CHAPTER 17

Naming Service

The Orbix naming service lets you associate names with objects. Servers can register object references by name with the naming service repository, and advertise those names to clients. Clients, in turn, can resolve the desired objects in the naming service by supplying the appropriate name.

The Orbix naming service implements the OMG COS Interoperable Naming Service, which describes how applications can map object references to names.

Benefits

Using the naming service can offer the following benefits:

- Clients can locate objects through standard names that are independent of the corresponding object references. This affords greater flexibility to developers and administrators, who can direct client requests to the most appropriate implementation. For example, you can make changes to an object's implementation or its location that are transparent to the client.
- The naming service provides a single repository for object references. Thus, application components can rely on it to obtain an application's initial references.

In this chapter

This chapter describes how to build and maintain naming graphs programmatically. It also shows how to use object groups to achieve load balancing. It contains these sections:

Many operations that are discussed here can also be executed administratively with Orbix tools. For more information about these and related configuration options, refer to the *Application Server Platform Administrator's Guide*.

Naming Service Design

Naming graph organization

The naming service is organized into a *naming graph*, which is equivalent to a directory system. A naming graph consists of one or more *naming contexts*, which correspond to directories. Each naming context contains zero or more name-reference associations, or *name bindings*, each of which refers to another node within the naming graph. A name binding can refer either to another naming context or to an object reference. Thus, any path within a naming graph finally resolves to either a naming context or an object reference. All bindings in a naming graph can usually be resolved via an *initial naming context*.

Example

Figure 23 shows how the Account interface described in earlier chapters might be extended (through inheritance) into multiple objects, and organized into a hierarchy of naming contexts. In this graph, hollow nodes are naming contexts and solid nodes are application objects. Naming contexts are typically intermediate nodes, although they can also be leaf nodes; application objects can only be leaf nodes.

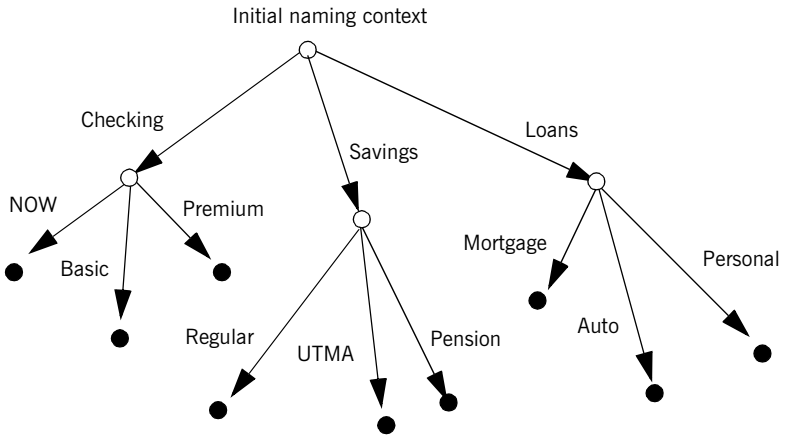

Figure 23: *A naming graph is a hierarchy of naming contexts*

Each leaf node in this naming graph associates a name with a reference to an account object such as a basic checking account or a personal loan account. Given the full path from the initial naming context—for example, `Savings/Regular`—a client can obtain the associated reference and invoke requests on it.

The operations and types that the naming service requires are defined in the IDL file `CosNaming.idl`. This file contains a single module, `CosNaming`, which in turn contains three interfaces: `NamingContext`, `NamingContextExt`, and `BindingIterator`.

Defining Names

Name sequence

A naming graph is composed of `Name` sequences of `NameComponent` structures, defined in the `CosNaming` module:

```
module CosNaming{
    typedef string Istring;
    struct NameComponent {
            Istring id;
            Istring kind;
    }
    typedef  sequence<NameComponent> Name;
    ...
};
```

A `Name` sequence specifies the path from a naming context to another naming context or application object. Each name component specifies a single node along that path.

Name components

Each name component has two string members:

- The `id` field acts as a name component's principle identifier. This field must be set.
- The `kind` member is optional; use it to further differentiate name components, if necessary.

Both `id` and `kind` members of a name component are used in name resolution. So, the naming service differentiates between two name components that have the same ids but different kinds.

For example, in the naming graph shown in Figure 23 on page 433, the path to a Personal loan account object is specified by a `Name` sequence in which only the `id` fields are set:

Figure 0.1:

Index	id	kind
0	Loans	
1	Personal	

In order to bind another Personal account object to the same Loan naming context, you must differentiate it from the existing one. You might do so by setting their `kind` fields as follows:

Figure 0.2:

Index	id	kind
0	Loans	
1	Personal	unsecured
1	Personal	secured

Note: If the `kind` field is unused, it must be set to an empty string.

Representing Names as Strings

The `CosNaming::NamingContextExt` interface defines a `StringName` type, which can represent a `Name` as a string with the following syntax:

`id[.kind][/id[.kind] ] ...`

Name components are delimited by a forward slash (`/`); `id` and `kind` members are delimited by a period (`.`). If the name component contains only the `id` string, the `kind` member is assumed to be an empty string.

`StringName` syntax reserves the use of three characters: forward slash (`/`), period (`.`), and backslash (`\`). If a name component includes these characters, you can use them in a `StringFormat` by prefixing them with a backslash (`\`) character.

The `CosNaming::NamingContextExt` interface provides several operations that allow conversion between `StringName` and `Name` data:

- `to_name()` converts a `StringName` to a `Name` (see page 438).
- `to_string()` converts a `Name` to a `StringName` (see page 440).
- `resolve_str()` uses a `StringName` to find a `Name` in a naming graph and returns an object reference (see page 450).

Note: You can invoke these and other `CosNaming::NamingContextExt` operations only on an initial naming context that is narrowed to `CosNaming::NamingContextExt`.

Initializing a Name

You can initialize a `CosNaming::Name` sequence in one of two ways:

- Set the members of each name component.
- Call `to_name()` on the initial naming context and supply a `StringName` argument. This operation converts the supplied string to a `Name` sequence.

Setting name component members

Given the loan account objects shown earlier, you can set the name for an unsecured personal loan as follows:

Example 38: *Initializing a name*

```
org.omg.CosNaming.NameComponent[] name =
    new org.omg.CosNaming.NameComponent[]
{
    new NameComponent("Loans", "");
    new nameComponent("Personal", "unsecured");
};
```

Converting a stringname to a name

The name shown in the previous example can also be set in a more straightforward way by calling `to_name()` on the initial naming context (see "Obtaining the Initial Naming Context" on page 441):

Example 39: *Using to_name() to initialize a Name*

```
// get initial naming context
org.omg.CosNaming.NamingContextExt root_cxt = ...;

org.omg.CosNaming.NameComponent[] name =
    root_cxt.to_name("Loans/Personal.unsecured");
```

The `to_name()` operation takes a string argument and returns a`CosNaming::Name`, which the previous example sets as follows:

Figure 0.3:

Index	id	kind
0	Loans	

Figure 0.3:

Index	id	kind
1	Personal	unsecured

Converting a Name to a StringName

You can convert a `CosNaming::Name` to a `CosNamingExt::StringName` by calling `to_string()` on the initial naming context. This lets server programs to advertise human-readable object names to clients.

For example, the following code converts `Name` sequence `name` to a `StringName`:

Example 40: *Converting a Name to a StringName*

```
// get initial naming context
org.omg.CosNaming.NamingContextExt root_cxt = ...;

// initialize name
org.omg.CosNaming.NameComponent[] name = ...;

...
org.omg.CosNaming.NamingContextExt.StringName str_n;
str_n = root_cxt.to_string(name);
```

Obtaining the Initial Naming Context

Clients and servers access a naming service through its initial naming context, which provides the standard entry point for building, modifying, and traversing a naming graph. To obtain the naming service's initial naming context, call `resolve_initial_references()` on the ORB. For example:

Example 41: *Obtaining the inital naming context*

```
// Initialize the ORB
global_orb = org.omg.CORBA.ORB.init(args, null);
// Get reference to initial naming context
org.omg.CORBA.Object obj =
   global_var.resolve_initial_references("NameService");
```

To obtain a reference to the naming context, narrow the result with `CosNaming.NamingContextExtHelper.narrow()`:

```
...
org.omg.CosNaming.NamingContextExt root_cxt;
if (root_cxt =
        org.omg.CosNaming.NamingContextExtHelper.narrow(obj)) {

} else {...} // Deal with failure to narrow()
...
```

A naming graph's initial naming context is equivalent to the root directory. Later sections show how you use the initial naming context to build and modify a naming graph, and to resolve names to object references.

Note: The `NamingContextExt` interface provides extra functionality over the `NamingContext` interface; therefore, the code in this chapter assumes that an initial naming context is narrowed to the `NamingContextExt` interface

Building a Naming Graph

A name binding can reference either an object reference or another naming context. By binding one naming context to another, you can organize application objects into logical categories. However complex the hierarchy, almost all paths within a naming graph hierarchy typically resolve to object references.

In an application that uses a naming service, a server program often builds a multi-tiered naming graph on startup. This process consists of two repetitive operations:

- Bind naming contexts into the desired hierarchy.
- Bind objects into the appropriate naming contexts.

Binding Naming Contexts

A server that builds a hierarchy of naming contexts contains the following steps:

1. Gets the initial naming context (see page 441).
2. Creates the first tier of naming contexts from the initial naming context.
3. Binds the new naming contexts to the initial naming context.
4. Adds naming contexts that are subordinate to the first tier:
   - Creates a naming context from any existing one.
   - Binds the new naming context to its designated parent.

The naming graph shown in Figure 23 on page 433 contains three naming contexts that are directly subordinate to the initial naming context: Checking, Loans, and Savings. The following code binds the Checking naming context to the initial naming context, as shown in Figure 24:

Example 42: *Binding a naming context to the initial naming context*

```
//get initial naming context
org.omg.CosNaming.NamingContextExt root_cxt = ...;

// create naming context
org.omg.CosNaming.NamingContext checking_cxt =
    root_cxt.new_context();

// initialize name
org.omg.CosNaming.NameComponent[] name = new NameComponent[1];
name[0] = new NameComponent("Checking", "");

// bind new context
root_cxt.bind_context(name, checking_cxt);
```

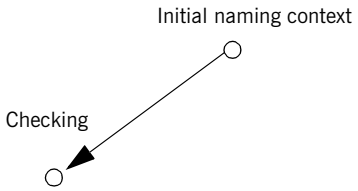

Figure 24: *Checking context bound to initial naming context*

Similarly, you can bind the Savings and Loans naming contexts to the initial naming context. The following code uses the shortcut operation `bind_new_context()`, which combines `new_context()` and `bind()`. It also uses the `to_name()` operation to set the `Name` variable.

Example 43: *Binding a naming context with bind_new_context()*

```
org.omg.CosNaming.NamingContext savings_cxt, loan_cxt;

// create naming contexts
name = root_cxt.to_name("Savings");
savings_cxt = root_cxt.bind_new_context(name);

name = root_cxt.to_name("Loan");
loan_cxt = root_cxt.bind_new_context(name);
```

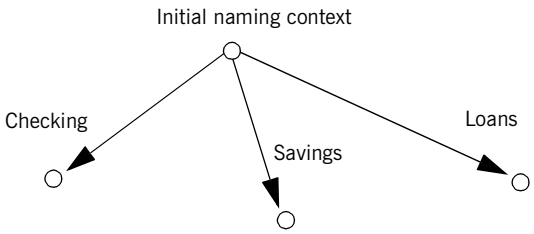

Figure 25: *Savings and Loans naming contexts bound to initial naming context*

Orphaned naming contexts

The naming service can contain naming contexts that are unbound to any other context. Because these naming contexts have no parent context, they are regarded as *orphaned*. Any naming context that you create with

new_context() is orphaned until you bind it to another context. Although it has no parent context, the initial naming context is not orphaned inasmuch as it is always accessible through resolve_initial_references(), while orphan naming contexts have no reliable means of access.

You might deliberately leave a naming context unbound—for example, you are in the process of constructing a new branch of naming contexts but wish to test it before binding it into the naming graph. Other naming contexts might appear to be orphaned within the context of the current naming service; however, they might actually be bound to a federated naming graph in another naming service (see "Federating Naming Graphs" on page 460).

Erroneous usage of orphaned naming contexts

Orphaned contexts can also occur inadvertently, often as a result of carelessly written code. For example, you can create orphaned contexts as a result of calling rebind() or rebind_context() to replace one name binding with another (see "Rebinding" on page 448). The following code shows how you might orphan the Savings naming context:

Example 44: *Orphaned naming contexts*

```
//get initial naming context
org.omg.CosNaming.NamingContextExt root_cxt = ...;

org.omg.CosNaming.NamingContext savings_cxt;

// initialize name
org.omg.CosNaming.NameComponent[] name = new NameComponent[1];
name[0] = new NameComponent("Savings", "");

// create and bind checking_cxt
savings_cxt = root_cxt.bind_new_context(name);

// make another context
org.omg.CosNaming.NamingContext savings_cxt2;
savings_cxt2 = root_cxt.new_context();

// bind savings_cxt2 to root context, savings_cxt now orphaned!
root_cxt.rebind_context(name, savings_cxt2);
```

An application can also create an orphan context by calling unbind() on a context without calling destroy() on the same context object (see "Maintaining the Naming Service" on page 458).

In both cases, if the application exits without destroying the context objects, they remain in the naming service but are inaccessible and cannot be deleted.

Binding Object References

After you construct the desired hierarchy of naming contexts, you can bind object references to them with the `bind()` operation. The following example builds on earlier code to bind a Basic checking account object to the Checking naming context:

Example 45: *Binding an object reference*

```
// object reference "basic_check" obtained earlier
...

name[0] = new NameComponent("Basic", "");
checking_cxt.bind(name, basic_check);
```

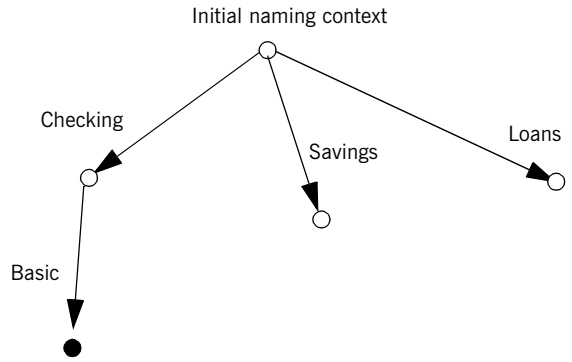

Figure 26: *Binding an object reference to a naming context*

The previous code assumes the existence of a `NamingContext` variable for the `Checking` naming context on which you can invoke `bind()`. Alternatively, you can invoke `bind()` on the initial naming context in order to bind `Basic` into the naming graph:

```
name = root_cxt.to_name("Checking/Basic");
root_cxt.bind(name, basic_check);
```

Note: Because the initial naming context is always available, it is the most reliable way to access all other contexts within a naming graph.

Rebinding

If you call `bind()` or `bind_context()` on a naming context that already contains the specified binding, the naming service throws an exception of `AlreadyBound`. To ensure the success of a binding operation whether or not the desired binding already exists, call one of the following naming context operations:

- `rebind()` rebinds an application object.
- `rebind_context()` rebinds a naming context.

Either operation replaces an existing binding of the same name with the new binding. Calls to `rebind()` in particular can be useful on server startup, to ensure that the naming service has the latest object references.

Note: Calls to `rebind_context()` or `rebind()` can have the undesired effect of creating orphaned naming contexts (see page 444). In general, exercise caution when calling either function.

Using Names to Access Objects

A client application can use the naming service to obtain object references in three steps:

1. Obtain a reference to the initial naming context (see page 441).
2. Set a CosNaming::Name structure with the full path of the name associated with the desired object.
3. Resolve the name to the desired object reference.

Setting object names

You specify the path to the desired object reference in a `CosNaming::Name`. You can set this name in one of two ways:

Explicitly set the `id` and `kind` members of each `Name` element. For example, the following code sets the name of a Basic checking account object:

Example 46: *Setting object name components*

```
org.omg.CosNaming.NameComponent[] name =
    new NameComponent[2];
name[0] = new NameComponent("Checking", "");
name[1] = new NameComponent("Basic", "");
```

Call `to_name()` on the initial naming context. This option is available if the client code narrows the initial naming context to the `NamingContextExt` interface. `to_name()` takes a `CosNaming::CosNamingExt::StringName` argument and returns a `CosNaming::Name` as follows:

Example 47: *Setting an object name with to_name()*

```
org.omg.CosNaming.NameComponent[] name =
    root_cxt.to_name("Checking/Basic");
```

For more about using a `StringName` with `to_name()`, see "Converting a stringname to a name" on page 438.

Resolving names

Clients call `resolve()` on the initial naming context to obtain the object associated with the supplied name:

Example 48: *Calling resolve()*

```
org.omg.CORBA.Object obj;
...
obj = root_cxt.resolve(name);
```

Alternatively, the client can call `resolve_str()` on the initial naming context to resolve the same name using its `StringName` equivalent:

Example 49: *Calling resolve_str()*

```
org.omg.CORBA.Object obj;
...
obj = root_cxt.resolve_str("Checking/Basic");
```

In both cases, the object returned in `obj` is an application object that implements the IDL interface `BasicChecking`, so the client narrows the returned object accordingly:

```
BasicChecking checking;
...
try {
    checking = BasicCheckingHelper.narrow(obj);
    // perform some operation on basic checking object
    ...
} // end of try clause, catch clauses not shown
```

Resolving names with corbaname

You can resolve names with a `corbaname` URL, which is similar to a `corbaloc` URL (see "Using corbaloc URL strings" on page 254). However, a `corbaname` URL also contains a stringified name that identifies a binding in a naming context. For example, the following code uses a corbaname URL to obtain a reference to a BasicChecking object:

Example 50: *Resolving a name with corbaname*

```
org.omg.CORBA.Object obj;
obj = orb.string_to_object(
     "corbaname:rir:/NameService#Checking/Basic"
);
```

A corbaname URL has the following syntax:

`corbaname:rir:[/NameService]#string-name`

`string-name` is a string that conforms to the format allowed by a `CosNaming::CosNamingExt::StringName` (see "Representing Names as Strings" on page 437). A corbaname can omit the `NameService` specifier. For example, the following call to `string_to_object()` is equivalent to the call shown earlier:

```
obj = orb.string_to_object("corbaname:rir:#Checking/Basic");
```

Exceptions Returned to Clients

Invocations on the naming service can result in the following exceptions:

NotFound The specified name does not resolve to an existing binding. This exception contains two data members:

`why` Explains why a lookup failed with one of the following values:

- `missing_node`: one of the name components specifies a non-existent binding.
- `not_context`: one of the intermediate name components specifies a binding to an application object instead of a naming context.
- `not_object`: one of the name components points to a non-existent object.

`rest_of_name` Contains the trailing part of the name that could not be resolved.

InvalidName The specified name is empty or contains invalid characters.

CannotProceed The operation fails for reasons not described by other exceptions. For example, the naming service's internal repository might be in an inconsistent state.

AlreadyBound Attempts to create a binding in a context throw this exception if the context already contains a binding of the same name.

Not Empty Attempts to delete a context that contains bindings throw this exception. Contexts must be empty before you delete them.

Listing Naming Context Bindings

In order to find an object reference, a client might need to iterate over the bindings in one or more naming contexts. You can invoke the `list()` operation on a naming context to obtain a list of its name bindings. This operation has the following signature:

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

`list()` returns with a `BindingList`, which is a sequence of `Binding` structures:

```
enum BindingType{ nobject, ncontext };

struct Binding{
    Name binding_name
    BindingType binding_type;
}
typedef sequence<Binding> BindingList
```

Iterating over binding list elements

Given a binding list, the client can iterate over its elements to obtain their binding name and type. Given a `Binding` element's name, the client application can call `resolve()` to obtain an object reference; it can use the binding type information to determine whether the object is a naming context or an application object.

For example, given the naming graph in Figure 23, a client application can invoke `list()` on the initial naming context and return a binding list with three `Binding` elements:

Figure 0.4:

Index	Name	BindingType
0	Checking	ncontext
1	Savings	ncontext

Figure 0.4:

Index	Name	BindingType
2	Loan	ncontext

Using a Binding Iterator

Limiting number of bindings returned by list()

In the previous example, `list()` returns a small binding list. However, an enterprise application is likely to require naming contexts with a large number of bindings. `list()` therefore provides two parameters that let a client obtain all bindings from a naming context without overrunning available memory:

how_many sets the maximum number of elements to return in the binding list. If the number of bindings in a naming context is greater than `how_many`, `list()` returns with its `BindingIterator` parameter set.

it is a `BindingIterator` object that can be used to retrieve the remaining bindings in a naming context. If `list()` returns with all bindings in its `BindingList`, this parameter is set to nil.

A `BindingIterator` object has the following IDL interface definition:

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

Obtaining remainder of bindings

If `list()` returns with a `BindingIterator` object, the client can invoke on it either `next_n()` to retrieve the next specified number of remaining bindings, or `next_one()` to retrieve one remaining binding at a time. Both functions return true if the naming context contains more bindings to fetch. Together, these `BindingIterator` operations and `list()` let a client safely obtain all bindings in a context.

Note: The client is responsible for destroying an iterator. It also must be able to handle exceptions that might return when it calls an iterator operation, inasmuch as the naming service can destroy an iterator at any time before the client retrieves all naming context bindings.

The following client code gets a binding list from a naming context and prints each element's binding name and type:

Example 51: *Obtaining a binding list*

```
// printing function
void
print_binding_list(org.omg.CosNaming.BindingListHolder bl)
{
    // extract the list of bindings
    org.omg.CosNaming.Binding[] list = bl.value;
    // iterate through list
    for( int i = 0; i < list.length; i++ ){
        System.out.print( list[i].binding_name[0].id;
        if( list[i].binding_name[0].kind != null )
            System.out.print(
                "(" + bl[i].binding_name[0].kind + ")");
        if( bl[i].binding_type ==
            org.omg.CosNaming.BindingType.ncontext )
            System.out.println( ": naming context" );
        else
            System.out.println( ": object reference" );
    }
}

void
get_context_bindings(omg.org.CosNaming.NamingContext cxt)
{
    org.omg.CosNaming.BindingListHolder b_list;
    org.omg.CosNaming.BindingIteratorHolder b_iter =
        new org.omg.CosNaming.BindingIteratorHolder();
    long MAX_BINDINGS = 50;

    // set up array to store binding list, put it in holder
    org.omg.CosNaming.Binding[] binding_list =
        new org.omg.CosNaming.Binding[MAX_BINDINGS];
    b_list =
        new org.omg.CosNaming.BindingListHolder(binding_list);

    // get first set of bindings from cxt
    cxt.list(MAX_BINDINGS, b_list, b_iter);
```

Example 51: *Obtaining a binding list*

```
    //print first set of bindings
    print_binding_list(b_list);

    // look for remaining bindings
    if( b_iter.value != null ) {
        org.omg.CosNaming.BindingIterator it = b_iter.value;
        do {
            boolean more = it.next_n(MAX_BINDINGS, b_list);
            // print next set of bindings
            print_binding_list(b_list);
        } while (more);

    // get rid of iterator
    it.destroy();
    }
}
```

When you run this code on the initial naming context shown earlier, it yields the following output:

```
Checking: naming context
Savings: naming context
Loan: naming context
```

Maintaining the Naming Service

Destruction of a context and its bindings is a two-step procedure:

- Remove bindings to the target context from its parent contexts by calling `unbind()` on them.
- Destroy the context by calling the `destroy()` operation on it. If the context contains bindings, these must be destroyed first; otherwise, `destroy()` returns with a `NotEmpty` exception.

These operations can be called in any order; but it is important to call both. If you remove the bindings to a context without destroying it, you leave an orphaned context within the naming graph that might be impossible to access and destroy later (see "Orphaned naming contexts" on page 444). If you destroy a context but do not remove its bindings to other contexts, you leave behind bindings that point nowhere, or *dangling bindings*.

For example, given the partial naming graph in Figure 27, you can destroy the Loans context and its bindings to the loan account objects as follows:

Example 52: *Destroying a naming context*

```
org.omg.CosNaming.NameComponent[] name;

// get initial naming context
org.omg.CosNaming.NamingContextExt root_cxt = ...;

// assume availability of Loans naming context variable
org.omg.CosNaming.NamingContext loans_cxt = ... ;

// remove bindings to Loans context
name = root_cxt.to_name("Loans/Mortgage");
root_cxt.unbind(name);
name = root_cxt.to_name("Loans/Auto");
root_cxt.unbind(name);
name = root_cxt.to_name("Loans/Personal");
root_cxt.unbind(name);

// remove binding from Loans context to initial naming context
name = root_cxt.to_name("Loans");
root_cxt.unbind(name);
```

Example 52: *Destroying a naming context*

```
// destroy orphaned Loans context
loans_cxt.destroy();
```

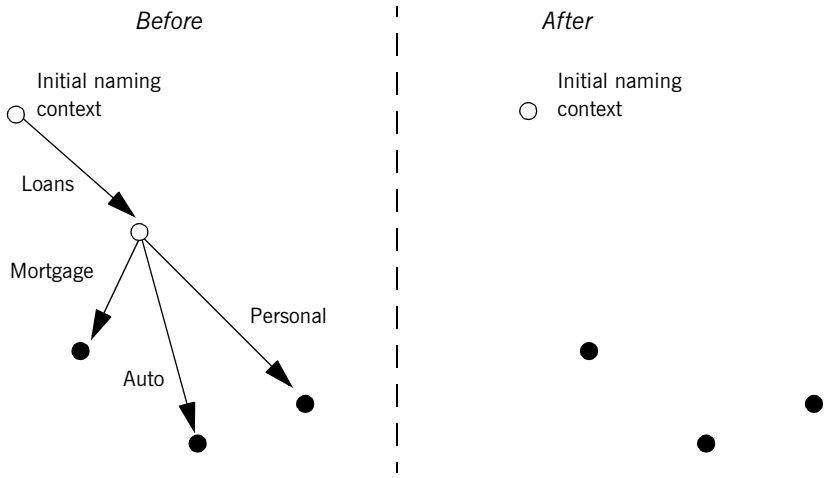

Figure 27: *Destroying a naming context and removing related bindings*

Note: Orbix provides administrative tools to destroy contexts and remove bindings. These are described in the *Application Server Platform Administrator's Guide*.

Federating Naming Graphs

A naming graph can span multiple naming services, which can themselves reside on different hosts. Given the initial naming context of an external naming service, a naming context can transparently bind itself to that naming service's naming graph. A naming graph that spans multiple naming services is said to be *federated*.

Benefits

A federated naming graph offers the following benefits:

- *Reliability*: By spanning a naming graph across multiple servers, you can minimize the impact of a single server's failure.
- *Load balancing*: You can distribute processing according to logical groups. Multiple servers can share the work load of resolving bindings for different clients.
- *Scalability*: Persistent storage for a naming graph is spread across multiple servers.
- *Decentralized administration*: Logical groups within a naming graph can be maintained separately through different administrative domains, while they are collectively visible to all clients across the network.

Federation models

Each naming graph in a federation must obtain the initial naming context of other members in order to bind itself to them. The binding possibilities are virtually infinite; however, two federation models are widely used:

- Hierarchal federation — All naming graphs are bound to a root server's naming graph. Clients access objects via the initial naming context of the root server.
- Fully-connected federation — Each naming graph directly binds itself to all other naming graphs. Typically, each naming graph binds the initial naming contexts of all other naming graphs into its own initial naming context. Clients can access all objects via the initial naming context of their local naming service.

Hierarchal federation

Figure 28 shows a hierarchal naming service federation that comprises three servers. The Deposits server maintains naming contexts for checking and savings accounts, while the Loans server maintains naming contexts for loan accounts. A single root server serves as the logical starting point for all naming contexts.

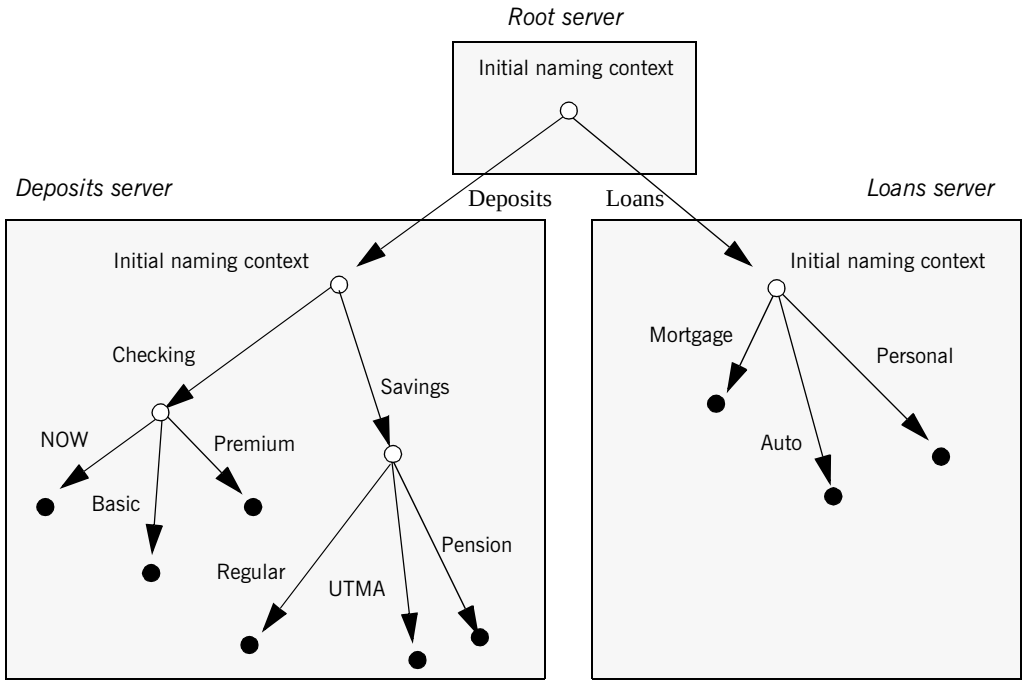

Figure 28: *A naming graph that spans multiple servers*

In this hierarchical structure, the naming graphs in the Deposits and Loans servers are federated through an intermediary root server. The initial naming contexts of the Deposits and Loans servers are bound to the root server's initial naming context. Thus, clients gain access to either naming graph through the root server's initial naming context.

The following code binds the initial naming contexts of the Deposits and Loans servers to the root server's initial naming context:

Example 53: *Federating naming graphs to a root server's initial naming context*

```
// Root server
...
public static void main (String[] args) {
    org.omg.CosNaming.NamingContextExt
        root_inc, deposits_inc, loans,_inc;
    org.omg.CosNaming.NameComponent[] name = new
   NameComponent[1];
    org.omg.CORBA.Object obj;
    org.omg.CORBA.ORB global_orb;
    String loans_inc_ior, deposits_inc_ior
...
    try {
        global_orb = org.omg.CORBA.global_orb.init(args, null);

        // code to obtain stringified IORs of initial naming
        // contexts for Loans and Deposits servers (not shown)
        ...

        obj = global_orb.string_to_object(loans_inc_ior);
        loans_inc =
           org.omg.CosNaming.NamingContextExtHelper.narrow(obj);
        obj = global_orb.string_to_object(deposits_inc_ior);
        deposits_inc =
           org.omg.CosNaming.NamingContextExtHelper.narrow(obj);

        // get initial naming context for Root server
        root_inc = ... ;

        // bind Deposits initial naming context to root server's
        // initial naming context
        name[0] = new NameComponent("Deposits", "");
        root_inc.bind_context(name, deposits_inc);

        // bind Loans initial naming context to root server's
        // initial naming context
        name[0] = new NameComponent("Loans", "");
        root_inc.bind_context(name, deposits_inc);
    }
}
```

This yields the following bindings between the three naming graphs:

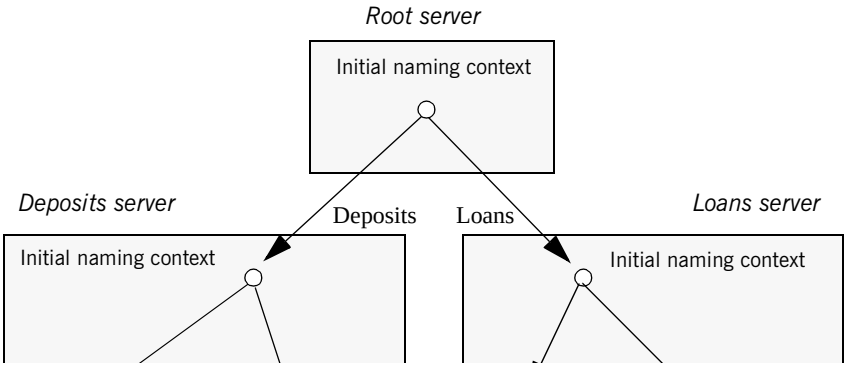

Figure 29: *Multiple naming graphs are linked by binding initial naming contexts of several servers to a root server.*

Fully-connected federation

In a purely hierarchical model like the naming graph just shown, clients obtain their initial naming context from the root server, and the root server acts as the sole gateway into all federated naming services. To avoid bottlenecks, it is possible to modify this model so that clients can gain access to a federated naming graph via the initial naming context of any member naming service.

The next code example shows how the Deposits and Loans servers can bind the root server's initial naming context into their respective initial naming contexts. Clients can use this binding to locate the root server's initial naming context, and then use root-relative names to locate objects.

Figure 30 shows how this federates the three naming graphs:

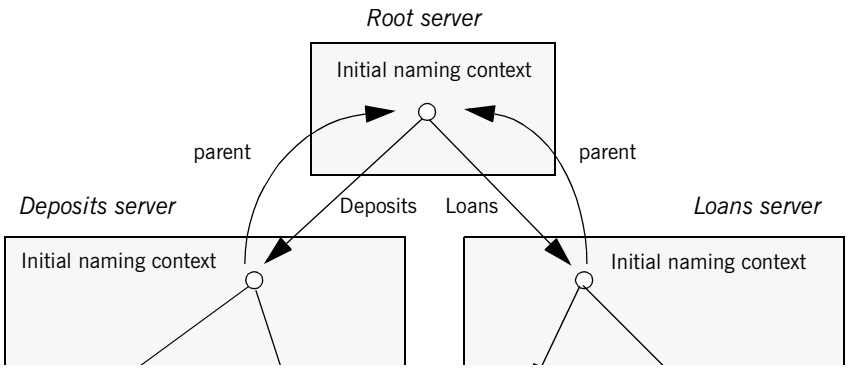

Figure 30: *The root server's initial naming context is bound to the initial naming contexts of other servers, allowing clients to locate the root naming context.*

The code for both Deposits and Loans server processes is virtually identical:

Example 54: *Federating naming graphs through the initial naming contexts of multiple servers*

```
public static void main (String[] args) {
    org.omg.CosNaming.NamingContextExt root_inc, this_inc;
    org.omg.CosNaming.NameComponent[] name =
        new NameComponent[1];
    org.omg.CORBA.Object obj;
    org.omg.CORBA.ORB global_orb;
    String root_inc_ior;
    ...
    try {
        global_orb = org.omg.CORBA.global_orb.init(args, null);

        // code to obtain stringified IORs of root server's
        // initial naming context (not shown)
        ...

        obj = global_orb.string_to_object(root_inc_ior);
        root_inc =
           org.omg.CosNaming.NamingContextExtHelper.narrow(obj);
```

Example 54: *Federating naming graphs through the initial naming contexts of multiple servers*

```
        // get initial naming context for this server
        this_inc = ... ;

        name[0] = new NameComponent("parent", "");

        // bind root server's initial naming context to
        // this server's initial naming context
        this_inc.bind_context(name, root_inc);
        ...
    }
```

Sample Code

The following sections show the server and client code that is discussed in previous sections of this chapter.

Server code

Example 55: *Server naming service code*

```
public static void main (String[] args) {
    org.omg.CosNaming.NamingContextExt root_cxt;
    org.omg.CosNaming.NamingContext
        checking_cxt, savings_cxt, loan_cxt;
    org.omg.CosNaming.NameComponent[] name;
    org.omg.CORBA.ORB orb;
    org.omg.CORBA.Object obj;
    Checking basic_check, now_check, premium_check;

    // Checking objects initialized from persistent data
    // (not shown)

    try {
        // Initialize the ORB
        orb = org.omg.CORBA.global_orb.init(args, null);

        // Get reference to initial naming context
        obj =
           global_orb.resolve_initial_references("NameService");
        root_cxt =
           org.omg.CosNaming.NamingContextExtHelper.narrow(obj);
        if( root_cxt != null ) {
            // build naming graph

            // initialize name
            name = root_cxt.to_name("Checking");
            // bind new naming context to root
            checking_cxt = root_cxt.bind_new_context(name);
```

Example 55: *Server naming service code*

```
            // bind checking objects to Checking context
            name = root_cxt.to_name("Checking/Basic");
            checking_cxt.bind(name, basic_check);
            name = root_cxt.to_name("Checking/Premium");
            checking_cxt.bind(name, premium_check);
            name = root_cxt.to_name("Checking/NOW");
            checking_cxt.bind(name, now_check);

            name = root_cxt.to_name("Savings");
            savings_cxt = root_cxt.bind_new_context(name);

            // bind savings objects to savings context
            ...

            name = root_cxt.to_name("Loan");
            loan_cxt = root_cxt.bind_new_context(name);

            // bind loan objects to loan context
            ...
        }
        else {...} // deal with failure to narrow()
        ...
    } // end of try clause, catch clauses not shown
    ...
}
```

Client code

Example 56: *Client naming service code*

```
public static void main (String[] args) {
    org.omg.CosNaming.NamingContextExt root_cxt;
    org.omg.CosNaming.NameComponent[] name;
    BasicChecking_var checking;
    org.omg.CORBA.Object obj;
    org.omg.CORBA.ORB global_orb;
...

    try {
        global_orb = org.omg.CORBA.global_orb.init (args, null);
```

Example 56: *Client naming service code*

```
        // Find the initial naming context
        obj =
            global_orb.resolve_initial_references("NameService");
        root_cxt =
           org.omg.CosNaming.NamingContextExtHelper.narrow(obj);
        if( root_cxt != null ) {
            obj = root_cxt.resolve_str("Checking/Basic");
            checking_var == BasicCheckingHelper.narrow(obj);
            if( checking_var != null ) {
               // perform some operation on basic checking object
                ...
            }
            else { ... } // Deal with failure to narrow()
        } else { ... } // Deal with failure to resolve object

    } // end of try clause, catch clauses not shown
    ...
}
```