CORBA

- Distributed Objects
- CORBA: Introduction and Overview
- CORBA Clients
- CORBA Servers
- CORBA Development Issues

- JavaIDL: *see course website*

1

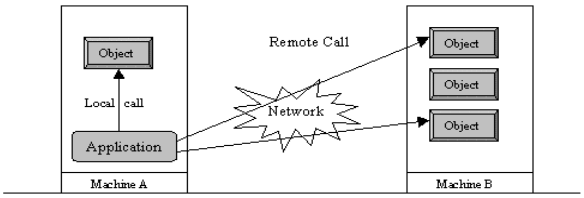
Distributed Objects

- Traditional enterprise apps are self-contained, monolithic apps
- Limited access to another's procedures and data
- Apply OO techniques for networked apps
- multi-tiered architecture (separation of concerns)
- 1st, 2nd, 3rd generation systems

2

Distributed Objects

- Local vs. Remote objects



- An enterprise app is a collection of co-operating objects located on different machines

3

Advantages

- Benefit from OO techniques
- Eliminates protocol design, which is error-prone
- Convenient resource sharing

4

Disadvantages

- Design and Impl'n is more complex
- Multiple failure modes
- Security issues
- Use of multiple technologies
- Testing and debugging

5

Available Technologies

- Many technologies available:
- Sockets (TCP, UDP) Not OO
- RPC (not OO)

- RMI
- CORBA

6

CORBA Introduction

- OMG
- Current state of OMG/CORBA efforts
- Object Model
- Client Server model and issues

7

OMG - Object Mgm't Group

- Non-profit consortium of software vendors
- Formed in 1989 by 8 companies
- Has more than 800 members now (?)
- http://www.omg.org
- Goals:
  - Develop specs to provide a common framework for distributed applications development
  - Promote a heterogeneous computing environment

8

CORBA

- Stands for: Common Object Request Broker Architecture
- A spec for creating distributed objects
- CORBA is NOT a programming language
- Its architecture is based on the object model
- Promotes design of applications as a set of cooperating objects
- OMG Object Model: object is defined as what the client could see.

9

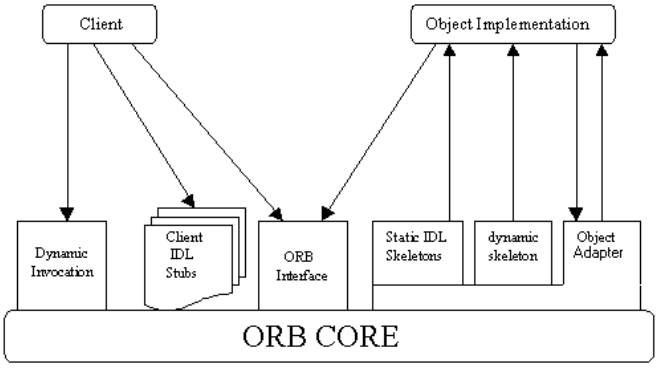
CORBA Objects

- clients are isolated from servers by interface
- CORBA objects vs. typical objects:
  - CORBA objects run on any platform
  - CORBA objects can be located anywhere on the network
  - CORBA objects can be written in any language that has IDL mapping

10

CORBA Architecture

- The structure of CORBA



11

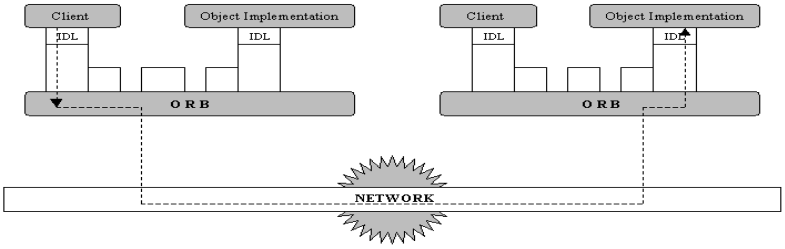
ORB

- The software that implements the CORBA specification
- Object bus that provides object location transparency
- Responsible for mechanisms to:
  - Find the object implementation of the request
  - Prepare object implem'n to receive the request
  - Communicate the data making up the request

12

ORB

- Client and object implem'n are isolated from the ORB by an IDL interface
- all requests (local or remote) are managed by the ORB



13

ORB

- Corba 1.0: every vendor would implement ORB differently, interoperability problems when communicating across ORBs
- Corba 2.0: specified GIOP, the General Inter-ORB Protocol:
  - collection of message requests ORBs can make over a network
  - GIOP maps ORB requests to different network transports
  - IIOP: Internet Inter-ORB Protocol maps GIOP messages to TCP/IP (mandatory for all ORBs)

14

Corba 3

- Released in 2002
- Major upgrades:
  - Java and Internet Integration
  - Quality of Service Control
  - The CORBAcomponent Architecture

15

CORBA Services

- Basic services that every object needs
- System level services with well-defined IDL interfaces
- They enhance functionality supported by ORBs

16

CORBA Services (Examples)

- **Naming Service**: find an object by name and bind to it
- **Event service**: supports notification of events to interested objects
- **Persistent object service**: provides a common set of interfaces for managing the state of objects
- **Trader service**: an alternative location facility to the naming service, finding objects by attributes in a wide-area network
- **Security service**: restrict access to objects/groups of objects to clients with appropriate privileges

17

IDL Basics

- Stands for: Interface Definition Language
- A CORBA object is specified with interface
- interface: contract between client and server
- specified in a special declarative language
- Lexical rules are same as C++ with new keywords
- IDL mapping to programming languages (e.g. C++, Java, etc) are provided in specs

18

CORBA IDL Interfaces

- IDL interface provides a description of services available to clients
- IDL interface describes an object with:
  - Attributes
  - Methods with their signatures
  - Exceptions
  - Inheritance information
  - Type and constant definitions

19

IDL Structure

```
module <identifier> {
  <type declarations>;
  <constant declarations>;
  <exception declarations;

   <interface definition>;
   <interface definition>;
};
```

20

IDL Structure (Modules)

- At the highest level, IDL definitions are packaged into module.
- A module is analogous to a package in Java
- Example:

```
module Bank {
   // body
};
```

21

IDL Structure (Interfaces)

- An IDL interface is the definition of an object
- IDL interfaces are analogous to Java interfaces
- An interface declares the operations that the CORBA object supports and its attributes

```
interface Account {
   // attributes
   // operations
};
```

22

IDL Structure (Interfaces)

- An interface may inherit from multiple interfaces
- It inherits all attributes and operations of its super-interfaces

```
interface JointSavingsAccount:
  JointAccount, SavingsAccount {
    // attributes
    // operations
};
```

23

IDL Structure (Attributes)

- They describe the variables (properties) of an interface
- An attribute can be **`readonly`** or read-write

```
interface Account {
   attribute string name;
   readonly attribute string sin;
   readonly attribute long accountNumber;
};
```

24

IDL Structure (Operations)

- They describe the methods of an interface
- They have a return type and parameters
- Parameters are flagged as:
  - **in**: parameter is passed from client to server
  - **out**: parameter is passed from server to client
  - **inout**: parameter is passed in both directions

```
void withdraw(in unsigned long amount);
void add(in long a, in long b, out long sum);
```

25

IDL Structure (Exceptions)

- IDL supports user-defined exceptions

```
exception InsufficientFunds {
  long currentBalance;
};

void withdraw(in unsigned long amount)
    raises (InsufficientFunds);
};
```

26

IDL Structure (Data Types)

- IDL supports a rich variety of data types
- Primitive:
  float, double, long, short (signed, unsigned), char, long long, boolean, octet
- Complex (discussed later):
  arrays, sequences, structures

27

IDL to Java Mapping

IDL	Java
boolean	boolean
octet	byte
char	char
string	String
short	short
long	int
long long	long
float	float
double	double
fixed	BigDecimal

28

Object Adapters

- Mediate between CORBA objects and programming language implementations
- Provide a number of services:
  - Creation of CORBA objects and their references
  - Dispatching requests to the appropriate servant that provides implementation for the target object
  - Activation and deactivation of CORBA objects

29

Object Adapters

- CORBA 2.0 defines the Basic Object Adapter (BOA)
- ORB vendors discovered that BOA is ambiguous and missing features so they developed proprietary extensions
- This resulted in poor portability between ORBs
- The new standard is: Portable Object Adapter (POA)

30

Anatomy of a CORBA-based App

- The steps involved:
  - Define an interface
  - Map IDL to Java (idlj compiler)
  - Implement the interface
  - Write a Server
  - Write a Client
  - Run the application
- Example: a step-by-step *Hello* example

31

Step 1: define the interface

- Hello.idl

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

32

Step 2: map Hello.idl to Java

- Use the idlj compiler (J2SE 1.3)

  idlj –fall Hello.idl

- This will generate:

  _HelloImplBase.java (server skeleton)
  HelloStub.java (client stub, or proxy)
  Hello.java
  HelloHelper.java
  HelloHolder.java
  HelloOperations.java

33

Step 3: implement the interface

- Implement the servant:

```
import HelloApp.*;
class HelloServant extends _HelloImplBase {
   public String sayHello() {
      return "\nHello There\n";
   }
}
```

34

Step 4: implement the server

- Import statements:

```
import org.omg.CosNaming.*;
import org.omg.CosNaming.NamingContextPackage.*;
import org.omg.CORBA.*;

class HelloServer {
   public static void main(String argv[]) {
     try {
```

35

Step 4: implement the server….

- Create and initialize the ORB:

```
ORB orb = ORB.init(argv, null);
```

- Create the servant and register it with ORB

```
HelloServant helloRef = new HelloServant();
orb.connect(helloRef);
```

36

Step 4: implement the server….

- Get the root NamingContext:

```
org.omg.CORBA.Object objRef =
    orb.resolve_initial_references("NameService");
NamingContext ncRef = NamingContextHelper.narrow(objRef);
```

37

Step 4: implement the server….

- Bind the object reference in naming

```
NameComponent nc = new NameComponent("Hello", " ");
NameComponent path[] = {nc};
ncRef.rebind(path, helloRef);
```

38

Step 4: implement the server….

- Wait for invocations from clients:

```
java.lang.Object sync = new java.lang.Object();
synchronized(sync) {
    sync.wait();
}
```

39

Step 4: implement the server….

- Catch the exceptions

```
    } catch(Exception e) {
        System.err.println("ERROR: " + e);
        e.printStackTrace(System.out);
    } // end catch
  } // end main()
} // end class
```

40

Step 5: write a client

- Import statements:

```
import HelloApp.*;
import org.omg.CosNaming.*;
import org.omg.CORBA.*;
class HelloClient {
    public static void main(String argv[]) {
      try {
```

41

Step 5: write a client….

- Create and initialize the ORB:

```
ORB orb = ORB.init(argv, null);
```

- Create the root naming context:

```
org.omg.CORBA.Object objRef =
   orb.resolve_initial_references("NameService");
NamingContext ncRef =
  NamingContextHelper.narrow(objRef);
```

42

Step 5: implement the client….

- Resolve the object reference in naming:

```
NameComponent nc = new NameComponent("Hello", " ");
NameComponent path[] = {nc};
Hello helloRef = HelloHelper.narrow(ncRef.resolve(path));
```

43

Step 5: implement the client….

- Call the object:

```
String Hello = helloRef.sayHello();
System.out.println(Hello);
```

- Catch exception:

```
} catch(Exception e) {
   System.out.println("ERROR : " + e);
   e.printStackTrace(System.out);
} } } // end catch, main, class
```

44

Step 6: run the application

- Run the naming service:

  prompt> tnameserver

- Run the server

  prompt> java HelloServer

- Run the client

  prompt> java HelloClient
  Hello There
  prompt>

45

CORBA Clients (details)

- Initializing the ORB
- Helper classes
- Holder classes
- Exceptions
- The naming service

46

Initializing the ORB

- Before invoking a CORBA object, you must first create an ORB object and initialize it
- public static void main(String argv[]) {
  try { ORB orb = ORB.init(argv, null); // .. }
  }
- Arguments to init():
  ORB init(String[] argv, Properties props)
  - **argv**: a list of command-line arguments to the app.
  - **props**: programmatically specify these options

47

Initializing the ORB….

- Command-line arguments (options):
  -ORBClass: that class that provides your ORB implementation (JavaSoft's ORB)
  -ORBSingletonClass: the class that provides your ORB singleton implementation (JavaSoft's ORB)
  -ORBInitialHost: the host where the naming service is running (default value: localhost)
  -ORBInitialPort: the port where the naming service is running. Default value: 900

48

Initializing the ORB….

- Properties:

  org.omg.CORBAClass
  org.omg.CORBA.ORBSingletonClass
  org.omg.CORBA.ORBInitialHost
  org.omg.CORBA.ORBInitialPort

- How to use these?

49

Initializing the ORB….

- Properties: alternative to command-line args
- Example:

```
Properties props = new Properties();
props.put("org.omg.CORBA.ORBInitialHost", initialHost);
props.put("org.omg.CORBA.ORBInitialPort", initialPort);
String noArgs[] = null;
ORB orb = orb.init(noargs, props);
```

50

Helper Classes

- They provide some utility functions (narrow)
- For interface Hello in module HelloApp, a helper class: HelloApp.HelloHelper is created
- Important method:

  Hello narrow(org.omg.CORBA.Object object)
- The naming service returns an Object, you must cast it to narrow it down….use narrow()

51

Holder Classes

- Used to implement “out” and “inout” parameters to CORBA operations.
- Why do we need them?

```
void op (inout long x); // alter the value of x
```

- But in Java:

```
int x = 10; op(x); // op is not allowed to change x
```

- However, “out” or “inout” mean that the actual value of the parameter will be changed.

52

Holder Classes

- Solution: use holder classes:
- For primitive data types: primitive holders

```
int x = 10;
IntHolder myx;
myx.value = x;
op(myx);
x = myx.value;
```

- BooleanHolder, CharHolder, StringHolder, etc

53

Holder Classes

- For user-defined types:

```
Employee employee = …;
EmployeeHolder emp;
emp.value = employee;
op(emp);
employee = emp.value;
```

54

Exceptions

- System exceptions

  org.omg.CORBA.SystemException

- User exceptions

  org.omg.CORBA.UserException

55

System Exceptions

- Those implicit, unpredictable exceptions that can arise as a result of a CORBA failure
- System exceptions are all implemented as a subclass of org.omg.CORBA.SystemException
- A subclass of java.lang.RuntimeException
- Therefore, any SystemException may be implicitly thrown by a CORBA operation without being declared in a throws clause

56

User Exceptions

- Those, explicit, predictable exceptions that are documented in IDL operation declarations.
- Raised by CORBA operations in response to specific situations (e.g. InsufficientFund)
- Implemented as a subclass of org.omg.CORBA.UserException
- A subclass of java.lang.Exception
- Therefore, they must be explicitly handled

57

CORBA vs. RMI

- CORBA interfaces defined in IDL, RMI interfaces are defined in Java
- CORBA is language-independent, RMI is not
- CORBA objects are not garbage collected, RMI objects are garbage collected automatically.
- RMI does not support “out” and “inout” operations since local objects are copied, and remote objects passed by reference to stub
- Communication protocols: IIOP vs. RMI

58

The Naming Service (Client's View)

- A tree-like directory for object references
- Much like a file system: provides directory structure for files
- Object references are stored by *name*
- Each object reference-name pair is called a *name binding*
- Name bindings may be organized under naming contexts (name binding itself)
- All bindings are stored under initial naming context (the only persistent binding)

59

The Naming Service….

- Your client's ORB must know the name and port# of a host running the naming service
- The naming service can either be the JavaIDL naming service or any COS-compliant service (COS: Common Object Services)
- To start: tnameserv –ORBInitialPort port#
  - The default port number is 900
- To stop: use relevant OS command (kill, ctrl-c)
- Namespace is lost if name server halts/restarts

60

The Naming Service (interfaces)

- org.omg.CosNaming:
  - **NamingContext**: primary interface to naming service
  - **NameComponent**: identify (name/kind) services
  - **BindingIterator**: iterating through the contents
  - **Binding**: a single entry in the naming service
  - **BindingList**: a list of entries in the naming service
  - **BindingType**: the type of an entry

61

Naming Service (*NamingContext*)

- Analogous to a directory on a file system
- Contains a series of named objects
- An object in a NamingContext may be another NamingContext (analogous to subdirectory)
- A reference to the top level NamingContext can be obtained with the ORB method:
  resolve_initial_references()

62

Naming Service (*NamingContext*)

- To get a reference to an object stored under NamingContext, use:

  resolve(NameComponent namePath)
- It throws: *NotFound*, *CannotProceed*, *InvalidName*
- This method returns org.omg.CORBA.Object
- Therefore, it must be narrowed to a particular interface using a helper's *narrow()*

63

Browsing the Naming Service

- The top level only….

```
import org.omg.CORBA.*;
import org.omg.CosNaming.*;
public class Browser {
  ORB orb = ORB.init (args, null);
  // obtain a reference to the naming service
  org.omg.CORBA.Object nc =
    orb.resolve_initial_references ("NameService");
   NamingContext namingContext =
    NamingContextHelper.narrow (nc);
```

64

Browsing the Naming Service….

```
BindingListHolder b1 = new BindingListHolder ();
BindingIteratorHolder b2 = new BindingIteratorHolder ();

// get initial content-list (retrieve up to 10, rest can be accessed
// by iterator)
namingContext.list (10, b1, b2);
// print out bindings
Binding[] bindings = b1.value;
if(bindings.length == 0) return;
```

65

Browsing the Naming Service….

```
for (int i = 0; i < bindings.length; i++) {
  Binding binding = bindings[i];
  NameComponent[] name = binding.binding_name;
  BindingType type = binding.binding_type;
  if (type == BindingType.nobject) {
     System.out.println (name[0].id + "-" + name[0].kind);
  } else { // BindingType.ncontext
   System.out.println (name[0].id + "-" + name[0].kind + "/");
  }
 }
```

66

CORBA Servers

- Implement the IDL interfaces by subclassing the appropriate pre-generated skeleton class
- Each class is called a *servant*
- The HelloServer Example
  - Initialize the ORB
  - Create initial objects (servants)
  - Connect each servant to the ORB
  - Bind the servants in the naming service
  - Wait for connections

67

CORBA Servers (ObjectImpl)

- When a servant extends the _interfaceObjectImpl, it is actually extending the orb.omg.CORBA.Portable.ObjectImpl class
- This class provides a variety of helper methods (including all methods of CORBA Object)

68

Naming Service (Server's View)

- Registering/Unregistering services:
  - **bind**: register the object under the specified name
  - **rebind**: identical to bind(), but an **AlreadyBound** exception won't be thrown – existing object replaced
  - **unbind**: unregister a CORBA object
- Creating new naming contexts:
  - **bind_new_context, new_context, bind_context**
- Destroying a naming context:
  - **destroy**: destroy an empty **NamingContext**

69

Clearing the Naming Service

- Steps:
  - Get a reference to initial naming context
  - Recursively iterate through the sub naming contexts
  - Call unbind
  - Call destroy

70

Advanced IDL

- IDL supports C/C++ style comments:
  // This is a comment
  /* This is another comment */
- Also, it supports:
  - conditionals (#if)
  - defines (#define)
  - includes (#include)
- idlj requires access to a C preprocessor (cpp)

71

Advanced IDL: Arrays

- IDL provides multidimensional fixed-size arrays
- The array size is fixed at compile time
- IDL arrays map directly into Java arrays
- Example:

```
interface Customer {
  attribute string address[4]; // 1-D array
  attribute short table[5][7];   // 2-D array
}
```

72

Advanced IDL: Sequences

- A sequence is a 1-D array that can be of variable size
- Two types:
  - Bounded sequences

```
sequence<long, 15>employee;
```

  - Unbounded sequences

```
sequence<long> employee;
```

73

Advanced IDL: Enumerations

- The *enum* data type defines an enumeration
  - A user-defined data type that can hold one of a fixed set of values
- Example:

```
enum CreditCard { visa, amex, discover };
interface Bank {
    void applyForCreditCard(CreditCard cc);
};
```

74

Enumerations (mapping to Java)

- An enum is mapped to a Java class with static variables representing the set of values
- Example:

```
// IDL
    enum EnumType {first, second, third};
// generated Java
    public class EnumType
    implements org.omg.CORBA.portable.IDLEntity {
        public static final int _first = 0;
        public static final EnumType first = new EnumType(_first);
        public static final int _second = 1;
        public static final EnumType second = new EnumType(_second);
        public static final int _third = 2;
        public static final EnumType third = new EnumType(_third);
        public int value() {...}
        public static EnumType from_int(int value) {...};
        // constructor
        protected EnumType(int) {...}
    };
```

- **To compare (using switch): (unknown.value() == _first)**

75

Advanced IDL: Structures

- The IDL type *struct* defines a structure
- Use a struct to group related data together
- Example:

```
struct Name {
    string firstName;
    string lastName;
};
interface Customer { attribute Name name; };
```

76

Structures (mapping to Java)

- A struct is mapped to a Java class that provides instance variables for the fields, and a constructor for all values, and a null constructor
- Example:

```
public class Name {
    public String firstName;
    public String lastname;
    public Name();
    public Name(String firstName, String lastName);
}
```

77

Advanced IDL: typedefs

- A typedef is an alias, or another name for an existing data type
- Example:

```
typedef long age;
interface Customer {
   age howOld;
}
```

- Typedefs of simple data types are mapped to the original (I.e. replaced by the more basic type)

78

Advanced IDL: Constants

- **1. Within an interface**:
  interface Foo { const long aLong = -32; };
- Mapped to: public interface Foo {
  public static final int aLong = (int) –32L; };
- **2. Not within an interface**:
  const string Message="hello";
- Mapped to: public interface Message {
  public static final String Message="hello"; };

79

Advanced CORBA Topics

- The Tie Mechanism
- Dynamic Invocation Interface (DII)
- Dynamic Skeleton Interface (DSI)
- Interface Repository (IR)

80

The Tie Mechanism

- All CORBA-based server programs we have seen so far extend a CORBA skeleton (ImplBase class) generated by the idlj compiler
- Since Java doesn't support multiple inheritance, we cannot inherit from a CORBA skeleton and another class
- Inheriting from a CORBA skeleton is not appropriate/uses up your one inheritance chance
- The tie mechanism offers an alternative to inheritance

81

How to use the tie mechanism?

- Implementation inheritance (Hello example)
  - HelloServant inherits its entire implementation from another class
  - Method requests for HelloServant are delegated to another idlj-generated class

82

Programming Example (Hello)

- Compile the IDL interface (Hello.idl) with the command: idlj –fall –tie Hello.idl
- This will generate two additional classes in the HelloApp directory
  1. _HelloOperations.java (the servant implements this interface)
  2. _HelloTie.java (acts as a skeleton, receiving requests from the ORB and delegating them to the servant that does the actual work

83

HelloBasic

- A new class: HelloBasic.java

```
public class HelloBasic {
   public String sayHello() {
      return “Hello World\n”;
   }
}
```

84

HelloServant

- HelloServant.java

```
class HelloServant extends HelloBasic implements
   _HelloOperations {
}
```

85

HelloServer

- HelloServer.java

```
class HelloServer {
   public static void main(String argv[]) {
      // create and initialize ORB
      …
      // Create servant and register it with ORB
      HelloServant servant = new HelloServant();
      Hello helloRef = new _HelloTie(servant);
      // connect ….etc
}
```

86

The Tie Mechanism vs. Inheritance

- Inheritance is easier as implementation objects look like normal object references
- If object implementations are in the same process as the client then method invocations are cheaper (because no delegation)

87

Dynamic Invocation Interface (DII)

- IDL interfaces used by a client are determined when the client is compiled
- Therefore, the developer is limited to using servers that contain objects that implement those interfaces
- This is not enough if an application (e.g. a distributed debugger) requires the use of interfaces that are not defined at the time the application was developed

88

DII

- The solution is provided by CORBA's DII
- DII allows an application to invoke operations from any interface (clients use the IR to learn about unknown object interfaces).
- Static operation requests are more efficient but DII is good for:
  - Clients issue requests for any operation (which may not be known as compile time)
  - Client operations don't need to be recompiled in order to access newly activated object implmnt's

89

Dynamic Skeleton Interface (DSI)

- DSI provides a way to deliver requests from an ORB to an object implementation without any compile-time knowledge of the objects it is implementing
- DSI is analogous to DII:
  - DII is for the client-side
  - DSI is for the server-side

90

Interface Repository (IR)

- IR is like a database that contains data that describes CORBA interfaces or types
- The data in the IR is equivalent to that in an IDL file except that data in the IR is represented in a such a way that makes it easier for clients to use
- Clients use the IR to learn about unknown object interfaces and they use DII to invoke methods on that object

91