IST-2001-37652

Hard Real-time CORBA

Title

HRT Protocol Specification

Authors

Miguel Segarra (SCILabs)
Carlos Moreno (SCILabs)
José A. Clavijo (SCILabs)

Reference IST37652/008 Deliverable 2.2
Date 2003-03-31
Release 1.0
Status Final
Clearance Consortium

Partners

Universidad Politécnica de Madrid
Lunds Tekniska Högskola
Technische Universität Wien
SCILabs Ingenieros

IST Project 2001-37652
HRTC
Hard Real-time CORBA

HRT Protocol Specification

Abstract:

The aim of this deliverable is find a suitable specification for easy plug-in of new Transports in an ORB for hard real-time purposes. This document describes the distinct alternatives for CORBA pluggable transports and compares the existing proposals for them. The document also extends the proposals to meet the requirements of hard real-time CORBA systems at the protocol plug-in level.

HRTC Partners:

Universidad Politécnica de Madrid
Lunds Tekniska Högskola
Technische Universität Wien
SCILabs Ingenieros.

Release Sheet (1)

Release:	**0.1 Draft**
Date:	2002 / 12 / 26
Scope	Initial version
Sheets	All

Release:	**0.2 Draft**
Date:	2003 / 02 / 11
Scope	Enhancements to HRTC plug-in description.
Sheets	All

Release:	**0.3 Draft**
Date:	2003 / 03 / 14
Scope	Content for temporal scopes and ETF registries has been removed. C++ source code example added.
Sheets	All

Release:	**1.0 Final**
Date:	2003 / 03 / 31
Scope	Comments from Thomas Losert have been added.
Sheets	All

Table of Contents

1 Introduction

This document studies the different responses to the Extensible Transport Framework for Real-Time CORBA Request For Proposal (ETF RFP). The objective is to analyse the existing transport plug-in framework proposals in order to carry out a comparison of them and to learn about their advantages an disadvantages. Once this is done, it is possible to build on top of existing specifications new interface operations to deal with hard real-time requirements without making unnecesary modifications to existing OMG specifications.

As a starting point, the general and specific requirements that the ETF proposals must comply with are presented. This is important in order to understand if the actual proposals can be extended to match the requirements of hard real-time. After this, a detailed analysis of the specifications is carried out. The analysis describes IDL interfaces, operations and call sequence in the client side as well as in the server side of a CORBA system[1] for the transport plug-in. Finally, modifications and extensions to the proposals are presented in order to deal with hard real-time requirements of distributed systems.

[1] It is not our intention to raise here a discussion on what the terms *client side* and *server side* mean. The terms are used in this document only to make clear the difference between the caller object and the called object.

2 About the contents of this document

This document is centered in the properties of communication transport plug-ins of CORBA brokers for hard real-time purposes. There are more problems regarding end-to-end predictability in CORBA systems than that of ensuring timely communication at the system network level. Those problems are not considered in this document. However, throughout the document some considerations are made regarding facilities that could be provided by ORBs at the layers above the communication transport.

3 Definitions, acronyms and abbreviations

3.1 Definitions

Client Side: Part of a CORBA application from which a connection is made.
Server Side: Part of a CORBA application from which a connection is accepted[2].

3.2 Acronyms

AMI: Asynchronous Method Invocation
CORBA: Common Object Request Broker Architecture
ETF: Extensible Transport Framework
GIOP: General Interoperability Protocol
HRT: Hard Real-Time
IDL: Interface Definition Language
Mars: Middleware and related services
OCI: Open Communications Interface
OMG: Object Management Group
ORB: Obejct Request Broker
Orbos: ORB and Object Services
OS: Operating System
POA: Portable Object Adapter
RFP: Request For Proposal
RT: Real-Time
ST: Smart Transducer
TTP: Time Triggered Protocol
UTC: Universal Time Coordinated

[2] The client and server side definitions made are only valid for the scope of this document.

3.3 Abbreviations

4 References to other documents

4.1 References to project documents

- IST37652 / 029 Domain Analysis for CORBA-based Control Systems.
- IST37652 / 036 Real-Time Protocols for Real-Time Control.

4.2 References to OMG documents

- Orbos / 2000-09-12 Extensible Transport Framework for Real-Time CORBA Request For Proposal.
- The Open Communications Interface (OCI). IONA and OOC. Available at http: / / www.omg.org / docs / orbos / 01-01-05.pdf
- Mars / 2002-09-06 Extensible Transport Framework Joint Revised Submission.
- Mars / 2002-04-03 Extensible Transport Framework Joint Revised Submission.
- Formal / 02-08-02 The Real-Time CORBA Specification v1.1
- Formal / 02-12-02 Common Object Request Broker Architecture: Core Specification version 3.0.2
- Formal / 03-01-01 Smart Transducers Specification v1.0
- Time Service Specification v1.1 May 2002

5 Requirements for the Extensible Transport Framework RFP

This section describes all the requirements for the Extensible Transport Framework RFP. The set of requirements has been included in this document as it is important that the modifications made for HRTC protocol plug-ins comply with the basic requirements of all protocol plug-ins. The requirements are presented in two different categories, General Requirements and Specific Requirements.

5.1 General Requirements

- *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.*

  This document addresses this issue.

- *Proposals shall specify operation behaviour, sequencing, and side-effects (if any).*

  This document addresses this issue.

- *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 document addresses this issue. There are no implied or hidden interfaces.

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

  The optional functionality has been clearly distinguished.

- *Proposals shall reuse existing OMG specifications including CORBA, CORBAservices, and CORBAfacilities in preference to defining new interfaces to perform similar functions.*

  We believe that this issue has been suffiently addressed. The document relies on CORBA, Real-Time CORBA and the Smart Transducers specifications.

- *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 favours upwards compatible proposals that minimize changes and extensions to existing OMG specifications.*

  No changes are needed to existing CORBA specifications.

- *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.*

  This issue has been sufficiently addressed.

- *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.*

  This document relies on existing interfaces where appropriate.

- *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.*

  This issue has been sufficiently addressed.

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

  This issue has been sufficiently addressed.

- *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.*

  Independency of implementation is guaranteed by the use of IDL to specify interfaces.

- *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.*

  We are not aware of any incompatibilities with ISO/IEC 10746. This document does not use UTC representation of time from the X/Open DCE Time Service used by the CORBA Time Service Specification. UTC time units are hundreds of nanoseconds which can a coarse granularity for certain hard real-time applications.

- *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?*
  - *Which accesses to security-sensitive objects must be subject to security policy control?*
  - *Does the proposed service or facility 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.*

- *What default policies should be applied to the security sensitive objects introduced by the proposal?*
- *Of what security considerations must the implementers of your proposal be aware?*

This document centers on the issue of providing hard real-time properties for CORBA transport plug-ins. Although security in this discussion is not a primary issue as most hard real-time systems are closed systems, there is nothing in this specification that precludes a plug-in developer to build a secure hard-real time protocol plug-in.

- *Proposals shall specify the degree of internationalization support that they provide. The degrees of support are as follows:*
  a) *Uncategorized: Internationalization has not been considered.*
  b) *Specific to <region name>: The proposal supports the customs of the specified region only, and is not guaranteed to support the customs of any other region. Any fault or error caused by requesting the services outside of a context in which the customs of the specified region are being consistently followed is the responsibility of the requester.*
  c) *Specific to <multiple region names>: The proposal supports the customs of the specified regions only, and is not guaranteed to support the customs of any other regions. Any fault or error caused by requesting the services outside of a context in which the customs of at least one of the specified regions are being consistently followed is the responsibility of the requester.*

The extensions proposed in this document make no restrictions on the internationalization support of CORBA.

5.2 Specific Requirements.

Scope of Proposals

- *"Proposals responding to this RFP shall define the concepts behind a separation of the messaging layer, such as GIOP, from the transport layer, such as the TCP/IP layer of IIOP. This shall include a formal call model between the layers."*

The extensions of this document only affect to the transport layer of the broker. The formal call model between transport (network protocol) and message layer (GIOP) is not modified by this document.

- *"Responses shall identify interfaces that make the real-time ORB core, facility, and service layers independent of the transport technology."*

  The extensions of this document are independent of the transport technology used. However some features may not be provided by some network protocols dependind on their capabilities (e.g. time trigered vs. event trigered).

- *"Submissions must clearly indicate what kinds of transports are supported through these interfaces, including semantic restrictions."*

  This issue has been addressed.

- *"Responses must clearly show how the proposed framework is extensible, permitting use of third-party transport solutions."*

  This issue has been addressed.

Mandatory Requirements

Responses shall specify interfaces and the appropriate semantics for the following:

a. *Profiles: Responses shall define the IOR profile architecture for the non-TCP transports such that it is possible for a transport author to create a transport plug-in for two different ORBs that enable application interoperability across that transport.*

This issue has been addressed.

b. Communication: Responses shall identify the interfaces and call sequence between the real-time ORB and the transport plug-in.

This issue has been addressed.

c. Selection: Responses shall identify how the real-time ORB selects a particular transport.

This issue has been addressed.

6 Response Comparison to the Extensible Transport Framework RFP

An study has been done of the most relevant responses to the OMG Extensible Transport Framework for real-time CORBA RFP. These responses have been either submitted or supported by some important vendors of real-time CORBA products in the market.

Proposal Number	Submission Date	Submitting companies.	Proof of Concept
1	2001/01/05	IONA Technologies, PLC Object Oriented Concepts, Inc.	ORBacus
2	2001/10/05	Highlander Engineering, Inc. Vertel Corporation	VisiBroker for Tornado
3	2002/04/03	Objective Interface Systems, Inc.	ORBexpress
4	2002/09/06	Borland Software Corp. Objective Interface Systems, Inc. VERTEL Corporation (with support from: Mercury Computer Systems, Inc).	ORBExpress

All the submitting companies have a large experience in the development and implementation of ORBs with plug-in transports.

As can be seen in the “Proof of Concept” column of the proposals table, all submitted specifications are part of existing real-time CORBA products of the submitting companies or have been tested in modifications of those products. The IDL interfaces proposed have been verified and compiled in the vendors’ IDL compilers.

The next sections describe the comparison analysis of the submitted proposals. Attention will be drawn only on the first and fourth proposals. There are no comments to the second and third submissions due to the fact that they were early proposals and the companies involved in the responses joined efforts in the fourth proposal as is shown in the submitting list of companies. Besides, the fourth proposal is based on the concepts included in the second and third proposals.

Otherwise, the first proposal is somewhat different from others and is based on the Open Communications Interface (OCI) although the concepts handled by all of them are quite similar.

The following sections present and describe the interfaces needed to allow a particular ORB to use an arbitrary transport protocol. In all the responses, the author of the plugin must provide the implementation for these interfaces.

6.1 Pluggable framework architectural overview

This section describes the relationships between the object and concepts that appear in the OCI and in the ETF proposed specifications.

CONCEPTS OF THE PLUGGABLE FRAMEWORK

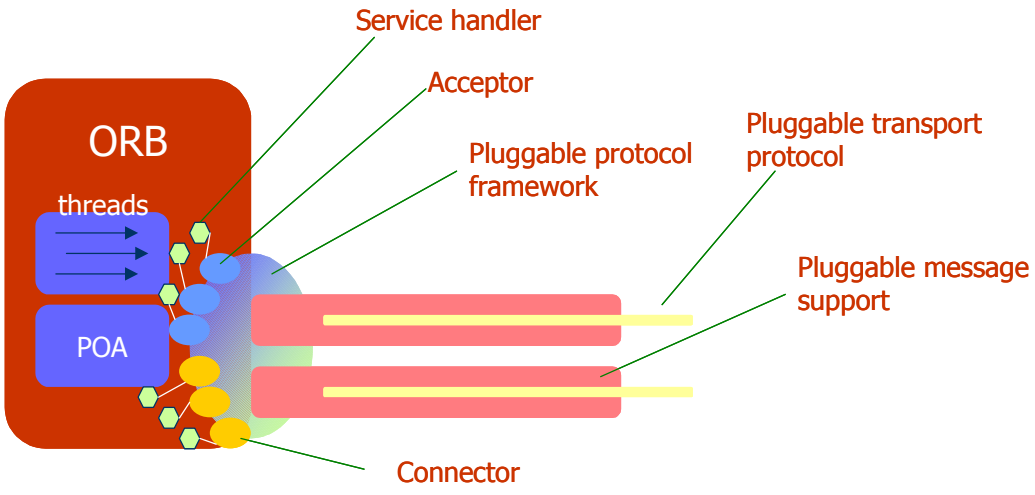

Figure 1: Pluggable framework architecture overview

Figure 1 shows the basic structure of an ORB that uses a pluggable transport framework. The figure shows the server side of an ORB in which a Portable Object Adaptor (POA) and a threadpool to serve requests

are depicted. For the purpose of this document the interesting part of the figure is that of the labeled elements of the figure. It shows that a pluggable protocol framework can be composed of two different levels of components:

- **Components of the pluggable message protocol:** For this document, it must be understood that the *message layer* is the *ORB message layer*. This means the GIOP message layer. It is not in the scope of this document to introduce a pluggable framework for the message layer (to allow the plugin of a Real-Time Interoperability Protocol or RIOP). Introducing a RIOP plugin probably means that interoperability with other CORBA brokers will be lost being this a major handicap for message protocols different than GIOP.
- **Components of the pluggable transport protocol:** This layer is placed under the message protocol layer (the GIOP message layer) and it is here where we want to be able to introduce protocol plugins for hard real-time communications. The transport protocol layer sits on the network protocol and provides a way to hook a transport protocol to the broker with independency of its developer.

The transport protocol framework is responsible for the creation of the acceptor and connector objects which in turn provide service handlers to carry out communication through a given network protocol.

Acceptor objects are passive entities that wait for requests of connection. Requests of connection are always initiated from the client side by connector objects which play the active role. As a result of connection acceptance by the acceptor, service handlers are created to carry on communication. As will be seen in the next pages, for the OCI the service handlers are mapped to the Transport object interface while for the ETF, service handlers are part of the Connection interface. In the latter, the Connection interface also defines the functionlity of connectors while for the OCI, the connector functionality is defined in the Connector interface. The role of acceptor is quite similar in both approaches, being called Acceptor interface for the OCI and Listener interface in the case of ETF.

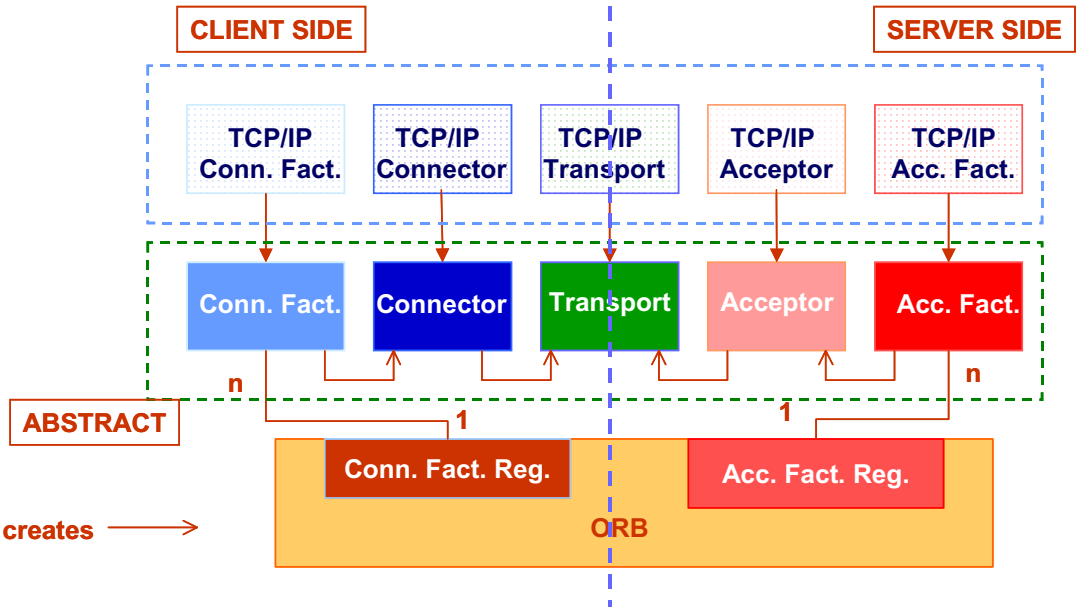

Figure 2: OCI architecture

OCI ARCHITECTURE

In the OCI architectural overview (Figure 2) there is a clear separation between the client side and the server side framework objects. There is also a “creates” relationship between the different objects of the framework. As shown in the figure, the factory registry objects for connector and acceptor factories can be related to *n* factory objects. Each of this factory objects is able to create acceptors or connectors for a certain type of transport protocol. At the same time, when a connection is accepted by the acceptor at the server side, the connector at the client side and the acceptor at the server side create a transport (service handler) that is used to carry on the communication. Notice that this mechanism allows to establish connections in advance which allows to avoid the overhead of the first invocation on a server object. In the case of a standard CORBA object the binding will be made at the time of the first invocation but in the case of a real-time application it is possible to do it before the first invocation is made. This will increase end-to-end predictability for the first request.

ETF ARCHITECTURE

Figure 3 shows the proposed architecture of ETF. It greatly resembles the OCI architecture but some objects are mirrored in the client and server sides. Notice that the figure shows a factory registry object in the ORB but no link between the registry and the factory objects is shown as they are not in the scope of the ETF proposal.

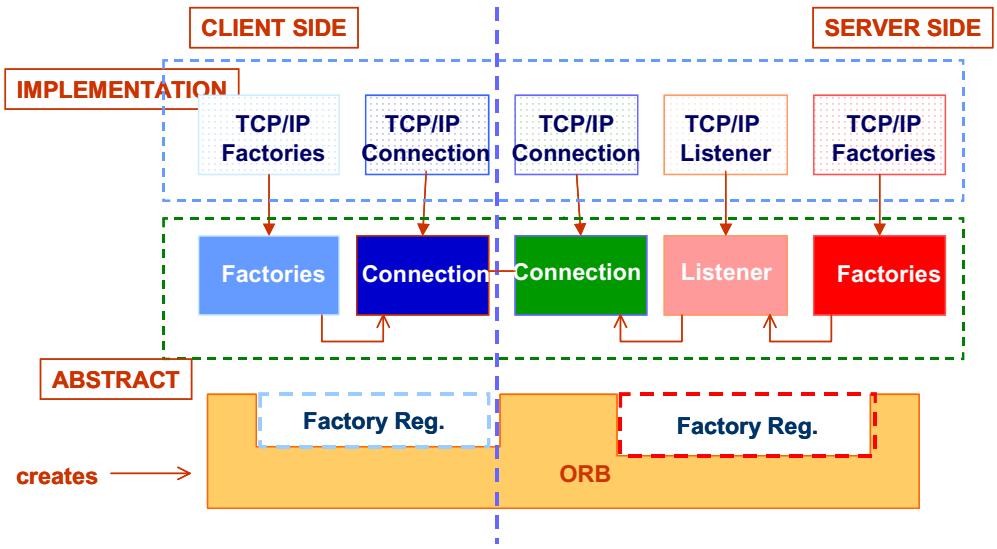

Figure 3: ETF architecture

Both specification proposals OCI and ETF take advantage of symmetry in a different way. For the OCI specification, there is conceptual symmetry for all objects in the client and server sides, only their names change; AccFactoryRegistry and ConnFactoryRegistry, AccFactory and ConnFactory, etc. The Transport interface is the same for both sides as it is the one in charge of communication. The only assimetry is seen in the role of the connector and the acceptor objects which have different functionality.

In the case of ETF there is also symmetry, the factory object is the same in the client and server side but there is also asymmetry due to the fact that the server side is composed also by Connection objects. This is needed because the Connection interface holds the operations to establish connections and to carry on communication. These operations were separated in the case of the OCI in the Connector and Transport interfaces. The Connection interface provides a functionality at the server side (that of establishing a connection) which does not belong to the server side as a passive entity. Without being a major drawback for ETF, this makes the OCI conceptually cleaner than ETF.

6.2 Common requirements

There are some common requirements that apply to all the proposals. This requirements are made on the transport mechanism used under the pluggable transport framework.

- **Connection oriented:** The undelying transport must be connection oriented as seen by the pluggable transport interface.
- **Reliable:** Arbitrary messages of any length must be sent to the remote endpoint. All internal details of the transport as packaging or packet reordering or dropping must be hidden.
- **Bi-direccional:** replies to requests must be reliable received.

It must be noticed that there are transports that do not comply with these features. The important point is that the above requirements are stated from the point of view of the pluggable transport framework and it is possible to build a transport plugin for unreliable transports by adding code on top of the transport or protocol stack to add the functionality which is missing.

6.3 Client side

In the client side, the ORB needs an interface to handle the connection and the data exchange (read and write operations), as well as an interface to manage profiles of any transport plugin, extract information from an IOR, add profile data, and other auxiliary operations.

As said in the previous section, we will center our attention on the first and fourth proposals. Both of them define operations to send and receive messages (octets streams of any length) as well as to control and handle the connections and time-outs.

CLIENT SIDE FOR THE OCI

The first submission (OCI) proposes the following interfaces:

- **OCI::Buffer**
  This interface manages the arrays of data octets that will be sent or received. The Buffer interface also holds a position counter so it is possible to know the amount of octets sent or received.
  - **Advance:** Increment the position counter by a quantity.
  - **Rest_length:** returns the rest length of the buffer.

- **Is_full:** checks if the buffer is full.

- **OCI::Transport**
  This interface is similar to the fourth proposal "Connection" interface. It has methods for sending and receiving messages. Also we can specify timeouts and close the transport. There is a handle to determine if the transport is ready to send and receive data. The objective of the handle is to find out if the transport is ready to send or receive messages.
  - **Close:** Closes the transport. Should call shutdown before closing. No further operation (read, write) can be executed on this transport.
  - **Shutdown:** This is a method to shutdown a transport. Upon calling shutdown, threads blocked on receive operations will return or throw an exception. After calling shutdown no operations on the associated TransportInfo object can be called.
  - **Receive, receive_detect and receive_timeout:** the receive method receives a buffer's content. This can be done either blocking until the buffer is full or just returning at the data arrival. Receive_detect is also able to return FALSE in case of a connection loss and receive_timeout can return before the buffer is full by specifying a timeout parameter.
  - **Send, send_detect** and **send_timeout:** these are the matching operations of the receive interface. Send sends a buffer contents blocking or not until the whole buffer has been sent. Send_detect is also able to detect a connection loss and send_timeout can return if the timeout expires before the whole buffer has been sent.
  - **Get_info:** returns the information object associated with the transport.

- **OCI::TransportInfo**
  Provides information about the Transport object. It has several attributes describing the acceptor object that created the transport (in the case of a server) or the connector object that created the transport (in the case of a client). The info object also is able to register a callback object in the case the transport is closed as well as information regarding the identification of the transport plugin.

- **OCI::CloseCB**

Provides an interface for a callback object. The object holds a callback function that will be executed before the transport is closed.

- **Close_cb:** callback function to invoke.

- **OCI::Connector**
  This interface defines the operations to allow clients establish a connection to a server. In addition, it has functions to manage IORs and extract profile information that satisfies the specified CORBA policies. As a result of establishing a connection, a transport object is created.
  - **Connect and connect_timeout:** It is used by clients to establish a connection to a server. It is possible with the timeout function to establish a timeout and to check on return whether a nil object reference for the transport was obtained.
  - **Get_usable_profiles:** This is a helper method that allows if an IOR matches a set of profiles for this connector.
  - **Get_info:** returns the informational object associated to this connector.
  - **Equal:** Determines whether this connector is interchaneable with other connector.

- **OCI::ConnectorInfo**
  Provides information about the Connector object. It is similar to the TransportInfo object but for connectors. It provides means to add a callback object that will be called upon connection.
  - **Connect_cb:** calback operation to add a callback object that will be called when a connection is made.

- **OCI::ConnectCB**
  Defines a callback object for connectors. The callback function is invoked when a new connection has been established.

CLIENT SIDE FOR THE ETF

In the fourth proposal, the funcionality described for the previous interfaces is achieved by the “Connection” and “Profile” interfaces.

- **Connection Interface**

  The fourth proposal includes the “**Connection**” concept. It is a link between the GIOP layer and the Extensible Transport Framework

layer. Such a “Connection” allows the GIOP layer to send and receive any kind of message, without knowing any details of the transport. The connection must be reliable (any arbitrary length message can be sent to a remote endpoint) and bi-direcctional (thus, we can send reliable requests). The transport must provide these characteristics, otherwise the plugin has to include support for these requirements in a layer on top of the transport and inside the plugin.

There are operations defined to:

- **Exchange data: read** and **write** functions. These operations must be completed successfully. In case of failure an exception is raised. It is also important to guarantee the integrity of the GIOP stream, so in the case a timeout occurs the plugin can decide to close the connection. In the case of **read**, the plugin cannot close the connection as the GIOP level is the only one which can determine the integrity of the GIOP stream.

  As stated in the submission, ORBs are allowed to call the write/read operations only from one thread at a time on the same connection object. However, multiple threads may call write/read operations simultaneously on different connections at the same time.

- **Connection Handling:** This interface also resembles the functionality of the OCI::Connector. The **connect** operation allow to establish a 1-to-1 connection with a server, by means of the endpoint supplied by the “Listener” object at the server side. There are also, **close** and **is_connected** operations. The **close** operation closes the connection using the transport mechanism for disconnection and releasing the associated resources. **Is_connected** is a helper operation to find out the state of a connection.

- **Reading the Server Profile:** In order to be able to connect to the”Listener” object at the server side, it is necessary to provide the connection with the server endpoint information. This is done by means of Profile instances which are created by the Listener objects and supplied to the **connect** operation. Sometimes, it is also useful to locate an established connection by looking at the profile stored by the

connection. The **get_server_profile** operation provides this service.

- **Request Dispatching Support: read** and **write** operations described so far match a “thread per connection / session” model of request handling. Dispatching models distinct from this can benefit from the operations **is_data_available** and **wait_next_data**. It allows a “temporary mapping of request execution objects to connections”. The first operation returns immediately if data is available whereas the second is able to wait for a specified timeout to wait for the arrival of new data.

There are other auxiliary attributes in the connection that allow to specify an identifier for the connection, to check for new data and for specific information regarding functionality of the GIOP version used.

- **Profile Interface**

  All the information of a particular transport is managed in the plugin via profile instances. The basic information stored in the profile is that of the server contact endpoint, so a client is able to connect to a server. The Profile interface defines the operations needed to process the profile data and to store it in IORs. It contains conversion operations such as **marshal** which is used to insert the transport specific information (byte order, GIOP version, endpoint address and other components) into a TaggedProfile which will be stored in an IOR.

  Also, there are other useful methods for matching (**is_equivalent**) and copying profiles (**copy**)as well as a **hash** function to improve the management of large sets of profiles. The GIOP version supported by the profile is indicated by the **version** attribute.

6.4 Server side

The ORB server side needs interfaces to handle client connection requests and to manage IOR profiles.

SERVER SIDE FOR THE OCI

In the first proposal (OCI) the following interfaces are in charge of performing this funcionality:

- **OCI::Acceptor.**
  Used by CORBA servers to accept client connection requests. To do this, the **listen**, **accept**, **connect_self** and **close** methods must be used. The Acceptor creates a Transport when a new connection to the server is accepted. It also includes operations to extract information from the IOR and add new profiles that match the object policies.
  - **Close:** closes the acceptor, accept or listen may not be called after the object has been closed.
  - **Listen:** prepares the acceptor to listen for new incoming connection requests. Until an acceptor is listening attempts of connection will result in communications failure exception.
  - **Accept:** accept is used to accept connection requests. This method can be blocked until a new connection is accepted. When a new connection has been accepted the operation returns an object reference to a transport object that can be used to send or receive octet streams.
  - **Connect_self:** this is a helper method used to unblock threads that are waiting blocked for incoming connection requests in the accept operation. It is useful when the acceptor is blocked waiting for connections in the accept call.
  - **Add profiles:** this methods adds new profiles that match the acceptor to an IOR.
  - **Get_profiles:** extract the profiles from an IOR which are local to the acceptor.
  - **Get_info:** retrieves an AcceptorInfo object with the information associated to this acceptor.

- **OCI::AcceptorInfo.**
  Information about the Acceptor object. This object allows to set a callback object that is invoked whenever a new connection is accepted by the acceptor. It allows also to consult the transport identification and a human readable description of the transport.
  - **Add_accept_cb:** Add a callback to be called when a new connection is requested.

- **OCI::AcceptCB.** Callback object. The callback operation is automatically invoked when a new connection has been accepted.

SERVER SIDE FOR THE ETF

In the fourth proposal (ETF) the interfaces involved are: Connection, Profile, Handle and Listener. The **"Listener"** will manage the endpoints for the client requests. These endpoints will be encapsulated in the profile and will be used by clients to request a new connection.

- **Listener Interface.**

  When a client needs to make a request to a particular CORBA server, it asks for a new connection. This interface provides operations to accept a new connection, allowing to block until the connection has been accepted and close the object (it implies all opened connections will be closed).

  The Listener interface has operations for the following functionality:

  - **Setup:** The **set_handle**, **accept** and **destroy** functions. The **set_handle** method establishes the link between the ORB and the server endpoint of the plugged transport (see the description of the Handle Interface). The method must not be invoked before any call to the accept operation. **Accept** returns an instance of the connection and blocks until the client connects to the server. **Destroy** closes the endpoint and destroys all the connections managed by it.

  - **Dispatch:** The proposal includes operations to manage idle connections. It is possible for the ORB to carry out some sort of "virtual disconnection" via the **completed_data** operation, the connection can be returned from the ORB to the listener and incoming data will be signalled to the handle by the listener. This means that the Listener object is the one in charge of handling idle connections. By means of the **is_data_available** operation it is possible to determine the state of the connection.

  - **Profile Data:** The "Listener" has an attribute that returns a copy of the profile that contains the endpoint address.

- **Handle Interface.**

The fourth proposal uses the "Handle" interface to provide the operations that allow interaction between the ORB and the plugged-in transport.

- **Add new Connections:** A "Handle" instance contains functionality to announce a new client connection to the ORB. The **add_input** operation serves for this purpose and it gives a chance to the ORB to reject the connection.

- **Signalling Incoming Data:** The **signal_data_available** function is used by the plugged-in transport when data arrives to the server endpoint. It initiates a new request dispatching cycle in the ORB. Any other incoming data for this connection is ignored until the connection is returned to the Listener by means of the **completed_data** method.

- **Client Side Close:** It is used by the plugin to signal the Handle that the client has closed the connection.

6.5 Factories

In the previous sections, we have described the interfaces the plugin transport needs to interact with the ORB. The instances of the objects that implement these interfaces are created by "Factories".

FACTORIES IN THE OCI

In the OCI Module there exist the following interfaces for the management of factories:

Acceptor	Connector
OCI::AccFactory	OCI::ConFactory
OCI::AccFactoryInfo	OCI::ConFactoryInfo
OCI::AccFactoryRegistry	OCI::ConFactoryRegistry

These interfaces contain operations to create the Acceptors and Connectors which are suitable for the particular parameters of the Transport. It is important to notice that this specification supports the concept of registry. So it is possible to register a factory for a specific transport and later refer to it.

The factory objects allow to create acceptors for an specific type of transport in the case of the server side while in the case of the client side allow to create connectors. For this, the **create_acceptor** and **create_connector** operations of the respective factories are used.

The factory registries provide means to register new connector or acceptor factories for an specific transport. This is done by using the **add_factory** operation of either interface. It is possible to retrieve a factory interface (**get_factory**) by its protocol identifier or to get the sequence of registered factories by a call to the **get_factories** operation.

FACTORIES IN THE ETF

There is also a factory interface definition in the fourth proposal. For this specification, all the creation operations are collected under the same factory interface. There are three creation funtions:

- **Create_connection(in RTCORBA::ProtocolProperties props);**
  This function will be used to create an instance of the "Connection" interface for a particular transport. As it is indicated in the submission, the connection properties are expressed in standard real-time CORBA form (as ProtocolProperties) and can be fixed either from client side or the server side (and exposed to the clients by IORs).

- **Create_listener(in RTCORBA::ProtocolProperties props);**
  This operation is called to create an instance of the "Listener" object. When the function returns, a new endpoint to listen for requests has been created. The profile associated to the listener will be accesible just after calling this operation. The ProtocolPolicies applied at the server side that are used in this operation have been set on the related Portable Object Adaptor (POA).

- **Demarshall_profile(inout IOP::TaggedProfile data, out IOP::TaggedComponentSeq components);**
  The objective of this operation is to create a new profile for this plugin by demarshalling the information found in the tagged profile.

There is a factory instance per plugged-in transport. The identifier of the factory for a transport is that of the transport and is identified by its profile tag number.

The fourth proposal does not provide a mechanism to register factories into the ORB and leaves the implementation details of these interfaces out of the proposal.

6.6 Zero Copy Interface

One of the basic things to do in order to improve performance is to avoid unnecessary copying of data objects. The ETF proposal makes provision for an optional zero copy interface for the connection object. This is useful in the case that the transport or communications library grants access to the buffers used in the protocol stack or is able to use buffers provided by the application or ORB. At least in the case that there is no access to the protocol stack advantage is taken as the copy between ORB and transport plugin buffers can be avoided.

The zero copy interface is implemented by inheriting from the connection interface and providing a BufferList interface representing either the data to be sent to the remote peer or the data received by the connection instance.

- **BufferList interface**
  The BufferList interface is a list of pointers to octet sequences which can be manipulated by the ORB or the transport layer without the need of copying their contents.
  - **Add_buffer.** This is the way used by the ORB to allocate a zero copy buffer. The returned buffer is identified by an index. This way it is possible to make further references to the buffer by its index.
  - **Get_buffer.** Retrieves a buffer by its index.

The BufferList also contains an attribute to identify the number of buffers that the BufferList is currently holding.

- **ConnectionZeroCopy**
  This interface contains the methods used to read and write data in a connection object with support for the zero copy mechanism.
  - **Write_zc.** This operation writes a zero copy buffer list to the transport. It supports the timeout mechanism basically to comply with real-time CORBA specifications.
  - **Read_zc.** This operation reads data into plug-in supplied buffer. It also supports the timeout mechanism.
  - **Create_buffer_list.** This method creates a buffer list.

6.7 Interface Mapping Between Proposals

In this section a mapping between both proposals is made as to show that they are basically similar and that both are built around the same concepts and patterns. Factories, connectors, acceptors and transports along with several data holder entities are all the concepts that need to be implemented to provide a pluggable transport framework to an object request broker.

The fourth proposal (Borland's) has a smaller amount of IDL code and interfaces but this is because in the OCI there is a clear separation between the client and the server side.

In the case of the ETF, an optional interface for zero copy buffers is provided. This can also be achieved in the OCI as the Buffer interface can be written to use as the protocol stack buffers. The following table shows the relationship between both specifications:

ETF module	OCI module
Profile	ProfileInfo
Buffer	Buffer
Connection	Connector Transport
Handle	AcceptCB ConnectCB CloseCB
Listener	Acceptor
Factories	AccFactory ConnFactory
Not provided by the specification	AccFactoryRegistry ConFactoryRegistry
Zero-copy interface	*Can be provided by the Buffer implementation*

The ETF leaves out the scope of the specification the definition of the registry object used to register new plugins into the ORB. Also, the ETF provides a zero-copy interface which can be resembled in the OCI by a proper implementation of the Buffer interface. Notice that it is not possible to implement a zero-copy interface if the underlaying network protocol stack does not allow to share a buffer with the application.

The OCI module provides the same interfaces in a different way ETF does. ETF interfaces provide the same functionality as the OCI in a more simplified way. Further, the ETF module makes use of the real-time CORBA ProtocolPolicies interface as a way to configure the transport plugin. One reason for this, is that the OCI is a much earlier proposal than ETF and ETF has taken benefit from this situation. Initially, OCI was not designed thinking in real-time CORBA but in adding a pluggable protocol framework to CORBA. This can also be seen as drawback for the ETF as it will not be possible to use the ETF plugin in a non real-time CORBA broker (because it is configured by the use of real-time CORBA protocol policies). On the other hand, OCI can be used either in standard or real-time CORBA brokers.

7 Extension of the ETF/OCI for hard real-time CORBA applications

In IST37652/036 RT-Protocols for real-time control, an identification of real-time systems interfaces has been done. Also the necessity to deal with time at the level of interfaces appears as a conclusion from that document. As stated before the so called real-time systems often rely on the concept of priority as a form of task precedence or thread elegibility of execution. Being this an intuitive concept, its mapping on time does not ensure any temporal properties of the system. This is why it is needed to directly deal with time in the components interface of real-time systems. Three unique interfaces existing in most real-time scenarios have been identified: the Real-time Service (RS) interface, the Diagnostic and Management (DM) interface and the Configuration and Planning (CP) interface. The separation of these interfaces is useful as the level of service or Quality of Service (QoS) provided by them is different in every case. Interfaces are separated according to the properties they offer as a component to the rest of the system architecture.

As stated in IST37652/029 Domain Analysis for CORBA-based Control Systems, there are several important issues that CORBA and RT-CORBA are lacking:

- **Deterministic transports.** IIOP (GIOP over TCP/IP) which is the most commonly used transport for CORBA brokers does not give any end-to-end timing guarantees. A minimal requirement of an upper bound for the end-to-end latency is needed.
- **Periodic activities.** In real-time communications a common case is that in which the sender periodically transmits messages to one or more receivers. There is a need in CORBA to specify periodical client invocations to servers.
- **Scheduling.** In order to guarantee communication timing contraints it is necessary to schedule access to the network. To be

able to achieve this, a global knowledge of network accesses must be available from the ORB level. Considering an static approach, a usual case in hard real-time systems, scheduling could be achieved by knowing the network time-slot asigned to a node or the maximum send rate for a node.

- **ORB footprint.** Close to the process resources are small. In many cases, hard real-time applications are close to the process so it is important to have a small ORB footprint. RT-CORBA sits on top of CORBA which size is not appropriate for this type applications. HRT-CORBA should be built on top of minimum CORBA with a modular approach in which features could be plugged-in.
- **Backward compatibility:** In order to preserve vertical integration with other CORBA implementations, interoperability must be maintained. This means at least keeping the GIOP layer and / or TCP / IP in a transport plugin used for non real-time services. GIOP and other ORB message protocols could be maintained in the case an ORB pluggable message framework is implemented.

7.1 RS interface

This interface is in charge of providing its service in a predictable temporal way to the rest of the environment. In the case of real-time CORBA the transport plugin is a component of the system architecture responsible for the communication of an octet stream between peer endpoints. Current interface definition (ETF or OCI module) is based on a *best-effort* approach where no control over the temporal behaviour of the transport layer can be exercised.

Communication activity is carried out by means of the **ETF::Connection** or by the **OCI::Transport** interfaces. The IDL excerpt shown below is the interface specification for **Connection** object of ETF.

```
// locality constrained
      local interface Connection
      {
            void write(in boolean isFirst,
                       in boolean isLast,
                       in Buffer data,
                           in unsigned long offset,
                           in unsigned long length,
                           in TimeBase::TimeT time_out);

            void read(inout Buffer data,
                      in unsigned long offset,
                          in unsigned long min_length,
```

```
                        in unsigned long max_length,
                        in TimeBase::TimeT time_out);
                // transport needs to set data.length() to
                // offset + number of bytes actually read

        void flush();

        void connect(in Profile server_profile,
                     in TimeBase::TimeT time_out);

        void close();

        boolean is_connected();

        Profile get_server_profile();

        boolean is_data_available();

        boolean wait_next_data(in TimeBase::TimeT time_out);

        readonly attribute long id;
        readonly attribute boolean supports_callback;
        readonly attribute boolean use_handle_time_out;
};
```

The basic functionality of this interface is provided by the methods **read** and **write**. As a consequence of the real-time CORBA timeout policy (which is the Messaging specification **RelativeRoundtripTimeoutPolicy**) the **TimeBase::TimeT time_out** parameter is part of the operations signature. But the degree of temporal control this kind of interface provides is very low. It is only possible to notice that a request cannot be delivered or a reply received after the timeout (e.g. deadline) has expired. This is not an acceptable approach for a hard real-time system.

Instead of this, functionality should be added to learn if it will be possible to deliver a request without occurring the timeout or reaching the deadline. Real-time CORBA 1.0 imposes the restriction of fixed priority scheduling. In practice, this means that to develop a real-time application with CORBA, a relationship between the priorities and deadlines of the system must be established. While, as said, the use of priorities is intuitive, it is certainly difficult to determine the level of determinism of a system or to say that the deadline will be met just because priorities have been assigned. In the best case, determinism will not be possible without global knowledge of the mapping of priorities in the whole system. And this means not only to CORBA priorities, it means from CORBA to native OS priorities and how these are handled by the OS. In general, real-time application developers perform off-line scheduling analysis to provide deterministic behaviour and base the core of the system on periodic tasks

with a time constant many times smaller than that of the system. It is also needed to handle aperiodic or sporadic tasks as a consequence of incoming events in the system. There are neither real-time CORBA nor Extensible Transport Framework tools to do this. The reason for this, is that the management of time poses difficult problems so CORBA does not address them at all. This is also true for most programming languages where usually there is only support for timers. To deal with the notion of time in real-time systems, there must exist hooks in the programming interface that let us deal with time at least in the following ways:

- Access to clock.
- Delaying of tasks.
- Handling of timeouts
- Deadline specification and scheduling.

CORBA COMMUNICATION MODEL

Most CORBA systems are based around the client-server model in which a client makes a request and blocks until a reply from the server is received. This is the synchronous two-way communication model commonly used in CORBA. The other most commonly used communication model in CORBA is the oneway communication in which the client does not wait for a reply. A drawback of oneway invocations in CORBA is that, although it is not possible to specify user exceptions for oneway invocations, the ORB is able to throw standard system exceptions which could be raised at the server side and catched at the client side.

The Messaging specification of CORBA introduced the AMI with which operations can be called asynchronously using the static invocation interface. The AMI offers two ways of invoking an object operation non-blocking the client, being most appealing to us the *callback* or reply handler model in which the ORB will invoke an application defined reply handler upon reply form the server. This concept of callback is common in event driven systems and has been widely implemented in different types of systems (MS Windows for instance).

EVENT-TRIGGERED VS TIME-TRIGGERED SYSTEMS

There is a great conceptual difference between time-triggered and event-triggered systems. In a time-triggered architecture, system activities are initiated by the progression of a globally synchronised time base while in an event driven system, activities are driven by other events different than the progression of time. In the previous section, the reply handler callback model is a pure event-triggered system as the handler is not activated

until the request from the server (the event) has been received. The callback model or the oneway invocation communication model are suitable for real-time applications in which the communication plug-in is event-driven. Unfortunately, this communication model alone is not good for time-triggered systems as activities are initiated synchronised with a global time base[3]. In a system with a time-triggered communication layer, the system activities must be driven from the communications layer and the transport plugin which must be time-aware. An event-triggered system is more flexible than a time-triggered system but the latter is more predictable. In a time-triggered approach all activities and communication take place at *predetermined* instants of time.

DRIVING THE SYSTEM FROM THE COMMUNICATIONS LAYER

For hard real-time applications the ORB must be aware of the progression of time. This must hold for the distributed system. With this requisite and for a time-triggered network protocol, the time should be obtained from the communication layer and made available to the application or the ORB. This can be achieved by extending current definitions for the OCI or the ETF submission.
A pluggable transport must be able to tell the application what time it is as well as a measure of time precision and accuracy. It is also needed for the application to learn when the next period or cycle of execution begins. Again this service must also be provided from the transport plug-in.
It must be noticed that time-awareness for the ORB or the application might be less precise than that of the transport protocol being then necessary to downgrade the time for the application.

7.2 Extensions to the Extensible Transport Framework

In order to drive the ORB/system activities from the communications layer it is needed to incorporate the notion of time into the pluggable protocol framework. The representation of time becomes then a crucial point as it should be simple and consume as less processing power as possible. In the OMG document Formal/03-01-01 Smart Transducer Specification a time instant is represented in IDL as

typedef long long TimeInstant;

[3] Actually, it is not needed to be aware of the time of the global time base. Only the difference of time with the time base and the accuracy/precision of the time base in the nodes is needed.

and its data representation is the following:

typedef long long TimeInstant:
This type is used for timestamps. The 40 upper bits represent the number of seconds (all 34841 years an overflow will occur) while the remaining 24 bits represent the fractions of a second, allowing an accuracy of 60 ns. In a system with external clock synchronization the 40 upper bits are initialized with the value 0 at 00:00:00 UTC on January 6, 1980, which is also the reference starting point (the epoch) for GPS-time. In this way every point in time 17420 years before and 17420 years after January 6, 1980 can be uniquely represented with an accuracy window of 60 ns. Stand-alone systems without external clock synchronization are set to 0 during initialization.

Formal/03-01-01 also defines a time duration **typedef long long TimeDuration** as:

typedef long long TimeDuration:
This type is used for durations that are represented in units of 2^{-24} *seconds (about 60 ns).*

There is also a representation for periodic instants:

```
struct Instants {
        TimeInstant instant;
        TimeDuration period;
};
```

struct Instants:
The first value (subfield instant) informs about the next instant when the most recent of the denoted events will occur. The second value (subfield period) is the period of the named data item.

This representation of time can be used in the pluggable protocol framework to add a service that allows the real-time system/ORB to learn when the next cycle of processing/execution must begin.

WHERE TO ASK FOR THE TIME IN THE PLUGGABLE TRANSPORT FRAMEWORK?

The pluggable transport is the entity in the CORBA architecture that is aware of global time. The pluggable/extensible transport framework must provide means to forward time to the ORB. Notice that forwarding the time to the ORB does not mean that the real-time application is time-

aware. The communication plugin interfaces are not available at the real-time CORBA application level. It will be necessary also to provide an interface for the application to gain knowledge of the progression of time.

Regarding the extensible protocol framework and as explained in the sections of the OCI and ETF, a pluggable protocol framework handles the concepts of registry, factory, connector, transport and acceptor. Only the transport is used when the system is working to communicate data. The reminding entities are used for configuration and setup of connections. Once a connection has been established, it is the transport object the one in charge of communications. This means the interface related to the transport either in the OCI or in the ETF submission should be extended to let the ORB ask about the progression of time. It is not possible to ask for the time without being connected to a tranport.

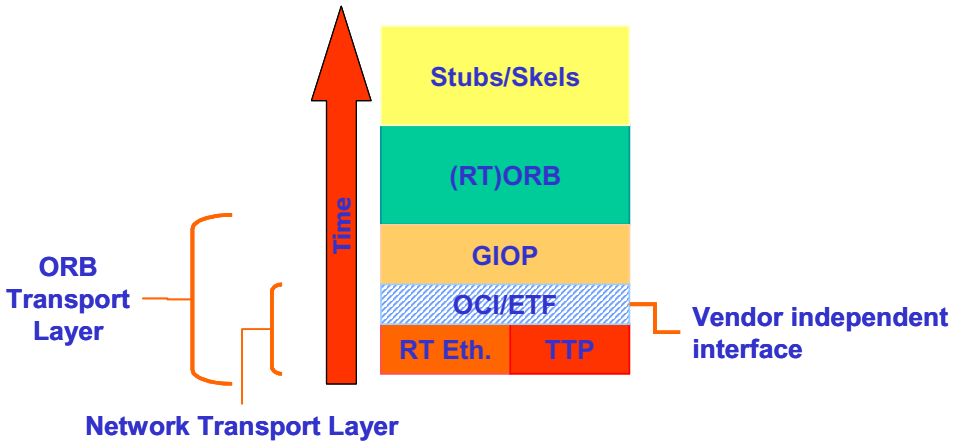

Figure 4: Propagation of time in the real-time ORB

Figure 4 shows the layered architecture of a real-time ORB with an extensible transport framework. The time arrow shows how propagation of time occurs in the ORB and in the CORBA application. In this approach, the global progression of time is only known by the transport layer (e.g. the TTP protocol) and the extensible prtocol framework interface provides means to allow the ORB ask for the time. The figure also shows that it is the message layer of the ORB (the GIOP layer) the one with complete access to the plugin interface (e.g. to the transport object) so it is not possible for the CORBA application to access directly to the transport to learn the time.

ASKING THE TIME IN THE ETF SUBMISSION

To ask for the time in the ETF submission, the interface which defines the role of the transport must be identified. In the ETF, this role is represented by the **Connection** interface. The **Connection** interface also serves as a connector and is able establish connections to servers. The following IDL code is the ETF Connection interface extended with operations to ask the global time as seen by communications protocol.

module ETF{

```
// declaration of time types
typedef long long ProtocolTime;           // a time instant
typedef long long ProtocolDuration;       // a duration
struct ProtocolInstant{       // current time and period
      octet messageID;        // message identifier
      ProtocolTime instant;
      ProtocolDuration period;
      octet precision;        // precision
};

// locality constrained
      local interface Connection
      {
            void write(in boolean isFirst,
                       in boolean isLast,
                       in Buffer data,
                           in unsigned long offset,
                           in unsigned long length,
                           in TimeBase::TimeT time_out);

            void read(inout Buffer data,
                      in unsigned long offset,
                          in unsigned long min_length,
                          in unsigned long max_length,
                          in TimeBase::TimeT time_out);
                     // transport needs to set data.length() to
                     // offset + number of bytes actually read

            void flush();

            void connect(in Profile server_profile,
                         in TimeBase::TimeT time_out);

            void close();

            boolean is_connected();

            Profile get_server_profile();

            boolean is_data_available();

            boolean wait_next_data(in TimeBase::TimeT time_out);
```

```
            readonly attribute long id;
            readonly attribute boolean supports_callback;
            readonly attribute boolean use_handle_time_out;

            // protocol time
            attribute readonly ProtocolInstant protocol_time;
            ProtocolInstant protocol_period( in octet messageID );
        };
};
```

In this IDL, time is represented as in the Smart Transducers Specification. This representation of time has the advantage that it is possible to synchronize a site with a signal from a GPS receiver (the time representation has a granularity of 60 ns) as the epoch is synchronised with that of GPS..

The **ProtocolTime** and **ProtocolDuration** have the same interpretation as in the Smart Transducers Specification. The same holds for the precision octet which represents the number of significant bits in the timestamp.

Time Precision (from formal/03-01-02):
The Precision represents the number of significant bits in the timestamp. This concludes in an error window of $2^{39-PREC}$ *seconds. Valid values are from 0 (no precision; the timestamp might be a random value) to 63 (an error window of about 60 nanoseconds). Note that this parameter refers to precision within an ST system, not to the accuracy between the clocks within an ST system and the external time reference.*

The **ProtocolInstant** structure is able to provide information for the current time, a period of time and the precision of the time measure. This can be done for a certain message identifier which will be mapped to a state variable of the system. With this data type, it is possible to add an attribute and an operation to the **Connection** interface.

The **protocol_time** attribute gives the global time as seen from the communication protocol and the precision of this data. The period and messageID fields are unused in this case.

For the **protocol_period** operation, the **messageID** informs the pluggable framework to obtain timing information for a certain message (messages occur periodically and are used to read / write state variables of the system), the **instant** data member of the struct is the time instant in which the next period for this message id will begin, **precision** is the precision of this measure and **period** is the period of update. Knowing the period of

update of the different messages of the application, the CORBA clients can be synchonised to make requests at proper times.

ASKING THE TIME FOR THE OCI

In the case of the OCI, the roles of the connector and the transport are clearly separated. The OCI has a **Connector** and a **Transport** interface so the additional interface for the time must be placed into the **Transport** interface. The IDL for this extension is the following.

```
//IDL

module OCI{
      // declaration of time types
      typedef long long ProtocolTime;          // a time instant
      typedef long long ProtocolDuration;      // a duration
      struct ProtocolInstant{      // current time and period
            octet messageID;
            ProtocolTime instant;
            ProtocolDuration period;
            octet precision;       // precision
      };

      interface Transport {
            readonly attribute ProtocolId id;
            readonly attribute ProfileId tag;
            readonly attribute OCI::Handle handle;
            void close();
            void shutdown();
            void receive(in Buffer buf, in boolean block);
            boolean receive_detect(in Buffer buf, in boolean
            block);
            void receive_timeout(in Buffer buf, in unsigned long
            timeout);
            void send(in Buffer buf, in boolean block);
            boolean send_detect(in Buffer buf, in boolean block);
            void send_timeout(in Buffer buf, in unsigned long
            timeout);
            TransportInfo get_info();

            // protocol time
            attribute readonly ProtocolInstant protocol_time;
            ProtocolInstant protocol_period( in octet messageID );

      };
};
```

The meaning of the data types and operations is the same as in the ETF submission case.

ASKING THE TIME FROM A CORBA APPLICATION

With the extensions for the Extensible Transport Framework it is possible for the ORB to request to the underlying communications layer the time in which a certain message must be written or read to/from the network. But the ORB is aware of neither the purpose or the messages handled by the application on top of it. The ORB cannot be laden with the task of actively informing the application about timing information. This task should be initiated by the application. In real-time CORBA, the **CORBA::Object** can be configured to used a certain set of communications protocols. This is called protocol configuration in real-time CORBA. At the client-side the policies for protocol configuration can be applied at the object level. This means that it is possible to establish a relationship between instances of protocols and objects in the client side of a real-time CORBA application. It then necessary to extend the object interface to let the application ask for the time.

As the field of application for timing operations is vertical rather than horizontal it is not a good idea to extend the **CORBA::Object** interface. Instead of this, it is better to extend the functionality of the object in the RTCORBA module so non real-time applications do not have the additional operations regarding handling of time.

Using pseudo-IDL the object interface can be extended in real-time CORBA as follows:

```
module RTCORBA{

        local interface RTObject{
                // protocol time
                attribute readonly ETF::ProtocolInstant protocol_time;
                ETF::ProtocolInstant protocol_period(
                                                        in octet messageID
                                                        );
        };

};
```

In the case of the RTObject the time type appears as ETF::ProtocolInstant, As the ETF also depends on Real-Time CORBA for the declaration of protocol policies it is better if the ProtocolInstant type is defined in the RTCORBA module. In the case of the ETF or the OCI the declaration of the time types should be removed from the IDL specification and an **#include "RTCORBA.idl"** directive should be used at the beginning of the file. The IDL follows for the RTCORBA module:

```
module RTCORBA{
      // declaration of time types
      long long ProtocolTime;       // a time instant
      long long ProtocolDuration;   // a duration
      struct ProtocolInstant{       // current time and period
            octet messageID;
            ProtocolTime instant;
            ProtocolDuration period;
            octet precision;        // precision
      };

      local interface RTObject{
            // protocol time
            attribute readonly ProtocolInstant protocol_time;
            ProtocolInstant protocol_period(
                                                in octet messageID
                                                );
      };

};
```

The **RTCORBA::RTObject** interface is not derived from the **CORBA::Object** interface as we are using pseudo IDL for which inheritance is not defined. Nevertheless, **RTObject** is conceptually an extension of the **CORBA::Object** interface. There is a single instance of **RTCORBA::RTObject** per instance of **CORBA::Object**. Notice that the interface extension has been declared to be local. The extension of the interface adds functionality only for handling of time purposes and this information shall only be valid for a given node.

LIFE AS A RTObject

Narrowing to a RTObject is not free, there are some rules when an object reference is a reference to a RTObject.

- RTObject instances are client-side objects.
- All CORBA::Object instances may become RTObject instances.
- A RTObject instance cannot be passed as a parameter of an IDL operation nor can it be stringified. Any attempt to do so shall return in a **MARSHAL** system exception with a Standard Minor Exception Code of 4 (attempt to marshal a local object).
- Once an object is narrowed to an RTObject it remains to be an RTObject even if it narrowed to other types. If the object is narrowed to a CORBA::Object, it is no longer a RTObject.

Making an RTObject a local object helps resolve the problem of implementing **_narrow()** and **_is_a()**. Remember that all CORBA clients

are based in the **CORBA::Object** interface. Although we can represent a client object as an **RTObject**, the server side will only be an **Object**. However, this artifact allows us to know the time for the pair object / protocol without modifying the existing CORBA specification.

SETTING A DEADLINE

Deadline support can be related to timeout support in CORBA. Real-Time CORBA uses the CORBA **Messaging::RelativeRoundtripTimeoutPolicy** to allow a timeout to be set for the receipt of a reply to an invocation. But for Real-Time CORBA the timeout policy is only used where it is set, on the client side. This means that the policy is not propagated with the request, making it impossible for the transport protocol plug-in or the ORB at the server side to decide on what request to execute first (it can only use request priorities). Only value information or priority information is transmitted, temporal information does not travel with the request. The relative roundtrip timeout policy interface is as follows:

```
const CORBA::PolicyType
        RELATIVE_RT_TIMEOUT_POLICY_TYPE = 32;
local interface RelativeRoundtripTimeoutPolicy : CORBA::Policy {
        readonly attribute TimeBase::TimeT relative_expiry;
};
```

Fortunately, the messaging specification of CORBA allows the policy to be propagated with the request within a **PolicyValue** in an **INVOCATION_POLICIES** service context. The **pType** of the **PolicyValue** has the value **REPLY_END_TIME_POLICY_TYPE** and the **pValue** is a CDR encapsulation containing the **relative_expiry** converted into a **TimeBase::TtcT** end time.

The messaging specification allows for other types of request and reply timeout and deadlines: **RequestStartTimePolicy**, **RequestEndTimePolicy**, **ReplyStartTimePolicy**, **ReplyEndTimePolicy** and **RelativeRequestTimeoutPolicy**. All these policies allow the application to specify a series of deadlines and timeouts in which requests and replies should happen. These policies are suitable for hard real-time applications except for the granularity of time used. Lifetime of requests and replies is specified in terms of structures from the CORBA Time Service Specification. Time is described as a 64-bit value which is the number of 100 nanoseconds from 15 October 1582 00:00 along with innacuracy and time zone information. This poses the problem that the precision of the global time used by the ORB in previous sections is up to 60 ns.

A HRTC ORB can decide either to implement only the Messaging specification reply and request timeout policies with a time granularity of 100 ns or to implement the following additional interfaces in module RTCORBA to decrease granularity to 60 ns. Instead of using the CORBA Time Service representation of time, hard real-time CORBA should use the same time representation of the network layer. Real-Time request lifetime policies can be defined in module RTCORBA as :

```
interface RT[Request/Reply][Start/end]TimePolicy{
        readonly attribute long long [start/end]_time;
};
```

or

```
interface RTRelative[Request/Roundtrip]TimeoutPolicy{
        readonly attribute long long relative_expiry:
};
```

and the RTORB shall provide methods for creating each of the policy types:

```
module RTCORBA{
        interface RTORB{
                RT[Request/Reply][Start/end]TimePolicy create_
        RT[Request/Reply][Start/end]TimePolicy(
                 in long long [start/end]_time
                );

        RTRelative[Request/Roundtrip]TimeoutPolicy create_
RTRelative[Request/Roundtrip]TimeoutPolicy( in long long
relative_expiry);

        };
};
```

It is important to notice that the time / timeout values handled by the invocation lifetime policies can be used by the ORB in order to schedule requests. They can also be used by the Extensible Transport Framework as the ORB will select either the methods of the ETF submission:

```
void write(in boolean isFirst, in boolean isLast, in Buffer data,
            in unsigned long offset, in unsigned long length,
            in TimeBase::TimeT time_out);
```

void read(inout Buffer data, in unsigned long offset,
in unsigned long min_length, in unsigned long max_length,
in TimeBase::TimeT time_out);

or the methods of the OCI:

void receive_timeout(in Buffer buf, in unsigned long timeout);
void send_timeout(in Buffer buf, in unsigned long timeout);

The Time Service Specification **defines TimeBase::TimeT** as:

typedef unsigned long long TimeT;

For the sake of clarity in the ETF, the signature of the operations shall read **unsigned long long** instead of **TimeT** just to make it clear that UTC time is not being used. For the OCI, the timeout argument should be changed to **unsigned long long**.

It should also be noticed that while the client part of the pluggable protocol plug-in is time-aware, the server part at this level is not. This is so because the time / timeout policy is embedded in a service context list in the GIOP message which is not accessible from the transport level. It is matter of the implementation to provide means to forward timing information to servers at the plug-in level.

REQUEST TIMESTAMPING

An **RTObject** can be requested to timestamp requests on arrival. This allows the application to have not only value information. The instant of arrival of a reply to a request can be known.

An **RTObject** must be instructed to timestamp replies to requests so the overhead of timestamping can be avoided if it is not necessary. For this a **TimestampPolicy** must be created by the RTORB.

```
module RTCORBA{
        local interface TimestampPolicy:CORBA::Policy{

        };
        interface RTORB{
                TimestampPolicy create_timestamp_policy();
        };
};
```

The **TimestampPolicy** can be used to configure a client-side real-time object via the CORBA::Object **set_policy_overrides** operation. The policy is applied only at the client side. After setting the policy subsequent invocations by the client object will be timestamped by the ORB. An object only has access to the timestamp of the last invocation. For this, an operation is provided in the **RTObject** interface.

```
module RTCORBA{
      // declaration of time types
      long long ProtocolTime;       // a time instant
      long long ProtocolDuration;   // a duration
      struct ProtocolInstant{       // current time and period
            octet messageID;
            ProtocolTime instant;
            ProtocolDuration period;
            octet precision;        // precision
      };

      local interface RTObject{
            // protocol time
            attribute readonly ProtocolInstant protocol_time;
            ProtocolInstant protocol_period(
                                              in octet messageID
                                              );
            ProtocolInstant time_stamp();
      };

};
```

The timestamp operation shall be called just after an invocation has been issued. Attempt to invoke time_stamp without setting the policy for the object shall result in a **INV_POLICY** system exception with **MINOR_CODE** of 1.

7.3 HRTC protocol properties

Protocol properties as specified by real-time CORBA can be described in IDL for HRTC protocols. The following is the IDL for HRTCProtocolProperties.

```
local interface HRTCProtocolProperties{
      attribute long min_delay; // minimum delay in microseconds
      attribute long avrg_delay; // average delay in microseconds
      attribute long max_delay; // maximum delay in microseconds
      attribute double packet_loss; // packet loss as probability
```

};

7.4 C++ source code example

The following excerpt of code shows how the client side of an application is able to ask the time associated to an specific network protocol.

```
// build factory object for the new transport
//(inherited from OCI factory classes)
// only connector is needed at the client side
HRTCConFactory ConFactory _HRTC;

// Get a reference to the RTORB
CORBA::Object_var obj = orb -> resolve_initial_references( "RTORB"
);
RTCORBA::RTORB_var rtorb = RTCORBA::RTORB::_narrow( obj );

// Get a reference to the OCI ConFactoryRegistry
CORBA::Object_var obj2 = orb -> resolve_initial_references(
"OCIConFactoryRegistry" );
      OCI::ConFactoryRegistry_var ConFactReg =
OCI::ConFactoryRegistry::_narrow( obj2 );

// Add an acceptor factory object
 ConFactReg ->add_factory(ConFactory _HRTC);

// Resolve the objects in the Naming Service
CORBA::Object_ptr nm = rtorb->
resolve_initial_references("NameService");

CosNaming::NamingContext_var nc =
            CosNaming::NamingContext::_narrow( nm );

// resolve the object by its name
CosNaming::Name name_rtObj;
name_rtObj.length(1);

name_rtObj [0].id = CORBA::string_dup( "RT_OBJECT" );
name_rtObj [0].kind = CORBA::string_dup( "" );

CORBA::Object_var object = nc ->resolve(name_rtObj );
RTCORBA::RTObject rt_object = RTCORBA::RTObject::_narrow(object);
RTObjectTest_var rt_test_obj = RTObjectTest::_narrow( rt_object );

// set an object protocol configuration policy override
HRTCProtocolProperties props; // assigned by default
RTCORBA::ProtocolList Prot_list;
Prot_list.length(0);
Prot_list[0].protocol_type = TAG_HRTC;
Prot_list[0].transport_protocol_propeerties = props;
// Prot_list[0].orb_protocol_propeerties - defaults to GIOP
RTCORBA::ClientProtocolPolicy_var clPol = rtorb->
create_Client_protocol_policy(Prot_list);
```

```
// assign the protocol policy to an object
// after this the object uses the HRTC protocol
rt_test_obj->set_policy_overrides( clPol );

// to ask for the time a connection must be validated
PolicyList_var incosistent_policies;
rt_test_obj->validate_connection(inconsistent_policies);

// get the period
RTCORBA::ProtocolInstant_var prot_period;
prot_period = rt_test_obj->protocol_period( SPEED_MGS_ID );

// get the time
RTCORBA::ProtocolInstant_var prot_time;
Prot_time = rt_test_obj->protocol_time();
```

Notice that for the **protocol_period** operation the argument `SPEED_MSG_ID` is passed. This supposes that application handles a message to read / write a state variable that measures the speed of an element of the system.

Appendix A: ETF module IDL

```
#include "orb.idl"
#include "IOP.idl"
#include "GIOP.idl"
#include "RTCORBA.idl"
#include "TimeBase.idl"


module ETF
{
      typedef sequence<octet> Buffer;

      // locality constrained
      local interface Profile
      {
            void marshal(inout IOP::TaggedProfile tagged_profile,
                         ionout IOP::TaggedComponentSeq
components);
                  // marshal() must set data.profile_data.length()
                  // to the index of the last octet marshalled + 1

            unsigned long hash();

            Profile copy();

            boolean is_equivalent(in Profile prof);


            readonly attribute GIOP::Version version;
      };


      // locality constrained
      local interface Connection
      {
            void write(in boolean isFirst,
                       in boolean isLast,
                       in Buffer data,
                           in unsigned long offset,
                           in unsigned long length,
                           in TimeBase::TimeT time_out);

            void read(inout Buffer data,
                      in unsigned long offset,
                          in unsigned long min_length,
                          in unsigned long max_length,
                          in TimeBase::TimeT time_out);
                  // transport needs to set data.length() to
                  // offset + number of bytes actually read

            void flush();
```

```
        void connect(in Profile server_profile,
                     in TimeBase::TimeT time_out);

        void close();

        boolean is_connected();

        Profile get_server_profile();

        boolean is_data_available();

        boolean wait_next_data(in TimeBase::TimeT time_out);

        readonly attribute long id;
        readonly attribute boolean supports_callback;
        readonly attribute boolean use_handle_time_out;
};


// locality constrained
local interface Handle
{
        boolean add_input(in Connection con);
        // tells the ORB that a new connection has come in
        // ORB returns false if it rejects new connection

        void signal_data_available(in Connection conn);

        void close_by_peer(in Connection conn);
};

// locality constrained
local interface Listener
{
        void set_handle(in Handle up);

        Connection accept();

        void destroy();

        void completed_data(in Connection conn);

        boolean is_data_available(in Connection conn);

        readonly attribute Profile endpoint;
};


// locality constrained
local interface Factories
{
        Connection
         create_connection(in RTCORBA::ProtocolProperties
props);

        Listener
```

```
                create_listener(in RTCORBA::ProtocolProperties
        props);

                Profile
                 demarshal_profile(inout IOP::TaggedProfile
                tagged_profile,
                                    out IOP::TaggedComponentSeq
components);

                readonly attribute IOP::ProfileId profile_tag;
        };


        // Optional zero copy connection interface
        // locality constrained
        local interface BufferList
        {
                unsigned long add_buffer(in unsigned long size,
                                         inout Buffer buf);
                // adds an additional buffer to the list.
                // returns the zero-origin index of the added buffer.
                // buf.length() should be set to the actual size of
the
                // memory allocated whether more or less than size

                readonly attribute unsigned long num_buffers;

                void get_buffer(in unsigned long index,
                                inout buffer buf);
                // populates the buf argument with the pointer to the
data
        };


        local interface ConnectionZeroCopy : Connection
        {
                BufferList  create_buffer_list();

                void write_zc(inout BufferList data,
                              in TimeBase::TimeT time_out);

                void readZC(inout BufferList data,
                            in unsigned long min_length,
                            in TimeBase::TimeT time_out);

        };
};
```

Appendix B: OCI module IDL

```
module OCI
{

interface TransportInfo;

interface ConnectorInfo;

interface AcceptorInfo;

interface AccFactoryInfo;

interface ConFactoryInfo;

interface Transport
{
    readonly attribute ProtocolId id;

    readonly attribute ProfileId tag;

    readonly attribute OCI::Handle handle;

    void close();

    void shutdown();

    void receive(in Buffer buf, in boolean block);

    boolean receive_detect(in Buffer buf, in boolean block);

    void receive_timeout(in Buffer buf, in unsigned long timeout);

    void send(in Buffer buf, in boolean block);

    boolean send_detect(in Buffer buf, in boolean block);

    void send_timeout(in Buffer buf, in unsigned long timeout);

    TransportInfo get_info();
};

interface CloseCB;

interface TransportInfo
{
    readonly attribute ProtocolId id;

    readonly attribute ProfileId tag;

    readonly attribute ConnectorInfo connector_info;

    readonly attribute AcceptorInfo acceptor_info;

    string describe();
```

```
    void add_close_cb(in CloseCB cb);

    void remove_close_cb(in CloseCB cb);
};


interface CloseCB
{
    void close_cb(in TransportInfo transport_info);
};

typedef sequence< CloseCB > CloseCBSeq;


interface Connector
{
    readonly attribute ProtocolId id;

    readonly attribute ProfileId tag;

    Transport connect();

    Transport connect_timeout(in unsigned long timeout);

    ProfileInfoSeq get_usable_profiles(in IOR ref,
                                in CORBA::PolicyList policies);

    boolean equal(in Connector con);

    ConnectorInfo get_info();
};

typedef sequence< Connector > ConnectorSeq;

interface ConnectCB;

interface ConnectorInfo
{
    readonly attribute ProtocolId id;

    readonly attribute ProfileId tag;

    string describe();

    void add_connect_cb(in ConnectCB cb);

    void remove_connect_cb(in ConnectCB cb);
};

interface ConnectCB
{
    void connect_cb(in TransportInfo transport_info);
};

typedef sequence< ConnectCB > ConnectCBSeq;
```

```
interface Acceptor
{
    readonly attribute ProtocolId id;

    readonly attribute ProfileId tag;

    readonly attribute OCI::Handle handle;

    void close();

    void listen();

    Transport accept(in boolean block);

    Transport connect_self();

    void add_profiles(in ProfileInfo profile_info, inout IOR ref);

    ProfileInfoSeq get_local_profiles(in IOR ref);

    AcceptorInfo get_info();
};

typedef sequence< Acceptor > AcceptorSeq;

interface AcceptCB;

interface AcceptorInfo
{
    readonly attribute ProtocolId id;

    readonly attribute ProfileId tag;

    string describe();

    void add_accept_cb(in AcceptCB cb);

    void remove_accept_cb(in AcceptCB cb);
};

interface AcceptCB
{
    void accept_cb(in TransportInfo transport_info);
};

typedef sequence< AcceptCB > AcceptCBSeq;

exception InvalidParam
{
    Param p;
    string reason;
};

interface AccFactory
{
```

```
        readonly attribute ProtocolId id;

        readonly attribute ProfileId tag;

        Acceptor create_acceptor(in ParamSeq params)
            raises(InvalidParam);

        AccFactoryInfo get_info();
    };

    typedef sequence< AccFactory > AccFactorySeq;

    interface AccFactoryInfo
    {
        readonly attribute ProtocolId id;

        readonly attribute ProfileId tag;

        string describe();
    };

    exception FactoryAlreadyExists
    {
        ProtocolId id;
    };

    exception NoSuchFactory
    {
        ProtocolId id;
    };

    interface AccFactoryRegistry
    {
        void add_factory(in AccFactory _factory)
            raises(FactoryAlreadyExists);

        AccFactory get_factory(in ProtocolId id)
            raises(NoSuchFactory);

        AccFactorySeq get_factories();
    };

    interface ConFactory
    {
        readonly attribute ProtocolId id;

        readonly attribute ProfileId tag;

        ConnectorSeq create_connectors(in IOR ref,
                                       in CORBA::PolicyList policies);

        boolean equivalent(in IOR ior1, in IOR ior2);

        unsigned long hash(in IOR ref, in unsigned long maximum);

        ConFactoryInfo get_info();
```

```
};

typedef sequence< ConFactory > ConFactorySeq;

interface ConFactoryInfo
{
    readonly attribute ProtocolId id;

    readonly attribute ProfileId tag;

    string describe();

    void add_connect_cb(in ConnectCB cb);

    void remove_connect_cb(in ConnectCB cb);
};

interface ConFactoryRegistry
{
    void add_factory(in ConFactory _factory)
   raises(FactoryAlreadyExists);

   ConFactory get_factory(in ProtocolId id)
              raises(NoSuchFactory);

   ConFactorySeq get_factories();
};

interface Current : CORBA::Current
{
    TransportInfo get_oci_transport_info();

    AcceptorInfo get_oci_acceptor_info();
};

}; // End module OCI
```

Appendix C: Messaging timeout policies

```
Module Messaging{

      // Timeout Policies
      const CORBA::PolicyType
      REQUEST_START_TIME_POLICY_TYPE = 27;
      local interface RequestStartTimePolicy : CORBA::Policy {
      readonly attribute TimeBase::UtcT start_time;
      };

      const CORBA::PolicyType REQUEST_END_TIME_POLICY_TYPE = 28;
      local interface RequestEndTimePolicy : CORBA::Policy {
      readonly attribute TimeBase::UtcT end_time;
      };

      const CORBA::PolicyType REPLY_START_TIME_POLICY_TYPE = 29;
      local interface ReplyStartTimePolicy : CORBA::Policy {
      readonly attribute TimeBase::UtcT start_time;
      };

      const CORBA::PolicyType REPLY_END_TIME_POLICY_TYPE = 30;
      local interface ReplyEndTimePolicy : CORBA::Policy {
      readonly attribute TimeBase::UtcT end_time;
      };

      const CORBA::PolicyType
      RELATIVE_REQ_TIMEOUT_POLICY_TYPE = 31;
      local interface RelativeRequestTimeoutPolicy : CORBA::Policy
      {
      readonly attribute TimeBase::TimeT relative_expiry;
      };

      const CORBA::PolicyType
      RELATIVE_RT_TIMEOUT_POLICY_TYPE = 32;
      local interface RelativeRoundtripTimeoutPolicy :
      CORBA::Policy {
      readonly attribute TimeBase::TimeT relative_expiry;
```

```
        };
    };
```