Interoperable Naming Service

BEA Systems, Inc.
Cooperative Research Centre for Distributed Systems Technology (DSTC Pty Ltd)
Inprise Corporation
IONA Technologies, PLC

OMG TC Document orbos/98-10-11
19 October 1998

Notices

Each of the following terms used in this publication is a trademark of another company:

OMG	Object Management Group
CORBA	Object Management Group
ORB	Object Management Group
IDL	Object Management Group

The term “CORBA” used throughout this publication refers to the Common Object Request Broker Architecture standards promulgated by the Object Management Group, Inc.

Introduction 1

BEA Systems, the Cooperative Research Centre for Distributed Systems Technology (DSTC), Inprise Corporation, and IONA Technologies are pleased to submit this Interoperable Naming Service specification in response to the OMG's Interoperability Name Service Enhancements RFP.

1.1 Submission Contact Points

Enquiries and comments about this submission are most welcome and should be directed to:

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

Michi Henning, Michael Neville, Ted McFadden
CRC for Distributed Systems Technology
University of Queensland
Brisbane 4072
Australia
Phone: +61 7 3365-4310
Fax: +61 7 3365-4311
Email: {michi, neville, mcfadden}@dstc.edu.au

Jeff Mischkinsky
Inprise Corporation
951 Mariner's Island Blvd. Suite 120
San Mateo, CA 94404

USA
Phone: +1 650 358-3049
Email: jeffm@inprise.com

Martin Chapman
IONA Technologies PLC
The IONA Building
Shellbourne Road
Dublin 4, Ireland
Phone: +353 1 662 5255
Fax: +353 1 662 5244
Email: mchapman@iona.com

1.2 Submission Overview

This submission is organized as follows:

- Chapter 2 describes the rationale for the service.
- Chapter 3 explains how clients and server obtain access to initial references and shows the URL format for names.
- Chapter 4 is the replacement chapter for the CORBAservices Naming Service Specification.

1.3 Scope of RFP

This RFP addresses the following points:

- Resolves open issues affecting interoperability with the current Naming Service.
- Provides a mechanism to allow multiple clients to access a common Initial Naming Context.
- Defines an interoperable, stringified form of a `CosNaming::Name`.
- Defines a URL format for names.

1.4 Proof of Concept

At the time of submission, the submitters have an implementation of this specification in testing. No problems that would endanger the technical viability of this specification were found.

The IDL presented in this submission has been compiled successfully with the following compilers:

- BEA M3 2.1
- DSTC internal research compiler
- Expersoft PowerBroker CORBAplus 2.2
- HP ORB Plus 2.6
- Inprise VisiBroker 3.2
- IONA Orbix 2.3c

1.5 Changes to Existing CORBA Specifications

This specification adds definitions to the ORB core, but does not add or change the existing `CosNaming` IDL interfaces. No changes to IIOP are required to implement this specification. This submission defines four new minor codes for the `BAD_PARAM` system exception. These minor codes will need to be assigned by the OMG.

1.6 Registrations with Other Standards Organizations

This specification introduces two new stringified URL schemes, `iioploc` and `iiopname`, in addition to the currently specified `IOR` scheme. If this submission is adopted, the OMG will need to register all three schemes with the Internet Assigned Numbers Authority (IANA).

This specification uses port 9999 as the default port for use with the `iioploc` and `iiopname` schemes. If this submission is adopted, the OMG will need to request assignment of this or another port number from IANA.

1.7 Addressed Requirements

1.7.1 Specific Requirements

This submission addresses all of the requirements stated in the RFP.

This submission is upwardly compatible with current OMG specifications.

1.7.2 General Requirements

This submission meets the requirements stated in section 5 of the RFP.

Design rationale 2

This section presents the rationale for the design of this submission. Further explanations can be found in the description of the relevant functionality.

2.1 RFP Requirements

The Interoperability Name Service Enhancements RFP (orbos/97-12-20) solicited proposals addressing several areas concerning the Naming Service. We have addressed these issues in a manner that is compatible with the current Naming Service specification. The RFP items are listed in the following sections.

2.1.1 Name String Syntax

The RFP states:

"Name string syntax is not specified. Interfaces for parsing names are not specified. These make it difficult for users to easily define and search for names. A common means for parsing name-strings and converting them to and from name-structures would increase interoperability."

We define a straightforward stringified representation of names that is user-friendly, provides a unique and easily comparable representation for a given name, and is interoperable between Naming Services.

2.1.2 Configuration of Initial Naming Context

The RFP states:

"There is no standard way to allow clients to independently bootstrap to a common naming context. For example, using a file system relies on the presence of a shared file system and an agreed upon location, it is not a particularly scalable solution."

This submission defines a simple mechanism by which `resolve_initial_references` can be configured to return a common naming context to any number of clients. The mechanism is scalable and supports other services besides the Naming Service.

2.1.3 URL Names

The RFP states:

"With the use of CORBA in the Internet, and it's associated Domain Name Service (DNS), it is desirable to support the use of a Uniform Resource Locator (URL) based names, thus allowing for example, Web-browser based access to CosNaming in a standard way."

Given that the current CORBA `IOR` format is already a valid URL but difficult for humans to transcribe, we have defined `iioploc` and `iiopname` URL schemes that are more user-friendly and similar to `ftp` and `http` URLs.

2.1.4 Interoperability Defects in the Current Naming Specification

Significance of `id` and `kind` in `CosName` Comparison

The RFP states:

"When a `CosNaming::Name` is evaluated as a sequence of components, it is not stated whether or not both the `id` and `kind` field of the two `CosNaming::NameComponent`s must match for the two components to be identical. Differing products have made different choices, leading to interoperability problems."

This submission clearly states the significance of the `id` and `kind` fields during comparison. Both `id` and `kind` fields are significant, even if empty.

Clarifying only this point would still leave a Naming Service specification that is too deficient to be interoperable. To meet the RFP objective of an interoperable naming service, we have addressed several other items listed below.

Issues Open Against the Naming Service

The OMG issues database contains a number of open issues raised against the Naming Service affecting interoperability. We felt that any revised service should at least address issues against the previous version. The updated specification presented here addresses these issues as follows:

- Issue 24, 280, return type of `list` incorrect

  No change. Fixing this would require changing an existing interface which is not possible without losing backward compatibility. This issue does not affect interoperability.

- Issues 64, 271, 272, 273, 274, values returned from iterator operations

This specification defines the semantics of iterator operations more precisely to avoid the ambiguities in the original specification. The semantics permit an implementation to avoid read-ahead or buffering during iteration.

- Issue 275, meaning of `how_many` in the `list` and `next_n` operations

  For `list`, a `how_many` value of zero indicates that all bindings are to be returned via an iterator. For `next_n`, a `how_many` value of zero raises an exception.

- Issue 276, fewer than `how_many` bindings returned from iterator

  This specification makes it clear under what circumstances an implementation can return fewer than the requested number of bindings.

- Issues 277, 278, destruction of iterators

  We rejected the idea of destroying an iterator as soon as the last binding is retrieved as “too clever”. However, the text makes it clear that iterators can be destroyed by an implementation without warning, and that clients must be prepared to handle `OBJECT_NOT_EXIST` exceptions from iterator operations.

- Issue 279, inheritance from `LifeCycleObject`

  No change. Fixing this would require changing an existing interface which is not possible without losing backward compatibility.

- Issue 281, Names Library benefit

  The Names Library is removed by this document (see section 2.2.1).

- Issue 298, meaning of `why` member in `NotFound` exception

  The semantics of the `why` member are specified.

- Issue 270, semantics of name equality

  This specification has well-defined semantics for equality of names.

2.2 Deprecations and Limitations

2.2.1 Names Library

The Names Library as described in the original specification claims that it hides the representation of names from client code and therefore permits the representation of names to evolve without affecting existing clients. However, the library as specified does not provide this functionality—the representation of names is just as visible in the Names Library as it is in the basic IDL. In fact, the Names Library offers only very few benefits (comparison and random insertion) over the basic IDL.

The Names Library requires functionality that can only be implemented as client-side code (is not provided by distributable object interfaces). This requires porting of the Names Library to all possible combinations of operating system, language binding, compiler version, and ORB version. In addition, we believe that specifying such client-side functionality is undesirable because it invalidates transparencies provided by CORBA, such as language, platform, and implementation independence.

The Names Library is therefore removed by this specification.

2.2.2 No Support for Wide Characters

The RFP calls for submissions upwardly compatible with the existing Naming Service. We do not see a way to provide this if names are allowed to contain wide characters. In addition, current ORBs do not yet support wide characters and strings, and language bindings for wide strings are still incomplete, which prohibits even proof-of-concept implementations. Therefore, this submission makes no attempt to support names containing wide characters.

Initial References and URLs 3

Note – Editing Instructions: The entire section 3.1 is to be inserted into the CORBA 2.3 Specification between section 4.7 Obtaining Initial Object References and the section 4.8 CurrentObject. Note that references to CORBA 2.3 refer to the draft chapters in ptc/98-10-xx. If the section numbers for CORBA 2.3 in these documents change before final publications, the cross references in this document must be adjusted.

3.1 Configuring Initial Service References

3.1.1 ORB-specific Configuration

It is required that an ORB can be administratively configured to return an arbitrary object reference from `CORBA::ORB::resolve_initial_references` for non-locality-constrained objects.

In addition to this required implementation-specific configuration, two `CORBA::ORB_init` arguments are provided to override the ORB initial reference configuration.

3.1.2 ORBInitRef

The ORB initial reference argument, `-ORBInitRef`, allows specification of an arbitrary object reference for an initial service. The format is:

```
-ORBInitRef <ObjectID>=<ObjectURL>
```

Examples of use are:

```
-ORBInitRef NameService=IOR:00230021AB...
```

```
-ORBInitRef \
NotificationService=iioploc://555objs.com/NotificationService
```

```
-ORBInitRef TradingService=iiopname://555objs.com/Dev/Trader
```

<ObjectID> represents the well-known ObjectID for a service defined in the CORBA specification, such as NameService. This mechanism allows an ORB to be configured with new initial service Object IDs that were not defined when the ORB was installed.

<ObjectURL> can be any of the URL schemes supported by CORBA::ORB::string_to_object (Sections 13.6.6 to 13.6.7 CORBA 2.3 Specification). If a URL is syntactically malformed or can be determined to be invalid in an implementation defined manner, ORB_init raises a BAD_PARAM exception.

Note – Editing Instructions: The iioploc and iiopname URL schemes are described in section 3.2 of this submission. That section is to be placed into the CORBA 2.3 document as the (new) section 13.6.7.

3.1.3 ORBDefaultInitRef

The ORB default initial reference argument, -ORBDefaultInitRef, assists in resolution of initial references not explicitly specified with -ORBInitRef. -ORBDefaultInitRef requires a URL that, after appending a slash '/' character and a stringified object key, forms a new URL to identify an initial object reference. For example:

```
-ORBDefaultInitRef iioploc://555objs.com
```

A call to resolve_initial_references("NotificationService") with this argument results in a new URL:

```
iioploc://555objs.com/NotificationService
```

That URL is passed to CORBA::ORB::string_to_object to obtain the initial reference for the service.

Another example is:

```
-ORBDefaultInitRef \
iiopname://555ResolveRefs.com,555Backup.com/Prod/Local
```

After calling resolve_initial_references("NameService"), one of the iiopname URLs

```
iiopname://555ResolveRefs.com/Prod/Local/NameService
```

or

```
iiopname://555Backup411.com/Prod/Local/NameService
```

is used to obtain an object reference from string_to_object. (In this example, Prod/Local/NameService represents a stringified CosNaming::Name).

Section 13.6.7 provides details of the `iioploc` and `iiopname` URL schemes. The `-ORBDefaultInitRef` argument naturally extends to URL schemes that may be defined in the future, provided the final part of the URL is an object key.

3.1.4 Configuration Effect on `resolve_initial_references`

Default Resolution Order

The default order for processing a call to `CORBA::ORB::resolve_initial_references` for a given `<ObjectID>` is:

1. Resolve with `-ORBInitRef` for this `<ObjectID>` if possible
2. Resolve with an `-ORBDefaultInitRef` entry if possible
3. Resolve with pre-configured ORB settings.

ORB Configured Resolution Order

There are cases where the default resolution order may not be appropriate for all services and use of `-ORBDefaultInitRef` may have unintended resolution side effects. For example, an ORB may use a proprietary service, such as `ImplementationRepository`, for internal purposes and may want to prevent a client from unknowingly diverting the ORB's reference to an implementation repository from another vendor. To prevent this, an ORB is allowed to ignore the `-ORBDefaultInitRef` argument for any or all `<ObjectID>`s for those services that are not OMG-specified services with a well-known service name as accepted by `resolve_initial_references`. An ORB can only ignore the `-ORBDefaultInitRef` argument but must always honor the `-ORBInitRef` argument.

3.1.5 Configuration Effect on `list_initial_services`

The `<ObjectID>`s of all `-ORBInitRef` arguments to `ORB_init` appear in the list of tokens returned by `list_initial_services` as well as all ORB-configured `<ObjectID>`s. Any other tokens that may appear are implementation-dependent.

The list of `<ObjectID>`s returned by `list_initial_services` can be a subset of the `<ObjectID>`s recognized as valid by `resolve_initial_references`.

3.2 Object URLs

Note – EDITING INSTRUCTIONS. This section is to be added as a new section after Section 13.6.6 Stringified Object References of the CORBA 2.3 Specification.

To address the problem of bootstrapping and allow for more convenient exchange of human-readable object references, `ORB::string_to_object` allows URLs in the `iioploc` and `iiopname` formats to be converted into object references. If conversion fails, `string_to_object` raises a `BAD_PARAM` exception with the following minor codes (to be assigned by the OMG):

- `BadSchemeName`
- `BadAddress`
- `BadSchemeSpecificPart`
- `Other`

iioploc URL

The `iioploc` URL scheme provides stringified object references that are easily manipulated in TCP/IP- and DNS-centric environments such as the Internet. An `iioploc` URL contains:

- an address
- an IIOP version number
- an object key

The object address and key are specified directly in the URL. Examples of `iioploc` URLs are:

```
iioploc://1.1@555xyz.com:9999/Dev/NameService
iioploc://555xyz.com/Prod/TradingService
```

The full syntax is:

```
<iioploc>     = “iioploc://”[<addr_list>][“/”<key_string>]
<addr_list>= [<address> “,”]* <address>
<address>   = [<version> <host> [“:” <port>]]
<host>      = DNS-style Host Name | ip_address
<version>   = <major> “.” <minor> “@” | empty_string
<port>      = number
<major>     = number
<minor>     = number
<key_string>= <string> | empty_string
```

Where:

addr_list: comma-separated list of addresses that is used in an implementation-defined manner to address this object

address: a single address

host: DNS-style host name or IP address. If not present, the local host is assumed.

version: a major and minor version number, separated by ‘.’ and followed by ‘@’. If the version is absent, 1.0 is assumed.

ip_address: numeric IP address (dotted decimal notation)

port: port number the agent is listening on (see below). Default is 9999.

key_string: a stringified object key

The `key_string` corresponds to the octet sequence in the `object_key` member of a GIOP `Request` or `LocateRequest` header as defined in section 15.4 of CORBA 2.3. The `key_string` uses the escape conventions described in RFC 2396 to map away from octet values that cannot directly be part of a URL. US-ASCII alphanumeric characters are not escaped. Characters outside this range are escaped, except for the following:

";" | "/" | ":" | "?"| ":" | "@" | "&" | "=" | "+" | "$" |
"," | "-" | "_" | "." | "!" | "~" | "*" | "'" | "(" | ")"

The `key_string` is not NUL-terminated.

`iioploc` Server Implementation

The only requirements on an object advertised by an `iioploc` URL are that there must be a software agent listening on the host and port specified by the URL. This agent must be capable of handling IIOP `Request` and `LocateRequest` messages targeted at the object key specified in the URL.

A normal CORBA server meets these criteria. It is also possible to implement lightweight object location forwarding agents that respond to IIOP `Request` messages with `Reply` messages with a `LOCATION_FORWARD` status, and respond to IIOP `LocateRequest` messages with `LocateReply` messages.

`iiopname` URL

The `iiopname` URL scheme is described in Chapter 3 of the CORBAservices specification. It extends the capabilities of the `iioploc` scheme to allow URLs to denote entries in a Naming Service. Resolving `iiopname` URLs does not require a Naming Service implementation in the ORB core. An example is:

```
iiopname://555objs.com/a/string/path/to/obj
```

This URL specifies that at host `555objs.com`, a object of type `NamingContext` (with an object key of `NameService`) can be found, or alternatively, that an agent is running at that location which will return a reference to a `NamingContext`. The (stringified) name `a/string/path/to/obj` is then used as the argument to a `resolve` operation on that `NamingContext`. The URL denotes the object reference that results from that lookup.

Future URL Schemes

Several currently defined non-CORBA URL scheme names are reserved. Implementations may choose to provide these or other URL schemes to support additional ways of denoting objects with URLs.

Table 3-1 lists the required and some optional formats.

Table 3-1 URL formats

Scheme	Description	Status
IOR:	Standard stringified IOR format	Required
iioploc:	IIOP specific stringified object reference	Required
iiopname:	IIOP CosName URL	Required
file://	Specifies a file containing a URL/IOR	Optional
ftp://	Specifies a file containing a URL/IOR that is accessible via ftp protocol.	Optional
http://	Specifies an HTTP URL that returns an object URL/IOR.	Optional

Naming Service Specification 4

4.1 Service Description

Note – EDITING INSTRUCTIONS - This chapter is a replacement for the CORBA Services Specification Chapter 3.

4.1.1 Overview

A name-to-object association is called a *name binding*. A name binding is always defined relative to a *naming context*. A naming context is an object that contains a set of name bindings in which each name is unique. Different names can be bound to an object in the same or different contexts at the same time. There is no requirement, however, that all objects must be named.

To *resolve a name* is to determine the object associated with the name in a given context. To *bind a name* is to create a name binding in a given context. A name is always resolved relative to a context — there are no absolute names.

Because a context is like any other object, it can also be bound to a name in a naming context. Binding contexts in other contexts creates a *naming graph* — a directed graph with nodes and labeled edges where the nodes are contexts. A naming graph allows more complex names to reference an object. Given a context in a naming graph, a sequence of names can reference an object. This sequence of names (called a *compound name*) defines a path in the naming graph to navigate the resolution process. Figure 4-1 shows an example of a naming graph.

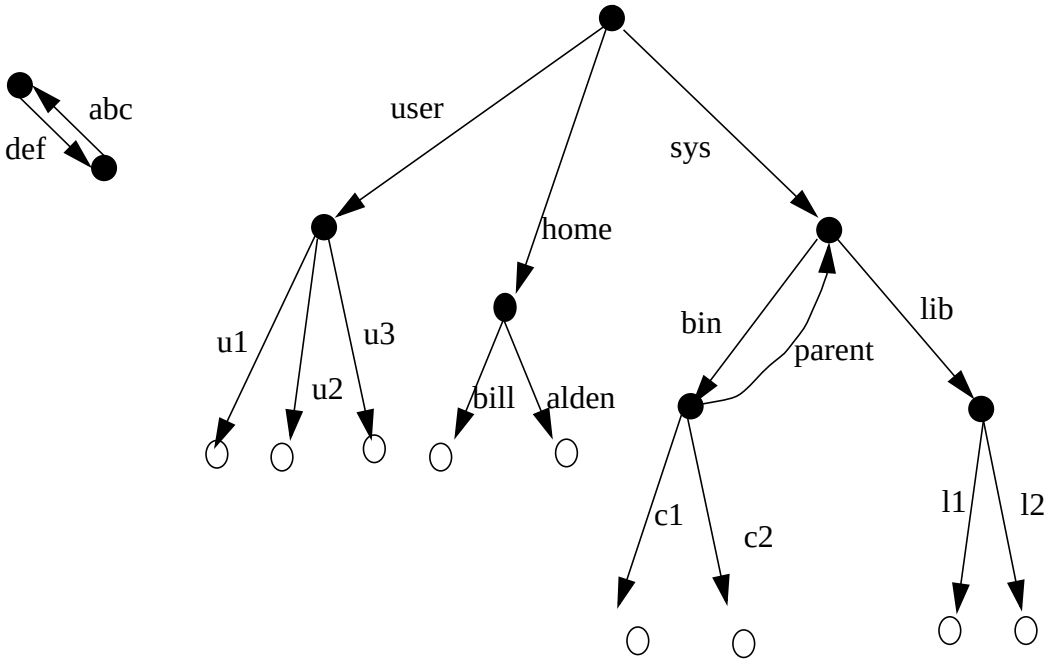

Figure 4-1 A Naming Graph

4.1.2 Names

Many of the operations defined on a naming context take names as parameters. Names have structure. A name is an ordered sequence of *components*.

A name with a single component is called a *simple name*; a name with multiple components is called a *compound name*. Each component except the last is used to name a context; the last component denotes the bound object. The notation:

```
component1/component2/component3
```

indicates a sequence of components.

Note – The slash (/) characters are simply a notation used here and are not intended to imply that names are sequences of characters separated by slashes.

A name component consists of two attributes: the *`id` attribute* and the *`kind` attribute*. Both the `id` attribute and the `kind` attribute are represented as IDL strings.

The `kind` attribute adds descriptive power to names in a syntax-independent way. Examples of the value of the `kind` attribute include *c_source*, *object_code*, *executable*, *postscript*, or “ ”. The naming system does not interpret, assign, or manage these values in any way. Higher levels of software may make policies about the use and management of these values. This feature addresses the needs of applications that use

syntactic naming conventions to distinguish related objects. For example Unix uses suffixes such as `.c` and `.o`. Applications (such as the C compiler) depend on these syntactic convention to make name transformations (for example, to transform `foo.c` to `foo.o`).

A sequence of `id` and `kind` pairs forming a name can be expressed as a single string using the syntax described in section 4.5. This allows names to be written down easily or to be presented as a strings in user interfaces. In addition, section 4.6 describes a way to express a name relative to a particular naming context in URL format. The URL representation provides a human-readable form of an object reference that is named in some naming context.

4.1.3 Example Scenarios

This section provides two short scenarios that illustrate how the naming service specification can be used by two fairly different kinds of systems -- systems that differ in the kind of implementations used to build the Naming Service and that differ in models of how clients might use the Naming Service with other object services to locate objects.

In one system, the Naming Service is implemented using an underlying enterprise-wide naming server such as DCE CDS. The Naming Service is used to construct large, enterprise-wide naming graphs where NamingContexts model "directories" or "folders" and other names identify "document" or "file" kinds of objects. In other words, the naming service is used as the backbone of an enterprise-wide filing system. In such a system, non-object-based access to the naming service may well be as commonplace as object-based access to the naming service.

The Naming Service provides the principal mechanism through which most clients of an ORB-based system locate objects that they intend to use (make requests of). Given an initial naming context, clients navigate naming contexts retrieving lists of the names bound to that context. In conjunction with properties and security services, clients look for objects with certain "externally visible" characteristics, for example, for objects with recognized names or objects with a certain time-last-modified (all subject to security considerations). All objects used in such a scheme register their externally visible characteristics with other services (a name service, a properties service, and so on).

Conventions are employed in such a scheme that meaningfully partition the name space. For example, individuals are assigned naming contexts for personal use, groups of individuals may be assigned shared naming contexts while other contexts are organized in a public section of the naming graph. Similarly, conventions are used to identify contexts that list the names of services that are available in the system (e.g., that locate a translation or printing service).

In an alternative system, the Naming Service can be used in a more limited role and can have a less sophisticated implementation. In this model, naming contexts represent the types and locations of services that are available in the system and a much shallower naming graph is employed. For example, the Naming Service is used to register the object references of a mail service, an information service, a filing service.

Given a handful of references to "root objects" obtained from the Naming Service, a client uses the Relationship and Query Services to locate objects contained in or managed by the services registered with the Naming Service. In such a system, the Naming Service is used sparingly and instead clients rely on other services such as query services to navigate through large collections of objects. Also, objects in this scheme rarely register "external characteristics" with another service - instead they support the interfaces of Query or Relationship Services.

Of course, nothing precludes the Naming Service presented here from being used to provide both models of use at the same time. These two scenarios demonstrate how this specification is suitable for use in two fairly different kinds of systems with potentially quite different kinds of implementations. The service provides a basic building block on which higher-level services impose the conventions and semantics which determine how frameworks of application and facilities objects locate other objects.

4.1.4 Design Principles

Several principles have driven the design of the Naming Service:

1. The design imparts no semantics or interpretation of the names themselves; this is up to higher-level software.
2. The design supports distributed, heterogeneous implementation and administration of names and name contexts.
3. Naming service clients need not be aware of the physical site of name servers in a distributed environment, or which server interprets what portion of a compound name, or of the way that servers are implemented.
4. The Naming Service is a fundamental object service, with no dependencies on other interfaces.
5. Name contexts of arbitrary and unknown implementation may be utilized together as nested graphs of nodes that cooperate in resolving names for a client. No “universal” root is needed for a name hierarchy.
6. Existing name and directory services employed in different network computing environments can be transparently encapsulated using name contexts. All of the above features contribute to making this possible.
7. The design does not address namespace administration. It is the responsibility of higher-level software to administer the namespace.

4.2 The `CosNaming` Module

The `CosNaming` module is a collection of interfaces that together define the Naming Service. This module contains three interfaces:

- The `NamingContext` interface
- The `BindingIterator` interface

- The `NamingContextExt` interface

This section describes these interfaces and their operations in detail.

The `CosNaming` module is shown below.

Note – `Istring` was a “placeholder for a future IDL internationalized string data type” in the original specification. It is maintained solely for compatibility reasons.

```
// File: CosNaming.idl
#ifndef _COSNAMING_IDL_
#define _COSNAMING_IDL_

#pragma prefix "omg.org"

module CosNaming {
      typedef string Istring;

      struct NameComponent {
            Istring id;
            Istring kind;
      };
      typedef sequence<NameComponent> Name;

      enum BindingType { nobject, ncontext };

      struct Binding {
            Name          binding_name;
            BindingType     binding_type;
      };

      // Note: In struct Binding, binding_name is incorrectly defined
      // as a Name instead of a NameComponent. This definition is
      // unchanged for compatibility reasons.
      typedef sequence <Binding> BindingList;

      interface BindingIterator;

      interface NamingContext {
            enum NotFoundReason {
               missing_node, not_context, not_object
            };

            exception NotFound {
                  NotFoundReason  why;
                  Name                rest_of_name;
            };

            exception CannotProceed {
                  NamingContext   cxt;
                  Name                rest_of_name;
            };

            exception InvalidName{};


            exception AlreadyBound {};

            exception NotEmpty{};
```

```
        void    bind(in Name n, in Object obj)
                        raises(
                              NotFound, CannotProceed,
                              InvalidName, AlreadyBound
                        );

        void    rebind(in Name n, in Object obj)
                        raises(NotFound, CannotProceed, InvalidName);

        void    bind_context(in Name n, in NamingContext nc)
                        raises(
                           NotFound, CannotProceed,
                           InvalidName, AlreadyBound
                        );

        void    rebind_context(in Name n, in NamingContext nc)
                        raises(NotFound, CannotProceed, InvalidName);

        Object  resolve (in Name n)
                        raises(NotFound, CannotProceed, InvalidName);

        void    unbind(in Name n)
                    raises(NotFound, CannotProceed, InvalidName);

        NamingContext   new_context();
        NamingContext   bind_new_context(in Name n)
                                raises(
                                      NotFound, AlreadyBound,
                                      CannotProceed, InvalidName
                                );

        void    destroy() raises(NotEmpty);

        void    list(
                        in unsigned long       how_many,
                        out BindingList        bl,
                        out BindingIterator    bi
                    );

};

interface BindingIterator {
        boolean next_one(out Binding b);
        boolean next_n(in unsigned long how_many, out BindingList bl);
        void    destroy();
};

interface NamingContextExt: NamingContext {
        typedef string StringName;
        typedef string Address;
        typedef string URLString;
```

```
            StringName      to_string(in Name n) raises(InvalidName);
            Name            to_name(in StringName sn)
                                raises(InvalidName);

            exception InvalidAddress {};

            URLString       to_url(in Address addr, in StringName sn)
                                raises(InvalidAddress, InvalidName);

            Object          resolve_str(in StringName n)
                                raises(
                                    NotFound, CannotProceed,
                                    InvalidName, AlreadyBound
                                );
        };
};
#endif // _COSNAMING_IDL_
```

Resolution of Compound Names

In this specification operations that are performed on compound names recursively perform a resolve operation on all but the last component of a name before performing the operation on the final name component. The general form is defined as follows:

ctx->op(<c1; c2; ...; cn>) equiv

ctx->resolve(<c1>)->resolve(<c2; cn-1>)->op(<cn>)

where ctx is a naming context, <c1; ...; cn> a compound name, and op a naming context operation.

4.3 `NamingContext` Interface

The following sections describe the naming context data types and interface in detail.

4.3.1 Structures

`NameComponent`

```
struct NameComponent {
    Istring id;
    Istring kind;
};
```

A name component consists of two attributes: the identifier attribute, `id`, and the kind attribute, `kind`.

Both of these attributes are arbitrary-length strings of ISO Latin-1 characters, excluding the ASCII `NUL` character.

When comparing two NameComponents for equality both the id and the kind field must match in order for two NameComponents to be considered identical. This applies for zero-length (empty) fields as well.

An implementation may place limitations on the characters that may be contained in a name component, as well as the length of a name component. For example, an implementation may disallow certain characters, may not accept the empty string as a legal name component, or may limit name components to some maximum length.

Name

A name is a sequence of NameComponents. The empty sequence is not a legal name. An implementation may limit the length of the sequence to some maximum. When comparing Names for equality, each NameComponent in the first name must match the corresponding NameComponent in the second Name for the names to be considered identical.

Binding

```
enum BindingType { nobject, ncontext };
struct Binding {
    Name binding_name;
    BindingType binding_type;
};
typedef sequence<Binding> BindingList;
```

This types are used by the NamingContext::list, BindingIterator::next_n and BindingIterator::next_one operations. A Binding contains a Name in the member binding_name, together with the BindingType of that Name in the member binding_type.

Note – The binding_name member is incorrectly typed as a Name instead of a NameComponent. For compatibility with the original CosNaming specification this incorrect definition has been retained. The binding_name is used as a NameComponent and will always be a Name with length of 1.

The value of *binding_type* is *ncontext* if a Name denotes a binding created with one of the following operations:

- bind_context
- rebind_context
- bind_new_context

For bindings created with any other operation, the value of BindingType is nobject.

4.3.2 Exceptions

The Naming Service exceptions are defined below.

NotFound

```
exception NotFound {
        NotFoundReason  why;
        Name rest_of_name;
};
```

This exception is raised by operations when a component of a name does not identify a binding or the type of the binding is incorrect for the operation being performed. The `why` member explains the reason for the exception and the `rest_of_name` identifies the portion of the name that caused the error:

- `missing_node`

  The first name component in `rest_of_name` denotes a binding that is not bound under that name within its parent context.

- `not_context`

  The first name component in `rest_of_name` denotes a binding with a type of `nobject` when the type `ncontext` was required.

- `not_object`

  The first name component in `rest_of_name` denotes a binding with a type of `ncontext` when the type `nobject` was required.

CannotProceed

```
exception CannotProceed {
        NamingContext  cxt;
        Name rest_of_name;
};
```

This exception is raised when an implementation has given up for some reason. The client, however, may be able to continue the operation at the returned naming context.

The `cxt` member contains the context that the operation may be able to retry from.

The `rest_of_name` member contains the remainder of the non-working name.

InvalidName

```
exception InvalidName {};
```

This exception is raised if a `Name` is invalid. A name of length zero is invalid (containing no name components). Implementations may place further limitations on what constitutes a legal name and raise this exception to indicate a violation.

AlreadyBound

exception AlreadyBound {};

Indicates an object is already bound to the specified name. Only one object can be bound to a particular `Name` in a context.

NotEmpty

exception NotEmpty {};

This exception is raised by `destroy` if the `NamingContext` contains bindings. A `NamingContext` must be empty to be destroyed.

4.3.3 Binding Objects

The binding operations name an object in a naming context. Once an object is bound, it can be found with the `resolve` operation. The Naming Service supports four operations to create bindings: `bind`, `rebind`, `bind_context` and `rebind_context`. `bind_new_context` also creates a binding, see section 4.3.6.

```
void bind(in Name n, in Object obj)
        raises(NotFound, CannotProceed, InvalidName, AlreadyBound);
void rebind(in Name n, in Object obj)
        raises(NotFound, CannotProceed, InvalidName);
void bind_context(in Name n, in NamingContext nc)
        raises(NotFound, CannotProceed, InvalidName, AlreadyBound);
void rebind_context(in Name n, in NamingContext nc)
        raises(NotFound, CannotProceed, InvalidName);
```

bind

Creates an `nobject` binding in the naming context.

rebind

Creates an `nobject` binding in the naming context even if the name is already bound in the context.

If already bound, the previous binding must be of type `nobject`; otherwise, a `NotFound` exception with a `why` reason of `not_object` is raised.

bind_context

Creates an `ncontext` binding in the parent naming context. Attempts to bind a nil context raise a `BAD_PARAM` exception.

rebind_context

Creates an ncontext binding in the naming context even if the name is already bound in the context.

If already bound, the previous binding must be of type ncontext; otherwise, a NotFound exception with a why reason of not_context will be raised.

Usage

If a binding with the specified name already exists, bind and bind_context raise an AlreadyBound exception.

If an implementation places limits on the number of bindings within a context, bind and bind_context raise the IMP_LIMIT system exception if the new binding cannot be created.

Naming contexts bound using bind_context and rebind_context participate in name resolution when compound names are passed to be resolved; naming contexts bound with bind and rebind do not.

Use of rebind_context may leave a potential orphaned context (one that is unreachable within an instance of the Name Service). Policies and administration tools regarding potential orphan contexts are implementation-specific.

If rebind or rebind_context raise a NotFound exception because an already existing binding is of the wrong type, the rest_of_name member of the exception has a sequence length of 1.

4.3.4 Resolving Names

The resolve operation is the process of retrieving an object bound to a name in a given context. The given name must exactly match the bound name. The naming service does not return the type of the object. Clients are responsible for "narrowing" the object to the appropriate type. That is, clients typically cast the returned object from Object to a more specialized interface. The IDL definition of the resolve operation is:

```
Object resolve (in Name n)
        raises (NotFound, CannotProceed, InvalidName);
```

Names can have multiple components; therefore, name resolution can traverse multiple contexts. These contexts can be federated between different Naming Service instances.

4.3.5 Unbinding Names

The unbind operation removes a name binding from a context. The definition of the unbind operation is:

```
void unbind(in Name n)
        raises (NotFound, CannotProceed, InvalidName);
```

4.3.6 Creating Naming Contexts

The Naming Service supports two operations to create new contexts: `new_context` and `bind_new_context`.

```
NamingContext new_context();
NamingContext bind_new_context(in Name n)
            raises(NotFound, AlreadyBound, CannotProceed, InvalidName);
```

new_context

This operation returns a new naming context. The new context is not bound to any name.

bind_new_context

This operation creates a new context and creates an `ncontext` binding for it using the name supplied as an argument.

Usage

If an implementation places limits on the number of naming contexts, both `new_context` and `bind_new_context` can raise the `IMP_LIMIT` system exception if the context cannot be created. `bind_new_context` can also raise `IMP_LIMIT` if the bind would cause an implementation limit on the number of bindings in a context to be exceeded.

4.3.7 Deleting Contexts

The `destroy` operation deletes a naming context.

```
void destroy()
        raises(NotEmpty);
```

This operation destroys its naming context. If there are bindings denoting the destroyed context, these bindings are *not* removed. If the naming context contains bindings, the operation raises `NotEmpty`.

4.3.8 Listing a Naming Context

The `list` operation allows a client to iterate through a set of bindings in a naming context.

```
void list (in unsigned long how_many,
           out BindingList bl, out BindingIterator bi);
};
```

`list` returns the bindings contained in a context in the parameter `bl`. The `bl` parameter is a sequence where each element is a `Binding` containing a `Name` of length 1 representing a single `NameComponent`.

The `how_many` parameter determines the maximum number of bindings to return in the parameter `bl`, with any remaining bindings to be accessed through the returned `BindingIterator bi.`

- A non-zero value of `how_many` guarantees that `bl` contains at most `how_many` elements. The implementation is free to return fewer than the number of bindings requested by `how_many`. However, for a non-zero value of `how_many`, it may not return a `bl` sequence with zero elements unless the context contains no bindings.
- If `how_many` is set to zero, the client is requesting to use only the `BindingIterator bi` to access the bindings and `list` returns a zero length sequence in `bl`.
- The parameter `bi` returns a reference to an iterator object.
  - If the `bi` parameter returns a non-nil reference, this indicates that the call to `list` may not have returned all of the bindings in the context and that the remaining bindings (if any) must be retrieved using the iterator. This applies for all values of `how_many`.
  - If the `bi` parameter returns a nil reference, this indicates that the `bl` parameter contains all of the bindings in the context. This applies for all values of `how_many`.

4.4 The `BindingIterator` Interface

The `BindingIterator` interface allows a client to iterate through the bindings using the `next_one` or `next_n` operations:

If a context is modified in between calls to `list`, `next_one`, or `next_n`, the behavior of further calls to `next_one` or `next_n` is implementation-dependent.

```
interface BindingIterator {
    boolean next_one(out Binding b);
    boolean next_n(in unsigned long how_many,
                   out BindingList bl);
    void destroy();
};
```

next_one

The `next_one` operation returns the next binding. It returns true if it is returning a binding, false if there are no more bindings to retrieve. If `next_one` returns false, the returned `Binding` is indeterminate

Further calls to `next_one` after it has returned false have undefined behavior.

next_n

`next_n` returns, in the parameter `bl`, bindings not yet retrieved with `list` or previous calls to `next_n` or `next_one`. It returns true if `bl` is a non-zero length sequence; it returns false if there are no more bindings and `bl` is a zero-length sequence.

The `how_many` parameter determines the maximum number of bindings to return in the parameter `bl`:

- A non-zero value of `how_many` guarantees that `bl` contains at most `how_many` elements. The implementation is free to return fewer than the number of bindings requested by `how_many`. However, it may not return a `bl` sequence with zero elements unless there are no bindings to retrieve.
- A zero value of `how_many` is illegal and raises a `BAD_PARAM` system exception.

`next_n` returns false with a `bl` parameter of length zero once all bindings have been retrieved. Further calls to `next_n` after it has returned a zero-length sequence have undefined behavior.

destroy

The `destroy` operation destroys its iterator. If a client invokes any operation on an iterator after calling `destroy`, the operation raises `OBJECT_NOT_EXIST`.

4.4.1 Garbage Collection of Iterators

Clients that create iterators but never call `destroy` can cause an implementation to permanently run out of resources. To protect itself against this scenario, an implementation is free to destroy an iterator object at any time without warning, using whatever algorithm it considers appropriate to choose iterators for destruction. In order to be robust in the presence of garbage collection, clients should be written to expect `OBJECT_NOT_EXIST` from calls to an iterator and handle this exception gracefully.

4.5 Stringified Names

Names are sequences of name components. This representation makes it difficult for applications to conveniently deal with names for I/O purposes, human or otherwise. This specification defines a syntax for stringified names and provides operations to convert a name in stringified form to its equivalent sequence form and vice-versa (see section 4.6.4).

A stringified name represents one and only one `CosNaming::Name`. If two names are equal, their stringified representations are equal (and vice-versa).

The stringified name representation reserves use of the characters ‘/’, ‘.’, and ‘\’. The forward slash ‘/’ is a name component separator; the dot ‘.’ separates `id` and `kind` fields. The backslash ‘\’ is an escape character (see section 4.5.2).

4.5.1 Basic Representation of Stringified Names

A stringified name consists of the name components of a name separated by a ‘/’ character. For example, a name consisting of the components “a”, “b”, and “c” (in that order) is represented as

```
a/b/c
```

Stringified names use the ‘.’ character to separate `id` and `kind` fields in the stringified representation. For example, the stringified name

```
a.b/c.d
```

represents the `CosNaming::Name`:

Index	id	kind
0	a	b
1	c	d

If a name component in a stringified name does not contain a ‘.’ character, the entire component is interpreted as the `id` field, and the `kind` field is empty. For example:

```
a/./c.d/.e
```

corresponds to the `CosNaming::Name`:

Index	id	kind
0	a	<empty>
1	<empty>	<empty>
2	c	d
3	<empty>	e

4.5.2 Escape Mechanism

The backslash ‘\’ character escapes the reserved meaning of ‘/’, ‘.’, and ‘\’ in a stringified name. The meaning of any other character following a ‘\’ is reserved for future use.

NameComponent Separators

If a name component contains a ‘/’ slash character, the stringified representation uses the ‘\’ character as an escape. For example, the stringified name

```
a/x\/y\/z/b
```

represents the name consisting of the name components “a”, “x/y/z”, and “b”.

id and kind Fields

The backslash escape mechanism is also used for '.', so `id` and `kind` fields can contain a literal '.'. To illustrate, the stringified name

```
a\.b.c\.d/e.f
```

represents the `CosNaming::Name`:

Index	id	kind
0	a.b	c.d
1	e	f

The Escape Character

The escape character '\' must be escaped if it appears in a name component. For example, the stringified name:

```
a/b\\/c
```

represents the name consisting of the components "a", "b\", and "c".

4.6 URL schemes

This section describes the Uniform Resource Locator (URL) schemes available to represent a CORBA object and a CORBA object bound in a `NamingContext`.

4.6.1 IOR

The string form of an IOR (**IOR:<hex_octets**>) is a valid URL. The scheme name is **IOR** and the text after the ':' is defined in the CORBA 2.3 specification, Section 13.6.6. The IOR URL is robust and insulates the client from the encapsulated transport information and object key used to reference the object. This URL format is independent of Naming Service.

4.6.2 iioploc

It is difficult for humans to exchange IORs through non-electronic means because of their length and the text encoding of binary information. The `iioploc` URL scheme provides URLs that are familiar to people and similar to `ftp` or `http` URLs.

The `iioploc` URL is described in the CORBA 2.3 Specification, Section 13.6.6. This URL format is independent of the Naming Service.

4.6.3 iiopname

An `iiopname` URL is similar to an `iioploc` URL. However, an `iiopname` URL also contains a stringified name that identifies a binding in a naming context.

`iiopname` Examples

```
iiopname://1.1@myhost.555xyz.com:9999/a/b/c
```

The URL denotes a naming context at `myhost.xyz.com:9999` (possibly returned in a `LocateReply` or `LOCATION_FORWARD` reply by an agent listening at that address). The agent at that hosts supports IIOP version 1.1. The name `a/b/c` is resolved against that context to yield the object reference denoted by the URL.

```
iiopname:///x/y/z
```

This URL refers to an agent supporting IIOP version 1.0 on the local host and default port. The naming context associated with the object key `NameService` is used to resolve the name `x/y/z`, which yields the object reference denoted by the URL.

```
iiopname:///
```

This URL represents the naming context returned by the agent running on the local host at the default port. It is equivalent to `iioploc:///NameService`.

`iiopname` Syntax

The full `iiopname` BNF is:

```
<iiopname>    = "iiopname://"[<addr_list>]["/"<string_name>]
<addr_list>   = [<address> ","]* <address>
<address>     = [<version> <host> [":" <port>]]
<host>        = DNS Style Host Name | ip_address
<version>     = <major> "." <minor> "@" | empty_string
<port>        = number
<major>       = number
<minor>       = number
<string_name>= stringified Name | empty_string
```

Where:

addr_list: comma-separated list of addresses that is used in an implementation-defined manner to address this object.

address: A single address

host: DNS-style host name or IP address. If not present, the local host is assumed.

version: a major and minor version number, separated by '.' and followed by '@'. If the version is absent, 1.0 is assumed.

ip_address: numeric IP address (dotted decimal notation).

port: port number object is listening on. Default is 9999.

string_name: a stringified Name with URL escapes as defined in section .

Multiple Addresses in `iiopname`*s*

```
iiopname:555xyz.com,555backup.com/very/critical/binding
```

An implementation resolves the stringified name `very/critical/binding` through the naming context identified by either `555xyz.com` or `555backup.com`. The order of processing of the address list to obtain the naming context is implementation-dependent.

Note – Unlike stringified names, `iiopname`s cannot be compared directly for equality as the address specification can differ for `iiopname` URLs with the same meaning.

`iiopname` *Character Escapes*

`iiopname` URLs use the escape mechanism described in the Internet Engineering Task Force (IETF) RFC 2396. These escape rules insure that URLs can be transferred via a variety of transports without undergoing changes. The character escape rules for the stringified name portion of an `iiopname` are:

US-ASCII alphanumeric characters are not escaped. Characters outside this range are escaped, except for the following:

“;” | “/” | “:” | “?”| “:” | “@” | “&” | “=” | “+” | “$” |
“,” | “-” | “_” | ”.” | “!” | “~” | “*” | “’” | “(“ | “)”

`iiopname` *Escape Mechanism*

The percent ‘%’ character is used as an escape. If a character that requires escaping is present in a name component it is encoded as two hexadecimal digits following a “%” character to represent the octet. (The first hexadecimal character represent the high-order nibble of the octet, the second hexadecimal character represents the low-order nibble.) If a ‘%’ is not followed by two hex digits, the stringified name is syntactically invalid.

Examples

Table 4-1

Stringified Name	After URL Escapes	Comment
a.b/c.d	a.b/c.d	URL form identical
<a>.b/c.d	%3ca%3e.b/c.d	Escaped “<“ and “>”
a.b/ c.d	a.b/%20%20c.d	Escaped two “ “ spaces
a%b/c%d	a%25b/c%25d	Escaped two “%” percents

`iiopname` Resolution

`iiopname`s can be implemented using the `iioploc` URL scheme. Given an `iiopname`:

```
iiopname://<addresses>[“/” <string_name>]
```

The `iiopname` is resolved by:

1. First constructing an `iioploc` URL of the form: `iioploc://<addresses>/NamingService`. `NamingService` is the object key for `iiopname`s.
2. This is converted to a naming context object reference with `CORBA::ORB::string_to_object`.
3. The `<string_name>` is converted to a `CosNaming::Name`.
4. The resulting name is passed to a `resolve` operation on the naming context.
5. The object reference returned by the `resolve` is the result.

Implementations are not required to use the method described and may make optimizations appropriate to their environment.

4.6.4 Converting between CosNames, Stringified Names, and URLs

The `NamingContextExt` interface, derived from `NamingContext`, provides the operations required to use URLs and stringified names.

```
module CosNaming {
    // ...
    interface NamingContextExt: NamingContext {
        typedef string StringName;
        typedef string Address;
        typedef string URLString;

        StringName     to_string(in Name n) raises(InvalidName);
        Name           to_name(in StringName sn)
                           raises(InvalidName);

        exception InvalidAddress {};

        URLString      to_url(in Address addr, in StringName sn)
                           raises(InvalidAddress, InvalidName);

        Object         resolve_str(in StringName n)
                           raises(
                               NotFound, CannotProceed,
                               InvalidName, AlreadyBound
                           );
    };
};
```

to_string

This operation accepts a `Name` and returns a stringified name. If the `Name` is invalid, an `InvalidName` exception is raised.

to_name

This operation accepts a stringified name and returns a `Name`. If the stringified name is syntactically malformed or violates an implementation limit, an `InvalidName` exception is raised.

resolve_str

This is a convenience operation that performs a resolve in the same manner as `NamingContext::resolve`. It accepts a stringified name as an argument instead of a `Name`.

to_url

This operation takes a URL `<address>` component such as

- `myhost.xyz.com`
- `myhost.555xyz.com,my_backup_host.555xyz.com:900`

and a stringified name. It then performs any escapes necessary on the stringified name and returns a fully formed URL string. An exception is raised if either the protocol or name parameters are invalid.

It is legal for the address and/or stringified_name to be empty. If the address is empty, it means the local host.

URL to Object Reference

Conversions from URLs to objects are handled by `CORBA::ORB::string_to_object` as described in the CORBA 2.3 Specification, Section 13.6.6.

4.7 Initial Reference to a NamingContextExt

An initial reference to an instance of this interface can be obtained by calling `resolve_initial_references` with an `ObjectID` of `NameService`.

4.8 Conformance Requirements

4.8.1 Optional Interfaces

There are no optional interfaces in this specification. A compliant implementation must implement all of the functionality and interfaces described.

4.8.2 Documentation Requirements

A compliant implementation must document all of the following:

- any limitations to the character values or character sequences that may be used in a name component
- any limitations to the length (including minimum or maximum) of a name component
- any limitations to number of name components in a name
- any limitations to the maximum number of bindings in a context
- any limitations to the total number of bindings (implementation-wide)
- any limitations to the maximum number of contexts
- the means provided to deal with orphaned contexts and bindings
- Any policy for dealing with potentially orphaned naming contexts. Orphaned contexts are contexts that are not bound in any other context within a naming server.
- Any policy for destroying binding iterators that are considered to be no longer in use.
- Under what circumstances, if any, a `CannotProceed` exception is raised.