Information System Infrastructure II

Naming

Hans-Arno Jacobsen
jacobsen@eecg.toronto.edu
www.eecg.toronto.edu/~jacobsen

Naming in distributed systems

- how does the client find the server in a distributed system?
- where does naming information intervene:
  - URLs
  - file system
  - NIS - Network Information System
  - NFS - Network File System
- how is a name www.ebroker.cad resolved in the network, i.e., how is the server that serves pages for ebroker found?
- can a service be discovered by looking for its properties?
- difference between domain name and address?
- what abstractions and services are available to do this?
  - our examples: DNS, JNDI, LDAP

(c) 2002 Hans-Arno Jacobsen (jacobsen@eecg.toronto.edu) - MIE491f
Outline

• Naming
• Case studies
  - Common Object Service Naming (i.e., CORBA)
  - LDAP - Lightweight Directory Service
  - JNDI - Java Naming and Directory Service
  - DNS - Domain Name system
• Reading
  - JNDI white paper, rather abstract
  - DNS research paper

Naming and directory service

• naming service maps names to addresses and vice versa
• directory service offers a mostly read-based data repository
• often combined in one service
• local directory service, e.g., finger information
• global directory service, i.e., global across the whole network
  - often distributed with data it manages distributed all across the network
  - often defines uniform namespace, i.e., provides the same view of the data no matter where one is at
Naming

- A name is a sequence of identifiers that are **bound** to an (object) reference/address and can be resolved into the reference/address.
- This allows human **legible** and efficient access to appropriate resources in distributed environments.
- Names can be organized in a hierarchical fashion to avoid naming conflicts (name clashes) and organize information.

Example:

```
UNIX File System
  root
    bin
    etc
    lib
      include

Internet domain names
    edu
      utoronto
      ubc
      sun
      ibm

com
org
```

In the tree structure, each intermediate node identifies a group of names (closely) related to each other. Those names form a **name space**, which can be also referred to as a naming context.

---

**Naming System:**

A general naming model.

```
name_1
name_2
... name_k

name space

Name Mapping Algorithm

Context

Universe of Objects
```

(c) 2002 Hans-Arne Jacobsen (jacobsen@eceg.toronto.edu) ME491f
CORBA Naming Service

The following slides up to "CORBA Naming Service", have been adapted from slides by Ian Gorton, Paul Greenfield Software Architectures and Component Technologies, CSIRO, Sydney.

CORBA Services

- An ORB defines the basic (remote) communications mechanism for distributed object systems
  - i.e., the plumbing, the pipes
- CORBA Services define a number of 'value-add' services that make building distributed systems easier
  - the taps, filters, pumps, heaters, etc
CORBA Services

- Naming service
- Transaction service
- Security service
- Event service
- Trading service
- Notification service
- and many more ...
- all typically quite complex components

CORBA Naming Service

- Copying stringified references from a server to all its clients is clumsy and does not scale.
- The Naming Service provides a way for servers to advertise references under a name, and for clients to retrieve them.
- The advantages are:
  - Clients and servers can use meaningful names instead of having to deal with stringified references.
  - By changing a reference in the service without changing its name, you can transparently direct clients to a different object.
  - May be used for load balancing.
CORBA Naming Service

- The Naming Service solves the **bootstrapping problem** because it provides a fixed point for clients and servers to rendezvous.
- *But how do clients find the naming service?*
- The Naming Service is much like a white pages phone book. Given a name, it returns an object reference.
- *Does it make sense to offer a yellow pages service for distributed applications? Do you know one on the Internet?*

Terminology

- **A name-to-IOR association is called a name binding**
- Each binding identifies exactly one object reference, but an object reference may be bound more than once (have more than one names associated).
- **A naming context** is an object that contains name bindings. The names within a context must be unique.
Terminology

- Naming contexts can contain bindings to other naming contexts, so naming contexts can form graphs.
- **Binding** a name to a context means to add a name-IOR pair to a context.
- **Resolving** a name means to look for a name in a context and to obtain the IOR bound under that name.

A naming service provides a graph of contexts that contain bindings to other contexts or objects.
Naming IDL

• The IDL for the Naming Service has the following overall structure:

```idl
// File: CosNaming.idl
module CosNaming {
    // Type definitions here...
    interface NamingContext {
        // ...
    };
    interface NamingContextExt : NamingContext {
        // ...
    };
    interface BindingIterator {
        // ...
    };
}
```

Name Representation

• A name component is a pair of strings.
• A sequence of name components forms a pathname through a naming graph:

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

• The kind field is meant to be used similarly to a file name extension (such as "filename.cpp").
Names

- Names are sequences of string pairs. We can show a name as a table, see below
- The same name can be written as a string as:
  - Ireland.Country/Guinness.Brewery

<table>
<thead>
<tr>
<th>Index</th>
<th>id</th>
<th>kind</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Ireland</td>
<td>Country</td>
</tr>
<tr>
<td>1</td>
<td>Guinness</td>
<td>Brewery</td>
</tr>
</tbody>
</table>

Obtaining an Initial Naming Context

- You must obtain an initial naming context before you can do anything with the service.
- The configured initial naming context is returned by
  `resolve_initial_references("NameService")`
- This returns an object reference to either a NamingContext or a NamingContextExt object. (For ORBacus, you always get a NamingContextExt interface.)
Example

// Initialize the ORB.
CORBA::ORB_var orb = CORBA::ORB_init(argc, argv);

// Get initial naming context.
CORBA::Object_var obj
    = orb->resolve_initial_references("NameService");

// Narrow to NamingContext
CosNaming::NamingContext_var inc; // Initial naming context
inc = CosNaming::NamingContext::_narrow(obj);
// ...

Exceptions

• The NamingContext interface defines a number of exceptions:

  interface NamingContext {
      enum NotFoundReason { missing_node, not_context, not_object }; 
      exception NotFound {
          NotFoundReason why;
          Name rest_of_name;
      };
      exception CannotProceed {
          NamingContext ctx;
          Name rest_of_name;
      };
      exception InvalidName {};
      exception AlreadyBound {};
      exception NotEmpty {};
      // ...
  };

Creating and Destroying Contexts

- **NamingContext** contains three operations to control the life cycle of contexts:

```java
interface NamingContext {
    // ...
    NamingContext new_context();
    NamingContext bind_new_context(in Name n) raises(
        NotFound, CannotProceed, InvalidName, AlreadyBound
    );
    void destroy() raises(NotEmpty);
    // ...
};
```

Creating Bindings

- Two operations create bindings to application objects and to contexts:

```java
interface NamingContext {
    // ...
    void bind(in Name n, in Object obj)
    raises(NotFound, CannotProceed, InvalidName, AlreadyBound );

    void bind_context(in Name n, in NamingContext nc)
    raises(NotFound, CannotProceed, InvalidName, AlreadyBound);  // ...
};
```
**Creation Example**

- To create a naming graph, you can use names that are all relative to the initial context or you can use names that are relative to each newly-created context.
- The code examples that follow create part of the following graph:

```
CosNaming::NamingContext_var inc = ...; // Get initial context
CosNaming::Name name;
name.length(1);
name[0].id = CORBA::string_dup("app2"); // kind is empty
CosNaming::NamingContext_var app2;
app2 = inc->bind_new_context(name); // inc -> app2
name.length(2);
name[1].id = CORBA::string_dup("collections");
CosNaming::NamingContext_var collections;
collections = inc->bind_new_context(name); // app2 -> collections
name[1].id = CORBA::string_dup("devices");
CosNaming::NamingContext_var devices;
devices = inc->bind_new_context(name); // app2 -> devices
name.length(3);
name[2].id = CORBA::string_dup("cd");
CosNaming::NamingContext_var cd;

```

**Code Example**

```
cosNaming::NamingContext_var inc = ...; // Get initial context
CosNaming::Name name;
nname.length(1);
nname[0].id = CORBA::string_dup("app2"); // kind is empty
CosNaming::NamingContext_var app2;
app2 = inc->bind_new_context(name); // inc -> app2
name.length(2);
nname[1].id = CORBA::string_dup("collections");
CosNaming::NamingContext_var collections;
collections = inc->bind_new_context(name); // app2 -> collections
name[1].id = CORBA::string_dup("devices");
CosNaming::NamingContext_var devices;
devices = inc->bind_new_context(name); // app2 -> devices
name.length(3);
nname[2].id = CORBA::string_dup("cd");
CosNaming::NamingContext_var cd;
cd = inc->bind_new_context(name); // devices -> cd
name.length(4);
nname[3].id = CORBA::string_dup("app2");
```
Example (cont)

inc-> bind_context(name, app2); // cd -> app2
CCS:: Controller_var ctrl = ...;
name. length(3);
name[ 2].id = CORBA:: string_dup("dev1");
inc-> bind( name, ctrl); // devices ->dev1
name[ 1].id = CORBA::string_dup("collections");
name[ 2].id = CORBA::string_dup("cd");
inc-> bind_context( name, cd); // collections -> cd

Rebinding

- The `rebind` and `rebind_context` operations replace an existing binding:

```c
interface NamingContext {
    // ...
    void rebind(in Name n, in Object obj)
    raises(NotFound, CannotProceed, InvalidName );

    void rebind_context(in Name n, in NamingContext nc)
    raises( NotFound, CannotProceed, InvalidName );
    // ...
};
```
Example

CORBA:: Object_ var obj = ...; // Get an object
CosNaming:: NamingContext_ var cxt = ...; // Get a context...

CosNaming:: Name name;
name. length( 1);
name[ 0]. id = CORBA:: string_ dup(" Some name");

cxt-> rebind( name, obj); // Fine

cxt-> rebind( name, obj); // Fine

Resolving Bindings

• The resolve operation returns the reference stored in a binding:

interface NamingContext {
    // ... 
    Object resolve(in Name n) raises(NotFound, CannotProceed, 
    InvalidName );
    // ...
};

• The returned reference is (necessarily) of type Object, so you must narrow it to the correct type before you can invoke operations on the reference.
Example

```cpp
CosNaming::NamingContext inc = ...; // Get initial context...
CosNaming::Name name;
name. length( 3);
name[ 0]. id = CORBA::string_dup(" app2");
name[ 1]. id = CORBA::string_dup(" devices");
name[ 2]. id = CORBA::string_dup(" dev1");
CORBA::Object_ var obj;
try {
    obj = inc->resolve( name);
    catch (const CosNaming::NamingContext::NotFound &) {
        // No such name, handle error...
        abort();
    }
    catch (const CORBA::Exception & e) {
        // Something else went wrong...
        cerr << e << endl;
        abort();
    }
} catch (const CORBA::Exception & e) {
    cerr << e << endl;
    abort();
}
```

Example (cont)

```cpp
if (CORBA::is_nil(obj)) {
    // Polite applications don't advertise nil references!
    cerr << "Nil reference for controller!" << endl;
    abort();
} try {
    MyObj_ var ctrl = MyObj::_narrow(obj);
} catch (CORBA::SystemException & e) {
    // Can't figure it out right now...
    cerr << "Can't narrow reference: " << e << endl;
    abort();
} if (CORBA::is_nil(ctrl)) {
    // Oops!
    cerr << "Someone advertised wrong type of object!" << endl;
    abort();
}
```
Orbacus Naming Service

- ORBacus Names is provided as the nameserv executable. Common options (use nameserv -h for a list):
  - -I : Print initial naming context IOR on stdout
  - -d database_file : Specifies database file for the service. (Without -d, the service is 
    not persistent and uses an in-memory database.).

nsadmin

- nsadmin provides a way to manipulate the Naming Service from the command line. Common options:
  - -b name IOR
    - Bind IOR under the name name.
  - -c name
    - Create and bind a new context under the name name.
  - -l [name]
    - List the contents of the context name. (Initial context, by default.)
  - -r name
    - Print the IOR for the binding identified by name.
Compiling and linking

- The header files for the service are in
  - include/ OB/ CosNaming. h
  - include/ OB/ CosNaming_skel. h
- The stubs and skeletons for ORBacus Names are pre-compiled and installed in
  - lib/ libCosNaming. sl
- To compile a client or server that uses ORBacus Names, compile with
  - -I /opt/ OB4/ include
  - -L /opt/ OB4/ lib -lCosNaming

Case study: LDAP
Lightweight Directory Access Protocol

- "Lightweight" means it is a "simplified" version of the X.500 directory standard.
- stores data in attribute-based manner
- design for data reading more than writing
- therefore no transaction support
- therefor no rollback support
- data storage maybe replicated, with potential inconsistent replicas that will eventually become consistent
- entries are organized in a tree structure called Directory Information Tree.

Attribute Abbreviations

uid: user id
sn: surname
dc: domain component
c: country
cu: organizational unit
cn: common name
st: state
dn: distinguished name
Searching in LDAP

- LDAP search filters are written in prefix (Polish) notation.
- use a set of operations:
  - & : and
  - >= : greater than or equal
  - | : or
  - <= : less than or equal to
  - ~ : approximately equal to

Examples:
- Where is Smith?
  \((\&(dc == ut)(ou = people))(uid = smith))\)
- or simply
  \((\&(dc = ece)(uid = Smith))\)

assuming there is only one Smith in ece

A naming service provides (at least) four functions

- Resolve (name, context). Given a naming context, this operation returns the value associated with the name.
- Bind (name, value, context). Associate a value with a name in the specified context.
- Unbind (name, value, context). This operation removes the association between the name and the value.
- List Name (context). This operation lists all the names in the naming system within the naming context.
Directory Service

- Directory Service is a special type of naming service. Each entity in the tree structure must have certain attributes to describe the entry:
- For example: the ECF network can be conceptually stored in the following directory:

```
ECF Network (subnet = “…” , subnetmask = “…”

Routers (number = “…”

workstations (number = “…”

NT Stations
cisco router 1 (address = “…”

sparc stations (number = “…”
```

Naming Resolution (three common ways):

1. **Simple Task Lookup:** The naming context is implemented using a table of (name, value) pairs. Used to
   1. register names
   2. memory address
   3. communication ports of computer systems

2. **Path Name:** A path name provides, in addition to the name, the reference of the context. A syntax needs to be defined to separate the components in the path.
   Examples of using path names are:
   1. ece.toronto.edu.ca
   2. /sports/hockey/maple leaves

3. **Search:** Collaboration, among table lookups and path names, is involving multiple naming service entities.
   Examples:
   1. Command search. System needs to review all system paths.
   2. Library search for symbols during program compilation.
**LDAP Applications:**

- LDAP is a specification. It is implemented in LDAP servers. For example, OpenLDAP is an open source LDAP implementation.

- The following diagram shows how applications use LDAP servers.

**Case study: JNDI**
Java Naming and Directory Interface (JNDI)

• A general programming interface that provides naming and directory functionality.

• Independent of every specific naming or directory service implementation. Applications can use JNDI to use services such as LDAP, DNS etc..

• Applications are able to store objects in naming services and retrieve objects by using JNDI.

The following is a code segment for storing and retrieving the room object using JNDI and LDAP directory service.

Our room object is defined using the following class diagram.

```
Room
- roomid : String
- size: int

+ Room (roomid: String, size: int)
+ getRoomId() : String
+ getSize(): int

[serializable]
```

(c) 2002 Hans- Arno Jacobsen (jacobson@ee.c.utoronto.ca) MIE491f
Our code starts with telling JNDI that we are going to use the LDAP server provided by Sun and our naming context starts with an entry called JNDI Tutorial.

```
Hashtable env = new Hashtable();
env.put (Context.INITIAL_CONTEXT_FACTORY,
        com.sun.jndi.ldap.ldapHxFactory)
env.put (context.PROVIDER_DRL,
        LDAP=//localhost:38910=JNDI Tutorial)
Context ctx = new InitialContext (env);

Let's create our room:
    Room mc406 = new Room("mc406", 30);
To store the room in the directory:
    ctx.bind ("cn = mc406", mc406);
To read the room back later:
    Room r = (Room) ctx.lookup("cn = mc406");
To remove the room from the directory
    ctx.unbind("cn = mc406");
```

Case study: DNS - Domain Name System