CORBA

Concurrent & Distributed Software

CORBA IDL

- Need to understand IDL-to-Java mapping or IDL-to-C++ mapping
  - usually a chapter in ORB programmer's manual
  - Chapter 20 of Orfali & Harkey
  - For C++, see Henning & Vinoski
- similar to C++ class declarations
- no code (implementation)
- Java issues – holder classes used for output parameters
- C++ issues - _var classes (smart pointers)

IDL

- Some features
  - oneway operations (must have void return type)
  - interfaces may be derived from other interfaces
    - multiple inheritance allowed
    - no state or code inherited since there is none in IDL
    - derived interfaces cannot redefine attributes or operations (although types, constants, exceptions can be redefined)
  - constructed types
    - struct, enum, union, sequence, array
    - sequences are variable length
    - arrays can be multidimensional

IDL cont'd

- Object references

```
interface account;
interface bank {
  account newAccount(in string name);
  void deleteAccount(in account a);
}
```

newAccount returns a reference to an account object,
deleteAccount takes an object reference as a parameter

IDL cont'd

- Attributes
  - default read/write; mapped to two functions
  - readonly attributes mapped to a single function
- Exceptions
  - user defined exceptions can contain any data field desired
  - any number of user exceptions can be listed for an operation
    - all operations, and attributes, can raise **system** exceptions

IDL -- user exceptions

```
Interface bank {
   exception reject {
     string reason;    // programmer chosen fields
};

account newAccount(in string name)
                    raises (reject);
```

Built in IDL types

- Object — root of all IDL interfaces
- NamedValue — a pair (string,value)
- TypeCode — representation of a type
- Principal — caller of an operation

All these are useful in DII/DSI world

Creating multiple copies of objects

- In distributed object systems, objects are always created by the server
  - a server process can be thought of as a "container" for objects
  - must distinguish between *CORBA objects* and other objects
- To create multiple objects (instantiations) of a class, use a *ClassFactory*

Example

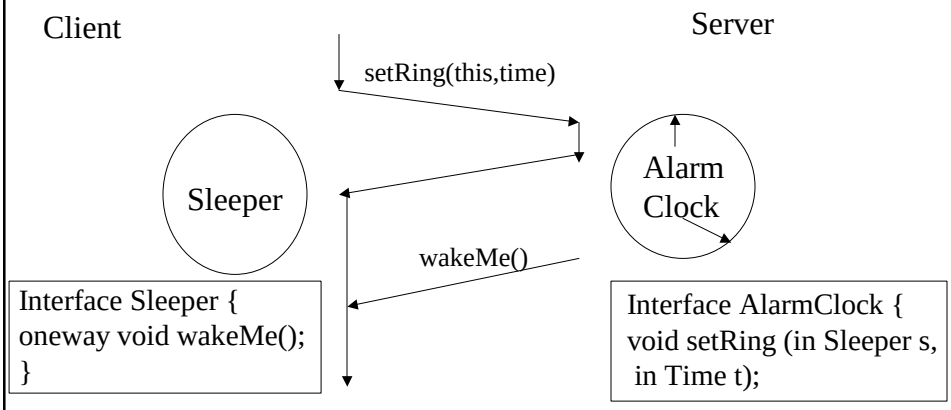
```
module Bank {
  interface Account {
    float balance();
  };
  interface AccountManager {
    Account open(in string name);
  };
};
```

Object stringification

- Can convert object references to strings and vice versa
  - useful for saving object references to a file
  - can be passed between processes
- ORB.object_to_string returns a stringified Internet (or Interoperable) Object Reference (IOR)
- ORB.object_to_string does reverse

Callbacks

- Useful for servers to call objects in clients
  - client object reference does not have to be registered



Approaches for object implementations

- Inheritance: ImplBase approach
  - implementation class that you write extends _<interface_name>ImplBase
  - uses up Java single inheritance
- Delegation: the Tie approach
  - _tie<interface_name> class inherits from ImplBase class ; delegator class that delegates every call to the real implementation class that you write

Delegation based approach

- The implementation class that you write should *implement* the Interface
  - can also *extend* a different class
  - useful for multiple inheritance

Example

```
module HelloApp
{
    interface Hello
    {
        string sayHello();
    };
};
```

`idltojava -ftie  Hello.idl`

This generates two additional files in a `HelloApp subdirectory`:

`_HelloOperations.java`

The servant class will implement this interface.

`_HelloTie.java`

This class acts as the skeleton, receiving invocations from the ORB and delegating them to the servant that actually does the work.

Example cont'd

```
class HelloBasic {
public String sayHello() {
   return "\nHello world !!\n";
   }
}

class HelloServant extends HelloBasic implements
                _HelloOperations
{
}
```

Example cont'd

```
public class HelloServer {
 public static void main(String args[])
{
 try{
 // create and initialize the ORB
ORB orb = ORB.init(args, null); //
 create servant and register it with the ORB
HelloServant servant = new HelloServant();
 Hello helloRef = new _HelloTie(servant);
orb.connect(helloRef);
```

```
org.omg.CORBA.Object objRef =
                orb.resolve_initial_references("NameService");
NamingContext ncRef = NamingContextHelper.narrow(objRef);
// bind the Object Reference in Naming
NameComponent nc = new NameComponent("Hello", "");
NameComponent path[] = {nc}; ncRef.rebind(path, helloRef);
// wait for invocations from clients
java.lang.Object sync = new java.lang.Object();
synchronized (sync) {
   sync.wait();
    }
 }
 catch (Exception e) {
    System.err.println("ERROR: " + e);
    e.printStackTrace(System.out);
   }
  }
}
```

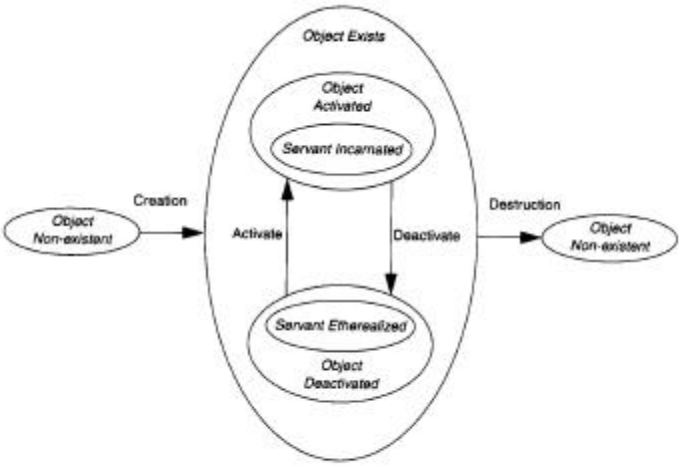
DII/DSI

- Useful for constructing requests (DII) or serving requests (DSI) at run-time
  - no pre-compiled stubs
  - more expensive
  - useful for agents, bridges (inter-operability)
- DII -- query the interface repository for information on operation to be invoked and construct request
- DSI -- servant class inherits from DynamicImplementation class and implements invoke operation that “deconstructs” the request

Portable Object Adaptor (POA)

- "BOA" done right
- deals with activation of objects and servers
- supports both IDL-generated skeletons and DSI

Life-span of a CORBA object



POA concepts

- Objects can be either *transient* or *persistent*
  - persistent objects outlive the processes (servers) they "live in" ; a persistent object spans multiple server lifetimes
  - terminology: *servant* = object implementation
- servant managers
  - An application can register servants directly with the POA OR it can supply servant manager objects to the POA that can create servants to carry out a request
  - you can supply your own or use the default servant manages supplied by the ORB

Servant Managers

- Objects that assist the POA in the management of your server-side objects
- POA invokes operations on servant managers to *create, activate, and deactivate servants*
  - note that there is a *clear distinction* between *creation* and *activation*
  - client only sees an object reference
  - servant managers must be registered with POA

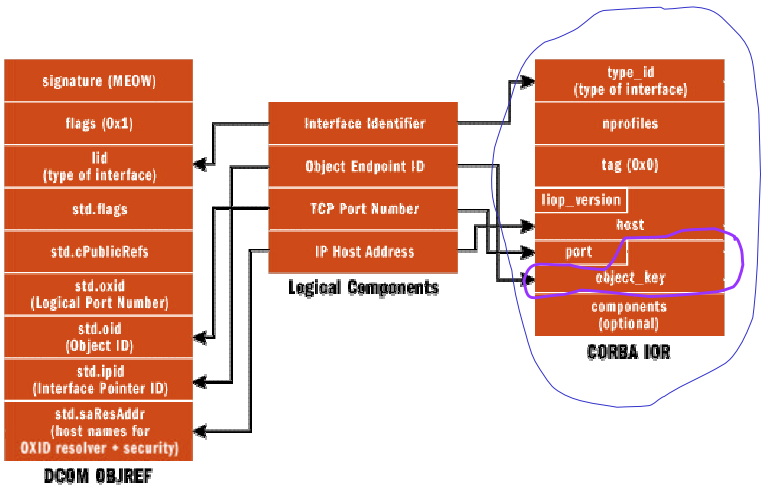
POAs

- A single server can support multiple POAs derived from the root POA (create_POA)
- Each POA can be customized (create_POA_policy)
- Each POA maintains a list of active servant managers
- Each POA also maintains a map of active objects (Object_ID to servant map)

Persistent Objects & References

- CORBA object references are unique
  - encapsulate both the POA and an *Object ID*
  - *Object ID* is a value used by the POA and your implementation to identify a particular object
    - no standard form, can be implementation specific (e.g., key of a DBMS record)
- Implementing persistent objects
  - providing the code for storing and restoring object state
  - maintaining the mapping between object references and object state

Corba IOR



Servant Managers

- Applications that activate all their objects at server start up time do not need servant managers
- Servant managers let POAs activate objects on demand
- Servant Managers are responsible for determining if an object exists, and managing the association between object ids and servants

Servant Managers cont'd

- Implement one of two interfaces
  - ServantActivator
    - typically used with persistent objects
    - RETAIN policy
  - ServantLocator
    - NON-RETAIN policy
- Both types of Servant Managers contain two operations -- one to find and return a servant, and the second to deactivate a servant

POA policies

- Threading
  - threading model
    - ORB_CTRL_MODEL
    - SINGLE_THREAD_MODEL
- Lifespan
  - persistence model for objects in the POA
    - TRANSIENT
    - PERSISTENT

POA Policies cont'd

- Object Id uniqueness
  - specifies whether servants activated by this POA have unique object ids
    - UNIQUE_ID
    - MULTIPLE_ID (e.g. when a single servant incarnates multiple CORBA objects)
- ID Assignment
  - who generates Object Ids
    - USER_ID (typically for persistent objects)
    - SYSTEM_ID (typically for transient objects)

POA Policies cont'd

- Servant Retention
  - whether the POA will retain active servants in an Active Object Map
    - RETAIN
    - NON_RETAIN
- Activation
  - does POA support implicit activation of objects
    - IMPLICIT_ACTIVATION (typically for transient objects)
    - NO_IMPLICIT_ACTIVATION

POA Policies cont'd

- Request Processing
  - how requests are processed
    - USE_ACTIVE_OBJECT_MAP_ONLY
    - USE_DEFAULT_SERVANT
    - USE_SERVANT_MANAGER

Policy Combinations

- RETAIN & USE_ACTIVE_OBJECT_MAP_ONLY
  - objects explicitly activated by application on startup
  - good for servers that manage a finite number of pre-started objects (or well known services)
- RETAIN & USE_SERVANT_MANAGER
  - ideal for servers that manage a large number of persistent objects
  - if POA does not find a servant in its active map, it invokes servant managers `incarnate()` method

Policy Combinations

- RETAIN & USE_DEFAULT_SERVANT
  - ideal for servers that support a large number of transient objects
- NON_RETAIN & USE_SERVANT_MANAGER
  - ideal if one servant is invoked per method call
  - POA calls `preinvoke` on servant manager of type ServantLocator

Object Activation

- POA object reference creation and object activation are decoupled
  - create_reference() or create_reference_with_id()
    - only create reference, not an active servant
- Object activation
  - explicitly via activate_object()
  - on-demand using a user-supplied servant manager
  - implicitly using a default servant (if IMPLICIT_ACTIVATION policy in effect)

Finding the Target Object

- ORB requests contain both POA id and Object ID
- server started if not already running
- if POA does not exist, it has to be recreated using an adapter activator
- POA handles request according to Request Processing policy

IIOP

- Inter-orb protocol
- IIOP is TCP/IP implementaion of GIOP
- all ORBs have bridges
- IOR: stringified representation of object reference
  - ***it's all you need to invoke a method on a remote object***

Garbage Collection

- Automatic reclamation of resources used by objects that are no longer in use by clients
  - Objects = CORBA objects? Servants?
  - What about persistent objects?
- Techniques
  - Shutting down the server periodically
  - "Evictor" design pattern ***Recommended strategy***
  - Time outs
  - Explicit keep-alive
  - Reverse keep-alive
  - Distributed reference counts
- Distributed garbage collection still an open research problem

Implementation Repositories

- Used for "indirect binding" for **persistent** references
  - Direct binding requires servers to be running when clients wan to use them
- Deliberately not standardized
  - Clients interact with implementation repositories in a standardized way but proprietary mechanisms exist between servers and their implementation repositories
  - Provides a point at which ORB vendors can provide additional features such as object migration, load balancing, etc.
- Responsibilities
  - Maintains a registry of known servers
  - It records which server is currently running on which host and what port
  - It starts servers on demand if they are registered for automatic startup

CORBA services

- A set of services useful for building applications
  - Naming
  - Trading (find objects given a constraint string)
  - Event (send messages to multiple receivers)
  - Transactions
  - Security
  - Persistence
  - Time, Licensing, Lifecycle, Properties, Relationships, Concurrency, Query, Externalization