# Enhancing the Adaptivity of Distributed Real-time and Embedded Systems via Standard QoS-enabled Dynamic Scheduling Middleware

Christopher Gill*, Louis Mgeta,Yuanfang Zhang, and Stephen Torri
*Box 1045, One Brookings Drive
Washington Univ., St. Louis, MO 63130
cdgill@cse.wuslt.edu

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

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

## Abstract

*To support the dynamically changing QoS needs of open distributed real-time embedded (DRE) systems, it is essential to propagate QoS parameters and to enforce task QoS requirements adaptably across multiple endsystems dynamically in a way that is simultaneously efficient, flexible, and timely. This paper makes three contributions to research on QoS-enabled middleware that supports these types of open DRE systems. First, it describes the design and implementation of a dynamic scheduling framework based on the OMG Real-Time CORBA 2.0 specification (RTC2) that provides capabilities for (1) propagating QoS parameters and a locus of excecution across endsystems via a distributable thread abstraction and (2) enforcing the scheduling of multiple distributable threads dynamically using standard CORBA middleware. Second, it examines the results of empirical studies conducted to validate our RTC2 framework in the context of open DRE systems. These experiments show that a range of policies for adaptive dynamic scheduling and management of distributable threads can be enforced efficiently in standard middleware for open DRE systems. Third, it presents results from case studies of multiple adaptive middleware QoS management technologies to monitor and control the quality, timeliness, and criticality of key operations (such as navigation and imagery transfer) adaptively in a representative DRE avonics system.*

**Keywords:** Real-time CORBA, adaptive systems, dynamic scheduling.

## 1. Introduction

**Emerging trends and challenges for DRE systems and middleware.** 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. QoS-enabled middleware has been applied successfully to certain types of DRE systems – primarily 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, middleware based on the Real-time CORBA 1.0 (RTC1) standard [1] supports statically scheduled DRE systems (such as avionics mission computing and industrial process controllers) in which task eligibility can be mapped to a fixed set of priorities.

For an important emerging class of open DRE systems (such as adaptive audio/video streaming [2], collaborative mission re-planning [3], and robotics applications designed for close interaction with their environments [4]), however, it is often not possible to know the entire set of application tasks that will run on the system, the loads they will impose on system resources in response to a dynamically changing environment, or the order in which the tasks will execute due to (1) variations in their operating environments, (2) contention with other tasks in the open system, or (3) to the loss or failure of resources (such as individual network links or endsystems). This dynamism can occur because the number of combinations in which application tasks can be mapped to system resources is too large to compute efficiently or because task run-time behaviors are simply too variable to predict accurately. In either case, open DRE systems must be able to adapt dynamically to handle changes in resource availability and QoS requirements.

Assuring effective QoS support in the face of dynamically changing requirements and resources – while also keeping the overhead of that assurance within reasonable bounds – requires a new generation of middleware mechanisms. In particular, there are a number of limitations with applying RTC1 middleware capabilities to open DRE systems with dynamic QoS requirements (e.g., by handling any dynamic variations in task eligibility by manipulating fixed task priorities at run-time), including:

- Limits on the number of priorities supported by common-off-the-shelf (COTS) real-time operating systems that can reduce the granularity at which dynamic variations in task eligibility can be enforced.
- Without middleware-mediated mechanisms for enforcing QoS, the application itself must provide priority manipulation mechanisms, which is tedious and error-prone for DRE application developers.
- Without open, well-defined, and replaceable scheduling mechanisms within the middleware itself, it is hard to improve performance through close integration of middleware features or to achieve sufficient flexibility to customize the policies for QoS enforcement so they meet the needs of each application.

**Solution approach → Adaptive DRE middleware via dynamic scheduling.** 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 define and describe their execution and resource requirements more accurately and adaptively. 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 endsystems they traverse and (2) a *scheduling service architecture* that allows applications to choose which mechanisms enforce task eligibility.

Since the RTC2 specification was only recently integrated into the OMG CORBA standard, there are no commercial products or research prototypes available for it yet. To facilitate the study of standards-based dynamic scheduling middleware, we have therefore implemented a RTC2 framework that enhances on our prior work with The ACE ORB (TAO) [6] (which is a widely-used open-source implementation of Real-time CORBA 1.0 [1]) and its Real-time Scheduling Service (which supports static [7] and dynamic scheduling algorithms [8]). This paper describes how we designed and optimized the performance of our RTC2 Dynamic Scheduling framework to address the following design challenges for adaptive DRE systems:

- Defining a means to install pluggable dynamic schedulers that support more adaptive 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 http://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 Object Request Broker (ORB); Section 3 presents empirical studies of micro-benchmarks conducted to validate our RTC2 approach and to quantify the costs of dynamic scheduling of distributable threads; Section 4 presents empirical results of broader case studies that applied multiple adaptive middleware technologies to DRE avionics applications with real-time image delivery requirements; Section 5 compares our work with related research; and Section 6 offers concluding remarks.

## 2. The Design and Implementation of a Dynamic Scheduling Framework for Real-Time CORBA 2.0

This section describes the key characteristics and capabilities of RTC2 specification and describes how the RTC2 dynamic scheduling framework that we have integrated with the TAO Real-time CORBA ORB helps with the design and implementation of adaptive DRE systems. Figure 1 illustrates the architecture of TAO's RTC2 framework.

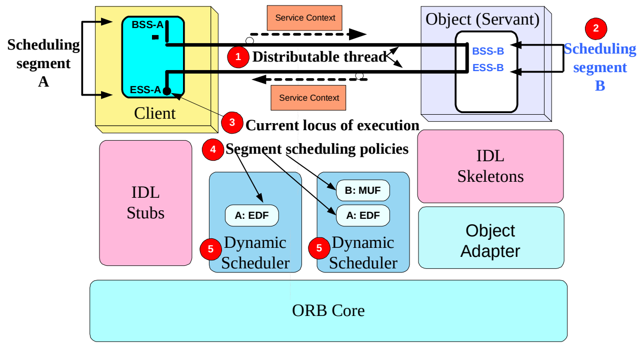


**Figure 1: TAO's RTC2 Architecture [9]**

The key elements of TAO's RTC2 framework shown in Figure 1 are:

1. *Distributable threads*, which applications use to traverse endsystems along paths that can be varied on-the-fly based on scheduling or application level decisions,
2. *Scheduling segments*, which map policy parameters to distributable threads at specific points of execution so that new policies can be applied and existing policies can be adapted at finer granularity,
3. *Current execution locus*, the head of an active distributable thread, which much like the head of an application or kernel thread can take alternative decision branches at run-time based on the state of the application or of the supporting system software,
4. *Scheduling policies*, which determine the eligibility of each thread based on parameters of the current scheduling segment within which that thread is executing, and
5. *Dynamic scheduler*, which reconciles and adapts the set of scheduling policies for all segments and threads on an endsystem, to determine which thread is active.

Distributable threads help to enhance the adaptivity of DRE systems end-to-end since they provide a more effective abstraction for dynamically managing the lifetime of sequential or branching distributed operations. The remainder of this section explains the concepts of *distributable threads* and the adaptive management of their real-time properties through the *pluggable scheduling framework* specified by RTC2. This section also outlines the design of these concepts in TAO's RTC2 framework implementation. We describe *scheduling points*, which govern how and when scheduling parameter values can be mapped to distributable threads. We then 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 safely.

## 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 *static* 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, and the values of at least some of those parameters are likely to vary during system execution. 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 and propagated 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 object methods 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 formalized in the RTC1 specification.

Each distributable thread in RTC2 is identified by a unique system wide identifier called a *Globally Unique Id (GUID )* [10]. A distributable thread may also have one or more execution scheduling parameters, *e.g.*, priority, time-constraints (such as deadlines), and importance. These parameters specify attributes used by RTC2 schedulers for resource arbitration, and also define and convey acceptable end-to-end timeliness bounds 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 a 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 how a local thread makes 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. For each of these topics we identify its relevance to adaptive DRE systems.

**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) [4] scheduling strategy. 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. It is possible to adapt the values of parameters for the current scheduling policy within a given scheduling segment, but to adapt the policy itself, a new scheduling segment must be entered.

**The Current interface.** The `RTScheduling` module's `Current` interface defines operations that begin, update (*e.g.*, to modify scheduling parameter values), 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) [11] on each endsystem along the path of the distributable thread. A nested scheduling segment keeps a reference to the `Current` instance of its enclosing scheduling segment. The following CORBA Interface Definition Language (IDL) fragment shows the operations in the `RTScheduling::Current` interface pertaining to scheduling segments and distributable threads that are implemented in TAO's RTC2 dynamic scheduling framework:

```
local interface RTScheduling::Current :
                RTCORBA::Current {
  void begin_scheduling_segment
    (in string scheduling_segment_name,
     in CORBA::Policy sched_param,
     in CORBA::Policy implicit_sched_param)
  raises(UNSUPPORTED_SCHEDULING_DISCIPLINE);

  void update_scheduling_segment
    (in string scheduling_segment_name,
     in CORBA::Policy sched_param,
     in CORBA::Policy implicit_sched_param)
  raises(UNSUPPORTED_SCHEDULING_DISCIPLINE);
  void end_scheduling_segment
    (in string scheduling_segment_name);

  DistributableThread spawn (
     in ThreadAction start,
     in CORBA::VoidData data,
     in string name,
     in CORBA::Policy sched_param,
     in CORBA::Policy implicit_sched_param,
     in unsigned long stack_size,
     in RTCORBA::Priority base_priority);
};
```

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. The call to `begin_scheduling_segment()` 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), and thus represents a point of fine-grain adaptability of overall system concurrency behavior.

**update_scheduling_segment()** – This operation is a scheduling point the application uses to interact with the RTC2 dynamic scheduler to update the scheduling parameters and 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, *e.g.*, by code outside a distributable thread.

**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 segment. Every `begin_scheduling_segment()` call 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 an `end_scheduling_segment()` operation, the distributable thread is operating with the scheduling parameter of the next outer scheduling segment scope. This nesting of scheduling segments represents an opportunity to manage scheduling behavior adaptively, by applying different scheduling policies for different modes of execution of each distributable thread, and mapping scheduling segments and their associated policies to application modes: *as a distributable thread enters or leaves a particular mode of execution, it simply pushes or pops an appropriate nested scheduling segment*.

**spawn()** – A distributable thread can create a new distributable thread by invoking the `spawn()` operation. If the scheduling parameters for the new distributable thread are 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. The `spawn()` operation therefore represents the creation of a new independently managed schedulable entity. The `name` parameter provides a name for the scheduling segment created by `spawn()`. The `sched_ param` and `implicit_sched_param` parameters provide the scheduling parameters for the new distributable thread. If `sched_param` is null, then the `implicit_sched_param` of the scheduling segment calling `spawn()` will become the new distributable thread's `sched_param.` The `data` parameter passed to the `spawn()` operation is then passed to the following `ThreadAction::do()` method invoked by `spawn()`:

```
module RTScheduling {
  local interface ThreadAction {
     void do(in CORBA::VoidData data);
  };
};
```

`ThreadAction::do()` is the entry point to the new scheduling segment and is called by `spawn()` in the context of the newly created distributable thread.

**Distributable thread location.** Now that we have explained the terminology and interfaces for distributable threads, and the adaptive behaviors that can be designed using those interfaces, 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.

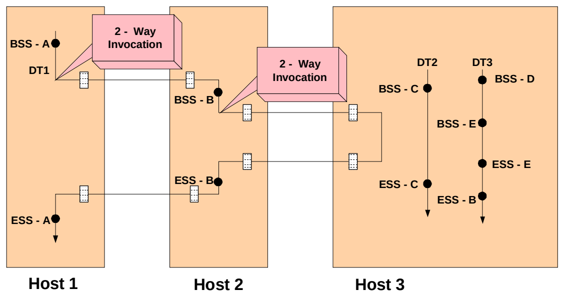


**Figure 2: Distributable Threads and Hosts They Span**

In Figure 2, DT1 makes a twoway 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 oneway invocation. DT4 is destroyed when the upcall completes on Host 2.

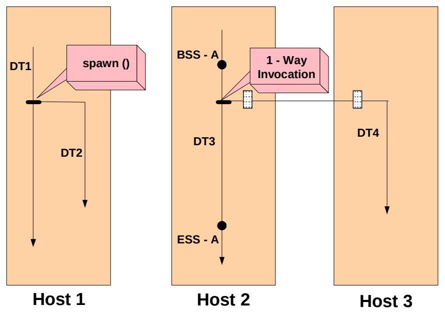


**Figure 3: Ways to Spawn a Distributable Thread**

## 2.2 Pluggable Scheduling

Pluggable scheduling helps to make DRE systems more adaptive since different scheduling strategies can be integrated in response to different application use cases and needs. 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, and yet the conditions under which that occur may vary significantly at run-time. This tension between dynamic environments and predictable resource management necessitates scheduling and dispatching mechanisms for these entities that are (1) based on the real-time requirements of each individual system, and (2) sufficiently flexible to be applied adaptively in the face of varying application requirements and run-time conditions. In the RTC2 specification, a local 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.

Various scheduling disciplines exist that require different scheduling parameters, such as MLF [4], EDF [12], MUF [4], or RMS+MLF [13]. 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 string `“RTScheduler”`. The `RTScheduling::Manager` interface shown below allows the application to install custom dynamic schedulers and obtain a reference to the one currently installed.

```
interface RTScheduling::Manager {
    Scheduler scheduler ();
    void scheduler (Scheduler);
};
```

The `RTScheduler_Manager` object can be obtained via an `ORB:resolve_initial_reference()` call using the string `“RTScheduler_Manager”`. 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.

**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` |

Similarly, the ORB interacts with the RTC2 dynamic scheduler at the specific scheduling points described in Section 2.3 to ensure proper dispatching and sharing of scheduling information across hosts. This is done through dynamic scheduler operations listed under *scheduler upcalls* in Table 2.

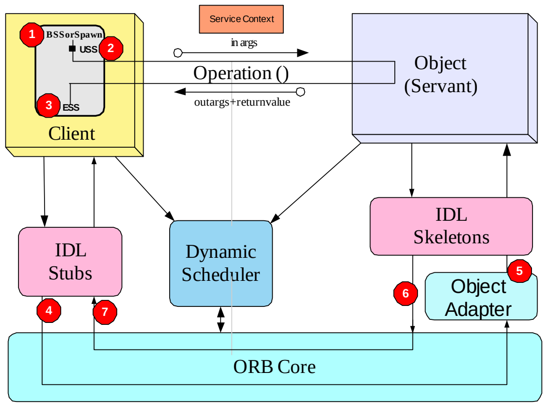
**Table 2: Summary of Scheduler Upcalls for ORB Invoked Scheduling Points**

| *ORB INTERCEPTS* | *SCHEDULER UPCALL* |
|---|---|
| `Outgoing request` | `Scheduler::send_request` |
| `Incoming request` | `Scheduler::receive_request` |
| `Outgoing reply` | `Scheduler::send_reply` |
| `Incoming reply` | `Scheduler::receive_reply` |

## 2.3 Scheduling Points

An application and ORB interact with the RTC2 dynamic scheduler at well-defined points to schedule distributable threads in a DRE system. These points can be defined *a priori* in more static systems, while in adaptive systems, the traversal of these points can be placed under adaptive control by the distributable threads, as Section 2.1 described.

These scheduling points allow an application and ORB to provide the RTC2 dynamic scheduler up-to-the-instant information about the competing tasks in the system, so it can make scheduling decisions in a consistent and predictable but also adaptive manner. We now describe these scheduling points, which are illustrated in Figure 4.



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 [9]**

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

**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 schedules the new scheduling segment based on its parameters and those of the active scheduling segments for other distributable threads in the system. This occurs whenever 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()` operation is invoked by a distributable thread to adapt its scheduling parameters, it updates scheduling parameters of the corresponding scheduling segment by making a call to `Scheduler::update_scheduling_segment()`.

**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 dynamic scheduler then reverts the thread to its original scheduling parameters. If a nested scheduling segment is terminated the dynamic scheduler invokes the `Scheduler::end_nested_scheduling_segment()` operation. 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 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 allows the application to inform the RTC2 dynamic scheduler that a distributable thread has been cancelled.

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 twoway 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 oneway 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 for collocated oneways, (2) scheduling overhead and complexity, (3) lack of interceptor support for collocated oneways, 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 oneway 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 recreate 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 twoway operations, and (e) to perform cleanup operations on the client when the replies from twoway 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, 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 schedul-ing 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 twoway 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 twoway 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

To manage adaptive behavior of distributable threads efficiently, predictably, and correctly, an RTC2 framework must resolve a number of design challenges. Specifically, when distributable threads are involved, two key adaptation strategies are problematic:

- Transferring ownership of storage, locks and other reserved system resources, and
- Truncating the execution of tasks that are no longer relevant or that risk interfering with system correctness.

Below we examine two technical challenges we faced when implementing RTC2 distributable threads in TAO: (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 Managing Distributable vs. OS Thread Identity*

**Context.** A key design issue with the RTC2 specification is that in modern ORB middleware with alternative concurrency strategies [14], a distributable thread may be mapped on each endsystem to several different OS threads over its lifetime.

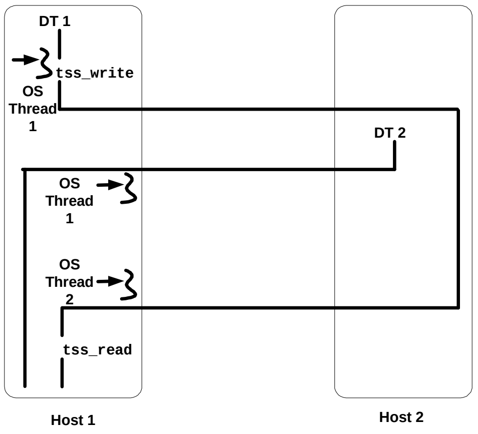


**Figure 5: TSS with Distributable Threads**

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 migrates back to endsystem A, DT2 migrates from endsystem B to endsystem A. For efficiency, flexible concurrency strategies such as thread pools [15] may map distributable threads to whatever local threads are available. For example, Figure 5 shows DT2 mapped to OS thread 1 and when DT1 migrates back to endsystem A it is mapped to OS thread 2.

**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. We quantify the cost of this approach in our experimental results presented in Section 3.1.

### *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's interface provides a `cancel()` operation that can be invoked to stop the corresponding distributable thread. 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.

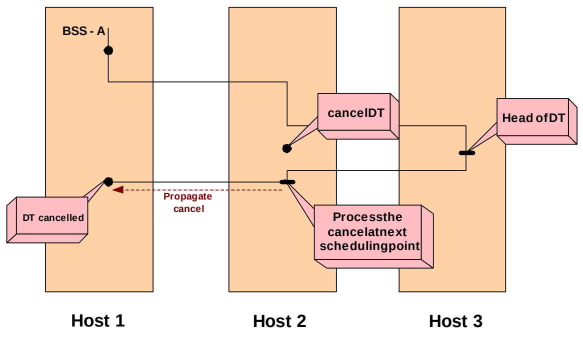


**Figure 6: Distributable Thread Cancellation [9]**

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 the distributable thread is cancelled, 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.

## 3. Empirical Evaluation of Real-Time CORBA 2.0 Dynamic Scheduling in TAO

This section presents the results of micro-benchmarks of our RTC2 implementation in TAO described in Section 2. Section 4 then present the results of a two broader case studies based on a production DRE application. The studies in this section serve primarily to quantify the overhead incurred by TAO's RTC2 dynamic scheduling framework. We first describe experiments that quantify the cost to support a representative thread-identity-aware mechanism based on thread specific storage (TSS) from the OS to the middleware. We then describe experiments that evaluated alternative mechanisms for scheduling distributable threads according to their *importance*.

## 3.1 Performance Overhead of Thread ID Management

**Experiment overview and configuration.** Section 2.4.1 describes the challenges associated with emulating distributable thread identity via thread-specific storage in middleware, rather than using OS-level TSS support. To quantify the additional overhead of TSS support in middleware, we conducted several experiments to compare and contrast the cost of creating TSS keys, and writing and reading TSS data on a single endsystem.

The experiments were conducted on a single-CPU 2.8 GHz Pentium 4 machine with 512KB cache and 512Mb RAM, running Red Hat Linux 9.0 (2.4.18 Kernel) with the KURT-Linux patches and using ACE version 5.3.2. The experiments were run as root, in the real-time scheduling class, and the experimental data were collected using the ACE high resolution timer. Experiments to assess the cost of TSS key creation were run by iteratively creating 500 different keys and measuring the time it took to create each one. Experiments to assess the cost of TSS write and read operations were run by repeatedly writing and then reading from one storage location associated with a single TSS key.

**Empirical results.** Figure 7 shows that the cost of creating the TSS keys in middleware was noticeably higher than the cost of creating TSS keys in the OS. Moreover, the slope at

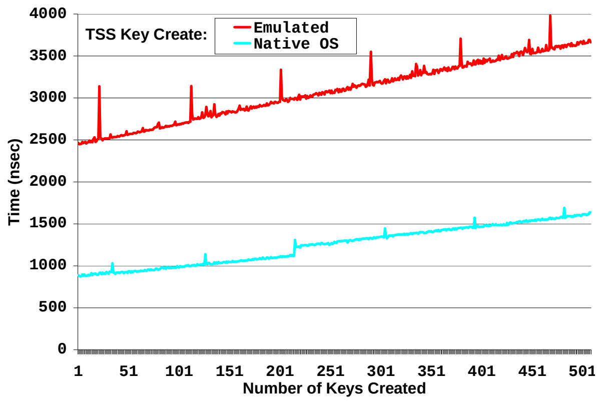


**Figure 7: TSS Key Creation Latency**

which the cost of key creation in middleware increased with each additional key was higher than the slope at which the cost of each additional key increased with OS-level TSS support. Similarly, Figure 8 shows that the cost of `read()` and `write()` operations in middleware TSS emulation were noticeably higher than in OS-level TSS.

**Analysis of the results.** Although the costs of middleware TSS emulation was higher (as is to be expected), these results also show that the cost of middleware emulation of TSS falls within reasonable limits. For key creation, both OS and middleware TSS support show linear cost increases with additional keys, and for read and write operations the cost of each read or write remains essentially constant over multiple iterations. Moreover, the total cost of create, read, and write operations is very small (< 4 usec even to create the 500th key) compared to the time-scales on which distributable threads operate (on the order of seconds) in the empirical studies described next in Section 3.2.

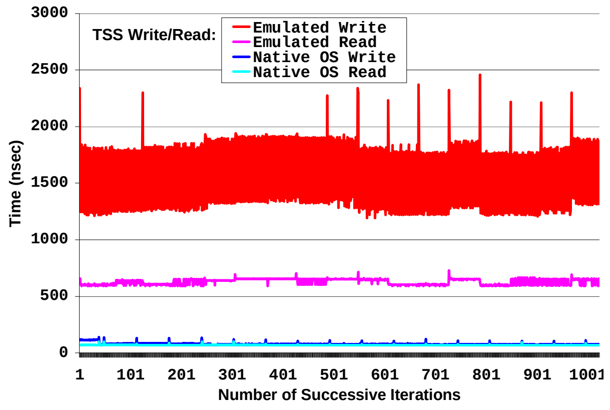


**Figure 8: TSS Write and Read Latency**

## 3.2 Dynamic Scheduling Performance

**Experiment overview and configuration.** To examine the ability of different scheduling mechanisms to respond adaptively to changes in parameters like task importance, we plugged two different implementations of the `Scheduler` interface – an *OS Thread Priority Scheduler* and a *Most Important First Scheduler* – into the TAO ORB version 1.3.2 to test the behavior of its RTC2 dynamic scheduling framework with different scheduling strategies. Both implementations of the `Scheduler` interface use a scheduling strategy that prioritizes distributable threads according to their importance.

The *OS Thread Priority (TP) Scheduler* is an RTC2 dynamic scheduler implementation that schedules the distributable threads by mapping each one's dynamic importance to native OS priorities. The onus of dispatching the distributable threads is thus delegated to the OS-level thread scheduler, according to the native OS priorities assigned to the local threads to which the distributable threads are mapped. The GUID of a migrating distributable thread is propagated in the GIOP service context. The importance of a migrating distributable thread is carried across the endsystems it traverses using the RTC1 `CLIENT_PROPAGATED` priority propagation model. It is possible for the application to change a distributable thread's importance dynamically. However, the resolution with which the TP Scheduler can enforce the ordering of distributable threads by their importance parameters is limited by the number of available OS priorities.

The *Most Important First (MIF) Scheduler* illustrated in Figure 9 is our RTC2 dynamic scheduler implementation that schedules distributable threads by their importance, with the most important thread scheduled to execute at any given time. The application again specifies the importance of the thread as a scheduling parameter to the MIF Scheduler. The MIF Scheduler maintains a ready queue that stores distributable threads in order of their importance, with the most important distributable thread that is ready for execution at head of the queue. 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 (which implies that it is the next to be executed), the MIF Scheduler signals the corresponding condition variable on which the local thread is waiting, and hence awakens it. With the MIF Scheduler, both the importance and GUID of the distributable thread are propagated in the GIOP service context.

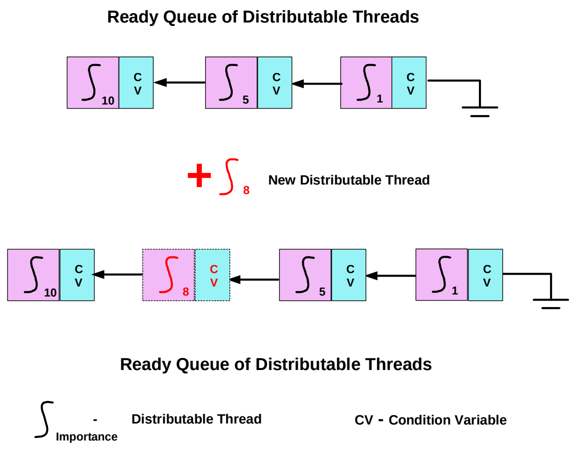


**Figure 9: MIF Scheduler - Insertion into Ready Queue**

The experimental configuration we used to examine both the TP 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: the execution times for local work before and after the remote invocation are separated by a '+'.

**Empirical results.** Figure 10 and Figure 11 show how distributable threads are scheduled dynamically as they enter and leave the system across multiple hosts by the TP Scheduler and the MIF Scheduler respectively. The start times of the distributable threads in both graphs are offset from T=0 by less than 1 sec, which is the time taken to initialize the experiments and start the distributable threads. Since both the TP and MIF schedulers use the same scheduling *policy* based on the importance of the distributable threads, the graphs are nearly identical. The graphs show the run-time distributable thread schedules on both the hosts. On host 1, DT1 and DT2 start at time (T) = 0 and DT3 starts at T=12. Since DT1 is of higher importance than DT2 it is scheduled to run first. On host 2, DT4 starts at T=0 and is scheduled for execution as DT5 is not ready to run till T=9.

**Table 3: Distributable Thread Schedule on Host 1**

| GUID | Start Time (secs) | Importance | Execution Time (secs) | | Span |
|---|---|---|---|---|---|
| | | | Local | Remote | |
| 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 |
|---|---|---|---|---|---|
| | | | Local | Dist | |
| 4 | 0 | 5 | 9 | N/A | Local |
| 5 | 9 | 7 | 3 | N/A | Local |

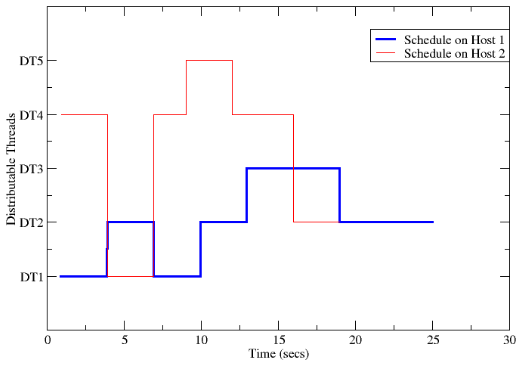


**Figure 10: OS Thread Priority Scheduler Graph**

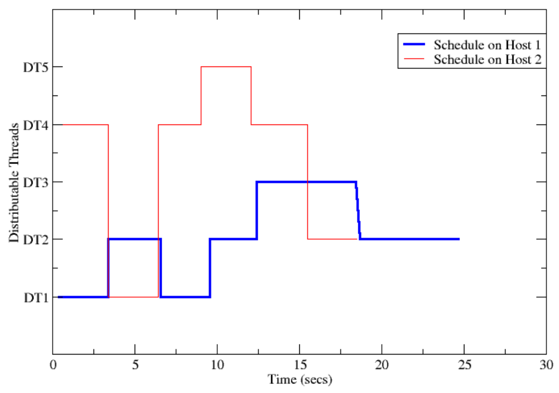


**Figure 11: MIF Scheduler Graph**

After executing for 3 secs DT1 makes a two-way operation invocation on host 2 and waits for a reply. DT4 is suspended to allow DT1 to execute on host 2 as DT1 is of higher importance. When DT1 is executing on host 2 the respective RTC2 dynamic scheduler on host 1 continues to schedule DT2 on host 1 as it has the highest importance on host1. DT1 completes execution on host 2 after T=3 and returns to host 1 and resumes execution, for T=3, after pre-empting DT2. DT4 resumes on host 2. DT1 completes its execution cycle of T=9 on host 1. Hence, DT2 is scheduled on host 1 for the next 3 secs. On host 2 DT5 enters the system at T=9. DT5 pre-empts DT4 as it is of higher importance and executes for 3 secs.

At T=12 DT2 has completed 6 secs of local execution and makes a twoway invocation to host 2. DT3 enters the system at T=12 and is scheduled for execution for 3secs. On host 2, DT5 completes its cycle of execution and DT4 is scheduled. DT2 does not get to execute immediately on host 2 as DT5 is of higher importance. After DT4 completes execution on host 2, DT2 is scheduled on host 2 for the next 3 secs. DT3 continues execution on host 1. DT2 completes execution on host 2 and returns to host 1. DT3 completes its cycle of execution and DT2 is scheduled till its cycle of local execution is complete.

**Analysis of the results.** The experimental results of the TP and MIF schedulers demonstrate 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 TP and MIF schedulers schedule the distributable threads based on their importance, both the graphs are nearly identical. The one small but important difference is 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 application level) compared to the TP 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. Empirical Case Studies of Adaptive Scheduling in Real-Time Avionics Systems

Although the micro-benchmarks in Section 3 help to quantify the behavior of TAO's RTC2 dynamic scheduling framework in a controlled environment, to truly evaluate how to apply adaptive dynamic scheduling techniques effectively in complex DRE systems requires case studies of actual systems. This section therefore describes two case studies that apply our dynamic scheduling middleware in the context of real-time avionics computing systems developed by The Boeing Company, running on production computing, communication, and avionics hardware. The first case study examines the effects of applying the cancellation mechanisms described in Section 2.4.2 to non-critical real-time operations. The second case study examines the performance of adaptive operation rate re-scheduling within a multi-layered resource management architecture for real-time image transfer.

### 4.1 Case Study 1 → Effects of Cancellation of non-Critical Operations

**Case study overview and configuration.** Many complex DRE systems perform a mixture of critical and non-critical real-time operations, for which it is desirable to maximize the ability of non-critical operations to meet their deadlines, while ensuring that *all* critical operations also meet their deadlines. When the CPU is overloaded (which can happen all too readily in open systems in dynamic operating environments), canceling some operations so that others are more likely to meet their deadlines is an important strategy for ensuring best use of the CPU resource. In this case study, we used an Operational Flight Program (OFP) system architecture based upon commercial hardware, software, standards, and practices that supports re-use of application components across multiple client platforms. The OFP is primarily concerned with integrating remote sensor and actuator systems throughout the aircraft with the cockpit information displays and controls used by the pilot and other aircraft personnel.

The system architecture for our first case study included an OFP consisting of approximately 70 operations, the Bold Stroke avionics domain-specific middleware layer [16] built upon TAO, the TAO Dynamic Scheduling Service, and the TAO Real-Time Event Service [17], configured for various scheduling strategies described in Sections 2 and 3. This middleware isolates applications from the underlying hardware and OS, enabling hardware or OS advances from the commercial marketplace to be integrated more easily with the avionics application. We conducted measurements of two key areas of resource management: *cancellation of non-critical operations* that are at risk of missing their deadlines, and *protecting critical operations*. The analysis below features a comparison of two canonical scheduling strategies, the hybrid static/dynamic Maximum Urgency First (MUF) [4] strategy (which assigns operations to strict priority lanes according to their criticality and the schedules them dynamically within each lane according to laxity) and the static Rate Monotonic Scheduling (RMS) [12] strategy (which assigns operations to strict priority lanes according to their rates of invocation, and schedules each lane in FIFO order). Measurements were made on 200 MHz Power PC Single Board Computers running the VxWorks 5.3 operating system.

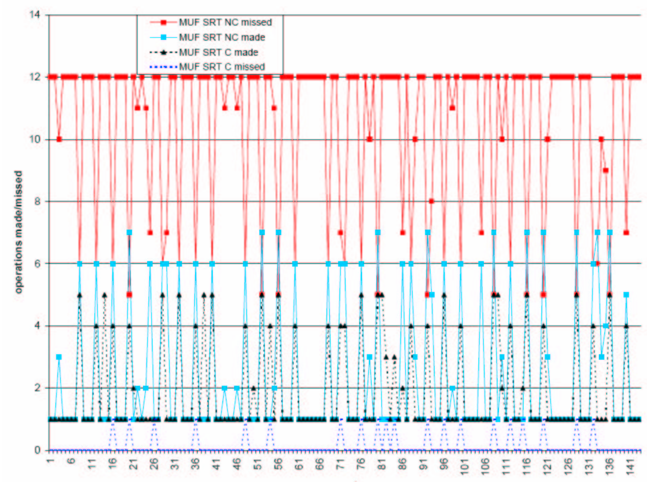


**Figure 12: Effects of non-Critical Operation Cancellation**

**Operation cancellation.** Figure 12 shows the effects of cancelling non-critical operations in the MUF hybrid static/dynamic scheduling strategy in conditions of CPU overload. Operation cancellation can potentially help reduce the amount of wasted work performed in operations that miss their deadlines. Assuming there is no residual value of an operation that completes past its deadline, this wasted time increases the amount of unusable overhead. We observed that while the MUF strategy with operation cancellation was more effective in limiting the number of operations that were dispatched and then missed their deadlines, the number of operations that made their deadlines in each case was comparable. We attribute this to the short execution times of several of the non-critical operations. In fact, the variation with cancellation had slightly lower numbers of non-critical operations that were successfully dispatched, as operation cancellation is necessarily pessimistic.

**Protecting critical operations.** We also compared the effects of non-critical operation cancellation on critical and non-critical operations in overload, in the hybrid static/dynamic MUF scheduling strategy and the static RMS strategy. Figure 13 shows the number of deadlines made and missed for each strategy.

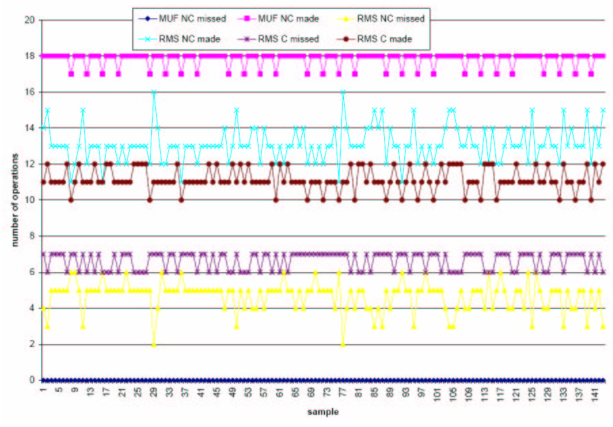


**Figure 13: Effects of Cancellation under Overload Conditions**

With no operation cancellation, MUF met all of its deadlines, while RMS missed between 2 and 6 critical operations per sample. Furthermore, MUF successfully dispatched additional non-critical operations. We investigated whether adding operation cancellation might have reduced the number of missed deadlines for critical operations with RMS, by reducing the amount of wasted work. However, it appears that the overhead of operation cancellation in fact makes matters worse, with between 6 and 7 misses per sample. We interpret this to mean that there were few opportunities for effective non-critical operation cancellation in RMS under the experimental conditions.

**Analysis of the case study.** The results from this case study offer the following insights about the use of adaptive middleware to support DRE systems more effectively. First, operation cancellation can be an effective way to shed tasks that cannot meet their deadlines during resource overload. Second, these results indicate that hybrid static/dynamic scheduling strategies are more likely to benefit from operation cancellation than purely static ones – because hybrid static-dynamic scheduling strategies prioritize critical tasks as a whole over non-critical ones, the availability of the CPU to non-critical tasks is more variable and more sparse so that more non-critical tasks are likely potential candidates for cancellation.  Moreover, because operation cancellation is necessarily pessimistic, it is essential to avoid overestimating the risk of operations missing their deadlines, which can result in overly aggressive cancellation degrading rather than improving the overall performance of the system.  As is true for many adaptive resource management techniques, the more accurate the information that the cancellation mechanism has about deadline failure risks, the more accurate its cancellation decisions and the better its effect on overall system performance.

## 4.2 Case Study 2 → Adaptive Scheduling Behavior in Multi-level Resource Management

**Case study overview and configuration.** Our second case study examines the performance of adaptive rescheduling of operation rates within the context of layered multi-level resource management [20]. We have applied the layered resource management architecture shown in Figure 14 to provide an open systems "bridge" between legacy on-board embedded avionics systems and off-board information sources and systems. The foundation of this bridge is the interaction of two Real-time CORBA [1] ORBs (TAO and ORBExpress) using a pluggable protocol to communicate over a very low (and variable) bandwidth Link-16 data network. We then applied several middleware technologies in higher architectural layers to manage key resources and ensure the timely exchange and processing of mission critical information. In combination, these techniques support browser-like connectivity between server and client nodes, with the added assurance of real-time performance in a highly resource-constrained and dynamic environment. The evaluation system described in this section leverages existing open middleware platforms similar to those described in Section 4.1.

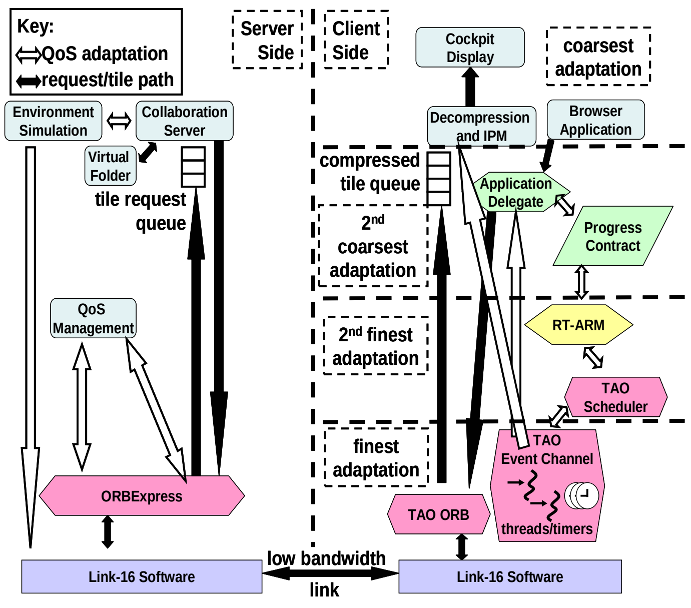


**Figure 14: Multi-Level Resource Management Model**

**System resource management model.** The resource management model for the evaluation system in this case study is illustrated in Figure 14. When a client-side operator requests an image, that request is sent from the *browser application* to an *application delegate* [20], which then sends a series of requests for individual tiles via TAO over a variable low-bandwidth Link-16 connection to the server. The delegate initially sends a burst of requests to fill the server request queue; it then sends a new request each time a tile is received. For each request, the delegate sends the tile's desired compression ratio, determined by the progress of the overall image download when the request is made.

On the server, an *ORBExpress/RT* Ada ORB [22] receives each request from the Link-16 connection, and from there each tile goes into a *queue of pending tile requests*. A collaboration server pulls each request from that queue, fetches the tile from the server's *virtual folder* containing the image, and compresses the tile at the ratio specified in the request. The collaboration server then sends the compressed tile back through ORBExpress and across Link-16 to the client. Server-side environmental simulation services emulate additional workloads that would be seen on the command and control (C2) server under realistic operating conditions. Back on the client, each compressed tile is received from Link-16 by TAO and delivered to a servant that places the tile in a queue where it waits to be processed. The tile is removed from the queue, decompressed, and then delivered by client-side operations to Image Presentation Module (IPM) hardware which renders the tile on the cockpit display. The decompression and IPM delivery operations are dispatched by a TAO Event Service [17] at rates selected by the TAO Dynamic Scheduling Service.

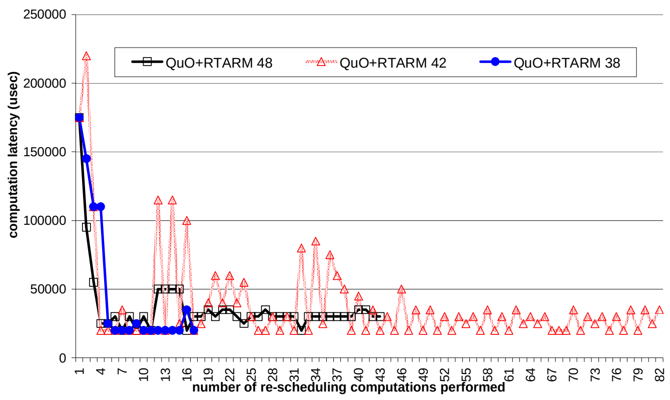


**Figure 15: Adaptive Schedule Computation Latency**

**Schedule re-computation latency.** We measured schedule re-computation overhead resulting from priority and rate reassignment by the TAO Dynamic Scheduling Service. Figure 15 plots schedule re-computations while the system is performing adaptation of both image tile compression and decompression and IPM operation rates, at deadlines for downloading the entire image of 48, 42, and 38 seconds. The key insight from these results is that the number and duration of re-scheduling computations is both (1) reduced overall compared to our earlier results [19] and (2) proportional to the degree of rate adaptation that is useful and necessary for each deadline.

The main feature of interest in Figure 15 is the downward settling of schedule computation times, as the ranges of available rates is narrowed toward a steady-state solution and the input set over which the scheduler performs its computation is thus reduced. We also observed an interesting *phase transition* in the number of re-computations between the infeasible and barely feasible deadlines. If we arrange trials in descending order according to the number of re-computations in each, we get 42, 46, 48, 50, 52, 54, and then 58 seconds, and then finally 38 second and 1 second deadlines showed the same minimal number of computations. The duration of the experiment for the 42 second deadline was comparable to that for other deadlines.

**Analysis of the case study.** This second case study demonstrates that adaptive rescheduling techniques can be applied to adjust the rates of operation invocation at run-time in response to dynamically varying environments. The convergence of the scheduling behavior toward lower latencies and smaller input sets is a good example of desirable adaptation performance in DRE systems. As future work, we are investigating questions of convergence and stability raised by this case study, by applying formal control theory to guide the adaptation of operation rates, image tile compression ratios, and other factors relevant to adaptive DRE systems.

## 5. Related Work

The *Quality Objects* (QuO) distributed object middleware is developed at BBN Technologies [20]. QuO is based on CORBA and provides the following support for agile applications running in wide-area networks: (1) *run-time performance tuning and configuration* through the specification of *QoS regions*, behavior alternatives, and reconfiguration strategies that allows the QuO run-time to adaptively trigger reconfiguration as system conditions change (represented by transitions between operating regions) and (2) *feedback* across software and distribution boundaries based on a control loop in which client applications and server objects request levels of service and are notified of changes in service. We have integrated our earlier dynamic scheduling service (Kokyu [23]) with the QuO framework and plan future integration of our RTC2 framework with QuO.

The *Realize* project at UCSB has developed an approach based on object migration and replication, to improve performance of soft real-time distributed systems [24], [25]. This approach constitutes a higher level of adaptive control for soft real-time QoS management, and is complementary to our RTC2 framework. In particular, a system developer might apply Realize to provide soft real-time load balancing across endsystems, using our RTC2 framework to integrate scheduling and dispatching of distributable threads that transit those endsystems.

The *Time-triggered Message-triggered Objects* (TMO) project [26] at the University of California, Irvine, supports the integrated design of distributed OO systems and real-time simulators of their operating environments. The TMO model provides structured timing semantics for distributed real-time object-oriented applications by extending conventional invocation semantics for object methods, i.e., CORBA operations, to include (1) invocation of time-triggered operations based on system times and (2) invocation and time bounded execution of conventional message-triggered operations. TMO is a compatible technology to our RTC2 framework, particularly for the open area of research of time-triggered management of distributable threads.

The RTC2 specification allows pluggable dynamic schedulers. However, this means that endsystems, or even segments within an endsystem, along an end-to-end path could be applying differing scheduling disciplines and scheduling parameters. For example, one endsystem could order the eligibility of distributable threads per the EDF scheduling discipline using deadlines, and another per the MUF scheduling discipline using criticality, deadlines, and execution times. RTC2 does not address the issue of interoperability of schedulers on the endsystems that a distributable thread spans. Juno [27], a meta-programming architecture for heterogeneous middleware interoperability, addresses the above issues. It formalizes the above problems, defines formalisms to express different instances of the problem and maps the formalized abstractions to a software architecture based on Real-time CORBA.

## 6. Concluding Remarks

The OMG 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 strategies, such as EDF, MLF, MUF, or RMS+MLF, as well as the TP and MIF strategies described in Section 3.

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 from 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 middleware scheduling frameworks [7] [8] [23] 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 2.4.
- System-wide dynamic scheduling is not yet as pervasive as fixed-priority static scheduling in practice, which has limited the scope of the RTC2 specification. In particular, it does not yet 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.
- Empirical case studies based on actual DRE systems (such as those presented in Section 4) are essential to (1) understand how techniques such as cancellation and adaptive rescheduling can be applied effectively in complex DRE systems and (2) determine the appropriate role of RTC2 mechanisms with respect to other middleware mechanisms that could be used alternately.

As future work, we are investigating what additional gains in QoS assurance and efficiency of resource management in open DRE systems can be achieved by integrating diverse scheduling policies and mechanisms within our RTC2 framework, according to the semantics of each particular open DRE system application. Our preliminary results [28] indicate that even more customized forms of scheduling can be achieved efficiently within the RTC2 framework.

## References

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

[2] D. Karr, C. Rodrigues, Y. Krishnamurthy, I. Pyarali, and D. Schmidt, “Application of the QuO Quality-of-Service Framework to a Distributed Video Application,” *DOA*, Rome, Italy, Sept. 2001.

[3] D. Corman, J. Gossett, D. Noll, “Experiences in a Distributed, Real-Time Avionics Domain - Weapons System Open Architecture, ISORC, Washington DC, April 2002.

[4] D. Stewart and P. 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, *September 2001*, www.omg.org/docs/ptc/01-08-34.pdf

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

[7] C. Gill, D. Levine, D. Schmidt, “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] C. Gill, D. Schmidt, and R. Cytron, “Multi-Paradigm Scheduling for Distributed Real-Time Embedded Computing”, IEEE Proceedings 91(1), Jan 2003.

[9] Y. Krishnamurthy, C. Gill, D. Schmidt, I. Pyarali, L. Mgeta, Y. Zhang, and S. Torri, “The Design and Implementation of Real-Time CORBA 2.0: Dynamic Scheduling in TAO”, RTAS 2004, Montreal, Canada, May 2004.

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

[11] D. C. Schmidt, M. Stal, H. Rohnert, and F. Buschmann, "Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects, Volume 2", Wiley, NY, 2000.

[12] C. Liu, J. Layland, “Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment,” *JACM*, vol. 20, January 1973.

[13] Chung, Liu, Lin, “Scheduling Periodic Jobs that Allow Imprecise Results,” *IEEE Transactions on Computers*, vol. 39, Sept 1990.

[14] I. 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.

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

[16] D. Sharp, “Reducing Avionics Software Cost Through Component Based Product Line Development”, Software Technology Conference, April 1998.

[17] T. Harrison, D. Levine, and D. Schmidt, “The Design and Performance of a Real-time CORBA Event Service,” OOPSLA ’97, Atlanta, GA, ACM, October, 1997.

[18] Huang, Jha, Heimerdinger, Muhammad, Lauzac, Kannikeswaran, Schwan, Zhao, and Bettati, “RT-ARM: A Real-Time Adaptive Resource Management System for Distributed Mission-Critical Applications", Workshop on Middleware for Distributed Real-Time Systems, IEEE RTSS, San Francisco, California, 1997.

[19] B. Doerr, T. Venturella, R. Jha, C. Gill, and D. Schmidt, “Adaptive Scheduling for Real-time, Embedded Information Systems,” *18th IEEE/AIAA DASC*, St. Louis, Oct. 1999.

[20] J. Gossett, C. Gill, J. Loyall, D. Schmidt, D. Corman, R. Schantz, and M. Atighetchi, “Integrated Adaptive QoS Management in Middleware: A Case Study”, RTAS 2004, Montreal, Canada, May 2004.

[21] J. Zinky, D. Bakken, and R. Schantz, “Architectural Support for Quality of Service for CORBA Objects,” *Theory and Practice of Object Systems*, vol. 3, no. 1, pp. 1–20, 1997.

[22] Objective Interface, “ORBExpress”, *www.ois.com*

[23] J. Loyall, J. Gossett, C. Gill, R. Schantz, J. Zinky, P. Pal, R. Shapiro, C. Rodrigues, M. Atighetchi, and D. Karr, “Comparing and Contrasting Adaptive Middleware Support in Wide-Area and Embedded Distributed Object Applications,” in *Proceedings of the 21st International Conference on Distributed Computing Systems (ICDCS-21)*, pp. 625–634, IEEE, April 2001.

[24] V. Kalogeraki, P. M. Melliar-Smith, and L. E. Moser, “Dynamic Migration Algorithms for Distributed Object Systems, 21st IEEE International Conference on Distributed Computing Systems (ICDCS), Phoeniz, AZ, April 2001.

[25] V. Kalogeraki, P. M. Melliar-Smith, and L. E. Moser, “Dynamic Scheduling of Distributed Method Invocations,” in *21st IEEE Real-Time Systems Symposium*,(Orlando, FL), IEEE, November 2000.

[26] K. H. K. Kim, “Object Structures for Real-Time Systems and Simulators,” *IEEE Computer*, pp. 62–70, Aug. 1997.

[27] A. Corsaro, D. Schmidt, C. Gill, and R. Cytron, “Formalizing Meta-Programming Techniques to Reconcile Heterogeneous Scheduling Policies in Open Distributed Real-Time Systems”, 3rd Int. Symp. on Distributed Objects and Applications, September 2001, Rome, Italy.

[28] M. Frisbee, D. Niehaus, V. Subramonian, and C. Gill, “Group Scheduling in Systems Software”, 12th Intl. Workshop on Parallel and Distributed Real-Time Systems, April 2004, Santa Fe, NM