The DCE ESIOP 14

This chapter specifies an Environment Specific Inter-ORB Protocol (ESIOP) for the OSF DCE environment, the DCE Common Inter-ORB Protocol (DCE-CIOP).

Contents

This chapter contains the following sections.

14.1 Goals of the DCE Common Inter-ORB Protocol

DCE CIOP was designed to meet the following goals:

- Support multi-vendor, mission-critical, enterprise-wide, ORB-based applications.
- Leverage services provided by DCE wherever appropriate.
- Allow efficient and straightforward implementation using public DCE APIs.

- Preserve ORB implementation freedom.

DCE CIOP achieves these goals by using DCE-RPC to provide message transport, while leaving the ORB responsible for message formatting, data marshaling, and operation dispatch.

14.2 DCE Common Inter-ORB Protocol Overview

The DCE Common Inter-ORB Protocol uses the wire format and RPC packet formats defined by DCE-RPC to enable independently implemented ORBs to communicate. It defines the message formats that are exchanged using DCE-RPC, and specifies how information in object references is used to establish communication between client and server processes.

The full OMG IDL for the DCE ESIOP specification is shown in Section 14.7, “OMG IDL for the DCE CIOP Module,” on page 14-25. Fragments are used throughout this chapter as necessary.

14.2.1 DCE-CIOP RPC

DCE-CIOP requires an RPC which is interoperable with the DCE connection-oriented and/or connectionless protocols as specified in the X/Open *CAE Specification C309* and the OSF *AES/Distributed Computing RPC Volume*. Some of the features of the DCE-RPC are as follows:

- Defines connection-oriented and connectionless protocols for establishing the communication between a client and server.
- Supports multiple underlying transport protocols including TCP/IP.
- Supports multiple outstanding requests to multiple CORBA objects over the same connection.
- Supports fragmentation of messages. This provides for buffer management by ORBs of CORBA requests which contain a large amount of marshaled data.

All interactions between ORBs take the form of remote procedure calls on one of two well-known DCE-RPC interfaces. Two DCE operations are provided in each interface:

- *invoke* - for invoking CORBA operation requests, and
- *locate* - for locating server processes.

Each DCE operation is a synchronous remote procedure call[1,2], consisting of a request message being transmitted from the client to the server, followed by a response message being transmitted from the server to the client.

---

1. DCE *maybe* operation semantics cannot be used for CORBA *oneway* operations because they are idempotent as opposed to at-most-once.
2. The deferred synchronous DII API can be implemented on top of synchronous RPCs by using threads.

Using one of the DCE-RPC interfaces, the messages are transmitted as pipes of uninterpreted bytes. By transmitting messages via DCE pipes, the following characteristics are achieved:

- Large amounts of data can be transmitted efficiently.
- Buffering of complete messages is not required.
- Marshaling and demarshaling can take place concurrently with message transmission.
- Encoding of messages and marshaling of data is completely under the control of the ORB.
- DCE client and server stubs can be used to implement DCE-CIOP.

Using the other DCE-RPC interface, the messages are transmitted as conformant arrays of uninterpreted bytes. This interface does not offer all the advantages of the pipe-based interface, but is provided to enable interoperability with ORBs using DCE-RPC implementations that do not adequately support pipes.

14.2.2 DCE-CIOP Data Representation

DCE-CIOP messages represent OMG IDL types by using the CDR transfer syntax, which is defined in “CDR Transfer Syntax” on page 13-4. DCE-CIOP message headers and bodies are specified as OMG IDL types. These are encoded using CDR, and the resulting messages are passed between client and server processes via DCE-RPC pipes or conformant arrays.

NDR is the transfer syntax used by DCE-RPC for operations defined in DCE IDL. CDR, used to represent messages defined in OMG IDL on top of DCE RPCs, represents the OMG IDL primitive types identically to the NDR representation of corresponding DCE IDL primitive types.

The corresponding OMG IDL and DCE IDL primitive types are shown in table Table 14-1.

Table 14-1 Relationship between CDR and NDR primitive data types

OMG IDL type	DCE IDL type with NDR representation equivalent to CDR representation of OMG IDL type
char	byte
wchar	byte, unsigned short, or unsigned long, depending on transmission code set
octet	byte
short	short
unsigned short	unsigned short

Table 14-1 Relationship between CDR and NDR primitive data types

OMG IDL type	DCE IDL type with NDR representation equivalent to CDR representation of OMG IDL type
long	long
unsigned long	unsigned long
long long	hyper
unsigned long long	unsigned hyper
float	float[1]
double	double[2]
long double	long double[3]
boolean	byte[4]

1. Restricted to IEEE format.
2. Restricted to IEEE format.
3. Restricted to IEEE format.
4. Values restricted to 0 and 1.

The CDR representation of OMG IDL constructed types and pseudo-object types does not correspond to the NDR representation of types describable in DCE IDL.

A wide string is encoded as a unidimensional conformant array of octets in DCE 1.1 NDR. This consists of an unsigned long of four octets, specifying the number of octets in the array, followed by that number of octets, with no null terminator.

The NDR representation of OMG IDL fixed-point type, **fixed**, will be proposed as an addition to the set of DCE IDL types.

As new data types are added to OMG IDL, NDR can be used as a model for their CDR representations.

14.2.3 DCE-CIOP Messages

The following request and response messages are exchanged between ORB clients and servers via the `invoke` and `locate` RPCs:

- *Invoke Request* identifies the target object and the operation and contains the principal, the operation context, a **ServiceContext,** and the in and inout parameter values.

- *Invoke Response* indicates whether the operation succeeded, failed, or needs to be reinvoked at another location, and returns a **ServiceContext**. If the operation succeeded, the result and the out and inout parameter values are returned. If it failed, an exception is returned. If the object is at another location, new RPC binding information is returned.
- *Locate Request* identifies the target object and the operation.
- *Locate Response* indicates whether the location is in the current process, is elsewhere, or is unknown. If the object is at another location, new RPC binding information is returned.

All message formats begin with a field that indicates the byte order used in the CDR encoding of the remainder of the message. The CDR byte order of a message is required to match the NDR byte order used by DCE-RPC to transmit the message.

14.2.4 Interoperable Object Reference (IOR)

For DCE-CIOP to be used to invoke operations on an object, the information necessary to reference an object via DCE-CIOP must be included in an IOR. This information can coexist with the information needed for other protocols such as IIOP. DCE-CIOP information is stored in an IOR as a set of components in a profile identified by either **TAG_INTERNET_IOP** or **TAG_MULTIPLE_COMPONENTS**. Components are defined for the following purposes:

- To identify a server process via a DCE string binding, which can be either fully or partially bound. This process may be a server process implementing the object, or it may be an agent capable of locating the object implementation.
- To identify a server process via a name that can be resolved using a DCE nameservice. Again, this process may implement the object or may be an agent capable of locating it.
- In the **TAG_MULTIPLE_COMPONENTS** profile, to identify the target object when request messages are sent to the server. In the **TAG_INTENET_IOP** profile, the **object_key** profile member is used instead.
- To enable a DCE-CIOP client to recognize objects that share an endpoint.
- To indicate whether a DCE-CIOP client should send a locate message or an invoke message.
- To indicate if the pipe-based DCE-RPC interface is not available.

The IOR is created by the server ORB to provide the information necessary to reference the CORBA object.

14.3 DCE-CIOP Message Transport

DCE-CIOP defines two DCE-RPC interfaces for the transport of messages between client ORBs and server ORBs[3]. One interface uses pipes to convey the messages, while the other uses conformant arrays.

The pipe-based interface is the preferred interface, since it allows messages to be transmitted without precomputing the message length. But not all DCE-RPC implementations adequately support pipes, so this interface is optional. All client and server ORBs implementing DCE-CIOP must support the array-based interface[4].

While server ORBs may provide both interfaces or just the array-based interface, it is up to the client ORB to decide which to use for an invocation. If a client ORB tries to use the pipe-based interface and receives a `rpc_s_unknown_if` error, it should fall back to the array-based interface.

14.3.1 Pipe-based Interface

The `dce_ciop_pipe` interface is defined by the DCE IDL specification shown below:

```
[                        /* DCE IDL */
uuid(d7d99f66-97ee-11cf-b1a0-0800090b5d3e),/* 2nd revision */
version(1.0)
]
interface dce_ciop_pipe
{
    typedef pipe byte message_type;
    void invoke ([in] handle_t binding_handle,
                 [in] message_type *request_message,
                 [out] message_type *response_message);
    void locate ([in] handle_t binding_handle,
                 [in] message_type *request_message,
                 [out] message_type *response_message);
}
```

ORBs can implement the `dce_ciop_pipe` interface by using DCE stubs generated from this IDL specification, or by using lower-level APIs provided by a particular DCE-RPC implementation.

---

3. Previous DCE-CIOP revisions used different DCE RPC interface UUIDs and had incompatible message formats. These previous revisions are deprecated, but implementations can continue to support them in conjunction with the current interface UUIDs and message formats.
4. A future DCE-CIOP revision may eliminate the array-based interface and require support of the pipe-based interface.

The `dce_ciop_pipe` interface is identified by the UUID and version number shown. To provide maximal performance, all server ORBs and location agents implementing DCE-CIOP should listen for and handle requests made to this interface. To maximize the chances of interoperating with any DCE-CIOP client, servers should listen for requests arriving via all available DCE protocol sequences.

Client ORBs can invoke OMG IDL operations over DCE-CIOP by performing DCE RPCs on the `dce_ciop_pipe` interface. The `dce_ciop_pipe` interface is made up of two DCE-RPC operations, `invoke` and `locate`. The first parameter of each of these RPCs is a DCE binding handle, which identifies the server process on which to perform the RPC. See “DCE-CIOP String Binding Component” on page 14-17, “DCE-CIOP Binding Name Component” on page 14-18, and “DCE-CIOP Object Location” on page 14-22 for discussion of how these binding handles are obtained. The remaining parameters of the `dce_ciop_pipe` RPCs are pipes of uninterpreted bytes. These pipes are used to convey messages encoded using CDR. The `request_message` input parameters send a request message from the client to the server, while the `response_message` output parameters return a response message from the server to the client.

Figure 14-1 illustrates the layering of DCE-CIOP messages on the DCE-RPC protocol as NDR pipes:

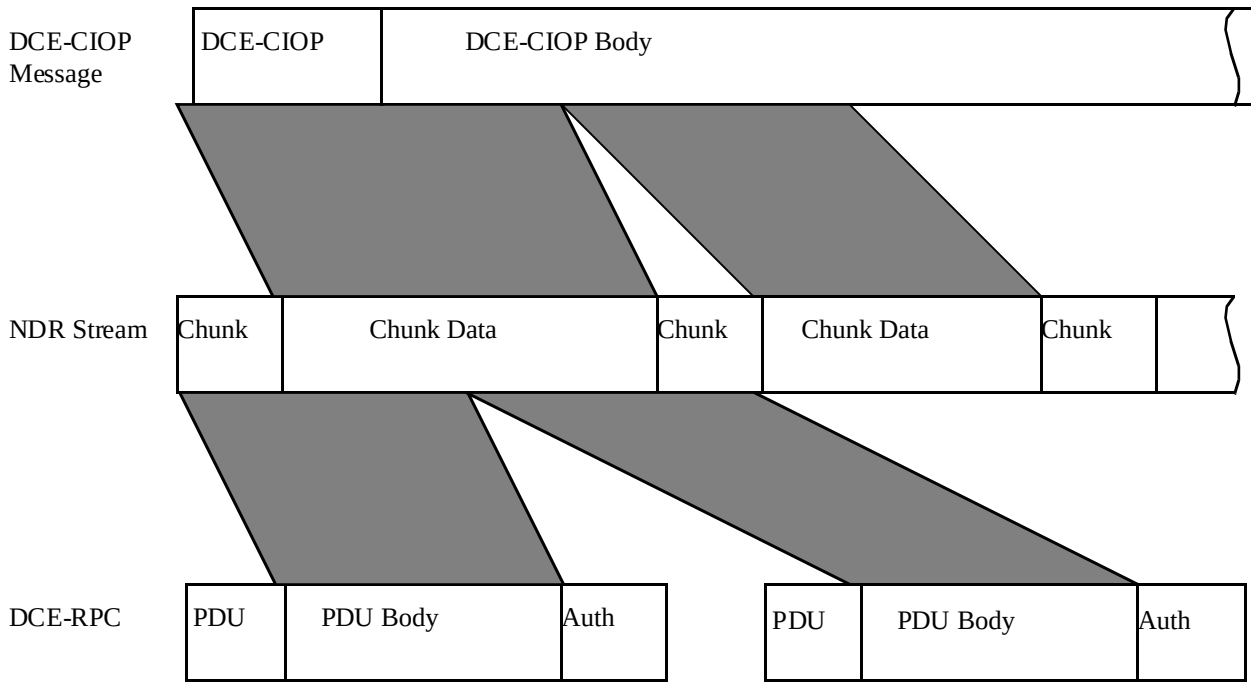

Figure 14-1 Pipe-based interface protocol layering.

The DCE-RPC protocol data unit (PDU) bodies, after any appropriate authentication is performed, are concatenated by the DCE-RPC run-time to form an NDR stream. This stream is then interpreted as the NDR representation of a DCE IDL pipe.

A pipe is made up of chunks, where each chunk consists of a chunk length and chunk data. The chunk length is an unsigned long indicating the number of pipe elements that make up the chunk data. The pipe elements are DCE IDL bytes, which are uninterpreted by NDR. A pipe is terminated by a chunk length of zero. The pipe chunks are concatenated to form a DCE-CIOP message.

Invoke

The `invoke` RPC is used by a DCE-CIOP client process to attempt to invoke a CORBA operation in the server process identified by the `binding_handle` parameter. The `request_message` pipe transmits a DCE-CIOP invoke request message, encoded using CDR, from the client to the server. See "DCE_CIOP Invoke Request Message" on page 14-11 for a description of its format. The `response_message` pipe transmits a DCE-CIOP invoke response message, also encoded using CDR, from the server to the client. See "DCE-CIOP Invoke Response Message" on page 14-12 for a description of the response format.

Locate

The `locate` RPC is used by a DCE-CIOP client process to query the server process identified by the `binding_handle` parameter for the location of the server process where requests should be sent. The `request_message` and `response_message` parameters are used similarly to the parameters of the `invoke` RPC. See "DCE-CIOP Locate Request Message" on page 14-14 and "DCE-CIOP Locate Response Message" on page 14-15 for descriptions of their formats. Use of the `locate` RPC is described in detail in "DCE-CIOP Object Location" on page 14-22.

14.3.2 Array-based Interface

The `dce_ciop_array` interface is defined by the DCE IDL specification shown below:

```
[                       /* DCE IDL */
uuid(09f9ffb8-97ef-11cf-9c96-0800090b5d3e),/* 2nd revision */
version(1.0)
]
interface dce_ciop_array
{
    typedef struct {
        unsigned long length;
        [size_is(length),ptr] byte *data;
    } message_type;

    void invoke ([in] handle_t binding_handle,
                 [in] message_type *request_message,
```

```
                    [out] message_type *response_message);

    void locate ([in] handle_t binding_handle,
                    [in] message_type *request_message,
                    [out] message_type *response_message);
}
```

ORBs can implement the `dce_ciop_array` interface, identified by the UUID and version number shown, by using DCE stubs generated from this IDL specification, or by using lower-level APIs provided by a particular DCE-RPC implementation.

All server ORBs and location agents implementing DCE-CIOP must listen for and handle requests made to the `dce_ciop_array` interface, and to maximize interoperability, should listen for requests arriving via all available DCE protocol sequences.

Client ORBs can invoke OMG IDL operations over DCE-CIOP by performing `locate` and `invoke` RPCs on the `dce_ciop_array` interface.

As with the `dce_ciop_pipe` interface, the first parameter of each `dce_ciop_array` RPC is a DCE binding handle that identifies the server process on which to perform the RPC. The remaining parameters are structures containing CDR-encoded messages. The `request_message` input parameters send a request message from the client to the server, while the `response_message` output parameters return a response message from the server to the client.

The `message_type` structure used to convey messages is made up of a `length` member and a `data` member:

- *length* - This member indicates the number of bytes in the message.
- *data* - This member is a full pointer to the first byte of the conformant array containing the message.

The layering of DCE-CIOP messages on DCE-RPC using NDR arrays is illustrated in Figure 14-2 below:

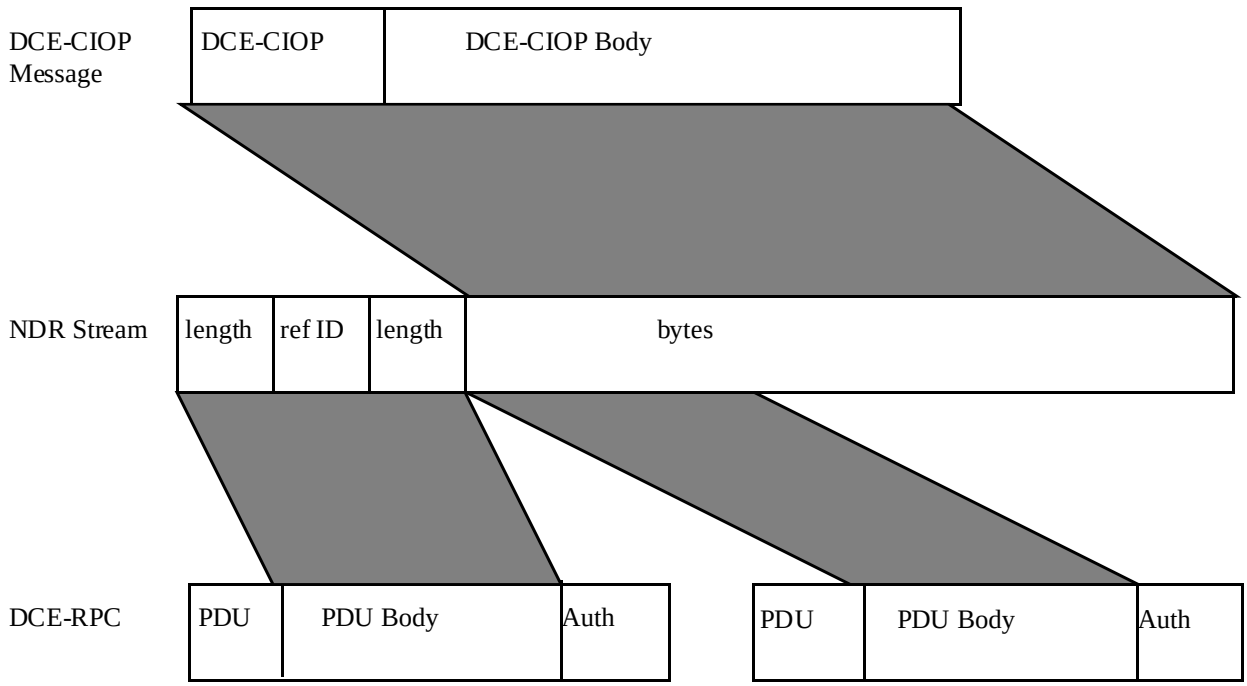

Figure 14-2 Array-based interface protocol layering.

The NDR stream, formed by concatenating the PDU bodies, is interpreted as the NDR representation of the DCE IDL `message_type` structure. The `length` member is encoded first, followed by the `data` member. The `data` member is a full pointer, which is represented in NDR as a referent ID. In this case, this non-NULL pointer is the first (and only) pointer to the referent, so the referent ID is 1 and it is followed by the representation of the referent. The referent is a conformant array of bytes, which is represented in NDR as an unsigned long indicating the length, followed by that number of bytes. The bytes form the DCE-CIOP message.

Invoke

The `invoke` RPC is used by a DCE-CIOP client process to attempt to invoke a CORBA operation in the server process identified by the `binding_handle` parameter. The `request_message` input parameter contains a DCE-CIOP invoke request message. The `response_message` output parameter returns a DCE-CIOP invoke response message from the server to the client.

Locate

The `locate` RPC is used by a DCE-CIOP client process to query the server process identified by the `binding_handle` parameter for the location of the server process where requests should be sent. The `request_message` and `response_message` parameters are used similarly to the parameters of the `invoke` RPC.

14.4 DCE-CIOP Message Formats

The section defines the message formats used by DCE-CIOP. These message formats are specified in OMG IDL, are encoded using CDR, and are transmitted over DCE-RPC as either pipes or arrays of bytes as described in "DCE-CIOP Message Transport" on page 14-5.

14.4.1 DCE_CIOP Invoke Request Message

DCE-CIOP invoke request messages encode CORBA object requests, including attribute accessor operations and `CORBA::Object` operations such as `get_interface` and `get_implementation`. Invoke requests are passed from client to server as the `request_message` parameter of an `invoke` RPC.

A DCE-CIOP invoke request message is made up of a header and a body. The header has a fixed format, while the format of the body is determined by the operation's IDL definition.

Invoke Request Header

DCE-CIOP request headers have the following structure:

```
module DCE_CIOP {                                          // IDL
    struct InvokeRequestHeader {
        boolean byte_order;
        IOP::ServiceContextList service_context;
        sequence <octet> object_key;
        string operation;
        CORBA::Principal principal;

        // in and inout parameters follow
    };
};
```

The members have the following definitions:

- **byte_order** indicates the byte ordering used in the representation of the remainder of the message. A value of FALSE indicates big-endian byte ordering, and TRUE indicates little-endian byte ordering.
- **service_context** contains any ORB service data that needs to be sent from the client to the server.

- **object_key** contains opaque data used to identify the object that is the target of the operation[5]. Its value is obtained from the **object_key** field of the **TAG_INTERNET_IOP** profile or the **TAG_COMPLETE_OBJECT_KEY** component of the **TAG_MULTIPLE_COMPONENTS** profile.
- **operation** contains the name of the CORBA operation being invoked. The case of the operation name must match the case of the operation name specified in the OMG IDL source for the interface being used.

  Attribute accessors have names as follows:
  - Attribute selector: operation name is "_get_<attribute>"
  - Attribute mutator: operation name is "_set_<attribute>"

  CORBA::Object pseudo-operations have operation names as follows:
  - **get_interface** - operation name is "_interface"
  - **get_implementation** - operation name is "_implementation"
  - **is_a** - operation name is "_is_a"
  - **non_existent** - operation name is "_non_existent"
- **Principal** contains a value identifying the requesting principal. No particular meaning or semantics are associated with this value. It is provided to support the `BOA::get_principal` operation.

Invoke Request Body

The invoke request body contains the following items encoded in this order:

- All in and inout parameters, in the order in which they are specified in the operation's OMG IDL definition, from left to right.
- An optional Context pseudo object, encoded as described in "Context" on page 13-18[6]. This item is only included if the operation's OMG IDL definition includes a context expression, and only includes context members as defined in that expression.

14.4.2 DCE-CIOP Invoke Response Message

Invoke response messages are returned from servers to clients as the `response_message` parameter of an `invoke` RPC.

---

5. Previous revisions of DCE-CIOP included an endpoint_id member, obtained from an optional TAG_ENDPOINT_ID component, as part of the object identity. The endpoint ID, if used, is now contained within the object key, and its position is specified by the optional TAG_ENDPOINT_ID_POSITION component.
6. Previous revisions of DCE-CIOP encoded the Context in the InvokeRequestHeader. It has been moved to the body for consistency with GIOP.

Like invoke request messages, an invoke response message is made up of a header and a body. The header has a fixed format, while the format of the body depends on the operation's OMG IDL definition and the outcome of the invocation.

Invoke Response Header

DCE-CIOP invoke response headers have the following structure:

```
module DCE_CIOP {                                          // IDL
    enum InvokeResponseStatus {
        INVOKE_NO_EXCEPTION,
        INVOKE_USER_EXCEPTION,
        INVOKE_SYSTEM_EXCEPTION,
        INVOKE_LOCATION_FORWARD,
        INVOKE_TRY_AGAIN
    };

    struct InvokeResponseHeader {
        boolean byte_order;
        IOP::ServiceContextList service_context;
        InvokeResponseStatus status;

        // if status = INVOKE_NO_EXCEPTION,
        // result then inouts and outs follow


        // if status = INVOKE_USER_EXCEPTION or
        // INVOKE_SYSTEM_EXCEPTION, an exception follows

        // if status = INVOKE_LOCATION_FORWARD, an
        // IOP::IOR follows
    };
};
```

The members have the following definitions:

- **byte_order** indicates the byte ordering used in the representation of the remainder of the message. A value of FALSE indicates big-endian byte ordering, and TRUE indicates little-endian byte ordering.
- **service_context** contains any ORB service data that needs to be sent from the client to the server.
- **status** indicates the completion status of the associated request, and also determines the contents of the body.

Invoke Response Body

The contents of the invoke response body depends on the value of the `status` member of the invoke response header, as well as the OMG IDL definition of the operation being invoked. Its format is one of the following:

- If the **status** value is **INVOKE_NO_EXCEPTION,** then the body contains the operation result value (if any), followed by all inout and out parameters, in the order in which they appear in the operation signature, from left to right.
- If the **status** value is **INVOKE_USER_EXCEPTION** or **INVOKE_SYSTEM_EXCEPTION**, then the body contains the exception, encoded as in GIOP.
- If the **status** value is **INVOKE_LOCATION_FORWARD**, then the body contains a new IOR containing a **TAG_INTERNET_IOP** or **TAG_MULTIPLE_COMPONENTS** profile whose components can be used to communicate with the object specified in the invoke request message[7]. This profile must provide at least one new DCE-CIOP binding component. The client ORB is responsible for resending the request to the server identified by the new profile. This operation should be transparent to the client program making the request. See "DCE-CIOP Object Location" on page 14-22 for more details.
- If the **status** value is **INVOKE_TRY_AGAIN**, then the body is empty and the client should reissue the `invoke` RPC, possibly after a short delay[8].

14.4.3 DCE-CIOP Locate Request Message

Locate request messages may be sent from a client to a server, as the `request_message` parameter of a `locate` RPC, to determine the following regarding a specified object reference:

- Whether the object reference is valid
- Whether the current server is capable of directly receiving requests for the object reference
- If not capable, to solicit an address to which requests for the object reference should be sent.

For details on the usage of the `locate` RPC, see "DCE-CIOP Object Location" on page 14-22.

Locate request messages contain a fixed-format header, but no body.

Locate Request Header

DCE-CIOP locate request headers have the following format:

7. Previous revisions of DCE-CIOP returned a MultipleComponentProfile structure. An IOR is now returned to allow either a TAG_INTERNET_IOP or a TAG_MULTIPLE_COMPONENTS profile to be used.
8. An exponential back-off algorithm is recommended, but not required.

```
module DCE_CIOP {                        // IDL
    struct LocateRequestHeader {
        boolean byte_order;
        sequence <octet> object_key;
        string operation;

        // no body follows
    };
};
```

The members have the following definitions:

- **byte_order** indicates the byte ordering used in the representation of the remainder of the message. A value of FALSE indicates big-endian byte ordering, and TRUE indicates little-endian byte ordering.
- **object_key** contains opaque data used to identify the object that is the target of the operation. Its value is obtained from the **object_key** field of the **TAG_INTERNET_IOP** profile or the **TAG_COMPLETE_OBJECT_KEY** component of the **TAG_MULTIPLE_COMPONENTS** profile.
- **operation** contains the name of the CORBA operation being invoked. It is encoded as in the invoke request header.

14.4.4 DCE-CIOP Locate Response Message

Locate response messages are sent from servers to clients as the `response_message` parameter of a `locate` RPC. They consist of a fixed-format header, and a body whose format depends on information in the header.

Locate Response Header

DCE-CIOP locate response headers have the following format:

```
module DCE_CIOP {                              // IDL
    enum LocateResponseStatus {
        LOCATE_UNKNOWN_OBJECT,
        LOCATE_OBJECT_HERE,
        LOCATE_LOCATION_FORWARD,
        LOCATE_TRY_AGAIN
    };
    struct LocateResponseHeader {
        boolean byte_order;
        LocateResponseStatus status;

        // if status = LOCATE_LOCATION_FORWARD, an
        // IOP::IOR follows
    };
};
```

The members have the following definitions:

- **byte_order** indicates the byte ordering used in the representation of the remainder of the message. A value of FALSE indicates big-endian byte ordering, and TRUE indicates little-endian byte ordering.
- **status** indicates whether the object is valid and whether it is located in this server. It determines the contents of the body.

Locate Response Body

The contents of the locate response body depends on the value of the `status` member of the locate response header. Its format is one of the following:

- If the `status` value is LOCATE_UNKNOWN_OBJECT, then the object specified in the corresponding locate request message is unknown to the server. The locate reply body is empty in this case.
- If the `status` value is LOCATE_OBJECT_HERE, then this server (the originator of the locate response message) can directly receive requests for the specified object. The locate response body is also empty in this case.
- If the `status` value is LOCATE_LOCATION_FORWARD, then the locate response body contains a new IOR containing a **TAG_INTERNET_IOP** or **TAG_MULTIPLE_COMPONENTS** profile whose components can be used to communicate with the object specified in the locate request message. This profile must provide at least one new DCE-CIOP binding component.
- If the status value is LOCATE_TRY_AGAIN, the locate response body is empty and the client should reissue the `locate` RPC, possibly after a short delay[9].

14.5 DCE-CIOP Object References

The information necessary to invoke operations on objects using DCE-CIOP is encoded in an IOR in a profile identified either by **TAG_INTERNET_IOP** or by **TAG_MULTIPLE_COMPONENTS**. The **profile_data** for the **TAG_INTERNET_IOP** profile is a CDR encapsulation of the **IIOP::ProfileBody_1_1** type, described in “IIOP IOR Profiles” on page 13-34. The **profile_data** for the **TAG_MULTIPLE_COMPONENTS** profile is a CDR encapsulation of the **MultipleComponentProfile** type, which is a sequence of

TaggedComponent structures, described in “An Information Model for Object References” on page 11-14.

DCE-CIOP defines a number of IOR components that can be included in either profile. Each is identified by a unique tag, and the encoding and semantics of the associated **component_data** are specified.

9. An exponential back-off algorithm is recommended, but not required.

Either IOR profile can contain components for other protocols in addition to DCE-CIOP, and can contain components used by other kinds of ORB services. For example, an ORB vendor can define its own private components within this profile to support the vendor's native protocol. Several of the components defined for DCE-CIOP may be of use to other protocols as well. The following component descriptions will note whether the component is intended solely for DCE-CIOP or can be used by other protocols, whether the component is required or optional for DCE-CIOP, and whether more than one instance of the component can be included in a profile.

A conforming implementation of DCE-CIOP is only required to generate and recognize the components defined here. Unrecognized components should be preserved but ignored. Implementations should also be prepared to encounter profiles identified by **TAG_INTERNET_IOP** or by **TAG_MULTIPLE_COMPONENTS** that do not support DCE-CIOP.

14.5.1 DCE-CIOP String Binding Component

A DCE-CIOP string binding component, identified by **TAG_DCE_STRING_BINDING**, contains a fully or partially bound string binding. A string binding provides the information necessary for DCE-RPC to establish communication with a server process that can either service the client's requests itself, or provide the location of another process that can. The DCE API routine `rpc_binding_from_string_binding` can be used to convert a string binding to the DCE binding handle required to communicate with a server as described in "DCE-CIOP Message Transport" on page 14-5.

This component is intended to be used only by DCE-CIOP. At least one string binding or binding name component must be present for an IOR profile to support DCE-CIOP.

Multiple string binding components can be included in a profile to define endpoints for different DCE protocols, or to identify multiple servers or agents capable of servicing the request.

The string binding component is defined as follows:

```
module DCE_CIOP {                                          \\ IDL
    const IOP::ComponentId TAG_DCE_STRING_BINDING = 100;
};
```

A **TaggedComponent** structure is built for the string binding component by setting the tag member to **TAG_DCE_STRING_BINDING** and setting the **component_data** member to the value of a DCE string binding. The string is represented directly in the sequence of octets, including the terminating NUL, without further encoding.

The format of a string binding is defined in Chapter 3 of the OSF *AES/Distributed Computing RPC Volume*. The DCE API function `rpc_binding_from_string_binding` converts a string binding into a binding handle that can be used by a client ORB as the first parameter to the `invoke` and `locate` RPCs.

A string binding contains:

- A protocol sequence
- A network address
- An optional endpoint
- An optional object UUID

DCE object UUIDs are used to identify server process endpoints, which can each support any number of CORBA objects. DCE object UUIDs do not necessarily correspond to individual CORBA objects.

A partially bound string binding does not contain an endpoint. Since the DCE-RPC run-time uses an endpoint mapper to complete a partial binding, and multiple ORB servers might be located on the same host, partially bound string bindings must contain object UUIDs to distinguish different endpoints at the same network address.

14.5.2 DCE-CIOP Binding Name Component

A DCE-CIOP binding name component is identified by **TAG_DCE_BINDING_NAME**. It contains a name that can be used with a DCE nameservice such as CDS or GDS to obtain the binding handle needed to communicate with a server process.

This component is intended for use only by DCE-CIOP. Multiple binding name components can be included to identify multiple servers or agents capable of handling a request. At least one binding name or string binding component must be present for a profile to support DCE-CIOP.

The binding name component is defined by the following OMG IDL:

```
module DCE_CIOP {                                                    // IDL
    const IOP::ComponentId TAG_DCE_BINDING_NAME = 101;

    struct BindingNameComponent {
        unsigned long entry_name_syntax;
        string entry_name;
        string object_uuid;
    };
};
```

A **TaggedComponent** structure is built for the binding name component by setting the tag member to **TAG_DCE_BINDING_NAME** and setting the **component_data member** to a CDR encapsulation of a **BindingNameComponent** structure.

BindingNameComponent

The **BindingNameComponent** structure contains the information necessary to query a DCE nameservice such as CDS. A client ORB can use the **entry_name_syntax, entry_name,** and **object_uuid** members of the **BindingName** structure with the `rpc_ns_binding_import_*` or `rpc_ns_binding_lookup_*` families of DCE

API routines to obtain binding handles to communicate with a server. If the `object_uuid` member is an empty string, a nil object UUID should be passed to these DCE API routines.

14.5.3 DCE-CIOP No Pipes Component

The optional component identified by **TAG_DCE_NO_PIPES** indicates to an ORB client that the server does not support the `dce_ciop_pipe` DCE-RPC interface. It is only a hint, and can be safely ignored. As described in "DCE-CIOP Message Transport" on page 14-5, the client must fall back to the array-based interface if the pipe-based interface is not available in the server.

```
module DCE_CIOP {
    const IOP::ComponentId TAG_DCE_NO_PIPES = 102;
};
```

A **TaggedComponent** structure with a `tag` member of **TAG_DCE_NO_PIPES** must have an empty **component_data** member.

This component is intended for use only by DCE-CIOP, and a profile should not contain more than one component with this tag.

14.5.4 Complete Object Key Component

An IOR profile supporting DCE-CIOP must include an object key that identifies the object the IOR represents. The object key is an opaque sequence of octets used as the **object_key** member in invoke and locate request message headers. In a **TAG_INTERNET_IOP** profile, the **object_key** member of the **IIOP::ProfileBody_1_1** structure is used. In a **TAG_MULTIPLE_COMPONENTS** profile supporting DCE-CIOP[10], a single **TAG_COMPLETE_OBJECT_KEY** component must be included to identify the object.

The **TAG_COMPLETE_OBJECT_KEY** component is available for use by all protocols that use the **TAG_MULTIPLE_COMPONENTS** profile. By sharing this component, protocols can avoid duplicating object identity information. This component should never be included in a **TAG_INTERNET_IOP** profile.

```
module IOP {                                        // IDL
const ComponentId TAG_COMPLETE_OBJECT_KEY = 5;
};
```

The sequence of octets comprising the **component_data** of this component is not interpreted by the client process. Its format only needs to be understood by the server process and any location agent that it uses.

---

10. Previous DCE-CIOP revisions used a different component.

14.5.5 Endpoint ID Position Component

An optional endpoint ID position component can be included in IOR profiles to enable client ORBs to minimize resource utilization and to avoid redundant locate messages. It can be used by other protocols as well as by DCE-CIOP. No more than one endpoint ID position component can be included in a profile.

```
module IOP {                                                    // IDL
const ComponentId TAG_ENDPOINT_ID_POSITION = 6;

struct EndpointIdPositionComponent {
unsigned short begin;
unsigned short end;
};
};
```

An endpoint ID position component, identified by **TAG_ENDPOINT_ID_POSITION**, indicates the portion of the profile's object key that identifies the endpoint at which operations on an object can be invoked. The **component_data** is a CDR encapsulation of an **EndpointIdPositionComponent** structure. The **begin** member of this structure specifies the index in the object key of the first octet of the endpoint ID. The **end** member specifies the index of the last octet of the endpoint ID. An index value of zero specifies the first octet of the object key. The value of **end** must be greater than the value of **begin**, but less than the total number of octets in the object key. The endpoint ID is made up of the octets located between these two indices inclusively.

The endpoint ID should be unique within the domain of interoperability. A binary or stringified UUID is recommended.

If multiple objects have the same endpoint ID, they can be messaged to at a single endpoint, avoiding the need to locate each object individually. DCE-CIOP clients can use a single binding handle to invoke requests on all of the objects with a common endpoint ID. See "Use of the Location Policy and the Endpoint ID" on page 14-24.

14.5.6 Location Policy Component

An optional location policy component can be included in IOR profiles to specify when a DCE-CIOP client ORB should perform a `locate` RPC before attempting to perform an `invoke` RPC. No more than one location policy component should be included in a profile, and it can be used by other protocols that have location algorithms similar to DCE-CIOP.

```
module IOP {                                        // IDL
    const ComponentId TAG_LOCATION_POLICY = 12;

    // IDL does not support octet constants
    #define LOCATE_NEVER = 0
    #define LOCATE_OBJECT = 1
    #define LOCATE_OPERATION = 2
    #define LOCATE_ALWAYS = 3
};
```

A **TaggedComponent** structure for a location policy component is built by setting the tag member to **TAG_LOCATION_POLICY** and setting the **component_data** member to a sequence containing a single octet, whose value is **LOCATE_NEVER**, **LOCATE_OBJECT, LOCATE_OPERATION**, or **LOCATE_ALWAYS**.

If a location policy component is not present in a profile, the client should assume a location policy of **LOCATE_OBJECT**.

A client should interpret the location policy as follows:

- **LOCATE_NEVER** - Perform only the `invoke` RPC. No `locate` RPC is necessary.
- **LOCATE_OBJECT** - Perform a `locate` RPC once per object. The **operation** member of the locate request message will be ignored.
- **LOCATE_OPERATION** - Perform a separate `locate` RPC for each distinct operation on the object. This policy can be used when different methods of an object are located in different processes.
- **LOCATE_ALWAYS** - Perform a separate `locate` RPC for each invocation on the object. This policy can be used to support server-per-method activation.

The location policy is a hint that enables a client to avoid unnecessary `locate` RPCs and to avoid `invoke` RPCs that return **INVOKE_LOCATION_FORWARD** status. It is not needed to provide correct semantics, and can be ignored. Even when this hint is utilized, an `invoke` RPC might result in an **INVOKE_LOCATION_FORWARD** response. See “DCE-CIOP Object Location” on page 14-22 for more details.

A client does not need to implement all location policies to make use of this hint. A location policy with a higher value can be substituted for one with a lower value. For instance, a client might treat **LOCATE_OPERATION** as **LOCATE_ALWAYS** to avoid having to keep track of binding information for each operation on an object.

When combined with an endpoint ID component, a location policy of **LOCATE_OBJECT** indicates that the client should perform a `locate` RPC for the first object with a particular endpoint ID, and then just perform an `invoke` RPC for other objects with the same endpoint ID. When a location policy of **LOCATE_NEVER** is combined with an endpoint ID component, only `invoke` RPCs need be performed. The **LOCATE_ALWAYS** and **LOCATE_OPERATION** policies should not be combined with an endpoint ID component in a profile.

14.6 DCE-CIOP Object Location

This section describes how DCE-CIOP client ORBs locate the server ORBs that can perform operations on an object via the `invoke` RPC.

14.6.1 Location Mechanism Overview

DCE-CIOP is defined to support object migration and location services without dictating the existence of specific ORB architectures or features. The protocol features are based on the following observations:

- A given transport address does not necessarily correspond to any specific ORB architectural component (such as an object adapter, server process, ORB process, locator, etc.). It merely implies the existence of some agent to which requests may be sent.
- The "agent" (receiver of an RPC) may have one of the following roles with respect to a particular object reference:
  - The agent may be able to accept object requests directly for the object and return replies. The agent may or may not own the actual object implementation; it may be a gateway that transforms the request and passes it on to another process or ORB. From DCE-CIOP's perspective, it is only important that invoke request messages can be sent directly to the agent.
  - The agent may not be able to accept direct requests for any objects, but acts instead as a location service. Any invoke request messages sent to the agent would result in either exceptions or replies with **INVOKE_LOCATION_FORWARD** status, providing new addresses to which requests may be sent. Such agents would also respond to locate request messages with appropriate locate response messages.
  - The agent may directly respond to some requests (for certain objects) and provide forwarding locations for other objects.
  - The agent may directly respond to requests for a particular object at one point in time, and provide a forwarding location at a later time.
- Server ORBs are not required to implement location forwarding mechanisms. An ORB can be implemented with the policy that servers either support direct access to an object, or return exceptions. Such a server ORB would always return locate response messages with either **LOCATE_OBJECT_HERE** or **LOCATE_UNKNOWN_OBJECT** status, and never **LOCATE_LOCATION_FORWARD** status. It would also never return invoke response messages with **INVOKE_LOCATION_FORWARD** status.
- Client ORBs must, however, be able to accept and process invoke response messages with **INVOKE_LOCATION_FORWARD** status, since any server ORB may choose to implement a location service. Whether a client ORB chooses to send locate request messages is at the discretion of the client.
- Client ORBs that send locate request messages can use the location policy component found in DCE-CIOP IOR profiles to decide whether to send a locate request message before sending an invoke request message. See "Location Policy Component" on page 14-20. This hint can be safely ignored by a client ORB.

- A client should not make any assumptions about the longevity of addresses returned by location forwarding mechanisms. If a binding handle based on location forwarding information is used successfully, but then fails, subsequent attempts to send requests to the same object should start with the original address specified in the object reference.

In general, the use of location forwarding mechanisms is at the discretion of ORBs, available to be used for optimization and to support flexible object location and migration behaviors.

14.6.2 Activation

Activation of ORB servers is transparent to ORB clients using DCE-CIOP. Unless an IOR refers to a transient object, the agent addressed by the IOR profile should either be permanently active, or should be activated on demand by DCE's endpoint mapper.

The current DCE endpoint mapper, rpcd, does not provide activation. In ORB server environments using rpcd, the agent addressed by an IOR must not only be capable of locating the object, it must also be able to activate it if necessary. A future DCE endpoint mapper may provide automatic activation, but client ORB implementations do not need to be aware of this distinction.

14.6.3 Basic Location Algorithm

ORB clients can use the following algorithm to locate the server capable of handling the `invoke` RPC for a particular operation:

1. Pick a profile with **TAG_INTERNET_IOP** or **TAG_MULTIPLE_COMPONENTS** from the IOR. Make this the *original* profile and the *current* profile. If no profiles with either tag are available, operations cannot be invoked using DCE-CIOP with this IOR.

2. Get a binding handle to try from the *current* profile. See "DCE-CIOP String Binding Component" on page 14-17 and "DCE-CIOP Binding Name Component" on page 14-18. If no binding handles can be obtained, the server cannot be located using the *current* profile, so go to step 1.

3. Perform either a `locate` or `invoke` RPC using the object key from the *current* profile.
   - If the RPC fails, go to step 2 to try a different binding handle.
   - If the RPC returns **INVOKE_TRY_AGAIN** or **LOCATE_TRY_AGAIN**, try the same RPC again, possibly after a delay.
   - If the RPC returns either **INVOKE_LOCATION_FORWARD** or **LOCATE_LOCATION_FORWARD**, make the new IOR profile returned in the response message body the *current* profile and go to step 2.
   - If the RPC returns **LOCATE_UNKNOWN_OBJECT**, and the *original* profile was used, the object no longer exists.
   - Otherwise, the server has been successfully located.

Any `invoke` RPC might return **INVOKE_LOCATION_FORWARD**, in which case the client ORB should make the returned profile the *current* profile, and re-enter the location algorithm at step 2.

If an RPC on a binding handle fails after it has been used successfully, the client ORB should start over at step 1.

14.6.4 Use of the Location Policy and the Endpoint ID

The algorithm above will allow a client ORB to successfully locate a server ORB, if possible, so that operations can be invoked using DCE-CIOP. But unnecessary `locate` RPCs may be performed, and `invoke` RPCs may be performed when `locate` RPCs would be more efficient. The optional location policy and endpoint ID position components can be used by the client ORB, if present in the IOR profile, to optimize this algorithm.

Current Location Policy

The client ORB can decide whether to perform a `locate` RPC or an `invoke` RPC in step 3 based on the location policy of the *current* IOR profile. If the *current* profile has a **TAG_LOCATION_POLICY** component with a value of **LOCATE_NEVER**, the client should perform an `invoke` RPC. Otherwise, it should perform a `locate` RPC.

Original Location Policy

The client ORB can use the location policy of the *original* IOR profile as follows to determine whether it is necessary to perform the location algorithm for a particular invocation:

- LOCATE_OBJECT or LOCATE_NEVER A binding handle previously used successfully to invoke an operation on an object can be reused for all operations on the same object. The client only needs to perform the location algorithm once per object.
- LOCATE_OPERATION A binding handle previously used successfully to invoke an operation on an object can be reused for that same operation on the same object. The client only needs to perform the location algorithm once per operation.
- LOCATE_ALWAYS Binding handles should not be reused. The client needs to perform the location algorithm once per invocation.

Original Endpoint ID

If a component with **TAG_ENDPOINT_ID_POSITION** is present in the *original* IOR profile, the client ORB can reuse a binding handle that was successfully used to perform an operation on another object with the same endpoint ID. The client only needs to perform the location algorithm once per endpoint.

An endpoint ID position component should never be combined in the same profile with a location policy of **LOCATE_OPERATION** or **LOCATE_ALWAYS**.

14.7 OMG IDL for the DCE CIOP Module

This section shows the DCE_CIOP module and DCE_CIOP additions to the IOP module.

```
module DCE_CIOP {
    struct InvokeRequestHeader {
        boolean byte_order;
        IOP::ServiceContextList service_context;
        sequence <octet> object_key;
        string operation;
        CORBA::Principal principal;

        // in and inout parameters follow
    };

    enum InvokeResponseStatus {
        INVOKE_NO_EXCEPTION,
        INVOKE_USER_EXCEPTION,
        INVOKE_SYSTEM_EXCEPTION,
        INVOKE_LOCATION_FORWARD,
        INVOKE_TRY_AGAIN
    };
    struct InvokeResponseHeader {
        boolean byte_order;
        IOP::ServiceContextList service_context;
        InvokeResponseStatus status;

        // if status = INVOKE_NO_EXCEPTION,
        // result then inouts and outs follow

        // if status = INVOKE_USER_EXCEPTION or
        // INVOKE_SYSTEM_EXCEPTION, an exception follows

        // if status = INVOKE_LOCATION_FORWARD, an
        // IOP::IOR follows
    };
    struct LocateRequestHeader {
        boolean byte_order;
        sequence <octet> object_key;
        string operation;

        // no body follows
    };

enum LocateResponseStatus {
LOCATE_UNKNOWN_OBJECT,
LOCATE_OBJECT_HERE,
LOCATE_LOCATION_FORWARD,
LOCATE_TRY_AGAIN
```

```
};
struct LocateResponseHeader {
boolean byte_order;
LocateResponseStatus status;

// if status = LOCATE_LOCATION_FORWARD, an
// IOP::IOR follows
};

const IOP::ComponentId TAG_DCE_STRING_BINDING = 100;

const IOP::ComponentId TAG_DCE_BINDING_NAME = 101;

struct BindingNameComponent {
unsigned long entry_name_syntax;
string entry_name;
string object_uuid;
};

const IOP::ComponentId TAG_DCE_NO_PIPES = 102;
};

module IOP {
const ComponentId TAG_COMPLETE_OBJECT_KEY = 5;

const ComponentId TAG_ENDPOINT_ID_POSITION = 6;

struct EndpointIdPositionComponent {
unsigned short begin;
unsigned short end;
};

const ComponentId TAG_LOCATION_POLICY = 12;

// IDL does not support octet constants
#define LOCATE_NEVER 0
#define LOCATE_OBJECT 1
#define LOCATE_OPERATION 2
#define LOCATE_ALWAYS 3
};
```

14.8 References for this Chapter

AES/Distributed Computing RPC Volume, P T R Prentice Hall, Englewood Cliffs, New Jersey, 1994

CAE Specification C309 X/Open DCE: Remote Procedure Call, X/Open Company Limited, Reading, UK