Object-Based Distributed Systems

INF5040/9040 Autumn 2015
Lecturer: Amir Taherkordi (ifi/UiO)

September 14, 2015

Outline

1. Local Procedure Call
2. Remote Procedure Call
3. Distributed Objects
4. Remote Method Invocation
5. Object Server
6. CORBA
7. Java RMI
8. Summary
Object-Based DS

Local Procedure Call

- Many distributed systems:
  - based on explicit message exchange between processes
- How is it done in a single machine?
  
  **Local Procedure Call, e.g.:**
  
  \[
  \text{count} = \text{read}(\text{fd}, \text{buf}, \text{nbytes});
  \]

- Parameter passing in a local procedure call
- Parameter passing:
  - call-by-value: \( \text{fd} \) and \( \text{nbytes} \)
  - call-by-reference: \( \text{buf} \)

Object-Based DS

Remote Procedure Call (1)

- Ideally:
  - make a remote call look as a local one
  - in other words: achieving **access transparency**
- The basic idea:
Object-Based DS

Remote Procedure Call (2)

- A RPC occurs in the following 10 steps:

  - Remote Procedure Call (2)

  - The net effect of these steps:

    To convert the local call by the client procedure to a local call to the server procedure without either client or server being aware of the intermediate steps or the existence of the network.

- These steps seem straightforward?
  - how about taking parameters by the client stub, packing them, and sending them to the server stub?
    - passing value parameters
    - passing reference parameters
Object-Based DS 2. Remote Procedure Call

Passing Value Parameters (1)

- Parameter **marshaling**: packing parameters in a message
- **add(i, j)** example:

  ![Diagram of marshaling](image)

  1. Client call to procedure
  2. Stub builds message
  3. Message is sent across the network
  4. Server OS hands message to server stub
  5. Stub unpacks message
  6. Stub makes local call to "add"

Passing Value Parameters (2)

- This model works as long as:
  - client and server machines are **identical**
  - all parameters and results are **scalar/base types** (int, char, boolean, …)
- **Challenges:**
  - in DS: each machine has its own representation of data: e.g., IBM mainframe: EBCDIC code, while IBM pc: ASCII
  - **byte numbering**: left-to-right or other way

  ![Byte numbering example](image)
Object-Based DS | 2. Remote Procedure Call

Passing Reference Parameters

- How to pass references (pointers)?
  - pointers are meaningful within the address space of the process
  - not possible to pass only the address of parameter

- One solution:
  1. copy the array into the message and send to the server
  2. server stub calls the server with a pointer to this array
  3. server makes changes to the array
  4. message will be sent back to the client stub
  5. client stub copies it back to the client

- How about pointers to arbitrary data structures:
  - e.g., complex graph
  - solution: passing pointer to server and generating special code for using pointers, e.g., code to make requests to client to get the data

Object-Based DS | 2. Remote Procedure Call

Stub Generation

- What we understood so far:
  - the same protocol for both client and server: e.g.,
    - agree on the format of messages
    - representation of simple data structure

- A complete example:

  ```
  foobar(char x; float y; int z[5] {…}
  ```

- Next step after defining RPC protocol:
  - implementing client and server stubs
  - stubs for the same protocol but different procedures
  - Differ only in their interface
### Outline

<table>
<thead>
<tr>
<th>Applications (+Distributed Objects)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Middleware Services</td>
</tr>
<tr>
<td>RMI and RPC</td>
</tr>
<tr>
<td>request-response protocols</td>
</tr>
<tr>
<td>marshalling and external data</td>
</tr>
<tr>
<td>representation</td>
</tr>
<tr>
<td>UDP og TCP</td>
</tr>
</tbody>
</table>

### Characteristics of Distributed Objects (1)

- **Distributed objects** execute in different processes:
  - each object has a **remote interface** for controlling access to its methods and attributes that can be **accessed from other objects** in other processes located on the same or other machines
  - declared via an **"Interface Definition Language"** (IDL)
- **Remote Method Invocation (RMI)**
  - method call from an object in one process to a (remote) object in another process
Characteristics of Distributed Objects (2)

- **Remote Object Reference (ROR):** unique identity of distributed objects
  - other objects that want to invoke methods of a remote object needs access to its ROR
  - RORs are "first class values"
    - can occur as arguments and results in RMI
    - can be assigned to variables
- Distributed objects are **encapsulated by interfaces**
- Distributed objects can raise "exceptions" as a result of method invocations
- Distributed objects have a set of **named attributes** that can be assigned values

The Type of a Distributed Object

- **Type of an object:**
  - Attributes, methods and exceptions are properties that objects can export to other objects
  - several objects can export the same properties (same type of objects)
  - the type is defined once
  - The object type is defined by the **interface specification** of the object
A remote method is declared by its **signature**

In **CORBA** the signature consists of
- a **name**
- a list of **in**, **out**, and **inout** **parameters**
- a **return value** **type**
- a list of **exceptions** that the method can raise

**void select**( **in** Date *d*) **raises** (**AlreadySelected**);
A client object can request the execution of a method of a distributed, remote object.

Remote methods are invoked by sending a message (including method name and arguments) to the remote object.

The remote object is identified and located using the remote object reference (ROR).

Clients must be able to handle exceptions that the method can raise.

Local objects can invoke: the methods in the remote interface + other methods implemented by a remote object.
Three main tasks:
- Interface processing
  - integration of the RMI mechanism into a programming language.
  - basis for realizing access transparency
- Communication
  - message exchange (request-reply protocol)
- Object location, binding and activation
  - locate the server process that hosts the remote object and bind to the server
  - activate an object-implementation
  - basis for realizing location transparency

RMI Interface Processing

Role of proxy and skeleton
Elements of the RMI Software (1)

- **RMI interface processing: Client proxy**
  - local “proxy” object for each remote object and holds a ROR (“stand-in” for remote object).
  - the class of the proxy-object has the same interface as the class of the remote object.
  - can perform **type checking** on arguments.
  - performs **marshalling** of requests and **unmarshalling** of responses.
  - transmits request-messages to the server and receive response messages.
  - Makes remote invocation transparent to client.

Elements of the RMI Software (2)

- **RMI interface processing: Dispatcher**
  - A server has one dispatcher for each class representing a remote object:
    - receives requests messages
    - uses *method id* in the request message to **select the appropriate method** in the skeleton (provides the methods of the class) and passes on the request message.
RMI interface processing: **Skeleton**
- **one** skeleton for **each** class representing a remote object
- provides the methods of the remote interface
- **unmarshals** the arguments in the request message and invokes the corresponding method in the remote object.
- **waits** for the invocation to complete and then
- **marshals** the result, together with any exceptions, in a reply message to the sending proxy’s method.

Elements of the RMI Software (4)

- Remote object reference module

![Diagram](image-url)
Object-Based DS 4. Remote Method Invocations

Generation of Proxies, Dispatchers and Skeletons

- **Design**
  - IDL definitions
  - Pre-compile
  - Client code
  - Proxies
  - Skeletons
  - Add server code
  - Compile
  - Client implementation
  - Server implementation

Object-Based DS 4. Remote Method Invocations

Server and Client Programs

- **Server** program contains
  - the **classes** for the **dispatchers** and **skeletons**
  - the implementation classes of all the **servants**
  - an **initialization section**
    - creates and initializes at least one servant
    - additional servants (objects) may be created in response to client requests
  - register zero or more **servants** with a **binder**
  - potentially one or more **factory methods** that allow clients to request creation of additional servants (objects)

- **Client** program contains
  - the classes and **proxies** for all the remote objects that it will invoke
RMI Name Resolution, Binding, and Activation

- **Name resolution**
  - mapping a *symbolic object name* to an ROR
  - performed by a name service (or similar)

- **Binding** in RMI
  - locating the server holding a remote object based on the ROR of the object and placing a proxy in the client process’s address space

- **Activation** in RMI
  - creating an active object from a corresponding passive object (e.g., on request).
  - register passive objects that are available for activation
  - activate server processes (and activate remote object within them)

---

RMI Sequence Diagram

![RMI Sequence Diagram](image-url)
Implicit and Explicit Binding

Distr_object* obj_ref;  // Declare a system wide object reference
obj_ref = lookup(obj_name);  // Initialize the reference to a distrb. obj
obj_ref->do_something();  // Implicit bind and invoke method

Distr_object* obj_ref;  // Declare a system wide object reference
Local_object* obj_ptr   // Declare a pointer to a local object
obj_ref = lookup(obj_name);  // Initialize the reference to a distrb. obj
obj_ptr = bind(obj_ref);  // Explicitly bind and get pointer to local proxy
obj_ptr->do_something();  // Invoke a method on the local proxy

Object Server

- The server
  - is designed to host distributed objects
  - provides the means to invoke local objects, based on requests from remote clients
- For an object to be invoked, the object server needs to know
  - which code to execute
  - which data it should operate
  - whether it should start a separate thread to take care of the invocation
Object-Based DS  5. Object Server

Activation Policies

- **Transient objects**: creating object at the first invocation request and destroying it when no clients are bound to it anymore
  - **advantage**: object uses server’s resources only it really needs
  - **drawback**: taking time to make an invocation (object needs to be created first)
  - **an alternative policy**: creating all transient objects during server initialization, at the cost of consuming resources even when no client uses the object.

- **Data and Code Sharing**:
  - sharing **neither code nor data**: e.g., for security reasons.
  - Sharing objects’ **code**: e.g., a database containing objects that belong to the same class

- **Policies with respect to threading**:
  - single thread
  - several threads, one for each of its objects: how to assign threads to objects and requests? One thread per object? One per request?

---

Object-Based DS  5. Object Server

Object Adaptor/Wrapper

- A mechanism to **group objects per policy**.
- software implementing a specific activation policy

Upon receiving invocation request:
- it is first dispatched to the appropriate object adapter
- adaptor **extracts an object reference** from an invocation request
- adaptor **dispatches** the **request** to the referenced object, but now following a specific activation policy, e.g., single-threaded or multithreaded mode
Object-Based DS

Outline

<table>
<thead>
<tr>
<th>Applications (+Distributed Objects)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Middleware Services</td>
</tr>
<tr>
<td>RMI and RPC</td>
</tr>
<tr>
<td>request-response protocols</td>
</tr>
<tr>
<td>marshalling and external data rep.</td>
</tr>
<tr>
<td>UDP og TCP</td>
</tr>
</tbody>
</table>

Common Object Request Broker Architecture (CORBA)
Object-Based DS

CORBA Middleware (1)

- Offers mechanisms that allow objects to invoke remote methods and receive responses in a transparent way
  - location transparency
  - access transparency
- The core of the architecture is the **Object Request Broker (ORB)**
- Specification developed by members of the Object Management Group (www.omg.org)

CORBA Middleware (2)

- Clients may invoke methods of remote objects without worrying about:
  - object location, programming language, operating system platform, communication protocols or hardware.

![Diagram showing object interaction with Object Request Broker (ORB) and IDL interfaces]

Different programming languages (or object models)

- Common object model
- RMI over IIOP
### Supporting Language Heterogeneity

- CORBA allows interacting objects to be implemented in different programming languages
- **Interoperability** based on a common object model provided by the middleware
- Need for *advanced mappings* (language bindings) between different object implementation languages and the **common object model**

### Elements of the Common Object Model

- Metalevel model for the type system of the middleware
- Defines the meaning of e.g.,
  - object identity
  - object type (interface)
  - operation (method)
  - attribute
  - method invocation
  - Exception
  - subtyping / inheritance
- Must be general enough to enable mapping to common programming languages
- **CORBA Interface Definition Language** (IDL)
CORBA IDL

- Language for specifying CORBA object types (i.e. object interfaces)
- Can express all concepts in the CORBA common object model
- CORBA IDL is
  - not dependent on a specific programming language
  - syntactically oriented towards C++
  - not computationally complete
- Different bindings to programming languages available

CORBA Architecture

[Diagram showing CORBA architecture with client, server, ORB core, and implementation interface repository.]
Object-Based DS

6. CORBA

CORBA Services

Object Request Broker (ORB)

Application objects

Domain Interfaces

Business objects
- E-Commerce
- Financial Domain
- Healthcare
- Telecomm

……

CORBA Facilities

Compound doc
Help
Printing
……

Object services

Naming
Life cycle
Persistence
Events
Transactions
Trading
Time

Concurrency
Relationships
Externalization
Querying
Licensing
Security
Properties

Description of the services:
Coulouris ch. 8, Figure 8.6

Java Remote Method Invocation (RMI)

Java
### Java RMI

- **Remote Method Invocation** (RMI) supports communication between different *Java Virtual Machines* (VM), and possibly over a network
- Provides tight integration with Java
- Minimizes changes in the Java language/VM
- Works for homogeneous environments (Java)
- Clients can be implemented as *applet* or *application*

### Java Object Model

- Interfaces and Remote Objects
- Classes
- Attributes
- Operations/methods
- Exceptions
- Inheritance
Java Interfaces to Remote Objects

- Based on the ordinary Java interface concept
- RMI does not have a separate language (IDL) for defining remote interfaces
  - pre-defined interface Remote
- All RMI communication is based on interfaces that extends java.rmi.Remote
- Remote classes implement java.rmi.Remote
- Remote objects are instances of remote class

Example

```java
interface Team extends Remote {
    String name() throws RemoteException;
    Trainer[] trained_by() throws RemoteException;
    Club club() throws RemoteException;
    Player[] player() throws RemoteException;
    void chooseKeeper(Date d) throws RemoteException;
    void print() throws RemoteException;
}
```

`interface name` declares the Team interface as “remote”
### Parameter Passing

- Atomic types transferred *by value*
- Remote objects transferred *by reference*
- Non-remote objects transferred *by value*

```java
class Address {
    public String street;
    public String zipcode;
    public String town;
};

interface Club extends Organisation, Remote {
    public Address addr() throws RemoteException;
    ...
};
```

*Returns a copy of the Address-object*

### Architecture of Java RMI

![Architecture of Java RMI Diagram]

- Client
- Server
- Stub
- Registry Interfaces
- Generic Dispatcher
- Activation Interfaces
- RMI Runtime (rmid, RMIregistry)

*Non-persistent name server*
Object-Based DS

Summary (1)

- Remote Procedure Calls
- Distributed objects executes in different processes.
  - remote interfaces allow an object in one process to invoke methods of objects in other processes located on the same or on other machines
- Object-based distribution middleware:
  - middleware that models a distributed application as a collection of interacting distributed objects (e.g., CORBA, Java RMI)

Summary (2)

- Implementation of RMI
  - proxies, skeletons, dispatcher
  - interface processing, binding, location, activation
- Object servers
  - object adapters and activation policies
- Principles of CORBA
  - clients may invoke methods of remote objects without worrying about: object location, programming language, operating system platform, communication protocols or hardware.
- Principles of Java RMI
  - similar to CORBA but limited to a Java environment