Distributed Objects and Remote Invocation

George Ajam, Lecture notes 26-4-2011
Introduction

• Distributed Applications: Applications that are designed to have cooperating programs that run in several different processes.

• Several familiar programming models that have been extended to distributed programs:
  1. Move from conventional procedure call to Remote Procedure Call (in different processes).
  2. Remote Method Invocations: allows different objects to call each others.
  3. Objects receive notification if other objects have been changed by some external events.
• Visibility: Object B is Visible to Object A if and only if A can invoke Methods defined in Object B.

• Types of visibility:
  1. Object B is a parameter to some function in object A.
  2. Object B is defined as a member variable in Object A.
  3. Object B is declared globally for Object A.
  4. Object B is defined in some function in Object A.
• RPC is the same of Remote Method Invocation (RMI) in the case of using objects environment.

Middleware:
• It is a software that provides programming model above the basic building blocks of processes and messages passing between objects.
• An important aspect in middleware is the location transparency.
• There is an independency from the details of communication protocol, OS, and HW.
• Location Transparency: it is not important for the client where is the object is located whether in the same system or in other system. Also the same thing for the event-based programs to compose Distributed Systems, because it is not important where are the objects that make the events and where the objects that receive these events.

• Communication Protocols: Protocols that support the middleware abstraction are independent from the networking protocols (request-reply protocol is independents from TCP or UDP).
• Computer Hardware: there is no difference between different architectures, middleware will handle that.
• OS: the services activities provided by middleware are independent from the Operating Systems used.
• Use of several Programming Languages: Middleware has the capability to allow using different programming languages. (different objects from different PLs).
Interfaces

- Modularity: most programs are designed as set of modules that communicate with each others.
- This communication can be by procedure calls, or by direct access to variables in another modules (public).
- Interface must be defined for each module in order to control the possible interactions between these modules.
- Interface for a module provides a mechanism for how to use services for that modules (procedures and variables).
- Interfaces considered as protections for that data in the module, they allow access to specific data.
Interfaces in the distributed systems:
• it is not allowed for a module to access variable in another module in the same process in the DS. Middleware breaks this rule by using getters and setters for the module.
• Caller and receiver in different process – call by value and call by reference are not allowed.
• Parameters are sent from module to another by messages (input).
• The result of the call – output – are parameters returned from the module by a message.
• Pointers can’t be transmitted between modules because they are different if defined in different processes. (different Computers).

• Service Interface: is the way the Server provides its services, it describes for the clients how to use different functions defined in the server (it’s a contract)-this is in client server model.

• Remote Interface: is how the object will invoke another object’s methods, how the inputs and outputs will take place.
Communication between distributed objects

- The Object Model.
- Distributed Objects.
- Distributed Object Model.
- Design Issues for RMI.
- Implementation of RMI.
The Object Model

- Object-Oriented Program is composed of interacting Objects that invoke methods.
- Object = set of data + set of methods.
- Encapsulation.
- In non-distributed system, there is possibility for accessing public variables.
- In distributed systems, there must be setters and getters.
• Object References: variables that hold an object.
• To invoke a method in the object, object reference must be followed by the method name and the list of arguments (if there are).
• Target or Receiver Object: is the object that its method is invoked.
• First-class values: are object references that are assigned to other variables, passed as arguments, or returned as return values.
Interfaces

- Interfaces for objects are the specification of the objects, it provides a complete description about the object (its methods and there arguments, their return values ad how and where they can be used) without specifying their implementation (encapsulation).
- In Java, a class may have several interfaces.
• Actions: an action in object-oriented program is initiated by the client object in order to invoke a method defined in the server object.
• The control passed from the client to the server, and when the server executed its method, the control returns back to the client.
• Invoking a method can result in:
  1. Change in receiver state.
  2. Invoking another methods in another objects. (state machine diagram).
• An event happened, triggers an action or an activity. (ex: hydroponics gardening system).
• Exceptions: programmers need to take in mind that system may have some errors during run time, so exceptions ease the writing of code in order to prevent these errors to take place (ex: reading from file, division by Zeros, wrong argument…etc).
• A block of code can *Throw* an exception to run another code in the case of a problem.
• This means that control passes to the *catch* block in the case of exception.
• Garbage Collection: it is a way for freeing the memory from objects that are not used anymore.
• Java has its own Garbage Collector.
• C++ does not have, programmer has to free it in its program (using destructors).
Distributed Objects

- State of the object: its variables with their values.
- The state of the program consists of its different objects states.
- In RMI, a message is a method invocation on one object from another object, both of them may reside in different processes.
- To allow a chain of related invocations, objects in some servers that act as servers, may act as clients and invoke other messages on another server objects in another systems.
• Encapsulation is very important in the case of using distributed objects, in order to enhance security.
• This means that the state of a server object can be changed only by its methods.
The distributed object model

- Each process contains a collection of objects.
- Some of these objects can receive remote and local invocations, others receive local invocations only.
Figure 1.

- B and F are remote Objects.
- Object C has a reference to Object E, so it can invoke a method in E. (E is visible to C).
- Remote Object Reference: object that can run methods on remote object must has a remote object reference for that object. A must has remote object reference for B.
- Remote interface: every remote object must has an interface which specifies which methods can be invoked remotely.
• Remote Reference Object: is a unique number that is assigned for a particular object in order to make it remote object, and this reference can be passed as argument and result of remote method invocation. (Like cardinality in Object Model).

• Remote interfaces: it is a set of rules, that make an object just to invoke specified methods in another object. In this case object a has an interface for object b.
• The usefulness of Interfaces, is that, you don’t need to know what is the programming language used in implementing a remote object in order to invoke its interface methods, instead, you can do that using any programming language considered as object based programming language.
Remote Object and its Remote Interface
• Actions in Distributed Object Systems: as in non-distributed systems, there is some action that makes an invocation of some method in an object, but with the difference in that this invocation takes place between different processes or different computers.
• When the invocation passes the boundaries for the process, it is converted to RMI, and in this case the remote object reference must be specified.
• Back to figure 1, object A may obtain a reference for Object F using Object B.
• Garbage Collection in Distributed Systems: if using JAVA for example, RMI system must provide automated Garbage Collection by default. This will be discussed in detail later.
• Exceptions: the distributed system may fail during to any reason. For example, a process that contains the remote object may be too busy to reply, or the computer that holds the whole process may fail or crash.
• So, RMI must be able to throw exceptions
The idea behind the use of events is that one object can react to a change occurring in another object. For example, when a user interacts with an application by mouse clicks, these interactions are seen as events that cause changes in the objects that compose the whole application, and may change in the whole state of the application.
• Distributed event-based systems extend the local event model by allowing multiple objects at different locations to be notified of events taking place at an object.
• Distributed Event-Based Systems has two characteristic:
  1. Heterogeneous: when using event notifications to communicate between objects in different processes, these objects can be communicate although they are not designed to interoperate. All what needed is that, the event-generating objects publish the types of events they offer, and the other objects subscribe to events and provide an interface for receiving notifications.
  2. Asynchronous: Notifications are sent asynchronously (without time arrangements – like emails). There is no need for the server object to still wait for the client to send it the message, instead the server object can serve other clients, and when it receives a message, it handles it. (there is no coupling between clients and servers).