The Design and Performance of a Pluggable Protocols Framework for Real-Time Distributed Object Computing Middleware*

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

Department of Computer Science, Washington University
St. Louis, MO 63130, USA
{coryan,fredk,schmidt,othman,parsons}@cs.wustl.edu

Abstract. To be an effective platform for performance-sensitive real-time and embedded 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 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 distributed object computing (DOC) 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 DOC middleware can yield both highly flexible/reusable designs and highly efficient/predictable implementations. In particular, 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 ∼189 μsecs. The ORB middleware accounts for approximately 48% or ∼90 μsecs of the total roundtrip latency. Using the specialized POSIX local IPC protocol reduces roundtrip latency to ∼125 μsecs. These results illustrate that (1) DOC 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

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

J. Sventek and G. Coulson (Eds.): Middleware 2000, LNCS 1795, pp. 372–395, 2000.

1 Introduction

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

Standard DOC 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 [4]. However, off-the-shelf DOC middleware generally lacks (1) support for QoS specification and enforcement, (2) integration with high-speed networking technology, and (3) efficiency, predictability, and scalability optimizations [5]. These omissions have limited the rate at which performance-sensitive applications, such as video-on-demand, teleconferencing, and avionics mission computing, have been developed to leverage advances in DOC middleware.

Overcoming DOC middleware limitations with pluggable protocols: To address the shortcomings of DOC middleware described above, we have developed *The ACE ORB* (TAO) [5]. TAO is open-source,[1] standards-based, high-performance, real-time ORB endsystem DOC middleware that supports applications with deterministic and statistical QoS requirements, as well as "best-effort" requirements. TAO is the first ORB to support end-to-end QoS guarantees over ATM/IP networks [6,7] and embedded backplanes [8,9].

We have used TAO to research many dimensions of high-performance and real-time ORB endsystems, including static [5] and dynamic [10] scheduling, request demultiplexing [11], event processing [8], ORB Core connection and concurrency architectures [12], IDL compiler stub/skeleton optimizations [13], systematic benchmarking of multiple ORBs [14], I/O subsystem integration [7], 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 implementation of a high-performance pluggable protocols framework for real-time communications middleware* 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 [16], which decouples TAO's ORB messaging and transport interfaces from its transport-specific protocol components. This design allows custom ORB

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

messaging and transport protocols to be configured flexibly and used transparently by CORBA applications. For example, if ORBs communicate over a high-speed networking 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 standard CORBA inter-ORB protocols like GIOP.

For OO researchers and practitioners, the results in this paper are important, because they demonstrate concretely that the ability of standards-based DOC middleware to support high-performance, real-time systems is largely an *implementation detail*, rather than an inherent liability. For instance, TAO's end-to-end latency overhead is only ∼110 μsecs using commercial off-the-self 200 MHz PowerPCs, a 320 Mbps VMEbus, and VxWorks, which is as fast, or faster, than many *ad hoc*, proprietary solutions [9]. These results motivate the use of well-tuned, standards-based DOC middleware, even for real-time embedded applications with very stringent QoS requirements. The paper also explores how patterns can be applied to resolve key design challenges. Our pattern-oriented OO design can also be extended to other pluggable protocol frameworks, either in standard middleware or in distributed applications using proprietary middleware.

Paper organization: The remainder of this paper is organized as follows: Section 2 motivates the requirements for standard CORBA pluggable protocols and outlines TAO's pluggable protocols framework; Section 3 describes the patterns that guide the architecture of TAO's pluggable protocols framework and resolve key design challenges. 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 The Design of a CORBA Pluggable Protocols Framework

The CORBA specification provides a standard for general-purpose DOC middleware. Within the scope of this specification, however, ORB implementors are free to optimize internal data structures and algorithms [11]. Moreover, ORBs may use specialized inter-ORB protocols and ORB services and still comply with the CORBA specification.[2] This section identifies contemporary limitations of and requirements for protocol support in CORBA ORBs and describes how TAO's pluggable protocols framework is designed to overcome these limitations.

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

2.1 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.

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 [18]). 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, typically only 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.

2.2 Pluggable Protocols Framework Requirements

The limitations of conventional ORBs described in Section 2.1 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 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 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. 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 [11].

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 [19], or dynamically, *i.e.*, during ORB run-time [20].

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 2.3 features and future enhancements: A pluggable protocol framework should support CORBA 2.3 [21] 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 future changes and enhancements to the CORBA specification, such as (1) fault tolerance [22], which supports group communication, (2) real-time CORBA [19], which includes features to reserve connection and threading resources on a per-object basis, (3) asynchronous messaging [23], which exports QoS policies to application developers, and (4) wireless access and mobility [24], 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 [19], 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 limitation of static 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.

2.3 Architectural Overview of TAO's Pluggable Protocols Framework

To meet the requirements outlined in Section 2.2, 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 1. This decoupling

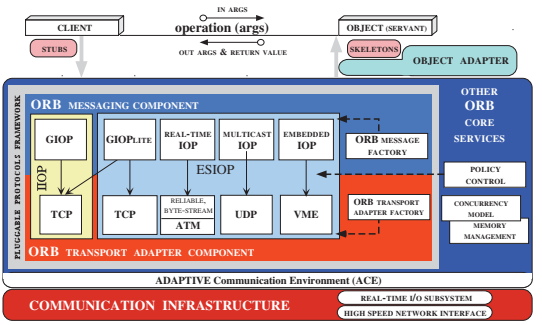

Fig. 1. TAO's Pluggable Protocols Framework Architecture

is essential to resolve several limitations of conventional ORBs outlined in Section 2.1, as well as to meet the requirements set forth in Section 2.2.

In general, the higher-level components and services of TAO use a facade [25] 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 pluggable protocols framework can be entirely transparent to CORBA application developers.

Figure 1 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.

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 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.

In general it is not necessary to re-implement all aspects of an ORB messaging protocol. For example, TAO has a highly optimized CDR implementation that can be used by new protocols [11]. 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.

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

ORB Transport Adapter Component This component maps a specific ORB messaging protocol, such as GIOP or DCE-CIOP, onto a specific instance of an underlying transport protocol, such as TCP or ATM. Figure 1 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 [26].

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 [23] and Real-time CORBA [19] 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 [19]. 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 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 Key Design Challenges and Pattern-Based Resolutions

The architecture overview in Section 2.3 outlines *how* TAO's pluggable protocols framework is designed. However, it does not motivate *why* this particular design was selected. In this appendix, we explore each feature in TAO's pluggable protocols framework and show how they achieve the goals described in Section 2.2. To clarify and generalize our approach, the discussion below focuses on the patterns [25] we applied to resolve the key design challenges we faced during the development process.

3.1 Adding New Protocols Transparently

Context: The QoS requirements of many applications can be supported solely by using default static protocol configurations, *i.e.*, GIOP/IIOP, described in section 2.1. However, applications with more stringent QoS requirements often require custom protocol configurations. Implementations of these custom protocols require several related classes, such as Connectors, Acceptors, Transports, and Profiles. To be integrated into a common framework, all these classes must be created consistently. In addition, many embedded and deterministic real-time systems require protocols to be configured *a priori*, with no additional protocols required once the application is configured statically. These types of systems typically cannot afford to incur the footprint overhead associated with dynamic protocol configurations.
Problem: It must be possible to add new protocols to the pluggable protocols framework without making *any* changes to the rest of the ORB. Thus, the framework must be open for extensions, but closed to modifications, *i.e.*, the Open-Closed principle [27]. Ideally, creating the 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* [25]. In the Abstract Factory pattern, a single class defines an interface for creating families of related objects, without specifying their concrete types. Subclasses of the Abstract Factory are responsible for creating concrete classes that collaborate among themselves. In the context of pluggable protocols, each Abstract Factory can create the `Connector`, `Acceptor`, `Profile`, and `Transport` classes for a particular protocol.
Applying the solution in TAO: In TAO, the role of the protocol registry is played by the `Connector_Registry` for the client and the `Acceptor_Registry` on the server. This registry is created by TAO's Resource Factory, which is a more general Abstract Factory that creates all the ORB's strategies and policies [15]. Figure 2 depicts the connector registry and its relation to the abstract factories.

Note that TAO does not use these Abstract Factories directly, however. Instead, these factories are accessed via the *Facade* [25] pattern in order to hide the complexity of manipulating multiple factories behind a simpler interface. The Registry described above plays the Facade role. As shown below, these patterns provide sufficient flexibility to add new protocols transparently to the ORB.

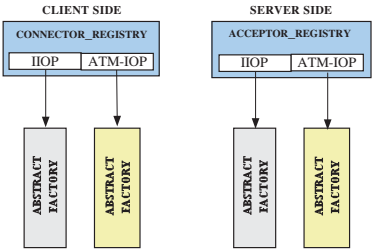

Fig. 2. TAO Connector and Acceptor Registries

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 specific protocol instance, notifying the client of its success or failure. Thereafter, the client simply invokes method invocations 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.

3.2 Adding New Protocols Dynamically

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

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

Solution: Use the Service Configurator [28] pattern, which decouples the implementation of a service from its configuration into the application. This pattern can be applied in either of the following ways:

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

This alternative is well suited for embedded systems with tight memory footprint constraints, because it minimizes the number of objects that are loaded dynamically. Implementations of the Service Configurator pattern can optimize for use cases where objects are configured statically. Embedded systems can exploit these optimizations to eliminate the need for loading objects dynamically in the pluggable protocols framework.

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

Applying the solution in TAO: TAO implements a class that maintains all parameters specified in a configuration script. Adding a new parameter to represent the list of protocols is straightforward, *i.e.*, the default registry simply examines this list and links the services into the address-space of the application, using the ACE[3] Service Configurator implementation [29] . Figure 3 depicts the connector registry and its relation to the Service Configurator.

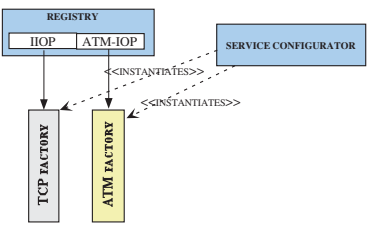

Fig. 3. TAO Connector Registry and the Service Configurator

3.3 Actively Establishing Connections

Context: When a client references an object, the ORB must obtain the corresponding profile list, which is derived from the IOR and a profile ordering policy, and transparently establish a connection to the server.
Problem: There can be one or more combinations of inter-ORB and transport protocols available in an ORB. For a given profile, the ORB must verify the presence of the associated IOP and transport protocol, if available. It must then locate the applicable `Connector` and delegate it to establish the connection.
Solution: We use the Connector pattern [31] to actively establish a connection to a remote object. This pattern decouples the connection establishment from the processing performed after the connection is successful. As before, the `Connector Registry` shown in Figure 4 is used to locate the right `Connector` for the current

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

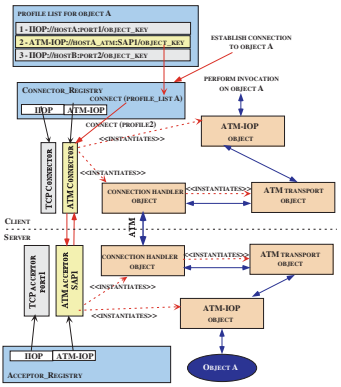

Fig. 4. Connection Establishment Using Multiple Pluggable Protocols

profile. The actual profile selected for use will depend on the set of Policies active at the time of connection establishment. However, once a profile is selected, the connector registry matches the profile type, represented by a well known tag, with an instance of a concrete `Connector`.

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

Figure 5 shows the base classes and their relations for IIOP. This figure shows an explicit co-variance between the `Profile` and the `Connector`s 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 successfully established, the `Profile` is passed a pointer to the particular IOP object and to the `Transport` objects that were created.

3.4 Passively Accepting Connections

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

Problem: The server must generate an IOR that includes all possible inter-ORB and transport-protocol-specific profiles for which the object can be accessed. As

with the client, it should be possible to add new protocols without changing the ORB.

Solution: We use the Acceptor pattern [31] to accept the connections. As with the Connector pattern, an Acceptor decouples the connection establishment from the processing performed on that connection. However, in the Acceptor pattern, the connection is accepted *passively*, rather than being initiated *actively*.

Applying the solution to TAO: Figure 5 illustrates how TAO's pluggable pro-

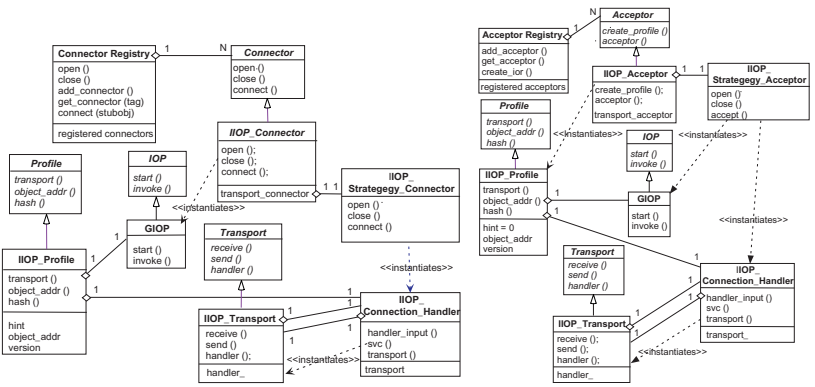

Fig. 5. Client and Server Pluggable Protocol Class Diagram

tocols framework leverages the design presented in Section 3.1. The concrete ACE `Service Handler` created by the ACE `Acceptor` is responsible for implementing the External Polymorphism pattern and encapsulating itself behind the `Transport` interface defined in our pluggable protocols framework.

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

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

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 [8]. 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 3, allowed 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 DOC middleware for avionics mission computing applications [8] and multimedia applications [32].

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.

For these experiments, we used the GIOP and GIOPlite [11] messaging protocols. 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).

Our benchmarks were run using the standard GIOP ORB messaging protocol, as well as TAO's GIOPlite messaging protocol. For the TCP/IP tests,

[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%.

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 (UIOP [5]) 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.

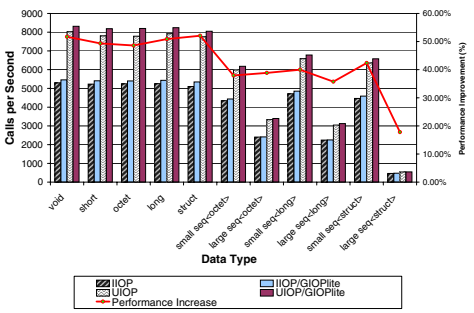

Fig. 6. TAO's Pluggable Protocols Framework Performance Over Local IPC and TCP/IP

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

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

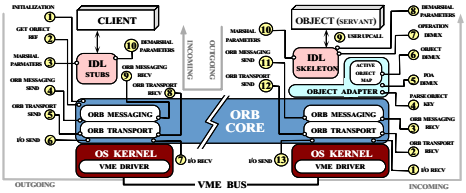

Fig. 7. Timeprobe Locations for Whitebox Experiment

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 6. All blackbox benchmarks were averaged over 100,000 two-way operation calls for each data type, as shown in Figure 6.

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 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 type `octet` and `long`. For complicated data types, such as a large `sequence` of `struct`s, 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's logical layers, or components, are shown in Figure 7 along with the timeprobe locations used for these benchmarks.

Measurement Techniques

One way to measure performance overhead of operations in complex DOC middleware is to use a profiling tool like 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 source code access.

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. 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 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 accurately measure communication delays end-to-end across distributed CPUs.

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

Whitebox Results

Figure 7 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 (μs) spent in each sequential activity that a TAO client performs to process an outgoing operation request and its reply.

Table 1. μseconds Spent in Each Client Processing Step

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

Server performance: Table 2 depicts the time in microseconds (μs) spent in each activity as a TAO server processes a request.

Table 2. μseconds Spent in Each Server Processing Step

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

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 8 compares the relative overhead attributable to the

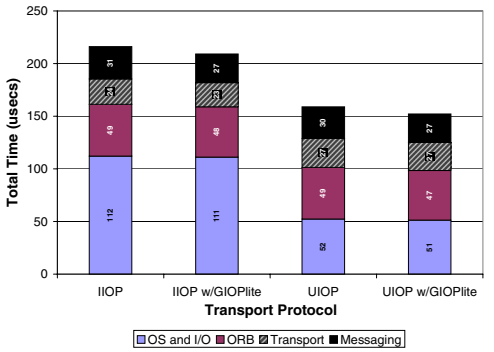

Fig. 8. Comparison of ORB and Transport/OS Overhead Using Timeprobes

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 `sequence`s vary in length. Moreover, TAO's (de)marshaling optimizations [13] incur minimal overhead when running between homogeneous ORB endsystems.

In Figure 9, 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.

TAO employs standard buffer size and data copy tradeoff optimizations. This optimization is demonstrated in Figure 9 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

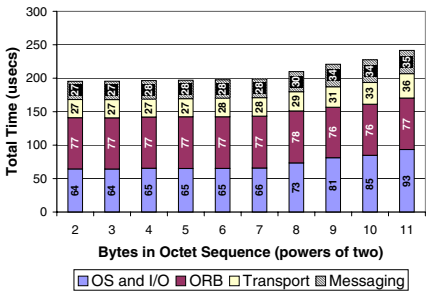

Fig. 9. ORB and Transport/OS Overhead Versus Parameter Size

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 8 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 that research and compares it with our work. *Configurable communication frameworks:* The x-kernel [34], Conduit+ [30], System V STREAMS [35], ADAPTIVE [36], and F-CSS [37] 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 DOC 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[15,30]. 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 [25]. 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 [38] Open Communications Interface (OCI) [39]. 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.

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.

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 in-

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

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

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 [26], with standard CORBA DOC middleware.

References

1. R. Johnson, "Frameworks = Patterns + Components," *Communications of the ACM*, vol. 40, Oct. 1997. 373
2. S. Vinoski, "CORBA: Integrating Diverse Applications Within Distributed Heterogeneous Environments," *IEEE Communications Magazine*, vol. 14, February 1997. 373
3. Object Management Group, *The Common Object Request Broker: Architecture and Specification*, 2.2 ed., Feb. 1998. 373
4. M. Henning and S. Vinoski, *Advanced CORBA Programming With C++*. Addison-Wesley Longman, 1999. 373
5. D. C. Schmidt, D. L. Levine, and S. Mungee, "The Design and Performance of Real-Time Object Request Brokers," *Computer Communications*, vol. 21, pp. 294–324, Apr. 1998. 373
6. G. Parulkar, D. C. Schmidt, and J. S. Turner, "a^m*athrmI*t$^{\mathrm{P}}$m: a Strategy for Integrating IP with ATM," in *Proceedings of the Symposium on Communications Architectures and Protocols (SIGCOMM)*, ACM, September 1995. 373

7. F. Kuhns, D. C. Schmidt, and D. L. Levine, "The Design and Performance of a Real-time I/O Subsystem," in *Proceedings of the 5th IEEE Real-Time Technology and Applications Symposium*, (Vancouver, British Columbia, Canada), pp. 154–163, IEEE, June 1999. 373
8. 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. 373, 385
9. F. Kuhns, C. O'Ryan, D. C. Schmidt, and J. Parsons, "The Performance of TAO's Pluggable Protocols Framework on High-speed Embedded Interconnects," Department of Computer Science, Technical Report WUCS-99-12, Washington University, St. Louis, 1999. 373, 374
10. 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, 2000. 373
11. 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. 373, 374, 376, 378, 385
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 2000. 373, 386
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*, vol. 17, Sept. 1999. 373, 375, 390
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. 373
15. D. C. Schmidt and C. Cleeland, "Applying Patterns to Develop Extensible ORB Middleware," *IEEE Communications Magazine*, vol. 37, April 1999. 373, 380, 392
16. F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal, *Pattern-Oriented Software Architecture - A System of Patterns*, Wiley and Sons, 1996. 373
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. 375
18. Compaq, Intel, and Microsoft, "Virtual Interface Architecture, Version 1.0." http://www.viarch.org, 1997. 375
19. Object Management Group, *Realtime CORBA Joint Revised Submission*, OMG Document orbos/99-02-12 ed., March 1999. 376, 377, 379
20. F. Kon and R. H. Campbell, "Supporting Automatic Configuration of Component-Based Distributed Systems," in *Proceedings of the 5th Conference on Object-Oriented Technologies and Systems*, (San Diego, CA), USENIX, May 1999. 376
21. Object Management Group, *The Common Object Request Broker: Architecture and Specification*, 2.3 ed., June 1999. 376
22. Object Management Group, *Fault Tolerance CORBA Using Entity Redundancy RFP*, OMG Document orbos/98-04-01 ed., April 1998. 376
23. Object Management Group, *CORBA Messaging Specification*, OMG Document orbos/98-05-05 ed., May 1998. 376, 379

24. 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. 376
25. E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Reading, MA: Addison-Wesley, 1995. 377, 380, 392
26. 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. 379, 393
27. B. Meyer, *Object Oriented Software Construction*. Englewood Cliffs, NJ: Prentice Hall, 1989. 380
28. P. Jain and D. C. Schmidt, "Service Configurator: A Pattern for Dynamic Configuration of Services," in *Proceedings of the 3rd Conference on Object-Oriented Technologies and Systems*, USENIX, June 1997. 381
29. 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. 382, 392
30. H. Hueni, R. Johnson, and R. Engel, "A Framework for Network Protocol Software," in *Proceedings of OOPSLA '95*, (Austin, Texas), ACM, October 1995. 391, 392
31. D. C. Schmidt, "Acceptor and Connector: Design Patterns for Initializing Communication Services," in *Pattern Languages of Program Design* (R. Martin, F. Buschmann, and D. Riehle, eds.), Reading, MA: Addison-Wesley, 1997. 382, 384
32. 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. 385
33. P. S. Inc., *Quantify User's Guide*. PureAtria Software Inc., 1996. 388
34. 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. 391
35. D. Ritchie, "A Stream Input–Output System," *AT&T Bell Labs Technical Journal*, vol. 63, pp. 311–324, Oct. 1984. 391
36. 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. 391
37. 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. 391
38. I. Object Oriented Concepts, "ORBacus." www.ooc.com/ob. 392
39. I. Object-Oriented Concepts, "ORBacus User Manual - Version 3.1.2." www.ooc.com/ob, 1999. 392