An Overview of the Real-Time CORBA Specification

OMG's Real-Time CORBA provides standard policies and mechanisms that support quality-of-service requirements end to end. Such support enhances the effectiveness of distributed object computing middleware as a platform for performance-sensitive real-time systems.

Douglas C. Schmidt
University of California, Irvine

Fred Kuhns
Washington University, St. Louis

A growing class of real-time systems require end-to-end support for various quality-of-service (QoS) aspects, including bandwidth, latency, jitter, and dependability. Applications include command and control,[1] manufacturing process control, videoconferencing, large-scale distributed interactive simulation, and test-beam data acquisition. These systems require support for stringent QoS requirements. Moreover, they have become enabling technologies for companies in markets where deregulation, global competition, and budget restrictions necessitate increased software productivity and quality.

To meet this challenge, developers are turning to distributed object computing middleware, such as the Common Object Request Broker Architecture, an Object Management Group (OMG) industry standard.[2] In complex real-time systems, DOC middleware resides between applications and the underlying operating systems, protocol stacks, and hardware. CORBA helps decrease the cycle time and effort required to develop high-quality systems by composing applications using reusable software component services rather than building them entirely from scratch. The Real-Time CORBA specification includes features to manage CPU, network, and memory resources.[3] This article describes the key Real-Time CORBA features most relevant to researchers and developers of distributed real-time and embedded systems.

REAL-TIME CORBA AT A GLANCE

The Real-Time CORBA 1.0 specification defines standard features that support end-to-end predictability for operations in *fixed-priority* CORBA applications (where operations are performed at prespecified priorities). This specification extends the existing CORBA standard and the recently adopted OMG Messaging specification.[4] In particular, RT-CORBA 1.0 leverages features from General Inter-ORB Protocol/Internet Inter-ORB Protocol (GIOP/IIOP) version 1.1 and the OMG CORBA Messaging specification's QoS policy framework. The forthcoming CORBA 3.0 standard will integrate all these features and specifications.

As Figure 1 shows, an ORB end system (that is, a node in the network) consists of network interfaces, operating system I/O subsystems, communication protocols, and CORBA-compliant middleware components and services.[5] The RT-CORBA specification identifies capabilities that ORB end systems must integrate and manage *vertically*, from network interface to application layer, and *horizontally* from peer to peer, to ensure end-to-end predictable behavior for activities that flow between CORBA clients and servers. The following list of capabilities begins with the lowest abstraction level and builds to higher-level services and applications.

Communication infrastructure resource management

An RT-CORBA end system must leverage policies and mechanisms in the underlying communication infrastructure that support resource guarantees. This support can range from choosing the connection for a particular invocation to exploiting advanced QoS features such as controlling the asynchronous transfer mode (ATM) virtual circuit cell pacing rate (the maximum relative bandwidth a given virtual circuit can use).

0018-9162/00/$10.00 © 2000 IEEE

OS scheduling mechanisms

Object request brokers exploit OS mechanisms to schedule application-level activities end to end. Because the RT-CORBA 1.0 specification targets fixed-priority real-time systems, these mechanisms correspond to managing OS thread-scheduling priorities. RT-CORBA focuses on operating systems that let applications specify scheduling priorities and policies. For example, the real-time extensions in IEEE Posix 1003.1c define a static priority first-in-first-out (FIFO) scheduling policy that meets this requirement.

Real-time ORB end system

ORBs communicate requests between clients and servers transparently. A real-time ORB end system must provide standard interfaces so that applications can specify their resource requirements to the ORB. The policy framework defined by the CORBA Messaging specification lets applications configure ORB end-system resources—such as thread priorities, buffers for message queuing, transport-level connections, and network signaling—to control ORB behavior.

Real-time services and applications

Having a real-time ORB manage end-system and communication resources provides only a partial solution. RT-CORBA ORBs must also preserve efficient, scalable, and predictable behavior end to end for higher-level services and application components. For example, a global scheduling service might manage and schedule distributed resources.[5,6] This service could interact with an ORB to provide mechanisms that support specifying and enforcing end-to-end operation timing behavior. Application developers could then structure their programs to exploit the features exported by the real-time ORB and its associated higher-level services.

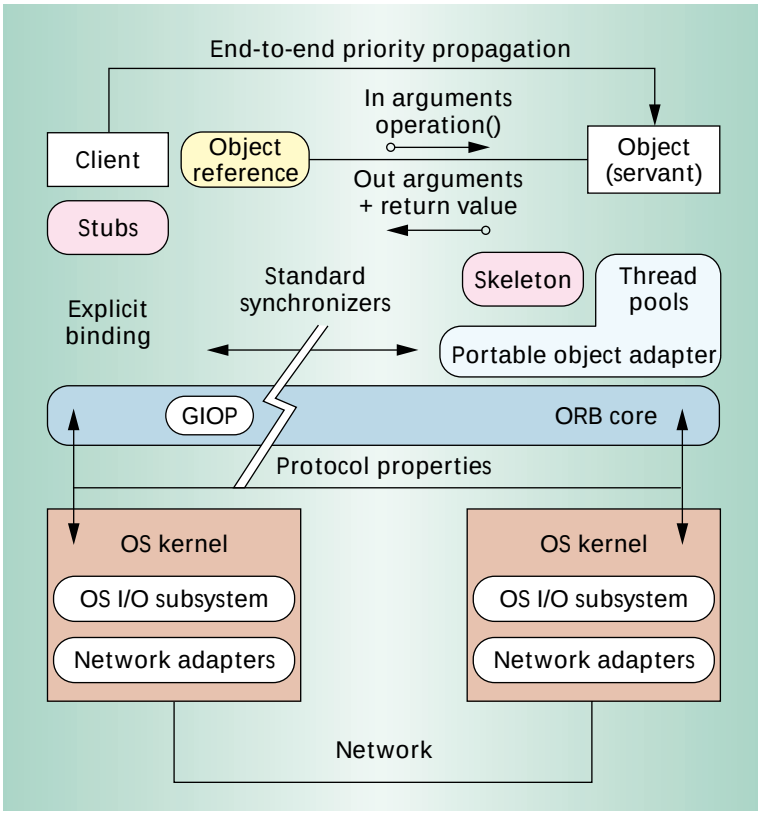

Figure 1. The roles of the ORB and end systems during object method invocations for Real-Time CORBA. A client obtains an object reference and performs method calls on the object. The client stub code provides the glue to translate the invocation into an ORB request. This request goes to the server, which may be at a different end system than that of the client. The ORB provides the lower-level services so that the method invocation can transparently go to the correct end system. On the server, the ORB hands the request to the portable object adapter (POA), which locates the object and method and performs the up call. Any return parameters go back to the client. (In the figure, red indicates the operating system and I/O subsystem; blue, the ORB core; light blue, the POA; and pink, the code generated by the IDL compiler.) The ORB can span more than one end system (as shown by the white zigzag line).

STANDARD INTERFACES AND POLICIES

To manage these capabilities, RT-CORBA defines standard interfaces and QoS policies that let applications configure and control

- processor resources using thread pools, priority mechanisms, intraprocess mutexes (mutual exclusion objects), and a global scheduling service;
- communication resources through protocol properties and explicit bindings; and
- memory resources via buffering requests in queues and bounding of thread pool size.

Applications typically specify these real-time QoS policies, along with other policies, when they call standard ORB operations, such as create_POA() or validate_connection(). For example, a QoS-enabled portable object adapter (POA) used to create an object reference ensures that any server-side policies that affect client-side requests are embedded within a *tagged component* in the object reference. Tagged components are name-value pairs for exporting attributes, such as security or QoS values, from a server to its clients within object references. Thus, clients who invoke operations on such object references can honor the policies required by the target object.

Figure 1 shows how the various RT-CORBA features relate to the existing CORBA standard.

MANAGING PROCESSOR RESOURCES

Strict control over the scheduling and execution of processor resources is essential for many fixed-priority real-time applications. Therefore, the RT-CORBA

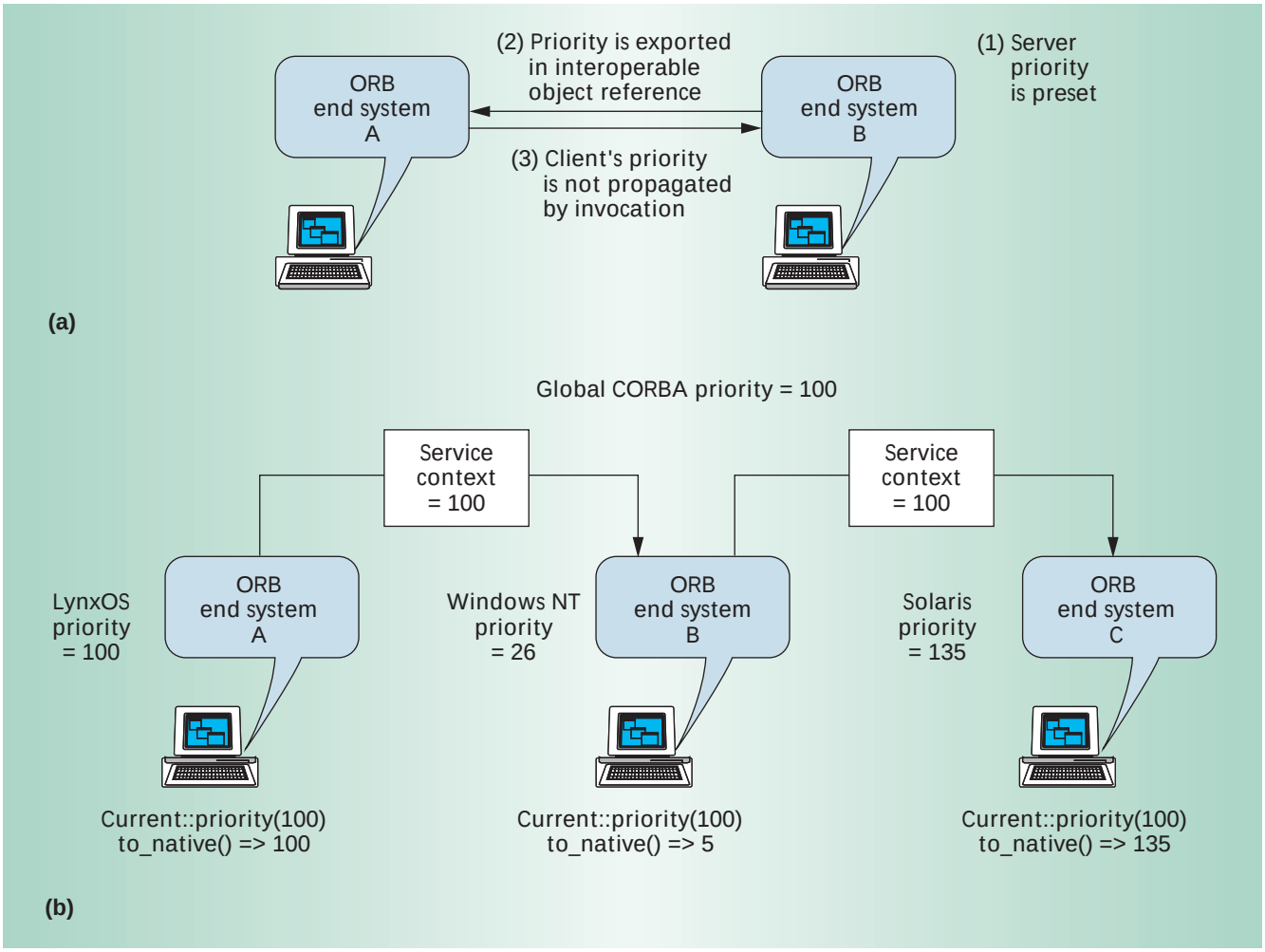

Figure 2. Real-Time CORBA priority models: (a) server declared, and (b) client propagated.

specification lets client and server applications

- determine the priority at which CORBA invocations will be processed,
- bound the priority of ORB threads, and
- ensure that intraprocess thread synchronizers have consistent semantics to minimize priority inversion.[7]

In addition, RT-CORBA lets servers predefine thread pools.

RT-CORBA's priority mechanisms cannot work miracles. ORB middleware cannot magically imbue a non-real-time OS or communication infrastructure with completely deterministic behavior. But in the appropriate environment certain RT-CORBA features can help application developers and integrators configure heterogeneous systems to preserve priorities end to end.

Priority mechanisms

Conventional CORBA ORBs provide no standard way for clients to indicate the relative priorities of their requests to ORB end systems. However, this feature is necessary to minimize end-to-end priority inversion and to bound latency and jitter for applications with deterministic real-time QoS requirements. Therefore, the RT-CORBA specification defines platform-independent mechanisms to control the priority of operation invocations.

Priority type system. The RT-CORBA specification defines two priority types—CORBA and native—to handle OS heterogeneity. Each one- or two-way CORBA operation gets a CORBA priority from 0 to 32,767. Each ORB end system along an activity path can map CORBA priorities to native priorities—priorities unique to a particular end system.

Priority models. The RT-CORBA specification defines a PriorityModel policy, shown in Figure 2, with two values: server declared and client propagated.

With server-declared priorities, a server dictates the priority at which an invocation on a particular object will execute. A server based on the PriorityModel policy's value in the POA where the object was activated designates the priority a priori. A single priority is encoded as a tagged component in the object reference, then published to the client as shown in Figure 2a.

Although the server declares the priority, the client ORB is aware of the selected priority model policy and can use this information internally. For example, the ORB may optionally use matching invocation priorities and priority bands with priorities advertised by a server to implement priority-banded connections. Thus, the ORB can guarantee that client invocations

on a particular object occur at the designated priority on the server.

The server-declared model, while useful for certain real-time applications, is not suitable in all cases. For instance, a server can avoid priority inversions by processing incoming requests at a priority equivalent to the client thread that originally invoked the operation.[7] The RT-CORBA client-propagated model lets clients declare invocation priorities that servers must honor. In this model, each invocation carries the operation's CORBA priority in the service context list that is tunneled with its GIOP request. Each ORB end system along the activity path between the client and server maps this end-to-end CORBA priority to a native OS priority. The end system then processes the request at this native priority. Moreover, if the client invokes a two-way operation, its CORBA priority determines the reply's priority.

In Figure 2b, a client on ORB end system A invokes a server on ORB end system C. End system C then invokes an intervening ORB end system, B. Each ORB end system runs operating systems with different native thread priority ranges. The client's CORBA priority propagates with the request. Each intervening server along the activity path maps the client's CORBA priority to a native priority appropriate for its host platform and end-to-end global priority. For example, on Windows NT, the global CORBA priority can map to a native OS priority of 26. On Solaris, the same global CORBA priority can map to a real-time thread with a priority of 135.

Priority transforms. The client-propagated and server-declared priority models are not sufficient for all applications. For example, the server-declared model only maps priorities to objects, which may be too coarse grained for more dynamic cases. Likewise, although the client-propagated model is more dynamic, some applications require additional control over the ultimate priority for processing a given invocation. For instance, a server might need different priority ceiling protocols, depending on whether the invocations are *inbound* (before the up call is performed) or *outbound* (before a client or servant performs a remote method invocation).

Therefore, the RT-CORBA specification lets a server application define priority transforms that set the priority for performing particular invocations. Priority might depend on external factors such as current server load, operation criticality,[5] or global scheduling service state.[6] The server implements transforms as *hooks* that it applies when it receives or sends requests. A transform hook receives the current CORBA priority and target object ID and may change the invocation priority, potentially by calling out to application-supplied code.

Inbound transforms occur during the invocation up call—that is, after reception by the ORB core but before the servant operation is dispatched in a server. *Outbound transforms*, on the other hand, occur when an *onward* operation is invoked from a servant. An onward operation occurs whenever a servant invokes an operation on an object.

The RT-CORBA specification lets a server application define priority transforms that set the priority for performing particular invocations.

Thread pools

Many embedded systems use multithreading to distinguish between different types of services, such as high- versus low-priority tasks,[1] and to support thread preemption to prevent unbounded priority inversion. Before the RT-CORBA specification, however, there was no standard application programming interface (API) for programming multithreaded CORBA servers. Thus, developers could not use CORBA to program multithreaded real-time systems without using proprietary ORB features.

A *reactive* concurrency model is one way to implement a server ORB without threads.[8] A server ORB reads each request from the underlying communication mechanism, processes it to completion, then retrieves the next request, and so forth. If all requests required a fixed, relatively small amount of processing, a reactive concurrency model might be feasible. However, many distributed applications have complex object implementations that run for variable—and sometimes long—durations. Moreover, to avoid unbounded priority inversion and deadlock, real-time applications often require some form of preemptive multithreading.

RT-CORBA addresses these concurrency issues by defining a standard *thread pool* model.[8] This model lets server developers preallocate thread pools and set certain thread attributes, such as default priority levels. Thread pools can help real-time ORB end systems and applications that want to leverage the benefits of multithreading while bounding the amount of memory resources that they consume. Moreover, developers can optimally configure thread pools to buffer or not buffer requests, thus providing further control over memory usage.

Developers can define and associate thread pools with POAs in an RT-CORBA server. Each POA must have one thread pool, although a thread pool can process requests for multiple POAs. RT-CORBA defines two different thread pool styles: without and with lanes.

Thread pools without lanes. The simplest RT-CORBA thread pool model lets developers control the overall concurrency level in server ORBs and applications. An ORB processes client messages using a thread pool having a fixed number of statically allocated threads. However, even when not used, these preallocated threads consume system resources. Therefore,

Developers can configure thread pools so that lanes with higher priorities borrow threads from lanes with lower priorities.

an RT-CORBA interface lets server developers preallocate an initial number of so-called static threads, but this pool can grow dynamically to handle bursts of client requests.

Server applications can use the create_threadpool() API to specify

- the default number of static threads created initially,
- the maximum number of threads that can be created dynamically, and
- the default priority of all these threads—these priorities may change dynamically on the basis of priority models or priority transforms.

If a request arrives and all existing threads are busy, the ORB can create a new thread to handle the request. However, once the maximum number of threads in the pool spawn, no additional threads are created.

Developers can optimally configure a pool for maximum buffer size or number of requests. If buffering is enabled for the pool, the request remains in the queue until a thread is available to process it. If no queue space is available, or if request buffering was not specified, the ORB should raise a *transient* exception, indicating a temporary resource shortage. When a client receives this exception, it can reissue the request at a later point.

Thread pools with lanes. Many real-time embedded systems applications statically associate global CORBA priorities to thread pools. For example, a telecommunications application can select three distinct priorities to represent low-latency, high-throughput, and best-effort request classes. Alternatively, a fixed set of rate groups with corresponding global CORBA priorities are convenient for applications with real-time periodic processing requirements. In these scenarios, partitioning the threads in a thread pool into different subsets, each with different priorities, is desirable. Therefore, RT-CORBA defines a thread-pool-with-lanes model, so developers can bound both the overall concurrency of a server and the amount of work performed at a given priority level.

For each lane in this thread pool model, the server specifies the CORBA priority, static thread count, and dynamic thread count. The server assigns the lane priority to dynamic threads. Developers can configure thread pools so that lanes with higher priorities borrow threads from lanes with lower priorities. The priority of a borrowed thread is temporarily raised to that of the lane borrowing it. When the invocation processing finishes, the thread returns to its original lane, and its priority reverts to its previous value. Thread pools with lanes can also support request buffering if no threads are available to process incoming requests.

Standard synchronizers

The original CORBA specification does not define a threading model. CORBA applications, therefore, lack a standard, portable API that could ensure semantic consistency between their synchronization mechanisms and the internal synchronization mechanisms used by an ORB. Real-time applications, however, require this consistency to enforce priority inheritance and priority ceiling protocols.[7] Therefore, the RT-CORBA specification defines a standard set of locality-constrained mutexes.

Global scheduling service

The scheduling abstractions defined by real-time operating systems such as VxWorks, LynxOS, and Posix 1003.1c implementations are relatively low level. For example, these abstractions require that developers map their high-level application QoS requirements into lower-level OS mechanisms, such as thread priorities and virtual circuit bandwidth/latency parameters. This manual mapping step is not intuitive for many application developers, who prefer to design in terms of object interfaces and object operations.

RT-CORBA defines a global scheduling service so that applications can specify scheduling requirements in a higher-level, clearer way.[3] This service, a CORBA object, allocates system resources to meet the QoS needs of the applications that share the ORB end system. Applications can use this service to specify their operations' processing requirements in terms of various parameters, such as worst-case execution time or period.

MANAGING INTER-ORB COMMUNICATION

Historically, the CORBA specification and conventional ORBs have supported *location transparency*, which shields applications from detecting whether components are distributed or collocated in the same process. Moreover, the features of the underlying OS, network, and bus are considered a black box. Although this encapsulation is useful for applications with best-effort QoS requirements, it is inadequate for those with more stringent QoS requirements.

RT-CORBA lets applications control the underlying communication protocols and end-system resources. The specification defines standard interfaces for selecting and configuring certain protocol properties. In addition, client applications can explicitly bind to server objects using priority bands and private connections.

Selecting and configuring protocol properties

CORBA uses inter-ORB communication mechanisms to exchange requests between clients and servers. These mechanisms are built on lower-level protocols that provide various types of QoS mechanisms. Inter-ORB protocol (IOP) instances contain

both an ORB protocol and a mapping to a specific underlying transport protocol. For example, IIOP is a mapping of GIOP onto the transfer control protocol/Internet protocol (TCP/IP). Thus, an IOP contains two protocol layers—ORB and transport—each having its own protocol properties set.

RT-CORBA defines an interface so that applications can specify ORB- and transport-specific protocol properties that control various communication protocol features, such as ATM virtual circuits or Internet *r*esource reser*v*ation *p*rotocol (RSVP) traffic specification. A Protocol struct defines each ORB or transport protocol properties tuple, as defined in this CORBA Interface Definition Language (IDL) definition:

```
interface ProtocolProperties {};

typedef struct {
  IOP::ProfileId protocol_type;
  ProtocolProperties
    orb_protocol_properties;
  ProtocolProperties
    transport_protocol_properties;
} Protocol;
typedef sequence <Protocol>
  ProtocolList;
```

The order in which protocol properties appear in the ProtocolList indicates the order of an application's protocol preferences. For example, a client may specify that IIOP is more preferable than other protocol combinations. To let applications select and configure their desired ORB or transport protocol properties, RT-CORBA defines the ClientProtocol and ServerProtocol QoS policies.

Server-side protocol properties. CORBA servers can use the ServerProtocol policy to decide which protocols to configure into an object reference. This policy can be passed with other POA policies when the create_POA() operation is invoked on the PortableServer: :POA interface. The ServerProtocol policy has two purposes:

- publish a list of available protocols to clients, and
- define protocol configuration attributes for server connections.

The POA ensures that the ordering of profiles in object references conforms to the ordering of protocols specified in the ServerProtocol policy. Thus, a server can export its protocol preferences to clients by passing them in object references whose profiles are arranged in a particular order. When a client receives the object reference, it can either accept the server's preference or use different selection criteria.

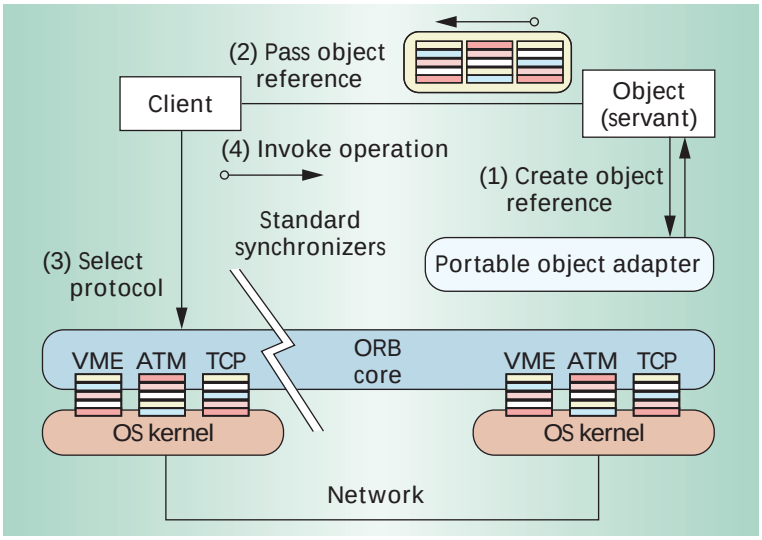

Figure 3. Configuring and selecting protocol properties. A server designates the protocols available to the client (the stacks indicate one object reference with multiple protocol-specific addresses). The server publishes the virtual machine environment (VME), ATM, or TCP protocols, in that order, in a tagged component in the object reference. The client then must abide by the ClientProtocol policy propagated by the server, and select from one of these three protocols. (The white zigzag line indicates one or more hosts.)

Client-side protocol properties. Client applications can use the ClientProtocol policy to decide which protocols to use when they connect to objects. This policy takes effect when a client obtains a binding to an object. The ClientProtocol policy indicates the protocol properties a client is interested in, as well as the ordering of its preferences.

Either a client or a server, but not both for the same object reference, can set the ClientProtocol policy. Servers can publish particular protocol requirements and preferences on a per-object basis; clients can change protocol policies on a per-invocation basis. If the server sets the ClientProtocol policy, it propagates this policy to the client in the object reference. Figure 3 shows how a server can designate the protocols available to the client. This feature lets a server enforce specific inter-ORB protocol requirements on clients.

Interface inheritance can define the particular properties for specific protocols. For example, the standard TCP properties are

```
interface TCPProtocolProperties
  : ProtocolProperties
{
  attribute long send_buffer_size;
  attribute long recv_buffer_size;
  attribute boolean keep_alive;
  attribute boolean dont_route;
  attribute boolean no_delay;
};
```

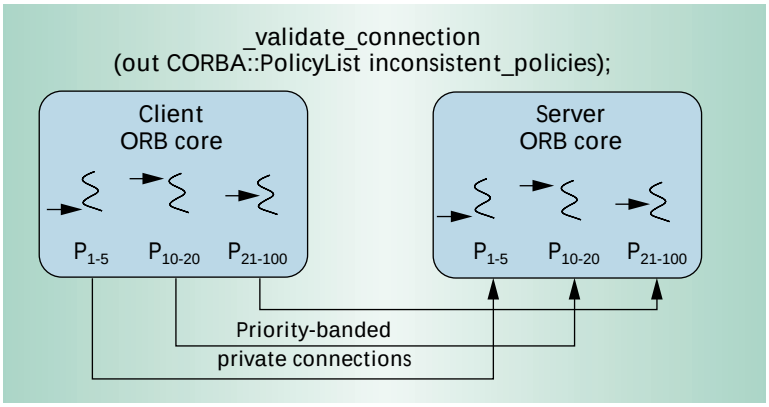

Figure 4. Explicit binding. Invocations flow from client to server through a preallocated connection (the curved lines represent threads) in a fixed priority range (for example, $P_{1\text{-}5}$ means priority values 1 through 5).

This protocol property interface lets applications set common attributes of TCP end points. For example, the send and receive buffer size attributes can set the size of end-point socket queues. Many TCP implementations use these values to determine the TCP window size, which in turn affects end-to-end throughput. If the keep_alive attribute is enabled, TCP will probe inactive connections to verify they are still valid. Finally, the no_delay attribute disables TCP's Nagle algorithm (a feature of the TCP that tries to minimize the number of small packets on the network, thereby reducing congestion) so that small requests can be sent even if earlier requests have not yet been acknowledged.

Explicit binding

The original CORBA specification supported only implicit binding.[2] This model established resources on demand—for example, after a client's first invocation on the server—along the activity path between a client and its server object.

Unfortunately, implicit binding is inadequate for real-time applications with deterministic QoS requirements. In particular, deferring object/server activation and resource allocation until runtime can significantly increase latency and jitter. Moreover, because of head-of-line blocking associated with connection queues processed in FIFO order, connection multiplexing can yield substantial priority inversion.[8]

To avoid these problems, RT-CORBA defines an explicit binding mechanism that uses the validate_connection operation defined in the CORBA::Object interface in CORBA Messaging. This mechanism lets clients preestablish connections to servers and control how client requests travel over these connections.

Priority-banded connections. These connections let clients specify explicit priorities for each network connection. Clients can also select the appropriate connection at runtime based on the priority that CORBA assigns the thread that invoked an operation. When clients establish connections explicitly, they must specify policies that define one or more priority bands.

The client exports priority-band information to the server within the service context of the first invocation sent across the connection. For instance, explicit binding information goes to the server in a request for _bind_priority_band(), which is an *implicit operation*. When a server receives a _bind_priority_band() request, it allocates resources to the connection. The server processes subsequent requests on this connection at the requested priority.

In the absence of a _bind_priority_band() operation, the client performs an implicit bind when sending the first invocation over the connection. This request's service context must contain the CORBA priority range—minimum and maximum values—for the banded connection. The server then allocates any necessary resources to ensure that subsequent requests arriving at this connection are processed at the desired priority.

Private connections. Many ORBs support multiplexed connections, which use limited OS resources more efficiently than do traditional nonmultiplexed connections (where each invocation can yield a new client-server connection).[8] However, real-time applications often require private (nonmultiplexed) connections, which are well suited for applications with deterministic QoS requirements. In this case, a connection cannot be reused for another two-way request until the reply for the previous request is received. To support this feature, RT-CORBA provides PrivateConnection. This policy lets clients select private connections that minimize the duration of any end-to-end priority inversions. Oddly, RT-CORBA has no API to explicitly request a multiplexed connection, treating this feature instead as an ORB implementation detail.

Figure 4 shows the use of priority-banded, private connections between a client and server. Private connections combine with priority banding. Thus, each client operation travels to the server through a preallocated connection that is assigned to a fixed-priority range. (A connection may be associated with a single priority or a range of priorities. For example, all invocations with priorities between 20 and 30 use connection X.) The server ORB processes the servant up call at the specified priority and sends the reply across the same nonmultiplexed connection. This combination of features preserves end-to-end prioritization and eliminates key sources of priority inversion.

Because of constraints on weight and power consumption, memory footprint, and performance, software development for real-time, embedded systems has historically lagged behind mainstream software development. As a result, these software sys-

tems are costly to evolve and maintain. Moreover, they are often so specialized that they cannot adapt readily to meet new market opportunities or technology innovations.

Meeting the QoS requirements of next-generation distributed applications requires an integrated architecture that can deliver end-to-end QoS support at multiple levels. DOC middleware based on RT-CORBA 1.0 offers solutions to some of the resource management challenges that face real-time-systems researchers and developers,[3] particularly for those systems that can employ fixed-priority scheduling.

However, an important class of mission-critical applications cannot meet their QoS requirements under dynamic load conditions using only the features standardized in the RT-CORBA 1.0 specification.[6] Moreover, developers of complex mission-critical applications will find it difficult to determine the priorities of various operations a priori unless they significantly underuse various resources, such as the CPU. To address these issues, the OMG is standardizing dynamic scheduling techniques, such as deadline-based[9] or value-based scheduling.

A freely available, open-source implementation of RT-CORBA called TAO (*T*he *A*CE (adaptive communication environment) *O*RB) is available at http://www.cs.wustl.edu/~schmidt/TAO.html. TAO has been used on a wide range of distributed real-time, embedded systems, including an avionics mission computing architecture for Boeing,[1] the Science Applications International Corp. (SAIC) next-generation Run Time Infrastructure implementation for the Defense Modeling and Simulation Organization's High Level Architecture, and high-energy physics experiments at the Stanford Linear Accelerator Center (SLAC) and the Keck Observatory in the Hawaiian Islands. ✷

Acknowledgments

This work was supported in part by Boeing, BBN, Cisco, the US Defense Department's Advanced Research Projects Agency Contract 9701516, NSF Grant NCR-9628218, Motorola, Siemens, and Sprint.

References

1. T.H. Harrison, D.L. Levine, and D.C. Schmidt, "The Design and Performance of a Real-Time CORBA Event Service," *Proc. 1997 ACM SIGPLAN Conf. Object-Oriented Programming Systems, Languages, and Applications (OOPSLA 97)*, ACM Press, New York, 1997, pp. 184-200.
2. Object Management Group, *The Common Object Request Broker: Architecture and Specification*, 2.3 ed., June 1999.
3. Object Management Group, *Realtime CORBA Joint Revised Submission*, OMG document, ORBOS/99-02-12 ed., Mar. 1999.
4. Object Management Group, *CORBA Messaging Specification*, OMG document, ORBOS/98-05-05 ed., May 1998.
5. D.C. Schmidt, D.L. Levine, and S. Mungee, "The Design and Performance of Real-Time Object Request Brokers," *Computer Comm.*, Apr. 1998, pp. 294-324.
6. C.D. Gill, D.L. Levine, and D.C. Schmidt, "The Design and Performance of a Real-Time CORBA Scheduling Service," to be published in *Int'l J. Time-Critical Computing Systems*, 2000.
7. R. Rajkumar, L. Sha, and J.P. Lehoczky, "Real-Time Synchronization Protocols for Multiprocessors," *Proc. 9th IEEE Real-Time Systems Symp.*, IEEE CS Press, Los Alamitos, Calif., 1988, pp. 259-269
8. D.C. Schmidt et al., "Software Architectures for Reducing Priority Inversion and Non-Determinism in Real-Time Object Request Brokers," to be published in *J. Real-Time Systems*, 2000.
9. Y.-C. Wang and K.-J. Lin, "Implementing a General Real-Time Scheduling Framework in the RED-Linux Real-Time Kernel," *Proc. 20th IEEE Real-Time Systems Symp.*, IEEE CS Press, Los Alamitos, Calif., 1999, pp. 246-255.

Douglas C. Schmidt is an associate professor in the Electrical and Computer Engineering Department at the University of California, Irvine. His interests include object-oriented techniques for developing high-performance, real-time distributed-object computing systems on parallel processing platforms running over high-speed ATM networks and embedded system interconnections. Schmidt received a BS and an MA in sociology from the College of William and Mary, Williamsburg, Virginia, and an MS and a PhD in information and computer science from the University of California, Irvine. He is a member of the IEEE, the ACM, and Usenix. Contact him at schmidt@uci.edu.

Fred Kuhns is a senior research associate in the Department of Computer Science at Washington University, St. Louis. His interests include operating system and network support for high-performance, real-time distributed-object computing systems. Kuhns received an MSEE from Washington University, St. Louis, and a BSEE from the University of Memphis. Contact him at fredk@cs.wustl.edu.