Applying Patterns to Develop a Pluggable Protocols Framework for ORB Middleware

Douglas C. Schmidt, Carlos O'Ryan, and Ossama Othman
{schmidt,coryan,ossama}@uci.edu
Electrical & Computer
Engineering Department
University of California, Irvine, USA

Fred Kuhns and Jeff Parsons
{fredk,parsons}@cs.wustl.edu
Department of Computer Science
Washington University
St. Louis, MO, USA*

This paper appeared as a chapter in the book *Design Patterns in Communications*, (Linda Rising, ed.), Cambridge University Press, 2001. An earlier version appear in the IFIP/ACM Middleware 2000 Conference, Pallisades, New York, April 3-7, 2000.

Abstract

To be an effective platform for performance-sensitive applications, off-the-shelf CORBA middleware must preserve the communication-layer quality of service (QoS) properties of applications end-to-end. However, the standard CORBA GIOP/IIOP interoperability protocols are not well-suited for applications with stringent message footprint size, latency, and jitter requirements. 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 the study of pluggable protocols frameworks for performance-sensitive CORBA middleware. First, we outline the key design challenges faced by pluggable protocols developers. Second, we describe how we resolved these challenges by developing a pluggable protocols framework for TAO, which is our high-performance, real-time CORBA-compliant ORB. Third, we present the results of benchmarks that pinpoint the impact of TAO's pluggable protocols framework on its end-to-end efficiency and predictability.

Our results demonstrate how the application of optimizations and patterns to CORBA middleware can yield both highly flexible/reusable designs and highly efficient/predictable implementations. These results illustrate that (1) CORBA middleware performance is largely an implementation detail and (2) the next-generation of optimized, standards-based CORBA middleware can replace many ad hoc and proprietary solutions.

1 Introduction

Standard CORBA middleware now available off-the-shelf allows clients to invoke operations on distributed components without concern for component location, programming language, OS platform, communication protocols and interconnects, or hardware [1]. However, conventional off-the-shelf CORBA middleware generally lacks (1) support for QoS specification and enforcement, (2) integration with high-speed networking technology, and (3) efficiency, predictability, and scalability optimizations [2]. 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 CORBA middleware.

To address the shortcomings of CORBA middleware mentioned above, we have developed *The ACE ORB* (TAO) [2], which is an open-source,[1] standards-based, high-performance, real-time ORB endsystem CORBA middleware that supports applications with deterministic and statistical QoS requirements, as well as "best-effort" requirements. This paper focuses on the design and implementation of a *pluggable protocols framework* that can efficiently and flexibly support 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 [3, 4], 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

*This work was supported in part by AFOSR grant F49620-00-1-0330, ATD, BBN, Boeing, Cisco, DARPA contract 9701516, Motorola Commercial Government and Industrial Solutions Sector, Motorola Laboratories, Siemens, and Sprint.

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

infrastructure, such as ATM AAL5 or specialized protocols like HPPI, then simpler ORB messaging and transport protocols can be configured to optimize 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 the standard CORBA General Inter-ORB Protocol (GIOP).

For OO researchers and practitioners, the results in this paper provide two important contributions:

1. We demonstrate empirically that the ability of standards-based CORBA middleware to support high-performance, real-time systems is largely an *implementation detail*, rather than an inherent liability, *e.g.*:

- TAO's end-to-end one-way latency overhead is only $\sim$110 μsecs using commercial off-the-self 200 MHz PowerPCs, a 320 Mbps VMEbus, and VxWorks.
- The overall roundtrip latency of a TAO two-way method invocation using the standard inter-ORB protocol and using a commercial, off-the-self Pentium II Xeon 400 MHz workstation running in loopback mode is $\sim$189 μsecs. The ORB middleware accounts for approximately 48% or $\sim$90 μsecs of the total roundtrip latency.
- Using the specialized POSIX local IPC protocol reduces roundtrip latency to $\sim$125 μsecs.

These results are as fast, or faster, than many *ad hoc*, proprietary solutions, thereby motivating the use of well-tuned, standards-based CORBA middleware, even for real-time embedded applications with stringent QoS requirements.

2. We explore how patterns can be applied to resolve key design challenges. TAO's pattern-oriented OO design can be extended to other pluggable protocols frameworks, either in standard middleware or in distributed applications using proprietary middleware.

The remainder of this paper is organized as follows: Section 2 outlines the CORBA protocol interoperability architecture; Section 3 motivates the requirements for standard CORBA pluggable protocols and outlines TAO's pluggable protocols framework; Section 4 describes the patterns that guide the architecture of TAO's pluggable protocols framework 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 [5] defines an architecture for ORB interoperability. Although a complete description of the model is beyond the scope of this paper, this section outlines the portions of the CORBA specification that are relevant to our present topic, *i.e.*, object addressing and inter-ORB protocols.

CORBA Object addressing: To identify objects, CORBA defines a generic format called the Interoperable Object Reference (IOR). An object reference identifies one instance of an object and associates one or more paths by which that object can be accessed. The same object may be located by different object references, *e.g.*, if a server is re-started on a new port or migrated to another host. Likewise, multiple server locations can be referenced by one IOR, *e.g.*, if a server has multiple network interfaces connecting it to distinct networks, there may be multiple network addresses.

References to server locations are called *profiles*. A profile provides 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 redundant addresses to increase fault-tolerance.

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

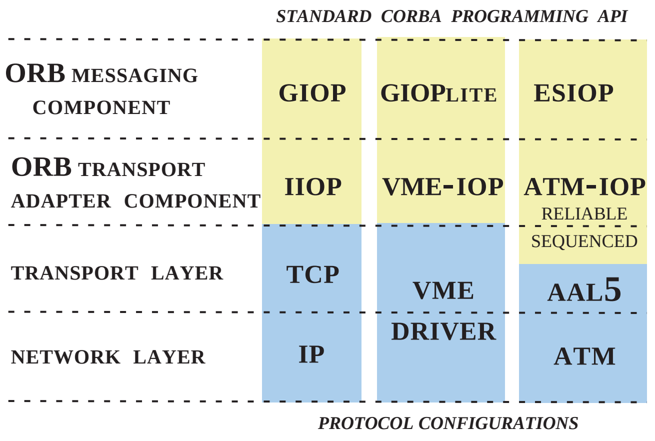


Figure 1: Relationship Between CORBA Inter-ORB Protocols and Transport-specific Mappings

In the CORBA protocol interoperability architecture, the standard *General Inter-ORB Protocol* (GIOP) is defined by the CORBA specification [5]. In addition, CORBA defines a TCP/IP mapping of GIOP, which is 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, which are known as *Environment Specific Inter-ORB Protocols* (ESIOP)s.

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. For example, the GIOP specification consists of the following:

• **A Common Data Representation (CDR) definition:** CDR is a transfer syntax that maps IDL types from their native host format to 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

• **GIOP transport adapter:** The GIOP specification describes the features of an ORB transport protocol that can carry GIOP messages. Such protocols must be reliable and connection-oriented. 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. An IOR may contain multiple profiles because there may be multiple ways to access a server, *e.g.*, through different physical network connections or alternate protocols.

[2]Version 1.1 of GIOP added a `Fragment` message and version 1.2 relaxes restrictions on message originators.

CORBA also defines other attributes that can be associated with a specific profile, group of profiles, or an entire IOR. These attributes are called *tagged components*. Tagged components can contain various types of QoS information dealing with security, server thread priorities, network connections, CORBA policies, or other domain-specific information.

An IIOP Version 1.0 profile contains the protocol version, hostname, and port number, as well as an object key that is used to demultiplex an object within a server's Object Adapter. In IIOP version 1.1, a new field was added to the GIOP header that defines a sequence of tagged components, which are name/value pairs that can be used for security, QoS, or other purposes. Tagged components may contain more information than just the object location. For example, IIOP 1.1 defines a flexible mechanism to include QoS parameters, security and authentication tokens, per-object policies for bridging with non-CORBA middleware, character set representations, and alternative addresses for a server.

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) [5]. The OMG is attempting to standardize other protocols for domains, such as wireless and mobile systems [6], which have unique performance characteristics and optimization points.

3 The Design of a CORBA Pluggable Protocols Framework

The CORBA specification provides a standard for general-purpose CORBA middleware. Within the scope of this specification, however, ORB implementors are free to optimize internal data structures and algorithms [7]. Moreover, ORBs may use specialized inter-ORB protocols and ORB services and still comply with the CORBA specification.[3] For example, ORB providers can develop additional ESIOPs for protocols such as ATM or VME, as shown in Figure 1.

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

This section first identifies the limitations of, and requirements for, protocol support in conventional CORBA ORBs. It then describes how TAO's pluggable protocols framework is designed to overcome these limitations.

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 [8]. 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 [9]. Likewise, network protocols, such as IPv4, lack packet admission policies and rate control capabilities, 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. VIA [10]). Domains where highly optimized ORB messaging and transport protocols are particularly important include (1) multimedia 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: Conventional ORBs support a limited number of statically configured protocols, often just GIOP/IIOP over TCP/IP.

2. Lack of protocol control interfaces: Conventional ORBs do not allow applications to configure key protocol policies and properties, such as peak virtual circuit bandwidth or cell pacing rate.

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

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

3.2 Pluggable Protocols Framework Requirements

The limitations of conventional ORBs described in Section 3.1 make it hard 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 for such a pluggable protocols framework for CORBA include the following:

1. Define standard, unobtrusive protocol configuration interfaces: To address the limitations of conventional ORBs, a pluggable protocols framework should define a standard set of APIs to install ESIOPs and their transport-dependent components. Most applications need not use this interface directly. Therefore, the pluggable protocols interface should be exposed only to application developers interested in defining new protocols or in configuring existing protocol implementations in novel 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. Developers should be able to set policies, however, that control the ORB's choice of protocols and protocol properties. Moreover, these interfaces should transparently support certain real-time ORB features, such as scatter/gather I/O, optimized memory management, and strategized concurrency models [7].

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 configure inter-ORB protocols transparently, either *statically* during ORB initialization [11] or *dynamically* during ORB run-time [12].

4. Support for multiple address representations: This requirement addresses the lack of support for multiple Inter-ORB protocols and dynamic protocol configurations in conventional ORBs. For example, each pluggable protocol implementation can potentially have a different profile and object addressing strategy. 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 standard features and future enhancements: A pluggable protocols framework should support standard CORBA [13] features, such as object reference forwarding, connection transparency, preservation of foreign IORs and profiles, and the GIOP 1.2 protocol, in a manner that does not degrade end-to-end performance and predictability. Moreover, a pluggable protocols framework should accommodate forthcoming enhancements to the CORBA specification, such as (1) *fault tolerance* [14, 15], which supports group communication, (2) *real-time properties* [11], which include features to reserve connection and threading resources on a per-object basis, (3) *asynchronous messaging* [16], which exports QoS policies to application developers, and (4) *wireless access and mobility* [6], which defines lighterweight Inter-ORB protocols for low-bandwidth links.

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 [11], such as resource pre-allocation and reservation, end-to-end priority propagation, and mechanisms to control properties specific to real-time protocols. These features should be implemented without modifying the standard CORBA programming APIs used by applications that do not possess real-time QoS requirements.

8. Dynamic protocol bindings: To address the limitations with static, inflexible protocol bindings in conventional ORBs, a pluggable protocols frameworks should support dynamic binding 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 Architectural Overview of TAO's Pluggable Protocols Framework

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 2. This decoupling is essential to resolve several limitations of conventional ORBs outlined in Section 3.1, as well as to meet the requirements set forth in Section 3.2.

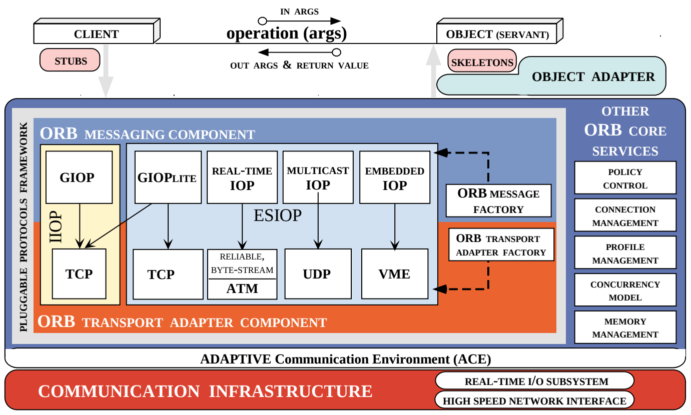


Figure 2: TAO's Pluggable Protocols Framework Architecture

In general, the higher-level components and services of TAO use the Facade pattern [17] 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, because applications typically access only the standard CORBA APIs, TAO's pluggable protocols framework can be used transparently by CORBA application developers.

The key TAO pluggable protocols framework components illustrated in Figure 2 are described below.

3.3.1 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. An ORB messaging protocol must 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 are free to implement optimized Inter-ORB protocols and enhanced transport adaptors, as long as they respect the ORB interfaces.

Each ORB messaging protocol implementation 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.

When adding a new IOP, it may not be necessary to re-implement all aspects of the ORB's messaging protocol. For example, TAO has a highly optimized CDR implementation that can be used by new IOPs [7]. TAO's CDR implementation contains highly optimized memory allocation strategies and data type translations. Thus, protocol developers can simply identify new memory or connection management strategies that can be configured into the existing CDR components.

Message factories are another key part of TAO's ORB messaging component. 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.

3.3.2 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 2 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 [18].

3.3.3 ORB Policy Control Component

It is not possible to determine *a priori* all attributes defined by all protocols. Therefore, TAO's pluggable protocols framework provides an extensible *policy control* component, which implements the QoS framework defined in the CORBA Messaging [16] and Real-time CORBA [11] specifications. This component allows applications to control the QoS attributes of configured ORB transport protocols.

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

Policies in CORBA can be set at the ORB, thread, or object level. Thus, application developers can set global policies that take effect for any request issued in a particular ORB. Moreover, these global settings can be overridden on a per-thread basis, a per-object basis, or even before a particular request. In general, CORBA's Policy framework provides very fine-grained control over the ORB behavior, while providing simplicity for the common case.

Certain 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 enables applications to select their protocol(s). This choice can be controlled by the `ClientProtocolPolicy` defined in the Real-time CORBA specification [11]. Using this policy, an application can indicate its preferred protocol(s) and TAO's policy control component then attempts to match that preference with its set of available protocols. TAO provides other policies that control the behavior of the ORB if an application's preferences cannot be satisfied. For example, an exception can be raised or another available protocol can be selected transparently.

3.3.4 Connection Management Services

Connection management services are a fundamental component of TAO's pluggable protocols framework. These services are responsible for creating ORB protocol objects dynamically and associating them with specific connections. They also interpret profiles and create object references on the server. By employing patterns and leveraging TAO's real-time features [19], protocol implementors can design high-performance IOPs that enforce stringent QoS properties.

The connection management services are implemented with *connectors*, *acceptors*, *reactors* and *registries* that keep track of available protocols, create protocol objects, and interpret profiles and object addresses. Acceptors and connectors implement the *Acceptor-Connector* pattern [3], which decouples the task of connection establishment and connection handler initialization from subsequent IOP message processing. The connectors and acceptors register themselves with their corresponding registries. The registries in turn keep track of available ORB message and transport protocols and are responsible for interpreting object references.

TAO's connection management services behave differently depending on whether the ORB plays the role of a client or a server, as outlined below.

Client ORB components: In the client ORB, the `Connector_Registry` and `Connector` establish connections to server objects and link the constituent objects together statically or dynamically. When a client application invokes an operation, it uses the list of profiles derived from the object's IOR.

For each inter-ORB and transport protocol combination available in the ORB, there is a corresponding `Connector` object responsible for performing the connection. The registry will cycle through the list of profiles for an object, requesting the appropriate connector to attempt a connection. If a connect succeeds, then the search is concluded and the successful profile is returned to the client. If no connect succeeds, the ORB throws a `transient` exception to the client.

Server ORB components: In the server ORB, an `Acceptor` waits passively for a connection event using a `Reactor` in accordance with the Reactor pattern [3]. Different concurrency architectures may be used, such as single-threaded, thread-per-connection or thread-per-priority [20]. The actual concurrency strategy used is provided as a service by TAO's ORB Core and the pluggable protocols framework. Regardless of the threading and connection concurrency strategy, the basic steps are the same:

1. An `Acceptor` listens to endpoints and waits for connection requests.
2. When a connection is accepted, a connection handler object and IOP object are created.

The `Acceptor_Registry` creates object references for registered server objects. When an object is advertised, the registry will request each registered `Acceptor` to create a profile for this object. The `Acceptor` will place in this profile the host address, the corresponding transport service access point (for example, port number for TCP/IP), and object key. All profiles are then bundled by the `Acceptor` into an IOR, which clients can use to access the object.

3.3.5 Multiple Profiles and Location Forwarding

As explained in Section 2, clients obtain interoperable object references (IORs), which are used to locate the objects upon which invocations are performed. An object reference includes at least one profile, which contains information for accessing an object through different network interfaces, shared memory, security restrictions, or QoS parameters. Multiple profiles could be used in a situation where an object resides on a server with multiple interfaces, *e.g.*, ATM and Ethernet. A profile will then be created for each of the two interfaces.

TAO's multiple profiles implementation incorporates support for *location forwarding*, which occurs when an ORB sends a request to a server object, and the server responds with a location forward reply. The location forward reply will include an IOR that the client decodes to get the list of forwarding profiles. The forwarding profiles will then replace the forwarded profile in the original profile list. Each new profile will then be tried in turn until one succeeds, is itself forwarded, or until all fail. If all forwarding profiles fail, the forwarding list is removed and the ORB continues with the next profile after the one that was forwarded initially.

There is no pre-defined limit on the number of location-forward messages that an ORB may receive. For example, if an invocation using a profile from the list of forwarding profiles should also be forwarded, the process will repeat recursively until the operation succeeds or all profiles have been tried. In practice, however, it is advantageous to limit the depth of recursion in case forwarding loops occur.

Multiple profiles can be used for other purposes, such as fault-tolerance [14, 15]. For example, consider an object that is replicated in three locations, *e.g.*, on different hosts, processes, or CPU boards in an embedded system. The IOR for this object would contain three profiles, one for for each object location. If an invocation fails using the first profile, TAO's pluggable protocols framework will transparently retry the invocation using the second profile that corresponds to the replicated object at a different location. By using some form of checkpointing or reliable multicast the state of these object instances can be synchronized.

Location forwarding can also be used for load balancing. For example, if one server becomes overloaded, it can migrate some of its objects to another server. Subsequent requests on the relocated object will then result in a location forward reply message. The message contains the new IOR for the relocated object. In the client ORB, TAO's pluggable protocols framework will then retry the object operation invocation using the new IOR transparently to the application. When system loads return to normal, the object can migrate back to the original server, and if the client performs another operation invocation, the forwarded server can reply with an exception indicating the object is no longer there. The client then retries at the original location transparently to the application.

3.4 Pluggable Protocols Scenarios

To illustrate how TAO's pluggable protocols framework has been applied in practice, we now describe two scenarios that require performance-sensitive and real-time CORBA support. These scenarios are based on our experience developing high-bandwidth, low-latency audio/video streaming applications [21] and avionics mission computing [22] systems. In previous work [20], 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.

3.4.1 Low-latency, High-bandwidth Multimedia Streaming

Multimedia applications running over high-speed networks require optimizations to utilize available link bandwidth, while still meeting application deadlines. For example, consider Figure 3, where network interfaces supporting 1.2 Mbps or 2.4 Mbps link speeds are used for a CORBA-based studio quality audio/video (A/V) application [21].

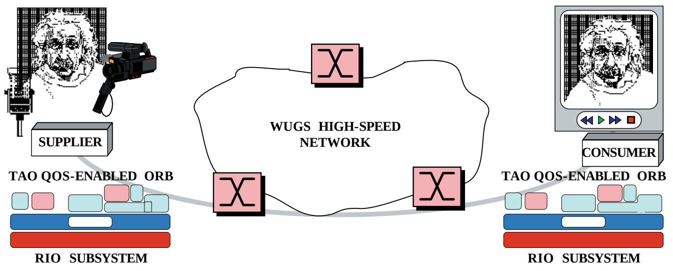


Figure 3: Example CORBA-based Audio/Video (A/V) Application

In this example, we use TAO's pluggable protocols framework to replace GIOP/IIOP with a custom ORB messaging and transport protocol that transmits A/V frames using TAO's real-time I/O (RIO) subsystem [20]. At the core of RIO is the high-speed ATM port interconnect controller (APIC) [23]. APIC is a high-performance ATM interface card that supports standard ATM host interface features, such as AAL5 (SAR). In addition, the APIC supports (1) shared memory pools between user and kernel space, (2) per-VC pacing, (3) two levels of priority queues, and (4) interrupt disabling on a per-VC bases.

We have leveraged the APIC features and the underlying ATM network to support end-to-end QoS guarantees for TAO middleware. In particular, pluggable ORB message and transport protocols can be created to provide QoS services to applications, while the ORB middleware encapsulates the actual resource allocation and QoS enforcement mechanisms. Leveraging the underlying APIC hardware requires the resolution of the following two design challenges:

Custom protocols: The first challenge is to create custom ORB messaging and transport protocols that can exploit high-speed ATM network interface hardware. A careful examination of the system requirements along with the hardware and communication infrastructure is required to determine (1) the set of optimizations required and (2) the best partitioning of the solution into ORB messaging, transport and policy components.

The A/V streaming application is primarily concerned with (1) pushing data to clients via one-way method invocations and (2) meeting a specific set of latency and jitter requirements. Considering this, a simple frame sequencing protocol can be used as the ORB's ESIOP. Moreover, because multimedia data has diminishing value over time, a reliable protocol like TCP is not required. The overhead of full GIOP is not required, therefore, nor are the underlying assumptions that require a transport protocol with the semantics of TCP.

A key goal of this scenario is to simplify the ORB messaging and transport protocol, while adding QoS-related information to support timely delivery of the video frames and audio. For example, a CORBA request could correspond to one video frame or audio packet. To facilitate synchronization between endpoints, a timestamp and sequence number can be sent with each request. The Inter-ORB messaging protocol can perform a similar function as the real-time protocol (RTP) and real-time control protocol (RTCP) [24].

The ORB messaging protocol can be mapped onto an ORB transport protocol using AAL5. The transport adapter is then responsible for exploiting any local optimizations to hardware or the endsystem. For example, conventional ORBs copy user parameters into internal buffers used for marshaling. These buffers may be allocated from global memory or possibly from a memory pool maintained by the ORB. In either case, at least one system call is required to obtain mutexes, allocate buffers, and copy the data. Thus, not only is an additional data copy incurred, but this scenario is rife with opportunities for priority inversion and jitter while waiting to acquire shared ORB endsystem resources.

Optimized protocol implementations: The second challenge is to implement an optimized pluggable protocol that implements the design described above. For example, memory can be shared throughout the ORB endsystem, *i.e.*, between the application, ORB middleware, OS kernel, and network interface, by allocating memory from a common buffer pool [23, 7]. This optimization eliminates memory copies between user- and kernel-space when data is sent or received. Moreover, the ORB endsystem can manage this memory, thereby relieving application developers from this responsibility. In addition, the ORB endsystem can manage the APIC interface driver, interrupt rates, and pacing parameters, as outlined in [20].

Figure 4 illustrates a buffering strategy where the ORB manages multiple pools of buffers to be used by applications sending multimedia data to remote nodes. These ORB buffers are shared between the ORB and APIC driver in the kernel. The transport adapter implements this shared buffer pool on a per-connection and possibly per-thread basis to minimize or reduce the use of resource locks. For example, in the strategy depicted in Figure 4, each active connection is assigned its

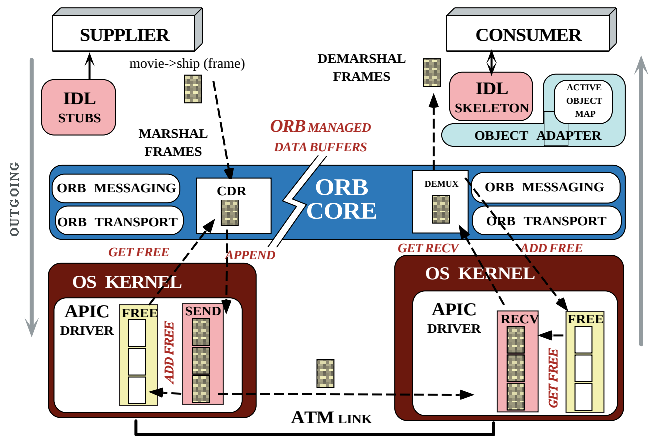


Figure 4: Shared Buffer Strategy

own send and receive queues. Likewise, there are two free buffer pools per connection, one for receive and one for send.

An ORB can guarantee that only one application thread will be active within the send or receive operation of the transport adapter. Therefore, buffer allocation and de-allocation can be performed without locking. A similar buffer management strategy is described in [23]

User applications can interact with the buffering strategy described above as follows:

- **Zero-copy:** The application requests a set of send buffers from the ORB that it uses for video and audio data. In this case, application developers must not reuse a buffer after it has been given to the ORB. When the original set of buffers are exhausted, the application must request additional buffers.

- **Single-copy:** The ORB copies application data into the ORB managed buffers. While this strategy incurs one data copy, the application developer need not be concerned with how or when buffers are used in the ORB.

Well-designed ORBs can be strategized to allow applications to decide whether data are copied into ORB buffers or not. For instance, it may be more efficient to copy relatively small request data into ORB buffers, rather than using shared buffers within the ORB endsystem. By using TAO's policy control component, this decision can be configured on a per-connection, per-thread, per-object or per-operation basis.

3.4.2 Low-latency, Low-jitter Avionics Mission Computing

Avionics mission computing applications [22] are real-time embedded systems that manage sensors and operator displays, navigate the aircraft's course, and control weapon release. CORBA 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 [25].

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 [8, 26]. 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 5 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

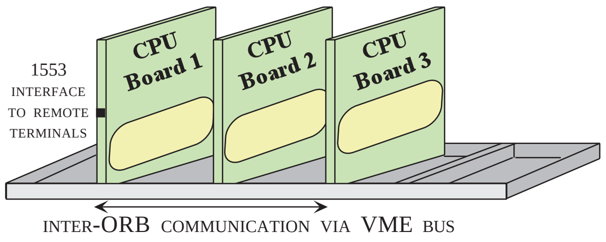

1553 INTERFACE TO REMOTE TERMINALS
CPU Board 1
CPU Board 2
CPU Board 3
INTER-ORB COMMUNICATION VIA VME BUS

Figure 5: 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 bus interface to communicate with so-called remote terminals, such as aircraft sensors for determining global position and forward- looking infrared radar [22]. 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 5 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. Leveraging the underlying bus hardware requires the resolution of the following two design challenges:

Custom protocols: 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.

Consider an embedded application that must periodically process sensor data. The sensor data is collected and forwarded aperiodically to a central, although redundant, processor. The sensor data is sent/received aperiodically. Therefore, the resulting bus transfers, interrupts, and driver processing can reduce the overall utilization of the system. For example, a DMA transfer between two CPU boards requires that the VMEBus, the source PCI bus and the destination PCI bus be acquired and data copied.

A more efficient protocol could buffer these one-way data transfers until a predetermined byte count or timeout value is reached. Thus the time required to acquire the different buses could be amortized over a larger data transfer. Additionally, given the periodic nature of the transfers rate monotonic analysis could be used to better predict system performance.

Optimized protocol implementations: To optimize the on-the-wire protocol message footprint we use a lightweight version of GIOP, called GIOPlite. GIOPlite is a streamlined version of GIOP that removes $\geq$15 extraneous bytes from the standard GIOP message and request headers.[4] 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). GIOPlite reduces the number of bytes transfered across the backplane per operation.

Another optimization that pertains to avionics mission computing involves the use of buffered one-way operations [19]. TAO's pluggable protocols framework has been optimized to send a series of queued one-way requests in a smaller number of ORB messages. For example, Figure 6 depicts the case where one-way CORBA invocations are buffered in the ORB for later delivery. In this case, a series of one-way invocations to the same object and for the same operation are queued in the same buffer and sent via a single ORB message. This results in an overall increase in throughput between CPU boards by amortizing key sources of communication overhead, such as context switching, synchronization, and DMA initialization.

[4]The request header size is variable. Therefore, 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%.

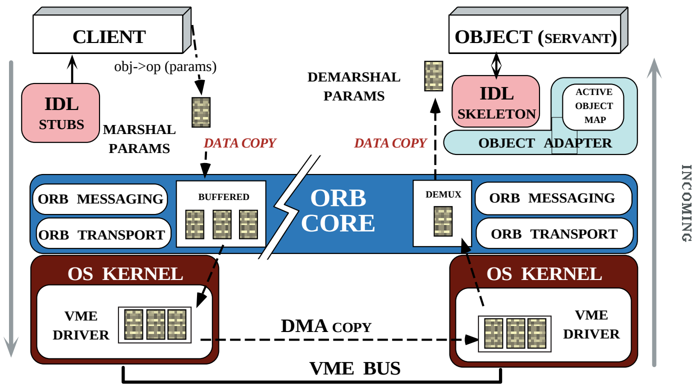


Figure 6: One-way Delayed Buffering Strategy

4 Key Design Challenges and Pattern-based Resolutions

Section 3.3 described *how* TAO's pluggable protocols framework is designed. It does not, however, motivate *why* this particular design was selected. In this section, we explore each feature in TAO's pluggable protocols framework and show how they achieve the goals described in Section 3.2. To clarify and generalize our approach, the discussion below focuses on the patterns [17] we applied to resolve the key design challenges we faced during the development process.

4.1 Adding New Protocols Transparently

Context: The QoS requirements of many applications can be supported solely by using default static protocol configurationns, *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 `Connector`s, `Acceptor`s, `Transport`s, and `Profile`s. To form a common framework, these classes must all 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 cannot afford the footprint overhead associated with dynamic protocol configurations.

Problem: It must be possible to add new protocols to TAO's 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 [27]. Ideally, creating a new protocol and configuring it into the ORB is all that should be required.

Solution: Use a *registry* to maintain a collection of *abstract factories*. In the Abstract Factory pattern [17], a single class defines an interface for creating families of related objects, without specifying their concrete types. Subclasses of an 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 role of the protocol registry is played by the `Connector_Registry` on the client and the `Acceptor_Registry` on the server. This registry is created by TAO's `Resource_Factory`, which is an abstract factory that creates all the ORB's strategies and policies [28]. Figure 7 depicts the `Connector_Registry` and its relation to the abstract factories.

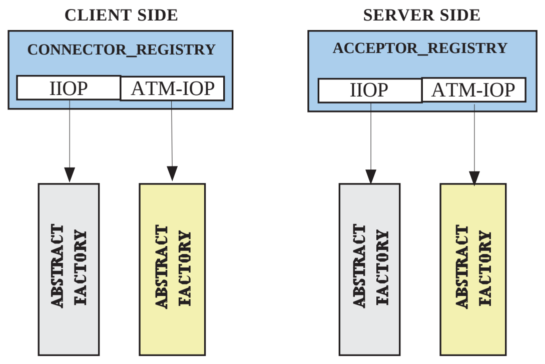


Figure 7: TAO Connector and Acceptor Registries

Note that TAO does not use abstract factories directly, however. Instead, these factories are accessed via the *Facade* [17] pattern to hide the complexity of manipulating multiple factories behind a simpler interface. The registry described above plays the role of a facade. As shown below, these patterns provide sufficient flexibility to add new protocols transparently to the ORB.

Establishing connections, manipulating profiles, and creating endpoints are delegated to the `Connector` and `Acceptor` registries respectively. Clients will simply provide the `Connector_Registry` with an opaque profile, which corresponds to an object address for a particular protocol instance. The registry is responsible for locating the correct concrete factory, to which it then delegates the responsibility for establishing the connection. The concrete factory establishes the connection using the corresponding protocol-specific instance, notifying the client of its success or failure. Thereafter, the client simply invokes CORBA operations using the selected protocol.

The server delegates endpoint creation to the `Acceptor_Registry` in a similar manner. The registry is passed an opaque endpoint representation, which it provides to the corresponding concrete factory for the indicated protocol instance. The concrete `Acceptor` factory creates the endpoint and enables the ORB to receive requests on the new endpoint.

4.2 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$\times$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 with the correct objects *dynamically*.

Solution: Use the Component Configurator [3] 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 Component Configurator pattern [3] can be used to dynamically load the registry class. This facade 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 Component 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 into the pluggable protocols framework dynamically.

2. Use the Component Configurator pattern to load the set of entries in a registry dynamically. For instance, a registry can simply parse a configuration script and link the services listed in it dynamically. This design is the most flexible strategy, but it requires more code, *e.g.*, to parse the configuration script and load the objects dynamically.

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 Component Configurator pattern implementation provided by ACE [29].[5] Figure 8 depicts the Connector_Registry and its relation to the ACE Component Configurator implementation.

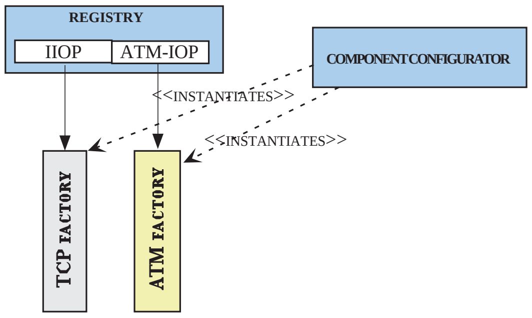


Figure 8: TAO Connector Registry and the ACE Component Configurator Implementation

4.3 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 data is a relatively expensive operation that should be avoided whenever possible. It is also useful to support multiple protocols (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 [17] to create the right Profile class for each protocol. This pattern defines a fixed interface to create an object, while allowing subclasses the flexibility to create the correct 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 manipulated. In the Connector case TAO interprets a Profile received remotely, whereas in the Acceptor it builds a Profile for a local object. As usual, the Connector_Registry and the Acceptor_Registry are used as facades [17] that locate the appropriate Connector or Acceptor and delegate the job of building the object to it.

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

Applying the solution in TAO: The Profile class is used to represent a protocol-specific profile. 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.

4.4 Decoupling ORB Messaging and Transport Protocol Implementations

Context: It is desirable to support alternative mappings between different ORB messaging protocols and ORB transport adaptors. For example, a single ORB messaging protocol, such as GIOP, can be mapped to any reliable, connection-oriented transport protocol, such as TCP. Alternatively, a single transport protocol can be the basis for alternative instantiations of ORB messaging protocols, *e.g.*, different versions of GIOP differing in the number and types of messages, as well as in the format of those messages.

An ORB messaging protocol imposes requirements on any underlying network transport protocols. For instance, the transport requirements assumed by GIOP described in Section 2 require the underlying network transport protocol to support a reliable, connection-oriented byte-stream. These requirements are fulfilled by TCP, thus leading to the direct mapping of GIOP onto this transport protocol. However, alternative network transport protocols, such as ATM with AAL5, encapsulation may be more appropriate in some environments. In this case, the messaging implementation must provide the missing semantics, such as reliability, to use GIOP.

Problem: The ORB Messaging protocol implementations must be independent of the adaptation layer needed for transports that do not satisfy all their requirements. Otherwise, the same messaging protocol may be re-implemented needlessly for each transport, which is time-consuming, error-prone, and 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 needed to ensure that only semantically compatible protocols are configured [30].

Solution: Use the Layers architectural pattern [4], which decomposes the system into groups of components, each one at a different level of abstraction.[6] The Layers architectural pattern can be implemented differently, depending on whether the ORB plays the role of a client or a server, as outlined below.

[6]Protocol stacks based on the Internet or ISO OSI reference models are common examples of the Layers architectural pattern.

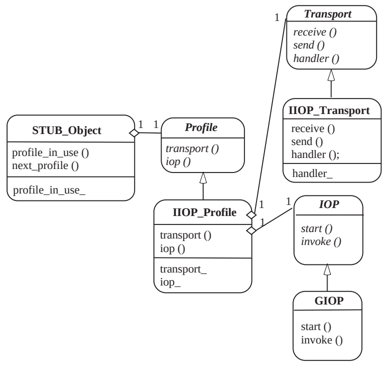


Figure 9: Client Inter-ORB and Transport Class Diagram

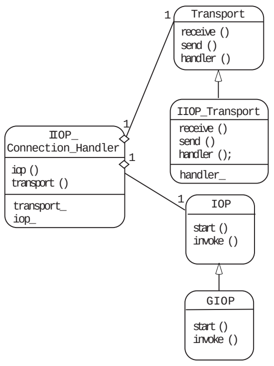


Figure 10: Server Inter-ORB and Transport Class Diagram

• **Client ORB:** 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, such as ATM, UDP, or TCP/IP, does not satisfy the requirements of the ORB messaging protocol, the ORB transport adapter component can implement them.

• **Server ORB:** 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 underlying network transport protocol. Otherwise, it must implement those missing requirements by building them into the concrete transport adapter component.

Applying the solution in TAO: As shown in Figure 9, TAO implements the messaging protocol and the transport protocol in separate components. The client ORB uses the current 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 4.8), with each `Connector` instance handling a particular ORB messaging/transport tuple.

Figure 10 illustrates how the server's implementation uses the same transport 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.

4.5 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 ignore certain communication errors selectively and re-issue the request transparently 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 architectural pattern [4] to limit exceptions to communicate failures only between the higher levels of the ORB and the application. Thus, the lower levels of the pluggable protocols framework simply use integral return values to indicate an error. These return values are transformed to the appropriate CORBA exception by the upper levels of the pluggable protocols framework when reporting the error to the application.

Applying the solution in TAO: Higher level methods in all components, *i.e.*, `Acceptor`s, `Connector`s, `Transport`s, and `Profile`s, in TAO's pluggable protocols framework raise CORBA exceptions if lower level methods return an integer value that indicates failure.

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. TAO must run on platforms that do not support native C++ exceptions, however. Therefore, it is already necessary to check return values, so there is no additional burden on TAO developers.

4.6 Adapting TAO to the ACE Framework

Context: TAO is built largely using the reusable and portable ACE framework [29] components, particularly `Reactor`s, `Acceptor`s, `Connector`s, `Service Handler`s, and ACE IPC wrapper facades [29]. 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 IPC wrapper facade components directly is infeasible because ACE avoids polymorphism at this level to eliminate the overhead of virtual methods by non-optimizing compilers [22]. Thus, 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 [31] 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 registered with an ACE `Reactor` that the ORB uses 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. As a practical consequence of this solution, there exist two sublayers within TAO's `Transport` object.

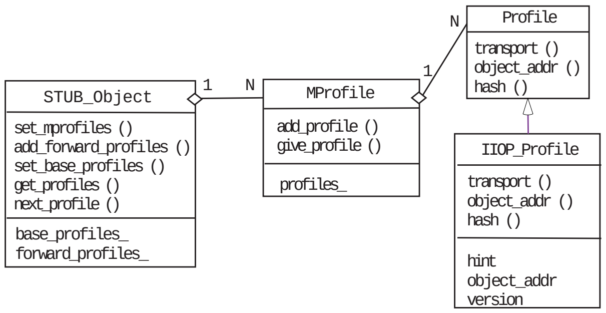


Figure 11: Class Diagram for Multiple Profile and Forwarding Support in TAO's Pluggable Protocols Framework

4.7 Multiple Profiles and Location Forwarding

Context: Object references may contain multiple profiles, and servers may specify alternate object references in response to a client's request. In addition, a CORBA-compliant ORB is required to try all object references and profiles until one succeeds *without* any client intervention.

Problem: Retries must occur transparently to the client application, even though profiles for different ORB protocols may be dissimilar and profile lists may be altered dynamically as a result of forwarding.

Solution: Apply the Proxy pattern [17] and use polymorphism and an efficient list processing strategy.

Applying the solution in TAO: Figure 11 depicts the class diagram for the solution. A `STUB_Object` is a client's local proxy for the (potentially) remote object. All communication with the server object is done through the stub proxy. While the server does not require a `STUB_Object`, the `Acceptor_Registry` will initialize an object's IOR using the `MProfile` and `Profile` classes.

Profile lists are maintained by an `MProfile` object. The profile list is stored as a simple array of pointers to `Profile` objects. All instances of IOP profiles are derived from this common `Profile` class. By relying on dynamic binding of objects, the base class can be used for both referencing and performing common method invocations on the concrete profile instances. The `MProfile` object can therefore maintain a list of `Profile` proxies to the actual concrete profile instances.

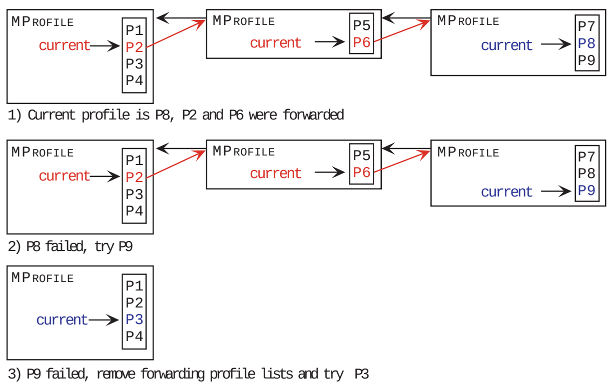
The `MProfile` object keeps track of the current profile and allows a user only to increment and decrement this reference. If the current profile is forwarded, a reference is kept in that profile to the forwarding `MProfile` object. Likewise, the forwarding `MProfile` contains a back pointer to the `MProfile` object that was forwarded. In this way, a list of `MProfile`s is maintained, corresponding to the initial and all forwarding profile lists.

When a client decodes the initial IOR, the resulting profile list is stored in an `MProfile` object. If the client receives a location forward, either as a result of a `Locate_Request` GIOP message or in a LOCATION_FORWARD reply, the received IOR is decoded and added to the `STUB_Object` using its `add_forward_profiles` method. The `MProfile` object that was forwarded keeps track of the current profile, marks it as being in a FORWARDING state, and sets a reference to the forwarding profile list. The `STUB_Object` maintains a reference to the initial profile list and to the current forwarding profile list, *i.e.*, the `MProfile` object, because the forwarding `MProfile` objects each contain a back pointer to the forwarded `MProfile` object.

Figure 12 illustrates how forwarding is represented using the `MProfile` objects. Not shown is the `STUB_Object`, which maintains references to the initial or unforwarded profile list, the current profile in use, and the last forwarding profile list. In effect, the `STUB_Object` and `MProfile` present the ordered profile list–P1, P2, P5, P6, P7, P8, P9, P3, P4–to the `Connection_Registry`.

4.8 Establishing Connections Actively

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



MProfile
current
P1
P2
P3
P4
P5
P6
P7
P8
P9
1) Current profile is P8, P2 and P6 were forwarded
2) P8 failed, try P9
3) P9 failed, remove forwarding profile lists and try P3

Figure 12: Object Reference Forwarding Example

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

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

Solution: We use the `Connector` component in the Acceptor-Connector pattern [3] to actively establish a connection to a remote object. This pattern decouples the connection establishment from the processing performed after the connection is successful. Figure 13 shows how multiple profiles may be used during connection establishment in both the client and server. This figure shows a connection to `Object A` being requested of the `Connector_Registry`. The registry will in turn try the profiles listed in the `supplied profile_list` for `Object A`. In this figure, the first profile is for an IIOP connection to `Host A` at `port 1`. Assuming the connect fails for some reason, the registry will try the second profile automatically. This profile contains a reference to the same host via ATM interface using an ESIOP.

Assuming the connect on the second profile succeeds, the `Connector` and `Acceptor` create their corresponding connection handlers and ATM-IOP transport objects. The connection handlers then create transport objects, which provide the mapping from the chosen transport protocol to a transport-independent interface used by the IOP messaging component. The connection handler is considered as part of the ORB transport adapter component.

Applying the solution in TAO: As described in Section 4.6, `Connectors` are adapters for the ACE implementation of the Acceptor-Connector pattern. Thus, they are lightweight objects that simply delegate to a corresponding ACE component. Figure 15 shows the base classes and their relations

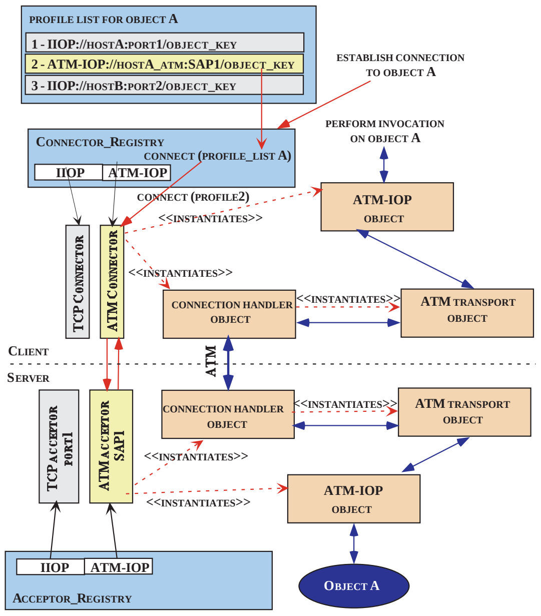


Figure 13: Connection Establishment Using Multiple Pluggable Protocols

for IIOP. This figure shows an explicit co-variance between

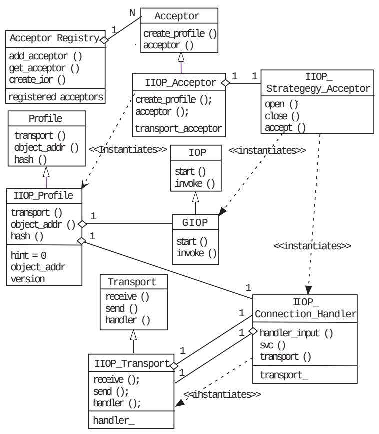


Figure 14: Server Pluggable Protocols Class Diagram

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 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 established successfully, the `Profile` is passed a pointer to the particular IOP object and to the `Transport` objects that were created.

4.9 Accepting Connections Passively

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 this 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: Use the `Acceptor` component in the Acceptor-Connector pattern [3] to accept the connections. An `Acceptor` accepts a connection *passively*, rather than being initiated *actively*, as with the `Connector` component described above.

Applying the solution to TAO: Figure 14 illustrates how TAO's pluggable protocols framework leverages the design

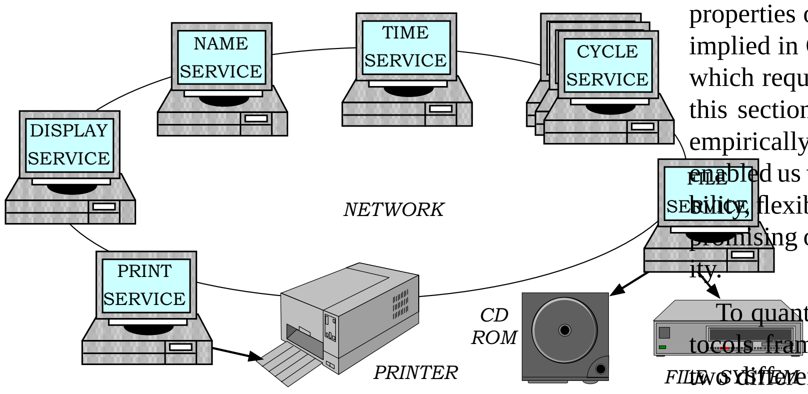


Figure 15: Client Pluggable Protocols Class Diagram

presented in Section 4.1. The concrete ACE `Service Handler` created by the ACE `Acceptor` is responsible for implementing the External Polymorphism pattern [31] and encapsulating itself behind the `Transport` interface defined in TAO's pluggable protocols framework.

As discussed in Section 4.6, TAO use the Adapter pattern [17] to leverage the ACE `Acceptor` implementation. This pattern also permits a seamless integration with lower levels of the ORB. In the Acceptor-Connector pattern, the `Acceptor` object is a factory that creates `Service Handlers`, which perform I/O with their connected peers. In TAO's pluggable protocols 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 the implementation role in the Bridge pattern [17]. As with the `Connector`, the `Acceptor` also acts as the interface role in the Bridge pattern, 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 that OO techniques are not suitable for real-time systems due to performance penalties [22]. 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 low latency and jitter. The results presented in this section are significant, therefore, because they illustrate empirically how the choice of patterns described in Section 4 enabled us to meet non-functional requirements, such as portability, flexibility, reusability, and maintainability, without compromising overall system efficiency, predictability, or scalability.

To quantify the benefits and costs of TAO's pluggable protocols framework, we conducted several benchmarks using two different ORB messaging protocols, GIOP and GIOPlite, and two different transport protocols, POSIX local IPC (also known as UNIX-domain sockets) and TCP/IP. These benchmarks are based on our experience developing CORBA middleware for avionics mission computing applications [22] and multimedia applications [21], as described in Section 3.4.

Note that POSIX local IPC is not a traditional high-performance networking environment. However, it does provide the opportunity to obtain an accurate measure of ORB and pluggable protocols framework overhead. Based on these measurements, we have isolated the overhead associated with each component, which provides a baseline for future work on high-performance protocol development and experimentation.

5.1 Hardware/Software Benchmarking Platform

All benchmarks in this section were run on a Quad-CPU Intel Pentium II Xeon 400 MHz workstation, with one gigabytes of RAM. The operating system used for the benchmarking was Debian GNU/Linux "potato" (glibc 2.1) with Linux kernel version 2.2.10. GNU/Linux is an open-source operating system that supports true multi-tasking, multi-threading, and symmetric multiprocessing.

For these experiments, we used the GIOP and GIOPlite [7] messaging protocols. GIOPlite is a streamlined version of GIOP that removes ≥15 extraneous bytes from the standard GIOP message and request headers.[7] 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).

Our benchmarks were run using the standard GIOP ORB messaging protocol, as well as TAO's GIOPlite messaging protocol. For the TCP/IP tests, the GIOP and GIOPlite ORI messaging protocols were run using the standard CORB IIOP transport adapter along with the Linux TCP/IP socke library and the loopback interface.

For the local IPC tests, GIOP and GIOPlite were used alon with the optimized local IPC transport adapter. This resulte in the following four different Inter-ORB Protocols: (1) GIO over TCP (IIOP), (2) GIOPlite over TCP, (3) GIOP over loca IPC[8] (UIOP), and (4) GIOPlite over local IPC. No change were required to our standard CORBA benchmarking tool called `IDL_Cubit` [32], for either of the ORB messaging an transport protocol implementations.

5.2 Blackbox Benchmarks

Blackbox benchmarks measure the end-to-end performance of a system from an external application perspective. In our experiments, we used blackbox benchmarks to compute the average two-way response time incurred by clients sending various types of data using the four different Inter-ORB transport protocols.

Measurement technique: A single-threaded client is used in the `IDL_Cubit` benchmark to issue two-way IDL operations at the fastest possible rate. The server performs the operation, which cubes each parameter in the request. For two-way calls, the client thread waits for the response and checks that it is correct. Interprocess communication is performed over the selected IOPs, 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 `long` sequences contain 4,096 bytes (1,024 four byte `long`s or 4,096 `octet`s) and the short sequences are 4 bytes (one four byte `long` or four `octet`s).

Blackbox results: The blackbox benchmark results are shown in Figure 16. All blackbox benchmarks were averaged

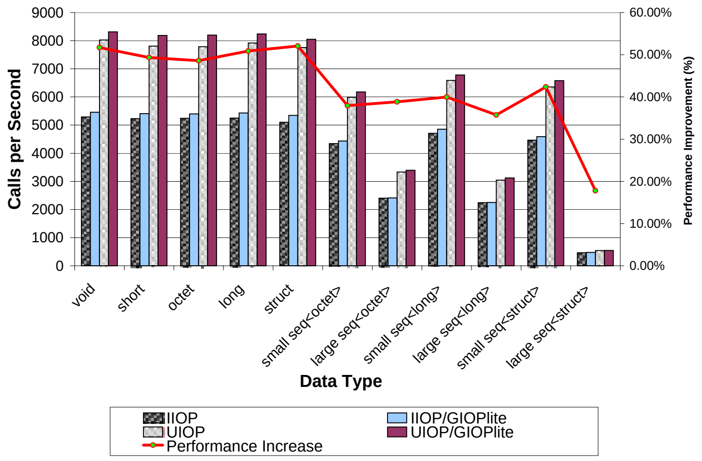


Figure 16: TAO's Pluggable Protocols Framework Performance Over Local IPC and TCP/IP

over 100,000 two-way operation calls for each data type, as shown in Figure 16.

UIOP performance surpassed IIOP performance for all data types. The benchmark results show how UIOP improves performance from 20% to 50% depending on the data type and size. For smaller data sizes and basic types, such as `octet` and `long`, the performance improvement is approximately 50%. For larger data payload sizes and more complex data types, however, the performance improvements are reduced. This result occurs due to the increasing cost of both the data copies associated with performing I/O and the increasing complexity of marshaling structures other than the basic data types.

For certain data types, additional improvements are obtained by reducing the number of data copies required. Such a situation exists when marshaling and demarshaling data of

[7]The request header size is variable. Therefore, 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%.

[8]For historical reasons, TAO retains the expression "UNIX-domain" in its local IPC pluggable protocol implementation, which is where the name "UIOP" derives from.

type `octet` and `long`. For complicated data types, such as a large `sequence` of `structs`, ORB overhead is particularly prevalent. Large ORB overhead implies lower efficiency, which accounts for the smaller performance improvement gained by UIOP over IIOP for complex data types.

GIOPlite outperformed GIOP by a small margin. For IIOP, GIOPlite performance increases over GIOP ranged from 0.36% to 4.74%, with an average performance increase of 2.74%. GIOPlite performance improvements were slightly better over UIOP due to the fact that UIOP is more efficient than IIOP. GIOPlite over UIOP provided improvements ranging from 0.37% to 5.29%, with an average of 3.26%.

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

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. The ORB's logical layers, or components, are shown in Figure 17 along with the timeprobe locations used for these benchmarks.

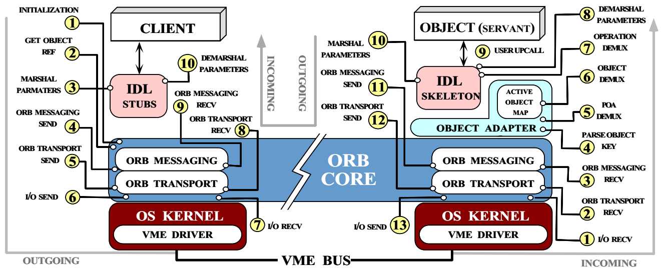


Figure 17: Timeprobe Locations for Whitebox Experiment

5.3.1 Measurement Techniques

One way to measure performance overhead of operations in complex CORBA middleware is to use a profiling tool, such as Quantify [33]. Quantify instruments an application's binary instructions and then analyzes performance bottlenecks by identifying sections of code that dominate execution time. Quantify is useful because it can measure the overhead of system calls and third-party libraries without requiring the source code.

Unfortunately, Quantify is not available for Linux kernel-based operating systems on which whitebox measurement of TAO's performance was performed. Moreover, Quantify modifies the binary code to collect timing information. It is most useful, therefore, to measure the *relative* overhead of different operations in a system, rather than measuring *absolute* run-time performance.

To avoid the limitations of Quantify, we therefore used a lightweight timeprobe mechanism provided by ACE to precisely pinpoint the amount of time spent in various ORB components and layers. The ACE 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, ACE'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 ACE 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 measure communication delays end-to-end accurately across distributed CPUs.

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

5.3.2 Whitebox Results

Figure 17 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. The results presented in the tables and figures that follow were averaged over 1,000 samples.

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:

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, which 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

Direction	Client Activities	Absolute Time (μs)
Outgoing	1. *Initialization*	6.30
	2. *Get object reference*	15.6
	3. *Parameter marshal*	0.74 (param. dependent)
	4. *ORB messaging send*	7.78
	5. *ORB transport send*	1.02
	6. *I/O*	8.70 (op. dependent)
	7. *ORB transport recv*	50.7
	8. *ORB messaging recv*	9.25
	9. *Parameter demarshal*	op. dependent

Table 1: μseconds Spent in Each Client Processing Step

connections are reused for repeated calls. For statically configured systems, such as avionics mission computing, TAO pre-establishes connections, so 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 messaging layer then passes the message to the ORB transport component for transmission to the server. If the request is a synchronous two-way operation, 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 [7], 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* operation represents the time the client spends in the receive system call. This time is generally dominated by the cost of copying data from the kernel to user supplied buffers.

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 and body to the ORB messaging component. If the message header is valid, then the remainder of the message is read. This also includes time when the client is blocked waiting for the server to read the supplied data.

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.

Direction	Server Activities	Absolute Time (μs)
Incoming	1. *I/O*	7.0 (op. dependent)
	2. *ORB transport recv*	24.8
	3. *ORB messaging recv*	4.5
	4. *Parsing object key*	4.6
	5. *POA demux*	1.39
	6. *Servant demux*	4.6
	7. *Operation demux*	4.52
	8. *User upcall*	3.84 (op. dependent)
Outgoing	9. *ORB messaging send*	4.56
	10. *ORB transport send*	93.6

Table 2: μseconds Spent in Each Server Processing Step

Each incoming server step is outlined below:

1. The *I/O* operation represents the time the server spends in the `read` system call.

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 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, *i.e.*, *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 [7].

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 [7], 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 [7] to locate the appropriate operation.

8. In the *parameter demarshal* step, the incoming request `in` and `inout` parameters are demarshaled. As with the client, the overhead of this step depends 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. Also included in the category is the time the server is blocked in the `send` operation while the client runs.

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

Depending on the type and number of operation parameters, the *ORB transport recv* step often requires the most ORB processing time. This time is dominated by the required data copies. These costs can be reduced significantly by using a transport adapter that implements a shared buffer strategy.

Component costs: Figure 18 compares the relative overhead attributable to the ORB messaging component, transport adaptor, ORB and OS for two-way `IDL_Cubit` calls to the `cube_void` operation for each possible protocol combination. This figure shows that when using IIOP the I/O and OS overhead accounts for just over 50% of the total round trip latency. It also shows that the difference in performance between IIOP and UIOP is due primarily to the larger OS and I/O overhead of TCP/IP, compared with local IPC.

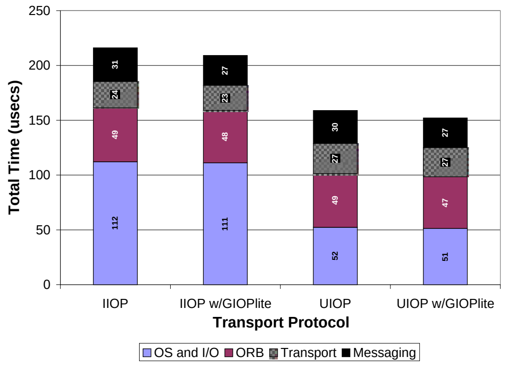


Figure 18: Comparison of ORB and Transport/OS Overhead Using Timeprobes

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 `sequence`s vary in length. Moreover, TAO's (de)marshaling optimizations [8] incur minimal overhead when running between homogeneous ORB endsystems.

In Figure 19, the parameter size is varied and the above test is repeated. It shows that as the size of the operation parameters increases, I/O overhead grows faster than the overall ORB overhead, including messaging and transport. This result illustrates that the overall 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.

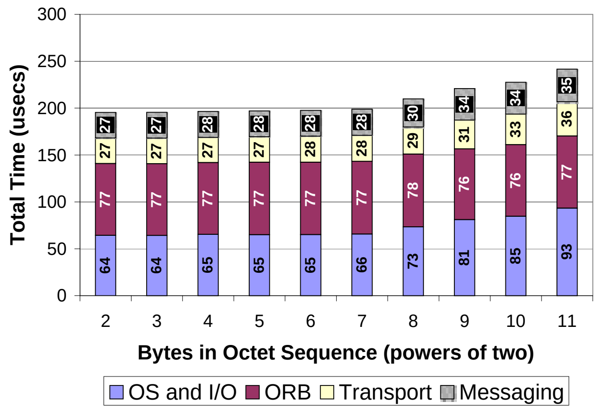


Figure 19: ORB and Transport/OS Overhead vs. Parameter Size

TAO employs standard buffer size and data copy tradeoff optimizations. This optimization is demonstrated in Figure 19 by the fact that there is a slight increase in the time spent both in the transport component and in the ORB itself when the sequence size is greater than 256 bytes. The data copy tradeoff optimization is fully configurable via run-time command line options, so it is possible to configure TAO to further improve performance above the 256 byte data copy threshold.

For the operations tested in the `IDL_Cubit` benchmark, the overhead of the ORB is dominated by memory bandwidth limitations. Both the loopback driver and local IPC driver copy data within the same host. Therefore, memory bandwidth limitations should essentially be the same for both IIOP and UIOP. This result is illustrated in Figure 18 by the fact that the time spent in the ORB is generally constant for the four protocol combinations shown.

In general, the use of UIOP demonstrates the advantages of TAO's pluggable protocols framework and how optimized, domain-specific protocols can be deployed.

6 Related Work

We have used TAO to research many dimensions of high-performance and real-time ORB endsystems, including static [2] and dynamic [25] scheduling, request demultiplexing [7], dispatching [34], and event processing [22], ORB Core connection [32] and concurrency architectures [35], IDL compiler stub/skeleton optimizations for synchronous [8] and asynchronous [36] communication, I/O subsystem integration [20], evaluation Real-time CORBA [11] features [19], fault tolerance features [14, 15], reflective QoS techniques the CORBA Component Model [37], multimedia streaming support [21], systematic benchmarking of multiple ORBs [38], and patterns for ORB extensibility [28] and optimization [7]. 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 that research and compares it with our work.

Configurable communication frameworks: The *x*-kernel [39], Conduit+ [30], System V STREAMS [40], ADAPTIVE [41], and F-CSS [42] 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 CORBA 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 [28, 30]. In particular, Conduit+ [30] 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 [17]. Likewise, the concurrency, connection management, and demultiplexing components in TAO's ORB Core and Object Adapter also have been explicitly designed using patterns such as Reactor, Acceptor-Connector, and Active Object [3].

CORBA pluggable protocols frameworks: The architecture of TAO's pluggable protocols framework is inspired by the ORBacus [43] Open Communications Interface (OCI) [44]. 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 protocols 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 it possible to write pluggable protocols that are portable among different ORB implementations and platforms.

However, using IDL also limits the the degree to which various optimizations can be applied at the ORB and transport protocol levels. For example, efficiently handling locality constrained objects, optimizing profile handling, strategized buffer allocation, or interfacing with optimized OS abstraction layers/libraries are not generally supported by existing IDL compilers. Additionally, changes to an IDL compiler's mapping rules on a per protocol basis is prohibitive.

In our approach we use C++ classes and optimized framework interfaces to allow protocol developers to exploit new strategies or available libraries. TAO uses the ACE framework [29] to isolate itself from non-portable aspects of underlying operating systems. This design leverages the testing, optimizations, implemented by ACE, enabling us to focus on the particular problems of developing a high-performance, real-time ORB.

Our framework allows each protocol implementation to represent a profile as it sees fit. Since these profiles are only created in a few instances, it is possible for them to parse the octet stream representation and store it in a more convenient format. The parsing can be also done on demand to minimize startup time. The protocol implementor is free to choose the strategy that best fits the application.

TAO implements a highly optimized pluggable protocols 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 [23, 45, 20, 9], embedded systems [8], or high-performance communication infrastructures like Fast Messages [18].

7 Concluding Remarks

To be an effective development platform for performance-sensitive applications, CORBA middleware must preserve end-to-end application QoS properties across the communication layer. 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 extend their communication infrastructure transparently 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 illustrates the performance of TAO's pluggable protocols framework empirically when running CORBA applications over high-speed interconnects, such as VME. Our benchmarking results demonstrate that applying appropriate optimizations and patterns to CORBA middleware can yield highly efficient and predictable implementations, without sacrificing flexibility or reuse. These results support our contention that CORBA middleware performance is largely an implementation issue. Thus, well-tuned, standard-based CORBA middleware like TAO can replace *ad hoc* and proprietary solutions that are still commonly used in traditional distributed applications and 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.

References

[1] M. Henning and S. Vinoski, *Advanced CORBA Programming With C++*. Reading, Massachusetts: Addison-Wesley, 1999.

[2] 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.

[3] 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.

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

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

[6] 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.

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

[8] 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*, vol. 17, Sept. 1999.

[9] 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.

[10] Compaq, Intel, and Microsoft, "Virtual Interface Architecture, Version 1.0." www.viarch.org, 1997.

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

[12] F. Kon and R. H. Campbell, "Supporting Automatic Configuration of Component-Based Distributed Systems," in *Proceedings of the* 5^{th} *Conference on Object-Oriented Technologies and Systems*, (San Diego, CA), pp. 175–178, USENIX, May 1999.

[13] Object Management Group, *The Common Object Request Broker: Architecture and Specification*, 2.3 ed., June 1999.

[14] B. Natarajan, A. Gokhale, D. C. Schmidt, and S. Yajnik, "DOORS: Towards High-performance Fault-Tolerant CORBA," in *Proceedings of the* 2^{nd} *International Symposium on Distributed Objects and Applications (DOA 2000)*, (Antwerp, Belgium), OMG, Sept. 2000.

[15] B. Natarajan, A. Gokhale, D. C. Schmidt, and S. Yajnik, "Applying Patterns to Improve the Performance of Fault-Tolerant CORBA," in *Proceedings of the* 7^{th} *International Conference on High Performance Computing (HiPC 2000)*, (Bangalore, India), ACM/IEEE, Dec. 2000.

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

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

[18] 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.

[19] C. O'Ryan, D. C. Schmidt, F. Kuhns, M. Spivak, J. Parsons, I. Pyarali, and D. Levine, "Evaluating Policies and Mechanisms for Supporting Embedded, Real-Time Applications with CORBA 3.0," in *Proceedings of the* 6^{th} *IEEE Real-Time Technology and Applications Symposium*, (Washington DC), IEEE, May 2000.

[20] F. Kuhns, D. C. Schmidt, C. O'Ryan, and D. Levine, "Supporting High-performance I/O in QoS-enabled ORB Middleware," *Cluster Computing: the Journal on Networks, Software, and Applications*, vol. 3, no. 3, 2000.

[21] 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.

[22] 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), pp. 184–199, ACM, October 1997.

[23] 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), pp. 179–187, IEEE, April 1997.

[24] H. Schulzrinne, S. Casner, R. Frederick, and V. Jacobson, "Rtp: A transport protocol for real-time applications," *Network Information Center RFC 1889*, January 1996.

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

[26] C. O'Ryan and D. C. Schmidt, "Applying a Real-time CORBA Event Service to Large-scale Distributed Interactive Simulation," in 5^{th} *International Workshop on Object-oriented Real-Time Dependable Systems*, (Monterey, CA), IEEE, Nov 1999.

[27] B. Meyer, *Object-Oriented Software Construction, Second Edition*. Englewood Cliffs, NJ: Prentice Hall, 1997.

[28] D. C. Schmidt and C. Cleeland, "Applying a Pattern Language to Develop Extensible ORB Middleware," in *Design Patterns in Communications* (L. Rising, ed.), Cambridge University Press, 2000.

[29] D. C. Schmidt, "Applying Design Patterns and Frameworks to Develop Object-Oriented Communication Software," in *Handbook of Programming Languages* (P. Salus, ed.), MacMillan Computer Publishing, 1997.

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

[31] 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, Massachusetts: Addison-Wesley, 1997.

[32] 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, special issue on Real-time Computing in the Age of the Web and the Internet*, vol. 21, no. 2, 2001.

[33] P. S. Inc., *Quantify User's Guide*. PureAtria Software Inc., 1996.

[34] I. Pyarali, C. O'Ryan, and D. C. Schmidt, "A Pattern Language for Efficient, Predictable, Scalable, and Flexible Dispatching Mechanisms for Distributed Object Computing Middleware," in *Proceedings of the International Symposium on Object-Oriented Real-time Distributed Computing (ISORC)*, (Newport Beach, CA), IEEE/IFIP, Mar. 2000.

[35] D. C. Schmidt, "Evaluating Architectures for Multi-threaded CORBA Object Request Brokers," *Communications of the ACM special issue on CORBA*, vol. 41, Oct. 1998.

[36] A. B. Arulanthu, C. O'Ryan, D. C. Schmidt, M. Kircher, and J. Parsons, "The Design and Performance of a Scalable ORB Architecture for CORBA Asynchronous Messaging," in *Proceedings of the Middleware 2000 Conference*, ACM/IFIP, Apr. 2000.

[37] N. Wang, D. C. Schmidt, K. Parameswaran, and M. Kircher, "Applying Reflective Middleware Techniques to Optimize a QoS-enabled CORBA Component Model Implementation," in *24th Computer Software and Applications Conference*, (Taipei, Taiwan), IEEE, Oct. 2000.

[38] 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.

[39] 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.

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

[41] 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.

[42] 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.

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

[44] I. Object-Oriented Concepts, "ORBacus User Manual - Version 3.1.2." www.ooc.com/ob, 1999.

[45] 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.