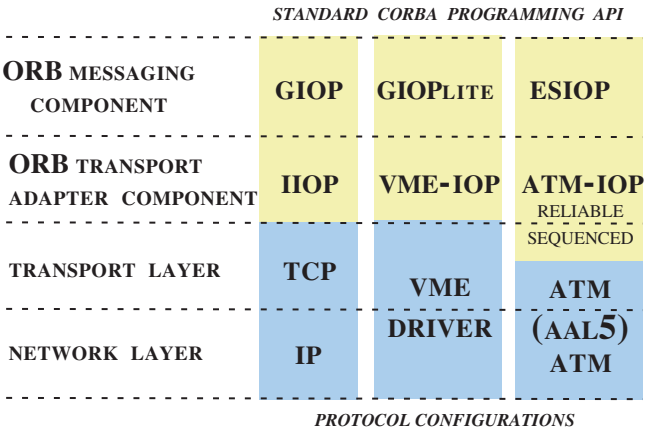

Figure 1: Relationship Between Inter-ORB Protocols and Transport-specific Mappings

Regardless of whether GIOP or an ESIOP is used, a CORBA IOP must define a data representation, an ORB message format, an ORB transport protocol or transport protocol adapter, and an object addressing format. Below, we outline how GIOP addresses each of these IOP elements.

GIOP synopsis: The GIOP specification consists of the following elements:

• **A Common Data Representation (CDR) definition:** CDR is a transfer syntax that maps IDL types from their native host format into a low-level *bi-canonical* representation, which supports both little-endian and big-endian formats. CDR-encoded messages are used to transmit CORBA requests and server responses across a network. All IDL data types are marshaled using the CDR syntax into an *encapsulation*, which is an octet stream that holds marshaled data.

• **GIOP message formats:** The GIOP specification defines seven types of messages that send requests, receive replies, locate objects, and manage communication channels. The following table lists the seven types of messages in GIOP 1.0[2] and the permissible originators of each type:

Message Type	Originator	Value
Request	Client	0
Reply	Server	1
CancelRequest	Client	2
LocateRequest	Client	3
LocateReply	Server	4
CloseConnection	Server	5
MessageError	Both	6

[2]Version 1.1 of GIOP added a `Fragment` message and version 1.2 relaxes the restrictions with respect to message originators.

• **GIOP transport assumptions:** The GIOP specification describes the features of an ORB transport protocol that can carry GIOP messages, and requires such a protocol to be a reliable, connection-oriented byte-stream. In addition, GIOP defines a connection management protocol and a set of constraints for GIOP message ordering.

• **Object addressing:** An Interoperable Object Reference (IOR) is a sequence of opaque *profiles*, each representing a protocol-specific representation of an object's location. For example, an IIOP profile includes the IP address and port number where the server accepts connections, as well as the object key that identifies an object within a particular server. Since there may be multiple paths to an object, the same IOR can contain multiple IIOP profiles, along with profiles for other protocols, such as GIOP over ATM or non-GIOP protocols.

ESIOP synopsis: In addition to the standard GIOP and IIOP protocols, the CORBA specification allows ORB implementors to define Environment Specific Inter-ORB Protocols (ESIOP)s. ESIOPs can define unique data representation formats, ORB messaging protocols, ORB transport protocols or transport protocol adapters, and object addressing formats. These protocols can exploit the QoS features and guarantees provided in certain domains, such as telecommunications or avionics, to satisfy performance-sensitive applications that have stringent bandwidth, latency, and jitter requirements.

Only one ESIOP protocol is defined in the CORBA 2.x family of specifications: the DCE Common Inter-ORB Protocol (DCE-CIOP) [3]. Two ESIOPs we are developing, GIOPlite and an ATM ESIOP, are shown in Figure 1. The OMG is considering other protocols for domains particular domains, such as wireless and mobile systems [16], that have unique performance characteristics and optimization points.

3 Motivation for a CORBA Pluggable Protocols Framework

The CORBA specification provides a standard for general-purpose OO communication middleware. Within the scope of the specification, ORB developers are free to optimize internal data structures and algorithms [9]. Moreover, ORBs may use specialized inter-ORB protocols and ORB services and still comply with the specification.[3] This section identifies the limitations of current ORBs with respect to their protocol support and enumerates the key requirements for a pluggable protocols framework to overcome these limitations.

[3]An ORB *must* implement GIOP/IIOP, however, to be interoperability-compliant.

3.1 Protocol Limitations of Conventional ORBs

CORBA's standard GIOP/IIOP protocols are well suited for conventional request/response applications with best-effort QoS requirements [12]. They are not well suited, however, for high-performance, real-time, and/or embedded applications that cannot tolerate the message footprint size of GIOP or the latency, overhead, and jitter of the TCP/IP-based IIOP transport protocol. For instance, TCP functionality, such as adaptive retransmissions, deferred transmissions, and delayed acknowledgments, can cause excessive overhead and latency for real-time applications [17]. Likewise, networking protocols, such as IPv4, lack the functionality of packet admission policies and rate control, which can lead to excessive congestion and missed deadlines in networks and endsystems.

Therefore, applications with more stringent QoS requirements need optimized protocol implementations, QoS-aware interfaces, custom presentations layers, specialized memory management (*e.g.*, shared memory between ORB and I/O subsystem), and alternative transport programming APIs (*e.g.*, sockets vs. TLI). Domains where highly optimized ORB messaging and transport protocols are essential include (1) teleconferencing applications running over high-speed networks, such as Gigabit Ethernet or ATM, and (2) real-time applications running over embedded system interconnects, such as VME or CompactPCI.

Conventional CORBA implementations have the following limitations that make it hard for them to support performance-sensitive applications effectively:

1. Static protocol configurations: Many ORBs support a limited number of statically configured protocols, typically only GIOP/IIOP.

2. Lack of protocol control interfaces: Many ORBs do not allow applications to define protocol policies and attributes, such as peak virtual circuit bandwidth.

3. Single protocol support: Many ORBs do not support simultaneous use of multiple inter-ORB messaging or transport protocols.

4. Lack of real-time protocol support: Many ORBs have limited or no support for specifying and enforcing real-time protocol requirements end-to-end across a backplane, network, Internet.

3.2 Pluggable Protocols Framework Requirements

The limitations of conventional ORBs described in Section 3.1 make it difficult for developers to leverage existing implementations, expertise, and ORB optimizations across projects or application domains. Defining a standard *pluggable protocols framework* for CORBA ORBs is an effective way to address this problem. The requirements of such a pluggable protocols framework for CORBA include the following:

1. Define standard, unobtrusive protocol configuration interfaces: To address limitations with conventional ORBs, a pluggable protocols framework should define a standard set of components and APIs to install ESIOPs and their transport-dependent instances. Most applications need not use this interface directly. Therefore, the pluggable protocol interface should be exposed only to developers interested in defining new protocols or in configuring existing protocol implementations in new ways.

2. Use standard CORBA programming and control interfaces: To ensure application portability, clients should program to standard application interfaces defined in CORBA IDL, even if pluggable ORB messaging or transport protocols are used. Likewise, object implementors need not be aware of the underlying framework. However, developers should be able to set policies that control the ORB's choice of protocol and protocol attributes. Moreover, these interfaces should transparently support certain real-time ORB features, such as scatter/gather I/O, optimized memory management, and strategized concurrency models [9].

3. Simultaneous use of multiple ORB messaging and transport protocols: To address the lack of support for multiple inter-ORB protocols in conventional ORBs, a pluggable protocols framework should support different messaging and transport protocols simultaneously within an ORB endsystem. The framework should transparently configure inter-ORB protocols either statically, *i.e.*, during ORB initialization [18], or dynamically, *i.e.*, during run-time connection establishment.

4. Support for multiple address representations: This requirement addresses the lack of support for multiple IOP support and dynamic protocol configurations in conventional ORBs. For example, each pluggable protocol implementation can potentially have a different profile and object addressing scheme. Therefore, a pluggable protocols framework should provide a general mechanism to represent these disparate address formats transparently, while also supporting standard IOR address representations efficiently.

5. Support CORBA 2.2 features and future enhancements: A pluggable protocol framework should support CORBA 2.2 features, such as object reference forwarding, connection transparency, preservation of foreign IORs and profiles, and the complete GIOP 1.1 protocol, in a manner that does not degrade end-to-end performance and determinism. Moreover, a pluggable protocols framework should accommodate future changes and enhancements to the CORBA specification, such as (1) the upcoming GIOP 1.2 protocol, which allows bi-directional requests over the same connection, (2) real-time

CORBA [18], which includes features to reserve connection and threading resources on a per-object basis, and (3) asynchronous messaging [19], which exports QoS policies to application developers.

6. Optimized inter-ORB bridging: A pluggable protocols framework should ensure that protocol implementors can create efficient, high performance inter-ORB *in-line bridges*. An in-line bridge converts inter-ORB messages or requests from one type of IOP to another. This makes it possible to bridge disparate ORB domains efficiently without incurring unnecessary context switching, synchronization, or data movement.

7. Provide common protocol optimizations and real-time features: A pluggable protocols framework should support features required by real-time CORBA applications [18], such as resource pre-allocation and reservation, end-to-end priority propagation, and mechanisms to control attributes specific to real-time protocols. These features should be implemented without modifying the standard CORBA programming APIs used by conventional, *i.e.*, non-real-time, applications.

8. Dynamic protocol bindings: To address the limitation of static protocol bindings in conventional ORBs, a pluggable protocols frameworks should support dynamic association of specific ORB messaging protocols with specific instances of ORB transport protocols. This design permits efficient and predictable configurations for both standard and customized IOPs.

3.3 Pluggable Protocol Scenarios

To illustrate the benefits of a pluggable protocols framework, we describe two scenarios where pluggable protocols are necessary to support performance-sensitive and real-time CORBA applications. These scenarios are based on our experience developing high-bandwidth, low-latency audio/video streaming applications [20] and avionics mission computing [10]. In previous work [7], we addressed the network interface and I/O system and how to achieve predictable, real-time performance. In the discussion below, we focus on ORB support for alternate protocols.

Low-latency, high-bandwidth multimedia streaming: Multimedia applications running over high-speed networks require special optimizations to utilize the full link speed. For example, consider Figure 2, where network interfaces supporting 1.2 Mbps or 2.4 Mbps link speeds are used for a CORBA-based audio/video (A/V) application. In this scenario, we will replace GIOP/IIOP with a custom ORB messaging and transport protocol that transmits A/V frames using AAL5 over ATM to take full advantage of a high-speed ATM port interconnect controller (APIC) [21]. The APIC supports (1) shared memory pools between user and kernel

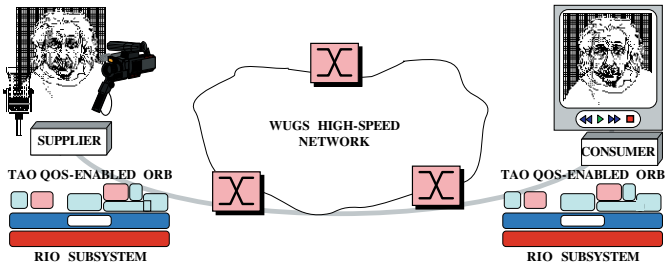

Figure 2: Example CORBA-based Audio/Video (A/V) Application

space, (2) per-VC pacing, (3) two levels of priority queues, and (4) interrupt disabling on a per-VC bases.

Leveraging the underlying APIC hardware requires the resolution of two design challenges:

1. Custom protocols: The first challenge is to create custom ORB messaging and transport protocols that can exploit the high-speed network interface hardware. For the A/V streaming application, a simple frame sequencing protocol can be used as an ESIOP. This ORB messaging protocol can be mapped onto an ORB transport protocol using AAL5.

2. Optimized protocol implementations: A second challenge is to optimize the protocol implementations, *e.g.*, by sharing memory between the application, OS kernel, and network interface. This sharing can be achieved by requiring the message encapsulation process to use memory allocated from a common buffer pool [21, 9], which eliminates memory copies between user- and kernel-space when data is sent or received. The ORB endsystem manages this memory, thereby relieving application developers from this responsibility. The ORB endsystem can also manage the APIC interface driver, interrupt rates, and pacing parameters, as outlined in [7].

Low-latency, low-jitter mission computing: Avionics mission computing applications are real-time embedded systems that manage sensors and operator displays, navigate the aircraft's course, and control weapon release. OO communication middleware for avionics mission computing applications must support deterministic real-time QoS requirements interoperating over shared memory, I/O buses, and traditional network interfaces. Support for deterministic real-time requirements is essential for mission computing tasks, such as weapon release and navigation, that must meet all their deadlines. Likewise, avionics software must support tasks, such as built-in-test and low-priority display queues, that can tolerate minor fluctuations in scheduling and reliability guarantees, but nonetheless require QoS support [8].

To enforce end-to-end application QoS guarantees, mission computing middleware must reduce overall inter-ORB communication latencies, maximize I/O efficiency, and increase overall system utilization [12, 22]. A particularly important

optimization point is the inter-ORB protocol itself, and the selection of an optimal transport protocol implementation for a particular platform.

For example, Figure 3 depicts an embedded avionics configuration with three CPU boards, each with an ORB instance. Each board is connected via a VME bus, which enables the

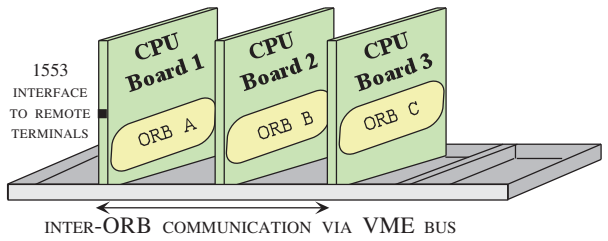

Figure 3: Example Avionics Embedded ORB Platform

ORBs on each CPU board to communicate using optimized inter-board communication, such as DMA between the individual board address spaces. CPU board 1 has a 1553 interface to communicate with so-called remote terminals, such as aircraft sensors for determining global position and forward-looking infrared radar [10]. This configuration allows ORB A to provide a bridging service that forwards ORB requests between ORBs B and C and remote terminals connected with board 1.

The scenario in Figure 3 motivates the need for multiple ORB messaging and transport protocols that can be added seamlessly to an ORB without affecting the standard CORBA programming API. For instance, ORB A could use a 1553 transport protocol adapter to communicate with remote terminals. Likewise, custom ORB messaging and transport protocols can be used to leverage the underlying VME bus hardware and eliminate sources of unbounded priority inversion.

With TAO's pluggable protocols framework, we can create optimized VME-based and 1553-based inter-ORB messaging and transport protocols. Moreover, by separating the IOP messaging from a transport-specific mapping, we can adapt TAO's pluggable protocols framework to different transmission technologies, such as CompactPCI or Fibrechannel, by changing only the transport-specific mapping of the associated inter-ORB messaging protocol.

4 Applying Patterns to Design TAO's Pluggable Protocols Framework

4.1 Architectural Overview

To meet the requirements outlined in Section 3.2, we identified logical communication component layers within TAO, factored out common features, defined general framework interfaces, and implemented components to support different concrete inter-ORB protocols. Higher-level services in the ORB, such as stubs, skeletons, and standard CORBA pseudo-objects, are decoupled from the implementation details of particular protocols, as shown in Figure 4. This decoupling is es-

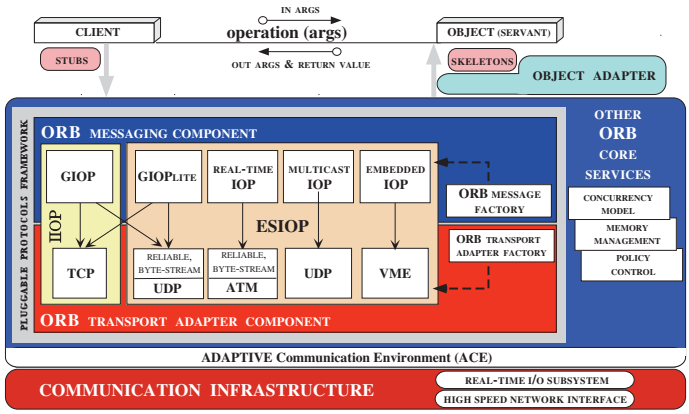

Figure 4: TAO's Pluggable Protocols Framework Architecture

sential to resolve several limitations with conventional ORBs outlined in Section 3.1, as well as to meet the requirements set forth in Section 3.2.

In general, the higher-level components and services of TAO use an abstract interface to access the mechanisms provided by its pluggable protocols framework. Thus, applications can (re)configure custom protocols without requiring global changes to the ORB. Moreover, since applications typically access only the standard CORBA APIs, TAO's pluggable protocols framework is entirely transparent to CORBA application developers.

Figure 4 also illustrates the key components in TAO's pluggable protocols framework: (1) the ORB messaging component, (2) the ORB transport adapter component, and (3) the ORB policy control component, all of which are outlined below.

ORB messaging component: This component is responsible for implementing ORB messaging protocols, such as the standard CORBA GIOP ORB messaging protocol, as well as custom ESIOPs. As described in Section 2, ORB messaging protocols should define a data representation, an ORB message format, an ORB transport protocol or transport adapter, and an object addressing format. Within this framework, ORB protocol developers can implement different data representation formats, ORB messaging protocols, and ORB connection management strategies.

Implementors of ORB messaging protocols must conform to a fixed interface that is used by TAO's ORB Core to exchange requests with peer ORBs. Each ORB messaging pro-

tocol inherits from a common base class that defines a uniform interface. This interface can be extended to include new capabilities needed by special protocol-aware policies. For example, ORB end-to-end resource reservation or priority negotiation can be implemented in an ORB messaging component. TAO's pluggable protocols framework ensures consistent operational characteristics and enforces general IOP syntax and semantic constraints, such as error handling.

Other key parts of TAO's ORB messaging component are its message factories. During connection establishment, these factories instantiate objects that implement various ORB messaging protocols. These objects are associated with a specific connection and ORB transport adapter component, *i.e.*, the object that implements the component, for the duration of the connection.

ORB transport adapter component: This component maps a specific ORB messaging protocol, such as GIOP or DCE-CIOP, onto a specific instance of an underlying transport protocol, such as TCP or ATM. Figure 4 shows an example in which TAO's transport adapter maps the GIOP messaging protocol onto TCP (this standard mapping is called IIOP). In this case, the ORB transport adapter combined with TCP corresponds to the transport layer in the Internet reference model. However, if ORBs are communicating over an embedded interconnect, such as a VME bus, the bus driver and DMA controller provide the "transport layer" in the communication infrastructure.

TAO's ORB transport component accepts a byte-stream from the ORB messaging component, provides any additional processing required, and passes the resulting data unit to the underlying communication infrastructure. Additional processing that can be implemented by protocol developers includes (1) concurrency strategies, (2) endsystem/network resource reservation protocols, (3) high-performance techniques, such as zero-copy I/O, shared memory pools, periodic I/O, and interface pooling, (4) enhancement of underlying communications protocols, *e.g.*, provision of a reliable byte-stream protocol over ATM, and (5) tight coupling between the ORB and efficient user-space protocol implementations, such as Fast Messages [23].

ORB policy control component: This component allows applications to explicitly control the QoS attributes of configured ORB transport protocols. Since it is not possible to determine *a priori* all attributes defined by all protocols, an extensible *policy control* component is provided by TAO's pluggable protocols framework. TAO's policy control component implements the QoS framework defined in the recently adopted CORBA Messaging [19] and Real-time CORBA [18] specifications.

The CORBA QoS framework allows applications to specify various *policies* at the ORB-, thread-, or object-level. Example policies relevant for pluggable protocols include buffer preallocations, fragmentation, bandwidth reservation, and maximum transport queue sizes. In general, the use of policies enables the CORBA specification to define semantic properties of ORB features precisely without (1) over-constraining ORB implementations or (2) increasing interface complexity for common use cases.

Some policies, such as timeouts, can be shared between multiple protocols. Other policies, such as ATM virtual circuit bandwidth allocation, may apply to a single protocol. Each configured protocol can query TAO's policy control component to determine its policies and use them to configure itself for user needs. Moreover, protocol implementations can simply ignore policies that do not apply to it.

TAO's policy control component also allows applications to select their protocol(s). This choice can be controlled by the `ClientProtocolPolicy` defined in the Real-time CORBA specification [18]. Using this policy, the application indicates its preferred protocol(s) and TAO's policy control component attempts to match that preference with the set of available protocols. Yet another policy controls the behavior of the ORB if an application's preferences cannot be satisfied, *e.g.*, either an exception is raised or another available protocol is selected transparently.

4.2 Design Challenges and Pattern-based Resolutions

The architecture overview in Section 4.1 outlines *how* TAO's pluggable protocols framework is designed. However, it does not motivate *why* this particular design was selected. Below, we explore each feature in TAO's pluggable protocols framework and show how they achieve the goals described in Section 3.2. Our discussion focuses on the patterns [24] we applied to resolve the key design challenges we faced.

4.2.1 Adding New Protocols Transparently

Context: The QoS requirements of many applications can be supported solely by using default static protocol configurations, *i.e.*, GIOP/IIOP, described in section 3.1. However, applications with more stringent QoS requirements often require custom protocol configurations. Implementations of these custom protocols require several related classes, such as Connectors, Acceptors, Transports, and Profiles. To be integrated into a common framework, all these classes must be created consistently. In addition, many embedded and deterministic real-time systems require protocols to be configured *a priori*, with no additional protocols required once the application is configured statically. These types of systems typically cannot afford to incur the footprint overhead associated with dynamic protocol configurations.

Problem: It should be possible to new protocols to the pluggable protocols framework without making *any* changes to the rest of the ORB. Thus, the framework must be open for extensions, but closed to modifications, *i.e.*, the Open-Closed principle [25]. Ideally, creating the new protocol and configuring into the ORB should be all that is required.

Solution: Use a Registry to maintain a collection of *Abstract Factories* [24]. In the Abstract Factory pattern, a single class defines an interface for creating families of related objects, without specifying their concrete types. Subclasses of the Abstract Factory are responsible for creating concrete classes that collaborate among themselves. In the context of pluggable protocols, each Abstract Factory can create the `Connector`, `Acceptor`, `Profile`, and `Transport` classes for a particular protocol.

Applying the solution in TAO: In TAO, the `Connector_Registry` plays the role of the protocol registry. This registry is created by TAO's Resource Factory, which is a more general Abstract Factory that creates all the ORB's strategies and policies [14]. Figure 5 depicts the connector registry and its relation to the abstract factories.

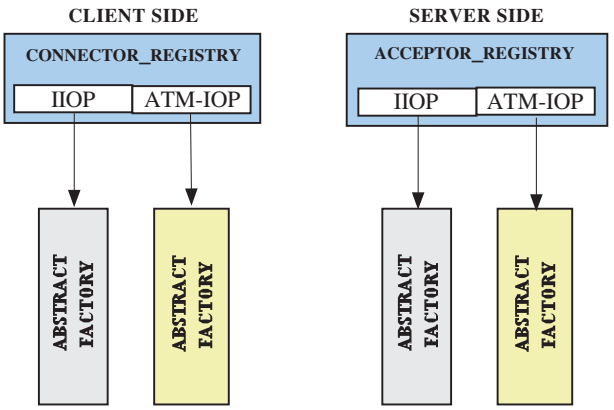

Figure 5: TAO Connector and Acceptor Registries

Note that TAO does not use these Abstract Factories directly, however. Instead, these factories are accessed via the *Facade* [24] pattern in order to hide the complexity of manipulating multiple factories behind a simpler interface. The Registry described above plays the Facade role. As shown below, these patterns provide sufficient flexibility to add new protocols transparently into the ORB.

4.2.2 Adding New Protocols Dynamically

Context: When developing new pluggable protocols, it is inconvenient to recompile the ORB and applications just to validate a new protocol implementation. Moreover, it is often useful to experiment with different protocols, *e.g.*, systematically compare their performance, footprint size, and QoS guarantees. Moreover, in 24×7 systems with high availability requirements, it is important to configure protocols dynamically, even while the system is running. This level of flexibility helps simplify upgrades and protocol enhancements.

Problem: How to populate the registry *dynamically* with the correct objects.

Solution: Use the Service Configurator [26] pattern, which decouples the implementation of a service from its configuration into the application. This pattern can be applied in either of the following ways:

1. The Service Configurator pattern can be used to dynamically load the registry class, which is a Facade that knows how to configure a particular set of protocols. To add new protocols, we must either implement a new Registry class or derive from an existing one.

   This alternative is well-suited for embedded systems with tight memory footprint constraints, since it minimizes the number of objects that are loaded dynamically. Implementations of the Service Configurator pattern can optimize for use cases where objects are configured statically. Embedded systems can exploit these optimizations to eliminate the need for loading objects dynamically in the pluggable protocols framework.
2. Use the Service Configurator to dynamically load the set of entries in the Registry. For instance, a registry can simply parse a configuration script and dynamically link the services listed in it. This is the most flexible approach, but it requires more code, *e.g.*, to parse the configuration script, load the objects dynamically, etc.

Applying the solution in TAO: TAO implements a class that maintains all parameters specified in a configuration script. Adding a new parameter to represent the list of protocols is straightforward, *i.e.*, the default registry simply examines this list and links the services into the address-space of the application, using the ACE[4] Service Configurator implementation [27] . Figure 6 depicts the connector registry and its relation to the Service Configurator.

4.2.3 Profile Creation

Context: The contents of a profile must be parsed to determine an object's location. In general, the format and semantics of the profile contents are protocol-specific. Therefore, a completely generic component for it cannot be written. Parsing the

[4]ACE provides a rich set of reusable and efficient components for high-performance, real-time communication, and forms the portability layer of TAO.

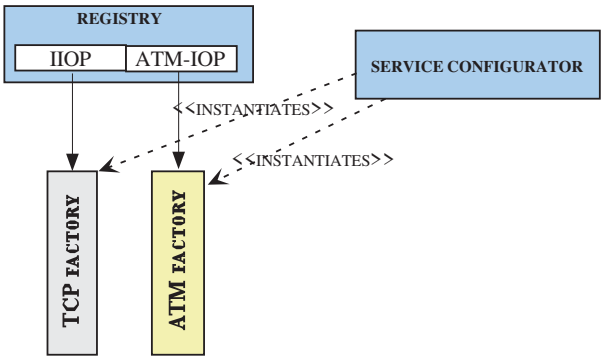

Figure 6: TAO Connector Registry and the Service Configurator

data is a relatively expensive operation that should be avoided whenever possible. It is also useful to keep multiple protocols available (see section 3.1), each one potentially using a different address representation (see section 3.2).

Problem: As new protocols are added to the system, new profile formats are introduced. It is essential that the correct parsing function be used for each profile format.

Solution: We use the Factory Method pattern [24] to create the right `Profile` class for each protocol. This pattern defines a fixed interface to create an object, while allow subclasses the flexibility to create the right type of object. Two of our classes play the Creator role in this pattern: (1) the `Connector`, using the `Profile`'s CDR representation for initialization and (2) the `Acceptor`, using the object key for initialization.

These two approaches are based on the two use cases in which a `Profile` object must be created. In the Connector case we are interpreting a `Profile` received remotely, whereas in the Acceptor case we are building a `Profile` for a local object. As usual, the `Connector_Registry` and the `Acceptor_Registry` are used as Facades [24] that locate the appropriate `Connector` or `Acceptor` and delegate to it the job of building the object.

Applying the solution in TAO: The `Profile` class is used to represent a profile specific to each protocol. This class provides an abstract interface for parsing, marshaling, hashing, and comparing profiles. In addition, it provides a unit of encapsulation to maintain information about forwarding and caching connections established to a particular server.

4.2.4 Decoupling ORB Messaging and Transport Protocol Implementations

Context: A single ORB messaging protocol can be mapped to multiple transport protocols. For example, GIOP can be mapped to any reliable, connection-oriented transport protocol, such as TCP or TP4. A single transport protocol could handle variations in the ORB messaging protocol, *e.g.*, versions of GIOP that differ in the number and types of messages defined, as well as in the format of those messages. However, it is certainly possible to write a single mapping to TCP/IP.

An ORB messaging protocol imposes requirements on the transport protocols it can use. For instance, the transport requirements assumed by GIOP are described in Section 2. In some cases, however, it may be desirable to map messaging protocols that do not support all the required semantics, but which can be implemented using the features of the transport. For example, GIOP over AAL5 may require an extra sub-layer to control re-transmissions, ACK messages, and timeouts.

Problem: The ORB Messaging protocol implementations must be independent of the adaptation layer needed for transports which do not satisfy all their requirements. Otherwise, the same messaging protocol may be reimplemented needless for each transport, which is time-consuming, error-prone, and potentially time/space inefficient. Likewise, for those transports that can support multiple ORB Messaging protocols, it must be possible to isolate them from the details of the ORB messaging implementation. Care must be taken, however, because not all ORB Messaging protocols can be used with all transport protocols, *i.e.*, some mechanism is need to ensure only semantically compatible protocols are configured cite-Johnson:95a.

Solution: Use the Layers architecture pattern [15], which decomposes the system into groups of components, each one at a different level of abstraction.[5] For the client, the ORB uses a particular ORB messaging protocol to send a request. This ORB messaging protocol delegates part of the work to the transport adapter component that completes the message and sends it to the server. If the low-level transport in use, *e.g.*, ATM, UDP, TCP/IP, etc., does not satisfy the requirements of the ORB messaging protocol, the ORB transport adapter component can implement them.

In the server, the transport adapter component receives data from the underlying communication infrastructure, such as sockets or shared memory, and it passes the message up to the ORB messaging layer. As with the client, this layer can be very lightweight if the requirements imposed by the ORB messaging layer are satisfied by the transport. Otherwise, it may need to implement those requirements by building upon the features offered by the concrete transport adapter component.

Applying the solution in TAO: As shown in Figure 7, TAO implements the messaging protocol and the transport proto-

[5] Protocol stacks based on the Internet or ISO OSI reference models are common examples of the Layers architecture.

col in separate components. The client ORB uses the current

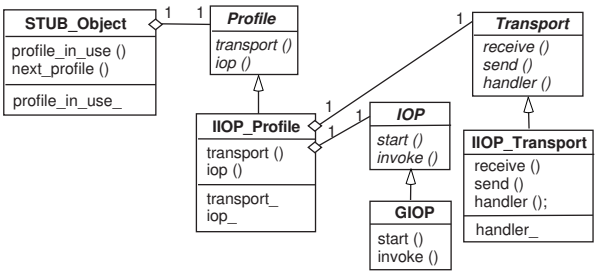

Figure 7: Client Inter-ORB and Transport Class Diagram

profile to find the right transport and ORB messaging implementations. The creation and initialization of these classes is controlled by the `Connector` (described in Section 4.2.7), with each `Connector` instance handling a particular ORB messaging/transport tuple.

Figure 8 illustrates how the server's implementation uses the same classes, but with a different relationship. In particular, the transport class calls back the messaging class when data is received from the IPC mechanism. As with the client, a factory, in this case the `Acceptor`, creates and initializes these objects.

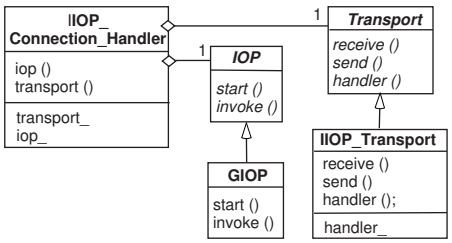

Figure 8: Server Inter-ORB and Transport Class Diagram

4.2.5 Exception Propagation and Error Detection

Context: The server and client use the same exceptions to inform the application of failures in the communication media. The ORB must be able to selectively ignore certain communication errors and transparently re-issue the request using alternative addresses or resources.

Problem: If the ORB uses exceptions to internally communicate failures, it can be confused by remote exceptions generated by the server.

Solution: Once again, apply the Layers pattern [15] to limit exceptions to communicate failures only between the higher levels of the ORB and the application. The pluggable protocols framework simply uses integral return values to indicate an error. These return values are transformed to the appropriate CORBA exception when reporting the error to the application.

Applying the solution in TAO: All components, *i.e.*, `Acceptors`, `Connectors`, `Transports`, and `Profiles`, in TAO's pluggable protocols framework return an integer to indicate failure. Higher-level components in TAO check those values and raise the appropriate CORBA exceptions.

One drawback of using return codes rather than exceptions is that it may burden ORB developers, who must explicitly check for errors, rather than writing exception handlers. Since TAO must run on platforms that do not support native C++ exceptions, however, it is already necessary to check return values, so there is no additional burden.

4.2.6 Adapting TAO to the ACE Framework

Context: As mentioned above, TAO is built using the reusable and portable ACE framework [27] components, particularly `Reactors`, `Acceptors`, `Connectors`, `Service Handlers`, and ACE socket IPC wrapper facades [14]. TAO's pluggable protocols framework uses inheritance and dynamic binding to configure these ACE components to create new protocols.

Problem: Using the lower-level ACE socket IPC wrapper facade components directly is infeasible since ACE avoids polymorphism at this level to eliminate the overhead of virtual methods by non-optimizing compilers [10]. For example, the ACE connectors for UNIX-domain sockets and Internet-domain sockets have no common ancestor that can be used to dispatch methods in subclasses polymorphically. However, a pluggable protocols framework must be able to establish connections using any protocol.

Solution: Use the External Polymorphism pattern [28] to encapsulate ACE components behind their TAO counterparts. This pattern enables classes that are not related by inheritance, or have no virtual methods, to be treated polymorphically.

Applying the solution in TAO: A TAO `Acceptor` contains an ACE `Acceptor`, which is ultimately registered with an ACE `Reactor` used by the ORB to demultiplex IOP events to the appropriate transport handlers. Eventually, the ACE IPC components accept a connection and creates an ACE `Service Handler` to handle the communication. Our TAO-level `Acceptor` encapsulates that `Service Handler` in a `Transport` adapter object and passes it up to the ORB.

4.2.7 Actively Establishing Connections

Context: When a client references an object, the ORB must obtain the corresponding profile list (which is derived from the

IOR and a profile ordering policy) and transparently establish a connection to the server.

Problem: There can be one or more combinations of inter-ORB and transport protocols available in an ORB. For a given profile, the ORB Core must verify the presence of the associated IOP and transport protocol, if available. It then must locate the applicable `Connector` and delegate to it to establish the connection.

Solution: We use the Connector pattern [29] to actively establish a connection to a remote object. This pattern decouples the connection establishment from the processing performed after the connection is successful. As before, the `Connector Registry` shown in Figure 9 is used to locate the right `Connector` for the current profile. This is simply a matter of matching the tags for the profile and the `Connector`.

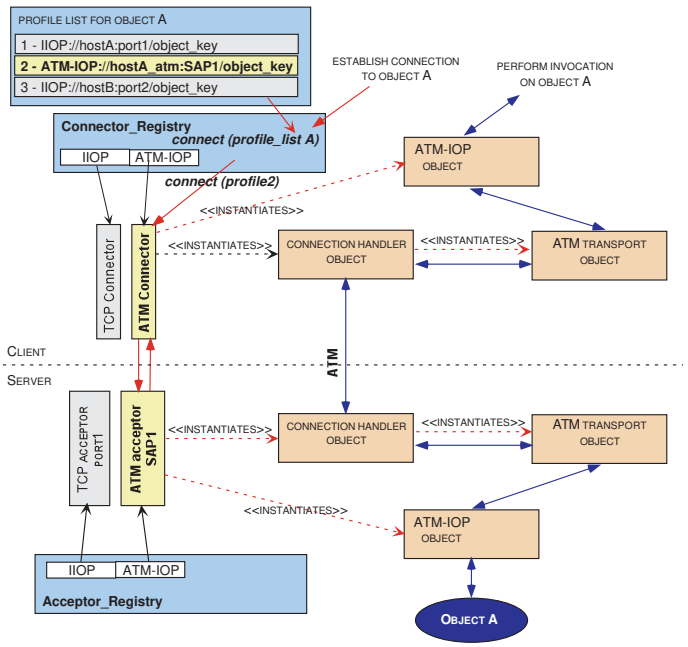

Figure 9: Connection Establishment Using Multiple Pluggable Protocols

Applying the solution in TAO: As described in Section 4.2.6, `Connectors` are adapters for the ACE implementation of the Connector pattern. Thus, they are typically lightweight objects that simply delegate to a corresponding ACE component.

Figure 10 shows the base classes and their relations for IIOP. This figure shows an explicit co-variance between the `Profile` and the `Connectors` for each protocol. In general, a `Connector` must downcast the `Profile` to its specific type. This downcast is safe because profile creation is

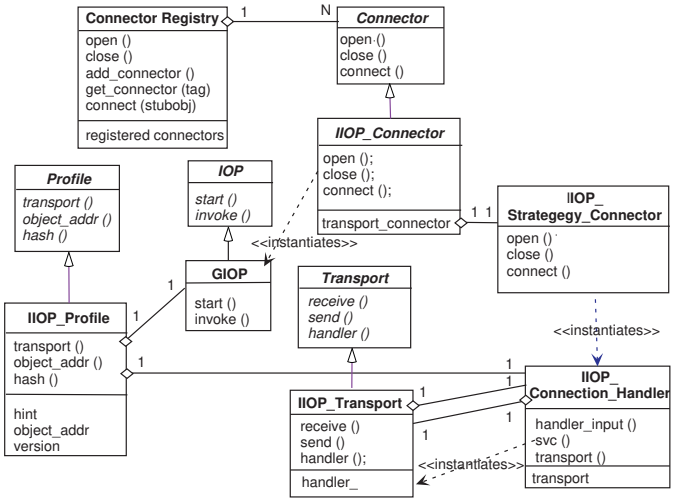

Figure 10: Client Pluggable Protocol Class Diagram

limited to the `Connector` and `Acceptor` registries. In both cases, the profile is created with a matching tag. The tag is used by the Connector Registry to choose the `Connector` that can handle each profile.

As shown in the same figure, the Connector Registry manipulates only the base classes. Therefore, new protocols can be added without requiring any modification to the existing pluggable protocols framework. When a connection is successfully established, the `Profile` is passed a pointer to the particular IOP object and to the `Transport` objects that were created.

4.2.8 Passively Accepting Connections

Context: A server can accept connections at one or more endpoints, potentially using the same protocol for all endpoints. The set of protocols that an ORB uses to play the client role need not match the set of protocols used for the server role. Moreover, the ORB can even be a "pure client," *i.e.*, a client that only makes requests, in which case it can use several protocols to make requests, but receive no requests from other clients.

Problem: The server must generate an IOR that includes all possible inter-ORB and transport-protocol-specific profiles for which the object can be accessed. As with the client, it should be possible to add new protocols without changing the ORB.

Solution: We use the Acceptor pattern [29] to accept the connections. As with the Connector pattern, an Acceptor decouples the connection establishment from the processing performed on that connection. However, in the Acceptor pattern, the connection is accepted *passively*, rather than being initiated *actively*.

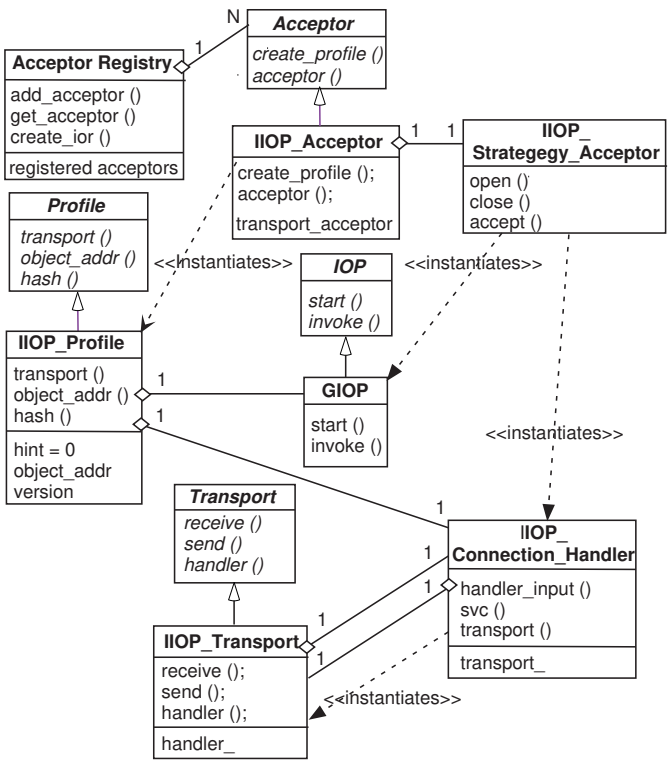

Figure 11: Server Pluggable Protocol Class Diagram

Applying the solution to TAO: Figure 11 illustrates how TAO's pluggable protocols framework leverage the design presented in Section 4.2.1. The concrete ACE `Service Handler` created by the ACE `Acceptor` is responsible for implementing the External Polymorphism pattern and encapsulating itself behind the `Transport` interface defined in our pluggable protocols framework.

As discussed in Section 4.2.6, we use the Adapter pattern to leverage the ACE implementation of the Acceptors. This pattern also permits a seamless integration with the lower levels of the ORB. In the Acceptor pattern, the `Acceptor` object is a factory that creates `Service Handlers`. `Service Handlers` are responsible for performing I/O with their connected peers. In TAO's pluggable protocol framework, the `Transport` objects are `Service Handlers` implemented as abstract classes. This design shields the ORB from variations in the `Acceptors`, `Connectors`, and `Service Handlers` for each particular protocol.

When a connection is established, the concrete `Acceptor` creates the appropriate `Connection Handler` and IOP objects. The `Connection Handler` also creates a `Transport` object that functions as a bridge. As with the `Connector`, the `Acceptor` also acts as a bridge object, hiding the transport- and strategy-specific details of the acceptor.

5 The Performance of TAO's Pluggable Protocols Framework

Despite the growing demand for off-the-shelf middleware in many application domains, a widespread belief persists in the embedded systems community that OO techniques are not suitable for real-time systems due to performance penalties attributed to the OO paradigm [10]. In particular, the dynamic binding properties of OO programming languages and the indirection implied in OO designs seem antithetical to real-time systems, which require predictable execution behavior and low latency. Therefore, the results presented in this section are significant since they illustrate empirically how the choice of patterns described in Section 4.2 makes it possible to implement very predictable, efficient, and scalable middleware *without* compromising non-functional requirements, such as portability, flexibility, reusability, and maintainability, offered by OO communication middleware.

To quantify the benefits and costs of TAO's pluggable protocols framework, we conducted a several benchmarks using two different ORB messaging protocols, GIOP and GIOPlite, and two different transport protocols, VME and Ethernet. These benchmarks are based on our experience developing OO communication middleware for avionics mission computing applications [10].

5.1 Hardware/Software Benchmarking Platform

All benchmarks in this section were run on two 200 MHz PowerPCs with 64 Mbytes of RAM connected with a 10 Mbps Ethernet and a 320 Mbps Dy4-178 VME bus. The OS used for the benchmarking was VxWorks 5.3.1, which is a real-time OS that supports multi-threading and interrupt handling.

Benchmarks were run using the standard GIOP ORB messaging protocol, as well as TAO's GIOPlite messaging protocols [9]. GIOPlite is a streamlined version of GIOP that removes $\geq$15 extraneous bytes from the standard GIOP message and request headers.[6] These bytes include the GIOP magic number (4 bytes), GIOP version (2 bytes), flags (1 byte), Request Service Context (at least 4 bytes), and Request Principal (at least 4 bytes).

For the Ethernet tests, the GIOP and GIOPlite ORB messaging protocols were run using the VxWorks TCP/IP socket

[6] Since the request header size is variable it is not possible to precisely pinpoint the proportional savings represented by these bytes. In many cases, however, the reduction is as large as 25%.

library.[7] For the VME tests, GIOP and GIOPlite ran over a custom backplane protocol that was integrated into TAO's pluggable protocols framework via an ORB transport adapter component and a VxWorks device driver. No changes were required to our standard CORBA benchmarking tool, called `IDL_Cubit` [11], for either of the ORB messaging and transport protocol implementations.

5.2 Blackbox Benchmarks

Blackbox benchmarks measure the end-to-end performance of a system from an external (*i.e.*, application) perspective. In our experiments, we used blackbox benchmarks to compute the average one-way or two-way response time incurred by clients sending various types of data using Ethernet and VME.

Measurement technique: The `IDL_Cubit` benchmark uses a single-threaded client that issues one-way or two-way IDL operations at the fastest possible rate. The server performs the operation, which cubes each parameter in the request. The client thread waits for the response and checks that it is correct. Interprocess communication is performed over Ethernet and VME, as described above.

We measure throughput for operations using a variety of IDL data types, including `void`, `sequence`, and `struct` types. The `void` data type instructs the server not to perform any processing other than that necessary to prepare and send the response, *i.e.*, it does not cube any input parameters. The `sequence` and `struct` data types exercise TAO's (de)marshaling engine. The `struct` contains an `octet`, a `long`, and a `short`, along with padding necessary to align those fields. We also measure throughput using long and short sequences of the `long` and `octet` types. The sequences of type `long` contain 4 and 1,024 members, while the sequences of type `octet` contain 16 and 4,096 members.

Blackbox results: The blackbox benchmark results are shown in Figure 12. Each bar in the chart represents latency averaged over many two-way operation calls. As expected, the 320 Mbps VME transport protocol significantly outperformed the 10 Mpbs Ethernet, even for small requests. For single parameters and short `sequence` parameters, round-trip latencies were 15-25% lower for VME. For long `sequence` parameters (where the message size was greater than Ethernet's MTU) the difference was even more dramatic, with VME outperforming Ethernet over 600%.

GIOPlite outperformed GIOP by a small margin, typically 2%. These results suggest that more substantial changes to the GIOP message protocol are required to achieve significant

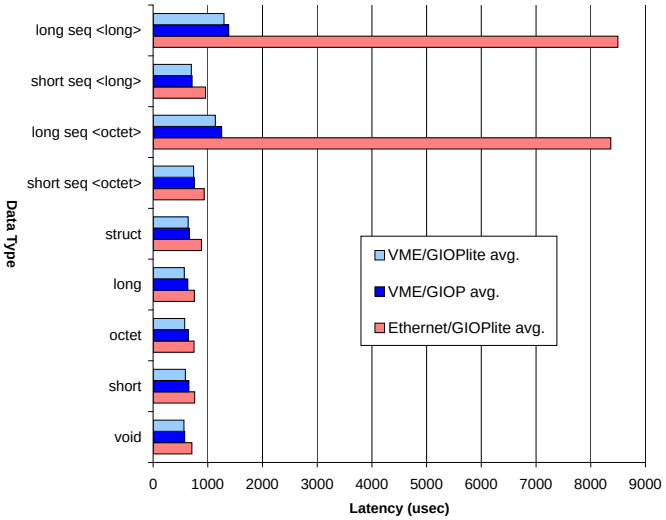

Figure 12: TAO's Pluggable Protocols Framework Latency for VME and Ethernet

performance improvements. However, these results also illustrate that GIOP message footprint has a relatively minor performance impact over high-speed networks and embedded interconnects. Naturally, the impact of GIOP message footprint for lower-speed links, such as second-generation wireless systems or low-speed modems, is more significant.

Figure 13 compares the overhead incurred in the ORB, OS, and VME driver for one-way `IDL_Cubit` calls transmitting sequences of type `long`, with lengths that are powers of 2, ranging from 4 bytes to 2,048 bytes.[8] The segments in each bar represent the combined VME/OS overhead and the ORB overhead. The latter is broken down into client-side and server-side processing time. The client side is further decomposed into

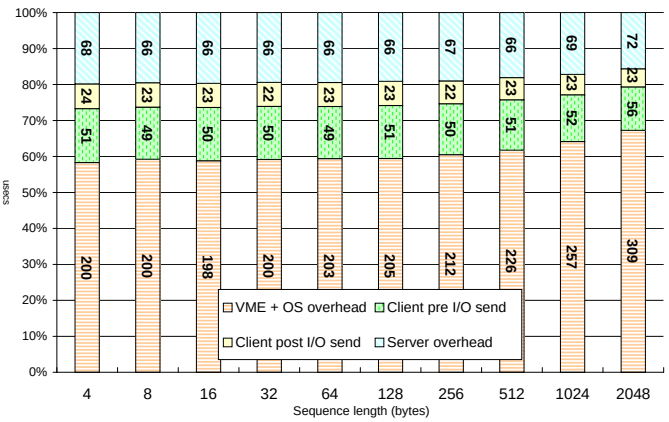

Figure 13: Comparison of ORB and VME/OS Overhead Using VMEtro Timeprobes

[7] The bandwidth of 10 Mbps Ethernet is much lower than 320 Mbps VME. Since the performance gains from GIOPlite were small, the results for the two protocols using Ethernet were nearly identical. Therefore, we only show the GIOPlite results.

[8] The X-axis is in logarithmic scale.

pre-I/O send processing time, *i.e.*, before any data has been sent to the server, and *post-I/O send processing time*, *i.e.*, after data transfer to the server has started. The time in μsecs for each step is displayed on its segment.

Client post-I/O send processing proceeds concurrently with the VME transfer. Thus, the sum of the four numbers on any segment is actually greater than the total time observed for the one-way operation. Our whitebox results illustrate that the total ORB overhead is considerably less than the VME overhead, even for very short data transfers.

Figure 13 also shows that, as the size of the operation parameters increases, VME overhead grows faster than the ORB overhead. This result illustrates that ORB overhead is largely independent of the request size. In particular, demultiplexing a request, creating message headers, and invoking an operation upcall are not affected by the size of the request.

The only overhead that depends on size is *(de)marshaling* (which depends on the type complexity, number, and size of operation parameters) and *data copying* (which depends on the size of the data). In our whitebox experiment, only the parameter size changes, *i.e.*, the `sequences` vary in length. Moreover, TAO's (de)marshaling optimizations [12] incur minimal overhead when running between homogeneous ORB endsystems.

For the operations tested in the `IDL_Cubit` benchmark, the overhead of the ORB is dominated by memory bandwidth limitations. Likewise, the performance of the VME driver is dominated by the backplane bandwidth. On the PowerPC/VME platform, memory bandwidth is much higher than the backplane bandwidth. Therefore, the VME overhead grows faster, as shown in Figure 13.

5.3 Whitebox Benchmarks

Whitebox benchmarks measure the performance of specific components or layers in a system from an internal perspective. In our experiments, we used whitebox benchmarks to pinpoint the time spent in key components in TAO's client and server ORBs.

5.3.1 Measurement Technique

We used a lightweight timeprobe mechanism provided by TAO to precisely pinpoint the amount of time spent in various ORB components and layers. The TAO timeprobe mechanism provides highly accurate, low-cost timestamps that record the time spent between regions of code in a software system. These timeprobes have minimal performance impact, *e.g.*, 1-2 μsec overhead per timeprobe, and no binary code instrumentation is required.

Depending on the underlying platform, TAO's timeprobes are implemented either by high-resolution OS timers or by high-precision timing hardware. An example of the latter is the VMEtro board, which is a VME bus monitor. VMEtro writes unique TAO timeprobe values to an otherwise unused VME address. These values record the duration between timeprobe markers across multiple processors using a single clock. This enables TAO to collect synchronized timestamps and accurately measure communication delays end-to-end across distributed CPUs.

Below, we examine the VMEtro-based client and server whitebox performance in detail.

5.3.2 Whitebox Results

Figure 14 shows the points in a two-way operation request path where timeprobes were inserted. Each labeled number in the figure corresponds to an entry in Table 1 and Table 2 below.

Client performance: Table 1 depicts the time in microseconds (μs) spent in each sequential activity that a TAO client performs to process an outgoing operation request and its reply. Each client outgoing step is outlined below:

Direction	Client Activities	Absolute Time (μs)
Outgoing	1. *Initialization*	36
	2. *Get object reference*	12
	3. *Parameter marshal*	operation dependent
	4. *ORB messaging send*	4
	5. *ORB transport send*	2
	6. *I/O send*	operation dependent
Incoming	7. *I/O receive*	operation dependent
	8. *ORB transport recv*	2
	9. *ORB messaging recv*	12
	10. *Parameter demarshal*	operation dependent

Table 1: μseconds Spent in Each Client Processing Step

1. In the *initialization* step, the client invocation is created, and constructors are called for the input and output Common Data Representation (CDR) stream objects that handle marshaling and demarshaling of operation parameters.

2. TAO's connector caches connections, so even though its `connect` method is called for every operation, existing connections are reused for repeated calls. For statically configured systems, such as avionics mission computing, TAO preestablishes connections, so even the initial connection setup overhead can be avoided entirely.

3. In the *parameter marshal* step, the outgoing `in` and `inout` parameters are marshaled. The overhead of this processing depends on the operation signature, *i.e.*, the number of data parameters and their type complexity.

4. In the `send` operation in the *ORB messaging* layer, the client creates a request header and frames the message. The

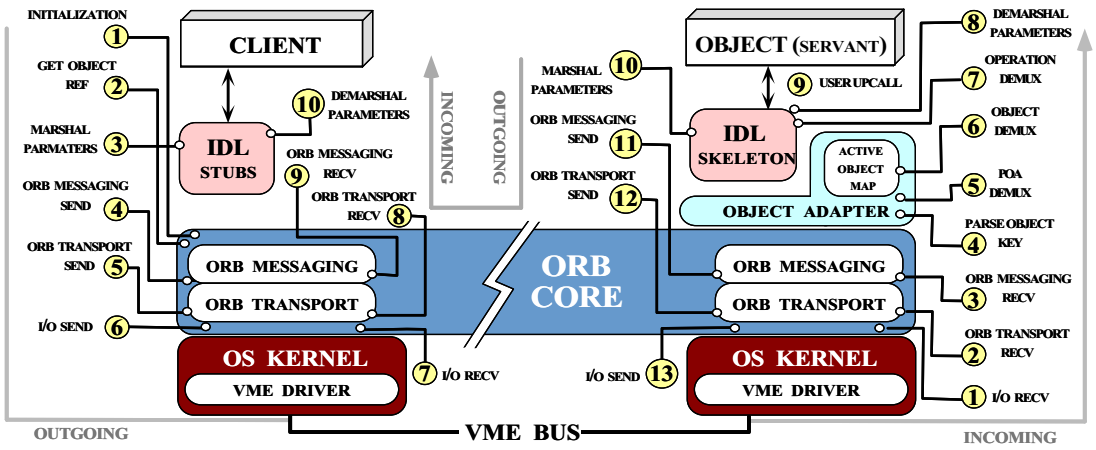

Figure 14: Timeprobe Locations for Whitebox Experiment

messaging layer then passes the message to the ORB transport component for transmission to the server. If the request is two-way, the transport component waits for and processes the response.

5. The `send` operation in the *ORB transport* component implements the connection concurrency strategy and invokes the appropriate ACE I/O operation. TAO maintains a linked list of CDR buffers [9], which allows it to use "gather-write" OS calls, such as `writev`. Thus, multiple buffers can be written atomically without requiring multiple system calls or unnecessary memory allocation and data copying.

6. The *I/O send* operation gets the peer I/O handle from the connection handler (the transport component) and performs the appropriate operation, such as copying the data over the VME using DMA.

Each client incoming step is outlined below:

7. The *I/O receive* operation copies the data from a kernel buffer to a receive CDR stream and returns control to the ORB transport component.

8. The `recv` operation in the *ORB transport* layer delegates the reading of the received messages header to the ORB messaging component. If the message header is valid, then the remainder of the message is read.

9. The `recv` operation in the *ORB messaging* layer checks the message type of the reply, and either raises an appropriate exception, initiates a location forward, or returns the reply to the calling application.

10. In the *parameter demarshal* step, the incoming reply `out` and `inout` parameters are demarshaled. The overhead of this step depends, as it does with the server, on the operation signature.

Server performance: Table 2 depicts the time in microseconds (μs) spent in each activity as a TAO server processes a request. Each incoming server step is outlined below:

Direction	Server Activities	Absolute Time (μs)
Incoming	1. *I/O receive*	operation dependent
	2. *ORB transport recv*	10
	3. *ORB messaging recv*	33
	4. *Parsing object key*	12
	5. *POA demux*	3
	6. *Servant demux*	6
	7. *Operation demux*	4
	8. *Parameter demarshal*	operation dependent
	9. *User upcall*	servant dependent
Outgoing	10. *Return value marshal*	operation dependent
	11. *ORB messaging send*	34
	12. *ORB transport send*	3
	13. *I/O send*	operation dependent

Table 2: μseconds Spent in Each Server Processing Step

1. The *I/O recv* operation copies the data from a kernel buffer to a CDR steam. This is the transition from the network transport layer to the ORB transport layer.

2. The `recv` operation in the *ORB transport* layer delegates the reading of the received message header to the ORB messaging component. If it is a valid message, then the remaining data is read and passed to the ORB messaging component.

3. The `recv` operation in the *ORB messaging* layer checks the type of the message and forwards it to the POA. Otherwise it handles the message or reports an error back to the client.

4. The *Parsing object key* step comes before any other POA activity. The time in the table includes the acquisition of a lock that is held through all POA activities (*POA demux*, *Servant demux*, and *Operation demux*).

5. The *POA demux* step locates the POA where the servant resides. The time in this table is for a POA that is one level deep, although in general, POAs can be many levels deep [9].

6. The *servant demux* step looks up a servant in the target POA. The time shown in the table for this step is based on TAO's active demultiplexing strategy [9], which locates a servant in constant time regardless of the number of objects in a POA.

7. The skeleton associated with the operation resides in the *operation demux* step. TAO uses perfect hashing [9] to locate the appropriate operation.

8. In the *parameter demarshal* step, the incoming request `in` and `inout` parameters are demarshaled. The overhead of this step depends, as it does with the client, on the operation signature.

9. The time for the *user upcall* step depends upon the actual implementation of the operation in the servant.

Each outgoing server step is outlined below:

10. In the *return value marshal* step, the `return`, `inout` and `out` parameters are marshaled. This time also depends on the signature of the operation.

11. The `send` operation in the *ORB messaging* layer passes the marshaled return data down to the ORB transport layer.

12. The `send` operation in the *ORB transport* layer adds the appropriate IOP header to the reply, sends the reply, and closes the connection if it detects an error.

13. The I/O `send` operation gets the peer I/O handle from the server connection handler and calls the appropriate `send` operation. As in the client-side I/O `send` operation described above, the server uses a gather-write I/O call.

Depending on the type and number of operation parameters, the *parameter demarshal*, *user upcall*, and *return value marshal* steps typically require the most ORB processing time. In contrast, the *ORB messaging* and *ORB transport* components in TAO's pluggable protocols framework require less than half, *i.e.*, ∼40-45%, of the ORB's overall request dispatch time. However, since the bulk of the time is spent in the VME driver and the OS, the total overhead attributed to TAO's pluggable protocol layer is ∼12

6 Related Work

The design of TAO's pluggable protocols framework is influenced by prior research on the design and optimization of protocol frameworks for communication subsystems. This section outlines this research and compares it with our work.

Configurable communication frameworks: The *x*-kernel [30], System V STREAMS [31], Conduit+ [32], ADAPTIVE [33], and F-CSS [34] 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.

The frameworks for communication subsystems listed above focus on implementing various protocol layers beneath relatively low-level programming APIs, such as sockets. In contrast, TAO's pluggable protocols framework focuses on implementing and/or adapting to transport protocols beneath a higher-level OO communication middleware API, *i.e.*, the standard CORBA programming API. Therefore, existing communication subsystem frameworks can provide building block protocol components for TAO's pluggable protocols framework.

Patterns-based communication frameworks: An increasing number of communication frameworks are being designed and documented using patterns [14, 32]. In particular, Conduit+ [32] is an OO framework for configuring network protocol 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 [24]. Likewise, the concurrency, connection management, and demultiplexing components in TAO's ORB Core and Object Adapter also have been explicited designed using patterns like Reactor, Acceptor-Connector, and Active Object [14].

CORBA pluggable protocol frameworks: The architecture of TAO's pluggable protocols framework is based on the ORBacus [35] Open Communications Interface (OCI). The OCI framework provides a flexible, intuitive, and portable interface for pluggable protocols. The framework interfaces are defined in IDL, with a few special rules to map critical types, such as data buffers.

Defining pluggable protocol interfaces with IDL permits developers to familiarize themselves with a single programming model that can be used to implement protocols in different languages. In addition, the use of IDL makes possible to write pluggable protocols that are portable among different ORB implementations and platforms.

Though the OCI pluggable protocols frameworks is useful for many applications and ORBs, it does not support zero-copy buffer interfaces. Moreover, its use of IDL to define the pluggable protocol API incurs overhead in CORBA-compliant ORBs that implement standard collocation semantics [9]. Therefore, to alleviate the drawbacks with OCI, TAO implements highly optimized pluggable protocol framework that is tuned for high-performance and real-time application requirements. For example, TAO's pluggable protocols framework can be integrated with zero-copy high-speed network interfaces [21, 36, 7, 17], embedded systems [12], or high-performance communication infrastructures like Fast Messages [23].

7 Concluding Remarks

To be an effective development platform for performance-sensitive applications, such as video-on-demand, teleconferencing, and avionics mission computing, based on CORBA must preserve communication layer QoS properties to applications end-to-end. It is essential, therefore, to define a pluggable protocols framework that allows custom inter-ORB messaging and transport protocols to be configured flexibly and transparently by CORBA applications.

This paper identifies the protocol-related limitations of current ORBs and describes a CORBA-based pluggable protocols framework we developed and integrated with TAO to address these limitations. TAO's pluggable protocols framework contains two main components: an ORB messaging component and an ORB transport adapter component. These two components allows applications developers and end-users to transparently extend their communication infrastructure to support the dynamic and/or static binding of new ORB messaging and transport protocols. Moreover, TAO's patterns-oriented OO design makes it straightforward to develop custom inter-ORB protocol stacks that can be optimized for particular application requirements and endsystem/network environments.

This paper empirically illustrates the performance of TAO's pluggable protocols framework when running CORBA applications over high-speed interconnects, such as VME. Our benchmarking results demonstrate that applying appropriate optimizations and patterns to OO communication middleware can yield highly efficient and predictable implementations, without sacrificing flexibility or reuse. These results support our contention that OO communication middleware performance is largely an implementation issue. Thus, well-tuned, standard-based OO communication middleware like TAO can replace *ad hoc* and proprietary solutions that are still commonly used in traditional embedded real-time systems.

Most of the performance overhead associated with pluggable protocols framework described in this paper stem from "out-of-band" creation operations, rather operations in the critical path. We have shown how patterns can resolve key design forces to flexibly create and control the objects in the framework. Simple and efficient wrapper facades can then be used to isolate the rest of the application from low-level implementation details, without significantly affecting end-to-end performance.

In future work, we will develop pluggable protocols for high-speed networks, such as ATM and Myrinet, as well. One focus of this work is to determine effective patterns for supporting advanced I/O features, such as buffer management schemes using intelligent I/O interfaces and shared memory, available in current high-speed network adaptors. In addition, we are exploring the integration of high-speed messaging protocols, such as Fast Messages [23], with standard CORBA OO communication middleware.

Acknowledgements

We would like to thank Greg Holtmeyer for his help in generating the whitebox and blackbox performance results in Section 5.

References

[1] R. Johnson, "Frameworks = Patterns + Components," *Communications of the ACM*, vol. 40, Oct. 1997.

[2] S. Vinoski, "CORBA: Integrating Diverse Applications Within Distributed Heterogeneous Environments," *IEEE Communications Magazine*, vol. 14, February 1997.

[3] Object Management Group, *The Common Object Request Broker: Architecture and Specification*, 2.2 ed., Feb. 1998.

[4] S. Vinoski and M. Henning, *Advanced CORBA Programming With C++*. Addison-Wesley Longman, 1999.

[5] 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.

[6] G. Parulkar, D. C. Schmidt, and J. S. Turner, "aItPm: a Strategy for Integrating IP with ATM," in *Proceedings of the Symposium on Communications Architectures and Protocols (SIGCOMM)*, ACM, September 1995.

[7] F. Kuhns, D. C. Schmidt, and D. L. Levine, "The Design and Performance of a Real-time I/O Subsystem," in *Proceedings of the 5^{th} IEEE Real-Time Technology and Applications Symposium*, (Vancouver, British Columbia, Canada), IEEE, June 1999.

[8] D. L. L. Christopher D. Gill and D. C. Schmidt, "The Design and Performance of a Real-Time CORBA Scheduling Service," *The International Journal of Time-Critical Computing Systems, special issue on Real-Time Middleware*, 1999, to appear.

[9] 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 5^{th} Conference on Object-Oriented Technologies and Systems*, (San Diego, CA), USENIX, May 1999.

[10] T. H. Harrison, D. L. Levine, and D. C. Schmidt, "The Design and Performance of a Real-time CORBA Event Service," in *Proceedings of OOPSLA '97*, (Atlanta, GA), ACM, October 1997.

[11] D. C. Schmidt, S. Mungee, S. Flores-Gaitan, and A. Gokhale, "Software Architectures for Reducing Priority Inversion and Non-determinism in Real-time Object Request Brokers," *Journal of Real-time Systems*, To appear 1999.

[12] A. Gokhale and D. C. Schmidt, "Optimizing a CORBA IIOP Protocol Engine for Minimal Footprint Multimedia Systems," *Journal on Selected Areas in Communications special issue on Service Enabling Platforms for Networked Multimedia Systems*, to appear, 1999.

[13] 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.

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

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

[16] Object Management Group, *Telecom Domain Task Force Request For Information Supporting Wireless Access and Mobility in CORBA - Request For Information*, OMG Document telecom/98-06-04 ed., June 1998.

[17] R. S. Madukkarumukumana and H. V. Shah and C. Pu, "Harnessing User-Level Networking Architectures for Distributed Object Computing over High-Speed Networks," in *Proceedings of the 2nd Usenix Windows NT Symposium*, August 1998.

[18] Object Management Group, *Realtime CORBA 1.0 Joint Submission*, OMG Document orbos/98-12-05 ed., December 1998.

[19] Object Management Group, *CORBA Messaging Specification*, OMG Document orbos/98-05-05 ed., May 1998.

[20] S. Mungee, N. Surendran, and D. C. Schmidt, "The Design and Performance of a CORBA Audio/Video Streaming Service," in *Proceedings of the Hawaiian International Conference on System Sciences*, Jan. 1999.

[21] Z. D. Dittia, G. M. Parulkar, and J. R. Cox, Jr., "The APIC Approach to High Performance Network Interface Design: Protected DMA and Other Techniques," in *Proceedings of INFOCOM '97*, (Kobe, Japan), IEEE, April 1997.

[22] A. Gokhale, D. C. Schmidt, C. O'Ryan, and A. Arulanthu, "The Design and Performance of a CORBA IDL Compiler Optimized for Embedded Systems," in *Submitted to the LCTES workshop at PLDI '99*, (Atlanta, GA), IEEE, May 1999.

[23] M. Lauria, S. Pakin, and A. Chien, "Efficient Layering for High Speed Communication: Fast Messages 2.x.," in *Proceedings of the 7th High Performance Distributed Computing (HPDC7) conference*, (Chicago, Illinois), July 1998.

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

[25] B. Meyer, *Object Oriented Software Construction*. Englewood Cliffs, NJ: Prentice Hall, 1989.

[26] P. Jain and D. C. Schmidt, "Service Configurator: A Pattern for Dynamic Configuration of Services," in *Proceedings of the 3^{rd} Conference on Object-Oriented Technologies and Systems*, USENIX, June 1997.

[27] D. C. Schmidt and T. Suda, "An Object-Oriented Framework for Dynamically Configuring Extensible Distributed Communication Systems," *IEE/BCS Distributed Systems Engineering Journal (Special Issue on Configurable Distributed Systems)*, vol. 2, pp. 280–293, December 1994.

[28] C. Cleeland, D. C. Schmidt, and T. Harrison, "External Polymorphism – An Object Structural Pattern for Transparently Extending Concrete Data Types," in *Pattern Languages of Program Design* (R. Martin, F. Buschmann, and D. Riehle, eds.), Reading, MA: Addison-Wesley, 1997.

[29] D. C. Schmidt, "Acceptor and Connector: Design Patterns for Initializing Communication Services," in *Pattern Languages of Program Design* (R. Martin, F. Buschmann, and D. Riehle, eds.), Reading, MA: Addison-Wesley, 1997.

[30] 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.

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

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

[33] 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.

[34] 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.

[35] I. Object Oriented Concepts, "ORBacus." www.ooc.com/ob.

[36] T. v. Eicken, A. Basu, V. Buch, and W. Vogels, "U-Net: A User-Level Network Interface for Parallel and Distributed Computing," in *15th ACM Symposium on Operating System Principles*, ACM, December 1995.