Real-time Method Invocations in Distributed Environments *

Victor Fay Wolfe and John K Black

Computer Science Dept.
University of Rhode Island
Kinston, RI 02881; USA
{wolfe,black}@cs.uri.edu

Bhavani Thuraisingham and Peter Krupp

The MITRE Corporation
Bedford, MA; USA
{thura,pck}@mitre.org

Abstract

Current distributed computing environments, such as the Object Management Group's Common Object Request Broker Architecture (CORBA), do not support real-time requirements. This paper presents the syntax, semantics and support for one necessary feature in a real-time distributed computing environment: timed distributed method invocations. *It presents the general concept and illustrates its application as an extension to CORBA.*

1 Introduction

A major trend in distributed computing is interest in *distributed computing environments*, such as the Open Software Foundation's (OSF's) Distributed Computing Environment (DCE) [1], the Object Management Group's (OMG's) Common Object Request Broker Architecture (CORBA) [2], the Arjuna system [3], and others. These environments provide a way for heterogeneous components to interoperate in a distributed system. Commercial distributed computing environments are adequate for applications such as document management and financial record keeping. However, current distributed computing environments are not well-suited for *real-time applications*. In a real-time application there are timing constraints that must be met for the application to be correct. Automated factory control, avionic navigation, military target tracking, and financial transactions are examples of such applications. These applications would benefit from heterogeneous component interoperability provided by common distributed computing environments, if these environments could support the real-time requirements of the applications.

Unfortunately, current commercial distributed computing environments offer little support for real-time requirements. For example, the languages used to describe interfaces to components in distributed environments, called *interface design languages* (IDLs), describe the interface to the functional behavior of distributed components, but do not explicitly describe timing their behavior. Furthermore, system services provided by the distributed environments offer little support for end-to-end real-time scheduling across the environment. In fact, some environments do not offer such basic services as synchronized clocks and bounded message latencies: these are essential in a distributed real-time applications.

We have been studying which features must be included in a distributed computing environment to support real-time requirements [4]. Since most distributed computing environments, like DCE and CORBA, use a client/server model with a remote procedure call (RPC) paradigm for connection, we started by defining the semantics and system support required for a timed RPC in a client/server model. Other work has investigated the semantics of timed synchronous communication [5, 6], but has not addressed the requirements of providing timed RPC in distributed computing environments. This paper presents our results including: expression of timing constraints on RPCs, support for establishing end-to-end timing constraints and scheduling parameters, and a list of necessary distributed computing environment services. Although these results can be applicable to general distributed computing environments, for concreteness we chose to describe timed RPC extensions to CORBA. Since CORBA is based on an object model, we call our timed RPC extensions *timed distributed method invocations*.

*This work is partially supported by the U.S. National Science Foundation, The U.S. Office of Naval Research, the U.S. Naval Undersea Warfare Center, and The U.S. Naval Research and Development Laboratories (NRaD). The views and conclusions expressed in this paper are those of the authors and do not reflect the polciies of the MITRE Corporation.

2 CORBA

CORBA is designed to allow a programmer to construct object-oriented programs without regard to traditional object boundaries such as address spaces or location of the object in a distributed system. That is, a client program should be able to invoke a method on a server object whether the object is in the client's address space or located on a remote node in a distributed system.

Two major components of CORBA are its Object Request Broker (ORB), and its Interface Definition Language (IDL). The CORBA ORB essentially enables communication between clients and remote server object implementations. The server object's source code implementation is the code and data that actually implements the object. The ORB provides all *services* that: locate a server object implementation for servicing a client's request; prepare the object implementation to receive the request; and communicate the data making up the request.

CORBA IDL is a declarative language that describes the *interfaces* to server object implementations, including the signatures of all server object methods that are callable by clients. The IDL grammar is a subset of ANSI C++ with additional constructs to support the method invocation mechanism. IDL also specifies C++-like exception raising and handling. IDL *does not* provide syntax for implementing methods: an IDL binding to the C language has been specified for that purpose. Other language bindings are being developed.

Consider an object that acts as a shared table for sensor data (represented as long integer values) for clients in a distributed system. An example of simple CORBA IDL for a *sensor_table* object is:

```
interface sensor_table {
  readonly attribute short max_length;
  short put_data(in short index,in long data);
  long get_data (in short index); }
```

The IDL keyword `interface` indicates a CORBA object (similar to a C++ *class* declaration). A `readonly` attribute is a data value in the object that a client may read (the IDL compiler generates a method for reading each attribute). The IDL example also specifies two methods: *put_data*, which stores a sensor value at a index into the table; and *get_data* which returns a sensor value given an index.

Client code to access a *sensor_table* object in a CORBA environment might look like:

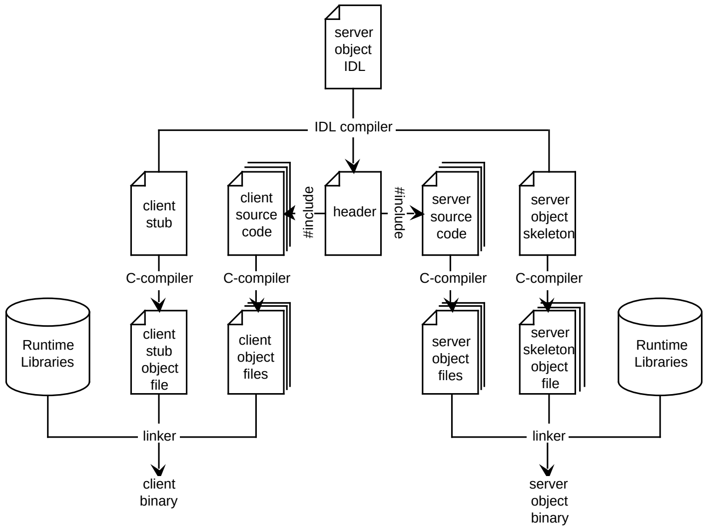

Figure 1: CORBA IDL Compilation Process

```
long retval;
sensor_table *p;
p = bind("my_sensor_table");
retval = p->get_data(500);
```

Here, the client declares a pointer, *p* to a *sensor_table* object called *my_sensor_table*. The client then makes a call to an ORB service to locate and bind the pointer to a reference to a remote server containing the *sensor_table* object. To retrieve a value from the *sensor_table* at index 500, the client issues the method invocation: `p->get_data(500)` . The *sensor_table* server would have previously had to have implemented and registered with the CORBA ORB.

The process of implementing a client and server object is shown in Figure 1. The IDL specification is processed by an IDL compiler, which generates a header file for the CORBA object, stub code for linking into the client, and skeleton code for the server object. The client stub contains code that hides details of interaction with the server from the client code. Client stubs stand in for normal method calls by transparently directing normal-appearing C++ method requests into the ORB. Server skeleton code is used by the ORB in forwarding method invocation requests to the server, and in returning results to the client.

3 Timed Distributed Method Invocation

This section describes the basic syntax and semantics of timed method invocations in a distributed computing environment. We establish five forms of *client-side timing constraints*: *deadlines*, *earliest start times*, *latest start times*, *periodic*, and *maximum execution time* constraints for method invocations. We also establish *server-side timing specifications* on the server's interface to indicate the expected execution times of the server methods. These specifications can include minimum, average, and maximum execution time for each method. In addition, we describe an exception mechanism to handle violations of the timing constraints. We also describe the support that must be present in a distributed computing environment to realize these semantics. The presentation of timed distributed method invocation is general, but the examples are done using a CORBA environment.

3.1 Expressing Timing Constraints

To specify timing values, we take advantage of CORBA IDL's ability to define new types from existing types (*typedef*) to define the *timespec* type. A *timespec* is a structure with two long integer fields: `tv_sec` for seconds, and `tv_nsec` for nanoseconds (it conforms to the POSIX 1003.1 standard specification). The *timespec* type would be pre-defined using a CORBA IDL *typedef* in the header file produced by the IDL compiler.

To express timing constraints for timed distributed method invocation, we use the `context` declaration that is native to CORBA's method declaration [2]. A context is an object that the ORB attaches to each method invocation. The client may add fields and values to the context using the native CORBA `set_one_value` call to the ORB's context interface. This call sets *fields* in the context to corresponding specified values. Fields within the context are available to the ORB and to the server. For real-time, we establish four predefined context fields (using the CORBA naming convention): `CORBA::_after`, `CORBA::_before`, `CORBA::_by`, and `CORBA::_execute`. The client setting these fields in the context establishes timing constraints for all subsequent method invocations. The client setting the `CORBA::_after` context field expresses the constraint that the start time of following distributed method invocations (on the server measured by the server's clock) must be after the time specified by the field. The client setting the `CORBA::_before` field specifies that if the distributed method invocation has not started on the server by the time specified by the field (measured on the server's clock) an E_START exception is raised in the client. The client setting the `CORBA::_by` context field specifies that if the client has not received return values from the invocation by the time specified in the field (measured on the client's clock), then an E_DEADLINE exception is raised in the client. The client setting the `CORBA::_execute` context field specifies that if a method invocation executes on the server for longer than the specified value, then an E_EXECUTE exception should be raised in the client. Periodic behavior involves the client invoking the same method at regular intervals. To do this, the client can use a loop to establish a regular series of fixed-duration time intervals called *period frames*, where the beginning of period frame i is the end of period frame $i-1$. The client uses an earliest start time to constrain the method invocation to start executing after the beginning of its frame and a deadline to constrain it to complete by the end of the frame.

Example Timing Constraint. Consider time-constraining method invocations on a *sensor_table* object from Section 2. The IDL compiler would ensure that the following extension is made the IDL declaration:

```
long get_cell(...);
context (CORBA::_after, CORBA::_before
         CORBA::_by, CORBA::_execute);
```

Note again that the `context` specification is standard CORBA IDL and uses the predefined real-time fields. To specify a deadline of 10 seconds from the time the client executes the call, the client first executes the standard CORBA call to set a value in the client context:

```
context.set_one_value (CORBA::_by, NOW.tv_sec+10);
```

NOW() is a direct interface to the distributed computing environment's Global Time Service which we discuss in Section 3.2.1. *NOW()* returns a *timespec* value representing the current time. In general, timing constraint values can be formed using arithmetic operations, maximum functions, and minimum functions involving *timespec* values and variables. The client can set context values for earliest and latest start times and worst case execution time similar to its setting of the deadline in this example.

Server-Side Timing Specifications. The server object's IDL contains *server-side timing specifications* of execution times for its methods. In CORBA IDL, these execution times would not require any extensions, only a programming convention that these times be specified as `readonly` attributes of the object. For example, the CORBA IDL for the *sensor_table* object of Section 2 could include:

```
interface sensor_table
  readonly timespec get_data_wc_etime;
  long get_data(in short index);
```

which specifies the worst case execution time for method *get_data*. An attribute whose name has "*wc_etime*" appended to an existing method name indicates to the IDL compiler that the attribute represents a worst case execution time for that method. Such a name carries the semantics that an E_EXECUTE exception should be raised in the server if the named method executes for longer than the specified time. These semantics support real-time analysis that depends on execution times of methods. The specification of execution times as attributes also allows the client to ascertain the timing properties of methods before executing them. For example, the client might want to test if there is enough time before its deadline to execute a distributed method before it actually calls the method. The execution time values can be inserted by the object implementor or it may be possible to use automated techniques to generate them [7].

3.2 Implementing Timing Constraints

Implementing the semantics of timed distributed method invocations imposes several requirements on the distributed computing environment. In particular, client-side timing constraints require support for *end-to-end* real-time scheduling. This support involves the automatic generation of intermediate timing constraints and the use of real-time scheduling at each step in the computing environment's distributed method invocation sequence shown in Figure 2.

3.2.1 Required Distributed Computing Environment Services

Distributed computing environment services are implemented by cooperation among the distributed nodes in the environment. Typical environments such as those conforming to the current CORBA and DCE specification provide at least distributed name service and remote procedure or method invocation. To support timed distributed method invocations, the following additional services are required.

Global Time Service. The computing environment must ensure that all clocks in the system are synchronized to within an ϵ skew of each other. This service is provided in the DCE environment [1] and has been proposed by the OMG for the CORBA specification. Clients and servers must be able to call this service to get the "current global time".

Real-Time Scheduling of Services. It is important that in *each* step of the distributed method invocation real-time scheduling is used. We do not specify this scheduling, only that it be based on a *priority* associated with the method invocation. A priority is an ordinal value specifying the relative scheduling position of a task. Most real-time scheduling algorithms are based on priority and differ in how that priority is assigned. For instance, rate monotonic scheduling assigns higher priorities to tasks with shorter periods. Earliest-deadline-first (EDF) scheduling assigns higher priorities to tighter deadlines [8]. Real-time requires that execution at each step in a distributed method invocation have an associated priority and that scheduling decisions, such as all queueing of service requests, enforce priority ordering.

Global Priority Service. In order for priorities to be consistent relative to all other requests in the distributed environment, the priority must be assigned globally. That is, a priority must be relative to *all* other tasks in the distributed environment. To establish such priorities, the environment must provide a Global Priority Service that accepts implementation-defined parameters for a task and returns a globally-relative priority value. For instance, an environment may implement EDF scheduling across its distributed system. In this case, the Global Priority Service would accept a task's deadline and return a priority that ensures that the task will receive all services before other tasks with looser deadlines.

Bounded Message Latency. The remote method invocation service of the environment must ensure a worst case bound of δ for the time it takes from sending of a message to the time the message arrives in the receiver's message buffer. This bound is necessary for establishing intermediate timing constraints, as discussed later.

3.2.2 Client Compilation

Let us examine what a CORBA IDL compiler for a C or C++ language client on POSIX-compliant operat-

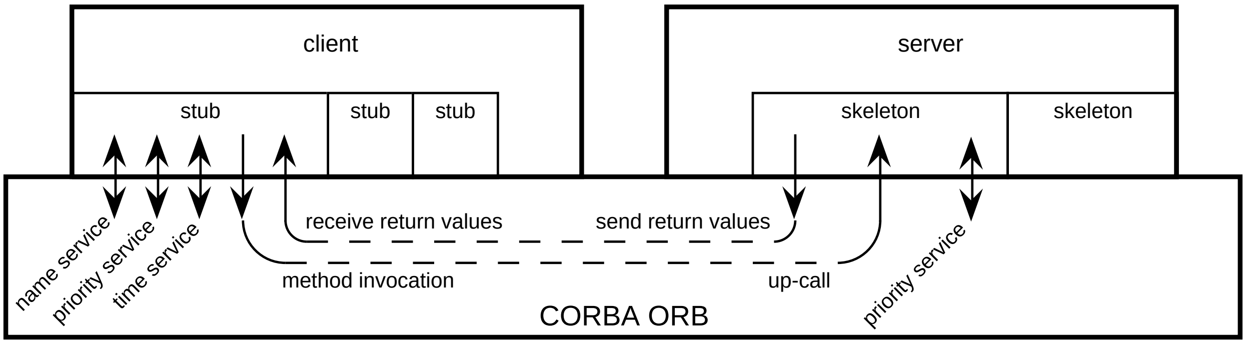

Figure 2: Typical Method Invocation Steps

ing system could produce for a the method *get_data* in our example. First, the IDL compiler would insert into the stub a call to the distributed environment's Global Priority Service to establish the priority of the client's method invocation request. Most likely, the input parameters to this Global Priority Service request would involve parameters from the real-time part of the client's context.

If the `CORBA::_AFTER` context field is set with time t_a, then the ORB communicates this context field to the server. The server then must delay (e.g. sleep) until its interpretation of t_a. The IDL compiler inserts client code into the stub that recognizes a `CORBA::_BEFORE` field set to value t_{bef}. This stub code requests a POSIX alarm at time $t_{bef} + \delta$. The IDL compiler also inserts stub code, after the code that sends a request message to the server, to wait for an acknowledgement from the server that the method invocation was started. The δ term allows for the message latency of the acknowledgement from the server to the client. Upon receiving the acknowledgement, the code inserted into the stub cancels the alarm. If the alarm is triggered before the acknowledgement arrives, the alarm sends a signal causing the client to raise the E_START exception. The IDL compiler inserts stub code that recognizes that the `CORBA::_BY` flag is set to time t_{by}. This stub code sets an alarm at time t_{by}. The IDL compiler also inserts stub code such that if the return message from the method invocation is received before t, the alarm is cancelled. Otherwise, the alarm sends a signal that raises an E_DEADLINE exception in the client. The ORB passes the `CORBA::_execute` value to the server. The server sets a virtual timer (a POSIX capability that measures time on the CPU) that expires at the maximum specified execution time and sends an E_EXECUTE exception to the client.

3.2.3 Server Compilation

Timed distributed method invocation requires that skeleton methods produced for the server by the IDL compiler also support acceptance of the client's timing constraints. For a client's latest start time constraint s_{client}, recall that the client needs an acknowledgement that the method has started by $s_{client} + \delta$ on the client's clock. The server uses the latest start time constraint: $s_{server} = s_{client} - \epsilon$ to pessimistically allow for an ϵ clock skew between itself and the client. For a deadline constraint d_{client}, recall that the return message must be received by the client at d_{client} measured on the client's clock. The server uses the deadline $d_{server} = d_{client} - \delta - \epsilon$ to pessimistically allow for δ message delivery time and an ϵ clock skew.

For worst case method execution times (specified in the IDL by appending "*wc_etime*" to a method declaration), the IDL compiler inserts code into the skeletons to set a virtual timer which would raise the E_EXECUTE exception in the server if it executes too long.

3.2.4 Example Timed Method Invocation

Consider the earlier example of making a call to retrieve data from the sensor table within a 10 second deadline. After the *sensor_table* object is processed by the IDL compiler, the client is compiled in its host language and linked with the IDL-produced stubs. The server code is written and compiled for its skeletons. The server is then executed, registered with the ORB, and awaits requests.

Assume that the client's *get_data* call is initiated at time 10:00, the environment's clock skew ϵ is 0.1sec, the environment's maximum message latency is 0.2sec and the client for *get_data* establishes the deadline `CORBA::_by` as 10:00:10 in its context. The stub code establishes the client's priority, based on `CORBA::_by`, by making a call to the ORB's Global Priority Service. The local operating system scheduler uses this priority to schedule the client's execution during the method invocation of *get_data*. In addition, all subsequent calls to the ORB, such as locating the server by name, use the client's priority when handling requests. The ORB (actually the stub code) places the context containing the deadline along with the actual parameter value (500) into the message that it sends to the server. The client's stub code also sets an alarm for its deadline at 10:00:10. The ORB uses the native network protocol (with the client's priority, if priority is supported in the network protocol) to transmit the message to the server's message buffer.

Assume that server receives the message at time t_{recv} = 10:00:00.2 (after the maximum message latency $\delta = 0.2$). When the server receives the message, it uses the time $d_{server} = (\ \texttt{CORBA::\_by} - \delta - \epsilon) = 10{:}00{:}10 - 0.2 - 0.1 = 10{:}00{:}09.7$ to establish its local deadline. The server uses d_{server} to call the Global Priority Service to set its priority. This priority is used both in scheduling the server on its local node and for handling server requests for ORB services. The server then executes the skeleton method for *get_data*, which in turn makes the up-call to execute the actual *get_data* method on the *sensor_table* object. Upon completion of the *get_data* method invocation, the skeleton code for *get_data* sends a return message to the client. When the return message arrives at the client, the client disables the alarm it had set for deadline 10:00:10. If the message does not arrive by 10:00:10, the alarm signal raises the E_DEADLINE exception in the client.

4 Conclusion

This paper has presented the syntax, semantics, and system support required for timed distributed method invocation – an important first step to realizing support for real-time requirements in distributed computing environments. We used the CORBA distributed computing environment as an example.

The current CORBA specification does not have to be altered, just augemented. A CORBA implementation's IDL compiler must be capable of producing stub and skeleton code that can handle timing block semantics. A real-time extension to CORBA must mandate a Global Time Service and Global Priority Service from its ORB. It further must mandate priority-based scheduling for all resource contention and bounded maximum message delays from its implementations.

There are many further extensions to distributed computing environments that would facilitate supporting real-time requirements. These include real-time concurrency control of method execution on server objects, hard guarantees of service times across the environment, guarantees of minimal inter-arrival time for server requests, interface-level support for multi-threading, and a global *event* service to provide a repository of times of occurrence for globally-named events (to allow specification of timing constraints relative to these events). We are currently studying these issues. We have also initiated efforts to form a special interest sub-group of the OMG to study incorporation of real-time features into the CORBA specification.

References

[1] Open Software Foundation, *Introduction to OSF DCE*. Prentice Hall, 1992.

[2] Object Management Group, *The Common Request Broker Architecture*. X/Open, 1992.

[3] S. K. Shrivistava, G. N. Dixon, and G. D. Parrington, "An Overview of Arjuna: A Programming Environment for Reliable Distributed Computing." *IEEE Software*, vol. 8, no. 1, pp. 63-73, January 1991.

[4] P. Krupp, A. Schafer, B. Thuraisingham, and V. F. Wolfe, "On real-time extensions to the common object request broker architecture," in *Proceedings of the Object Oriented Programming, Systems, Languages, and Applications (OOPSLA) '94 Workshop on Experiences with the Common Object Request Broker Architecture (CORBA)*, Sept. 1994.

[5] I. Lee and S. Davidson, "Adding Time to Synchronous Process Communications," *IEEE Transactions on Computers*, August 1987.

[6] T. Baker and O. Pazy, "Real-time features for Ada 9x," in *IEEE Real-Time Systems Symposium*, Dec. 1991.

[7] W. Pugh and T. M. (Editors), *Proceedings of the ACM SIGPLAN workshop on language, compiler and tool support for real-time systems*. ACM SIGPLAN, 1994.

[8] C. L. Liu and J. W. Layland, "Scheduling algorithms for multiprogramming in a hard-real-time environment," *Journal of the ACM*, vol. 20, pp. 46–61, 1973.