

Dynamic Scheduling Real-Time CORBA 2.0 Joint Final Submission

OMG Document **orbos/2001-04-01**

This document is final submission in response to the RFP
contained in OMG document number orbos/99-03-32
entitled "Dynamic Scheduling".

Eternal Systems, Inc.

Lockheed-Martin Federal Systems, Inc.

Objective Interface Systems, Inc.

Tri-Pacific Software, Inc.

Vertel Corporation

Supporting organizations:

The MITRE Corporation

1.1 SPAWAR Systems Center SD

Copyright Waiver

See inside cover for copyright statement.

1.2 Submission Contact Points

The contact points for the co-submitting companies are:

Vana Kalogeraki
Eternal Systems, Inc.
P.O. Box 13963
Santa Barbara, CA 93107, USA
Phone: +1 805 893 7788
Fax: +1 805 893 3262
E-mail: vana@alpha.ece.ucsb.edu

Tom Barker
Lockheed-Martin Federal Systems, Inc.
Owego, NY, USA
Phone: +1 607 751 3794
E-mail: thomas.barker@lmco.com

Bill Beckwith
Objective Interface Systems, Inc.
1892 Preston White Dr., Suite 300
Reston, VA 20191-5448, USA
Phone: +1 703 295 6500
Fax: +1 703 295 6501
E-mail: bill.beckwith@ois.com

Peter Kortmann
Tri-Pacific Software, Inc.
1070 Marina Village Parkway
Suite 202
Alameda, CA 94501, USA
Phone: +1 510 814 1775
Fax: +1 510 814 1788
E-mail: peter@tripac.com

Shahzad Aslam-Mir
Vertel Corporation
5825 Oberlin Dr. Ste# 300
San Diego, CA 92121, USA
Phone: +1 858 824 4128
Fax: +1 858 824 4110
E-mail: sam@expersoft.com

The contact points for the supporting companies are:

E. Douglas Jensen
The MITRE Corp.
202 Burlington Road
Bedford, MA 01730-1420, USA
Phone: +1 781 271 2514
Fax: +1 781 271 4686
E-mail: jensen@mitre.org

Dock Allen
The MITRE Corp.
202 Burlington Road
Bedford, MA 01730-1420, USA
Phone: +1 781 271 8216
Fax: +1 781 271 4686
E-mail: dock@mitre.org

Russell E. Johnston
SPAWAR Systems Center
53140 Systems Street
San Diego, CA 92152, USA
Phone: +1 619 553 4096
Fax: +1 619 553 6405
E-mail: russ@spawar.navy.mil

1.3 Guide to the Material in this Submission

This section of the proposal contains the Part I items as suggested by the RFP, including a statement of conformance to the mandatory and optional requirements of the RFP.

The following section contains an overview and rationale of the additions and changes proposed by this submission.

The next section contains the conformance points proposed by this submission, and the other items suggested for the contents of Part III.

1.4 Statement of Proof of Concept

The IDL in this submission was successfully compiled by Objective Interface's *ORBexpress* RT IDL compiler.

At the time of this submission, the submitting and supporting organizations for this submission have drawn upon the following experiences to validate the concepts in this submission:

- Objective Interface had an internal prototype implementation of a real-time ORB with replaceable scheduling.
- Tri-Pacific has commercial products that implement the Scheduling Service of the Real-Time CORBA 1.0 specification.
- Several engineers from MITRE have experience implementing an operating system kernel with replaceable dynamic scheduling.

1.5 Resolution of RFP Mandatory and Optional Requirements

1.5.1 General Requirements on Proposals

“5.1 Mandatory Requirements”

“5.1.1 Proposals shall express interfaces in OMG IDL. Proposals should follow accepted OMG IDL and CORBA programming style. The correctness of the IDL shall be verified using at least one IDL compiler (and preferably more than one). In addition to IDL quoted in the text of the submission, all the IDL associated with the proposal shall be supplied to OMG in compiler-readable form.”

All interfaces introduced in this submission are expressed in IDL and have compiled by a conforming IDL compiler.

“5.1.2 Proposals shall specify *operation behavior, sequencing, and side-effects* (if any).”

The operation behavior, sequencing, and side effects are described in the final section.

“5.1.3 Proposals shall be *precise and functionally complete*. There should be no implied or hidden interfaces, operations, or functions required to enable an implementation of the proposed specification.”

This submission is believed to be precise and functionally complete.

“5.1.4 Proposals shall clearly distinguish *mandatory* interfaces and other specification elements that all implementations must support from those that may be *optionally* supported.”

Mandatory and optional compliance points are stated in the section 7.

“5.1.5 Proposals shall *reuse* existing OMG specifications including CORBA, CORBA services, and CORBA facilities in preference to defining new interfaces to perform similar functions.”

This submission revises the Real-Time CORBA 1.0 specification.

“5.1.6 Proposals shall justify and fully specify any *changes or extensions* required to existing OMG specifications. This includes changes and extensions to CORBA inter-ORB protocols necessary to support interoperability. In general, OMG favors *upwards compatible* proposals that minimize changes and extensions to existing OMG specifications.”

Since Dynamic Scheduling Real-Time CORBA represents the anticipated second phase of Real-Time CORBA 1.0 this submission modifies the Real-Time CORBA 1.0 specification in order to create a single, consistent, coherent specification for timeliness in CORBA

systems. This modification includes deprecating the Scheduling Service in the Real-time CORBA 1.0 specification. The Real-time CORBA 1.0 Scheduling Service provides for a scheduler that implements a form of fixed priority scheduling. Since this form of scheduler can be implemented with this specification the previous Scheduling Service is no longer needed.

“5.1.7 Proposals shall factor out functions that could be used in different contexts and specify their interfaces separately. Such *minimality* fosters re-use and avoids functional duplication.”

The submitters believe that the interfaces are optimally factored.

“5.1.8 Proposals shall use or depend on other interface specifications only where it is actually necessary. While re-use of existing interfaces to avoid duplication will be encouraged, proposals should avoid gratuitous use.”

It is believed that there are no unnecessary dependencies on other specifications.

“5.1.9 Proposals shall specify interfaces that are compatible and can be used with existing OMG specifications. Separate functions doing separate jobs should be capable of being used together where it makes sense for them to do so.”

The interfaces in this submission are believed to be compatible with other OMG specifications.

“5.1.10 Proposals shall preserve maximum *implementation flexibility*. Implementation descriptions should not be included, however proposals may specify constraints on object behavior that implementations need to take into account over and above those defined by the interface semantics.”

Keeping this submission free of any unnecessary implementation description has preserved implementation flexibility. The examples provided herein are not intended to constrain implementation.

“5.1.11 Proposals shall allow *independent implementations* that are *substitutable* and *interoperable*. An implementation should be replaceable by an alternative implementation without requiring changes to any client.”

The only interfaces required for interoperability are specified in IDL. Note however that scheduler implementations will typically have dependencies on the operating systems that may impair the ability of independent substitution. These dependencies are beyond the scope of the submission.

“5.1.12 Proposals shall be compatible with the architecture for system distribution defined in ISO/IEC 10746, Reference Model of Open Distributed Processing (ODP). Where such compatibility is not achieved, the response to the RFP must include reasons why

compatibility is not appropriate and an outline of any plans to achieve such compatibility in the future.”

This submission does not change the overall CORBA services architecture.

“5.1.13 In order to demonstrate that the service or facility proposed in response to this RFP, can be made secure in environments requiring security, answers to the following questions shall be provided:

What, if any, are the security sensitive objects that are introduced by the proposal?”

No security sensitive objects are introduced by this submission.

“Which accesses to security-sensitive objects must be subject to security policy control?”

Not applicable.

“Does the proposed service or facility need to be security aware?”

While this submission does not require that a scheduler be security aware it is anticipated that in a secure environment that scheduler implementations may need to be security aware.

“What CORBA security level and options are required to protect an implementation of the proposal? In answer to this question, a reasonably complete description of how the facilities provided by the level and options (e.g. authentication, audit, authorization, message protection etc.) are used to protect access to the sensitive objects introduced by the proposal shall be provided.”

Not applicable (since there are no security sensitive objects).

“What default policies should be applied to the security sensitive objects introduced by the proposal?”

Not applicable (since there are no security sensitive objects).

“Of what security considerations must the implementers of your proposal be aware?”

There are no special security considerations required for implementers of this submission. Note, however, that the reverse of this statement is not true. An implementer of a security service that is intended for use with a scheduler must be aware of the security service’s impact on the timeliness properties of the application and the scheduler.

1.5.2 Specific RFP Requirements

1.5.2.1 RFP Statements of Scope

The following technologies were listed in the Dynamic Scheduling RFP as “in scope”.

“synchronization (e.g. locks and mutexes) and the allocation of synchronization resources”

“flexible communication”

“predictable management of memory and buffers”

“setting and determining the characteristics of a schedulable entity, such as the policy by which it is to be scheduled, and any parameters to the scheduling policy, such as deadline or period”

“scheduling policy and parameter inheritance by less critical schedulable entities allowing them to temporarily “inherit” a more critical entity’s scheduling policy and parameters so that the less critical schedulable entity can more quickly release a resource that is blocking the more critical entity”

“scheduling policy and parameter based resolution of contention for services and resources”

1.5.2.2 Mandatory Requirements

“1. Proposals shall be extensions to adopted OMG specifications; they shall not re-specify existing functionality provided by adopted OMG specifications.”

This specification does not re-specify existing capabilities provided by previously adopted OMG specifications other than the Scheduling Service in Real-time CORBA 1.0. This specification extends the previously adopted OMG specification for the RTCORBA 1.0 Scheduling Service. This specification builds on the fixed priority scheduler in the 1.0 specification by adding additional capability and scheduler types.

“2. Proposals shall extend the definition of a ‘schedulable entity’ determined by Real-time CORBA 1.0 to provide an understanding of dynamic scheduling and end-to-end predictability, and shall define one or more models for the execution and scheduling of these entities.”

This submission replaces the definition of the “schedulable entity” in the Real-time CORBA 1.0 specification. See section 2.1.2 for more details.

“3. Proposals shall specify interfaces for assigning, discovering, and altering the dynamic scheduling parameters including but not

limited to deadlines, importance, delay, and period, or parameters that express the same information, of any 'schedulable entity'."

This submission provides generic interfaces for the definition, assignment, alteration, and retrieval of any type of scheduling parameter.

"4. Proposals shall specify a small set of scheduling policies, and describe how the behavior of the policies with respect to management of all system resources is influenced by the dynamic scheduling parameters defined by the submission. The defined policies shall include the capability to provide fixed priority support in addition to dynamic scheduling support. There shall also be a capability for applications to extend scheduling policy definitions, and to create new policies."

To avoid confusion with CORBA policies, this submission uses the term scheduling discipline instead of scheduling policy.

This submission provides for a broad set of scheduling disciplines without constraining the support of additional scheduling disciplines.

"5. Proposals shall specify a mechanism and its interfaces for propagating the equivalent (as interpreted at the servant) of a client's scheduling policy and parameters relevant to that scheduling policy from a client to a servant. In addition, proposals shall describe the influence these policies and their parameters have upon the resource management of the servant."

This submission affords the propagation of scheduling parameters from the client to the servant via a scheduler implementation's use of service contexts.

"6. Proposals shall specify mechanisms, and interfaces to those mechanisms, for avoiding or bounding failure to satisfy the furnished scheduling policy request (scheduling policy inversion), and shall define the conditions under which scheduling policy inversion occurs."

The submitters believe that the interfaces and semantics of this specification allow for implementations that that avoid or bound scheduling discipline priority inversions. Note that these mechanisms do not guarantee that poor implementations achieve this goal.

"7. Proposals shall specify interfaces to provide scheduling policy-driven allocation and reallocation of resources for any resource whose use impacts the execution time, or the end-to-end predictability thereof. The goal of the scheduling policy driving the allocation and reallocation is to achieve sufficient end-to-end completion time predictability of the schedulable entity."

This submission provides for the implementation of scheduling disciplines for managing resource allocation via admission

control, but does not specify the interfaces necessary to implement such a scheduler.

“8. Proposals shall define interfaces to mechanisms for reporting violation of scheduling parameters to the application, and/or to a third party. For each violation to be reported, proposals shall define where in the sequence of scheduling and executing the schedulable entity the violations would be reported. For example, reports might be made when the schedulable entity is initially scheduled, when some other schedulable entity’s admission to the schedule causes load shedding or reduced service to the subject schedulable entity, or at some point during execution of the schedulable entity.”

Scheduling violations are communicated to applications via the CORBA::SCHEDULING_FAILURE exception.

“9. Proposals shall define any required exception reporting mechanisms, and interfaces to the mechanisms.”

The submission defines interfaces and the semantics of the required exception reporting mechanisms.

“10. Proposals shall specify the semantics of the proposed dynamic scheduling with respect to fixed priority interfaces and mechanisms.”

The interfaces and semantics of a fixed priority scheduler is provided in this submission.

1.5.2.3 Optional Requirements

1. Proposals may specify an interaction protocol (i.e. analogous to GIOP/IIOP) to provide interoperability between conforming implementations of the proposed standard, supported by reasons why GIOP/IIOP is not sufficient for achieving end-to-end predictability that is acceptable to the intended applications.

This submission does not modify or replace GIOP or IIOP.

2. Proposals may describe mechanisms for managing load allocation in the context of dynamic scheduling, including static allocation, dynamic allocation, or both, and any interfaces to these mechanisms.

This submission provides an infrastructure that facilitates schedulers that manage load allocation, either statically or dynamically, but does not specify the interfaces or semantics to such a scheduler.

2 Submission Overview and Rationale

2.1 Overview

2.1.1 Dynamic Scheduling

In real-time, we distinguish between two types of distributed systems based on how the system is used, and its impact on the underlying infrastructure. There are *static* and *dynamic* distributed systems.

Static distributed systems are those where the processing load on the system is within known bounds such that a-priori analysis can be performed. This means that the set of applications that the system could be running is known in advance, and that the workload that will be imposed on each application can be predicted within a known bound. Such systems often have a limited number of application configurations that can be executed, sometimes referred to as system modes. In these systems, a schedule for the execution of applications can be worked out in advance for each system mode, with a bounded amount of variation. As a result, the underlying infrastructure (operating system and middleware) need only be able to support executing that schedule.

One common approach to static systems is the use of Operating System Priorities to manage deadlines. Offline analysis is performed to map different application temporal requirements (such as frequency of execution) onto the available priorities. If the underlying infrastructure always respects these priorities, including preempting low priority threads when higher priority threads become eligible to run, and providing priority inheritance, this is sufficient.

Dynamic distributed systems, on the other hand, may not have a sufficiently predictable workload to allow this approach. It may be that the set of applications is either too large or not known in advance, the processing requirements for an application is too variable to be pre-planned, the arrival time of the inputs is too variable, or some other source of variability. For these types of systems, the underlying infrastructure must be able to satisfy real-time requirements in a dynamically changing environment.

This proposed specification is focused on systems in which the discipline for scheduling CORBA ORB and application threads (e.g., highest priority first, or earliest deadline first, or least laxity first) may be chosen by the application or system designers; and the scheduling input values needed by a scheduler to control execution (called *scheduling parameter elements* in this document) for that scheduling discipline (e.g., priority, deadline, expected execution time) may be changed by the application dynamically (i.e., at any time). In contrast, Real-Time CORBA 1.0 (ORBOS/99-02-12 with ORBOS/99-03-29) is focused on fixed priority systems.

2.1.2 *Distributable Thread*

This proposal replaces the term and concept of an *activity* that appeared as a design and analysis suggestion in Real-Time CORBA 1.0 with a specification for an end-to-end schedulable entity termed *distributable thread*.

2.2 *Rationale*

Dynamic scheduling is widely employed in real-time and distributed real-time computing systems. This proposal extends Real-Time CORBA 1.0 to encompass these dynamic systems as well as static systems.

In most real-time systems, especially distributed systems, cost-effectiveness demands that the computing system employ as much application-specific knowledge about the application and its execution environment as feasible. Much of this knowledge can be best captured in the scheduling discipline. This proposal allows such application-specific scheduling disciplines to be implemented by a pluggable scheduler.

An end-to-end execution model is essential to achieving end-to-end predictability of timeliness in a distributed real-time computing system. This is especially important in dynamically scheduled systems. The end-to-end execution model may be provided according to a formal standard specification (as herein), or as an ad hoc, custom-made creation by multiple different application programmers.

2.3 *Goals of this Proposal*

This proposed specification generalizes the Real-Time CORBA 1.0 (ORBOS/99-02-12 and ORBOS/99-03-29) specification to meet the requirements of a much greater segment of the real-time computing field. There are three major generalizations:

- any scheduling discipline may be employed;
- the scheduling parameter elements associated with the chosen discipline may be changed at any time during execution;
- the schedulable entity is a *distributable thread* that may span node boundaries, carrying its scheduling context among scheduler instances on those nodes.

While the interfaces have been changed, this proposed specification is backward compatible with the semantics of the Scheduling Service defined in the Real-Time CORBA 1.0 specification. Implementations of both specifications may be used in different ORB instances within the same system. Not all features of this specification can be used in such mixed systems.

This proposed specification imposes no requirements on base real-time operating systems, other than the conventional ability to dispatch threads in a pre-emptive fashion.

2.4 Scope

This proposed specification adds interfaces for a small set of well known scheduling disciplines to Real-time CORBA as optional compliance points. This proposed specification does not attempt to provide all the interfaces necessary for interoperability of dynamically scheduled applications and schedulers in heterogeneous systems. Rather, the submission provides a framework upon which schedulers can be built and lays the foundation for future full interoperability.

This proposed specification defines a set of ORB/scheduler interfaces that will allow the development of portable (i.e. ORB implementation independent) schedulers. For the defined disciplines, this proposed specification also specifies interfaces that will allow the development of portable (i.e. ORB and scheduler implementation independent) applications. Portable application interfaces for other scheduling disciplines is left to future specifications.

Note that most scheduler implementations will extensively utilize features of the underlying operating system, and in some cases the networking software. This aspect of scheduler implementation is outside of the scope of this submission. Therefore, the submission does not provide the portability of schedulers except with respect to ORB interactions.

This submission does not provide interoperability between scheduling disciplines and thus not between different scheduler implementations. A scheduling framework is provided and the mechanism used for passing information between scheduler instances is provided via GIOP service contexts. However, the format and content of the information passed in the GIOP service contexts are not specified. On-the-wire interoperability between scheduling disciplines and the corresponding scheduler implementations is left to future specifications.

The submission provides an abstraction for distributed real-time programming (the *distributable thread*). The submission does not attempt to address more advanced issues such as fault tolerance, propagation of system information and control along the path of a distributable thread, etc. These facilities may be provided in a subsequent revision of this submission.

3 Concepts

3.1 Sequencing: Scheduling and Dispatching

Usually multiple execution entities (hereafter referred to as “threads”) contend for one or more exclusively accessed resources – notably processor cycles, but also others, both physical (e.g., communication paths) and logical (e.g., synchronizers). This contention must be resolved into a sequence of resource accesses – e.g., thread executions. In general, contention for all shared resources should be resolved in a consistent manner, although this is not yet common practice – e.g., processors may be allocated by priority, networks by first come first served, locks by serializability, disks by head movement distance, etc. All resource contention can be resolved by one of two sequencing means: either scheduling or dispatching.

Thread *scheduling* is deciding in what order they all will execute. Each time thread scheduling is performed, a sequence is established – a schedule – for all threads ready at that time. Scheduling is performed *statically* (prior to execution time), by a person or a program, or *dynamically* (at execution time) by a user or the system software.

Thread *dispatching* is granting resource access – e.g., running the currently most eligible thread. When scheduling is employed, dispatching occurs in schedule order.

Thread scheduling is not always necessary nor computationally feasible – dispatching alone may be sufficient.

Moreover, some actions are never threads and thus not schedulable – most commonly, interrupt service routines and certain OS services, which execute either when invoked or automatically as needed (other OS services are scheduled in concert with application entities).

Dispatching, when scheduling is not employed, establishes a thread resource access – e.g., execution sequence – one thread or non-schedulable action at a time.

The execution sequence may change at a sequencing point (either a *scheduling point* or a *dispatching point*), such as when a thread becomes ready or blocked, or a thread contends for a resource, or a thread time constraint is violated.

Contention for execution (and all other sequentially shared physical and logical resources) generally should be resolved according to an application-specific *sequencing optimality criterion* that seeks maximal usefulness to the system. In real-time systems, that usefulness is based primarily on (but not limited to) timeliness and predictability of timeliness. (Other factors, not related to real-time, commonly found in sequencing criteria include relative importance, precedence constraints, resource ownership, etc.)

Sequencing optimality criteria are what define *timeliness* for a given system or application. Consequently, they also distinguish hard and soft real-time. *Hard real-time* has a single timeliness factor in its sequencing optimality criterion: always meet all hard deadlines. *Soft real-time* includes all other possible timeliness factors in sequencing (usually scheduling) optimality criteria – very common examples are “minimize mean weighted tardiness,” “minimize the number of missed deadlines according to importance,” and “minimize maximum tardiness.”

Informally, a property is *predictable* to the degree that it is known in advance. One end point of the predictability scale is *determinism*, in the sense that the property is known exactly in advance. The other end point of the predictability scale can be characterized as maximum entropy, in the sense that nothing at all is known in advance about the property. In stochastic real-time systems (which include hard real-time systems as a special case), one well-defined way to measure predictability is coefficient of variation C_v , which is defined as $\text{variance}/\text{mean}^2$. The deterministic distribution, $C_v = 0$, and the extreme mixture of exponentials distribution is an example of a maximally non-deterministic property whose $C_v = \infty$.

In every real-time system, timeliness of each application and system action is somewhere on this predictability scale. Hard real-time systems have deterministic timeliness in the sense that they always meet all of their hard deadlines. Soft real-time systems have non-deterministic timeliness – e.g., characterized stochastically, such as minimizing either mean or maximum tardiness.

Given a sequencing optimality criterion, a *sequencing discipline* is selected or devised to satisfy it. There are a great many widely used sequencing disciplines; common examples in real-time computing systems include highest priority first (or just “priority”), earliest deadline first (EDF), and least laxity first (LLF). There may be more than one discipline that satisfies a given criterion – e.g., the hard real-time criterion is satisfied by: the EDF and LLF disciplines (under specific conditions), among others; or appropriate assignment and manipulation of priorities. Conversely, a specific discipline may be suitable for different criteria: EDF satisfies the hard real-time criterion, and also satisfies the soft real-time criterion “minimize maximum tardiness” (among others); priorities can be used to satisfy either the hard or various soft real-time criteria.

When scheduling is employed, the sequencing discipline is usually called a *scheduling discipline*, and when only dispatching is employed, the discipline is usually called a *dispatching rule*.

A sequencing discipline is implemented by a *sequencing algorithm*. In general, a discipline can be implemented by many different possible algorithms.

This proposal uses the term *scheduling* to include the case when scheduling (and thus dispatching in schedule order) is employed, and the case when only dispatching is employed, because both of those

cases involve selecting a sequencing optimality criterion and a corresponding discipline and algorithm.

3.2 Well Known Scheduling Disciplines

There are many widely used scheduling disciplines, but real-time computing theory and practice are focused on a small number of them, some of which are summarized below. The constructs in the IDL will become clear later in this proposed specification.

3.2.1 Fixed Priority Scheduling

The fixed priority scheduling discipline provides for pre-emptive scheduling or dispatching of threads based on a simple numeric priority. When a higher priority thread is created or becomes unblocked, it pre-empts a lower priority executing thread and executes immediately.

```

module FP_Scheduling
}
  loc {
    include Scheduling
  }
}

{
  Segment
}
  {
    Priority
  }
}

loc {
  include Segment
}
}
  {
    Segment
  }
}

loc {
  include SegmentPolicy
}
}
  {
    SegmentPolicy
  }
}

loc {
  include SegmentPolicy
}
}
  {
    SegmentPolicy
  }
}

}

```

Note that an analysis technique for scheduling fixed priority systems is Rate Monotonic Analysis (RMA). The rate monotonic analysis assigns fixed priorities to periodic threads based on their execution rates or periods – the thread having the highest rate (shortest period) is assigned the highest priority. Normally, the characteristics of all threads and their execution environment are known in advance, and rate monotonic scheduling is statically performed off-line. In this case, the Real-Time CORBA 1.0 Fixed Priority discipline can be employed. Often thread behavior or execution environment characteristics such as system

loading vary with some dynamic parameter, time or date, operational status of supporting systems, etc.

3.2.2 *Earliest Deadline First (EDF)*

The earliest deadline first discipline uses the execution completion deadline of the threads as the basis for their execution eligibility – a thread that has a shorter (closer) deadline is more eligible than one with a longer deadline. In some cases, a thread’s deadline is constant during the thread’s lifetime, and in other cases it changes (for example, a thread’s deadlines may be nested). When EDF is used to meet deadlines (i.e., for hard real-time), it requires that all deadlines can be met, in which case it is most often employed statically. Other factors can be used in conjunction with deadlines to create enhanced EDF-like disciplines that always meet all deadlines if possible, and that shed or defer load when overloaded. When EDF is used to minimize maximum tardiness (i.e., for soft real-time), it may be employed either statically or dynamically. EDF can be employed either as a scheduling discipline or as a dispatching rule.

```

module F_Scheduling
}
  loc} in{e Schedule}
    }Scheduling$chedule}
  }
}
}
}SchedulingP{e}
}
  }me{ime}deadline}
  long           im{p}ce}
}
  loc} in{e SchedulingP{e}Polic}
  }Polic}
}
  }{e SchedulingP{e}}ue}
  }c SchedulingP{e}Polic}
    c{e}
  }n SchedulingP{e}}ue}
}
}

```

3.2.3 *Least Laxity First (LLF)*

A least laxity (or “time to go”) first discipline assigns execution eligibility based on laxity value, where

$$\text{laxity} = \text{deadline} - \text{current time} - \text{estimated remaining computation time.}$$

A thread with lower laxity is more eligible than one with higher laxity. An LLF discipline is sometimes used for environments where thread execution time requirements vary significantly. In such environments, a thread with a long execution time may be released prior to threads with less laxity becoming ready-to-run. The laxity estimate is updated as the thread execution duration estimate is updated at run time. An LLF discipline may specify that a thread with negative laxity should not (continue to) execute. Thus, LLF is primarily a dynamic discipline. LLF may be used either to meet deadlines (i.e., for hard real-time) or to

not have trans-node end-to-end timeliness requirements that are used by the node schedulers. That is the common non-real-time case.

Case 2 is that scheduling occurs independently on each node, but an application behavior that involves more than one node propagates its end-to-end timeliness context, which is then used by each node's scheduler while the behavior is active at that node. System-wide scheduling is coherent but not generally globally optimal. That is the distributed real-time case this proposed specification explicitly addresses.

Case 3 is that scheduling on each node is global in the sense that there is a logically singular system-wide scheduling algorithm instantiated on all nodes, and node instances of this algorithm interact to schedule all nodes in a globally optimal way. This proposed specification does not explicitly support case 3 because such scheduling is very difficult (intractable in general) from both the conceptual and implementation standpoints, but desires not to preclude it.

Case 4 is all the multi-level scheduling cases: there is at least one level of "meta-scheduling" above the case 1 or 2 node schedulers that seeks to improve global optimality by adaptively adjusting some combination of scheduling parameter elements, schedulable entity, scheduling contexts, scheduling algorithms, scheduling disciplines, and node load balancing. Case 4 includes sub-cases corresponding to cases 2 and 3. This proposed specification does not explicitly support case 4, but again, desires not to preclude it.

3.4 *Distributable Thread*

This proposed specification replaces the term and concept of an *activity* that appeared as a design and analysis suggestion in Real-Time CORBA 1.0 with a definition for an end-to-end schedulable entity termed *distributable thread*. Real-Time CORBA 1.0 left the details of the activity abstraction unspecified, but such a distributed execution model is essential in dynamically scheduled real-time CORBA systems. The term has been changed to avoid conflict with prior usage in CORBA specifications such as Workflow Management (formal/00-05-02) and Additional Structuring Mechanisms for the OTS Specification (orbos/00-04-02).

The defining characteristic of any real-time distributed computing system, whatever its programming model, is that the end-to-end timeliness (optimality and predictability of optimality, as defined in Section 3.1) of trans-node application behaviors is acceptable to the application.

In most cases, the fundamental requirement for achieving acceptable end-to-end timeliness is that a trans-node application behavior's timeliness properties and parameters – time constraints, expected execution time, execution time received thus far, etc. – be explicitly employed for resource management (scheduling, etc.) consistently on each node involved in that application trans-node behavior. As stated in

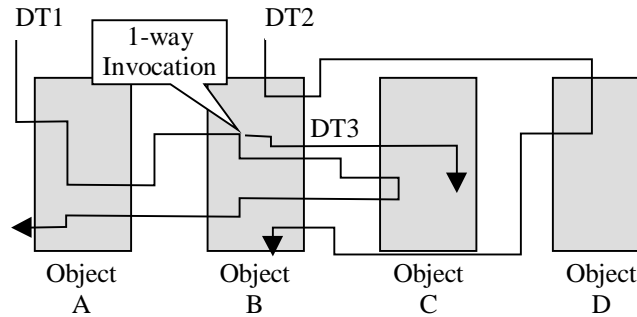


Figure 2: Distributed Threads

A distributable thread can extend and retract its locus of execution points among operations in object instances across physical computing nodes by location-independent invocations and (optionally) returns. Within each node, the flow of control is equivalent to normal local thread execution.

The synchrony of a conventional two-way operation invocation (or RPC) programming model is often cited as a concurrency limitation. But that criticism does not apply to the distributable thread model. A distributable thread is a sequential abstraction, like a local thread. A distributable thread is always executing somewhere, while it is the most eligible there – it is not doing send/wait’s as with conventional operation invocations. Remote invocations and returns are scheduling events at both client and servant nodes. Each node’s processor is always executing the most eligible distributable thread while the others wait. A distributable thread always has exactly one execution point (*head*) in the whole system. New distributable threads may be created, or sleeping ones awakened, when needed. An application or system may have multiple distributable threads. Multiple distributable threads execute concurrently and asynchronously, by default. Distributable threads synchronize through operation execution; the writers of each object control distributable thread concurrency in that object. An exception that occurs anywhere along a distributable thread’s locus of execution can be forwarded to and raised at the head of that distributable thread. Subsequently, the exception propagates from the head back up the distributable thread to the nearest enclosing exception handler.

Distributable thread-based programming models imply the need for a number of supporting facilities; these programming models can be differentiated by the facilities that they provide, and the approaches employed to provide them. Not all, or any, of these facilities are included in this proposed specification. These supporting facilities include (but are not limited to) the following.

- Some asynchronous “happenings” (i.e., changes in system state) of interest to a distributable thread may have to be coordinated with current distributable thread execution. For example, a violated time constraint, or the failure of a node or network path over which a distributable thread is extended, might require notification of the distributable thread’s head – as soon as possible, if the distributable thread is currently

executing, and otherwise as soon as the distributable thread becomes the most eligible to execute. Certain other events that occur at the distributable thread's head – e.g., synchronous exceptions (e.g., traps) and asynchronous exceptions (e.g., time constraint expirations) – may require the distributable thread to execute a local exception handler and then return back up the invocation chain to execute one or more appropriate exception handlers at those places. After such an exception, the programming model could allow the distributable thread to either continue execution where the exception was initially delivered (a *continuation* model) or terminate, or the model could require that the distributable thread always terminate (a *termination* model).

- Distributable thread control actions – e.g., suspend, resume, abort, time constraint change, etc. – may have to be propagated to, and carried out at, the distributed thread's head.
- Mechanisms may have to be provided to support maintaining correctness of distributed execution, and consistency of distributed data – in both cases, as defined by the application – for concurrent activities of one or more applications.
- The code that is responsible for detecting/suspecting failure for an appropriate set of nodes may require visibility to failures locally perceived by a distributable thread.

All of these facilities generally would be required to be timely – e.g., subject to completion time constraints.

4 Overview of the Programming Model

This section presents an overview of the application programming model that is being provided. Since the proposed specification defines a scheduling framework, as well as a limited set of scheduling disciplines, this section deals with the concepts that apply across schedulers and scheduling disciplines.

4.1.1 Scheduler

In this proposed specification a scheduler is realized as an extension to Real-time CORBA that utilizes the scheduling needs and resource requirements of one or more applications to manage the order of execution of those applications on the distributed nodes of a CORBA system. A scheduler provides operations for applications to announce their requirements, which the scheduler takes into consideration when it affects the order in which threads are dispatched by the operating system.

A scheduler will be run in response to specific application requests, such as defining new scheduling parameter elements, and in response to specific application actions, such as CORBA invocations. The latter will be implemented using the CORBA Portable Interceptor interfaces. The scheduler utilizes the information provided in these interfaces to manipulate which threads are most eligible for execution by the underlying operating system. This control is via whatever interfaces the operating system provides, which are outside of the scope of CORBA. Thus, although scheduler implementations could be independent of any particular ORB implementation, as long as the ORB conforms to this specification, the scheduler will be closely tied to the operating system. The scheduler architecture is based on the premise that a distributed application can be considered to be a set of distributable threads (see Section 3.4), which may interact in a number of ways, sharing resources via mutexes, sharing transports, parent/offspring relationships, etc. The mechanisms of interaction are irrelevant to this submission.

The scheduler architecture assumes that the problem of satisfying scheduling needs can be addressed by managing the allocation of resources to distributable threads. The distributable thread provides a vehicle for carrying scheduling information across the distributed system.

Distributable threads interact with the scheduler at specific scheduling-points, including application calls, locks and releases of resources, and at pre-defined locations within CORBA invocations. The latter are required because CORBA invocations are points at which the distributable thread may transition to another processor, and the scheduling information must be reinterpreted on the new processor.

4.1.1.1 Scheduler Characteristics

This proposed specification does not assume a single scheduling discipline for Real-time CORBA. Schedulers are developed to implement a particular scheduling discipline or disciplines. Both available products and technical literature abound with examples of schedulers implementing various scheduling disciplines. This proposed specification defines only the interface between the ORB/application and the scheduler, and is intended to foster the development of schedulers that are not dependent on any particular ORB (although a particular scheduler implementation may choose to take advantage of the features of a particular ORB). Note that schedulers will likely be dependent on the underlying operating system, and this submission does not address these operating system interfaces, since they are outside of the scope of CORBA.

This proposed specification addresses schedulers that will optimize execution for the application scheduling needs on a processor-by-processor basis (see Section 3.3, case 2). That is, as the execution of an application distributable thread moves from processor to processor, its scheduling needs are carried along and honored by the scheduler on each processor. This does not preclude the development of schedulers that perform global optimization, but this proposed specification does not specifically address that type of scheduler.

The schedulers considered in relation to this specification will have in common processing stages where they acquire information about the demand for resources, an optional processing stage where they plan the processing schedule (when scheduling, as opposed to dispatching alone, is used), and a processing phase where they affect how threads are dispatched by the operating system. This proposed specification does not impose any requirement on how the scheduler developer defines these processing stages. The proposed specification does define the minimum set of scheduling points (points in time or code when the scheduler will execute).

This proposed specification also provides the scheduler APIs for a small set of scheduling disciplines, including fixed priority, as defined in Real-Time CORBA 1.0. This supports application portability for these disciplines.

The current proposed specification does not address full interoperability across scheduler vendor implementations; to achieve this, one would have to define the scheduling discipline, the scheduling parameter elements, and the service context that is used to propagate the scheduling characteristics of the application. The submitters believe that more implementation experience is needed before full interoperability is possible. Therefore, this proposed specification only provides a complete API definition for a limited set of well-understood scheduling disciplines and does not define a standard service context for any scheduling disciplines. Future specifications will define standardized service contexts and the APIs for additional disciplines.

4.1.1.2 Scheduling Parameter Elements

This proposed specification defines a *scheduling parameter* as a container of potentially multiple values called *scheduling parameter elements*. The scheduling parameter elements are the values needed by a scheduling discipline in order to make scheduling decisions for an application. A scheduling discipline may have no scheduling parameter elements, only one, or several; the number and meaning of the scheduling parameter elements is scheduling discipline specific. A single scheduling parameter element is associated with an executing thread via the `begin_scheduling_segment` operation. A thread executing outside the context of a scheduling segment has no scheduling parameter associated with it and is scheduled by the native scheduling of the operating system, typically priority based.

Some scheduling disciplines will acquire the information about application resource and scheduling requirements at system/application design time (static scheduling); these schedulers often load the resulting scheduling information into a data structure which is accessed at run time. Other schedulers are intended to react to dynamic runtime system demands (dynamic scheduling). This specification addresses provides a general scheduler interface that can be used by either type of scheduling discipline.

The submission also allows various types of interactions for static scheduling. The specific approach to be used will be discipline-specific. For example, the application may provide its scheduling parameter elements, and the associated names, in advance so that the scheduler can store them internally; this could be done during some form of application initialization. Alternatively, the application can provide scheduling parameter elements each time it invokes scheduler operations.

The specific information needed by a scheduler will depend on which discipline(s) it implements. For example, simple deadline scheduling may need only the thread's deadline and the amount of CPU time that the thread will consume. Another discipline might utilize relative importance as one of its inputs. This specification has defined a standard interface for passing a set of scheduling discipline information to a scheduler. The definition of the structure, types, and the handling of missing parameter elements are discipline-specific.

4.1.1.3 Pluggable Scheduler and Interoperability

This proposed specification provides a "pluggable" scheduler. A particular ORB in the system may have any scheduler installed, or may have no scheduler. If an ORB has a scheduler installed, all applications run on that ORB are "under the purview" of that scheduler.

Application components may interoperate, in the context of a particular scheduling discipline, as long as their ORBs have compatible schedulers installed (meaning that the schedulers implement the same discipline, and follow a CORBA standard for that discipline) and the scheduler implementations use a compatible service context. As noted

above, the current proposed specification does not define any standard service contexts, although future specifications are anticipated in this area.

A scheduler may choose to support multiple disciplines, but the current proposed specification does not address this.

4.1.1.4 *Distributable Threads*

A distributable thread (see Section 3.4) is the fundamental abstraction of application execution in this proposed specification. A distributable thread incorporates the sequence of actions associated with a user-defined portion of the application that may span multiple processing nodes, but that represents a single logical thread of control. Distributed applications will typically be constructed as several distributable threads that execute logically concurrently.

More precisely, a distributable thread is the locus of execution between points in the application that are significant to the application developer, and it carries the scheduling context of the application from node to node as control passes through the system. It might encompass part of the execution of a local (or native) thread or multiple threads executing in sequence on one or more processors. If it encompasses multiple threads, then it also encompasses the various phases, i.e., "in-transit", "static", "active", etc., which might occur as the locus of execution moves among threads.

A distributable thread may have a scheduling parameter containing multiple element values associated with it. These scheduling parameter elements become the scheduling control factor for the distributable thread and are carried with the distributable thread via CORBA requests and replies. Scheduling parameter elements can be associated with a thread by the application invoking the `begin_scheduling_segment` operation (see "Scheduling Segments, Parameter Elements, and Schedulable Entities" below). The application may call the `end_scheduling_segment` operation to create a distributable thread and a corresponding native thread in the current processor and associate scheduling parameter elements with it.

A distributable thread has at most one head (execution point) at any moment in time. If there is a branch of control, as occurs with a CORBA oneway invocation, the originating distributable thread remains at the client and continues execution (as long as it remains the most eligible).

Each distributable thread has a globally unique id within the system, which can be accessed via the `get_thread_id` operation. The distributable thread id can be used to obtain a reference to a distributable thread, via the `lookup` operation. This reference can then be used to cancel that distributable thread, via the `cancel` operation. The `cancel` operation results in a `ThreadException` system exception being raised in the cancelled distributable thread.

4.1.1.5 *Implicit Forking and Joining*

Typically, an intrinsic part of any concurrency model is the semantics for the creation of new execution contexts, or *forking*, and the synchronization of multiple execution contexts, or *joining*.

Explicit forking is provided for in this specification by the `fork` operation. Due to time constraints explicit joining was provided by this specification. Future finalizations and revision task forces are encouraged to provide for this capability.

Certain aspects of the core CORBA programming model and the programming model of various CORBA services introduce the implicit forking of distributable threads. One example in the core CORBA specification is *oneway* invocations if made with a synchronization scope of `SYNC_NONE` or `SYNC_WITH_TRANSPORT`. This occurs because the distributable thread making the invocation is unblocked before the operation on the servant executes.

When a distributable thread executing a scheduling segment implicitly forks another distributable thread, the forked distributable thread's scheduling parameter is determined as follows:

- If the implicit scheduling parameter is set for the innermost scheduling segment of the forking distributable thread then the ORB must use this value in implicitly forking any distributable threads.
- Otherwise, the ORB must use the operative scheduling parameter of the innermost scheduling segment for the implicit forking of any distributable threads.

As with forking, there are certain aspects of the core CORBA programming model and the programming model of various CORBA services that introduce the implicit joining of distributable threads. An example of an implicit join is the polling mode introduced by asynchronous messaging. This occurs because the distributable thread calling the poll operation can wait to "join up with" the distributable thread that ran the operation on the servant to get the results of the asynchronous invocation. Note that the initial asynchronous invocation call is an implicit fork that results in the distributable thread used to run the operation on the servant.

When a distributable thread executing a scheduling segment implicitly joins another distributable thread, there is not inheritance or propagation of either distributable thread's scheduling parameter to the other distributable thread.

4.1.1.6 *Scheduling Segments, Parameter Elements, and Schedulable Entities*

In this proposed specification, distributable threads consist of one or more (potentially nested) *scheduling segments*. Within a distributable

thread, scheduling segments can be sequential and/or nested. Nesting creates *scheduling scopes*.

Each scheduling segment represents a sequence of control flow with which a particular set of scheduling parameter elements is associated. A scheduling segment is delineated by `begin_scheduling_segment` and `end_scheduling_segment` statements in the code. The application may use the segment name on the end statement, as an error check. The scheduling parameter associated with a distributable thread may be updated with a call to `update_scheduling_segment`.

At runtime, a scheduling segment has a single starting point, and a single ending point (although it could be coded with multiple possible ending points, during execution only one ending point can be invoked). Segments may span processor boundaries. This proposed specification places no restrictions on the placement of `begin_scheduling_segment`'s and `end_scheduling_segment`'s; an `end_scheduling_segment` may occur on a different processor than the `begin_scheduling_segment`, and may even occur somewhere up the chain of CORBA requests.

As a distributable thread moves from object instance to object instance through CORBA invocations, it may extend (and possibly retract) itself through one or more processes or processors. When this happens, the distributable thread may be contending with a new set of distributable threads for resources.

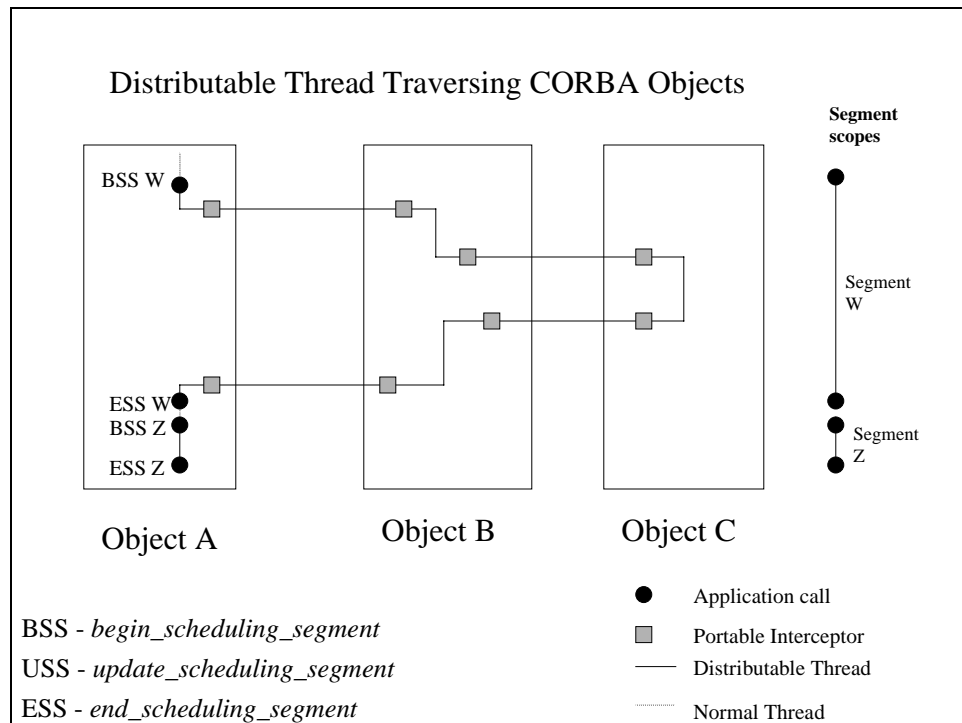


Figure x1 – A Distributable Thread with two sequential segments.

Figure x1 illustrates a simple distributable thread which contains two sequential segments. The distributable thread begins in object instance A, with segment W, and traverses object instances B and C before returning to A, where the first segment ends and a new segment (Z) begins. Portable interceptors are invoked each time the distributable thread transitions to another object instance via a CORBA request (on both the client and servant side) and again as the distributable thread returns. Note that these object instances could be on different processors.

Suppose the scheduling discipline is Earliest Deadline First, which implies that the illustrated distributed thread must (implicitly) carry its deadline along as it progresses through the various processor environments. Further, assume that the scheduling discipline calls for scheduling segments that have missed their deadline to be terminated. This last condition implies that the scheduler must be maintaining a list of deadlines. The `begin_scheduling_segmen`, `update_scheduling_segmen` and `end_scheduling_segmen` operations serve to enter, update or remove deadlines, but the scheduler must also address what happens when a set deadline expires.

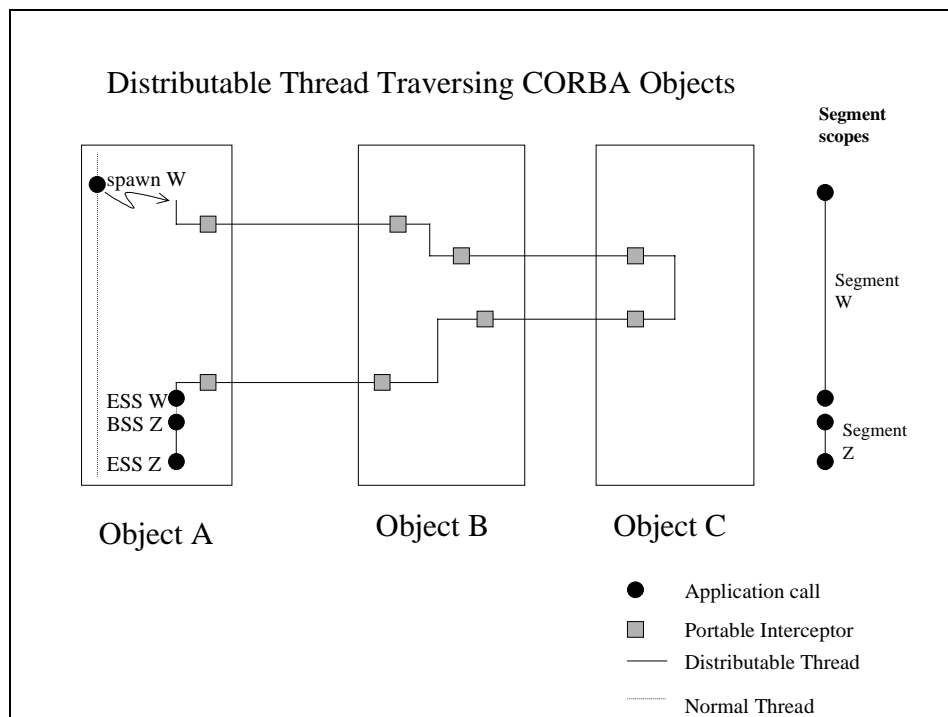


Figure x2 – A Distributable Thread created by a spawn operation

Figure x2 illustrated the use of a *spawn* to create a distributable thread. Note that the spawn also serves as the beginning for the initial segment (W) of the distributable thread.

Some scheduling disciplines may support the nesting of scheduling segments, which permits independently developed software components to define their own scheduling segments. The component

would create an additional scheduling segment by embedding one or more pair of calls to `begin_scheduling_segment` and `end_scheduling_segment`. The handling of unspecified parameter elements (defaulting) is discipline-specific. In some cases, unspecified elements will use the values from the next outer segment (if any). In other cases, predefined or application defined default values might be used.

Each `begin_scheduling_segment` provides a new set of scheduling parameter elements for the distributable thread. If the distributable thread is already in a segment, these new parameter elements will replace the current set until a matching `end_scheduling_segment` occurs. An `end_scheduling_segment` statement causes the distributable thread to return to the previous scheduling parameter (if any). Thus, a distributable thread may contain multiple scheduling segments that are executed sequentially, each of which may contain nested segments. This proposed specification does not place any limits on the level of nesting that a scheduling discipline will support.

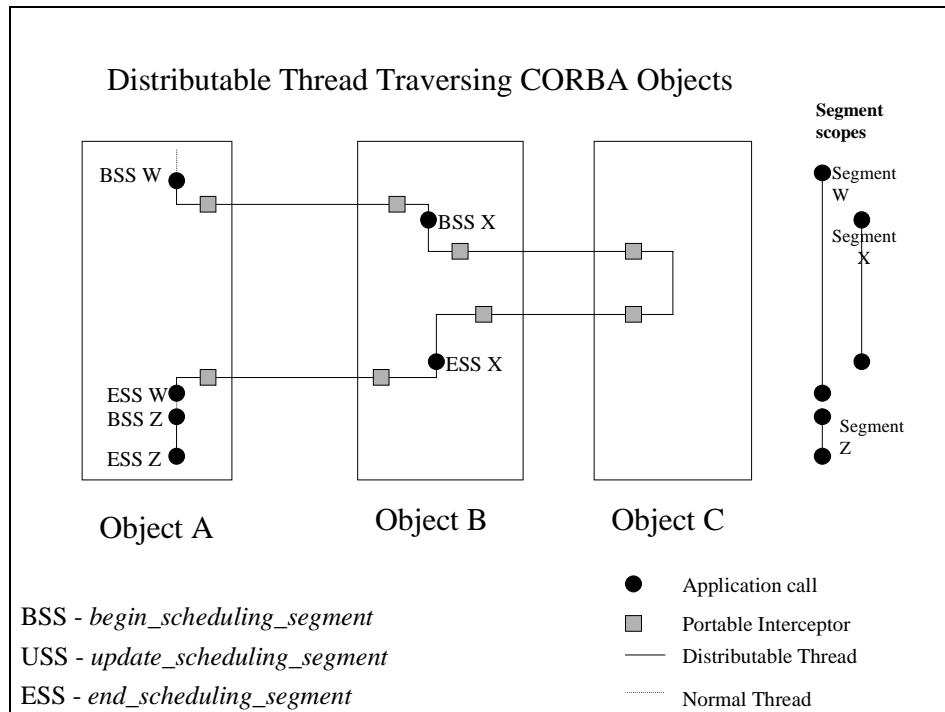


Figure x3 – Distributable Thread with Nested Segments

Figure x3 illustrated segment nesting. In this case, segment X is nested within segment W. At the point where segment X begins, the scheduling context of segment W is logically pushed onto a stack, and segment W's scheduling parameter elements are used for the distributable thread. When segment X ends, the distributable thread returns to the scheduling parameter elements for segment W.

In the case of EDF, all of these segments involve the requirement that they complete by some deadline, but they would probably be different

deadlines. In the case of nested segments (W, X, and Y) the tightest deadline may come from any of the segments.

A distributed thread executing in a single object instance may, at different times, have different deadlines. Note that where the distributed thread first executes in object instance B its deadline will be the deadline for segment W. However, as soon as segment X begins, the deadline must be selected from the tighter of the outer (W) or inner(X) scheduling segment.

It is expected that each instance of the scheduler must monitor the time constraints of every distributed thread that is currently traversing its node.

A scheduling parameter element that is created in one object instance must be considered in other object instances as the distributed thread passes through them. In the illustration, the deadline established in object instance A must be considered with respect to all other deadlines that exist in the domain of object B, and similarly as the distributed thread extends to object C.

How a scheduler addresses distributed dynamic scheduling is implementation dependent, but it is likely that the features of the portable interceptor will be required. By requiring use of an interceptor that targets the scheduler for the outgoing and incoming sides of the connection at both the client and server sides, the scheduler can address these characteristics. A client-side outgoing interceptor can address moving the deadline compliance monitoring while the associated server side incoming interceptor can address the continuing deadline compliance monitoring and distributed thread scheduling with respect to the server side workload.

The application may also invoke the scheduler within a segment, either to allow the scheduler to notify the application if it has had a scheduling failure (such as a missed deadline), or to modify the current segment's scheduling parameter elements. This is done via the `update_scheduling_segment` operation. The update operation allows the application to occasionally check in with the scheduler, and can also be used to change scheduling parameter elements dynamically, without creating a new segment.

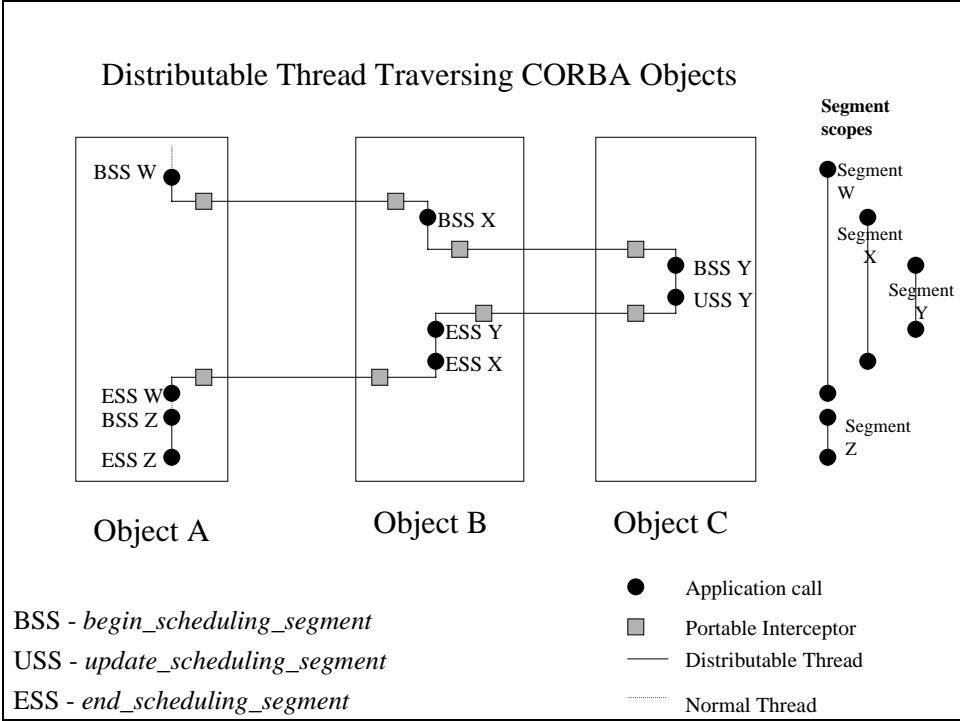


Figure x4 – illustrates the remaining features of scheduling segments, namely the use of multi-level nesting, updates, and the flexible placement of ends. Note that there are two levels of nesting within segment W. In this example, an `update_scheduling_segment` is called within segment Y. Any exceptions for the distributable thread could be delivered at this point, rather than waiting for the next portable interceptor call or the `end_scheduling_segment`. In addition, the application could provide new scheduling parameter elements on the update, without returning to the next upper scheduling scope. Note also, this in this example, segment Y is begun in object instance C, but ended in object instance B, which was the invoker of object instance C.

The application can obtain a list of the current scheduling segment names, innermost scope first, via the `current_scheduling_segment_name` operation.

4.1.2 Scheduling Points

There are a number of *scheduling points*, which are points in time and/or code at which the scheduler is run and may result in an alteration of the current schedule. These include all begins and ends, access to shared resources, and points at which control transfers between processing nodes (i.e. CORBA requests). Because these scheduling points may result in schedule changes, they may also be a point at which dispatching occurs.

The following set of scheduling points is defined:

- Creation of a distributable thread (via `begin_scheduling_block` or `begin_scheduling_block`)
- Termination or completion of a distributable thread
- `begin_scheduling_block`
- `end_scheduling_block`
- `end_scheduling_block`
- A CORBA operation invocation, specifically the request and reply interception points provided in the Portable Interceptor specification
- Creation of a resource manager
- Blocking on a request for a resource via a call to `lock_resource` or `lock_resource`
- Unblocking as a result of the release of a resource via a call to `unlock_resource`

4.1.3 Schedule-Aware Resources

This proposed specification permits the application to create a scheduler-aware resource locally via the *create* operation in a *ResourceManager*; these resources can have scheduling information associated with them via the *register_resource* operation. For example, a servant thread could have a priority ceiling if the application were using fixed priority scheduling. The scheduler will run when these resources are locked or released, so that the scheduling discipline is maintained.

Any scheduling information associated with these resources is scheduling discipline-specific.

4.1.4 Exceptions

The submission defines the following exceptions related to scheduling:

CORBA::SCHEDULER_FAULT – this indicates that the scheduler itself has experienced an error.

CORBA::SCHEDULE_FAILURE – this indicates that the distributable thread has violated the constraints its scheduling parameter. For example, this exception could occur when a deadline has been missed or a segment has used more than its allowed CPU time.

CORBA::THREAD_CANCELLED – indicates that the distributable thread receiving the exception has been cancelled. This may occur

because a distributable thread cancels another distributable thread thereby causing the CORBA::THREAD_CANCELLED exception to get raised at the subsequent head of the cancelled distributable thread.

RTScheduling::DistributableThread::UNSUPPORTED_SCHEDULING_DISCIPLINE – indicates that the scheduler was passed a scheduling parameter inappropriate for the scheduling discipline(s) supported by the current scheduler.

4.1.5 Summary

An application consists of one or more distributable threads (as well as possibly local processor threads which are not part of distributable threads). Each distributable thread will execute through one or a series of (distributed) scheduling segments, including some that may have nested segments. These segments represent regions of execution that have their own scheduling parameter elements. Within these scheduling segments, additional calls may be made to alter the scheduling parameter elements and/or to just allow the scheduler to run.

Distributable threads may evolve from application threads, due to a `{begin_scheduling_segmen}` operation, or be generated by `{}` operations. Distributable threads may be cancelled by another distributable thread, and cancelled distributable threads will be notified of the cancellation via an exception.

These distributable threads may share local resources utilizing resource manager lock, try_lock, and unlock operations which are schedule-respecting.

5 Scheduler Interoperability, and Portability

5.1 Scheduler Interoperability

A CORBA ORB supporting dynamic scheduling will interoperate with an ORB that does not support this capability. The scheduling parameter for a Distributable Thread is passed to the other ORB in the service context field and they can be ignored by the other ORB.

5.2 Scheduler Portability

This proposal addresses the issue of portability between the ORB and scheduler and also between the application and the scheduler. This proposal provides that capability in that it makes the ORB/scheduler interfaces available to applications.

5.3 Dynamic Scheduling Interoperation

This submission does not address interoperation between different dynamic schedulers.

Dynamic Scheduling is an extension of and modification to the RT CORBA specification. Application functions that are scheduled using the fixed priority methods will interoperate with dynamic scheduling tasks. This specification offers the application developer several options with regard to mixed mode operations. For example, a band of priorities can be reserved for dynamically scheduled activities. That band may be located at the high or low end of the priority range or it may be placed in the middle of the priority band. When activities have a priority higher than the dynamic scheduling band then dynamic scheduled activities will only run during what would otherwise be idle time. When dynamic scheduling is given top priority the scheduler resources might be dedicated to some activities while the remainder of the activities are dispatched during periods when the dynamically scheduled activities are not ready to execute.

Schedulers may be constructed so that dynamic scheduling systems can provide services to non-dynamically scheduled CORBA client applications. Requests from such a client would be treated as any processing that occurs without a scheduling parameter set. When dynamically scheduled clients make requests to non-dynamically scheduled servants then the added information carried in the service contexts is ignored. The request is valid but is not dynamically scheduled.

6 Dynamic Scheduling Interfaces

6.1 ThreadAction Interface

6.1.1.1 IDL

```
module $scheduling
{
}

{loc} inherits ThreadAction
{
    void do_n({id})
}
}
```

6.1.1.2 Semantics

The ThreadAction interface is used to provide an entry point for newly spawned distributable threads. The ThreadAction interface serves as a parent type for user implemented ThreadAction objects. The do operation in a ThreadAction object does nothing.

6.2 DistributableThread Interface

6.2.1 spawn Operation

6.2.1.1 IDL

```
module $scheduling
{
}

{loc} inherits ThreadAction
{
    void spawn(
        in ThreadAction
        in unsigned long stack_size,
        in {id} method_name,
        in {id} priority)
}
}
```

6.2.1.2 Semantics

The spawn operation creates a new O/S thread and makes that thread a distributable thread with a stack size at least as large as the value passed in the stack_size parameter. The initial CORBA base priority is the value passed by the priority parameter. The

new distributable thread calls the do operation on the `DistributableThread` object passed via the start parameter.

6.2.2 *UNSUPPORTED_SCHEDULING_DISCIPLINE* exception

6.2.2.1 IDL

```

module Scheduling
{
    local interface DistributableThread
    {
        exception SchedulingException
    }
}

```

6.2.2.2 Semantics

The `UNSUPPORTED_SCHEDULING_DISCIPLINE` exception is raised when a scheduling parameter argument isn't appropriate for the installed scheduler instance.

6.2.3 *begin_scheduling_segment* Operation

6.2.3.1 IDL

```

module Scheduling
{
    local interface DistributableThread
    {
        void begin_scheduling_segment(
            in string name,
            in PolicySet policySet,
            in PolicySet policySet)
            SchedulingException
    }
}

```

6.2.3.2 Semantics

The `begin_scheduling_segment` operation raises the `RTScheduling::DistributableThread::UNSUPPORTED_SCHEDULING_DISCIPLINE` exception when the scheduling_parameter argument didn't have an appropriate value for the active scheduling discipline.

The `begin_scheduling_segment` operation raises the `CORBA::SCHEDULE_FAILURE` exception when the scheduling segment failed its schedule.

The `begin_scheduling_segment` operation raises the `CORBA::SCHEDULER_FAULT` exception when the scheduler has had internal error.

The `begin_scheduling_segment` operation raises the `CORBA::THREAD_CANCELLED` exception when the distributable thread was cancelled.

The `begin_scheduling_segment` operation raises the `CORBA::BAD_PARAM` exception when the `scheduling_parameter`, any elements of the scheduling parameter, or the name parameter was invalid for the installed scheduler.

The `begin_scheduling_segment` operation begins a scheduling segment. A scheduling segment is a window of execution where a distributable thread is executing a particular region of code. The scheduler conditions execution of a particular scheduling segment using the passed `scheduling_parameter` argument.

The name parameter provides identification for the region of code that comprises the scheduling segment. Some schedulers may support nesting of scheduling segments. If a scheduler does not support nesting of scheduling segments this operation raises `CORBA::SCHEDULE_FAILURE`.

A `scheduling_parameter` contains elements that are a value or set of values appropriate for the active scheduling discipline. The `scheduling_parameter` used by the scheduler and set by the application.

The requirements for the "scheduling_parameter" and "name" parameters are scheduling discipline defined. It is expected that at least one these parameters ("scheduling_parameter" or "name") is a non-null argument.

In addition, the `begin_scheduling_segment` operation provides a scheduling point for the scheduler and gives the scheduler an opportunity to cancel a distributable thread by raising the `CORBA::THREAD_CANCELLED` exception while is executing in a scheduling segment.

The `begin_scheduling_segment` call converts the calling thread into a distributable thread if it isn't already one.

Turns a user thread into a distributable thread if it isn't already one.

6.2.4 *update_scheduling_segment Operation*

6.2.4.1 IDL

```
module Scheduling
{
    local interface UpdateSegment
    {
    }
}
```


6.2.5 *end_scheduling_segment Operation*

6.2.5.1 *IDL*

```
module Scheduling
{
    local interface Scheduler
    {
        void end_scheduling_segment(
            string name);
    }
}

```

6.2.5.2 *Semantics*

The `end_scheduling_segment` operation raises the CORBA::SCHEDULE_FAILURE exception when the scheduling segment failed its schedule.

The `end_scheduling_segment` operation raises the CORBA::SCHEDULER_FAULT exception when the scheduler has had internal error.

The `end_scheduling_segment` operation raises the CORBA::THREAD_CANCELLED exception when the distributable thread was cancelled.

The `end_scheduling_segment` operation raises the CORBA::BAD_PARAM exception when the name parameter was invalid for the installed scheduler.

The `end_scheduling_segment` operation ends a scheduling segment. Each call to a `end_scheduling_segment` operation should match a call to `begin_scheduling_segment` made in the same distributable thread. If `end_scheduling_segment` is called in a distributable thread that does not have a matching call to `begin_scheduling_segment` raises CORBA::SCHEDULE_FAILURE.

The `end_scheduling_segment` operation provides the scheduler with a scheduling point and provides an opportunity for the scheduler to check for a scheduling failure.

If a non-null string is passed via the name parameter then the scheduler can verify the name with the name passed in the corresponding `begin_scheduling_segment` call. If a null string is passed then no verification takes place.

6.2.8 Scheduler::scheduling_policies Attribute

6.2.8.1 IDL

```
module Scheduling
{
    local interface Scheduler
    {
        sequence<Policy>
            scheduling_policies();
    }
}
;
```

6.2.9 Scheduler::poa_policies Attribute

6.2.9.1 IDL

```
module Scheduling
{
    local interface Scheduler
    {
        sequence<Policy>
            poa_policies();
    }
}
;
```

6.2.9.2 Semantics

The `scheduling_policies` attribute allows the ORB to request the list of POA policies that the scheduler requires to be applied to all POA's associated with this ORB. A null list is an acceptable result value.

6.2.10 Scheduler::scheduling_discipline_name Attribute

6.2.10.1 IDL

```
module Scheduling
{
    local interface Scheduler
    {
        string
            scheduling_discipline_name();
    }
}
;
```


The requirements for the "`scheduling_discipline`" and "`name`" parameters are scheduling discipline defined. It is expected that at least one these parameters ("`scheduling_discipline`" or "`name`") is a non-null argument.

This is useful for schedulers that associate some scheduling information with a shared resource. An example of this type of scheduler would be a fixed priority scheduler that uses some form of priority ceiling protocol.

7 Conformance

This specification makes changes to the Real-time CORBA 1.0 specification that an implementation must conform to comply with the Real-time CORBA 1.0 specification.

In addition, the implementation of this specification is optional for implementations of the Real-time CORBA 1.0 specification. The implementation of the basic Dynamic Scheduling infrastructure (i.e. the implementation of all interfaces and associated semantics not associated with a particular scheduler) is the most basic form of compliance with this specification.

The implementation of each scheduler (i.e. the implementation of all interfaces and associated semantics for that scheduling discipline) is optional and separate compliance point for a conforming implementation of the basic Dynamic Scheduling infrastructure.

8 Extensions to core CORBA

8.1 Additional System Exceptions

This specification adds three system exceptions to core CORBA:

- CORBA::SCHEDULER_FAULT,
- CORBA::SCHEDULE_FAILURE, and
- CORBA::THREAD_CANCELLED.

See section 4.1.4 for more details.

8.2 IDL Extension: “static” for local interface operations

This specification provides a new IDL keyword that modifies the semantics of operations in local interfaces. The syntax is similar to that of CORBA operations, but the resulting type cannot be remotely invoked. The static meta-type is intended to provide implementers of Dynamic Scheduling CORBA with a language-independent mechanism for declaring operations that do not traverse across the execution context of the ORB.

8.2.1 Grammar

The grammar for specifying static operations is defined by the following BNF:

```
<op_dcl> ::= "static" <op_id> <parameter_dcls> [ <raises_expr> ]
```

```
<op_id> ::= <interface_identifier> "::" <op_identifier>
```

8.2.2 Semantics

The semantics associated with static operations are as follows:

Static operations can be accessed only in the context of the local ORB under which they are created. Any attempt to access a static method through the object reference as a normal CORBA operation, even though it looks like a regular method, will result in a CORBA::BAD_OPERATION exception.

Static operations are associated with the interface in which they are defined. They do not belong to a particular implementation instance of the interface (class) and they can be initialized even before any object is instantiated for the interface to which they belong.

