

The Design and Performance of a Pluggable Protocols Framework for Object Request Broker Middleware

Fred Kuhns, Carlos O’Ryan, Douglas C. Schmidt, Ossama Othman, and Jeff Parsons

{fredk,coryan,schmidt,othman,parsons}@cs.wustl.edu

Department of Computer Science, Washington University

St. Louis, MO 63130, USA *

This paper appeared in the proceedings of the IFIP 6th International Workshop on Protocols For High-Speed Networks (PFHSN ’99), August 25–27, 1999, Salem, MA.

Abstract

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

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

Our results demonstrate how applying optimizations to communication middleware can yield highly flexible/reusable designs and highly efficient/predictable implementations. In particular, the overall round-trip 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 ~ 125 μ secs. The ORB middleware accounts for approximately 48% or ~ 60 μ secs of the total round-trip latency. These results illustrate that (1) communication middleware performance is largely

an implementation detail and (2) the next-generation of optimized, standards-based CORBA middleware can replace ad hoc and proprietary solutions.

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

1 Introduction

Current trends and limitations: During the past decade, there has been substantial R&D emphasis on *high-speed networking* and *performance optimizations* for network elements and protocols. As a result, networks are now available off-the-shelf that can support Gbps on every port, e.g., Gigabit Ethernet and ATM switches. Moreover, 622 Mbps ATM connectivity in WAN backbones is becoming commonplace. In networks and GigaPoPs, such as the Advanced Technology Demonstration Network (ATDnet) [1], 2.4 Gbps (OC-48) link speeds are being deployed. However, the general lack of robust and flexible *communication middleware* for programming, provisioning, and controlling these networks has limited the rate at which applications have been developed to leverage advances in high-speed networking.

Communication middleware resides between client and server applications in distributed systems. It simplifies application development by providing a uniform view of heterogeneous networks, protocols, and OS layers. At the heart of communication middleware are *Object Request Brokers* (ORBs), such as CORBA [2], DCOM [3], and Java RMI [4], which eliminate many of tedious, error-prone, and non-portable aspects of developing and maintaining distributed applications programmed using low-level mechanisms like sockets. In particular, ORBs automate common network programming tasks, such as object location, object activation, parameter (de)marshaling, socket and request demultiplexing, fault recovery, and security.

During the past decade there has also been substantial R&D emphasis on communication middleware. As a result, communication middleware is now available off-the-shelf that al-

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

lows clients to invoke operations on distributed components without concern for component location, programming language, OS platform, communication protocols and interconnects, or hardware [5]. However, the general lack of support in this off-the-shelf communication middleware for QoS specification and enforcement features, integration with high-speed networking technology, and performance, predictability, and scalability optimizations [6], has limited the rate at which applications have been developed to leverage advances in communication middleware.

Overcoming communication middleware limitations with pluggable protocols: To address the shortcomings of communication middleware described above, we have developed *The ACE ORB* (TAO) [6]. TAO is open-source,¹ standards-based, high-performance, real-time ORB endsystem communication middleware that supports applications with deterministic and statistical QoS requirements, as well as “best-effort” requirements. TAO is the first ORB to support end-to-end QoS guarantees over ATM/IP networks [7, 8].

We have used TAO to research many dimensions of high-performance and real-time ORB endsystems, including static [6] and dynamic [9] scheduling, request demultiplexing [10], event processing [11], ORB Core connection and concurrency architectures [12], IDL compiler stub/skeleton optimizations [13], systematic benchmarking of multiple ORBs [14], I/O subsystem integration [8], and patterns for ORB extensibility [15]. This paper focuses on a previously unexamined dimension in the high-performance and real-time ORB endsystem design space: *the design and performance of a pluggable protocols framework* that supports high-speed protocols and networks, real-time embedded system interconnects, and standard TCP/IP protocols over the Internet.

Paper organization: The remainder of this paper is organized as follows: Section 2 outlines the CORBA protocol interoperability architecture; Section 3 motivates the need for a CORBA pluggable protocols framework and describes how TAO’s pluggable protocols framework is designed; Section 4 illustrates the performance characteristics of TAO’s pluggable protocols framework; Section 5 compares TAO with related work; and Section 6 presents concluding remarks.

2 Overview of the CORBA Protocol Interoperability Architecture

The CORBA specification [2] defines an architecture for ORB interoperability. Although a complete description of the model is beyond the scope of this paper, this section outlines the

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

portions of the CORBA specification that are relevant to our present topic, *i.e.*, object addressing and inter-ORB protocols.

Object addressing synopsis: To identify objects, CORBA defines a generic format called the Interoperable Object Reference (IOR). An object reference identifies one instance of an object and associates one or more paths or routes 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 enhance fault-tolerance.

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

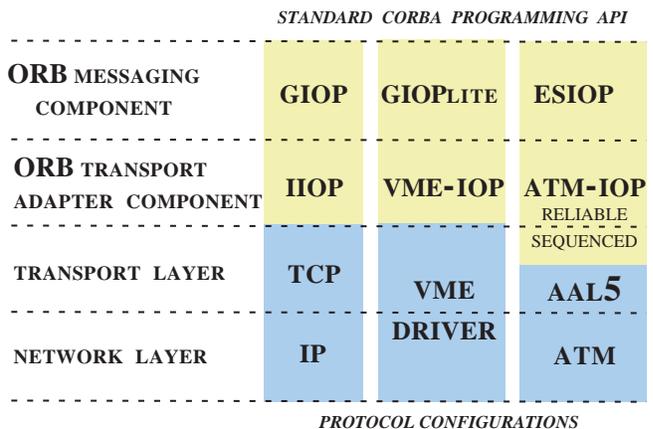


Figure 1: Relationship Between Inter-ORB Protocols and Transport-specific Mappings

In the CORBA protocol interoperability architecture, the standard *General Inter-ORB Protocol* (GIOP) is defined by the CORBA specification [2]. In addition, CORBA defines a TCP/IP mapping of GIOP, which is called the *Internet Inter-ORB Protocol* (I-IOP). ORBs must support I-IOP 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. Below, we outline how GIOP defines each of these IOP elements.

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

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

- **GIOP message formats:** The GIOP specification defines seven types of messages that send requests, receive replies, locate objects, and manage communication channels. The following table lists the seven types of messages in GIOP 1.0² 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. There may be multiple paths or routes to an object. Therefore, the same IOR can contain multiple IIOP profiles, along with profiles for other protocols, such as GIOP over ATM or non-GIOP protocols.

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

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) [2]. Figure 1 illustrates two ESIOPs we are developing, GIOPlite and an ATM ESIOP. In addition, the OMG is considering other protocols for domains such as wireless and mobile systems [16], which have unique performance characteristics and optimization points.

3 A Pluggable Protocols Framework for CORBA

The CORBA specification provides a standard for general-purpose communication middleware. Within the scope of this specification, however, ORB implementors are free to optimize internal data structures and algorithms [10]. Moreover, ORBs may use specialized inter-ORB protocols and ORB services and still comply with the CORBA specification.³ This section identifies the limitations of current ORBs with respect to their protocol support, describes how TAO's pluggable protocols framework is designed to overcome these limitations, and then describes how TAO can be applied to develop middleware for high-performance multimedia applications.

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

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

Therefore, applications with more stringent QoS requirements need optimized protocol implementations, QoS-aware interfaces, custom presentations layers, specialized memory management (*e.g.*, shared memory between ORB and I/O subsystem), and alternative transport programming APIs (*e.g.*, sockets vs. TLI). Domains where highly optimized ORB messaging and transport protocols are 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:** Many ORBs support a limited number of statically configured protocols, typically only GIOP/IIOP over TCP/IP.
- 2. Lack of protocol control interfaces:** Many 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:** Many ORBs do not support simultaneous use of multiple inter-ORB messaging or transport protocols.
- 4. Lack of real-time protocol support:** Many ORBs have limited or no support for specifying and enforcing real-time protocol requirements 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 of such a pluggable protocols framework for CORBA include the following:

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

2. Use standard CORBA programming and control interfaces: To ensure application portability, clients should program to standard application interfaces defined in CORBA IDL, even if pluggable ORB messaging or transport protocols are used. Likewise, object implementors need not be aware of the underlying framework. However, developers should be able to set policies that control the ORB's choice of 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 [10].

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

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 scheme. Therefore, a pluggable protocols framework should provide a general mechanism to represent these disparate address formats transparently, while also supporting standard IOR address representations efficiently.

5. Support CORBA 2.2 features and future enhancements: A pluggable protocol framework should support CORBA 2.2 features, such as object reference forwarding, connection transparency, preservation of foreign IORs and profiles, and the complete GIOP 1.1 protocol, in a manner that does not degrade end-to-end performance and predictability. Moreover, a pluggable protocols framework should accommodate future changes and enhancements to the CORBA specification, such as (1) the GIOP 1.2 protocol, which allows bi-directional requests over the same connection, (2) real-time CORBA [18], which includes features to reserve connection and threading resources on a per-object basis, and (3) asynchronous messaging [19], which exports QoS policies to application developers.

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

7. Provide common protocol optimizations and real-time features: A pluggable protocols framework should support

features required by real-time CORBA applications [18], such as resource pre-allocation and reservation, end-to-end priority propagation, and mechanisms to control properties specific to real-time protocols. These features should be implemented without modifying the standard CORBA programming APIs used by conventional applications that do not possess real-time QoS requirements.

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

3.3 Architectural Overview of TAO's Pluggable Protocols Framework

To overcome the limitations described in Section 3.1, 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

pluggable protocols framework can be entirely transparent to CORBA application developers.

Figure 2 also illustrates the key components in TAO's pluggable protocols framework: (1) the ORB messaging component, (2) the ORB transport adapter component, and (3) the ORB policy control component, which are outlined 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. As described in Section 2, ORB messaging protocols should define a data representation, an ORB message format, an ORB transport protocol or transport adapter, and an object addressing format. Within this framework, ORB protocol developers are free to implement optimized Inter-ORB protocols and enhanced transport adapters, as long as the ORB interfaces are respected.

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.

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 proto-

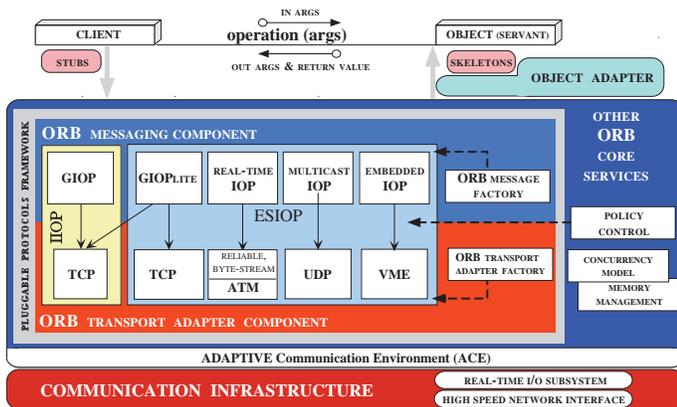


Figure 2: TAO's Pluggable Protocols Framework Architecture

essential to resolve several limitations of conventional ORBs outlined in Section 3.1.

In general, the higher-level components and services of TAO use a facade [20] interface to access the mechanisms provided by its pluggable protocols framework. Thus, applications can (re)configure custom protocols without requiring global changes to the ORB. Moreover, because applications typically access only the standard CORBA APIs, TAO's

col over ATM, and (5) tight coupling between the ORB and efficient user-space protocol implementations, such as Fast Messages [21].

3.3.3 ORB Policy Control Component

This component allows applications to control the QoS attributes of configured ORB transport protocols explicitly. 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 [19] and Real-time CORBA [18] specifications.

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 [18]. Using this policy, an application can indicate its preferred protocol(s) and TAO’s policy control component attempts to match that preference with the set of available protocols. TAO provides other policies that controls the behavior of the ORB if an application’s preferences cannot be satisfied. For example, either an exception can be raised or another available protocol can be selected transparently.

3.4 A Pluggable Protocol Scenario

To illustrate how TAO’s pluggable protocols framework can be applied in practice, we now describe a scenario where pluggable protocols can be used to support performance-sensitive

and real-time CORBA applications. This scenario is based on our experience developing high-bandwidth, low-latency audio/video streaming applications [22] and avionics mission computing [11] systems. In previous work [8], we addressed the network interface and I/O system and how to achieve predictable, real-time performance. In the discussion below, we focus on ORB support for alternate protocols.

Low-latency, high-bandwidth multimedia streaming: Multimedia applications running over high-speed networks require optimizations to utilize available link bandwidth, while still meeting application deadlines. For example, consider Figure 3, where network interfaces supporting 1.2

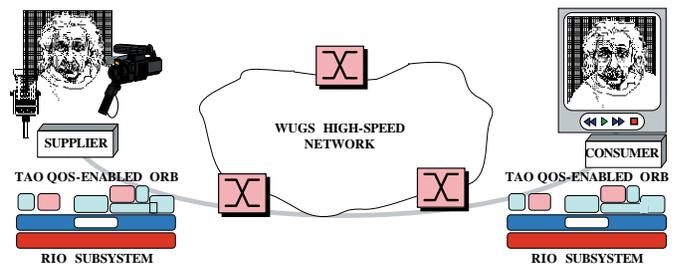


Figure 3: Example CORBA-based Audio/Video (A/V) Application

Mbps or 2.4 Mbps link speeds are used for a CORBA-based studio quality audio/video (A/V) application.

In this example, we can 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 [8, 23]. At the core of RIO is the high-speed ATM port interconnect controller (APIC) [24]. 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 are leveraging 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:

- 1. 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 in order to determine both the set of optimizations required and the best par-

tioning of the solution into ORB messaging, transport and policy components.

The A/V streaming application is primarily concerned with pushing data to clients (i.e., one-way method invocations) and with 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, such as TCP, is not required. Thus, the overhead of full GIOP is not required, 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) [25].

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 for ORB endsystem resources.

2. 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 [24, 10]. 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 [8].

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 re-

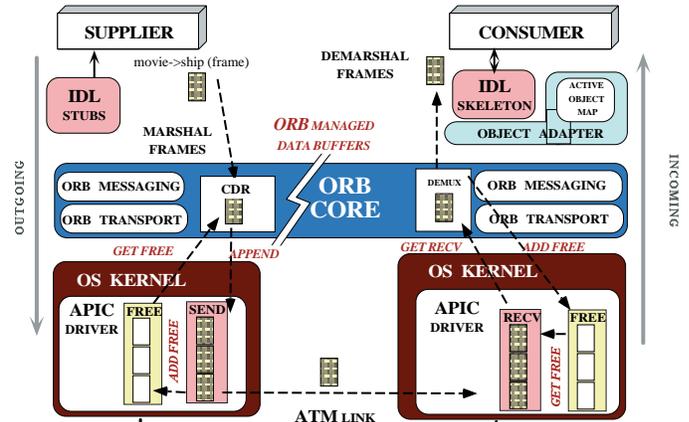


Figure 4: Shared Buffer Strategy

duce the use of resource locks. For example, in the scheme depicted in Figure 4, each active connection is assigned its 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 scheme is described in [24]

The following two approaches are ways that the buffering scheme described above can interact with user applications:

1. *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.
2. *Single-copy* – The ORB copies application data into the ORB managed buffers. While this scheme 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 protocol policies, this decision can be configured on a per-connection, per-thread, per-object or per-operation basis.

4 The Performance of TAO’s Pluggable Protocols Framework

Despite the growing demand for off-the-shelf middleware in many application domains, a widespread belief persists in the embedded systems community that OO techniques are not suitable for real-time systems due to performance penalties attributed to the OO paradigm [11]. 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 it is possible to implement very predictable, efficient, and scalable middleware *without* compromising non-functional requirements, such as portability, flexibility, reusability, and maintainability, offered by CORBA.

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 communication middleware for avionics mission computing applications [11] and multimedia applications [22].

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 in high-performance protocol development and experimentation.

4.1 Hardware/Software Benchmarking Platform

All benchmarks in this section were run on a Quad-CPU Intel Pentium II Xeon 400 MHz workstation, with one gigabyte 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.

Our benchmarks were run using the standard GIOP ORB messaging protocol, as well as TAO’s GIOPLite messaging protocol, described in Section 3.4. For the TCP/IP tests, the GIOP and GIOPLite ORB messaging protocols were run using the standard CORBA IIOp transport adapter along with the Linux TCP/IP socket library and the loopback interface. For the local IPC tests, GIOP and GIOPLite were used along with the optimized local IPC transport adapter. This resulted

in four different Inter-ORB Protocols: GIOP over TCP (IIOp), GIOPLite over TCP, GIOP over local IPC (UIOP⁴) and GIOPLite over local IPC. No changes were required to our standard CORBA benchmarking tool, called IDL_Cubit [12], for either of the ORB messaging and transport protocol implementations.

4.2 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.

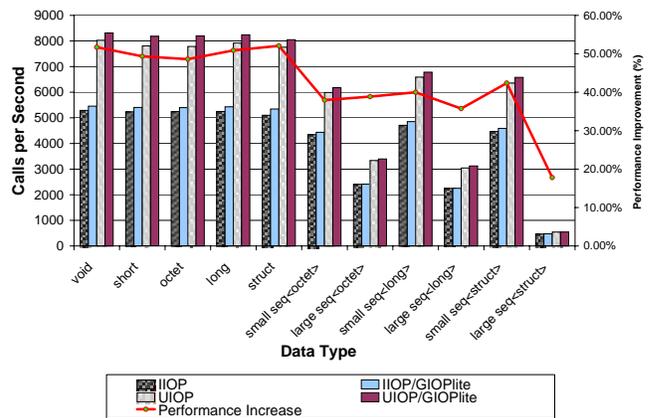


Figure 5: TAO’s Pluggable Protocols Framework Performance Over Local IPC and TCP/IP

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 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

⁴For historical reasons, TAO retains the expression “UNIX-domain” in its local IPC pluggable protocol implementation, which is where the name “UIOP” derives from.

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 `longs` or 4,096 `octets`) and the short sequences are 4 bytes (one four byte `long` or four `octets`).

Blackbox results: The blackbox benchmark results are shown in Figure 5. All blackbox benchmarks were averaged over 100,000 two-way operation calls for each data type, as depicted by the bars in Figure 5.

UIOP performance surpassed IIOP performance for all data types. The benchmarks show 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%. However, for larger data payload sizes and more complex data types, the performance improvements are reduced. This is a direct result of the increasing cost of both the data copies associated with performing I/O and 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 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.

4.3 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 logical layers, or components, are shown

in Figure 6 along with the timeprobe locations used for these benchmarks.

4.3.1 Measurement Techniques

One way to measure performance overhead of operations in complex communication middleware is to use a profiling tool like Quantify [26]. 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 source code access.

Unfortunately, Quantify is not available for Linux kernel-based operating systems for which whitebox measurement of TAO's performance is needed. Moreover, Quantify modifies the binary code to collect timing information. Therefore, it is most useful for measuring *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 TAO to precisely pinpoint the amount of time spent in various ORB components and layers. The TAO timeprobe mechanism provides highly accurate, low-cost timestamps that record the time spent between regions of code in a software system. These timeprobes have minimal performance impact, *e.g.*, 1-2 μ sec overhead per timeprobe, and no binary code instrumentation is required.

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

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

4.3.2 Whitebox Results

Figure 6 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 which follow were averaged over 1,000 samples.

Client performance: Table 1 depicts the time in microseconds (μ s) spent in each sequential activity that a TAO client

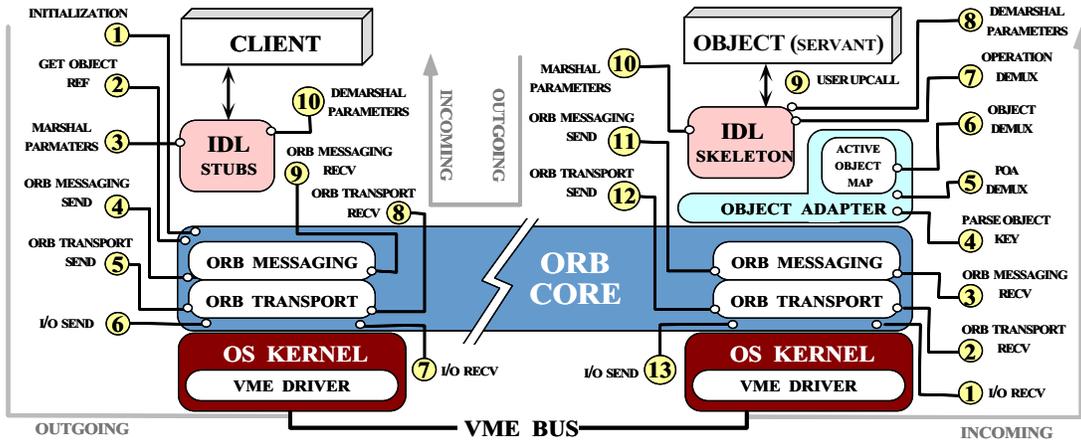


Figure 6: Timeprobe Locations for Whitebox Experiment

performs to process an outgoing operation request and its reply.

Table 1: μ seconds Spent in Each Client Processing Step

Direction	Client Activities	Absolute Time (μ s)
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 rcv	50.7
	8. ORB messaging rcv	9.25
	9. Parameter demarshal	op. dependent

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 that handle marshaling and demarshaling of operation parameters.
2. TAO's connector caches connections, so even though its `connect` method is called for every operation, existing connections are reused for repeated calls. For statically configured systems, such as avionics mission computing, TAO 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

two-way, the transport component waits for and processes the response.

5. The `send` operation in the *ORB transport* component implements the connection concurrency strategy and invokes the appropriate ACE I/O operation. TAO maintains a linked list of CDR buffers [10], which allows it to use "gather-write" OS calls, such as `writnev`. 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.

Table 2: μ seconds Spent in Each Server Processing Step

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

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, then the remaining data is read and passed to the ORB messaging component.

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

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

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

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 [10], 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 [10] to locate the appropriate operation.

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

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

Each outgoing server step is outlined below:

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

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

12. The *send* operation in the *ORB transport* layer adds the appropriate IOP header to the reply, sends the reply, and closes the connection if it detects an error. 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. As in the client-side *I/O send* operation described above, the server uses a gather-write I/O call.

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

Component costs: Figure 7 compares the relative over-

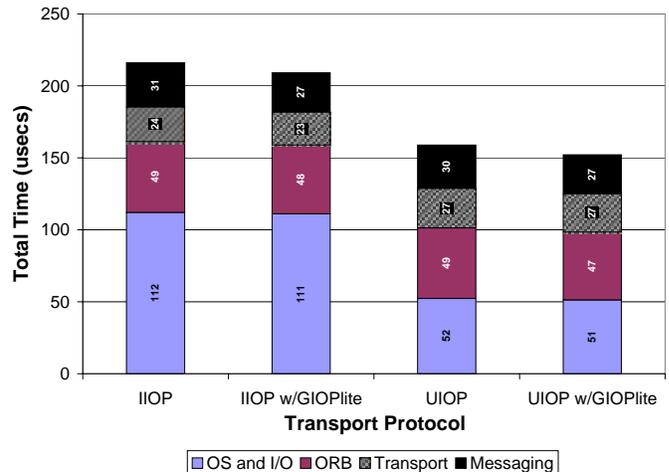


Figure 7: Comparison of ORB and Transport/OS Overhead Using Timeprobes

head 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 primarily due to the larger OS and I/O overhead that TCP/IP has, as compared to local IPC.

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

In Figure 8, the parameter size is varied and the above test is repeated. It shows that as the size of the operation param-

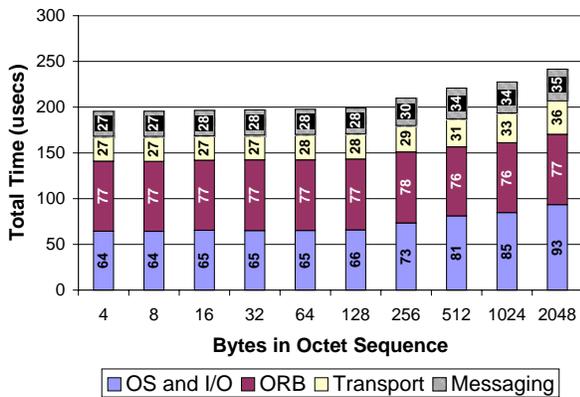


Figure 8: ORB and Transport/OS Overhead Versus Parameter Size. As the parameter size increases, I/O overhead grows faster than the overall ORB overhead (including messaging and transport). This result illustrates that the overall ORB overhead is largely independent of the request size. In particular, demultiplexing a request, creating message headers, and invoking an operation upcall are not affected by the size of the request.

TAO employs standard buffer size and data copy tradeoff optimizations. This optimization is demonstrated in Figure 8 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 7 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 this framework and how optimized, domain-specific protocols can be deployed.

5 Related Work

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

Configurable communication frameworks: The *x*-kernel [27], Conduit+ [28], System V STREAMS [29], ADAPTIVE [30], and F-CSS [31] 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 communication middleware API, *i.e.*, the standard CORBA programming API. Therefore, existing communication subsystem frameworks can provide building block protocol components for TAO’s pluggable protocols framework.

Patterns-based communication frameworks: An increasing number of communication frameworks are being designed and documented using patterns [15, 28]. In particular, Conduit+ [28] 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 [20]. Likewise, the concurrency, connection management, and demultiplexing components in TAO’s ORB Core and Object Adapter also have been explicitly designed using patterns like Reactor, Acceptor-Connector, and Active Object [15].

CORBA pluggable protocol frameworks: The architecture of TAO’s pluggable protocols framework is based on the ORBacus Open Communications Interface (OCI) [32]. The OCI framework provides a flexible, intuitive, and portable interface for pluggable protocols. The framework interfaces are defined in IDL, with a few special rules to map critical types, such as data buffers.

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

Though the OCI pluggable protocols framework is useful for many applications and ORBs, the following aspects make it less suitable for high-performance and real-time systems:

- **IDL interfaces add extra overhead:** As mentioned above, the use of IDL has several advantages. However, unless new IDL mapping rules are approved for locality constrained objects, an ORB must set up a nontrivial amount of context information, *e.g.*, to handle POA Servant Managers [33], to make local invocations have the same semantics as remote invocations. Although overhead can be minimized by using *ad hoc* optimizations, some additional method invocation overhead will be incurred by common IDL mappings.

In contrast, the framework we propose utilizes regular C++ classes, this limits the portability of the system, but completely eliminates the overhead introduced by the IDL interfaces.

- **The current OCI version does not support zero-copy buffers:** The OCI interfaces do not currently support zero-copy I/O; which would permit the ORB to marshal data directly into kernel buffers making a single copy or at most one copy. TAO supports the use of different buffering strategies which allow protocol developers to implement schemes where memory can be shared between the application, ORB and/or I/O subsystem. By supporting different buffering strategies, the effectiveness of the framework for high-performance communication links is enhanced. For example, the transport adapter could manage a per-connection set of buffer pools. By strategizing the CDR classes' use of internal buffers, protocol implementers can focus on optimizing for specific hardware or communication channels rather than building general software components.

- **The current OCI version does not optimize profile parsing:** Parsing an IOP profile is a relatively expensive operation. The OCI framework does not provide any means to manipulate a pre-parsed profile, which is a common use-case.

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.

- **ACE and OCI interfaces require extra adaptation layers:** TAO uses the ACE framework [34] to isolate itself from non-portable aspects of underlying operating systems. This

design leverages the testing, optimizations, wide range of platforms, and the communication patterns supported and implemented by ACE, enabling us to focus on the particular problems of developing a high-performance, real-time ORB. Using the OCI IDL-derived interfaces incurs an extra layer of adaptation between ACE and TAO, which unnecessarily increases framework overhead.

TAO implements a highly optimized pluggable protocol framework that is tuned for high-performance and real-time application requirements. For example, TAO's pluggable protocols framework can be integrated with zero-copy high-speed network interfaces [24, 8].

However, TAO's pluggable protocols framework does not preclude the use of more general frameworks like the ORBacus OCI. In fact, we plan to implement OCI as a pluggable protocol into TAO, thereby allowing application developers to test and use OCI pluggable protocols. If applications have very stringent performance requirements, developers can use the internal TAO pluggable protocol framework to obtain the higher performance and greater predictability.

6 Concluding Remarks

To be an effective development platform for performance-sensitive applications, OO middleware must preserve communication layer QoS properties of applications end-to-end. It is essential, therefore, to define a pluggable protocols framework that allows custom inter-ORB messaging and transport protocols to be configured flexibly and transparently by CORBA applications.

This paper identifies the protocol-related limitations of current ORBs and describes a CORBA-based pluggable protocols framework we developed and integrated with TAO to address these limitations. TAO's pluggable protocols framework contains two main components: an ORB messaging component and an ORB transport adapter component. These two components allows applications developers and end-users to transparently extend their communication infrastructure to support the dynamic and/or static binding of new ORB messaging and transport protocols. Moreover, TAO's 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 empirically the performance of TAO's pluggable protocols framework when running CORBA applications over high-speed interconnects, such as ATM. Our benchmarking results demonstrate that applying appropriate optimizations to communication middleware can yield highly efficient and predictable implementations, without sacrificing flexibility or reuse. These results support our contention that communication middleware performance is largely an imple-

mentation issue. Thus, well-tuned, standard-based communication middleware like TAO can replace *ad hoc* and proprietary solutions that are still commonly used in traditional distributed applications and embedded real-time systems.

We are currently developing pluggable protocols for high-speed networks such as ATM and Myrinet. One focus of our future work is to determine effective patterns for supporting advanced I/O features, such as buffer management schemes using intelligent I/O interfaces and shared memory, available in current high-speed network adaptors. In addition, we are exploring the integration of high-speed messaging protocols, such as Fast Messages [21], with standard CORBA communication middleware.

References

- [1] ATD, "Advanced Technology Demonstration Network." <http://www.atd.net/>.
- [2] Object Management Group, *The Common Object Request Broker: Architecture and Specification*, 2.2 ed., Feb. 1998.
- [3] D. Box, *Essential COM*. Addison-Wesley, Reading, MA, 1997.
- [4] A. Wollrath, R. Riggs, and J. Waldo, "A Distributed Object Model for the Java System," *USENIX Computing Systems*, vol. 9, November/December 1996.
- [5] S. Vinoski, "CORBA: Integrating Diverse Applications Within Distributed Heterogeneous Environments," *IEEE Communications Magazine*, vol. 14, February 1997.
- [6] D. C. Schmidt, D. L. Levine, and S. Mungee, "The Design and Performance of Real-Time Object Request Brokers," *Computer Communications*, vol. 21, pp. 294–324, Apr. 1998.
- [7] G. Parulkar, D. C. Schmidt, and J. S. Turner, "a¹t^Pm: a Strategy for Integrating IP with ATM," in *Proceedings of the Symposium on Communications Architectures and Protocols (SIGCOMM)*, ACM, September 1995.
- [8] F. Kuhns, D. C. Schmidt, and D. L. Levine, "The Design and Performance of a Real-time I/O Subsystem," in *Proceedings of the 5th IEEE Real-Time Technology and Applications Symposium*, (Vancouver, British Columbia, Canada), IEEE, June 1999.
- [9] C. D. Gill, D. L. Levine, and D. C. Schmidt, "The Design and Performance of a Real-Time CORBA Scheduling Service," *The International Journal of Time-Critical Computing Systems, special issue on Real-Time Middleware*, 1999, to appear.
- [10] I. Pyarali, C. O’Ryan, D. C. Schmidt, N. Wang, V. Kachroo, and A. Gokhale, "Applying Optimization Patterns to the Design of Real-time ORBs," in *Proceedings of the 5th Conference on Object-Oriented Technologies and Systems*, (San Diego, CA), USENIX, May 1999.
- [11] T. H. Harrison, D. L. Levine, and D. C. Schmidt, "The Design and Performance of a Real-time CORBA Event Service," in *Proceedings of OOPSLA ’97*, (Atlanta, GA), ACM, October 1997.
- [12] D. C. Schmidt, S. Mungee, S. Flores-Gaitan, and A. Gokhale, "Software Architectures for Reducing Priority Inversion and Non-determinism in Real-time Object Request Brokers," *Journal of Real-time Systems*, To appear 1999.
- [13] 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*, Sept. 1999.
- [14] 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.
- [15] D. C. Schmidt and C. Cleland, "Applying Patterns to Develop Extensible ORB Middleware," *IEEE Communications Magazine*, vol. 37, April 1999.
- [16] Object Management Group, *Telecom Domain Task Force Request For Information Supporting Wireless Access and Mobility in CORBA - Request For Information*, OMG Document telecom/98-06-04 ed., June 1998.
- [17] R. S. Madukkarumukumana and H. V. Shah and C. Pu, "Harnessing User-Level Networking Architectures for Distributed Object Computing over High-Speed Networks," in *Proceedings of the 2nd Usenix Windows NT Symposium*, August 1998.
- [18] Object Management Group, *Realtime CORBA Joint Revised Submission*, OMG Document orbos/99-02-12 ed., March 1999.
- [19] Object Management Group, *CORBA Messaging Specification*, OMG Document orbos/98-05-05 ed., May 1998.
- [20] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, MA: Addison-Wesley, 1995.
- [21] 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.
- [22] 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.
- [23] F. Kuhns, D. C. Schmidt, and D. L. Levine, "The Design and Performance of RIO – A Real-time I/O Subsystem for ORB Endsytms," in *Proceedings of the International Symposium on Distributed Objects and Applications (DOA’99)*, (Edinburgh, Scotland), OMG, Sept. 1999.
- [24] Z. D. Dittia, G. M. Parulkar, and J. R. Cox, Jr., "The APIC Approach to High Performance Network Interface Design: Protected DMA and Other Techniques," in *Proceedings of INFOCOM ’97*, (Kobe, Japan), IEEE, April 1997.
- [25] H. Schulzrinne, S. Casner, R. Frederick, and V. Jacobson, "Rtp: A transport protocol for real-time applications," *Network Information Center RFC 1889*, January 1996.
- [26] P. S. Inc., *Quantify User’s Guide*. PureAtria Software Inc., 1996.

- [27] 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.
- [28] H. Hueni, R. Johnson, and R. Engel, "A Framework for Network Protocol Software," in *Proceedings of OOPSLA '95*, (Austin, Texas), ACM, October 1995.
- [29] D. Ritchie, "A Stream Input–Output System," *AT&T Bell Labs Technical Journal*, vol. 63, pp. 311–324, Oct. 1984.
- [30] 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.
- [31] 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.
- [32] I. Object-Oriented Concepts, "ORBacus User Manual - Version 3.1.2." www.ooc.com/ob, 1999.
- [33] M. Henning and S. Vinoski, *Advanced CORBA Programming With C++*. Addison-Wesley Longman, 1999.
- [34] D. C. Schmidt and T. Suda, "An Object-Oriented Framework for Dynamically Configuring Extensible Distributed Communication Systems," *IEE/BCS Distributed Systems Engineering Journal (Special Issue on Configurable Distributed Systems)*, vol. 2, pp. 280–293, December 1994.