Realtime CORBA

Joint Revised Submission

Alcatel
Hewlett-Packard Company
Highlander Communications, L.C.
INPRISE Corporation
IONA Technologies
Lockheed Martin Federal Systems, Inc.
Lucent Technologies, Inc.
Nortel Networks
Objective Interface Systems, Inc.
Object-Oriented Concepts, Inc.
Sun Microsystems, Inc.
Tri-Pacific Software, Inc.

supported by

France Telecom
Humboldt-University
MITRE Corp.
Motorola, Inc.
University of Rhode Island
Washington University

OMG TC Document orbos/98-10-05

October 18, 1998

Introduction 1

1.1 Cosubmitting Companies

The following companies are pleased to jointly submit this proposal in response to the OMG Realtime CORBA 1.0 RFP (OMG document orbos/97-09-31)

- Alcatel
- Hewlett-Packard Company
- Highlander Communications, L.C.
- INPRISE Corporation
- IONA Technologies
- Lockheed Martin Federal Systems, Inc.
- Lucent Technologies, Inc.
- Nortel Networks
- Objective Interface Systems, Inc.
- Object-Oriented Concepts, Inc.
- Sun Microsystems, Inc.
- Tri-Pacific Software, Inc.

1.2 Proof of Concept

This proposal is the product of the experience of the submitters and supporting organizations in designing realtime distributed systems.

1.3 Submission Contact Points

The editor and primary contact point for this submission is:

Jonathan Currey
Highlander Communications, L.C.
206 East Pine Street
Lakeland, FL 33801
USA
phone: +1 941 686 7767
email: jon@highlander.com

The contact points for the other co-submitting companies are:

Michel Ruffin
Alcatel Alsthom Recherche
Route de Nozay
91460 Marcoussis
France
phone: +33 1 6963 1357
email: Ruffin@aar.alcatel-alsthom.fr

Jishnu Mukerji
Hewlett-Packard New jersey Labs
300 Campus Drive, MS 2E-62
Florham Park, NJ 07932
USA
phone: +1 914 443 7528
email: jis@fpk.hp.com

Jeff Mischkinsky
INPRISE Corporation
951 Mariner's Island Blvd.
Suite 460
San Mateo, CA 94404
USA
phone: +1 650 358-3049
email: jeffm@visigenic.com

Oisin Hurley
IONA Technologies
The IONA Building
Shelbourne Road,
Dublin 4
Ireland
email: ohurley@iona.com

Tom Barker
Lockheed-Martin Federal Systems
Owego
USA
phone: +1 607 751-3794
email: thomas.barker@lmco.com

Judy McGoogan
Lucent Technologies, Inc.
Room 5B-427
2000 N. Naperville Road
Naperville, IL 60566
USA
phone: +1 630 713-7355
email: jmcgoogan@lucent.com

Dave Stringer
Nortel Networks
London Road
Harlow
Essex, CM17 9NA
UK
phone: +44 1279 403712
email: drs@nortel.com

Bill Beckwith
Objective Interface Systems, Inc.
1892 Preston White Drive
Reston, Virginia 20191-5448
USA
phone: +1 703 295 6519
email: bill.beckwith@ois.com

Marc Laukien
Object-Oriented Concepts, Inc
44 Manning Road
Billerica, MA 01821
USA
phone: +1 978 439 92 85
email: ml@ooc.com

Michel Gien
Sun Microsystems
Consumer and Embedded Division
6, avenue Gustave Eiffel
F-78182, Saint-Quentin-en-Yvelines cedex
France
phone: +33 1 39 44 74 22
email: Michel.Gien@sun.com

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

Response to RFP Requirements 2

The following sections list the requirements from the Realtime CORBA 1.0 RFP (OMG orboss/97-09-31) and describe how this submission responds to each of them.

2.1 Mandatory Requirements

- *Extensions to OMG Specifications*

  This proposal does not re-specify existing functionality provided by OMG specifications. Realtime CORBA is therefore specified as an extension to CORBA.

- *Define a "Schedulable Entity"*

  This proposal discusses "activity" as a design concept and uses threads as provided by an underlying OS as a schedulable entity to implement that concept. It also defines an optional Fixed Priority Scheduling Service to help application programmers schedule activities.

- *Interfaces for Priority control of Schedulable Entity*

  This proposal defines a universal, platform-independent priority scheme called CORBA Priority. A CORBA Priority may be associated with the current thread by setting the priority attribute of the RT_CORBA::Current object. A PriorityMapping interface is defined to map the CORBA Priority to/from the native priority scheme of a given scheduler.

- *Mechanism for propagating client priority to the server*

  This proposal defines a ServerPriorityModelPolicy which is used to determine the priority at which a server handles requests from clients. Two models are supported:

  - CLIENT_PRIORITY_PROPAGATION: in which the server honors the priority of the request set by the client, and
  - SERVER_SET_PRIORITY: in which the server handles requests at a set priority.

In both models, the client application's CORBA priority is propagated in a new service context which is passed in the invocation request message. In the CLIENT_PRIORITY_PROPAGATION model, the server ORB will map the CORBA PRIORITY to its local RTOS priority and execute the invocation.

The proposal also defines a PriorityDerivationPolicy which allows the application programmer to choose whether onward invocations from servant application code will be made at either the current base or derived priority of the dispatch thread.

- *Mechanisms for avoiding or bounding priority inversion*

  The mechanism described above for propagating client priority to the server was designed as one tool for application programmers to use to minimize and bound priority inversion in CORBA invocations. Other tools that are specified include:

  - a mutex interface that can be used to coordinate contention for system resources and that allows applications to use the same mutex implementation as the ORB,
  - policies for specifying and configuring communication protocols,
  - a threadpool abstraction used to manage threads of execution on the server side, and
  - policies to be used with the explicit_bind operation (discussed below) that let the client:
    1) set up multiple transport connections - each dedicated to carrying invocations of distinct bands of CORBA priorities; and/or
    2) specify use of non-multiplexed connections.

- *Mechanisms for bounding method invocation blocking*

  The mechanisms described above are also useful for minimizing/bounding method invocation blocking. The Scheduling Service is another mechanism that aids in this.

  In addition, there is an open issue for the submitters to examine the relationship of this proposal to the Messaging Service. As part of that review, they plan to investigate the timeout capability that Messaging specifies.

- *Define "resources" for purposes of resource management*

  For this proposal, resources include: threads, threadpools, transport connections, and request buffers.

- *Mechanisms for management of resource allocation*

  A mutex interface is defined that can be used to coordinate contention for system resources. Management of thread priorities is described above. An API is defined for threadpool management. Transport connections are managed via the use of protocol policies and the explicit_bind operation.

- *Mechanism for client and server side protocol selection*

This proposal provides the ability for an application to associate protocols with a Realtime POA. Any objects activated within that Realtime POA domain may use any protocols which have been associated with that POA. All protocols supported by a Realtime POA will be exported within the Object References. The Client may explicitly select a protocol via an object scope ProtocolPolicy.

- *Interfaces for explicitly setting up and configuring a binding*

  The proposal defines an explicit_bind interface on the client side that provides a connection to the server object prior to the first operation invocation upon that object.

  It also defines optional policies that can be used by this interface for:

  - client-side protocol specification and configuration
  - priority band creation
  - request for the client to have a non-multiplexed connection to the server

- *Refer to POA, rather than the BOA*

  All references within this document refer to the POA rather than the BOA. Indeed many of the ideas are based upon the framework of Policy association and the use of child POAs.

  The use of child POAs constrains policy locality to that portion of a Realtime ORB which requires such tight controls. At the same time, this allows the non-realtime portions of such an ORB to use the ORB services with less specification on behavior.

2.2 Optional Requirements

- *Optionally Specify an interface for client request/reply time-out*

  The submitters plan to examine the relationship of this proposal to the Messaging Service. As part of that review, they plan to investigate the timeout capability that Messaging specifies.

- *Optionally Specify an interface for installation of user-provided transport protocols*

  This proposal does not discuss the interface by which an ORB vendor will provide the ability to substitute a user-provided transport protocol. However, it supports selection and use of such user-provided transport protocols via the selection of protocols using the ProtocolPolicy

- *Optionally Specify a RT Interaction protocol interoperability between RT ORBs*

  Since CORBA Priority is passed in the service context, this specification does not need to define a new protocol for this.

- *Optionally Define run-time interfaces for a "schedulable entity"*

Not addressed.

2.3 Issues to be Discussed

- *Assumptions made about the underlying operating system*

  It is possible for an OS that doesn't implement some or all of the POSIX Real-Time Extensions to support end-to-end predictability, but specifying the required OS features is beyond the scope of this specification.

- *Relationship to POSIX*

  If an OS implements the IEEE POSIX 1003.1-1996 Real-Time Extensions, it has the necessary features to facilitate end-to-end predictability.

- *Relationship to Concurrency Service, Time Service, Transaction Service, and Event Service*

  This proposal is not dependent on these services.There is no restriction in RT_CORBA on invoking these services. However, application programmers should note that these services as currently specified do not include any constraints on their realtime behavior. Thus, using them could impact end-to-end predictability.

- *Relationship to Security Service*

  This proposal is orthogonal to the Security Service.

- *Definition of "binding"*

  The semantics of explicit_bind are discussed.

- *Relationship to Messaging Service*

  There is an open issue for the submitters to examine the relationship of this proposal to the Messaging Service. Specifically, they plan to investigate differences/similarities between this specification's explicit_bind operation and Messaging's set_policy_override and validate_connection operations.

- *How to build Realtime CORBA Applications*

  The chapter on the Fixed Priority Scheduling Service addresses this.

Realtime CORBA 3

3.1 Objectives and Scope of Specification

The goal of this specification is to provide a standard for CORBA ORB implementations that support end-to-end predictability. For the purposes of this specification, "end-to-end predictability" of timeliness in a fixed priority CORBA system is defined to mean:

- respecting thread priorities between client and server for resolving resource contention during the processing of CORBA invocations;
- bounding the duration of thread priority inversions during end-to-end processing;
- bounding the latencies of operation invocations.

A Realtime CORBA system will include the following four major components, each of which must be designed and implemented in such a way as to support end-to-end predictability, if end-to-end predictability is to be achieved in the system as a whole:

1. the scheduling mechanisms in the OS;

2. the Realtime ORB;

3. the communication transport;

4. the application(s).

The scope of this specification is limited to the affect of the Realtime ORB upon end-to-end predictability within the system. In addressing this, requirements are placed upon the other components of the system. These are specified in such a way that they may be satisfied by as wide as possible a variety of implementations.

Nevertheless, satisfying the requirements that Realtime CORBA places upon them will not in itself guarantee that the other components of the system can support end-to-end predictability. If an OS implements the IEEE POSIX 1003.1-1996 Real-Time Extensions, it has the necessary features to facilitate end-to-end predictability. It is

possible for an OS that doesn't implement some or all of the POSIX Real-Time Extensions specification to support end-to-end predictability, but specifying the required OS features is beyond the scope of this specification.

3.2 Realtime CORBA Architecture

3.2.1 RT_CORBA module

Realtime CORBA is specified as an extension to the CORBA Specification. All CORBA IDL specified by Realtime CORBA is contained in a new RT_CORBA module.

3.2.2 RT_CORBA::ORB

An implementation of Realtime CORBA must be capable of producing one or more Realtime CORBA ORB objects. Realtime CORBA ORBs are represented by the RT_CORBA::ORB IDL type, which is derived from CORBA::ORB:

```
//IDL module RT_CORBA {

    interface ORB : CORBA::ORB {

        ...

    };

};
```

Initializing a RT_CORBA::ORB triggers initialization of the Realtime extensions to the ORB interface and makes that ORB instance ready to perform with the behavior specified in the following sections of the Realtime CORBA specification.

Additionally, the interface has a number of operations, that manage the creation and destruction of the other Realtime CORBA IDL interface types, which are defined in sections below.

3.2.3 Realtime CORBA Configuration

The configuration of all RT CORBA features is handled through the CORBA::Policy mechanism. RT CORBA defines a number of new Policy types, instances of which are created using the existing CORBA::ORB::create_policy interface, supplied with the appropriate new PolicyType values.

3.3 Activities and Realtime CORBA

Note – Realtime CORBA does not define a CORBA Activity entity, and hence does not define any IDL for activities. Instead, Realtime CORBA works in terms of threads and invocations made from threads. This leaves applications, and possibly future OMG specifications, free to define and use the activity concept. This section discusses how activities might be supported in terms of the entities used in Realtime CORBA.

An activity is a concept that is sometimes used in the design and implementation of realtime systems, where it might be defined as a sequence of control flow that can traverse across system boundaries. Activities are a useful abstraction for describing distributed priority propagation.

Where activities are defined, the lifetime of an activity may vary according to the needs of the application developer. Typically an activity would start with the beginning of the execution of a thread in some client that then makes invocations. The activity would end when the originating client thread completes. Any time that an existing activity invokes a remote oneway operation a new, temporary activity could be considered to have been created. The creation point of this second activity would be the point at which the thread of the invoking activity is released to continue execution. When exactly this point occurs would depend upon on the synchronization scope policy in place for the oneway operation invocation.

3.4 Thread Scheduling

Realtime CORBA uses threads as a schedulable entity. Generally, a thread represents a sequence of control flow within a single node. In systems that support multiple address spaces, there typically can exist multiple threads per address space. Realtime CORBA specifies interfaces through which the characteristics of a thread that are of interest can be manipulated.

Note – The Realtime CORBA view of a thread is compatible with the POSIX definition of a thread.

3.5 Native Thread Priorities

A realtime operating system (RTOS) sufficient to use for implementing a Realtime ORB compliant with this specification will have some discrete representation of a thread priority. This representation typically specifies a range of priorities and a direction in which the priorities have higher value. The particular range and direction in this priority representation varies from RTOS to RTOS. This specification refers to this RTOS specific thread priority representation as a **native thread priority scheme**. The priority values of this scheme are referred to as **native thread priorities**.

Native thread priorities are used to designate the execution eligibility of threads. The ordering of native thread priorities is such that a thread with higher native priority is executed at the exclusion of any threads in the system with lower native priorities.

A native thread priority is an integer value that is the basis for resolving competing demands of threads for resources. Whenever threads compete for processors or ORB implementation-defined resources, the resources are allocated to the thread with the highest native thread priority value.

The **base native thread priority** of a thread is defined as the native priority with which it was created, or to which it was later set. The initial value of a thread's base native priority is dependent on the semantics of the specific operating environment. Hence it is implementation specific.

At all times, a thread also has a **derived native thread priority**, which is the result of considering its base native thread priority together with any priorities it inherits from other threads. At any time, the derived native thread priority of a thread is the maximum of all the priorities the thread is inheriting at that instant. For a thread that is not suspended, its base native thread priority is always a source of priority inheritance.

Priority inheritance is the term used for this process by which the native thread priority of other threads is used in the evaluation of a thread's derived native thread priority. A **priority inheritance protocol** must be used by a conforming Realtime CORBA ORB to implement the execution semantics of threads and mutexes. It is an implementation issue as to whether the Realtime ORB implements simple priority inheritance, immediate ceiling locking protocol, original ceiling locking protocol or some other priority inheritance protocol.

Whichever priority inheritance protocol is used, the native thread priority ceases to be inherited as soon as the condition calling for the inheritance no longer exists. At the point when a thread stops inheriting a native thread priority from another source, its derived native thread priority is re-evaluated.

The thread's derived native priority is used when the thread competes for processors. Similarly, the thread's derived priority is used to determine the thread's position in any queue (i.e., dequeuing occurs in native thread priority order).

Native priorities have an IDL representation in Realtime CORBA, which is of type short :

```
module RT_CORBA {

   typedef short NativePriority;

};
```

This means that native priorities must be integer values in the range -32768 to +32767. However, for a particular RTOS, the valid range will be a sub-range of this range.

The NativePriority type is used in defining mappings between native and CORBA priority, as described in the section on CORBA Priority Mappings, below.

Realtime CORBA does not support the direct use native priorities : instead, the application programmer uses CORBA Priorities, which are defined in the next section. However, applications will still use native priorities where they make direct use of RTOS features.

3.6 CORBA Priority

Realtime CORBA defines a universal, platform independent priority scheme called **CORBA Priority**. It is introduced to overcome the heterogeneity of different native priority schemes, and allows Realtime CORBA applications to make prioritized CORBA invocations in a consistent fashion between nodes with different native priority schemes.

For consistency, Realtime CORBA applications always use CORBA Priority to express the priorities in the system, even if all nodes in a system use the same native thread priority scheme, or when using the alternate, server-set priority model.

A RT_CORBA::Priority type is defined:

```
//IDL
module RT_CORBA {

   typedef short Priority;
   const Priority minPriority =     0;
   const Priority maxPriority = 32767;

};
```

A signed short is used in order to accommodate the Java language mapping. However, only values in the range 0 (minPriority) to 32767 (maxPriority) are valid. Numerically higher CORBA Priority values are defined to be of higher priority.

A CORBA Priority may be associated with the current thread, by setting the priority attribute of the RT_CORBA::Current object:

```
//IDL
module RT_CORBA {

   interface Current : CORBA::Current {
      attribute RT_CORBA::Priority priority;
   };

};
```

A CORBA system exception is thrown if an attempt is made to set the priority to a value outside the range 0 to 32767.

Upon setting this attribute, the CORBA Priority value is mapped to a native priority value and the native priority of the current thread is immediately set to that value. bCORBA Priority mappings are described in the next section.

Once a thread has a CORBA Priority value associated with it, the behaviour when it makes an invocation upon a CORBA Object depends on which value of the ServerPriorityModelPoloicy that CORBA Object supports:

3.6.1 Client Priority Propagation Model

If the object that is invoked upon supports the CLIENT_PRIORITY_PROPAGATION value of the ServerPriorityModelPolicy, the CORBA Priority is carried with the CORBA invocation and is used to ensure that all threads subsequently executing on behalf of the invocation (on client or server) run at the appropriate priority. The propagated CORBA Priority becomes the CORBA Priority of any such threads, and the threads run at a native priority mapped from that CORBA Priority.

The CORBA Priority is propagated in a CORBA Priority service context which is passed in the invocation request message.

```
module IOP {

    const ServiceId      CorbaPriority = ??;
                         // <number to be assigned by OMG>

};
```

The context_data contains the RT_CORBA::Priority value as a CDR encapsulation of a short type.

Note – The CorbaPriority const should be added to a future version of GIOP.

The thread that dispatches the invocation (i.e. runs the servant code) initially has the CORBA Priority of the invoking thread. Therefore if, as part of the processing of this request it makes CORBA invocations to other objects, these onward invocations will be made with the same CORBA Priority. If the CORBA Priority of the dispatch thread is changed by the application, any subsequent onward invocations will be made with this new priority.

The above scenario does not consider the effect of priority inheritance. Its possible effect on the propagated CORBA Priority is discussed in the sections on the CORBA Mutex interface and the PriorityDerivationPolicy, below.

3.6.2 *Server-Set Priority Model*

If the Object that is being invoked upon supports the SERVER_SET_PRIORITY value of the ServerPriorityModelPolicy, then in the same way as for the client priority propagation model, any threads on the client side that subsequently run on behalf of the invocation are run at a native priority mapped from the CORBA Priority, and the CORBA Priority value is passed with the invocation, in a service context.

Issue – Under the Server-Set priority model, the submitters have identified circumstances in which it would be desirable not to send the CORBA Priority in a service context, but also other circumstances in which it is desirable. A third server priority model may be required.

However, the propagated CORBA Priority it is not used to determine the priority of threads on the server-side running on behalf of that invocation. Instead, server-side threads running on behalf of the invocation run at a native priority mapped from the CORBA Priority associated with that CORBA Object, which is given in the server_priority attribute of the ServerPriorityModelPolicy used at its creation.

If as part of the processing of the request, the servant code makes CORBA invocations to other objects, these onward invocations will be made with the CORBA Priority of the server. If the CORBA Priority of the dispatch thread is changed by the application, any subsequent onward invocations will be made with this new priority.

This scenario does not consider the effect of priority inheritance. Its possible effect on the priorities, including the CORBA Priority that is used to make onward calls from servant code, is considered in the sections on the CORBA Mutex interface and the PriorityDerivationPolicy, below.

Issue – Whether CORBA Priority values may be returned to the caller in a reply message service context is still being investigated. The following choices all have merits : a value must always be returned; must always be returned if the value has changed; may be returned (by a particular implementation); may not be returned.

3.7 CORBA Priority Mappings

Priority values specified in terms of the CORBA Priority scheme must be mapped into the native priority scheme of a given scheduler before they can be applied to the underlying schedulable entities. On occasion, it is necessary for the reverse mapping to be performed, to obtain a CORBA Priority to represent the present native priority of a thread. The latter can occur, for example, when priority inheritance is in use, or when wishing to introduce an already running thread into a CORBA system at its present (native) priority.

To allow the Realtime ORB and applications to do both of these things, Realtime CORBA defines a PriorityMapping interface:

```
//IDL
module RT_CORBA {

   // Locality Constrained interface
    interface PriorityMapping    {

       boolean to_native (in Priority      corba_priority,
                          out NativePriority native_priority);

       boolean to_CORBA  (in NativePriority native_priority,
                          out Priority corba_priority);

   };

};
```

Only one PriorityMapping object is active (or "installed") at any one time, per ORB instance. Conformant Realtime CORBA implementations must provide a "default PriorityMapping", which is installed by default. However, the particular mappings that the default provides are an implementation issue. Applications may install their own PriorityMapping object. PriorityMapping installation is explained in the next section.

The priority mappings between native and CORBA priority are defined by the implementations of the to_native and to_CORBA operations of a PriorityMapping object. The to_native operation accepts a CORBA Priority value an an in parameter and maps it to a native priority, which is given back as an out parameter. Conversely, to_CORBA accepts a NativePriority value as an in parameter and maps it to a CORBA Priority value, which is again given back as an out parameter.

As the mappings are used by the ORB, and may be used more than once in the normal execution of an invocation, their implementations should be as efficient as possible. For this reason, the mapping operations may not raise any CORBA exceptions, including system exceptions. The ORB is not restricted from making calls to the to_native and/or to_CORBA operations from multiple threads simultaneously. Thus, the implementations should be re-entrant.

Rather than raising a CORBA exception upon failure, a boolean return value is used to indicate mapping failure or success. If the priority passed in can be mapped to a priority in the target priority scheme, TRUE is returned and the value is returned as the out parameter. If it cannot be mapped, FALSE is returned and the value of the out parameter is undefined.

to_native and to_CORBA must both return FALSE when passed a priority that is outside of the valid priority range of the input priority scheme. For to_native this means when it is passed a short value outside of the CORBA Priority range, 0-32767 (i.e. a negative value.) For to_CORBA this means when it is passed a short value outside of the native priority range used on that system. This range will be implementation specific.

Neither to_native nor to_CORBA is obliged to map all valid values of the input priority scheme (the CORBA Priority scheme or the native priority scheme, respectively.) This allows mappings to be produced that do not use all values of the native priority scheme of a particular scheduler and/or that do not use all values of the CORBA Priority scheme.

The mappings do not have to be idempotent : they are not obliged to yield the same output value every time they are given a particular input value. However, they should be idempotent to produce a reasonably schedulable system.

When the ORB receives a FALSE return value from a mapping operation that is called as part of the processing of a CORBA invocation, processing of the invocation proceeds no further, and if possible a system exception is raised to the application making the invocation. Note that it may not be possible to raise an exception to the application if the failure occurs on a call to a mapping operation made on the server side of an oneway invocation.

Issue – Particular system exceptions have not yet been assigned in this or other places where Realtime CORBA may raise a system exception.

3.7.1 Installation of CORBA Priority Mappings

The realtime ORB, RT_CORBA::ORB, provides an operation for the installation of a new PriorityMapping:

```
// IDL
module RT_CORBA {

  // Locality Constrained interface
  interface ORB : CORBA::ORB    {

    ...

      void install_priority_mapping (in PriorityMapping pm);

    ...

  };

};
```

Only one PriorityMapping may be installed at any one time, so installing a new one replaces the one that was previously installed.

To create a consistently schedulable system, a new priority mapping should only be installed in the interval between the initialization of the realtime ORB and making first use of CORBA Priority. However, because of the potential overhead incurred by tracking whether a previously installed PriorityMapping has been used yet, later installation is not trapped by the install_priority_mapping operation.

3.8 *Mutex interface*

The Mutex interface provides the mechanism for coordinating contention for system resources.

Realtime CORBA specifies a RT_CORBA::Mutex locality constrained interface, so that applications can use the same mutex implementation as the ORB.

```
//IDL
module RT_CORBA {

   // locality constrained interface
    interface Mutex     {

       void lock();

       void unlock();

       boolean try_lock(in TimeBase::TimeT max_wait);
          // if max_wait = 0 then return immediately

    };

   interface ORB : CORBA::ORB     {

     ...
       Mutex create_mutex();
     ...

   };

};
```

A new RT_CORBA::Mutex object is obtained using the create_mutex() operation of RT_CORBA::ORB.

A Mutex object has two states: locked and unlocked. Mutex objects are born in the unlocked state. When the Mutex object is in the unlocked state the first thread to call the lock() operation will cause the Mutex object to change to the locked state. Subsequent threads that call the lock() operation while the Mutex object is still in the locked state will block until the owner thread unlocks it by calling the unlock() operation. Implementations must ensure that the lock operations are atomic in the presence of multiple processors if the system has multiple processors.

The try_lock() operation works like the lock() operation except that if it does not get the lock within max_wait time it returns FALSE. If the try_lock() operation does get the lock within the max_wait time period it returns TRUE.

A conforming ORB implementation must provide a implementation of Mutex that implements some form of priority inheritance protocol. This may include, but is not limited to, simple priority inheritance or a form of priority ceiling locking protocol. The mutex returned by create_mutex must have the same priority inheritance properties as those used by the ORB to protect resources.

If an ORB implementation offers a choice of priority inheritance protocols, or offers a protocol that requires configuration, selection or configuration will be controlled through an implementation specific interface.

While a thread executes in a region protected by a mutex object, it can be preempted only by threads whose derived native thread priorities are higher than either the ceiling or derived (inherited) priority of the mutex object.

The effect of priority protocols on the execution of Realtime CORBA application code is handled through the PriorityDerivationPolicy policy object, which is described in the server-side configuration section, below.

3.9 Server-side Configuration

New policies are defined, to cover the configuration of the following server-side RT CORBA features :

- protocol selection
- protocol configuration
- server-side thread configuration (through Threadpools)
- server priority model (inherited from client v. set by server)
- handling of priority derivation resulting from use of priority inheritance protocols on mutexes.

Which of the CORBA policy application points (ORB, POA, Current) a given policy may be applied at is given along with the description of each policy, below. An attempt to apply a policy at an inappropriate level will lead to a WrongPolicy exception being raised.

3.9.1 ProtocolPolicy

The ProtocolPolicy policy type is used to configure the selection and configuration of communication protocols in RT CORBA.

```
// IDL module RT_CORBA {

   // Locality Constrained interface
   interface ProtocolProperties {};

   struct Protocol {
       IOP::ProfileId     protocol_type;
       ProtocolProperties orb_protocol_properties;
       ProtocolProperties transport_protocol_properties;
    };

   typedef sequence <Protocol> ProtocolList;

   // Protocol Policy
    const CORBA::PolicyType PROTOCOL_POLICY_TYPE = ??;

    // Locality Constrained interface
    interface ProtocolPolicy : CORBA::Policy     {
       readonly attribute ProtocolList protocols;

     };

 };
```

A ProtocolPolicy allows any number of protocols to be specified and, optionally, configured at the same time. The order of the Protocols in the ProtocolList indicates the order of preference for the use of the different protocols. Information regarding the protocols must be placed into IORs in that order, and the client should take that order as the default order of preference for choice of protocol to bind to the object via.

The type of protocol is indicated by an IOP::ProfileId (from the specification of the CORBA IOR), which is an unsigned long. This means that a protocol is defined as a specific pairing of an ORB protocol (such as GIOP) and a transport protocol (such as TCP.) Hence IIOP would be selected, rather than GIOP plus TCP being selected separately. IIOP in particular is represented by the value TAG_INTERNET_IIOP (or the value 0, that it is defined as.)

A Protocol type contains a ProfileId plus two ProtocolProperties, one each for the ORB protocol and the transport protocol.

The properties are provided to allow the configuration of protocol specific configurable parameters. Specific protocols have their own protocol configuration interface that inherits from the RT_CORBA::ProtocolProperties interface. A nil reference for either ProtocolProperties indicates that the default configuration for that protocol should be used. (Each protocol will have an implementation specific default configuration, that may be overridden by applying the protocol policy at ORB scope. See policy scope, below.)

```
//IDL
module RT_CORBA {
    interface TCPProtocolProperties : ProtocolProperties    {
        attribute long     send_buffer_size;
         attribute long     recv_buffer_size;
         attribute boolean  keep_alive;
         attribute boolean  dont_route;
         attribute boolean  no_delay;
    };
};
```

TCP is the only protocol that RT CORBA specifies a ProtocolProperties interface for. A similar interface is not specified for GIOP, as GIOP has no configurable properties.

ProtocolProperties should be defined for any other protocols useable with an RT CORBA implementation, but unless they are standardized in an OMG specification their name and contents will be implementation specific. ProtocolProperties for other protocols may be standardized in the future, and a ProtocolProperties interface should be specified in the standardization of any other protocol, if it is to be useable in a portable way with RT CORBA.

Scope of ProtocolPolicy Policy

Applying a ProtocolPolicy to the creation of a POA controls the protocols that references created by that POA will support (and their configuration if non- nil ProtocolProperties are given.) If no ProtocolPolicy is given at POA creation, the POA will support the default protocols associated with the ORB that created it. (Note that supplying a ProtocolPolicy overrides, rather than supplementing or sub-setting, the default selection of protocols associated with the ORB.)

The ORB's default protocols, and their order of preference, are implementation specific. The default may be overridden by applying a ProtocolPolicy at the ORB level. As a consequence, portable applications must override all defaults to ensure the same behavior between ORB implementations.

Only one ProtocolPolicy should be included in a given PolicyList, and including more than one will result in a CORBA system exception being raised.

Protocol Configuration Semantics

Note that the above API only allows policies to be set at POA creation time. No API is proposed to allow (re)configuration of any policy after POA creation.

The protocol configuration selected at the time of POA creation is used to determine the server-side configuration that is to be used by the protocol in question for all connections from clients to objects that have references created by that POA.

However, as the configuration semantics of a protocol (such as whether a particular property can be configured on a per-connection basis or only globally for that instance of the protocol) are protocol specific, the exact semantics of protocol configuration via ProtocolProperties are not specified by RT CORBA, and must be specified on a per-protocol basis.

If a protocol offers a configurable property that can only be configured at some scope wider than that of the individual POA (say at the scope of the ORB instance), it can choose either to:

- change that property at the wider scope when a different value is requested for the creation of a new POA. This will ensure that the new POA gets the configuration requested, but will also affect the configuration of new and possibly existing connections made to other CORBA Objects via the same protocol. The exact scope and semantics of the property change must be given as part of the documentation of the ProtocolProperties interface for that protocol.
- not change the property, but instead raise an InvalidPolicy exception and fail to create the new POA. In this way, the original value of the property is preserved for the existing references that use it. Once again, this behaviour must be covered in the documentation of the ProtocolProperties interface for that protocol.

Which of the two strategies a protocol uses is an implementation issue.

3.9.2 Threadpool Policy

A threadpool abstraction is used to manage threads of execution on the server- side of the RT CORBA ORB.

Threadpools offer the following features:

- preallocation of threads. This helps reduce priority inversion, by allowing the application programmer to ensure that there are enough thread resources to satisfy a certain number of concurrent invocations, and also helps reduce latency and increase predictability, by avoiding the destruction and recreation of threads between invocations.
- partitioning of threads. Having multiple thread pools, associated with different POAs allows one part of the system to be isolated from the thread usage of another, possibly lower priority, part of the application system. This can again be used to reduce priority inversion.
- bounding of thread usage. A threadpool can be used to set a maximum limit on the number of threads that a POA or set of POAs may use. In systems where the total number of threads that may be used is constrained, this can be used in conjunction with threadpool partitioning to avoid priority inversion by thread starvation.

Threadpools are managed using operations on the Realtime ORB:

```
//IDL
module RT_CORBA {

   // Threadpool types
   typedef unsigned long ThreadpoolId;

   struct ThreadpoolLane {
       Priority      lane_priority;
       unsigned long  static_threads;
       unsigned long  max_threads;
    };

   typedef sequence <ThreadpoolLane> ThreadpoolLanes;

   // Threadpool Policy
   const CORBA::PolicyType THREADPOOL_POLICY_TYPE = ??;

   interface ThreadpoolPolicy : CORBA::Policy     {

       readonly attribute ThreadpoolId threadpool;

   };

   interface ORB : CORBA::ORB {

       ...

      exception InvalidThreadpool {};

      ThreadpoolId create_threadpool ( in unsigned long  stacksize,
                                       in unsigned long  static_threads,
                                       in unsigned long  max_threads,
                                       in Priority           default_priority );

      ThreadpoolId create_threadpool_with_lanes (
                                       in unsigned long   stacksize,
                                       in ThreadpoolLanes lanes,
                                       in boolean allow_borrowing );

      void destroy_threadpool ( in ThreadpoolId threadpool )
            raises (InvalidThreadpool);

        ...

     };

};
```

The create_threadpool and create_threadpool_with_lanes operations allow two different styles of threadpool to be created : with or without lanes.

In both cases, the stacksize parameter is used to specify the stack size, in bytes, that each thread must have allocated.

To create a threadpool without lanes the following parameters must also be specified:

- static_threads, which specifies the number of threads that will be pre-created and assigned to that threadpool at the time of its creation. An exception is raised if this number of threads cannot be created, in which case no threads are created and no threadpool is created.
- max_threads, which specifies the maximum number of threads that the threadpool may hold. If this is a value greater than static_threads, additional threads will be created dynamically, individually and upon demand, when the static threads are all in use and an additional thread is required to service an invocation. Whether a dynamically created thread is destroyed as soon as it is not in use, or is retained forever or until some condition is met is an implementation issue.

If max_threads is the same as static_threads no additional threads may be dynamically created, and only the static threads are available. In either case, once the maximum number of threads has been reached, no additional threads will be added to the threadpool, and any additional invocations will block waiting for one of the existing threads to become available.

If max_threads is zero, no limit is placed on the number of threads that the threadpool may grow to hold. max_threads may not have a non-zero value less than min_threads, and attempting to create a threadpool with such a value will result in the create_threadpool operation failing with an exception.

- default_priority, which specifies the CORBA priority that the static threads will be created with. (Dynamic threads may be created directly at the priority they are required to run at to handle the invocation they were created to handle.)

To create a threadpool with lanes, a lanes parameter must be configured, instead of the static_threads, max_threads and default_priority parameters. The lanes specifies a number of ThreadpoolLanes, each of which must have the following parameters specified :

- lane_priority, which specifies the CORBA Priority that all threads in this lane (both static, and dynamically allocated ones) will run at.
- static_threads, which specifies the number of threads that will be pre-created, but in this case allocated to this specific lane, rather than the pool as a whole.
- max_threads, which specifies the maximum number of threads that may be allocated to this lane. The relationship between the value of max_threads and static_threads is the same as in the case of threadpools without lanes : it determines whether and if so how many additional threads may be dynamically created (but in this case the dynamic thread are specific to this lane and are created with the CORBA Priority specified by lane_priority.)

Additionally, to create a threadpool with lanes, the allow_borrowing boolean parameter must be configured to indicate whether the borrowing of threads by one lane from a lower priority lane is permitted or not.

If thread borrowing is permitted, when a lane of a given priority exhausts its maximum number of threads and requires an additional thread to service an additional invocation, it may "borrow" a thread from a lane with a lower priority. The borrowed thread has its CORBA Priority raised to that of the lane that requires it. When the thread is no longer required, its priority is lowered once again to its previous value, and it is returned to the lower priority lane. The thread will be borrowed from the highest priority lane with threads available. If no lower priority lanes have threads available, the lane wishing to borrow a thread must wait until one becomes free (which will quite possibly be one of its own.)

More generally, for both threadpools with and without lanes, if the priority of a thread is changed whilst dispatching an invocation, it is restored to its original priority before returning it to the threadpool.

When a threadpool is successfully created, using either method, a ThreadpoolId identifier is returned. This can later be passed to destroy_threadpool to destroy the threadpool. If a threadpool cannot be created because the parameters passed in do not specify a valid threadpool configuration, a CORBA system exception is raised. If a threadpool cannot be created because there are insufficient operating system resources, a system exception is raised.

The same threadpool may be associated with a number of different POAs, by using a ThreadpoolPolicy containing the same ThreadpoolId in each POA_create.

Scope of ThreadpoolPolicy:

The ThreadpoolPolicy may be applied at the POA and ORB level. A POA may only be associated with one threadpool, hence only one ThreadpoolPolicy should be included in the PolicyList specified at POA creation.

A ThreadpoolPolicy may be applied at the ORB level, where it assigns the indicated threadpool as the default threadpool to use in the subsequent creation of POAs, until the default is again changed. The default is used if a ThreadpoolPolicy is not specified in the polices used at the time of POA creation.

3.9.3 Server Priority Model Policy

The overall goal of Real-Time CORBA is to minimize and bound priority inversion in CORBA invocations. One mechanism that is employed to achieve this is propagation of the activity priority from the client the server, with the requirement that the server side ORB make the up-call at this priority (subject to any priority inheritance protocols that are in use.)

However, in some scenarios, it is sufficient to design the application system by setting the priority of servers, and having them handle all invocations at that priority.

Hence, RT CORBA supports two models for the priority at which a server handles requests from clients, which are selected by use of the provided ServerPriorityModelPolicy interface :

```
//IDL
module RT_CORBA {

   // Server Priority Model Policy
   const CORBA::PolicyType
                    SERVER_PRIORITY_MODEL_POLICY_TYPE = ??;

   enum ServerPriorityModel    {

      CLIENT_PRIORTY_PROPAGATION, SERVER_SET_PRIORITY
    };

    interface ServerPriorityModelPolicy : CORBA::Policy    {

        readonly attribute ServerPriorityModel server_priority_model;
        readonly attribute Priority server_priority;

    };

};
```

- CLIENT_PRIORTY_PROPAGATION: in which the server honours the priority of the request, set by the client. Requests from non-RT CORBA ORBs (i.e. ORB's that do not propagate a CORBA Priority in the request's service contexts) are handled at the priority specified by the server_priority attribute of the policy.

The Client application's CORBA priority (set via the RT_CORBA::Current priority attribute) will be propagated to the server ORB in the service context of IIOP messages.

The server ORB will use this CORBA PRIORITY and map it with the to_native mapping operation to its local RTOS priority and execute the invocation. We will call this Client assigned priority model.

- SERVER_SET_PRIORITY: in which the server handles requests at a set priority, which is configured by the server_priority attribute of the policy.

In this model the server side processing of an invocation will use a server specified CORBA priority to perform processing of client invocations.

3.9.4 Priority Derivation Policy

Realtime CORBA offers the application programmer the choice of onward invocations from servant application code being made at either the current base or derived priority of the dispatch thread. The choice is made using the PriorityDerivationPolicy:

```
//IDL
module RT_CORBA {

   // Priority Derivation Policy
   const CORBA::PolicyType PRIORITY_DERIVATION_POLICY_TYPE = ??;

   enum PriorityDerivationPolicy    {

      USE_BASE_PRIORITY, USE_DERIVED_PRIORITY

   };

   interface PriorityDerivationPolicy : CORBA::Policy    {

      readonly attribute PriorityDerivationPolicy derivation_policy;

   };

};
```

If the USE_BASE_PRIORITY value is selected, the base CORBA Priority of the dispatch thread at the time of the further invocation is used as the CORBA Priority for that invocation.

If the USE_DERIVED_PRIORITY value is selected, the derived CORBA Priority of the dispatch thread at the time of the further invocation is used as the CORBA Priority for that invocation. It is an implementation issue whether or not arriving at the derived CORBA Priority involves mapping from a native priority, using the to_CORBA priority mapping operation.

Note that the priority attribute of RT_CORBA::Current always reflects the base, rather than derived, priority of the current thread.

3.10 Client-side Configuration

3.10.1 Explicit Binding

Issue – The explicit_bind operation has been specified by considering the requirements of Realtime CORBA in isolation. The relationship to the Messaging specification - and in particular the set_policy_overrides and validate_connection operations - is currently being studied by the submitters.

Note that the requirements outlined in the issue on the PriorityBandedConnectionsPolicy, below, are relevant to this, as the submitters believe this requires functionality beyond that specified by Messaging.

Explicit binding offers the following features :

- connection to server object prior to the first operation invocation upon that object, similar to the Messaging Service's validate_connection.
- optional client side protocol specification and configuration.
- optional priority band creation
- optional request for the client to have a non-multiplexed connection to the server.

The following IDL is defined for explicit binding:

```
//IDL
module RT_CORBA {

   // Locality Constrained interface
   interface ORB : CORBA::ORB     {

      ...

      exception WrongPolicy {};

      Object explicit_bind (in Object     o,
                            in CORBA::PolicyList policies)
         raises (WrongPolicy);
      ...

   };

};
```

A new object reference is returned, and the existing object reference passed in is still valid for application use, or destruction. This respects the immutability of object references.

The following client-side policies are used to provide the features outlined above:

3.10.2 ProtocolPolicy (Client Side)

The ProtocolPolicy policy defined for server side configuration is also applicable on the client side.

When applied to an explicit bind, the ProtocolList indicates the protocols that may be used to make a connection to the specified object, in order of preference. If the ORB fails to make a connection because none of the protocols is available on the client ORB, a CORBA system exception is raised. If one or more of the protocols is available, but the ORB still fails to make a connection a CORBA system exception is raised. In both cases no binding is made.

If it is necessary to know which protocol a binding was successfully made via, a single protocol should be passed into each of a succession of explicit binds until one of them is successful.

If no BindingProtocolPolicy is provided, then the protocol selection is made by the ORB based on the target object's available protocols, as described in its IOR, and the protocols supported by the client ORB.

3.10.3 PriorityBandedConnectionsPolicy

To reduce priority inversion due to use of a non-priority respecting transport protocol, RT CORBA provides the facility for a client to communicate with a server via multiple connections, with each connection handling invocations that are made at a different CORBA priority or range of CORBA priorities. The selection of the appropriate connection is transparent to the application, which uses a single object reference as normal.

The PriorityBandedConnectionsPolicy is defined thus:

```
//IDL
module RT_CORBA {

   struct PriorityBand    {
      Priority low;
      Priority high;
   }

   typedef sequence <PriorityBand> PriorityBands;

   // PriorityBandedConnectionPolicy
   const CORBA::PolicyType
            PRIORITY_BANDED_CONNECTIONS_POLICY_TYPE = ??;

   interface PriorityBandedConnectionPolicy : CORBA::Policy    {

      readonly attribute PriorityBands priority_bands;

    };

};
```

The PriorityBands attribute of the policy may be assigned any number of PriorityBands. PriorityBands that cover a single priority (by having the same priority for their low and high values) may be mixed with those covering ranges of priorities. No priority may be covered more than once. The complete set of priorities covered by the bands do not have to form one contiguous range, nor do they have to cover all CORBA Priorities.

Once the binding has been successfully made, an attempt to make an invocation with a CORBA Priority which is not covered by one of the bands will fail, with a CORBA system exception. Hence, a policy specifying only one band can be used to restrict a client's invocations to a range of priorities

If no bands are provided, then a single connection will be established.

Issue – No mechanism is specified for the banding information to be communicated from the client to the server. Whilst implementations are possible that do not require the banding information to be propagated, the submitters are considering specifying a protocol for the communication of the banding information from the client to the server at the time of band-connection establishment. One solution being considered is an implicit operation (similar to is_a.)

Note that this issue is being considered in the study of the relationship between Realtime CORBA and the Messaging specification.

3.10.4 PrivateConnectionPolicy

This policy allows a client to obtain a private transport connection which will not be multiplexed (shared) with other client-server object connections.

```
//IDL
module RT_CORBA {

   // Private Connection Policy

    const CORBA::PolicyType PRIVATE_CONNECTION_POLICY_TYPE = ??;

    interface PrivateConnectionPolicy : CORBA::Policy {};

};
```

Note that it is not possible to explicitly request a multiplexed connection. Whether multiplexing is appropriate or not is a protocol specific issue, and hence an ORB implementation issue. By not requesting a private connection the application indicates to the ORB that a multiplexed connection would be acceptable. It is up to the ORB implementation to make use of this indication.

3.11 Request Buffers

Issue – This topic is one that the submitters are still examining, to understand if, and if so how, it should be specified. The text below explains the problem being addressed.

Just as it is necessary to provide control over processing resources, i.e. threadpools, so it assists developers if control can be exercised over storage resources. In particular the storage resources directly associated with the passage of an actviity through an ORB.

These resources are the buffers in which requests may be held prior to being given to a thread but after being received from a communications end-point. The characteristics of systems to which Real-time CORBA will be applied can vary greatly. Some systems will be sensitive to response times and hence the latency implied by queues (even prioritized queues) would not be a natural choice. Other systems will be sensitive to throughput. Such systems may wish to trade-off latency for a better utilization of resources.

For a system where the frequency of arrival of requests is statistical rather than precise, it is inevitable that load will be uneven, with peaks and troughs. Often it is not practical to allocate a thread to every request as soon as it is received form the end-point and then leave it to the RTOS's scheduler to manage the requests. Such a policy would be too profligate with relatively expensive thread resources.

Neither is it satisfactory to adopt a "lazy consumer" policy with respect to the end-point. Leaving the storage of requests as the sole responsibility of the end-point forfeits the chance to share storage resources across multiple end-points. Only by sharing resources can some systems deal with a distribution of load on those end-points, that varies with time, in a predictable fashion.

Treating request buffers as a manageable resource provides a developer with control over the handling of an activity following a message being received from an end-point and prior to a thread being allocated.

Realtime CORBA Scheduling Service 4

4.1 Introduction

This section describes the Realtime CORBA Scheduling Service. The Scheduling Service uses the primitives of the Realtime ORB to facilitate enforcing various fixed-priority realtime scheduling policies across the Realtime CORBA system in a way that abstracts away from the application some of the low-level realtime constructs. The Scheduling Service does not impose any new requirements on Realtime or non-Realtime ORBs beyond what appears in the RT CORBA specification or CORBA specification respectively.

The primitives added in Realtime CORBA to create a Realtime ORB are sufficient to achieve realtime scheduling, but effective realtime scheduling is complicated. For applications to ensure that their execution is scheduled according to a uniform policy, such as global Rate Monotonic Scheduling, requires that the RT ORB primitives be used properly and that their parameters be set properly in all parts of the CORBA system.

Not only is determining the proper use and correct parameters difficult, but once it is done, the application code becomes substantially more complex - making analysis and modification very difficult. The Scheduling Service specified in this section addresses these problems because an instance of the Scheduling Service embodies a uniform scheduling policy, and because the simple Scheduling Service interface abstracts away much of the complexity from application code.

An application that uses an implementation of the Scheduling Service is assured of having a uniform realtime scheduling policy, such as global rate-monotonic scheduling with priority ceiling, enforced in the entire system. That is, a Scheduling Service implementation will choose CORBA priorities, POA policies, and priority mappings in such a way to realize a uniform realtime scheduling policy. Different implementations of the Scheduling Service can provide different realtime scheduling policies.

The Scheduling Service abstraction of scheduling parameters (such as CORBA Priorities) is through the use of "names". The application code uses names (strings) to specify CORBA Activities and CORBA objects. The Scheduling Service internally associates those names with scheduling parameters and policies for the named Activity or the named CORBA object. This abstraction improves portability with regard to realtime features, eases uses of the realtime features, and reduces the chance for errors.

Each name used by the Scheduling Service method invocations must be unique. The Scheduling Service is designed to work in a "closed" CORBA system where fixed priorities are needed for a static set of clients and servers. Therefore, it is assumed that the system designer has identified a static set of CORBA Activities, the CORBA objects that the Activities use, and has determined scheduling parameters, such as CORBA priorities, for those Activities and objects. In that process, names are uniquely assigned to those Activities and Objects and the names are associated to scheduling parameters. This association of names to scheduling parameters is then used to configure the Scheduling Service.

The capabilities provided by the Scheduling Service are not orthogonal to the primitives provided by the Realtime ORB. In fact, most of the capabilities provided by the Scheduling Service are expected to be implemented by the Scheduling Service invoking the Realtime CORBA primitives in a way that ensures a uniform realtime scheduling policy is enforced.

4.2 IDL

```
module RTScheduling {

   exception UnknownName {};

   // locality constrained interface
   interface ClientScheduler {

     void schedule_activity(in string name)
        raises(UnknownName);

  };

   // locality constrained interface
   interface ServerScheduler {

      PortableServer::POA create_POA (
                     in PortableServer::POA parent,
                     in string adapter_name,
                     in PortableServer::POAManager a_POAManager,
                     in CORBA::PolicyList policies)
            raises ( PortableServer::POA::AdapterAlreadyExists,
                     PortableServer::POA::InvalidPolicy );

   void schedule_object(in Object obj, in string name)
      raises(UnknownName);

   };

};
```

4.3 Semantics

A CORBA client obtains a local reference to a ClientScheduler object. Whenever the client begins a region of code with a new deadline or priority (indicating a new CORBA Activity), it invokes "schedule_activity" with the name of the new activity. The Scheduling Service associates a CORBA priority with this name (assuming the name is valid--otherwise an exception is thrown), and it invokes appropriate RT ORB and RTOS primitives to schedule this activity.

The "create_POA" method accepts parameters allowing it to create a POA. This POA will enforce all of the non-realtime policies in the Policy List input parameter. All realtime policies for the returned POA will be set internally by this scheduling service method. This ensures a selection of realtime policies that is consistent with the scheduling policy being enforced by the Scheduling Service implementation. The Scheduling Service implementation should clearly document what POA RT policies it will use under various conditions.

"Schedule_object" is provided to allow the Scheduling Service to achieve object-level control over scheduling of the object. RT POA policies in the RT ORB allow some control over the scheduling of object invocations, but must do so for all objects managed by each POA. Some realtime scheduling, such as priority ceiling concurrency control, requires object-level scheduling. The "schedule_object" call will install object-level scheduling with scheduling parameters, such as, for example, the priority ceiling for the object. These scheduling parameters are derived internally by the Scheduling Service using the name passed into the call.

4.4 Example

Assume a CORBA object with "method1" and "method2". A client wishes to call method1 under one deadline and method2 under a different deadline. Here is sketch psuedocode of what the client and server (main) would look like with the Scheduling Service.

Step 0

Assume that at system startup an implementation of the Scheduling Service is started and that Scheduling Service instance installs a mapping object using the RT ORB install_priority_mapping call.

Client

```
1 RTScheduling::ClientScheduler sched;

2 obj = bind to server object

3 sched->schedule_activity ("activity1");

4 obj->method1( params );

5 sched->schedule_activity ("activity2");

6 obj->method2(params );
```

Server Main

```
1 RTScheduler::ServerScheduler sched;

2 PortableServer::POA poa1;

3 PolList = make a policy list of non-RT policies for a POA

4 poa1 = sched->create_POA(parent_poa, "adapter1", a_POAManager, PolList);

5 obj = poa1->creat_object ( params );

6 sched->schedule_object(obj, "Object1" );

...
```

Explanation of Example

In Step 0 the Scheduling Service installs a priority mapping that is consistent with the policy that implementation of the Scheduling Service is enforcing. For instance, a priority mapping for an analyzable Deadline Monotonic policy might be different than the priority mapping for an analyzable Rate Monotonic policy. Thus we assume that the Scheduling Service will want to install a mapping that it has configured to be suitable for the policy that the Scheduling Service implementation is enforcing.

There are no RT ORB calls in the example. We expect that it is possible (but not required) that there will be no direct calls to RT ORB primitives if the Scheduling Service is used.

Note that there are no CORBA priorities specified only names for the two CORBA Activities in the client. This facilitates plugging in different fixed priority scheduling policies by choosing a implementation of the Scheduling Service to use. Recall that the Scheduling Service implementation associates the names "activity1" and

"activity2" in the schedule_activity calls in the client (lines 3 and 5 respectively in the client outline) with CORBA priorities. The use of names instead of actual CORBA priorities in application code has two major advantages.

First, the use of names instead of priority numbers allows changing of scheduling policy (e.g. from Deadline Monotonic to Rate Monotonic) without changing or re-compiling application code. If the chosen Scheduling Service was enforcing Deadline Monotonic Scheduling it might, for instance, internally use CORBA priority 10 for "activity1" and CORBA priority 12 for "activity2". If a different implementation of the Scheduling Service were being used, it might internally use completely different CORBA priorities for these two CORBA activities to realize a different scheduling policy (e.g. Rate Monotonic instead).

Second, the use of names instead of priority numbers allows changing *any* CORBA priority without having to find and possibly re-order CORBA priority numbers in application code. The Scheduling Service is the central place to change CORBA priorities. Again, changes in priority can be made without re-compiling application code.

The server in the example has two Scheduling Service calls. The first call accepts the normal parameters to create a POA, except that line 3 of the server example above states that the policy list input parameter has only non-RT policies. This is because the Scheduling Service will set the RT policies itself when it creates the POA in the Scheduling Service call in line 4. This way, the Scheduling Service can select RT policies (thread pools, protocols, concurrency, server priority, etc) that make sense under the uniform scheduling policy that the implementation of that Scheduling Service is enforcing. It also relieves the application programmer from having to determine all of those (relatively complicated) policies themselves.

The second Scheduling Service call in the server is the "schedule_object" call in line 6. This call allows the Scheduling Service to associate a name with the object. Any RT scheduling parameters for this object, such as the priority ceiling for the object, are assumed to be internally associated with the object's name by the Scheduling Service implementation. Thus, the call in Line 6 associates the scheduling parameters (e.g. priority ceiling) with the object reference, perhaps to enforce priority ceiling concurrency control on that object.

Conformance Issues 5

5.1 Introduction

This section specifies the points that must be met for a compliant implementation of Realtime CORBA.

5.2 Compliance

An ORB implementation compliant with Realtime CORBA must implement all of Realtime CORBA, as defined in section 3. Hence there is a single mandatory compliance point.

The Realtime CORBA Scheduling Service, as defined in section 4, is a separate and optional compliance point. An ORB implementation compliant with Realtime CORBA may or may not choose to offer an implementation of the Realtime CORBA Scheduling Service.

END OF DOCUMENT