Parametric Time-Based Dispatching in CORBA Distributed Environments

Sameh M. Elsharkawy
FirstWave S.I.O.N.
11800 Sunrise Valley Drive
Reston, Virginia 20191
sharkawy@ieee.org

Ashok K. Agrawala
Department of Computer Science
University of Maryland
agrawala@cs.umd.edu

Abstract

This paper presents a complete system model for scheduling and dispatching CORBA service requests that have hard as well as soft real-time timing requirements in a distributed environment. The model provides the methodology for expressing and enforcing time-based QoS constraints for real-time CORBA service requests. Therefore, the presented model complements the state of the art in real-time scheduling in CORBA-based systems. The model is based on using the distributed time-based parametric scheduling method. This method is used to verify the feasibility of a distributed periodic hard real-time task set and to generate dispatching calendars that dynamically adapt to run-time environment.

1 Introduction

With the increasing scale and complexity of current distributed applications, Object Oriented (OO) software design methodology is gaining popularity in this field. This is due to the increased flexibility, reusability, and maintainability of object oriented systems. The use of large scale distributed object oriented programming requires seamless interoperability among heterogeneous client and server objects. The Common Object Request Broker Architecture (CORBA), developed by the Object Management Group (OMG), is becoming a renowned standard for object-oriented distributed environments. CORBA is a middleware that establishes the relationship between client and server objects while hiding the implementation and location details of both objects. This simplifies the process of developing more complex and flexible distributed applications and makes the development cycle much shorter.

The CORBA middleware model lacks the specification for services and interfaces to express and enforce end-to-end real-time constraints on client/server interactions. A Real-Time Special Interest Group (RTSIG) was formed by the OMG to layout the requirements for extensions and modifications in the CORBA model to support real-time applications [16]. The RTSIG produced a whitepaper [14] describing the requirements for real-time CORBA architecture and object services. The RT-CORBA specification as well as most of the currently existing RT-CORBA implementations mainly focus on priority based scheduling, whether it is statically or dynamically assigned, without much consideration to time-based QoS requirements.

Some of the current major real-time CORBA projects as described in the OMG’s Request for Information (RFI) on RT CORBA [14, 16] are summarized below:

NRaD URI RT CORBA, developed at the University of Rhode Island and the US Navy NRaD facility. The model provides CORBA Global priority services, real-time event handling, and concurrency control, which provide an environment for carrying out timed distributed method invocations (TDMI) [15, 16, 3].

Attack Submarine CORBA, developed by the Naval Undersea Waterfare Center and Lockheed Martin [7]. This system provided a CORBA system to interconnect Sonar, Radar, Navigation and Combat control subsystems. It introduced a latency server to provide latency estimates for real-time CORBA operations.

MITRE RT-CORBA, developed by Peter Krupp and Bhavani Thuraising at MITRE in Bedford, MA [15, 1]. Their work identified a framework for the use of real-time CORBA in command and control systems. It provided distributed scheduling using rate-monotonic and deadline monotonic techniques.

CHORUS/COOL RT CORBA, introduced by Chorus systems. It provides a real-time ORB (CHORUS/COOL) implemented on top of their RTOS (CHORUS/ClassiX). Most of the real-time services are provided by the RTOS and not by the COOL ORB.
The RT-CORBA specifications mainly target ORB end-system to ensure end-to-end predictable behavior, OS scheduling mechanism, and requirements to be provided and managed by the communications infrastructure. The distributed nodes’ operating systems as well as the underlying network must support real-time constraint enforcement.

In order to achieve timely execution of service requests, the client must have a way to express the required timing requirements, and the CORBA system must provide methods for expressing and enforcing timing constraints. The CORBA specifications mainly target fixed priorities [12]. The fixed or dynamic priority scheduling methods supported by the RT-CORBA does not provide the proper platform for applications with time-based QoS requirements because they cannot guarantee proper timely execution for non-preemptive hard real-time tasks.

The released RT-CORBA specifications describe the requirements to be provided and managed by the communication infrastructure, OS scheduling mechanism, and ORB end-system to ensure end-to-end predictable behavior. The RT-CORBA specifications mainly target fixed priority real-time systems and provide applications with interfaces to manage their resources, thread assignment, and priorities [12]. The fixed or dynamic priority scheduling methods supported by the RT-CORBA does not provide the proper platform for applications with time-based QoS requirements because they cannot guarantee proper timely execution for non-preemptive hard real-time tasks.

The presented real-time CORBA system model provides time-based execution capability for client and server tasks while introducing minimal or no changes to the standard CORBA specifications. Therefore, in conjunction with existing priority-based CORBA scheduling, the presented time-based CORBA scheduling method serves to round out the state of the art in real-time scheduling for CORBA-based systems.

### 1.1 Distributed Parametric Scheduling

The major scheduling method used in this model is the distributed time-based parametric scheduling method. Parametric scheduling was initially introduced by M. Saksena et al [11] and was extended to include periodic tasks by S. Choi [2]. The method was further extended to be used for scheduling and dispatching real-time tasks in distributed environments [5]. The use of parametric time-based scheduling, provides off-line feasibility guarantees to meet timing requirements for hard real-time tasks while the parametric dispatching mechanism maintains a flexible run-time environment that makes use of slack time with limited overhead.

Parametric scheduling methods use Fourier Motzkin variable elimination technique [4] to verify the schedulability of a real-time task set and calculate dynamic calendars for dispatching jobs at run-time. Dynamic calendars represent the start time of each job \( \tau_i \) with two parametric functions \( f_{\min}^{\ominus}, f_{\max}^{\ominus} \) whose evaluation generate the minimum and maximum feasible starting times of that job.

The details of the distributed time-based parametric scheduling method are beyond the scope of this paper. Therefore, the method is used as a black box with known inputs and outputs. The method’s inputs consist of a distributed set of hard real-time non-preemptive periodic tasks with inter-task relative timing and communication constraints. As output, the method verifies the feasibility of the system’s timing requirements, generates dynamic calendars for run-time dispatching, and calculates worst case message delivery times for all communication channels [5].

### 1.2 Distributed Parametric Dispatching

A separate run-time task dispatcher runs on each one of the system nodes. A node dispatcher uses the local node’s calendar to populate its run-time task Dependency graph. The Dependency graph is a graph-like structure that includes all the node’s active task instances along with their timing requirements, and inter-task timing dependencies. The dispatcher maintains a Time Ordered List (TOL) of all ready-to-execute jobs, on the local node, ordered according to their minimum feasible starting time. It also maintains an External Event Queue (EEQ) to handle external events from remote tasks. At run-time, node dispatchers start by synchronizing their local calendars’ starting times. At the unified starting time, node dispatchers start executing their applications’ tasks according to their parametric calendars.

### 2 Problem Description

Our main objective is to develop a complete model for scheduling and dispatching hard and soft real-time service requests in a CORBA-based distributed environment. The model presented consists of two major phases. The off-line system schedulability analysis phase. During this phase, all system’s functional and timing information are aggregated and evaluated for feasibility. The scheduling phase is performed by a central node that is reachable by all the system’s distributed nodes. The second phase is the run-time task execution phase. In this phase, each of the system’s nodes independently executes its local load of tasks in conformance with the global system feasibility. The model uses an agent object at run-time to execute each node’s task dispatching calendar [10, 9].
The environment consists of \( M \) computer nodes connected via a mesh network of point-to-point dual simplex links. A link connecting \( \text{Node}_i \) to \( \text{Node}_j \) is referred to as \( \text{Link}_{i,j} \). A node in the system can have several incoming and outgoing links attached to it, each of which can operate in parallel with the others. Nodes’ clocks do not have to be synchronized, but the clocks’ rates are assumed to be equal. The maximum skew between clocks’ rates on different nodes is assumed to be very small compared to message transfer delays.

\[ \text{Figure 1. Synchronous Service-Request} \]

\[ \text{Figure 2. Asynchronous Service-Request} \]

2.1 Task Model

For a task model to be used in a CORBA-based distributed environment, it has to be based on the client-server Remote Procedure Call (RPC) scheme, which constitutes the main inter-object interaction method in CORBA. At the same time, in order to use the distributed dynamic time-based scheduling method, the task model has to follow the guidelines of the task model supported by that method \cite{5}. Therefore, in this section we present a task model that is based on CORBA’s client-server model, and its mapping to the dynamic time-based scheduling task model.

OO real-time applications consist of a set of distributed objects that reside on the system’s distributed nodes. Applications objects’ interfaces are specified using CORBA IDL. Timing specifications of the objects’ real-time methods and services need to be specified as well. Client objects’ methods are periodically invoked in the context of client tasks. Client tasks invoke methods on server objects by sending service requests, which result in the generation of server tasks. Therefore, on each node, runs a set of real-time client and server tasks. Client tasks invoke methods on the server objects by sending service requests to their server tasks and then receiving return values as responses from them, similar to Remote Procedure Calls (RPC) shown in figures 1 and 2. The task model distinguishes two main task types:

**Hard real-time tasks (HRT):** which are periodic tasks that must be guaranteed to finish their execution before their specified deadlines. Hard real-time tasks must specify their timing requirements prior to run-time for schedulability analysis. All hard real-time tasks’ periods share a least common multiple \( \text{LCM} \) of their execution periods, which is used as a scheduling window that constitutes the task execution pattern to be repeated indefinitely for the lifetime of the application.

**Soft real-time tasks (SRT):** these are periodic or aperiodic preemptive tasks that have timing requirements to be satisfied on a best effort basis.

Hard real-time client tasks can only request services from hard real-time server objects. Similarly, soft real-time tasks can only request services from soft real-time server objects. As scheduling of soft real-time tasks is done on a best-effort basis, the rest of the model description will focus on hard real-time tasks.

The objects considered for scheduling are client tasks and the servers’ response tasks. Client tasks are mainly generated by real-time applications to perform the different computations of the application. Server tasks are generated by CORBA’s object adaptors as a result of real-time service requests to server objects. The resource to be scheduled is the CPU time required to execute services provided by the client and server objects on all nodes. Clients should specify the type and parameters of the Quality of Service (QoS) guarantees required for each of the requested services. The QoS requirements vary according to the type of service method required. There are two types of service requests:

**Synchronous service requests,** where the client task is blocked, waiting for a response from the server object as shown in (Figure 1).

**Asynchronous service requests,** where the client task can resume its execution as soon as the service request has been generated. The client task does not require the server task to return any results in this case. The server task deadline is specified as part of the client’s service request (Figure 2).

Each hard real-time client task needs to specify its timing parameters prior to run-time for the feasibility analysis process. These requirements are represented by:
• Task period $P$: The repetition cycle for the task. One instance of the task is executed in each period.

• Low jitter $\lambda$: The minimum distance between the start times of two consecutive task instances is $P - \lambda$.

• High jitter $\eta$: The maximum distance between the start times of two consecutive task instances is $P + \eta$.

• Minimum execution time $l_c$: The minimum value for the task's execution time $e$.

• Maximum execution time $u_c$: The maximum value for the task's execution time $e$.

• Release time $r_c$: The time at which the task instance becomes ready for execution. It is measured from the start of the task’s period.

• Deadline $d_c$: The maximum feasible finish time for the task measured from the start of the task’s period.

The $l$, $u$, $\lambda$, $\eta$, and $P$ parameters can be left unspecified in case of best effort soft real-time service requests.

Service requests to local server objects’ methods are compiled into the clients’ tasks by the IDL compiler. However, remote service requests are implemented using communication messages and therefore, they would be modeled as communication channels and their timing requirements as communication timing constraints. For the dynamic time-based distributed task model, system task instances are non-preemptive and communication messages can only be generated at the end of system task instances (jobs). In order to use the dynamic time-based task model, periodic client tasks are partitioned into non-preemptive periodic sub-tasks, or jobs, at points where they generate service requests to methods of server objects residing on a node other than their own. Each client task's jobs inherit some of its timing parameters, namely the task’s period of execution, low jitter and high jitter. The remaining parameters (minimum and maximum execution times, release time, and deadline) are calculated by the system’s off-line scheduler as shown in section 4.2. Server tasks, generated as a result of remote service requests, also share the client tasks’ inherited parameters. The rest of their timing parameters are calculated by the off-line scheduler as well. A maximum of two communication channels are created as a result of each service request. One communication channel is created from the calling client job to the server job to deliver the service request call. The second is created from the server job to the next client job to deliver the return value in the case of synchronous service requests. The worst case message delivery times for the call and response communication channels are $q_c$ and $q_r$, respectively. Service requests' task model representation is shown in figure 3. Therefore, a client task that makes $n$ remote service requests to server methods is partitioned into $n + 1$ non-preemptive jobs $c_1, c_2, c_3, \ldots, c_n, c_{n+1}$. It also results in the generation of $n$ server jobs $s_1, s_2, s_3, \ldots, s_n$. At the end of each client job $c_k (k : 1 \rightarrow n)$, the task generates a service request that results in the creation of server job $s_k$ to perform the required service. In the case of a synchronous service request, the call result is delivered to the start of $c_{k+1}$. For each service request the client task generates, it has to specify its functional as well as its timing requirements. Such requirements are specified by means of the following parameters:

• Server object.

• Server method.

• Service request type (synchronous or asynchronous).

• Minimum call generation time, which is the minimum time interval between the task’s execution start time and the call’s generation time $MinCT_{c,k}$.

• Maximum call generation time, which is the maximum time interval between the task’s execution start time and the call’s generation time $MaxCT_{c,k}$.

• Min. execution time of server job $l_{s,k}$.

• Max. execution time of server job $u_{s,k}$.

• Deadline $d_{s,k}$ of the service request (In case of asynchronous calls only).

3 System Architecture

The off-line schedulability analysis phase and the run-time task execution phase are performed by means of two types of system CORBA objects, the Real-time Scheduler object and the Node Dispatcher objects. The system objects register themselves with the CORBA naming service to facilitate access to the objects’ services using the objects’ unique names. The system objects are described below.

3.1 Real-time Scheduler

The Real-time Scheduler is a single central CORBA object that runs on any of the distributed system nodes as long
as it is reachable from all other nodes. The scheduler object acts as a central repository of all system information such as nodes, objects, methods, tasks, jobs, and timing requirements’ specifications. The scheduler performs its major task of feasibility verification in the off-line system schedulability analysis phase. It is also active at the system’s run-time phase to provide the individual node dispatchers with their initialization information and execution calendars for the hard real-time applications and service requests. The scheduler object provides IDL interface methods that can be directly called by the node dispatchers to perform the various system schedulability services. The scheduler methods provide the following services:

1. Register system nodes, objects, and methods.
2. Register periodic invocations of client objects’ methods to create client tasks.
3. Register service requests from client tasks to server objects’ methods.
4. Start the system feasibility evaluation function, and retrieve its result.
5. Retrieve information about a specific system node, object, method, or task.
6. Retrieve the dynamic run-time calendars for individual nodes.

The details of the registration and schedulability analysis processes performed by the real-time scheduler are described in section 4.

3.2 Node Dispatcher

An instance of the dispatcher object runs on each node in the system. The dispatcher is responsible for analyzing the node’s local application objects and dispatching run-time tasks according to the system’s timing requirements.

In the off-line system analysis phase, the node dispatcher runs as part of the CORBA IDL compiler, or as a pre-processor for it. The dispatcher parses the application objects’ IDL code and timing requirements’ specifications and register them with the central real-time scheduler. At run-time, the node dispatcher runs in the context of two objects, the Dispatcher Event Server and the HRT Server Agent. At run-time, a node’s dispatcher objects are responsible for executing local hard real-time client jobs as well as server jobs generated as a result of external service requests according to the node’s parametric dispatching calendar.

The node dispatcher’s components and its role in dispatching hard real-time CORBA service requests are illustrated in figure 4 and described in details in section 5.

4 Schedulability Analysis

The schedulability analysis process concentrates on applications with hard real-time (HRT) QoS constraints only. Soft real-time (SRT) applications use priority based scheduling at run-time and do not require feasibility guarantees prior to run-time as they are executed on a best effort basis in this system model. The schedulability verification process is performed off-line prior to run-time to provide feasibility guarantees to all HRT applications’ tasks.

The central Real-time Scheduler CORBA object verifies the feasibility of the distributed real-time applications by processing the collected information from the nodes’ dispatchers. The schedulability analysis process is performed in four consecutive phases, system registry phase, parameter calculation phase, feasibility verification phase and finally, calendars calculation phase (figure 5).

4.1 System Registry

The first step in the system’s schedulability verification process is the system registry phase. In this step, the node dispatchers start by parsing the application’s objects’ source code, IDL interface definition code and timing requirements’ specifications. The dispatcher identifies the local application objects and parameters and registers them with
the central real-time scheduler. The information provided by each one of the node dispatchers is:

- Client and server objects.
- Objects’ public methods.
- Periodic invocations of client objects’ methods.
- Service requests to remote server objects.
- Timing parameters for client objects’ methods \((l_c, u_c)\), periodic method invocations \((F, r, d, \lambda, \eta)\), and service requests \((l_s, u_s, \text{Min}\,CT, \text{Max}\,CT)\).

After all node dispatchers have finished their registration process, the scheduler aggregates the global system functional and temporal requirements into a system wide task information repository. The collected system information is then used for the system’s feasibility evaluation process. The scheduler maps all client service requests to appropriate instances of their corresponding server objects, and generates tasks on the server nodes to execute these services. If any of the system service requests is not matched with an appropriate server method, the system of tasks is concluded to have an inconsistency and the schedulability analysis phase is terminated.

**4.2 Parameter Calculation**

After mapping all clients’ service requests to the appropriate server objects’ methods, the scheduler starts the process of partitioning the client tasks into periodic client jobs and calculating their individual timing parameters. The client jobs creation process starts with the calculation of the minimum and maximum feasible execution times \((l_{ck}, u_{ck})\) for each client job \(c_k\) using equation set 1.

\[
l_{ck} = \max(0, \min_{k} C T_k - \max_{k+1} C T_{k+1}) \\
u_{ck} = \min(u_c, \max_{k} C T_k - \min_{k+1} C T_{k+1}) \\
\forall k : 1 \leq k \leq n + 1
\]

Where:
- \(\text{Min}\,CT_0 = 0\)
- \(\text{Max}\,CT_0 = 0\)
- \(\text{Min}\,CT_{n+1} = l_c\)
- \(\text{Max}\,CT_{n+1} = u_c\)

After calculating the minimum and maximum boundaries for client jobs’ execution times, the scheduler uses these values to calculate the ready-times and deadlines for all the client and server jobs using equation set 2. The server job deadline \(d_{sk}\) is specified explicitly by the client in case of asynchronous calls and calculated using the specified equation for synchronous calls. After all jobs resulting from the task partition process have been created, the scheduler registers them in the global task information registry.

**Client jobs parameters**:

\[
\begin{align*}
    r_{c1} &= r \\
    r_{ck} &= r + \sum_{j=1}^{k-1} l_{cj} + \sum_{j=1}^{k-1} \text{sync}(l_{sj}) \\
    d_{ck} &= d - \sum_{j=k+1}^{n} l_{cj} - \sum_{j=k+1}^{n} \text{sync}(l_{sj}) \\
    d_{cn} &= d
\end{align*}
\]

**Server jobs parameters**:

\[
\begin{align*}
    r_{sk} &= r_{ck} + l_{ck} \\
    d_{sk} &= d - \sum_{j=k+1}^{n} l_{cj} - \sum_{j=k+1}^{n} \text{sync}(l_{sj})
\end{align*}
\]

The \(\text{sync}(\cdot)\) function definition:

\[
\text{sync}(l_{sj}) = \begin{cases} 
    l_{sj}, & \text{for synchronous calls} \\
    0, & \text{otherwise}
\end{cases}
\]

For every remote service request, the scheduler needs to calculate the worst case message delivery times for its call and response messages, \(q_c\) and \(q_r\) respectively. The scheduler uses the Real-time Channels method, described in section 1.1, to calculate these parameters. Using \(q_c\) and \(q_r\), the scheduler generates timing constraints among the involved client and server jobs using equation set 3. The timing constraints define the timing interaction between the client and server jobs in order to guarantee the global task set feasibility. The constraints enforce a minimum distance of \(q_c\) between the calling client job \(c_k\) and the server job \(s_k\). They also enforce a minimum distance of \(q_r\) between the server job \(s_k\) and the client job to receive the service request’s result \(c_{k+1}\), if any. In the case of synchronous requests, it is the next job \(c_{k+1}\).

\[
\begin{align*}
    s_{sk} - f_{ck} &\geq q_c \\
    s_{ck} - f_{sk} &\geq q_r \\
    s_{c(k+1)} - f_{ck} &\geq 0
\end{align*}
\]
The parameter calculation process represents the mapping of the CORBA task model described in section 2.1 to the distributed parametric scheduling task model.

4.3 Feasibility Verification

The scheduler starts the feasibility verification step by aggregating all system’s client jobs, server jobs, timing constraints, and communication constraints generated in the previous phases of the schedulability analysis process into a unified task registry. It uses the unified information in building the global constraints graph required for verifying the global system schedulability. The scheduler uses the distributed time-based parametric scheduling method to verify the feasibility of the system as described in section 1.1.

4.4 Calendar Calculation

After establishing global system schedulability, a separate dispatching parametric calendar is calculated for each node in the distributed system. The calendar contains parametric functions to be calculated by the run-time dispatcher of the node to determine a feasible range for the starting time of every HRT task to be executed on this node. The calendars are constructed in the calendar calculation phase of the distributed time-based parametric scheduling method.

The scheduler provides interfaces for retrieving calendar information at run-time. These interfaces are queried by the nodes’ run-time dispatchers in order to retrieve their individual run-time calendars and initialize their internal run-time dispatching data structures. The dispatcher maintains the Dependency Graph (DG), which is a dynamic table of all run-time values of the tasks’ calendar functions and their parameters. The dispatcher also maintains a Time Ordered List (TOL) of all the ready to execute jobs ordered according to their minimum feasible starting times.

On every system node, the dispatcher retrieves its individual node calendar for hard real-time tasks. It then starts preparing the run-time execution environment by initializing the information in the run-time dispatching data structures. It dynamically generates source code and IDL interface files for run-time dispatcher objects responsible for executing client tasks and service requests. For every method in the real-time server objects residing on the local node, the dispatcher generates IDL code for a wrapper method that has a similar prototype. Wrapper methods are compiled as part of the run-time dispatcher’s event-server object. They are used to direct all hard real-time service requests to the run-time dispatcher which in turn sends it to the application’s server methods at the appropriate times. Redirection of service requests is performed in order to enforce the system’s timing constraints and achieve local and global schedulability.

5 Run-time Execution Model

At run-time, the HRT node dispatcher runs in the context of two objects, the Dispatcher Event Server and the HRT Server Agent (figure 4). The dispatcher event server includes a wrapper method for every server object method on the local node. It is responsible for receiving all the node’s incoming HRT service requests. A dedicated object adaptor services requests to the dispatcher event server. The object adaptor uses a thread pool policy for dispatching requests in order to be able to service multiple requests at the same time. The HRT server agent executes the local node’s client jobs and server jobs generated as a result of external service requests. It runs the dynamic time-based dispatching algorithm described in section 1.2. It is essential for the HRT server agent to run as a dedicated thread with the highest system priority in order to run HRT jobs non-preemptively. Ready-to-run HRT jobs are inserted into a time ordered list (TOL) ordered according to the jobs’ execution start times.

On every system node, the dispatcher retrieves its individual node calendar for hard real-time tasks. It then starts preparing the run-time execution environment by initializing the information in the run-time dispatching data structures. It dynamically generates source code and IDL interface files for run-time dispatcher objects responsible for executing client tasks and service requests. For every method in the real-time server objects residing on the local node, the dispatcher generates IDL code for a wrapper method that has a similar prototype. Wrapper methods are compiled as part of the run-time dispatcher’s event-server object. They are used to direct all hard real-time service requests to the run-time dispatcher which in turn sends it to the application’s server methods at the appropriate times. Redirection of service requests is performed in order to enforce the system’s timing constraints and achieve local and global schedulability.

5.1 Service-Request Execution Scheme

Client tasks residing on remote nodes can send service requests to local node’s server objects by directly calling the wrapper methods of the dispatcher event server using RPC scheme. The event server encapsulates the arriving remote call’s method, arrival time, and function parameters into an event message and forwards it to the HRT server agent through the External Event Queue (EEQ). In case of synchronous service requests, the event server thread blocks, waiting for the server method to finish its execution. The HRT server agent polls EEQ after finishing the execution of each HRT job. When it finds a service request event in EEQ, it sends it by recording its parameters and updating the parametric functions of the corresponding server job to indicate the arrival of that event. When the server job becomes ready for execution, it is inserted into TOL, and then dispatched to the CPU for execution. When a server job, resulting from a synchronous service request, finishes its execution, the HRT server agent sends an event message to the event server. The event includes the job’s finish time, method return value, and output parameters. On receiving the event message, the event server thread resumes its execution. It communicates the service request’s return values back to the remote caller’s client task. The service request execution scheme is illustrated in figure 6.
6 Conclusion

In this paper, we presented a system model for a time-based real-time CORBA environment. The model provides an overall CORBA scheduling, dispatching, and service specification environment that supports real-time QoS requirement specification and enforcement. The model presents a transformation mechanism for representing CORBA’s distributed objects’ model as a distributed task set with inter-task relative timing constraints. It uses the distributed dynamic time-based scheduling method to verify the task set feasibility. At run-time, the model provides time-based dispatching of distributed tasks and service requests while preserving the RPC interface among CORBA applications’ objects. The complete model was implemented and tested using simulated sample real-time distributed systems.

We believe that the presented time-based CORBA scheduling and dispatching model can be applied to many distributed object oriented real-time applications that have strict and complex timing constraints in resource management and task communication. This CORBA model provides applications with time-based end-to-end QoS guarantees for service requests, flexible management of slack-time at run-time, and proper temporal dispatching of distributed tasks and service requests. These features differentiate our time-based CORBA model from existing, mostly priority-based, real-time CORBA models.

References
