

# Real-time CORBA 2.0: Dynamic Scheduling

Yamuna Krishnmaurthy  
OOMWorks LLC  
Metuchen, NJ  
[yamuna@oomworks.com](mailto:yamuna@oomworks.com)

October 15, 2002

## Abstract

This paper details the design for implementing dynamic scheduling in TAO. This will include implementing the interfaces defined by OMG's Real-time CORBA 2.0 specification (RTC2) [1] and integrating the Kokyu Scheduling Framework.

## Keywords

Real-time CORBA, Dynamic Scheduling, Distributable Threads, Kokyu Scheduler.

## 1. Motivation

Real-time distributed systems are of two types (1) *Static* and (2) *Dynamic*. *Static* distributed real-time systems are those where the set of applications that will run in the system, their load and lengths of execution are known. Hence, these applications can be scheduled a priori and the feasibility of that schedule can be checked before the system is active. *Dynamic* real-time distributed systems, however, may not have the knowledge of the applications that will run on their system or the order in which they will be executed. This may be because the number of applications is too large or their processing requirements are too variable. In this case, the system should be able to adapt to real-time requirements in a dynamically changing environment. RTC2 addresses the latter class of real-time systems

## 2. From RT-CORBA 1.0 to RT-CORBA 2.0

RT-CORBA 1.0 (RTC1) [2] lends itself to static or fixed priority real-time systems but fails to accommodate the requirements for dynamic real-time systems. RTC2 attempts to overcome this shortcoming by extending RTC1 to provide interfaces and mechanisms to plug in dynamic schedulers and interact with them. It allows the user to specify and use the scheduling discipline and scheduling parameters that most accurately define and describe the application execution and resource requirements. RTC2 specifically introduces two new concepts (1) *Distributable Thread* and (2) *Scheduling Service Architecture*.

### 2.1 Distributable Thread

The fundamental requirement for a real-time application is that end-to-end timeliness requirements be explicitly employed for resource management. This should be done consistently on each distributed node that the application traverses.

In real-time CORBA systems, application end-to-end timeliness requirements must be acquired from the client, propagated with

the invocation, and deposited in the servant. For dynamic real-time systems, the fixed priority propagation in RTC1 is not sufficient. A natural abstraction is suggested by CORBA's control flow programming model – a thread that can execute operations in objects without regard for physical node boundaries. In RTC2, this programming model abstraction is termed the *Distributable Thread (DT)*. It replaces the concept of an activity that was introduced by in RTC1. The DT is a schedulable entity.

Each DT has a unique system wide id. It may have one or more execution scheduling parameters, e.g., priority, time-constraints such as deadlines, and importance. These scheduling parameters specify the acceptable end-to-end timeliness for completing the sequential execution of operations in object instances that may reside on multiple physical nodes. Within each node, the flow of control of the DT is equivalent to normal local thread execution. Every DT has only one execution point at any given instance in the whole system, i.e., a DT cannot simultaneously execute on multiple nodes it spans.

Scheduling segment is a code sequence whose execution needs to be scheduled as per the scheduling parameters specified by the application. The DT is the loci of execution that spans multiple nodes and scheduling segments.

The following figures show the different spans of DTs:

BSS - begin\_scheduling\_segment      ● Application Call  
ESS - end\_scheduling\_segment      ■ Interceptor Call

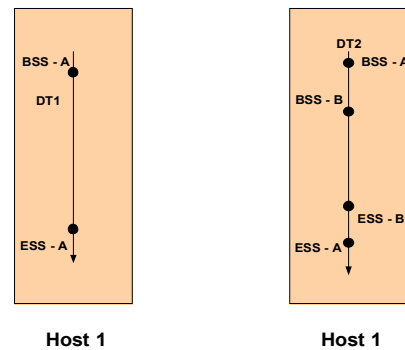
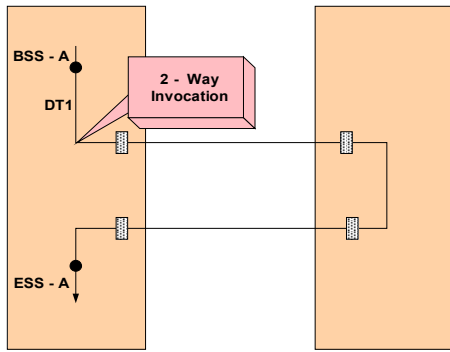


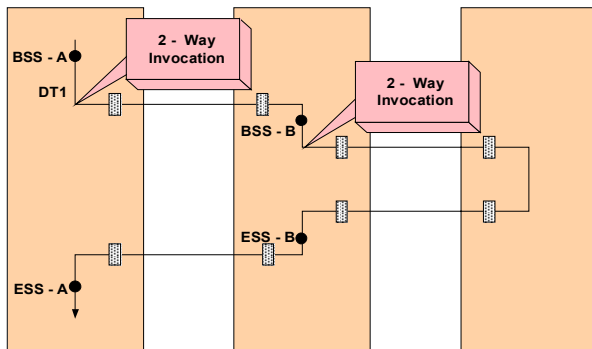
Figure 1: Distributable Thread on Single Host

Figure 1 - DT1 is a simple DT that does not traverse node boundaries. DT2 does not traverse node boundaries, but it has a nested scheduling segment (BSS-B to ESS-B).



Host 1 Host 2

(a)

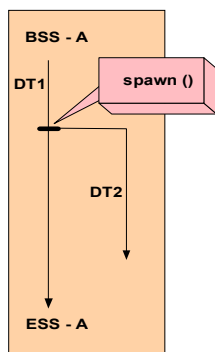


Host 1 Host 2 Host 3

(b)

Figure 2: Distributable Thread on Multiple Hosts

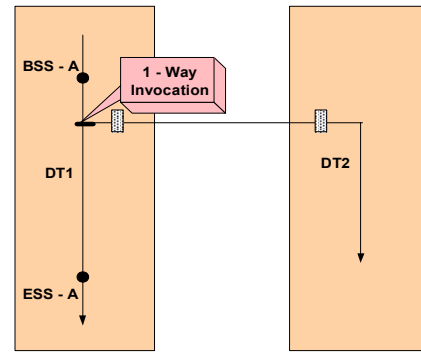
Figure 2 – (a) DT1 makes two-way invocation on an object on a different host. (b) DT1 makes a two-way invocation on an object on a different host. It also has a nested segment started on Host2 (BSS-B to ESS-B).



Host 1

Figure 3: Distributable Thread Spawn

Figure 3 – DT2 is created with a `spawn()` operation. `spawn()` can be called only within the context of another scheduling segment. It creates a new native thread that runs as a new DT (DT2) with a new guid. `ThreadAction::do()`, described in Section 3.1.2.1, is the entry point for the new DT created.



Host 1

Host 2

Figure 4: Distributable Thread making One-Way Invocation

Figure 4 – DT2 is implicitly created on Host 2 to service a one-way invocation. It has a new guid that is created just before the request is sent out.

## 2.2 Scheduling Service Architecture

RTC2 defines a Scheduling Service interface described in Section 3.4 that provides a mechanism for plugging-in different schedulers. The application passes its scheduling requirements to the scheduler using these interfaces. Similarly, the ORB also interacts with the scheduler at specific scheduling points for proper dispatching and sharing of scheduling information across nodes. These scheduling points are described in Section 3.7.

## 3. Design of the Dynamic Scheduling Service

The following sections describe the syntax and semantics of the various interfaces specified by RTC2 that will be implemented in TAO to support dynamic scheduling. TAO specific interfaces are also described that are required to address areas underspecified by RTC2.

### 3.1 RTScheduling::Current Interface

The `RTScheduling::Current` interface derives from the `RTCORBA::Current` interface.

```
module RTScheduling
{
    local interface Current : RTCORBA::Current
```

```
{
}
}
```

Each distributable thread has a unique instance of the current object managed in TSS (Thread Specific Storage). It will support nested scheduling segments by keeping a reference to the previous current instance. The following subsections describe the various operations defined in the `RTScheduling::Current` interface.

### 3.1.1 Id Related Operations

```
module RTScheduling
{
  local interface Current : RTCORBA::Current
  {
    typedef sequence<octet> IdType;

    // a globally unique id
    readonly attribute IdType id;

    // This operation is redundant as the
    // above id attributes serves the
    // same purpose of accessing the id
    IdType get_current_id();

    // returns a null reference if
    // the distributable thread is
    // not known to the local scheduler
    DistributableThread lookup(in IdType id);
  };
};
```

**lookup()** - This operation returns a reference to the distributable thread interface corresponding to the thread id, from the `DT_Hash_Map` maintained by the ORB. `DT_Hash_Map` is a hash map that stores the DT's with their guid as the key. This may be used, for example, by one distributable thread to get the reference to another distributable thread to cancel it.

**Globally Unique Id** - The globally unique id of each DT will be generated as defined by the internet draft <http://www.opengroup.org/dce/info/draft-leach-uuids-guids-01.txt>. This GUID generator will be added to ACE.

### 3.1.2 Distributed Thread Creation

There are two ways of creating distributable threads. They are (1) `spawn()` and (2) `begin_scheduling_segment()` operations.

#### 3.1.2.1 Spawn Operation

The `spawn()` operation starts a new DT. If not explicitly specified, the scheduling parameters for the new DT are the implicit scheduling parameters of the DT calling `spawn()`. Hence, `spawn()` can only be called by another DT. `spawn()` will be implemented using `ACE_Task` and `ACE_Thread_Manager` to create and manage the new DT.

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

**ThreadAction::do()** – This operation is called by the new DT created and basically constitutes the entry point to the new scheduling segment.

The following is the IDL definition for the `spawn` operation:

```
local interface RTScheduling::Current :
RTCORBA::Current
{
  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);
};
```

`data`, `name`, `sched_param` and `implicit_sched_param` are TAO-specific. `data` is needed as an argument to the `ThreadAction::do()` that will be called by the `spawn()`. `name` is necessary to provide a name for the scheduling segment created by `spawn()`. `sched_param` and `implicit_sched_param` are necessary to provide the scheduling parameters for the new DT. If `sched_param` is null, then the `implicit_sched_param` of the scheduling segment calling `spawn()` will become the new DT's `sched_param`.

### 3.1.3 Begin, Update and End scheduling segments

RTC2 defines the operations that the application developer calls in order to begin, update and end a particular segment of code, which constitutes the DT and needs to be scheduled. The operations defined are:

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

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

    void end_scheduling_segment
        (in string name);
};
```

#### 3.1.3.1 begin\_scheduling\_segment()

This operation is called by the application developer to start a scheduling segment. If the caller is not already a DT, a new DT is created. If the caller is already a DT, a nested scheduling segment is created. It is also a scheduling point where the application interacts with the scheduler to schedule the currently executing thread.

The name parameter is the name given to the scheduling segment. Every nested scheduling segment has a unique name.

The sched\_param parameter provides the scheduling parameters for the scheduled segment. The scheduling parameter values depend on the specific scheduling discipline used, e.g., EDF, MUF, LLF.

The implicit\_sched\_param parameter provides the scheduling parameters for an implicitly created DT, e.g., to process a one-way invocation.

#### 3.1.3.2 update\_scheduling\_segment()

This operation is a scheduling point for the application to interact with the scheduler to update/change the scheduling parameters and to check if the schedule is feasible. It must be called only within a scheduling segment.

#### 3.1.3.3 end\_scheduling\_segment()

This operation marks the end of a scheduling segment. This also marks the end of the DT if the segment is not nested. Every call to begin\_scheduling\_segment() should have a corresponding end\_scheduling\_segment() and it should be on the same host and thread in which its corresponding begin\_scheduling\_segment() is called. The thread executing the code segment between a begin\_scheduling\_segment() and end\_scheduling\_segment() call is the schedulable entity that is scheduled by the scheduler.

After an end\_scheduling\_segment() operation, the DT is operating with the scheduling parameter of the next outer scheduling segment scope. If this operation is performed at the outermost scope, the result is that the processing for that thread reverts back to the fixed priority scheduling where the active thread priority is the sole determinant of the threads eligibility for execution.

### 3.1.4 Attributes

The RTScheduling::Current interface also defines the following attributes:

#### 3.1.4.1 Scheduling Parameters

```
local interface RTScheduling::Current :
    RTCORBA::Current
{
    readonly attribute
    CORBA::Policy scheduling_parameter;

    readonly attribute
    CORBA::Policy
    implicit_scheduling_parameter;
};
```

**scheduling\_parameter** – It is the scheduling parameters corresponding to the innermost scheduling segment.

**implicit\_scheduling\_parameter** – It is the scheduling parameter for any implicitly created DTs.

### 3.1.4.2 Scheduling Segment Names

```
local interface RTScheduling::Current :
    RTCORBA::Current
{
    typedef sequence<string> NameList;
    readonly attribute NameList
        current_scheduling_segment_names;
};
```

**current\_scheduling\_segment\_names** - This is a sequence of names of all the nested scheduling segments ordered from innermost segment name to the outermost segment name. The name is stored in each current corresponding to the scheduling segments in a DT.

## 3.2 RTScheduling::ResourceManager Interface

A ResourceManager interface is provided in order to create scheduler aware resources. These resources can have scheduling parameters associated with them. The scheduling parameters that can be associated with the resource are scheduler specific. The scheduler will be called whenever the resource is acquired or released.

```
local interface
RTScheduling::ResourceManager :
    RTCORBA::Mutex
{
};
```

## 3.3 RTScheduling::DistributableThread Interface

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

**cancel()** - This operation causes the `CORBA::THREAD_CANCELLED` exception to be raised in the context of the DT at the next scheduling point for the DT. This exception is propagated to where the DT started as illustrated in Figure 5.

A DT can be cancelled anytime during its execution on any host that it currently spans. As shown in Figure 5, the DT was cancelled on Host 2, even though it is currently executing on Host 3.

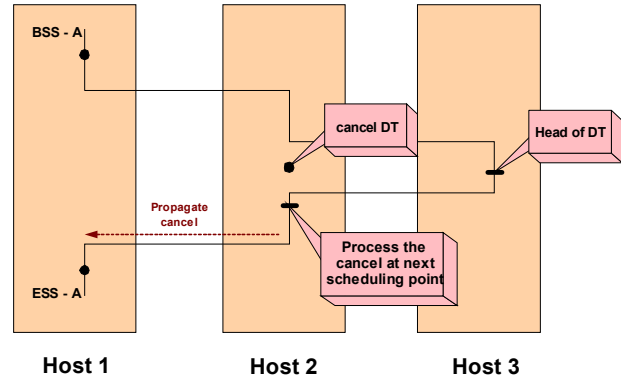


Figure 5: Distributable Thread Cancellation

The thread cancelled exception is propagated to the start of the DT. As shown in Figure 5, the `CORBA::THREAD_CANCELLED` exception is propagated from Host 2 to Host 1 where the DT started. However, the cancellation is not forwarded to the head of the DT if it is not on the same host. Thus, the cancellation will only be processed once the DT returns to Host 2 from Host 3.

Note that while the DT is a local interface, the head of the distributable thread may not be executing within the same address space as thread calling `cancel()`. Hence, `cancel()` is implemented by setting a flag in the `DistributableThread` interface to mark it as cancelled. The DT reference, on which to call `cancel()`, can be obtained from the `DT_Hash_Map` in the ORB with the corresponding guid.

At the next local scheduling point of the DT a check for cancellation of the DT will be performed. If the flag is set, then the DT is cancelled and the `CORBA::THREAD_CANCELLED` exception is raised and the relevant resources are released. This requires the addition of the `cancel()` operation on the `RTScheduling::Scheduler` interface as explained in Section 3.5.1.

Once `CORBA::THREAD_CANCELLED` is raised, the DT is cancelled and now runs as a normal (non-DT) thread. If the application catches the `CORBA::THREAD_CANCELLED` exception, it should re-throw it so that the exception can be propagated to the start of the DT.

## 3.4 RTScheduling::Scheduler Interface

The scheduler interface has the semantics of an abstract class from which specific scheduling discipline implementations will derive. The scheduler will be installed in the ORB and can be queried with `resolve_initial_references()` using “RTScheduler”. The following are the operations and attributes defined in the scheduler interface by RTC2.

### 3.4.1 Attributes

#### 3.4.1.1 Scheduling Policies

This attribute allows the ORB to request the list of POA policies that the scheduler requires to be applied to all the POA’s associated with this ORB.

```
local interface RTScheduling::Scheduler
{
    attribute CORBA::PolicyList
        scheduling_policies;
};
```

#### 3.4.1.2 Scheduling Discipline Name

This attribute contains the name of the scheduling discipline that the corresponding scheduler is implementing. This may be needed by the ORB or application.

```
local interface RTScheduling::Scheduler
{
    readonly attribute string
        scheduling_discipline_name;
};
```

### 3.4.2 Operations

#### 3.4.2.1 Create Resource Manager

This operation is used by the application to create a scheduler aware resource protection primitive, and associate a name with it.

```
local interface RTScheduling::Scheduler
{
    ResourceManager create
```

```
(in string name,
 in CORBA::Policy sched_param);
};
```

There could be a requirement for a scheduler to create UUID aware resources. For example, there might be a requirement to allow a DT to reacquire a recursive lock in a recursive invocation, that maybe serviced by a different OS thread. The lock implementation should allow this to proceed since it is logically the same DT even though it is serviced by different OS threads at different points. It is up to the scheduler to provide this capability if required.

#### 3.4.2.2 Setting Scheduling Parameter

This operation associates the supplied scheduling parameter and name with the supplied servant resource. The scheduling parameter values are specific to the scheduling discipline implemented by the scheduler. This is useful for schedulers that associate some scheduling information with a shared resource.

```
local interface RTScheduling::Scheduler
{
    void set_scheduling_parameter
        (inout PortableServer::Servant
 resource,
         in string name,
         in CORBA::Policy
 scheduling_parameter);
};
```

## 3.5 TAO Specific Addendums

### 3.5.1 Extensions to RTScheduling::Scheduler

RTC2 does not define operations on the `RTScheduling::Scheduler` interface that the ORB or application can use to interact with the scheduler to begin, update and end scheduling segments, as defined in the `RTScheduling::Current` interface. However, the following TAO specific operations are added to `RTScheduling::Scheduler` interface.

```
include ``PortableInterceptor.pidl``
local interface RTScheduling::Scheduler
{
    void begin_new_scheduling_segment
        (in IdType guid,
         in string name,
```

```

    in CORBA::Policy sched_param,
    in CORBA::Policy implicit_sched_param)
raises
    (UNSUPPORTED_SCHEDULING_DISCIPLINE);

void begin_nested_scheduling_segment
    (in IdType guid,
     in string name,
     in CORBA::Policy sched_param,
     in CORBA::Policy implicit_sched_param)
raises
    (UNSUPPORTED_SCHEDULING_DISCIPLINE);

void update_scheduling_segment
    (in IdType guid,
     in string name,
     in CORBA::Policy sched_param,
     in CORBA::Policy implicit_sched_param)
raises
    (UNSUPPORTED_SCHEDULING_DISCIPLINE);

void end_scheduling_segment
    (in IdType guid,
     in string name);

void end_nested_scheduling_segment
    (in IdType guid,
     in string name,
     in CORBA::Policy outer_sched_param);

void send_request
    (in ClientRequestInfo ri)
raises (ForwardRequest);

void receive_request
    (in ServerRequestInfo ri,
     out IdType guid,
     out string name,
     out CORBA::Policy sched_param,
     out CORBA::Policy implicit_sched_param)
raises (ForwardRequest);

void send_reply
    (in ServerRequestInfo ri);

void send_exception
    (in ServerRequestInfo ri)

```

```

    raises (ForwardRequest);

void send_other
    (in ServerRequestInfo ri)
raises (ForwardRequest);

void receive_reply
    (in ClientRequestInfo ri);

void receive_exception
    (in ClientRequestInfo ri)
raises (ForwardRequest);

void receive_other
    (in ClientRequestInfo ri)
raises (ForwardRequest);

void cancel (in IdType guid);
};

```

**begin\_new\_scheduling\_segment()** – This operation is called when `Current::begin_scheduling_segment()` is invoked by a non-DT thread or when a new DT is created through `Current::spawn()`. The `guid` passed corresponds to the new `guid` generated for the new DT. All the other parameters are passed through from the call on the current interface. The scheduler should schedule the corresponding thread with the scheduling parameters specified.

**begin\_nested\_scheduling\_segment()** – This operation is called when `Current::begin_scheduling_segment()` is invoked by a DT to create a nested scheduling segment.

**update\_scheduling\_segment()** – This operation is called when `Current::update_scheduling_segment()` is invoked by a DT thread. The scheduler should update the scheduling parameters of the corresponding scheduling segment.

**end\_scheduling\_segment()** – This operation is called when `Current::end_scheduling_segment()` is invoked by a DT to end the outer most scheduling segment. The scheduler should end the scheduling segment and reschedule the corresponding thread to the scheduling parameters when the thread was not running as a DT.

**end\_nested\_scheduling\_segment()** – This operation is called when `Current::end_scheduling_segment()` is invoked by a DT to end a nested scheduling segment. The scheduler should end the scheduling segment and use the

outer\_sched\_param parameter to reset the DT to the scheduling parameters of the enclosing scope.

**send\_request()** – This operation is similar to the `send_request()` interceptor call and is called during the `send_request()` interception point. This method was added to facilitate the implementation of one-way requests. The one-way request requires the creation of a new DT with a new guid. This guid should be created before the request is sent out. The ORB interceptor creates this new guid at the `send_request()` interception point and calls the `Scheduler::send_request()`. It is the responsibility of the scheduler to populate the service context in the `Scheduler::send_request()` operation with the scheduling parameters of the new DT.

**receive\_request()** – This operation is similar to the `receive_request()` interceptor call and is called during the `receive_request()` interception point. It is the responsibility of the scheduler to unmarshal the scheduling information in the service context that is received. The scheduler uses this information to schedule the thread servicing the request. The ORB requires this information (that it gets from the out parameters of this request) to reconstruct a `RTScheduling::Current` on the server.

**send\_reply(), send\_exception(), send\_other()** – These operations are similar to their respective interceptor calls and are called during the corresponding interception points by the Scheduling Interceptors, Section 3.5.3. The scheduler can implement these methods if required.

**receive\_reply(), receive\_exception(), receive\_other()** – These operations are similar to their respective calls and are called during the corresponding interception points by the Scheduling Interceptors, Section 3.5.3. The scheduler can implement these methods if required.

**cancel()** – This operation is called once a DT is cancelled. The scheduler will need to be updated to reflect the cancellation of the DT.

### 3.5.2 RTScheduler Manager

This interface is required to manage the schedulers in TAO.

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

The `RTScheduler_Manager` instance can be obtained from the ORB using `resolve_initial_reference()` with “`RTScheduler_Manager`”. It can be used by the application to install an application specific scheduler.

### 3.5.3 Scheduling Interceptors

The ORB registers the following interceptors. These are required for (a) intercepting one-way invocations, where a new guid needs to be created, (b) performing cleanup operations for the DT, on the server, when the replies are sent from the server, and (c) perform cleanup operations on the client when the replies are received if necessary.

```
interface RTScheduling::ClientInterceptor :
    ClientRequestInterceptor
{
    void send_request
        (in ClientRequestInfo ri)
        raises (ForwardRequest)

    void receive_reply
        (in ClientRequestInfo ri);

    void receive_exception
        (in ClientRequestInfo ri)
        raises (ForwardRequest);

    void receive_other
        (in ClientRequestInfo ri)
        raises (ForwardRequest);
};
```

```
interface RTScheduling::ServerInterceptor :
    ServerRequestInterceptor
{
    void receive_request
        (in ServerRequestInfo ri)
        raises (ForwardRequest);

    void send_reply
        (in ServerRequestInfo ri);

    void send_exception
        (in ServerRequestInfo ri)
        raises (ForwardRequest);
};
```

```

void send_other
  (in ServerRequestInfo ri)
  raises (ForwardRequest);
};

```

### 3.6 Passing Scheduling Parameters with Invocations

Scheduling parameters of a DT are propagated as it spans nodes so the corresponding schedulers on the other hosts can schedule it. The scheduling information of the DT making a remote request is passed to the remote host using the GIOP service contexts of the request. The scheduler populates and reads information from the service contexts at interception points for sending and receiving requests. As explained in Section 3.5.1, the scheduler is required to implement the Scheduler::send\_request() and Scheduler::receive\_request() operations. These methods are called by the corresponding methods in the scheduling interceptors that are registered with the ORB (Section 3.5.3).

**Two-Way Invocations** - In a two-way invocation the Client Scheduling Interceptor is called when sending a request. It in turn calls the Scheduler::send\_request() that populates the service context with the scheduling information that is sent with the request. The Server Scheduling Interceptor is called when the request reaches the server. It in turn calls the Scheduler::receive\_request(), which reads the scheduling information from the service context received. The ORB in turn uses this information to create a RTScheduling::Current for the thread servicing the upcall.

In collocated two-ways, the thread making the request also services the request. Hence it does not have to be re-scheduled for servicing the request.

**One-Way Invocations** - A one-way invocation requires the creation of a new DT. The Client Scheduling Interceptor creates a new guid and a new RTScheduling::Current and populates it with the implicit scheduling parameters of the DT invoking the one-way. It then calls the Scheduler::send\_request() to populate the service context. Once the service context is populated, the RTScheduling::Current is restored to the scheduling parameters of the DT making the one-way invocation. The server side behaves similar to that explained in the two-way invocation.

Collocated one-ways will not result in the creation of new DTs in TAO for the following reasons:

- Lack of interceptor support for collocated one-ways
- Lack of ORB support to execute collocated calls in a separate thread
- Overhead of DT creation for collocated one-ways

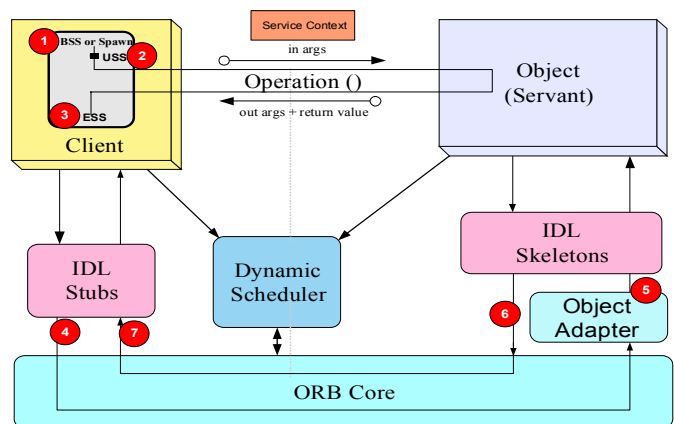
- Scheduling overhead and complexity

### 3.7 Scheduling Points

RTC2 defines scheduling points where the application and ORB interact with the scheduler. These are shown in Figure 5. Scheduling points 1, 2, 3 are where the application interacts with the Scheduler, whereas 4, 5, 6, and 7 are where the ORB interacts with the Scheduler.

Besides these scheduling points, RTC2 also defines the following scheduling points:

- Creation of a resource manager
- Blocking on a request for a resource via a call to RTScheduling::ResourceManager::lock() or RTScheduling::ResourceManager::try\_lock()
- Unblocking as a result of a release of a resource via a call to RTScheduling::ResourceManager::unlock()



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 6: Scheduling Points

### 4. Dynamic Scheduler Framework Testing

To test the correctness of TAO's Dynamic Scheduling Framework, simple schedulers will be plugged in that implement the RTScheduling::Scheduler interface. The schedulers are described in the following subsection.

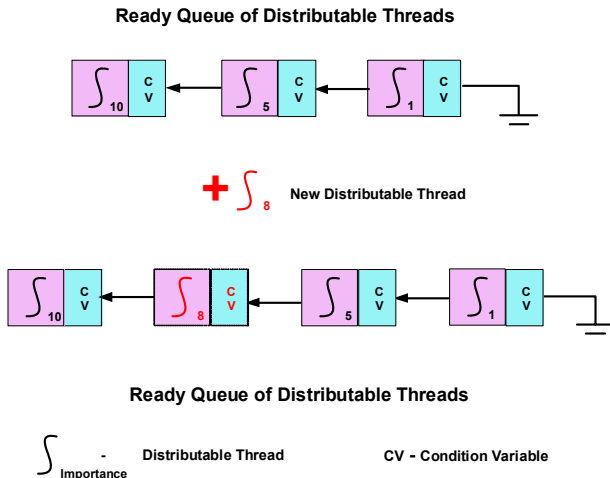
## 4.1 Fixed Priority Scheduler

The fixed priority scheduler schedules the DTs by mapping its importance to native priorities. The onus of dispatching is delegated to the OS scheduler that schedules the DTs according to their native priorities.

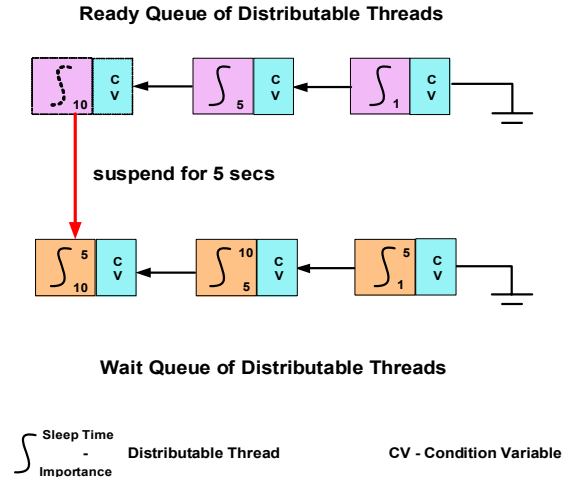
## 4.2 Most Important First (MIF) Scheduler

The MIF scheduler schedules the DTs by their importance, with the most important thread getting to execute. The importance of the thread is specified as a scheduling parameter by the application to the scheduler. The scheduler maintains two queues: ready and wait queues. The ready queue stores DTs in the order of their importance, to be scheduled with the head of the queue having the most important thread ready for execution. Each DT is waiting on a condition variable. When the DT reaches the head of the queue, which implies that it is the next DT to be executed, the scheduler signals the corresponding condition variable that the thread is waiting on and hence awakens it. The wait queue has the threads that are suspended for a specific time by waiting on a condition variable. They are awakened by the scheduler after their specific sleep time by signaling the condition variable they are waiting on.

This scheduler will also allow the application to update/change the importance of an already scheduled DT by re-sorting the ready queue based on the importance of the new DT that has been added or re-positioning a DT that is already in the queue.



**Figure 7: MIF Scheduler - Insertion of new DT in ready queue**



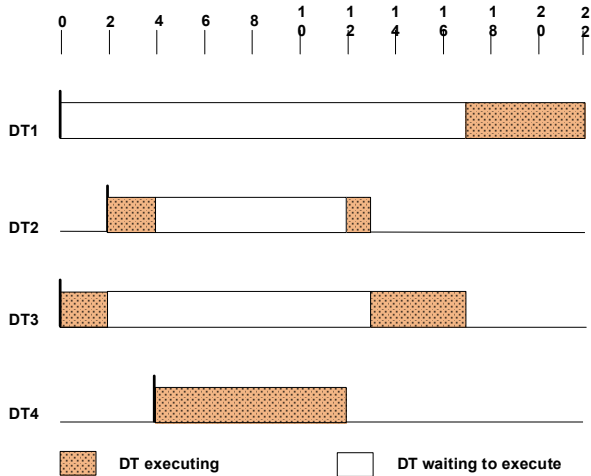
**Figure 8: MIF Scheduler – Moving a suspended DT from ready to wait queue**

## 4.3 Test

Plugging in the schedulers described in Sections 4.1 and 4.2 will test TAO's Dynamic Scheduling Framework. A set of distributable threads will be started and scheduled as per the table in Figure 9(a). The importance of the DT is the scheduling parameter used by the scheduler. The expected schedule of the specified DTs will be pre-determined as shown in Figure 9(b). The actual schedule of the schedulers for the specified set of DTs and their scheduling parameter will be compared with the pre-determined schedule to prove the correctness.

Distributable Thread	Start Time	Execution Time	Importance
DT1	0	5	2
DT2	2	3	5
DT3	0	6	4
DT4	4	8	1

(a)



(b)

Figure 9: Testing TAO’s Dynamic Scheduler Framework (a) Table of DTs with their scheduling parameters (b) The pre-determined schedule of execution of the DTs as expected from the simple schedulers

## 5. Kokyu Scheduler Framework

Once the preliminary testing of TAO’s Dynamic Scheduling Framework is done with the simple schedulers described in Section 4, the Kokyu Scheduler Framework [3] will replace the simple scheduler. The Kokyu Scheduler will be adapted to the dynamic scheduling framework by implementing the scheduler interface specified by RTC2 and the interfaces defined in Section 3.5.

## 6. Shortcomings of the RTC2 Specification

This section describes some shortcomings of RTC2 and how they were resolved.

### 6.1 Redundant Operations

`RTScheduling::Current::get_current_id()` (Section 3.1.1) is redundant as the `readonly` attribute `id` provides the accessor methods for the DT’s `id`.

### 6.2 Insufficient Operation Parameters

The parameters passed to `RTScheduling::Current::spawn()` are insufficient to realize the semantics of the operation. Additional parameters have been added as described in Section 3.1.2.1.

### 6.3 Insufficient Operations

RTC2 specifies the interface `RTScheduling::Scheduler` for the scheduler. The application and the ORB should use the operations defined on this interface to interact with the scheduler. However, the operations defined on the interface were insufficient for a meaningful interaction with the scheduler. Hence, operations were added to this interface as described in Section 3.5.

## 7. Conclusion

TAO’s Dynamic Scheduler Framework will implement the RTC2 specification as detailed in this paper. It will be tested with simple schedulers for correctness. The Kokyu Scheduler framework will later be adapted to the dynamic scheduling framework and plugged-in as the scheduler and tested.

## 8. Timeline

**Nov 30, 2002** – Finish implementation of the all the interfaces.

**Dec 15, 2002** – Implement the simple scheduler

**Dec 31, 2002** – Test the Dynamic Scheduling Framework with the simple scheduler

**Jan 15, 2003** – Adapt the Kokyu Scheduler to the Dynamic Scheduling Framework

**Jan 31, 2003** – Test the Kokyu Scheduler with the Dynamic Scheduling Framework.

## REFERENCES

- [1] Real-Time CORBA 2.0: Dynamic Scheduling Specification, OMG Final Adopted Specification, *September 2001*, <http://cgi.omg.org/docs/ptc/01-08-34.pdf>.
- [2] Real-Time CORBA Specification, *August 2002*, <http://cgi.omg.org/docs/formal/02-08-02.pdf>
- [3] Flexible Scheduling in Middleware for Distributed Rate-Based Real-Time Applications, Chris D. Gill, *December*

## Appendix: Implementing RTScheduling::Current

The application uses the operations defined on the RTScheduling::Current to interact with the Scheduler. This section provides pseudo code illustrating how the Current is implemented. The code can also be found in the following document:

<http://www.oomworks.com/webdocs/Dynamic-Scheduling.cpp>

```
Current_i::Current_i
(in IdType guid,
 in string name,
 in CORBA::Policy sched_param,
 in CORBA::Policy implicit_sched_param,
 in Current_i prev_current,
 in DistributableThread dt)
{
 // Store all of the above as members.
}

void
Current_i::begin_scheduling_segment
(in string name,
 in CORBA::Policy sched_param,
 in CORBA::Policy implicit_sched_param)
raises (UNSUPPORTED_SCHEDULING_DISCIPLINE)
{
 // GUID.
IdType guid = 0;

 // DT pointer.
DistributableThread *dt = 0;

 // Check if caller is not already a DT.
if (dt_ == 0) {

 // Generate GUID.
guid = ACE::new_guid ();

 // Inform scheduler of new DT.
orb->scheduler_->begin_new_scheduling_segment
(guid,
 name,
 sched_param,
 implicit_sched_param);

 // Create new DT.
dt = new DistributableThread ();
```

```
 // Add new DT to map.
int result =
 orb->dt_map_->add (guid, dt);

 // On failure, cancel DT.
if (result != 0)
 cancel_thread ();

} else { // A nested segment.

 // Check if DT has been cancelled.
if (dt_->cancelled ())
 cancel_thread ();

 // GUID
guid = guid_;

 // DT.
dt = dt_;

 // Inform scheduler of start of nested
 // scheduling segment.
orb->scheduler_->begin_nested_scheduling_segment
(guid_,
 name,
 sched_param,
 implicit_sched_param);
}

 // Create new current.
Current_i* new_current
 = new Current_i (guid,
                 name,
                 sched_param,
                 implicit_sched_param,
                 this,
                 dt);

 // Install new current in the ORB.
orb->rt_current_ =
 new_current;
}

void
Current_i::update_scheduling_segment
(in string name,
 in CORBA::Policy sched_param,
 in CORBA::Policy implicit_sched_param)
raises (UNSUPPORTED_SCHEDULING_DISCIPLINE)
{
 // Check if DT has been cancelled.
```

```

if (dt_>cancelled ())
    cancel_thread ();

// Let scheduler know of the updates.
orb->scheduler_->update_scheduling_segment
    (guid_,
     name,
     sched_param,
     implicit_sched_param);

// Remember the new values.
sched_param_ = sched_param;
implicit_sched_param_ =
    implicit_sched_param;
name_ = name;
}

void
Current_i::end_scheduling_segment
    (in string name)
{
    // Check if DT has been cancelled.
    if (dt_>cancelled ())
        cancel_thread ();

    // If this was the outermost scope.
    if (prev_current_->dt_ == 0)
    {
        // Let the scheduler know that the DT is
        // terminating.
        orb->scheduler_->end_scheduling_segment
            (name);

        // Cleanup DT.
        cleanup_DT ();
    }
    else { // A Nested segment.

        // Inform scheduler of end of nested
        // scheduling segment.
        orb->scheduler_->end_nested_scheduling_segment
            (guid_,
             name,
             prev_current_->sched_param_);

        // Cleanup current.
        cleanup_current ();
    }
}

DistributableThread
Current_i::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)
{
    // Check if DT has been cancelled.
    if (dt_>cancelled ())
        cancel_thread ();

    // Generate GUID.
    guid = ACE::new_guid ();

    // Create new DT.
    DistributableThread *dt =
        new DistributableThread ();

    // Add new DT to map.
    int result =
        orb->dt_map_->add (guid, dt);

    // Create new task for new DT.
    DTTask *dttask
        = new DTTask (thread_manager_,
                     this,
                     start,
                     data,
                     guid,
                     name,
                     sched_param,
                     implicit_sched_param,
                     dt);

    // Activate thread.
    dttask->activate (... ,
                    base_priority,
                    ... ,
                    stack_size);

    return dt;
}

// This class provides an entry point for the
// new DT.
class DTTask : public ACE_Task_Base
{
    DTTask
    (in ACE_Thread_Manager manager,
     in RTScheduling::Current current,
     in ThreadAction start,

```

```

in CORBA::VoidData data,
in IdType guid,
in string name,
in CORBA::Policy sched_param,
in CORBA::Policy implicit_sched_param,
in DistributableThread dt)
{
    // Store all of the above as members.
}

int svc (void)
{
    // Create new current.
    Current_i* new_current
        = new Current_i (guid_,
                        name_,
                        sched_param_,
                        implicit_sched_param_,
                        0,
                        dt_);

    // Install new current in the ORB.
    orb->rt_current_ =
        new_current;

    // Inform scheduler of new DT.
    orb->scheduler_->begin_new_scheduling_segment
        (guid_,
         name_,
         sched_param_,
         implicit_sched_param_);

    // Invoke entry point into new DT.
    start->do (data_);

    // Let the scheduler know that the DT is
    // terminating.
    orb->scheduler_->end_scheduling_segment (name_);

    // Cleanup DT.
    new_current->cleanup_DT ();

    // Delete this support class.
    delete this;
}

void
Current_i::send_request (in ClientRequestInfo ri)
    raises (ForwardRequest)
{
    // Temporary current.
    Current_i *new_current = 0;

    // If this is a one way request
    if (ri.oneway) {

        // Generate GUID.
        IdType guid = ACE::new_guid ();

        // Create new DT.
        DistributableThread *dt =
            new DistributableThread ();

        // Add new DT to map.
        int result =
            orb->dt_map_->add (guid, dt);

        // Create new temporary current. Note that
        // the new <sched_param> is the current
        // <implicit_sched_param> and there is no
        // segment name.
        new_current
            = new Current_i (guid,
                            0,
                            implicit_sched_param_,
                            0,
                            this,
                            dt);

        // Install new current in the ORB.
        orb->rt_current_ =
            new_current;
    }

    // Scheduler populates the service context with
    // scheduling parameters.
    orb->scheduler_->send_request (ri);

    // If this is a one way request
    if (ri.oneway) {

        // Cleanup temporary DT.
        new_current->cleanup_DT ();
    }
}

void
Current_i::receive_request
    (in ServerRequestInfo ri)
    raises (ForwardRequest)
{
    IdType guid;
    String name;
    CORBA::Policy sched_param;

```

```

CORBA::Policy implicit_sched_param;

// Scheduler retrieves scheduling parameters
// from request and populates the out
// parameters.
orb->scheduler_->receive_request
    (ri,
     guid,
     name,
     sched_param,
     implicit_sched_param);

// Create new DT.
DistributableThread *dt =
    new DistributableThread ();

// Add new DT to map.
int result =
    orb->dt_map_->add (guid, dt);

// Create new current.
Current_i* new_current
    = new Current_i (guid,
                    name,
                    sched_param,
                    implicit_sched_param,
                    0,
                    dt);

// Install new current in the ORB.
orb->rt_current_ =
    new_current;
}

void
Current_i::send_reply (in ServerRequestInfo ri)
{
    // Inform scheduler that upcall is complete.
    orb->scheduler_->send_reply (ri);

    // Cleanup DT.
    cleanup_DT ();
}

void
Current_i::send_exception
    (in ServerRequestInfo ri)
    raises (ForwardRequest)
{
    // Inform scheduler that upcall is raising
    // exception.
    orb->scheduler_->send_exception (ri);

    // Cleanup DT.
    cleanup_DT ();
}

void
Current_i::send_other (in ServerRequestInfo ri)
    raises (ForwardRequest)
{
    // Inform scheduler that upcall is sending
    // other.
    orb->scheduler_->send_other (ri);

    // Cleanup DT.
    cleanup_DT ();
}

void
Current_i::receive_reply (in ClientRequestInfo ri)
{
    // Inform scheduler that reply was recieved.
    orb->scheduler_->receive_reply (ri);
}

void
Current_i::receive_exception
    (in ClientRequestInfo ri)
    raises (ForwardRequest)
{
    // If the remote host threw a THREAD_CANCELLED
    // exception, make sure to take the appropriate
    // local action.
    if (received_exception ==
        CORBA::THREAD_CANCELLED)
    {
        // Perform the necessary cleanup as the
        // thread was cancelled.
        cancel_thread ();
    }
    else
    {
        // Inform scheduler that exception was
        // received.
        orb->scheduler_->receive_exception (ri);
    }
}

void
Current_i::receive_other (in ClientRequestInfo ri)
    raises (ForwardRequest)
{
    // Inform scheduler that client is receiving

```

```

    // other.
    orb->scheduler_->receive_other (ri);
}

void
Current_i::cleanup_DT (void)
{
    // Remove DT from map.
    orb->dt_map_->remove (guid_);

    // Delete DT.
    delete dt_;

    // Cleanup current.
    cleanup_current ();
}

void
Current_i::cleanup_current (void)
{
    // Update to previous current in the ORB.
    orb->rt_current_ =
        prev_current_;

    // Delete this current.
    delete this;
}

void
Current_i::cancel_thread ()
{
    // Let the scheduler know that the thread has
    // been cancelled.
    orb->scheduler_->cancel (guid_);

    // Remove DT from map.
    orb->dt_map_->remove (guid_);

    // Delete DT.
    delete dt_;

    // Remove all related nested currents.
    delete_all_currents ();

    // Throw exception.
    throw CORBA::THREAD_CANCELLED;
}

```