

Reflective ORBs: Supporting Robust, Time-critical Distribution

Ashish Singhai, Aamod Sane, and Roy Campbell

University of Illinois, Department of Computer Science
1304 W. Springfield Ave., Urbana IL 61801 USA
<http://choices.cs.uiuc.edu>

1 Introduction

Modern applications of computers such as video-on-demand require real-time response and need distributed implementations. Object Request Brokers (ORBs) [9] provide a solution to the distribution problem by allowing method invocation on remote objects. However, mere remote method invocation is not enough in a distributed setting: application programs also require features like fault-tolerance and load-balancing. Integrating all possible functionality into an ORB would result in a complex, monolithic program, so we need a modular architecture for ORBs. In this paper, we show how reflection enables the construction of a composable ORB that can be customized to support new features.

After reviewing Common Request Broker Architecture (CORBA [9]), we discuss reflection [5] and its application to ORBs. Then we consider the requirements for a real-time ORB and show how we use reflection to build real-time support in our ORB. Later we address other services like fault-tolerance and show that systematic application of reflection can result in a plug and play system. We conclude with initial performance results and discussion of future work.

2 Standard ORBs

A minimal ORB architecture (Fig. 1) has the following elements. (The names in the parentheses refer to Object Management Group (OMG) terminology.)

- Servers (Implementation): Servers implement desired behavior. The ORB facilitates client access to the servers.
- Object References (ObjRef): Remote representations of objects that enable communication with servers.
- Client-side Invocation (Stubs): Stubs located on the client side package and transmit the method invocation (method-id, parameters) to the server.
- Server-side Invocation (Skeleton): The server-side ORB receives the client request and calls the server. The “Skeleton” objects perform this function.

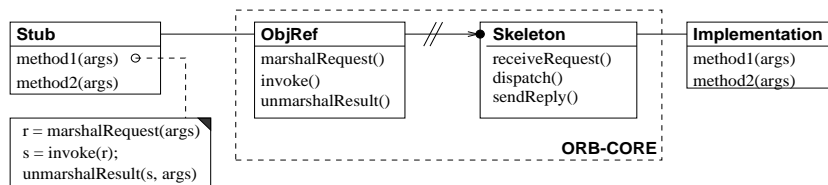


Fig. 1. A minimal ORB architecture

A client application gets an object reference to a server through the ORB. Servers register themselves with the ORB to allow clients to access them. Thus, the primary user interface to the ORB¹ consists of methods for registering and querying about services.

We provide an extended interface that allows applications to adjust various aspects of the ORB. These include method dispatch, memory and concurrency management, object creation and destruction, object reference management and marshaling. The resulting reflective architecture supports a variety of features without changing the basic ORB architecture.

3 Reflection in ORBs

A reflective system gives a program access to its definition and the evaluation rules and defines an interface for altering them. In an ORB, client method calls represent the ‘program’, the ORB implementation the ‘evaluator’ and evaluation is just method invocation. A reflective ORB lets clients redefine the evaluation semantics.

For instance, in a real-time ORB, a client must transmit method completion deadlines to the server. The server uses the deadline to schedule the method call. Except for scheduling the method call and (un)marshaling the deadline, rest of the ORB remains unaffected.

Our reflective ORB accommodates these changes by reifying method call processing in the form of *Invoker* and *Dispatcher* objects. Client programs supply a subclass of *Invoker* that knows about marshaling with deadlines. This provides a reflective control interface allowing application programmers to interact with the reified objects and modify the functionality. Figures 2 and 3 show the client-side and server-side architecture, respectively, of our ORB with the reified objects.

As another example, consider incorporating fault-tolerance by replicating client method calls and merging their responses. In our ORB, the *Invoker* objects control method dispatch on the client-side. Applications can install a specialized *Invoker* that replicates calls and merges results.

¹ In the CORBA parlance, the application environment contains a *pseudo object reference* to the ORB. This reference may be used to access naming services that return object references.

4 Real Time Support in ORBs

Real time support in an ORB entails the following requirements.

- Timing: For all methods, we must know the estimates for the worst case execution time, memory requirements and I/O bandwidth requirements. For all method calls, clients must indicate start and end deadlines.
- Priority: The ORB must support prioritized use of resources. In particular, blocking and synchronization must preserve priorities [1].

These requirements mean changes to service registration, access to object references and method execution.

- Service Registration: Services must express estimated resource requirements (memory, threads, I/O bandwidth) to the ORB.
- Access to Object References: In order to gain access to a service, clients specify parameters, e.g., priority, type of service (periodic, aperiodic.)
- Method Invocation: Client method calls have to transmit additional parameters such as deadlines. This changes the format of the network packets.
- Method Execution: Servers schedule incoming client method calls depending on the service (periodic or deadline driven.) Therefore, servers must provide their own scheduling policies.

The Reflective Interface The reflective interface of our ORB allows applications access to method invocation, execution, and registration components to implement these changes.

- On the client side, programmers can set *Marshaler* and *Invoker* objects on a per-service basis. Thus programs may add support for deadlines.
- Clients can specialize *Invoker* objects to handle rejected invocations (e.g., by retrying with a more relaxed deadline.)
- *Dispatcher* objects optionally interpose schedulers with different policies in the dispatch path.

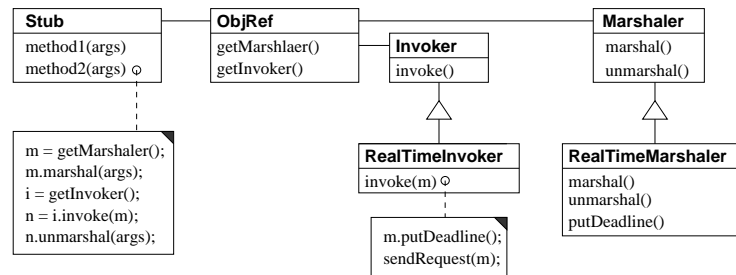


Fig. 2. Client-side ORB architecture for our RT-ORB

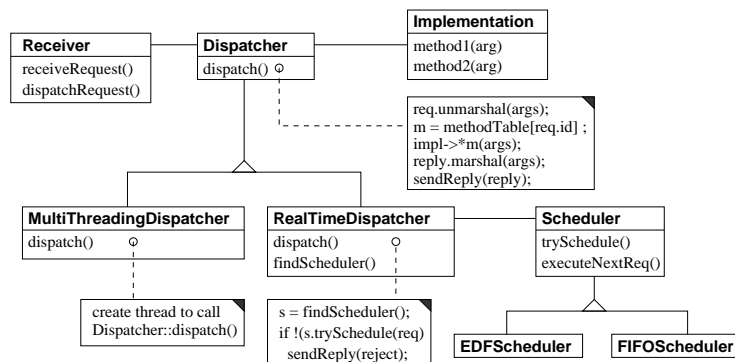


Fig. 3. Server-side ORB architecture for our RT-ORB

Operating System Interface Besides the changes in the external ORB interface, a real-time ORB also requires a different interface from the operating system.

- To facilitate controlling resource utilization, the operating system should support page locking, thread and process priority control, real-time timers [3].
- Operating system supported synchronization constructs within the server-side ORB require new queuing disciplines that prevent priority inversion [1].
- Server scheduling policies such as rate monotonic scheduling [7] require operating system support for fixed priorities and preemption.

Our ORB accesses operating system services provided by the ACE [11] library. While our ORB is predictable in its memory and CPU usage, it can support real-time applications only if the operating system provides required facilities.

5 Supporting Fault-Tolerance and Load Balancing

We implement fault-tolerance and load balancing using the same reflective interface developed for real-time support.

Load Balancing Load balancing is implemented by distributing client requests among several servers [12, 14]. Recall that we changed the client-side ORB method `dispatch` to add deadlines (by changing the *Invoker* object, Sec. 4.) In the present case, we change the method `dispatch` to send the client requests to one among a group of compatible servers. The rest of the ORB is oblivious to this change.

Fault-Tolerance We implement fault-tolerance as a variation of load-balancing. We change method `dispatch` to send client requests to all servers in a group of replicated servers. In addition, the client side ORB merges multiple replies to present a single reply to the client.

Apart from changing method `dispatch`, we create *ClientInteraction* and *ServerInteraction* interfaces to implement policies for managing server groups, evaluating load metrics and failure detection.

6 Preliminary Performance Results and Status

Our ORB currently supports basic object distribution, real-time scheduling of client calls, and simple fault-tolerance by replicating objects. The following table shows the timing for null method calls on the basic ORB, ORB with real-time scheduling using Earliest Deadline First (EDF) and First In First Out (FIFO) algorithms, and a server duplicated on two machines: These numbers are averaged over 5000 iterations of null calls, with a single client (to avoid contention.)

ORB type	Time (μs)
Basic	1962
Real Time (EDF)	3172
Real Time (FIFO)	3227
Replication	2616

Figure 4 exhibits the results from an experiment, which changes scheduling policies at run-time. The first phase uses the EDF policy. The second and third phases change it to FIFO and NONE (No Scheduling) respectively. In each phase, we have 10 clients, 5 of them requiring 3 seconds of processing time every t seconds, and the other 5 requiring 7 seconds of processing time every t seconds. We adjust t to change the targeted CPU utilization. EDF scheduling [2] performs the best. FIFO and NONE exhibit similar behavior at lower CPU utilizations since enough capacity is available. But with increasing contention, performance of the NONE scheme deteriorates.

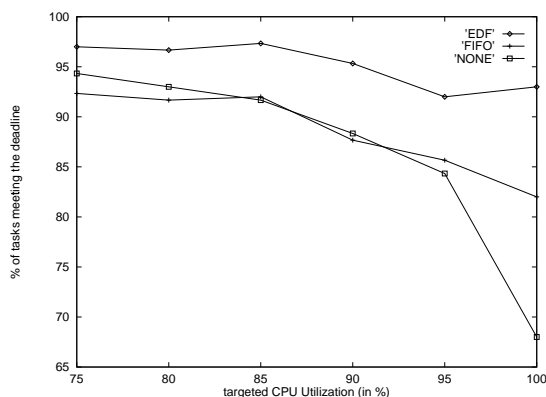


Fig. 4. Performance of various scheduling policies in our RT-ORB.

We plan to use the adaptive capabilities of the ORB to build a video server, operating in a dynamic, heterogeneous environment. Using the reflective capabilities of the ORB, it will use different communication protocols and adapt to varying network conditions.

7 Related Work

The OMG special interest group on Real-Time CORBA is studying the issues involved with real-time processing in CORBA; but there is no concrete specification from OMG yet. The Electra [8] ORB supports fault-tolerance using reliable multicast. Instead of using reflection to implement fault-tolerance, it introduces the notion of “group object reference” as a separate construct and uses the group communication facilities of the underlying system (ISIS or HORUS.) No commercial ORB has any real-time features to the best of our knowledge.

Real-time method invocations in CORBA has been considered by Wolfe, et. al. [15]. Their approach involves transmitting the timing information using the *context* field from the CORBA specification. Takashio, et. al. [13], have mentioned Time Polymorphic Invocations (TPI) using their Distributed Real-Time Object (DRO) model. They assume existence of multiple implementations of the same method with different resource requirements and at run-time execute the one that is feasible according to the timing constraints.

Honda and Tokoro [4] develop a language with timing specifications and schedulers at the meta level. This is similar to the way we reify marshaling, unmarshaling and scheduling using *Invoker*, *Marshaler*, and *Dispatcher*.

Schmidt, et. al. [10], develop an architecture for real-time additions to CORBA. They discuss optimizations for high performance, the development of a real-time inter-orb protocol and real-time scheduling. However, their architecture does not appear to be explicitly targeted to support fault-tolerance and load-balancing.

8 Conclusion

We have shown how reflection allows us to build a modular ORB that may be customized to support real-time processing, fault tolerance, and load balancing. Reflective facilities created for one feature help us in supporting additional features without drastically changing the initial architecture. We support changes by reifying the structure and evaluation strategy in the ORB; as a result, changing the ORB amounts to creating new subclasses and using the corresponding objects, so system performance is practically unaffected.

In our ORB, we use reflection in a limited way. The underlying language and the fine-grained features of the ORB are not reflective. We explicitly choose the entities we reify, and it is possible that additional changes will need new objects.

Moreover, the changes we consider largely leave the semantics of the programs unchanged. Real time processing has the most impact on the ORB because it changes the semantics to the greatest degree. For instance, we must expose the synchronization structure of the ORB and allow users to change the queue disciplines. Fault-tolerance and load balancing required relatively benign changes.

This mild form of reflection has been practised in many domains with considerable success [5]. It is interesting to speculate about language support for such limited reflection. A language with explicit support for frameworks might allow programmers to specify the parts of the framework structure and processing that would be reified. Support for Aspects [6] would achieve the same effect.

Acknowledgements We gratefully acknowledge the help provided by Mallikarjun Shankar, Amitabh Dave and Zhigang Chen.

References

1. Özalp Babaoglu, Keith Marzullo, and Fred B. Schneider. A formalization of priority inversion. *Real-Time Systems*, pages 285–303, 1993.
2. Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. A task scheduling problem. In *Introduction to Algorithms*, chapter 17.5. The MIT Press, 1992.
3. Bill O. Gallmeister. *POSIX.4: Programming for the Real World*. O'Reilly & Associates, Inc., 1995.
4. Y. Honda and M. Tokoro. Time-dependent programming and reflection: Experiences on R2. Technical Report SCSL-TR-93-017, Sony CSL, 1993.
5. Gregor Kiczales. Towards a new model of abstraction in software engineering. In *Proc. IMSA '92 Workshop on Reflection and Meta-level Architectures*, 1992.
6. Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. Technical Report SPL97-008 P9710042, XEROX PARC, 1997. <http://www.parc.xerox.com>.
7. C. L. Liu and J. W. Layland. Scheduling algorithms for multiprogramming in hard real-time environment. *Journal of the ACM*, 20:46–61, 1973.
8. Silvano Maffei and Douglas C. Schmidt. Constructing reliable distributed communication systems with CORBA. *IEEE Communications Magazine*, 14(2), 1997.
9. Object Management Group. *The Common Object Request Broker: Architecture and Specification*, 1996. Document PTC/96-08-04, Revision 2.0.
10. Douglas C. Schmidt, Aniruddha Gokhale, Timothy H. Harrison, David Levine, and Chris Cleeland. TAO: a high-performance endsystem architecture for real-time CORBA. (RFI response to OMG-SIG Real-Time CORBA), 1997.
11. Douglas C. Schmidt and Tatsuya Suda. An Object-Oriented Framework for Dynamically Configuring Extensible Distributed Communication Systems. *IEE/BCS Distributed Systems Engineering Journal*, 2:280–293, 1994.
12. Ashish Singhai, Swee Lim, and Sanjay R. Radia. The SCALR framework for internet services. submitted for publication, 1997.
13. Kazunori Takashio and Mario Tokoro. Time polymorphic invocation: A real-time communication model for distributed systems. In *Proc. 1st IEEE Workshop on Parallel and Distr. Real-Time Systems*, 1993.
14. Y. T. Wang and R. J. T. Morris. Load sharing in distributed systems. *IEEE Transaction on Computers*, C-34(3):204–217, 1985.
15. Victor Fay Wolfe, John K. Black, Bavani Thuraisingham, and Peter Krupp. Real-time method invocations in distributed environments. In *Proc. HiPC'95 Intl. Conf. on High-Performance Computing*. IEEE, 1995.