The Design and Implementation of Real-Time CORBA 2.0: Dynamic Scheduling in TAO

Yamuna Krishnamurthy and Irfan Pyarali
OOMWORKS LLC
Metuchen, NJ
{yamuna, irfan} @oomworks.com

Christopher Gill, Louis Mgeta, Yuanfang Zhang, and Stephen Torri
Washington University
St. Louis, MO
{cdgill,lmm1,yfzhang,storri} @cse.wustl.edu

Douglas C. Schmidt
Vanderbilt University
Nashville, TN
d.schmidt@vanderbilt.edu

Abstract

In an emerging class of open distributed real-time and embedded (DRE) systems with stringent but dynamic QoS requirements, there is a need to propagate QoS parameters and enforce task QoS requirements across multiple endsystems in a way that is simultaneously efficient and adaptable. The Object Management Group's (OMG) Real-Time CORBA 2.0 specification (RTC2) defines a dynamic scheduling framework for propagating and enforcing QoS parameters dynamically in standard CORBA middleware.

This paper makes two contributions to research on middleware for open DRE systems. First, it describes the design and capabilities of the RTC2 dynamic scheduling framework provided by TAO, which is our open-source CORBA standards-based Object Request Broker (ORB). Second, it describes and summarize the results of empirical studies we have conducted to validate our RTC2 framework in the context of open DRE systems. The results of those experiments show that a range of policies for adaptive scheduling and management of distributable threads can be enforced efficiently in standard middleware for open DRE systems.

1. Introduction

Emerging trends. Developing distributed real-time and embedded (DRE) systems whose quality of service (QoS) can be assured even in the face of changes in available resources is an important and challenging R&D problem. It is useful to distinguish two types of QoS assurance for DRE systems: (1) *static*, in which QoS requirements are known *a priori* and can be enforced efficiently due to that knowledge and (2) *dynamic*, in which QoS requirements can vary at run-time and their enforcement requires more flexible mechanisms that may incur more overhead. Many DRE systems have a combination of static and dynamic QoS requirements, and hybrid static/dynamic enforcement mechanisms have been shown useful in previous work [1].

Static approaches are often suitable for *closed* DRE systems, where the set of application tasks that will run in the system and the loads they will place on system resources *are* known in advance. For example, the OMG's Real-time CORBA 1.0 (RTC1) specification [2] supports statically scheduled DRE systems in which task eligibility can be mapped to a fixed set of priorities. Such closed systems can be scheduled *a priori.*

Dynamic approaches to resource management are often essential for *open* DRE systems, which consist of independently developed application components that are distributed across host endsystems sharing a common network. Due to the heterogeneity of the components, the complexity of their modes of interaction, and the dynamic environments in which they operate, it is hard to specify the resource requirements of open DRE systems *a priori*.

New middleware challenges and solutions. The OMG Real-Time CORBA 2.0 specification (RTC2) [5] addresses the limitations with the fixed-priority mechanisms specified by RTC1. In particular, RTC2 extends RTC1 by providing interfaces and mechanisms that applications can use to plug in dynamic schedulers and interact with them across a distributed system. RTC2 therefore gives application developers more flexibility to specify and use scheduling disciplines and parameters that accurately define and describe their execution and resource requirements. To accomplish this, RTC2 introduces two new concepts to Real-time CORBA: (1) *distributable threads* that are used to map end-to-end QoS requirements to sequential and branching distributed computations across the

Proceedings of the 10th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'04)
1080-1812/04 $ 20.00 © 2004 IEEE

endsystems they traverse and (2) a *scheduling service architecture* that allows applications to choose which mechanisms enforce task eligibility.

We have implemented a RTC2 prototype that enhances on our prior work with The ACE ORB (TAO) [6] and its Real-time Scheduling Service [7][8] . This paper describes how we designed and optimized the performance of our RTC2 Dynamic Scheduling framework to address the following design challenges:

- Defining a means to install pluggable dynamic schedulers that support scheduling policies and mechanisms for a wide range of DRE applications,
- Creating an interface that allows customization of interactions between an installed RTC2 dynamic scheduler and an application,
- Portable and efficient mechanisms for distinguishing between distributable thread and OS thread identities, and
- Safe and effective mechanisms for canceling distributable threads to give applications control over distributed concurrency.

The results of our efforts have been integrated with the TAO open-source software release and are available from `deuce.doc.wustl.edu/Download.html`.

Paper organization. The remainder of this paper is organized as follows: Section 2 describes the RTC2 specification and explains the design of our RTC2 framework, which has been integrated with the TAO open-source Real-time CORBA ORB; Section 3 presents empirical studies we conducted to validate our RTC2 approach and to quantify the costs of dynamic scheduling of distributable threads; and Section 4 offers concluding remarks.

2. Dynamic Scheduling Framework Design and Implementation for RT CORBA 2.0

This section describes the key characteristics and capabilities of RTC2 specification and describes the RTC2 dynamic scheduling framework that we have integrated with the TAO Real-time CORBA ORB. The key elements of TAO's RTC2 framework, which Figure 1 illustrates, are:

1. *Distributable threads*, which applications use to traverse endsystems,
2. *Scheduling segments*, which map policy parameters to distributable threads at specific points of execution,
3. *Current execution locus*, the head of an active thread,
4. *Scheduling policies*, which determine the eligibility of each thread based on parameters of the scheduling segment within which that thread is executing, and
5. *Dynamic scheduler*, which reconciles the set of scheduling policies for all segments and threads on an endsystem, to determine which thread is active.

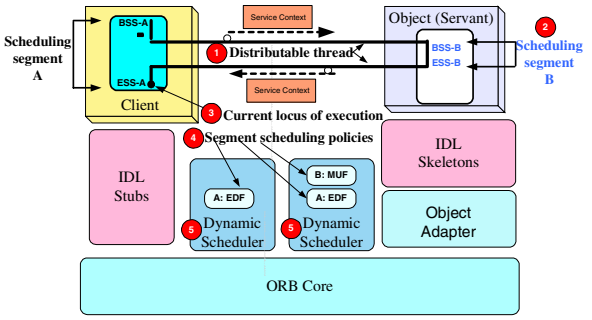


Figure 1: TAO's RTC2 Architecture

The remainder of this section explains the concepts of *distributable threads* and the *pluggable scheduling framework* specified by RTC2 and outlines the design of these concepts in TAO's RTC2 framework implementation. We also describe *scheduling points*, which govern how and when scheduling parameter values can be mapped to distributable threads. We conclude by discussing issues related to distributable thread identity (such as the need to emulate OS-level thread-specific mechanisms for storage or locking in middleware) and examine the interfaces and mechanisms needed to cancel distributable threads.

2.1. Distributable Threads

DRE applications must manage key resources, such as CPU cycles, network bandwidth, and battery power, to ensure predictable behavior along each end-to-end path. In RTC1-based DRE systems, application end-to-end priorities can be acquired from clients, propagated with invocations, and used by servers to arbitrate access to endsystem CPU resources. For dynamic DRE systems, the fixed-priority propagation model provided by RTC1 is insufficient because these DRE systems require more information than just priority, e.g., they may need deadline, execution time, and laxity. A more sophisticated abstraction than priority is thus needed to identify the most eligible schedulable entity, and additional scheduling parameters may need to be associated with it so that it can be scheduled appropriately.

A natural unit of scheduling abstraction suggested by CORBA's programming model is a thread that can execute operations in objects without regard for physical endsystem boundaries. In the RTC2 specification, this programming model abstraction is

termed a *distributable thread*, which can span multiple endsystems and is the primary schedulable entity in RTC2-based DRE applications. A distributable thread replaces the concept of an *activity* that was introduced but not formally specified in the RTC1 specification.

Each distributable thread in RTC2 is identified by a unique system wide identifier called a *Globally Unique Id (GUID)* [9]. A distributable thread may have one or more execution scheduling parameters, e.g., priority, time-constraints (such as deadlines), and importance. These parameters specify the acceptable end-to-end timeliness for completing the sequential execution of operations in CORBA object instances that may reside on multiple physical endsystems.

Within each endsystem, the flow of control of the distributable thread is mapped onto the execution of a local thread provided by the OS. At any given instant, each distributable thread has only one execution point in the whole system, *i.e.*, a distributable thread does not execute simultaneously on multiple endsystems it spans. Instead, it executes a code sequence consisting of nested distributed and/or local operation invocations, similar to the way a local thread executes through a series of nested local operation invocations. Below, we describe the key interfaces and properties of distributable threads in the RTC2 specification and explain how we implement those aspects in TAO.

Scheduling segment. A distributable thread comprises one or more *scheduling segments.* A scheduling segment is a code sequence whose execution is scheduled according to a distinct set of scheduling parameters specified by the application. For example, the worst-case execution time, deadline, and criticality of a real-time operation is used by the Maximum Urgency First (MUF) scheduling strategy [4]. These parameters can be associated with a segment encompassing that operation on a particular endsystem, e.g., as shown for segment B in Figure 1. The code sequence that a scheduling segment comprises can include remote and/or local operation invocations.

Appropriate operations defined in the `RTScheduling::Current` interface described below are used to open or close a scheduling segment. A DRE application opens a scheduling segment by calling the `begin_scheduling_segment()` operation and then closes the scheduling segment by calling the `end_scheduling_segment()` operation. Nested scheduling segments are allowed so that different parts of a scheduling segment can be scheduled with different sets of scheduling parameters.

Our RTC2 implementation in TAO currently places a restriction that the calls to begin and end each scheduling segment should be made on the same physical endsystem. We will lift this restriction in future work so that the begin and end points can be on different hosts. We will provide different strategies for the cases where all begin and end points are or are not collocated, however, since supporting distributed segment begin and end points incurs more overhead.

The Current interface. The `RTScheduling module's Current` interface defines operations that begin, update, and end the scheduling segments described above, as well as create and destroy distributable threads. Each scheduling segment has a unique instance of this `Current` object managed in local thread specific storage (TSS) [10] on each endsystem along the overall path of the distributable thread. A nested scheduling segment keeps a reference to the `Current` instance of its enclosing scheduling segment. Each operation in the Current interface of the RT Scheduling module is described below.

begin_scheduling_segment() – A DRE application calls this operation to start a scheduling segment. If the caller is not already within a distributable thread, a new distributable thread is created. If the caller is already within a distributable thread, a nested scheduling segment is created. This call is also a *scheduling point*, where the application interacts with the RTC2 dynamic scheduler to select the currently executing thread (Section 2.3 describes scheduling points in depth).

update_scheduling_segment() – This operation is a scheduling point for the application to interact with the RTC2 dynamic scheduler to update the scheduling parameters and to check whether or not the schedule remains feasible. It must be called only from within a scheduling segment. A `CORBA::BAD_INV_ORDER` exception is thrown if this operation is called outside a scheduling segment context.

end_scheduling_segment() – This operation marks the end of a scheduling segment and the termination of the distributable thread if the segment is not nested within another scheduling segment. Every call to `begin_scheduling_segment()` should have a corresponding call to `end_scheduling_segment()`. As noted earlier, in TAO's RTC2 prototype the segment begin and end calls should be on the same host and in the same thread. After a call to `end_scheduling_segment()`, the distributable thread is operating with the scheduling parameter of the next outer scheduling segment scope. If this operation is performed at the outermost scope, the processing for the native thread that the distributable thread is mapped onto reverts back to the fixed priority OS scheduling where the active thread priority is the sole determinant of the threads eligibility for execution.

spawn() – A distributable thread can create a new distributable thread by invoking the `spawn()` operation. If the scheduling parameters for the new distributable thread not specified explicitly, the implicit scheduling parameters of the distributable thread calling `spawn()` are used. The `spawn()` operation can only be called by a distributable thread, otherwise a `CORBA::BAD_INV_ ORDER` exception is thrown.

Distributable thread location. Now that we have explained the terminology and API for distributed threads we can illustrate how all the pieces fit together. A distributable thread may be entirely local to a host or it may span multiple hosts by making remote invocations. Figures 2 and 3 therefore illustrate the different spans that are possible for distributable threads. In these figures, calls made by the application are shown as solid dots, while calls made by interceptors within the middleware are shown as shaded rectangles.

In Figure 2, DT1 makes a two-way invocation on an object on a different host and also has a nested segment started on Host 2 (BSS-B to ESS-B within BSS-A to ESS-A). DT2 and DT3 are simple distributable threads that do not traverse host boundaries. DT2 has a single scheduling segment (BSS-C to ESS-C), while DT3 has a nested scheduling segment (BSS-E to ESS-E within BSS-D to ESS-D). In Figure 3 DT2 is created by the invocation of the `RTScheduling::Current::spawn()` operation within DT1, while DT4 is implicitly created on Host 2 to service a one-way invocation. DT4 is destroyed when the upcall completes on Host 2.

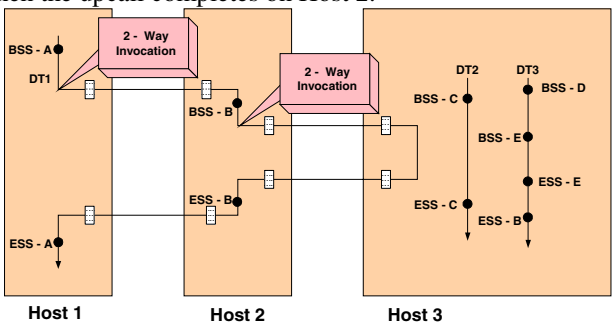


Figure 2: Distributable Threads and the Hosts They Span

2.2. Pluggable Scheduling

Different distributable threads in a DRE system contend for shared resources, such as CPU cycles. To support the end-to-end QoS demands of open DRE systems, it is imperative that such contention be resolved predictably, which necessitates scheduling and dispatching mechanisms for these entities based on the real-time requirements of the system. In the RTC2 specification, the scheduling policy decides the sequence in which the distributable threads should be given access to the resources and the dispatching mechanism grants the resources according to the sequence decided by the scheduling policy.

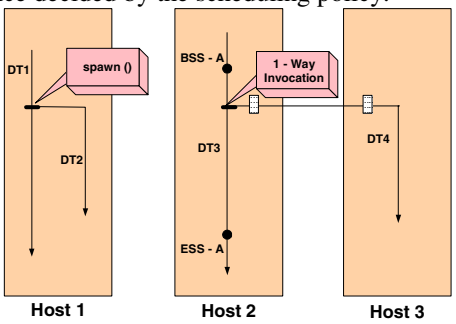


Figure 3: Ways to Spawn a Distributable Thread

Various scheduling disciplines exist that require different scheduling parameters, such as MLF, EDF, MUF [4], or RMS+MLF [11]. One or more of these scheduling disciplines (or any other discipline the system developer chooses) may be used by an open DRE system to fulfill its scheduling requirements. Supporting this flexibility requires a mechanism by which different dynamic schedulers (each implementing one or more scheduling disciplines) can be plugged into an RTC2 implementation.

The RTC2 specification provides a common CORBA IDL interface, `RTScheduling::Scheduler`. This interface has the semantics of an abstract class from which specific dynamic scheduler implementations can be derived. In the RTC2 specification, the dynamic scheduler is installed in the ORB and can be queried with the standard CORBA `ORB::resolve_initial_references` factory operation using the name "`RTScheduler`." The application then interacts with the installed RTC2 dynamic scheduler (e.g., passing its scheduling requirements) using operations defined in the `RTScheduling::Scheduler` interface that is listed under Scheduler Upcalls in Table 1. Similarly, the ORB interacts with the RTC2 dynamic scheduler at the points described in Section 2.3 to ensure proper dispatching and sharing of scheduling information across hosts. This is done through scheduler operations listed under *scheduler upcalls* in Table 2.

2.3. Scheduling Points

An application and ORB interact with the RTC2 dynamic scheduler at pre-defined points to schedule distributable threads in a DRE system. These scheduling points allow an application and ORB to provide the RTC2 dynamic scheduler information about the competing tasks in the system, so it can make

scheduling decisions in a consistent and predictable manner. We now describe these scheduling points, which are illustrated in Figure 4.

Table 1: Summary of Scheduler Upcalls for User Invoked Scheduling Points

User invokes	*Scheduler upcall*
`Current::spawn`	`Scheduler::begin_new_scheduling_segment`
`Current::begin_scheduling_segment`	`Scheduler::begin_new_scheduling_segment`
`Current::begin_scheduling_segment`	`Scheduler::begin_nested_scheduling_segment`
`Current::Update_scheduling_segment`	`Scheduler::Update_scheduling_segment`
`Current::end_scheduling_segment`	`Scheduler::end_nested_scheduling_segment`
`Current::end_scheduling_segment`	`Scheduler::end_scheduling_segment`
`DistributableThread::Cancel`	`Scheduler::cancel`

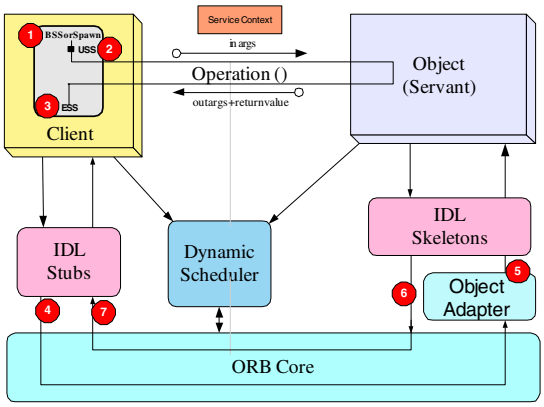
Table 2: Summary of Scheduler Upcalls for ORB Invoked Scheduling Points

ORB intercepts	*Scheduler upcall*
`Outgoing request`	`Scheduler::send_request`
`Incoming request`	`Scheduler::receive_request`
`Outgoing reply`	`Scheduler::send_reply`
`Incoming reply`	`Scheduler::receive_reply`

Scheduling points 1-3 in Figure 4 are points where an application interacts with the RTC2 dynamic scheduler and are summarized in Table 1. The key application-level scheduling points and their characteristics are:

New distributable threads and segments. When a new scheduling segment or new distributable thread is created, the RTC2 dynamic scheduler must be informed so that it can schedule the new segment. The RTC2 dynamic scheduler then schedules the new scheduling segment based on its parameters and those of the active scheduling segments for other distributable threads in the system when application code outside a distributable thread calls the `begin_new_scheduling_segment()` operation to create a new distributable thread, or when code within a distributable thread makes a call to `begin_nested_scheduling_segment()` to create a nested scheduling segment.

Changes to scheduling segment parameters. When the `Current::update_scheduling_segment()` op-eration is invoked by a distributable thread to change its scheduling parameters, it updates scheduling parameters of the corresponding scheduling segment via `Scheduler::update_scheduling_segment()`.



1. BSS - RTScheduling::Current::begin_scheduling_segment() or RTScheduling::Current::spawn()
2. USS - RTScheduling::Current::update_scheduling_segment()
3. ESS - RTScheduling::Current::end_scheduling_segment()
4. send_request() interceptor call
5. receive_request() interceptor call
6. send_reply() interceptor call
7. receive_reply() interceptor call

Figure 4: RTC2 Scheduling Points

Termination of a scheduling segment or distributable thread. The RTC2 dynamic scheduler should be informed when `Current::end_scheduling_segment()` is invoked by a distributable thread to end a scheduling segment or when a distributable thread is cancelled, so it can reschedule the system accordingly. Hence, the `Current::end_scheduling_segment()` operation invokes the `end_scheduling_segment()` operation on the RTC2 dynamic scheduler to indicate when the outermost scheduling segment is terminated. The RTC2 dynamic scheduler then reverts the thread to its original scheduling parameters. If a nested scheduling segment is terminated, an automatic invocation of `Scheduler::end_nested_scheduling_segment()` occurs. The RTC2 dynamic scheduler then ends the scheduling segment and resets the distributable thread to the scheduling parameters of the enclosing scheduling segment scope.

As is described in Section 2.4.2, a distributable thread can also be terminated from the application or another distributable thread by calling the `cancel()` operation on the distributable thread. When the distributable thread is cancelled, the `Scheduler::cancel` operation is called automatically by the RTC2 framework, which informs the RTC2 dynamic scheduler of the distributable thread cancellation.

Scheduling points 4-7 in Figure 4 are points where an ORB interacts with the RTC2 dynamic scheduler, i.e., when remote invocations are made between different hosts, and are summarized in Table 2. Collocated invocations occur when the client and server are located in the same process. In collocated two-way invocations, the thread making the request

also services the request. Unless a scheduling segment begins or ends at that point, therefore, the distributable thread does not have to be rescheduled by the RTC2 dynamic scheduler. Collocated one-way invocations do not result in creation of a new distributable thread in TAO's RTC2 implementation due to (1) the overhead of distributable thread creation, (2) scheduling overhead/complexity, (3) lack of interceptor support for collocated one-ways, and (4) lack of support for executing collocated calls in separate threads.

The ORB interacts with the RTC2 dynamic scheduler at points where the remote operation invocations are sent and received. Client-side and server-side interceptors are therefore installed to allow interception requests as they are sent and received. These interceptors are required (a) to intercept where a new distributable thread is spawned in one-way operation invocations and create a new GUID for that thread on the server, (b) to populate the service contexts, sent with the invocation, with the GUID and required scheduling parameters of the distributable thread, (c) to re-create distributable threads on the server, (d) to perform cleanup operations for the distributable thread on the server when replies are sent back to a client for two-way operations, and (e) to perform cleanup operations on the client when the replies from two-way operations are received. These interception points interact with the RTC2 dynamic scheduler so it can make appropriate scheduling decisions. The key RTC2 ORB-level scheduling points and their characteristics are described below.

Send request. When a remote operation invocation is made, the RTC2 dynamic scheduler must be informed to ensure that it can (1) populate the service context of the request to embed the appropriate scheduling parameters of the distributable thread and (2) potentially re-map the local thread associated with the distributable thread to service another distributable thread. As discussed in Section 2.4, when the distributable thread returns to that same ORB during a nested upcall, it may be mapped to a different local thread than the one with which it was associated previously. The client request interceptor's `send_request()` operation is invoked automatically just before a request is sent. This operation in turn invokes `Scheduler::send_request()` with the scheduling parameters of the distributable thread that is making the request. The scheduling information in the service context of the invocation enables the RTC2 dynamic scheduler on the remote host to schedule the incoming request appropriately.

Receive request. When a request is received, the server request interceptor's `receive_request()` operation is invoked automatically by the RTC2 framework before the upcall to the servant is made. This operation in turn invokes `Scheduler::receive_request()`, passing it the received service context that contains the GUID and scheduling parameters for the corresponding distributable thread. It is the responsibility of the RTC2 dynamic scheduler to unmarshal the scheduling information in the service context that is received. The RTC2 dynamic scheduler uses this information to schedule the thread servicing the request, and the ORB requires it to reconstruct a `RTScheduling::Current`, and hence a distributable thread, on the server.

Send reply – When the distributable thread returns via a two-way reply to a host from which it migrated, the `send_reply()` operation on the server request interceptor is called automatically by the RTC2 framework just before the reply is sent. This operation in turn calls the `Scheduler::send_reply()` `operation` on the server-side RTC2 dynamic scheduler so it can perform any scheduling of the thread making the upcall as required by the scheduling discipline used so the next eligible distributable thread in the system is executed.

Receive reply. Distributable threads migrate across hosts through two-way calls. The distributable thread returns to the previous host, from where it migrated, through the reply of the two-request. When the reply is received the client request interceptor's `receive_reply()` operation is invoked. This operation in turn invokes `Scheduler::` `receive_reply()` on the client-side RTC2 dynamic scheduler, which then performs any scheduling related decisions required by the scheduling discipline, as a distributable thread re-enters the system.

2.4. Challenges of Implementing an RTC2 Framework in TAO

To manage distributable threads correctly, an RTC2 framework must resolve a number of design challenges. Below we examine two challenges we faced when implementing distributable threads (1) managing distributable vs. OS thread identities and (2) canceling distributable threads. For each challenge, we describe the context in which the challenge arises, identify the specific problem that must be addressed, describe our solution for resolving the challenge, and explain how this solution was applied to TAO's RTC2 framework.

2.4.1. Distributable vs. OS Thread Identity

Context. A key design issue with the RTC2 specification is that in modern ORB middleware with alternative concurrency strategies [11], a distributable

thread may be mapped on each endsystem to several different OS threads over its lifetime. Figure 5 illustrates how a distributable thread can use thread-specific storage (TSS), lock resources recursively so that they can be re-acquired later by that same distributable thread, or perform any number of other operations that are sensitive to the identity of the distributable thread performing them.

In Figure 5, distributable thread DT1 associated with OS thread 1 writes information into TSS on endsystem A and then migrates to endsystem B. Before DT1 makes a nested two way call back to an object on endsystem A, DT2 migrates from endsystem B to endsystem A. For efficiency, flexible concurrency strategies (such as thread pools [2]) may map distributable threads to whatever local threads are available. For example, Figure 5 shows DT2 mapped to OS thread 1 and when DT1 returns to endsystem A it maps to OS thread 2.

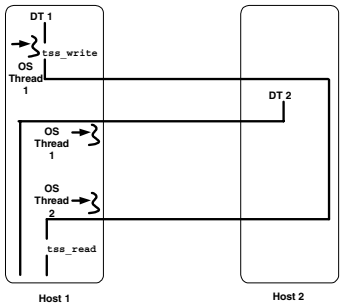


Figure 5: TSS with Distributable Threads

Problem. Problems can arise when DT1 wants to obtain the information it previously stored in TSS. If native OS-level TSS was used, OS thread 2 cannot access the TSS for OS thread 1, so DT1's call to `tss_read()` in Figure 5 will fail. Moreover, the OS-level TSS mechanism does not offer a way to substitute the OS thread identity used for a TSS call, even temporarily.

Solution. To resolve these problems, some notion of distributable thread identity is needed that is separate from the identities of operating system threads. Likewise, mechanisms are needed that use distributable thread GUIDs rather than OS thread IDs, which results in an emulation of OS-level mechanisms in middleware that can incur additional overhead.

2.4.2. Canceling a Distributable Thread

Context. DRE applications may need to cancel distributable threads that become useless due to deadline failure or to changing application requirements at run-time, or that might interfere with other distributable threads that have become more important. In the RTC2 specification, a distributable thread supports the following `DistributableThread` interface whose `cancel()` operation can be invoked to stop the corresponding distributable thread.

```
local interface
RTScheduling::DistributableThread
{
  // raises CORBA::OBJECT_NOT_FOUND
  // if distributable thread  is not
known
  void cancel();
};
```

The `DistributableThread` instance is created when the outer most scheduling segment is created. All nested scheduling segments are associated with the same distributable thread that they constitute.

Problem. Safe and effective cancellation of a distributable thread requires that two conditions are satisfied: (1) cancellation must only be invoked on a distributable thread that in fact exists in the system and multiple cancellation of a distributable thread must not occur and (2) because a distributable thread may have locked resources or performed other operations with side effects outside that distributable thread, the effects of those operations must be reversed before the distributable thread is destroyed.

Solution. To cancel a distributable thread, the application can only call the `cancel()` operation on the *instance* of the distributable thread that is to be cancelled. Moreover, once cancellation is successful that instance becomes invalid for further cancellation. In the TAO RTC2 framework, this operation causes the `CORBA:: THREAD_CANCELLED` exception to be (1) raised in the context of the distributable thread at the next scheduling point for the distributable thread and (2) propagated to where the distributable thread started, as illustrated in Figure 6. A distributable thread can be cancelled any time on any host that it currently spans. As shown in Figure 6, the distributable thread was cancelled on Host 2, even though it is currently executing on Host 3.

When the `cancel()` operation is called, a thread cancelled exception is propagated to the start of the distributable thread. As shown in Figure 6, the `CORBA::THREAD_CANCELLED` exception is propagated from Host 2 to Host 1 where the distributable thread started. Since the cancellation is not forwarded to the head of the distributable thread if it is not on the same host, the cancellation will only be processed after the distributable thread returns to Host 2 from Host 3.

Note that while the distributable thread is a local interface, the head of the distributable thread may not be executing within the same address space as the thread calling `cancel()`. Hence, `cancel()` is implemented by setting a flag in the `DistributableThread` interface to mark it as cancelled. At the next local scheduling point of the

distributable thread a check for cancellation of the distributable thread will be performed. If the flag is set, then the distributable thread is cancelled and the `CORBA:: THREAD_CANCELLED` exception is raised and the relevant resources are released. After `CORBA::THREAD_ CANCELLED` is raised and the distributable thread is cancelled, the local thread that the distributable thread was mapped is released, possibly to be used by another distributable thread.

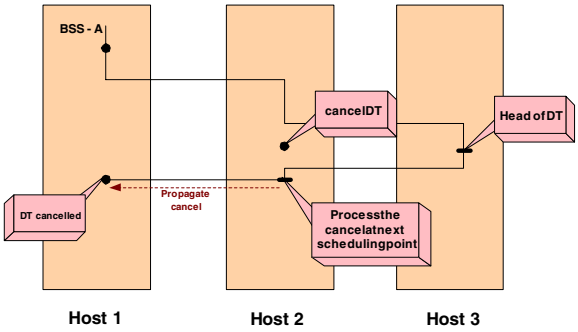


Figure 6: Distributable Thread Cancellation

3. Empirical Studies

This section describes empirical benchmarks we conducted to validate TAO's RTC2 architecture described in Section 2. We describe experiments that evaluated alternative mechanisms for scheduling distributable threads according to *importance*.

Experiment overview and configuration. We plugged two RTC2 dynamic scheduler implementations into the TAO ORB to test the behavior of its RTC2 dynamic scheduling framework with different scheduling mechanisms. Both implementations use a scheduling policy that *prioritizes* distributable threads. Our *Fixed Priority (FP) Scheduler* implementation schedules distributable threads by mapping each one's dynamic importance to native OS priorities. Our *Most Important First (MIF) Scheduler* implementation maintains its own ready queue that stores distributable threads in order of their importance, and the local thread to which each distributable thread in the queue is mapped waits on a condition variable. When a distributable thread reaches the head of the queue (*i.e.*, it is next to execute), the MIF Scheduler signals the corresponding condition variable on which the local thread is waiting, to awaken it.

The experimental configuration we used to examine both the FP and MIF schedulers is identical. The test consisted of a set of local and distributed (spanning two hosts) distributable threads. The hosts were both running RedHat Linux 7.1 in the real-time scheduling class. The local distributable threads consisted of threads performing CPU bound work on the local host for a given execution time. The distributed distributable threads (1) performed the specified local CPU bound work on the local host, (2) then made the remote invocation performing CPU bound work on the remote host for a given execution time, and (3) came back to the local host to perform the specified local CPU bound work. Tables 3 and 4 show the scheduling parameters of distributable threads on Host 1 and Host 2 respectively: execution times for local work before and after the remote invocation are separated by a '+'. In the Execution Time column, "Loc." represents local execution times while "Rem." represents remote execution times.

Table 3: Distributable Thread Schedule on Host 1

GUID	Start Time (secs)	Importance	Execution Time (secs)		Span
			Loc.	Rem.	
1	0	9	3+3	3	Dist
2	0	3	6+6	3	Dist
3	12	1	6	N/A	Local

Table 4: Distributable Thread Schedule on Host 2

GUID	Start Time (secs)	Importance	Execution Time		Span
			Loc.	Rem.	
4	0	5	9	N/A	Local
5	9	7	3	N/A	Local

Summary of Empirical results. Since both the FP and MIF schedulers use the same scheduling *policy* based on the importance of the distributable threads, the resulting performance was nearly identical. This result demonstrates that system-wide dynamic scheduling can be achieved with TAO's RTC2 framework when tasks (represented by the distributable threads) enter and leave the system dynamically. Since both the FP and MIF schedulers schedule the distributable threads based on their importance, it is not surprising based on *policy* that the performance was nearly identical. The one small but important difference from a *mechanism* perspective was in the times at which the threads are suspended and resumed, due to the context switch time for the MIF scheduler (which is at the middleware level) compared to the FP scheduler (which is at the OS level). These results validate our hypothesis that dynamic schedulers implementing different scheduling

disciplines *and even using different scheduling mechanisms* can be plugged into TAO's RTC2 framework to schedule the distributable threads in the system according to a variety of requirements, while maintaining reasonable efficiency.

4. Concluding Remarks

The Real-time CORBA 2.0 (RTC2) specification defines a dynamic scheduling framework that enhances the development of open distributed real-time and embedded (DRE) systems that possess dynamic QoS requirements. The RTC2 framework provides a *distributable thread* capability that can support execution sequences requiring dynamic scheduling and enforce their QoS requirements based on scheduling parameters associated with them. The RTC2 distributable threads abstraction can extend over as many hosts that the execution sequence may span. Flexible scheduling is achieved by plugging in dynamic schedulers that implement different scheduling disciplines, such as EDF, MLF, MUF, or RMS+MLF.

TAO's implementation of the RTC2 specification has addressed a broader set of issues than the standard itself covers, such as mapping distributable and local thread identities, supporting hybrid static and dynamic scheduling, and defining efficient mechanisms for enforcing a variety of scheduling policies. We learned the following lessons based on our experience developing and empirically evaluating TAO's RTC2 framework:

- RTC2 is a good beginning towards addressing the dynamic scheduling issue in DRE systems. To achieve correctness, however, there is a need for a robust implementation of a Scheduling Service that works in conjunction with the RTC2 framework. By integrating our earlier work on the Kokyu scheduling framework [7], [8] within the RTC2 standard, we have provided a wider range of scheduling policies and mechanisms.
- Some features that are implemented for the efficiency of thread and other resource management can hinder the correct working of the RTC2 framework. For example, managing distributable threads is more costly and complicated due to sensitivity of key mechanisms to their identities, as is discussed in Section 0.
- In practice, relatively few DRE applications need system-wide dynamic scheduling, which has limited the scope of the RTC2 specification. In particular, it presently does not address interoperability of the dynamic schedulers on different hosts. Instead, it only ensures propagation of timeliness requirements of an execution sequence across the hosts it spans so it can be scheduled on each host.

References

[1] Gill, Schmidt, Cytron, "Multi-Paradigm Scheduling for Distributed Real-Time Embedded Computing", IEEE Proc. 91(1), Jan. 2003.

[2] Real-Time CORBA Specification, Aug. 2002, www.omg.org/docs/formal/02-08-02.pdf

[3] Pyarali, D. Schmidt, R. Cytron, "Techniques for Enhancing Real-Time CORBA Quality of Service", IEEE Proc., 91(7), July 2003.

[4] D. B. Stewart and P. K. Khosla, "Real-Time Scheduling of Sensor-Based Control Systems," in Real-Time Programming (W. Halang and K. Ramamritham, eds.), Tarrytown, NY: Pergamon Press, 1992.

[5] Real –Time CORBA 2.0: Dynamic Scheduling specification, OMG Final Adopted Specification, Sept. 2001, www.omg.org/docs/ptc/01-08-34.pdf

[6] Schmidt, Levine, and Mungee. "The Design and Performance of the TAO Real-Time Object Request Broker", Computer Communications 21(4), April 1998.

[7] Gill C., Levine D., Schmidt, D. "The Design and Performance of a Real-Time CORBA Scheduling Service," The International Journal of Time-Critical Computing Systems 20(2), Kluwer, March 2001.

[8] Gill, Schmidt, and Cytron, "Multi-Paradigm Scheduling for Distributed Real-Time Embedded Computing", IEEE Proceedings 91(1), Jan 2003.

[9] UUIDs and GUIDs Internet-Draft, Paul J. Leach, Rich Salz, www.opengroup.org/dce/info/draft-leach-uuids-guids-01.txt

[10] Douglas C. Schmidt, Michael Stal, Hans Rohnert, and Frank Buschmann, "Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects, Volume 2", Wiley, NY, 2000.

[11] Chung, Liu, Lin, "Scheduling Periodic Jobs that Allow Imprecise Results," IEEE Transactions on Computers, vol. 39, Sept 1990.

[12] Pyarali, C. O'Ryan, D. Schmidt, N. Wang, V. Kachroo, A. Gokhale, "Using Principle Patterns to Optimize Real-Time ORBs", IEEE Concurrency, Vol. 8, No. 1, Jan-Mar 2000.