Portable Interceptors

Joint Initial Submission

Inprise Corporation
BEA Systems, Inc.

With support and collaboration from:
Highlander Communications, L.C.

OMG TC Document orbos/99-04-05

Table of Contents

Preface 1

1.1 Cosubmitting Companies

The following companies are pleased to jointly submit this specification in response to the OMG Portable Interceptors RFP (Document orbos/98-09-11):

- Inprise Corporation
- BEA Systems

Supporting companies are:

- Highlander Communications, L.C.

1.2 Status of this document

This document is an initial submission produced for the May, 1999 OMG Technical Committee meeting in Tokyo.

1.3 Guide to the Submission

Chapter 1 provides contact information and a guide to this submission.

Chapter 2 is the proof of concept statement.

Chapter 3 explains how this submission satisfies the requirements of the RFP.

Chapter 4 provides some of the rationale for this submission.

Chapter 5 describes the model and semantics of portable interceptors as specified in this submission.

Chapter 6 specifies the conformance requirements.

Chapter 7 specifies changes to CORBA 2.3.

1.4 Missing Items

There are no missing items.

1.5 Conventions

IDL appears using this font.

`Concrete programming language (Java, C++, etc.) code appears using this font.`

Please note that any change bars have no semantic meaning. They are present for the convenience of readers and submitters (and the editor who wants to be able to tell what changed between various drafts). In this document, they are purely leftovers from the internal editing process and do not indicate changes from previously published versions.

1.6 Submission Contact Points

Jeff Mischkinsky
Inprise Corporation
951 Mariner's Island Blvd.
San Mateo, CA 94404
USA
phone: +1 650 358 3049
email: jeffm@inprise.com

Dan Frantz
BEA Systems, Inc.
436 Amherst Street
Nashua, NH 03063
USA
phone: +1 603 579 2519
email: dan.frantz@beasys.com

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

Proof of Concept 2

This submission is based on the design and implementation work for the next release of VisiBroker.

Response to RFP Requirements 3

The following lists the requirements from the Portable Interceptors RFP (orbos/98-09-11) and describes how this submission addresses them.

3.1 Scope

Proposals responding to this RFP shall provide a ***portable definition of interceptors,*** *so that system services and users may "plug into" ORB processing at particular points.*

A detailed explanation of how the requirement is met will be provided in a future revised submission.

The system interceptors must be capable of being used with Security and Transactions. System interceptors are for the benefit of the ORB and service providers and are not visible to user programmers.

A detailed explanation of how the requirement is met will be provided in a future revised submission.

CORBA 2.2 specifies two types of ***request-related interceptors****: request- level, using structured requests, as in DII/DSI operations, and message- level, dealing with buffers of information, eventually resulting in transport processing. For this RFP, the submission's request-level interceptors may continue the CORBA 2.2 usage of DII-related and DSI-related structures or may use other means to provide access to request information.*

A detailed explanation of how the requirement is met will be provided in a future revised submission.

It is desirable that any solution for system interceptors for Security and Transactions be general enough that it might be usable for other ORB services.

A detailed explanation of how the requirement is met will be provided in a future revised submission.

Solutions shall also provide user level interceptors. No specific type of user interceptor is required. It is up to submitters to suggest the points at which hooking into the ORB might be useful. One example that has been suggested is pre- and post-method invocation.

A detailed explanation of how the requirement is met will be provided in a future revised submission.

3.2 Mandatory Requirements

A submission shall specify two kinds of interceptors: system and user.

A detailed explanation of how the requirement is met will be provided in a future revised submission.

A submission shall provide a complete architectural model of interceptors in ORB and Object Adapter processing. This includes what flexibility an ORB has in implementing interceptors, how the ORB will handle errors in interceptor processing, and the legitimate actions of an interceptor (including what kind of invocations an interceptor can perform). A submission shall describe under what conditions and how the ORB deals with recursive calls, whether generated locally or remotely. Allowable flexibility, error handling, and actions may be different for different kinds of interceptors.

A detailed explanation of how the requirement is met will be provided in a future revised submission.

System interceptors for ORB Services shall include Request-Related and IOR-management interceptors. For these system interceptors, a submission shall provide interfaces so that an interceptor can set and query GIOP service contexts (for request-related system interceptors) and IOR profile information (for IOR-management system interceptors). The submission shall specify methods of limiting the invocation of such operations to system interceptors.

A detailed explanation of how the requirement is met will be provided in a future revised submission.

*A submission shall specify a mechanism for **user interceptors** at request-level and at additional points in ORB processing. The mechanism need not be the same as system interceptors at the request-level.*

A detailed explanation of how the requirement is met will be provided in a future revised submission.

A submission shall provide for multiple interceptors (system and user) to be called at each point they are applicable and shall specify how the multiple interceptors are called (for example, each called serially by the ORB or daisy-chained).

A detailed explanation of how the requirement is met will be provided in a future revised submission.

A submission's system interceptor model shall be detailed enough that Security and Transactions services could be reasonably implemented using it. An analysis of how each of these services could use system interceptors is required. The analysis will not be normative for those services, but must be detailed enough to show the applicability of the model. The analysis shall include a specification of what information a Security or Transaction Service needs at various points and how the system interceptor model provides that information.

A detailed explanation of how the requirement is met will be provided in a future revised submission.

To meet the negotiation part of certain security mechanisms, a submission shall specify that at least some system, request-related interceptors must be capable of holding a request in abeyance while it performs communication of its own, perhaps at a lower protocol level. For example, a client-end, message-level, security interceptor might do handshaking with a target before issuing a request using the negotiated security.

A detailed explanation of how the requirement is met will be provided in a future revised submission.

A submission shall specify administrative interfaces including, at a minimum, registering and unregistering interceptors, both system and user.

A detailed explanation of how the requirement is met will be provided in a future revised submission.

3.3 Optional Requirements

There were no optional requirements identified in the RFP.

3.4 Issues to be discussed

Submissions shall discuss the security of the interceptor architecture, that is, who is allowed to add system interceptors to the ORB and under what conditions. For system interceptors, this might state that adding interceptors is controlled administratively. For user level interceptors, the submission shall discuss what considerations are needed to protect the ORB and the service contexts of a request.

A detailed explanation of how the requirement is met will be provided in a future revised submission.

By the time the first submissions are due, it is likely that the submissions for the ORB-related Firewall and Messaging RFPs will have reached final approval, as will have several modifications to GIOP. These changes may add new CORBA Core and GIOP capabilities not present when CORBA 2.2 Interceptors were first specified. If so, submissions should take the new specifications into account. For example,

- *The Messaging RFP introduces a new programming invocation model, asynchronous invocation with the reply being a call-back invocation directed to another process or machine. How does this affect the presumed symmetry of interceptors in CORBA 2.2?*

A detailed explanation of how the requirement is met will be provided in a future revised submission.

- *The Firewall RFP and the Interop RTFs may allow redirection of a request to an address different from that bound to originally. Does this make a difference?*

A detailed explanation of how the requirement is met will be provided in a future revised submission.

Overall Design Rationale 4

This chapter discusses some of the rationale behind the choices that were made for this mapping.

This information will be provided in a future revised submission.

Portable Interceptors 5

5.1 Introduction

The Portable Interceptor module defines a set of API hooks known as interceptors which provide a way for plugging in additional ORB behavior such as support for transactions and security.

This specification outlines a generic framework which may be augmented in the future to add new interceptor types as required. The specification defines several forms of interceptors including:

- Request Interceptors - These are system level interceptors which can be used to manipulate service contexts and examine other request level data before it is transmitted.
- Message Interceptors - These are system level interceptors which can be used to modify messages as they are sent and received by the ORB.
- ORB Interceptors - These are system level interceptors which can be used to intercept certain ORB operations and cause other interceptors to be installed.
- POA Interceptors - These are system level interceptors which can be used to intercept certain POA operations and cause other interceptors to be installed.
- User Interceptors - This submission does not include a specification for user level interceptors, but will include one in a revised submission. User level interceptors are similar to request level interceptors but are not able to manipulate service contexts or other protocol related constructs.

In addition to defining the interceptors it is also necessary to introduce a set of user APIs to manipulate the data structures which define ORB interoperability. Since these APIs are used by the interceptor APIs, they are introduced first.

Interceptors are installed and managed by a set of interfaces which is also described in this document.

5.2 Interoperability APIs

The core data structures as defined in the CORBA IOP module define the standard set of data structures which define interoperability between ORBs. This interoperability requirement was to support communication between ORBs, and not communication between an ORB and an ORB interceptor programmer. Hence a new set of APIs is defined to provide simpler interaction between the ORB and interceptor programmer. An additional set of interfaces and valuetypes in the IOP module comprises the core set of APIs and is defined as follows:

```
module IOP {

    typedef unsigned long ProfileId;

    struct TaggedProfile {
        ProfileId tag;
        sequence<octet>profile_data;
    };

    abstract valuetype ProfileValue {
        readonly attribute ProfileId tag;
        TaggedProfile toTaggedProfile();
    };

    valuetype UnknownProfile : ProfileValue {
        CORBA::OctetSequence getProfileData();
    };

    interface ProfileValueFactory {
        ProfileValue create(in TaggedProfile profile);
    };

    struct IOR {
        string type_id;
        sequence<TaggedProfile>profiles;
    };

    valuetype IORValue {
        public string type_id;
        public sequence<ProfileValue> profiles;
        IOR toIOR();
    };

    typedef unsigned longComponentId;

    struct TaggedComponent {
        ComponentIdtag;
        sequence<octet>component_data;
    };
```

```
abstract valuetype ComponentValue {
    readonly attribute ComponentId tag;
    TaggedComponent toTaggedComponent();
};

valuetype UnknownComponent : ComponentValue {
    CORBA::OctetSequence getComponentData();
};

interface ComponentValueFactory {
    ComponentValue create(in TaggedComponent component);
};

typedef unsigned longServiceID;

struct ServiceContext {
    ServiceID context_id;
    CORBA::OctetSequencecontext_data;
};

abstract valuetype ServiceValue {
    readonly attribute ServiceID context_id;
    ServiceContext toServiceContext();
};

typedef sequence<ServiceValue> ServiceValueList;

valuetype UnknownService {
    CORBA::OctetSequence getContextData();
};

interface ServiceValueFactory {
    ServiceValue create(in ServiceContext ctx);
};

typedef sequence<ServiceContext> ServiceContextList;

interface EntityRegistry {
    void registerProfileFactory(in ProfileId tag,
                                in ProfileValueFactory factory);
    void registerComponentFactory(in ComponentId tag,
                                  in ComponentValueFactory factory);
    void registerServiceFactory(in ServiceID tag,
                                  in ServiceValueFactory factory);

    ProfileValueFactory getProfileValueFactory(in ProfileId tag);
    ComponentValueFactory getComponentFactory(
                                          in ComponentId tag);
    ServiceValueFactory getServiceFactory(in ServiceID tag);
};
```

```
};
```

5.2.1 IOR manipulation

IORs are manipulated using the **IORValue**, **ProfileValue**, and **ComponentValue** valuetypes. These valuetypes simply provide an abstraction of the existing data structures defined in IDL, and provide methods to map from the abstraction to the on-the-wire format.

IORValues are created automatically by the ORB before being passed to interceptor calls. Typically the ORB will create an **IORvalue** immediately after receiving the IOR on the wire. For an **IORValue** to be constructed it will need to construct the appropriate **ProfileValue** components, which in turn may need to construct **ComponentValue**s. To support the creation of **ProfileValue**s and **ComponentValue**s, a factory must be installed for each profile or component tag supported. The factory may be directly implemented by the ORB or the end-user, but must be installed in the **EntityRegistry**.

5.2.2 Service Context manipulation

Service contexts are manipulated using the **ServiceValue** valuetype. Just as with IOR manipulation, factories must be created and installed so that the ORB can create Service valuetypes from their marshalled state.

5.2.3 Entity Registry

The entity registry is responsible for registering and returning **ProfileValueFactory**s, **ComponentValueFactory**s, and **ServiceValueFactory**s. A maximum of one factory may be installed for each “tag” associated with a profile, component, or service context. The entity registry may be obtained by invoking **ORB.resolve_initial_references()** with the string “IOPEntityRegistry” as an argument. If an attempt is made to retrieve a factory for an unknown tag, the EntityRegistry will always return a factory capable of creating **UnkownProfileValue**s, **UnknownComponentValue**s, or **UnknownServiceValue**s.

5.2.4 GIOP specific APIs

The APIs defined above are truly generic in that they only encompass those data structures defined in the CORBA IOP module. Additional APIs are required for specialization within the GIOP set of on-the-wire protocols. In particular all GIOP protocols must support the notion of an opaque object key and the GIOP protocol version to be used for client-server communication must be specified in the IOR. These concepts are capture in the **GIOP::ProfileBodyValue** and **GIOP::ObjectKey** valuetypes which are defined in IDL below.

The ProfileBodyValue extends the ProfileValue and adds to GIOP version and object key to the information model required for all GIOP based protocols. The ObjectKey class provides an opaque view of the ObjectKey. In addition, subclasses of the ObjectKey valuetype may be provided by a particular ORB vendor if desired.

```
module GIOP {
    struct Version {
        octet major;
        octet minor;
    };

    abstract valuetype ObjectKey {
        CORBA::OctetSequence toOctetSequence();
    };

    valuetype ProfileBodyValue : IOP::ProfileValue {
        public GIOP::Version version;
        public ObjectKey object_key;
    };

};
```

5.2.5 Protocol Specific APIs

Each specific protocol (i.e., IIOP) will typically be described by a specified ProfileBody format. In order to manipulate the protocol specific portions of the ProfileBody a subtype of IOP::ProfileValue must be specified for each a defined ProfileBody format. The revised version of this submission will include such definitions for all protocols defined in the CORBA specification, including IIOP.

5.3 Interceptor Management

Interceptors are installed by invoking an operation on an **InterceptorManager**. An **InterceptorManager** is defined for each type of Interceptor. An instance of an **InterceptorManager** may be obtained by invoking an operation on an **InterceptorManagerControl** interface. The IDL is defined as follows.

```
module PortableInterceptor {

    interface InterceptorManager {
    };

    interface InterceptorManagerControl {
        InterceptorManager get_manager(in string name);
    };
};
```

Both the InterceptorManager and InterceptorManagerControl interface and any subtypes are locality-constrained objects and exhibit the same behavior as all other locality constrained objects defined in the CORBA specification.

The InterceptorManager interface provides no operations and is typically subtyped as needed for each class of interceptor defined in the system. An **IntercpetorManager** manages a set of interceptors which are typically installed in a chain.

The InterceptorManagerControl interface is used to obtain an instance of a particular InterceptorManager. The instance to be returned is specified by the name parameter and is specific to a particular instance of an InterceptorManagerControl object. For example, both the ORB and the POA provide InterceptorManagerControl objects to add interceptors to either the ORB or the POA. Though the set of available **InterceptorManager**s and their names vary depending on the InterceptorManagerControl.

5.3.1 ORB InterceptorManagerControl

One instance of the InterceptorManagerControl is the ORB InterceptorManagerControl which is available by invoking **ORB::resolve_initial_references("InterceptorManagerControl")**. The ORB interceptor manager control is used to install many of the global ORB interceptors.

5.4 Request Interceptors

Request interceptors are provided to allow a service provider to specify a "hook" which the ORB will invoke during its processing of a request to provide a service.

```
module PortableInterceptor {

    native Cookie;

    interface ClientRequestInterceptor {
        void preinvoke(in Object target,
                       in CORBA::Identifier operation,
                       inout IOP::ServiceValueList service_contexts,
                       out Cookie the_cookie);

        void postinvoke(in Object target,
                       in IOP::ServiceValueList service_contexts,
                       in CORBA::Environment env,
                       in Cookie the_cookie);

        void exception_occurred(in Object target,
                                in CORBA::Environment env,
                                in Cookie the_cookie);
    }
```

```
        interface ClientRequestInterceptorManager : InterceptorManager {
            void add(in ClientRequestInterceptor interceptor);
        };

        interface ServerRequestInterceptor {
            void preinvoke(in Object target,
                           in CORBA::Identifier operation,
                           in IOP::ServiceValueList service_contexts,
                           out Cookie the_cookie);

            void postinvoke(in Object target,
                           inout IOP::ServiceValueList service_contexts,
                           in CORBA::Environment env,
                           in Cookie the_cookie);

            void exception_occurred(in Object target,
                                    in CORBA::Environment env,
                                    in Cookie the_cookie);
        };

        interface ServerRequestInterceptorManager : InterceptorManager {
            void add(in ServerRequestInterceptor interceptor);
        };
    };
```

There are two interfaces which define two different types of request interceptors: the ClientRequestInterceptor and the ServerRequestInterceptor. Both interfaces contain nearly equivalent operations, but differ in the way they handle ServiceValue manipulation. Both interceptor types are managed by their corresponding InterceptorManager. Each request interceptor manager contains a single add operation which can be used to add a new instance of an interceptor to the chain of already installed interceptors.

5.4.1 ClientRequestInterceptor

The ClientRequestInterceptor contains two methods which are invoked during normal processing of requests and an additional operation for reporting exceptions which occurred during the processing of other interceptors. All operation invocations called at the “intercept points” are called by the same thread which invoked the operation on the target object.

The preinvoke method is called before the client request is marshalled and specifies the target object of the operation, the operation name, and a ServiceValueList. The interceptor may add new **ServiceValue**s to the **ServiceValueList** which will then be propagated to the client as part of the request.

The postinvoke method is called after the reply has been received from the target object. The service_contexts parameter specifies any service contexts that were received in a reply. The env parameter will contain any system exception which was received in the reply or a system exception if receipt of the reply failed due to some reason such as communication failure.

The **exception_occurred()** method is invoked if an interceptor in the chain throws an exception. This allows other interceptors in the chain to be notified of the exceptional condition, and perhaps take some action such as rolling back a transaction.

5.4.2 ServerRequestInterceptor

The **ServerRequestInterceptor**s are symmetric to the **ClientRequestInterceptor**s except that service contexts may only be manipulated in postinvoke instead of preinvoke.

The same thread is used to invoke preinvoke, the servant, and the postinvoke methods.

5.5 Message Interceptors

Message Interceptors allow for the manipulation of ORB messages before they are sent and after they are received. The message interceptors are described briefly below and a future revised submission will contain more complete information.

```
module PortableInterceptor {

    interface MessageInterceptor {
        void set_next(in MessageInterceptor interceptor);

        void write_message(in boolean isFirst, in boolean isLast,
                in CORBA::OctetSequence data, in long offset,
                in long length);

        void read_message(in boolean isFirst, in boolean isLast,
                in CORBA::OctetSequence data, in long offset,
                in long length);

        void flush();
        void close();
    };
};
```

MessageInterceptors may be used on either the client or server to modify messages before they are sent over the wire. MessageInterceptors are conceptually organized as a protocol with the ORB writing to the first message interceptor, and the second message interceptor writing to the third, etc. The last message interceptor in the stack is typically a transport layer such as TCP. MessageInterceptors are installed on a per-connection basis and must be installed independently for each connection an ORB makes to a server or for each connection received by a server.

The **set_next()** operation informs the MessageInterceptor which instance of a MessageInterceptor is next. This operation will be the first operation invoked by the ORB before any other operation is invoked.

The **read_message()** and **write_message()** operations are called when a message is to be read or written. Because ORBs may use different buffering schemes, multiple calls to **read_message()** and **write_message()** may be required to read/write a complete message. The boolean flags isFirst and **isLast** are used to indicate whether or not the particular buffer being passes is the first or last buffer for a particular message. If the message may be contained in a single buffer then both flags will be true.

The **flush()** operation is used to force a message interceptor to write any data it may be holding temporarily. The **MessageInterceptor** shall flush data by invoking **write_message()** with the remainder of the data on the next **MessageInterceptor**.

The close operation is used to indicate the ORB is closing the connection, and the **MessageInterceptor** shall take any action as required.

5.6 ORB Interceptors

ORB interceptors will be described in more detail in a revised version of this submission. For the purposes of this submission, only **BindInterceptor**s are discussed. The following ORB interceptors are expected to be added:

- Interceptor to allow for additional policies to be created when the **ORB::create_policy** method is invoked. For example, can be used to add policies specific to security or transactions.
- Interceptor to allow for additional objects to be returned from **ORB::resolve_initial_references** and **ORB::list_initial_services**. This can be used for example to add transaction and security **Current** interfaces to an ORB.

5.6.1 BindInterceptor

A BindInterceptor is called after a “binding” has been established between an object reference and a remote server. This typically means a connection to the server has been opened but no requests have been issued. For the purpose of this specification, it is only required that the ORB has selected a single profile from an IOR for communication with the server. The following IDL described the BindInterceptor APIs.

```
module PortableInterceptor {

    interface BindInterceptor {
        void bind(in Object target, in IOP::IORValue ior,
                  in long profileIndex,
                  in InterceptorManagerControl interceptorControl);
    };
```

```
        interface BindInterceptorManager : InterceptorManager {
            void add(in BindInterceptor interceptor);
        };
    };
```

POALifeCycleInterceptors are installed via the POALifeCycleInterceptorManager interface. An instance of a POALifeCycleManager is available by invoking **get_manager("Bind")** on the ORB InterceptorManagerControl instance (Section 5.3.1, "ORB InterceptorManagerControl," on page 5-20).

The bind() method is invoked every time the ORB is trying to invoke an operation on an object which has never previously been invoked by the ORB. The target parameter indicates the target of the request, the ior parameter indicates the IORValue corresponding to the target, and the profileIndex indicates which profile in the IOR is being used for communication with the target object. The interceptorControl parameter is an instance of a Bind InterceptorManagerControl, and is described below.

5.6.1.1 Bind InterceptorManagerControl

Each client object reference has a per-object Bind InterceptorManagerControl which is used to install per-object interceptors.Currently, the following per-object Interceptors are available.

- ClientRequestInterceptor - manager is available by invoking get_manager("ClientRequest") on the Bind InterceptorManagerControl.

5.7 POA Interceptors

The following hooks are provided:

- IOR templates. A per-POA IOR template.
- POA lifecycle hooks. Called during POA creation and destruction
- Additional POA hooks will be provided in a revised submission.

All POA interceptors are defined in the already existing PortableServer module.

5.7.1 IOR templates

Each POA instance shall maintain an IOR template, which is an IORValue which contains information common to all objects managed by that POA (typically everything except the repository_id and ObjectID of the object). This template is passed to certain POA interceptors which may modify or augment the template as necessary to implement a particular service. This will typically mean adding additional components to existing profiles in the IOR template to support transactions or security.

5.7.2 POALifeCycleInterceptor

The POALifeCycleInterceptor is a POA interceptor which is invoked every time a POA is created or destroyed. The IDL for POALifeCycleInterceptor is defined as follows:

```
module PortableServer {

    interface POALifeCycleInterceptor {
        void create(in PortableServer::POA poa,
                    inout IOP::IORValue iorTemplate,
                    in InterceptorManagerControl poaAdmin);
        void destroy(in PortableServer::POA poa);
    };

    interface POALifeCycleInterceptorManager : InterceptorManager {
        void add(in POALifeCycleInterceptor interceptor);
    };

};
```

POALifeCycleInterceptors are installed via the POALifeCycleInterceptorManager interface. An instance of a POALifeCycleManager is available by invoking **get_manager("POALifeCycle")** on the ORB InterceptorManagerControl instance (Section 5.3.1, "ORB InterceptorManagerControl," on page 5-20).

The create method is invoked when a new POA is created either explicitly through a call to create_POA or via an AdapterActivator. In the case of an AdapterActivator, the interceptor is called only after the unknown_adapter method returns successfully from the AdapterActivator.

The create method is passed a reference to the recently created POA, a copy of the POA's IOR template, and a reference to the POA's InterceptorManagerControl (Section 5.7.3, "POA InterceptorManagerControl," on page 5-26). If the create method throws a system exception, the exception will be propagated to the caller responsible for attempting to create the POA. No further interceptors in the chain will be called if any previous interceptors throws an exception. The POA will be destroyed and will appear to the user as if it hadn't been created.

The destroy operation is called when a POA has been destroyed and all of its objects have been etherealized. It is guaranteed that destroy will be called on all interceptors before create will be called again for a POA with the same name.

If the destroy operation throws a system exception it is ignored, and the remaining interceptors will continue to be called.

POALifeCycleInterceptors are installed globally in the InterceptorManager. Multiple POALifeCycleInterceptors may be installed and they will be called in the order in which they were installed. The order is very critical since it will affect the order in which per-POA interceptors are installed.

5.7.3 POA InterceptorManagerControl

Each POA has a per-POA InterceptorManagerControl which is made available to the user during the POALifeCycleInterceptor::create call. The POA InterceptorManagerControl can be used to get access to all per-POA InterceptorManagers for the purpose of installing per-POA Interceptors. Currently, the following per-POA Interceptors are available.

- ServerRequestInterceptor - manager is available by invoking get_manager("ServerRequest") on the POA InterceptorManagerControl.

5.7.4 ServerRequestInterceptors and the POA

When ServerRequest interceptors are installed on a POA, the intercept points must be more completely specified relative to other POA "interceptors" such as ServantManagers.

The ServerRequestInterceptor::preinvoke method is invoked before POA ServantLocators are invoked, but after any Servant or AdapterActivators have been invoked.

The ServerRequestInterceptor::postinvoke method is called after the servant has been invoked and any ServantLocator has been invoked.

5.8 Consolidated IDL

The consolidated IDL will be listed here in a future revision.

Conformance Issues 6

6.1 Introduction

This chapter specifies the compliance points for this specification

6.2 Compliance

This submission proposes extensions to the CORBA core. As such all the specified functionality is required by compliant implementations of the CORBA core.

Changes to CORBA 2.3 7

This submission proposes the following changes to CORBA 2.3 in order to support portable interceptors:.

7.1 Changes to CORBA 2.3

A complete listing of changes will be provided in a revised submission.