Evaluating Architectures for Multi-threaded CORBA Object Request Brokers

Douglas C. Schmidt
schmidt@cs.wustl.edu
Department of Computer Science, Washington University
St. Louis, MO 63130, USA*

This paper will appear in the Communications of the ACM Special Issue on CORBA edited by Krishnan Seetharaman, October 1998, Vol. 41, No. 10.

Introduction

CORBA Object Request Brokers (ORBs) deliver client requests to servants and return responses to clients [1]. To accomplish this, ORBs manage transport connections, perform transport endpoint demultiplexing, and provide the multi-threading architecture used by applications. The architecture used to multi-thread an ORB has a substantial impact on its performance and predictability [2]. A key challenge for ORB developers and application programmers, therefore, is to devise threading architectures that can handle multiple client requests efficiently.

Multi-threading allows operations to execute simultaneously without impeding the progress of other operations. Likewise, multi-threading can minimize latency and ensure predictability in real-time systems [2]. This paper describes and evaluates common CORBA multi-threading architectures used by ORB implementations, including CORBAplus, HP ORB Plus, miniCOOL, MT-Orbix, TAO, and VisiBroker.

Sidebar: Overview of Multi-threading

A thread is a single sequence of execution steps performed in the context of a process [3]. In addition to its own instruction pointer, a thread contains resources like a run-time stack of method activation records, a set of registers, and thread-specific storage. A preemptive multi-threaded OS, such as Solaris or Windows NT, provides a scheduler that ensures each thread of control runs according to its priority and/or its execution quantum.

Contemporary operating systems support the concurrent execution of multiple processes, each containing one or more threads. As shown in Figure 1, a process serves as the unit of

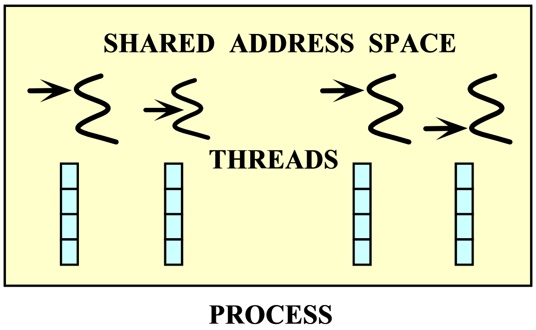

Figure 1: Threads in a Process

protection and resource allocation within a separate hardware-protected address space. A thread serves as the unit of execution that runs within a process address space that is shared with other threads.

Motivation for Multi-threading Object Request Brokers

The following are common motivations for developing and using multi-threaded ORBs:

- *Simplify program design* by allowing multiple servants to execute independently using conventional programming abstractions like synchronous CORBA remote method requests and replies;
- *Improve end-to-end throughput and latency performance* by using the parallel processing capabilities of multiprocessor hardware platforms and by overlapping computation with communication;
- *Improve perceived response time* for interactive client applications, such as user interfaces or network management tools, by associating separate threads with different operations so client operations do not block indefinitely.

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

Programming applications on CORBA ORBs that lack multi-threading is hard, particularly for developers of servers or real-time applications. Without multi-threading capabilities, developers must ensure that requests can be handled quickly enough that new requests are not starved or unduly delayed. In practice, however, many requests cannot be serviced quickly enough in a single-threaded ORB to avoid starving clients.

With multiple threads, each request can be serviced in its own thread, independent of other requests. This way, clients are not starved by waiting for their requests to be serviced. Likewise, system resources are conserved since creating a new thread is typically much less expensive than creating an entirely new process [3].

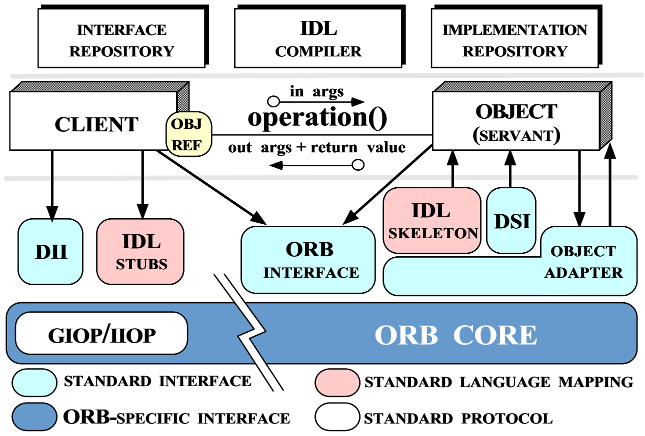

Figure 2: Components in the CORBA 2.x Reference Model

Sidebar: Overview of the CORBA ORB Reference Model

CORBA Object Request Brokers (ORBs) [4] allow clients to invoke operations on distributed objects without concern for the following issues [1]:

Object location: CORBA objects either can be collocated with the client or distributed on a remote server, without affecting their implementation or use.

Programming language: The languages supported by CORBA include C, C++, Java, Ada95, COBOL, and Smalltalk, among others.

OS platform: CORBA runs on many OS platforms, including Win32, UNIX, MVS, and real-time embedded systems like VxWorks, Chorus, and LynxOS.

Communication protocols and interconnects: The communication protocols and interconnects that CORBA can run on include TCP/IP, IPX/SPX, FDDI, ATM, Ethernet, Fast Ethernet, embedded system backplanes, and shared memory.

Hardware: CORBA shields applications from side-effects stemming from differences in hardware, such as storage layout and data type sizes/ranges.

Figure 2 illustrates the components in the CORBA 2.x reference model, all of which collaborate to provide the portability, interoperability, and transparency outlined above. Each component in the CORBA reference model is outlined below:

Client: This program entity performs application tasks by obtaining object references to objects and invoking operations on them. Objects can be remote or collocated relative to the client. Ideally, accessing a remote object should be as simple as calling an operation on a local object, *i.e.*, `object→operation(args)`. Figure 2 shows the underlying components described below that ORBs use to transmit remote operation requests transparently from client to object.

Object: In CORBA, an object is an instance of an Interface Definition Language (IDL) interface. The object is identified by an *object reference*, which uniquely names that instance across servers. An *ObjectId* associates an object with its servant implementation, and is unique within the scope of an Object Adapter. Over its lifetime, an object has one or more servants associated with it that implement its interface.

Servant: This component implements the operations defined by an OMG Interface Definition Language (IDL) interface. In languages like C++ and Java that support object-oriented (OO) programming, servants are implemented using one or more class instances. In non-OO languages, like C, servants are typically implemented using functions and `structs`. A client never interacts with a servant directly, but always through an object identified by an object reference.

ORB Core: When a client invokes an operation on an object, the ORB Core is responsible for delivering the request to the object and returning a response, if any, to the client. For objects executing remotely, a CORBA-compliant ORB Core communicates via a version of the General Inter-ORB Protocol (GIOP), most commonly the Internet Inter-ORB Protocol (IIOP) that runs atop the TCP transport protocol. An ORB Core is typically implemented as a run-time library linked into both client and server applications.

ORB Interface: An ORB is an abstraction that can be implemented various ways, *e.g.*, one or more processes or a set of libraries. To decouple applications from implementation details, the CORBA specification defines an interface to an ORB. This ORB interface provides standard operations to initialize and shutdown the ORB, convert object references to strings and back, and create argument lists for requests made through the *dynamic invocation interface* (DII).

OMG IDL Stubs and Skeletons: IDL stubs and skeletons serve as a "glue" between the client and servants, respectively, and the ORB. Stubs provide a strongly-typed, *static invocation interface* (SII) that marshals application parameters into a common data-level representation. Conversely, skeletons demarshal the data-level representation back into typed parameters that are meaningful to an application.

IDL Compiler: An IDL compiler transforms OMG IDL definitions into stubs and skeletons that are generated automatically in an application programming language like C++ or Java. In addition to providing programming language transparency, IDL compilers eliminate common sources of network programming errors and provide opportunities for automated compiler optimizations [5].

Dynamic Invocation Interface (DII): The DII allows clients to generate requests at run-time, which is useful when an application has no compile-time knowledge of the interface it accesses. The DII also allows clients to make *deferred synchronous* calls, which decouple the request and response portions of two-way operations to avoid blocking the client until the servant responds. In contrast, in CORBA 2.x, SII stubs only support *two-way*, *i.e.*, request/response, and *one-way*, *i.e.*, request-only operations.[1]

Dynamic Skeleton Interface (DSI): The DSI is the server's analogue to the client's DII. The DSI allows an ORB to deliver requests to servants that have no compile-time knowledge of the IDL interface they implement. Clients making requests need not know whether the server ORB uses static skeletons or dynamic skeletons. Likewise, servers need not know if clients use the DII or SII to invoke requests.

Object Adapter: An Object Adapter associates servants with objects, creates object references, demultiplexes incoming requests to servants, and collaborates with the IDL skeleton to dispatch the appropriate operation upcall on a servant. CORBA 2.2 portability enhancements [4] define the Portable Object Adapter (POA), which supports multiple nested POAs per ORB. Object Adapters enable ORBs to support various types of servants that possess similar requirements. This design results in a smaller and simpler ORB that can support a wide range of object granularities, lifetimes, policies, implementation styles, and other properties.

Interface Repository: The Interface Repository provides run-time information about IDL interfaces. Using this information, it is possible for a program to encounter an object whose interface was not known when the program was compiled, yet, be able to determine what operations are valid on the object and make invocations on it using the DII. In addition, the Interface Repository provides a common location to store additional information associated with interfaces to CORBA objects, such as type libraries for stubs and skeletons.

Implementation Repository: The Implementation Repository [7] contains information that allows an ORB to activate servers to process servants. Most of the information in the Implementation Repository is specific to an ORB or OS environment. In addition, the Implementation Repository provides a common location to store information associated with servers, such as administrative control, resource allocation, security, and activation modes.

Evaluating Multi-threading Architectures for Object Request Brokers

This section describes and evaluates common ORB multi-threading architectures that are used by one or more CORBA implementations. Each architecture is evaluated in terms of its ability to support the aggregate processing capacity of ORB endsystem components and application operations in one or more threads.

There are a variety of strategies for structuring the multi-threading architecture in an ORB [8]. Below, we describe a number of alternative ORB Core multi-threading architectures, focusing on server-side multi-threading.

The Thread-per-Request Architecture

• **Overview:** The thread-per-request architecture [9] handles each request from a client in a separate thread of control. As shown in Figure 3, the components in the thread-per-request architecture include an I/O thread and one or more dynamically spawned threads. The I/O thread `selects` (**1**) on the socket endpoints, `reads` (**2**) new client requests, and (**3**) spawns a new thread for each request. The newly spawned thread dispatches the operation upcall (**4**) and exits when the upcall completes (**5**).

• **Advantages:** The main advantage of thread-per-request is that it is straightforward to implement. This architecture is particularly useful for ORBs that handle long-duration requests, such as database queries, from multiple clients.

• **Disadvantages:** The disadvantage with thread-per-request is that it can consume a large number of OS resources if many clients make requests simultaneously. Moreover, it is inefficient for short-duration requests because it incurs excessive thread creation overhead. In addition, thread-per-request architectures are not suitable for real-time applications since

[1] The OMG has standardized a static asynchronous method invocation interface in the Messaging specification [6], which will appear in CORBA 3.0.

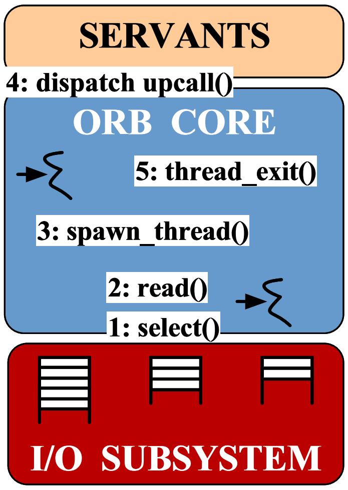

Figure 3: Server-side Thread-per-Request Multi-threading Architecture

the overhead of spawn a thread for each request can be non-deterministic.

The HP ORBPlus ORB uses the thread-per-request architecture. MT-Orbix can be configured to use thread-per-request.

The Thread-per-Connection Architecture

• **Overview:** The thread-per-connection architecture is a variation of thread-per-request that amortizes the cost of spawning the thread across multiple requests from the same client process. As shown in Figure 4, the components in

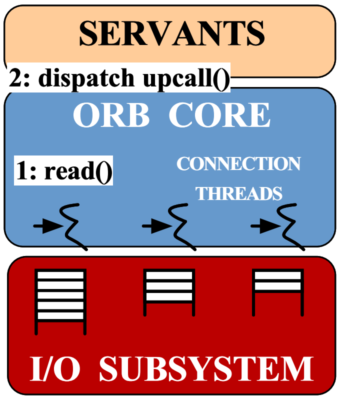

Figure 4: Server-side Thread-per-Connection Multi-threading Architecture

the thread-per-connection architecture are a set of connection threads, each of which is dedicated to handle a separate client for the duration of its connection. Each connection thread `reads` (**1**) a new request directly from its socket endpoint, dispatches the upcall (**2**), and then returns to `read` the next request from its connection.

• **Advantages:** Like thread-per-request, thread-per-connection is straightforward to implement. It is well suited for ORBs that perform long-duration conversations with multiple clients.

• **Disadvantages:** The primary disadvantage with thread-per-connection is that it does not support load balancing effectively. Moreover, for clients that make only a single request to each server, thread-per-connection is equivalent to the thread-per-request architecture.

VisiBroker from Inprise, the TAO ORB [2], and SunSoft IIOP implement the thread-per-connection architecture.

The Thread-per-object Architecture

• **Overview:** The thread-per-object architecture[2] associates a thread for each object, *e.g.*, a video-on-demand session, registered in the ORB's Object Adapter. As shown in Figure 5, the components in the thread-per-object architecture

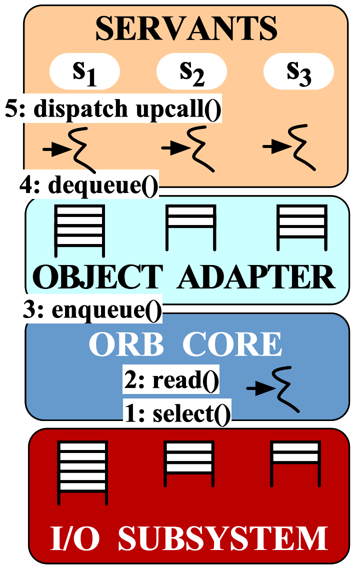

Figure 5: Server-side Thread-per-object Multi-threading Architecture

are an I/O thread and a set of threads each of which is dedicated to handle a separate object's servant, *e.g.*, S_1, S_2, and S_3. The I/O thread `selects` (**1**) and `reads` (**2**) a new request from a socket endpoint and passes the request to the Object Adapter. The Object Adapter then (**3**) inserts the request into a queue associated with a servant and the servant's thread. This thread will dequeue requests from its queue (**4**) and dispatch the upcall on the servant (**5**).

• **Advantages:** Thread-per-object is useful for programmers who want to minimize the amount of rework required to multi-thread existing single-threaded servants. So long as all

[2] This architecture is also known as "thread-per-object" [10].

methods in a servant only access servant-specific state there is no need for explicit synchronization operations.

• **Advantages:** A disadvantage with thread-per-object is that it does not support load balancing effectively since it serializes request processing in each servant. Therefore, if one servant receives considerably more requests than others it can become a performance bottleneck. In addition, thread-per-object designs are prone to deadlock on nested callbacks.

MT-Orbix can be configured to support thread-per-object.

Thread Pool Architectures

A thread pool [11] is another variation of the thread-per-request architecture that amortizes thread creation costs by pre-spawning a pool of threads. A thread pool architecture is useful for ORBs that want to bound the number of OS resources they consume. Client requests can be executed concurrently until the number of simultaneous requests exceeds the number of threads in the pool. At this point, additional requests must be queued until a thread becomes available.

Thread pool is a common architecture for structuring ORB multi-threading, particularly for real-time ORBs [2]. Below, we describe and evaluate several common thread pool architectures.

The Worker Thread Pool Architecture

As shown in Figure 6, the components in a worker thread pool

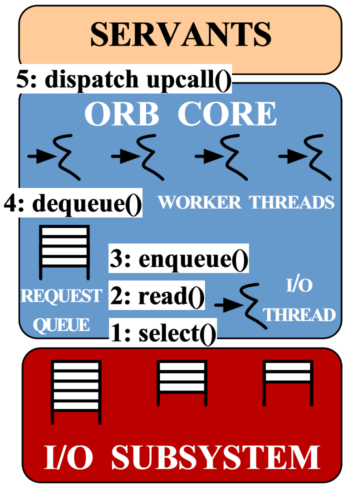

Figure 6: Server-side Worker Thread Pool Multi-threading Architecture

include an I/O thread, a request queue, and a pool of worker threads. The I/O thread `selects` (**1**) on the socket endpoints, `reads` (**2**) new client requests, and (**3**) inserts them into the tail of the request queue. A worker thread in the pool dequeues (**4**) the next request from the head of the queue and dispatches it (**5**).

• **Advantages:** The chief advantage of the worker thread pool multi-threading architecture is its ease of implementation. In particular, the request queue provides a straightforward producer/consumer design.

• **Disadvantages:** The disadvantages of this model stem from the excessive context switching and synchronization required to manage the request queue, as well as request-level priority inversion caused by connection multiplexing.

The Expersoft CORBAplus ORB uses the worker thread pool architecture.

The Leader/Follower Thread Pool Architecture

• **Overview:** The leader/follower thread pool architecture is an optimization of the worker thread pool model. As shown in Figure 7, a pool of threads is allocated and a leader thread

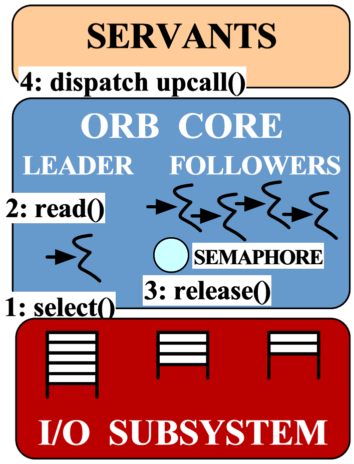

Figure 7: Server-side Leader/Follower Multi-threading Architecture

is chosen to `select` (**1**) on connections for all servants in the server process. When a request arrives, this thread reads (**2**) it into an internal buffer. If this is a valid request for a servant, a follower thread in the pool is released to become the new leader (**3**) and the leader thread dispatches the upcall (**4**). After the upcall is dispatched, the original leader thread becomes a follower and returns to the thread pool. New requests are queued in socket endpoints until a thread in the pool is available to execute the requests.

• **Advantages:** Compared with the worker thread pool design, the chief advantage of the leader/follower thread pool architecture is that it minimizes context switching overhead incurred by incoming requests. Overhead is minimized since the request need not be transferred from the thread that read it to another thread in the pool that processes it.

• **Disadvantages:** The leader/follower architecture's disadvantages are largely the same as with the worker thread design. In addition, it is harder to implement the leader/follower model than the worker thread pool model.

Sun's miniCOOL ORB uses the leader/follower thread pool architecture.

Hybrid Architectures

Several architectures for structuring ORB concurrency combine a number of the other multi-threading architectures described above.

Threading Framework Architecture

• **Overview:** A very flexible way to implement an ORB multi-threading architecture is to allow application developers to customize hook methods provided by a *threading framework*. One way of structuring this framework is shown in Figure 8. This design is based on the MT-Orbix thread filter framework, which implements a variant of the Chain of Responsibility pattern [12].

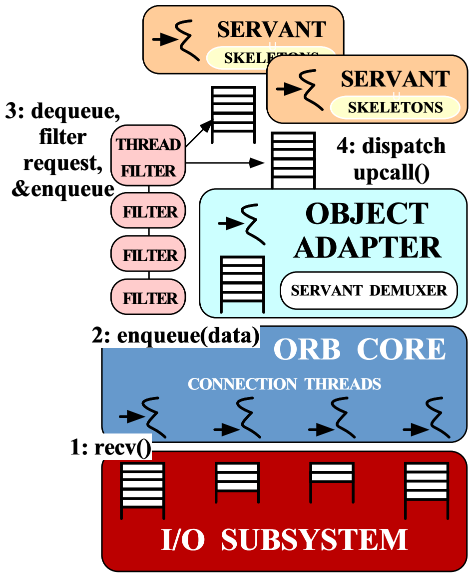

Figure 8: Server-side Thread Framework Multi-threading Architecture

In MT-Orbix, an application can install a thread filter at the top of a chain of filters. Filters are application-programmable hooks that can perform a number of tasks. Common tasks include intercepting, modifying, or examining each request sent to and from the ORB.

In the thread framework architecture, a connection thread in the ORB Core `reads` (**1**) a request from a socket endpoint and enqueues the request on a request queue in the ORB Core (**2**). Another thread then dequeues the request (**3**) and passes it through each filter in the chain successively. The topmost filter, *i.e.*, the thread filter, determines the thread to handle this request. In the *thread-pool* model, the thread filter enqueues the request into a queue serviced by a thread with the appropriate priority. This thread then passes control back to the ORB, which performs operation demultiplexing and dispatches the upcall (**4**).

• **Advantages:** The main advantage of a threading framework is its flexibility. The thread filter mechanism can be programmed by server developers to support various multi-threading strategies. For instance, to implement a thread-per-request [9] strategy, the filter can spawn a new thread and pass the request to this new thread. Likewise, the MT-Orbix threading framework can be configured to implement other multi-threading architectures such as thread-per-object [10] and thread pool [11].

• **Disadvantages:** The disadvantage with a threading framework is that its generality can significantly increase locking overhead. For instance, locks must be acquired to insert requests into the queue of the appropriate thread of a thread pool. The overhead from locking can greatly reduce throughput and increase latency [2].

The Reactor-per-Thread-Priority Architecture

• **Overview:** The `Reactor`-per-thread-priority architecture is based on the Reactor pattern [13], which integrates transport endpoint demultiplexing and the dispatching of the corresponding event handlers. This threading architecture associates a group of `Reactors` with a group of threads running at different priorities. As shown in Figure 9, the components in

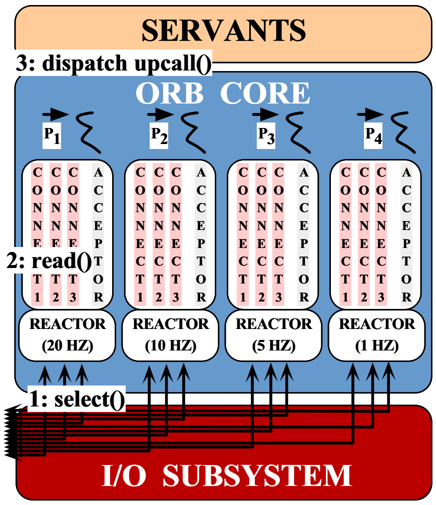

Figure 9: Server-side Reactor-per-Thread-Priority Multi-threading Architecture

the Reactor-per-thread-priority architecture include multiple pre-allocated Reactors, each of which is associated with its own real-time thread of control for each priority level, *e.g.*, $P_1 \ldots P_4$, in the ORB. For instance, avionics mission computing systems [14] commonly execute their tasks in fixed priority threads corresponding to the *rates*, *e.g.*, 20 Hz, 10 Hz, 5 Hz, and 1 Hz, at which operations are called by clients.

Within each thread, the Reactor demultiplexes (**1**) all incoming client requests to the appropriate connection handler, *i.e.*, connect_1, connect_2, etc. The connection handler reads (**2**) the request and dispatches (**3**) it to a servant that executes the upcall at its thread priority.

Each Reactor in an ORB server thread is also associated with an Acceptor [15]. The Acceptor is a factory that listens on a particular port number for clients to connect to that thread and creates a connection handler to process the GIOP requests. In the example in Figure 9, there is a listener port for each priority level.

• **Advantages:** The advantage of the Reactor-per-thread-priority architecture is that it minimizes priority inversion and non-determinism. Moreover, it reduces context switching and synchronization overhead by requiring the state of servants to be locked only if they interact across different thread priorities. In addition, this multi-threading architecture supports scheduling and analysis techniques that associate priority with rate, such as Rate Monotonic Scheduling (RMS) and Rate Monotonic Analysis (RMA) [16, 17].

• **Disadvantages:** The disadvantage with the Reactor-per-thread-priority architecture is that it serializes all client requests for each Reactor within a single thread of control, which can reduce parallelism. To alleviate this problem, a variant of this architecture can associate a *pool* of threads with each priority level. Though this will increase potential parallelism, it can incur greater context switching overhead and non-determinism, which may be unacceptable for certain types of real-time applications.

The TAO real-time ORB uses the Reactor-per-thread-priority architecture.

Concluding Remarks

It is hard to program multi-threaded applications, particularly servers, since developers must ensure that access to shared data is serialized properly. Moreover, the techniques required to control and terminate threads are complicated, non-portable, and non-intuitive. In addition, not all platforms provide good support for threads or thread-aware debuggers.

When the complexities of concurrency control and synchronization are handled by an ORB, however, the benefits of multi-threading often outweigh the disadvantages. In particular, multi-threaded ORBs can yield simpler servant implementations than single-threaded ORBs. Much of this simplicity derives from the fact that request scheduling and operation dispatching are handled by the ORB, rather than the application. Therefore, servers need not be concerned with the duration of each operation they execute.

In addition, when designed and used properly, multi-threaded ORBs can improve the efficiency and predictability of client and server applications. For instance, multiple client requests can be serviced simultaneously by one or more CORBA servants. Each servant can be processed in a separate thread of control, which can be mapped to CPU and executed in parallel on multi-processor platforms.

An increasing number of commercial and research CORBA ORBs implement one or more of the multi-threading architectures described in this article. By evaluating the properties of each architecture, CORBA developers can make better choices for their distributed applications. Empirical benchmarks are ultimately the best way to determine how well ORBs implement these architectures perform in practice. In particular, there is a significant difference between ORBs that use user-level threads and kernel-level threads. Moreover, some ORB run-time systems do not use kernel-threads explicitly, and rely on their own schedulers and interrupt mechanisms. An ORB benchmarking test suite that compares the performance of various ORB multi-threading architecture is freely available at www.cs.wustl.edu/∼/schmidt/TAO.html.

Acknowledgements

Thanks to Lewis Stiller <stiller@franz.com> for his helpful comments on this paper.

References

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

[2] D. C. Schmidt, S. Mungee, S. Flores-Gaitan, and A. Gokhale, "Alleviating Priority Inversion and Non-determinism in Real-time CORBA ORB Core Architectures," in *Proceedings of the 4^{th} IEEE Real-Time Technology and Applications Symposium*, (Denver, CO), IEEE, June 1998.

[3] J. Eykholt, S. Kleiman, S. Barton, R. Faulkner, A. Shivalingiah, M. Smith, D. Stein, J. Voll, M. Weeks, and D. Williams, "Beyond Multiprocessing... Multithreading the SunOS Kernel," in *Proceedings of the Summer USENIX Conference*, (San Antonio, Texas), June 1992.

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

[5] E. Eide, K. Frei, B. Ford, J. Lepreau, and G. Lindstrom, "Flick: A Flexible, Optimizing IDL Compiler," in *Proceedings of ACM SIGPLAN '97 Conference on Programming Language Design and Implementation (PLDI)*, (Las Vegas, NV), ACM, June 1997.

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

[7] M. Henning, "Binding, Migration, and Scalability in CORBA," *Communications of the ACM special issue on CORBA*, vol. 41, Oct. 1998.

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

[9] D. Schmidt and S. Vinoski, "Comparing Alternative Programming Techniques for Multi-threaded CORBA Servers: Thread-per-Request," *C++ Report*, vol. 8, February 1996.

[10] D. Schmidt and S. Vinoski, "Comparing Alternative Programming Techniques for Multi-threaded CORBA Servers: Thread-per-Object," *C++ Report*, vol. 8, July 1996.

[11] D. Schmidt and S. Vinoski, "Comparing Alternative Programming Techniques for Multi-threaded CORBA Servers: Thread Pool," *C++ Report*, vol. 8, April 1996.

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

[13] D. C. Schmidt, "Reactor: An Object Behavioral Pattern for Concurrent Event Demultiplexing and Event Handler Dispatching," in *Pattern Languages of Program Design* (J. O. Coplien and D. C. Schmidt, eds.), pp. 529–545, Reading, MA: Addison-Wesley, 1995.

[14] 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.

[15] 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.

[16] C. Liu and J. Layland, "Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment," *JACM*, vol. 20, pp. 46–61, January 1973.

[17] M. H. Klein, T. Ralya, B. Pollak, R. Obenza, and M. G. Harbour, *A Practitioner's Handbook for Real-Time Analysis: Guide to Rate Monotonic Analysis for Real-Time Systems*. Norwell, Massachusetts: Kluwer Academic Publishers, 1993.