ABOUT DISTRIBUTED AND MOBILE OBJECTS AND COMPONENTS IN JAVA

M. Ivanović, N. Ibrajter

Faculty of Natural Sciences, University of Novi Sad
Trg Dositeja Obradovića 4., 21000 Novi Sad, Yugoslavia
{mira, natasha}@im.ns.ac.yu

Abstract: This paper is a short overview of basis of distributed programming, as well as mobile and component-base programming in Java. The most popular commercial tools are examined and evaluated.

Keywords: distributed systems, ORB, mobile agents, component-based programming, CORBA, Caffeine, Voyager, RMI, COM/DCOM, Java.

1. Introduction

Object-oriented programming is based on modeling problems with objects, meaningful elements of a problem. As objects contain meaning, it is not hard to imagine how convenient it would be to move an object with desired meaning (behavior) from one application to another, or just to remotely invoke desired method of the object. But, one object can invoke some other object only if both of them are part of the same local address space and are referred to by some local reference or pointer.

Distributed object computing is technology that enables objects from different programs to access each other. Traditional communication between different software applications requires sending sets of row data back and forth.

An object is called ***mobile*** if it is enabled to move between two or more applications. Both the state and the code are transferred.

Writing source code is tiresome and error-prone process, but it was basis of software development. It would be much easier to have small bundles of programming functionality that can be easily incorporated into program with simple drag-and-drop operation. These instant parts of source code are called ***components*** [1]. Components integration framework defines the API (Application Programming Interface) for allowing components developed by independent developers to work together in the same program without additional overhead for the user of the components.

2. Distributed Objects

When two different applications communicate, they actually send sets of raw data back and forth between two computers. These sets of data are called ***packets***. The application that requests a certain service is called ***client***, and the one that provides the service is called ***server***.

The client must be able to convert its request for service into packets, and send it to server. The packets contain the identifying information about the object in the server, which is the target, the information about the specific method of the object that is used, and it must send all the parameters for the method. Software developer must be able to convert all the types into raw data. This process is called ***externalization***. The server side, receiving requests from client must interpret all the received data in order to invoke the right method of the right object with the right parameters. It is developer's responsibility to write a code that will do the externalization, as well as the reverse process on the server side.

In order to make application communication easier, ORB (*Object Request Broker*) was developed. It is a tool that enables objects to communicate on a computer network (that is way they are called brokers) by sending requests and getting answers on those requests.

2.1 What does it take to make object communicate with ORB?

The first step in making object communication is to create ***interface*** for that object. Interface defines which methods the object offers for remote invocation. It is a sort of promise that objects really knows to perform all the actions listed in the interface. This interface is used by ORB to enable interaction between objects written in different programming languages. Interface is written in IDL (*Interface Definition Language*) and stored in separate file. The ORB uses information in the interface to determine which packets the object can and cannot receive. As the ORB completely automates the interaction of objects and encapsulates all the details about networking software and hardware, the developer doesn't need to mess with streams and packets.

With ORB, it is possible to write object-oriented software that employs a mixture of local and remote objects. The remote objects appear identical to the local ones, and this property is called a *location transparency*.

3. CORBA – Common Object Request Broker

CORBA is a common standard which enables ORBs from different vendors to interoperate [5]. CORBA provides a standard IDL syntax. Any language can work with this IDL by providing a mapping between its own and the CORBA IDL constructs. CORBA introduced a standard API (*Application Programming*

Interface) for the most of the functions of an ORB: initializing ORB, invoking methods on remote objects, mapping data types between programming languages.

3.1 How CORBA works?

When, for instance, client requests service from a server, it seems to it that method is invoked on a local object. But, it is not so: in object-oriented programming only local objects can be target of object invocation. A new object, which takes place of a distributed object in local address space of the client, (Fig. 1.) is created. It is called ***stub*** and it supports all the methods specified in the distributed object IDL file. Stub provides the illusion that that the remote object can be invoked in the local address space.

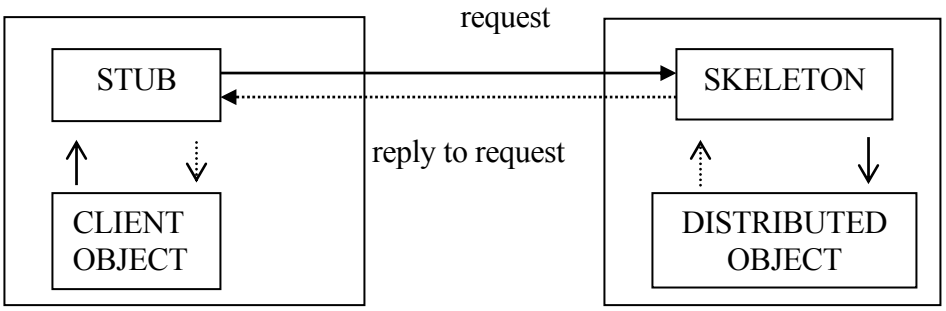

Figure 1: CORBA distributed object's communication

Stub is automatically generated by distributed object toolkit, by compiling the IDL file. The client's invocations of remote object methods are accepted by the stub and forwarded to the remote object. CORBA loosely standardizes what each of ORBs should put in the stub and what APIs should be offered to the CORBA developer for manipulating the stubs. In this way, stubs generated with one CORBA product are loosely compatible with other CORBA products. On the server side exists the code, ***skeleton***, which accepts requests from the network and invokes the appropriate methods. Skeleton accepts raw data from the network connection, turns it back in method invocation, and actually invokes the method. Skeletons are automatically generated.

The way to register an object that can be remotely accessible, to expose it to the rest of the network is to use an ***object adapter***. It provides the unique, or universal addresses for the distributed objects. The earliest versions of CORBA standard specified a certain API, which CORBA products should have implemented in order to be CORBA compatible. That special API was for the Basic Object Adapter (BOA). BOA had operations required for maintaining and exposing objects as able for remote invocation and included operations of registering and unregistering object available for remote invocation. The fact that BOA was too

loosely specified caused adoption of a new object adapter standard called Portable Object Adapter (POA).

3.2 More on CORBA

CORBA standardize the way in which ORB turns actions on the object into packets, down to bits. Thus, CORBA in fact standardize streams and packets, allowing any CORBA compatible software to communicate, apart from the vendors. CORBA provides a common standard for packet format-General Interoperable Protocol (GIOP). It standardizes the way TCP/IP is used to communicate between clients and servers-Internet Interoperable Protocol (IIOP).

Limitations to CORBA

Since CORBA works with many programming languages (such as C, C++, COBOL, ADA, Smalltalk, Java), it can provide only a minimal toolset common to these very different languages. As the least-common-denominator, CORBA can not have features such as garbage collection (C++ lacks it), serialization of objects (not available in COBOL), primitive meta-class support (poorly supported in Java).

CORBA does not support mobile code, although Object Management Group (OMG) is working on an extension called Objects-By-Value and mobile agent framework [1].

Distributed systems consist of few servers and many clients. Servers are usually kept under close supervision, and software upgrading on server side is not a problem. However, upgrading client side software can be very daunting task. Distributed object tools do not offer any solution to this problem.

CORBA does not support broadcast communication, nor the way to encapsulate clients and servers in distributed system within components. CORBA objects are essentially static. In spite of all this, CORBA has successfully provided powerful capabilities in every language it was implemented in and made distributed software developer life much easier.

4. Mobile Objects

The mobile object applications exchange complete objects, including their state (all the current values of all member variables of the object) and implementation. Distributed applications, in opposite, communicate whit each other by sending packets.

Mobile objects move their code and state between two or more applications. Java enables such behavior through ***Object Serialization*** and ***ClassLoader*** [1]. Object Serialization provides an automatic mechanism for reading and writing state of

an object to a data stream. ***ClassLoader*** locates and loads the bytecode for a Java class, even across the network, see Fig. 2.

Mobile object differ from mobile agents in not being able to act on their own. Mobile objects are not autonomous and they cannot move themselves.

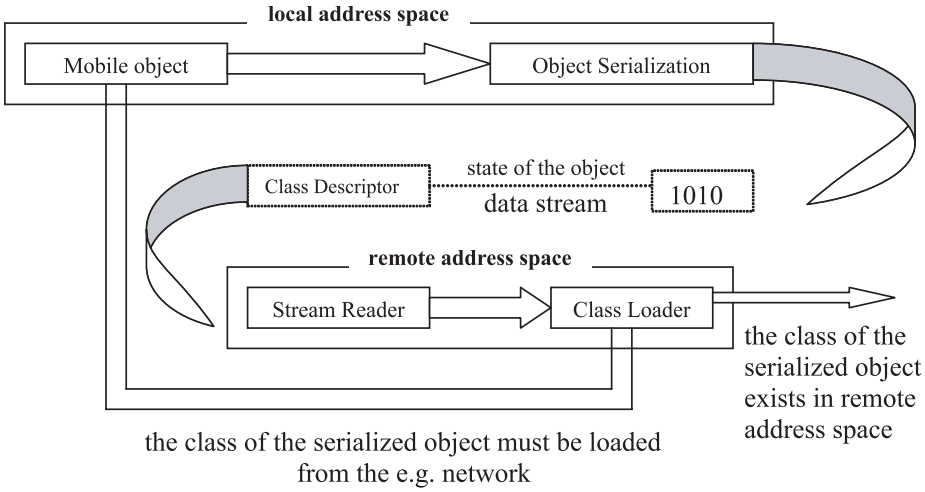

Figure 2: Mobile object moving mechanism

4.1 Object state

Object state is a state of all member variables of the object. The state of an object does not include source code in Java (this is not correct for some weakly typed languages). Object Serialization is responsible for copying the object to a stream. Mobile object tools should just write and read the stream. In order to be serializable, object must implement one of two special base interfaces: ***java.io.Serializable***, or ***java.io.Externalizable***. The first interface provide automatic serialization to a stream, and the objects that implement the second interface must implement their own methods that are responsible for writing and reading the object to a stream, see Fig. 2.

Object serialization automatically copies member variables of an object into the stream, but only if they are not static or transient.

4.2 Class loading

The state of an object is not an object make. No code is associated with the values of the member variables. In mobile object construction, the flexibility of Java class loader is critical. When the object arrives in a new address space, the required class for the object should be found and an instance should be created

with the serialized state. If the class for the object is available in local address space, there is no need for a class to be downloaded from network, see Fig. 2.

The first thing that Object serialization writes to the stream is a ***ClassDescriptor***-object that describes the class for that particular object. The reader of the stream uses this object as information about the class that is to be instantiated.

4.3 Problems

Mobile objects move across the network. This is not complete truth. What really happens is cloning object inside a remote address space. After the "moving" of a mobile object, in a mobile object system exist two copies of the same object: one in the local address space, and the other in the remote address space, see Fig. 3.

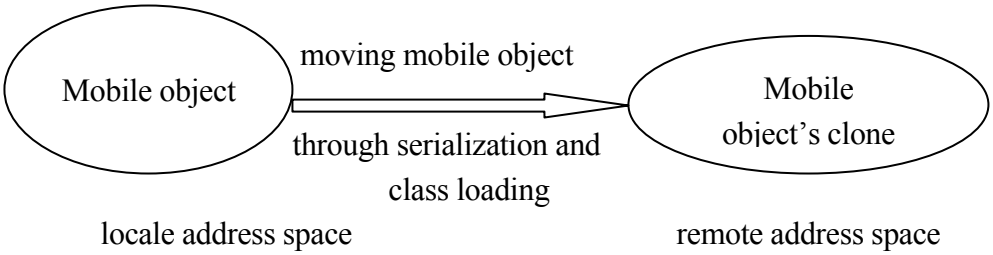

Figure 3: Moving mobile objects

If the object in the local address space has some references that point to it, it is impossible to destroy it, because of the basic premises of the languages with garbage-collection capabilities. Modifying all the references is sometimes impossible. Some mobile object tools provide certain mechanisms for securing object identity, but it varies from tool to tool.

The issue of the security is always weak point in distributed systems, and it is not different with the mobile object/agent systems. Java's ***SecurityManager***, built on a send-box principle, is an effective way of protecting hosts from malicious agents. The problem of protecting agents/objects from malicious host has not yet been successfully solved.

5. Mobile Components

Mobile objects become too low level and complex to work with. Mobile components in association with component integration framework enable developer to use pieces of already existing functionality and compose them (by few drag-and-drop mouse operations) in a new application. ***JavaBeans*** form the component standard for Java [2]. An object with a set of associated objects represents ***JavaBean***. Each ***JavaBean*** may have properties, methods and events; just like an object. Beans are stored in JAR files. JAR file is a ZIP archive that contains all the

files relevant to the ***JavaBean***, together with a metadata file, which specify which classes are Beans and which are just associated classes and resource files. The Jar file contains Beans in two formats: as a class or as a serialized object. This second approach enables Bean, together with its current state to be pickled into the JAR file, so the Bean does not loos all of its state between invocations.

In March 1998., ***Enterprise Java Beans*** (EJB) was first released. According to Sun specification, the EJB architecture is a component architecture for the development and deployment of object-oriented distributed enterprise-level applications. Applications written using EJB architecture are scalable, transactional and multi-user secure [1].

6. Caffeine

Caffeine is a tool in Borland's VisiBroker for Java. Its design enables writing CORBA applications for Java developers as simple as possible. CORBA requires the developer to learn IDL and to use the special automatically generated files called stubs and skeletons. Caffeine compiles Java interface directly into IDL. Any object that implements that Java interface may be used as a distributed object, without requiring a ***stub*** or ***skeleton***. The CORBA ***stubs*** and ***skeletons*** still exist, but they are hidden from the developer. Converting Java interface into a CORBA interface is quite difficult. CORBA has a limited set of data types, and Java has an extensible set of types. Mapping is accomplished through Object Serialization. The state of a native Java object can be communicated between different Java applications developed with Caffeine. In Caffeine, the stub is responsible for providing the mechanism for moving native Java objects across the network. Skeleton is responsible for reading native Java objects, transferred as IN parameters in a remote invocation, and for writing those Java objects which are used as OUT parameters in a CORBA system.

Formally, mobile object cannot be developed in CORBA (OMG does not have a specification for a mobile object). Never the way, Caffeine vendors implemented mobile object technology in their product.

6.1 Mobility features and security

The Caffeine has a feature called pass-by-value. Pass-by-vale is a common feature in distributed system in which the values for an object move from one place to another. The Caffeine also includes implementation mobility. Thus, the Caffeine has basic mobile object tool features, but does not support moving threads, customized class loading and other.

The Caffeine is compatible with SSL protocol for protecting TCP/IP connection from prying eyes and unauthorized modifications. No other security subject is addressed by this tool.

7. Voyager

Voyager is an ORB developed by ObjectSpace. Its team merged distributed object and agent technology. Voyager agents can move between different applications, in opposite to distributed objects, which are restricted to the application in which they were created. Voyager enables agents to be constructed in remote processes. It also enables a host of other features for supporting mobile agents, including security and persistency engine.

Voyager has a notion of ***place process***. It provides home for agents moving from host to host. This place process is called ***voyager*** in the Voyager environment. Voyager objects, which are remotely constructed, live within the voyager place process. Its remote reference is called ***proxy***.

7.1 Mobility features

Mobility implementation in Voyager can take several different forms. First, object can be treated as an agent. For Voyager agent is an object that can move itself around the network. Other applications can interact with this agent object through the agent's **proxy**, which can be listed in **Namespace**. Making an object into an agent in Voyager is a meter of accessing the ***Agent facet*** of an object. Voyager supports a notion of **facets** for objects. **Facets** are value-added interfaces that Voyager supports for providing additional functionality to objects. Each **facet** represents a particular type of functionality that can be used to manipulate objects in Voyager. By invoking the **Agent facet** of an object, an agent moves, or can be moved, across the network. The moving of the **Agent** object is accomplished through Object Serialization. The **Agent** is actually cloned in a destination process and a **forwarder** is left to redirect any future requests to the new host of the object. The original instance of the object is destroyed. All requests for **Agent** received during **Agent** serialization should be temporarily queued.

Second, Voyager objects can be explicitly moved to a new location. In order to move an object to a new destination explicitly, **moveTo()** method on the ***Mobility facet*** must be invoked. The mechanism is much the same as for moving **Agent** object. Neither **Agents**, nor explicitly moved object ever become a true part of the host application and they are accessed through their **proxy**.

Third, objects, which are part of the remote invocation are serialized and copied out to the remote host as a part of remote invocations. A remote virtual machine

reads the serialized object, the data is deserialized and converted back into an instance of the object. This way of moving object provides the highest level of integration between mobile object and its host application. The objects moved in this way, will not move to another location on their own. References to them are normal Java object references to a local instance of the object that lives within the application.

All these mobile objects, whether agents or simple serializable objects, must **java.io.Serializable** interface whether directly or through indirectly through the **Agent** class.

Voyager uses a set of resource class loaders to load classes and other resources from URLs or other resource sources. An application can register its own resource loader.

7.2 Security

Voyager protects hosts from malicious objects using normal Java sandbox security model [1]. It is implemented in ***VoyagerSecurityManager***, which is not used by default. By invoking operations on the **VoyagerSecurityManager**, the Java Virtual Machine can check to see if particular object has rights to perform a specific operation.

7.3 Object identity

In the **Agent** and **Mobility** approach in Voyager, Agent objects can be accessed only through a virtual reference. Direct Java references to the underlying instance of the Java object are not allowed.

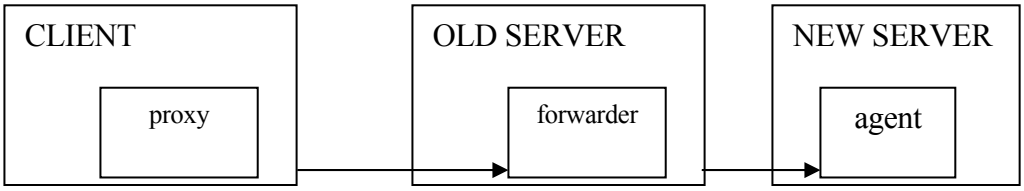

Figure 4: Voyager solution for object identity

The virtual reference can be updated with the new position of the object, after its migration, see Fig. 4. This capability avoids confusion that might be associated with cloning the object to a new virtual machine, as is usually the case with mobile object tools. Any virtual references pointing back at the old instance of the object may be forwarded on a new location, because Voyager provides a ***Forwarder*** object to accomplish this task.

The "ordinary" serializable object cloned across the network causes any local Java object references (in the local virtual machine that point to the serializable

object before it is cloned) still point back to the same local instance. It is up to the developer never to use the old instance and to redirect all future requests to the new instance in the remote virtual machine.

7.4 Concurrency

Voyager provides a straightforward thread-pooling facility. Java threads are facades over operating system threads. Voyager applications have a thread pool that is used to recycle threads for later use in order to prevent the operating system overhead normally incurred when threads are repeatedly created and destroyed.

7.5 Object sharing

Most of the Internet communication is on a point-to-point basis – a single client to a single server [1]. A distributed system does not fit well with this approach. Typical situation in distributed system is one server and many clients. The host's performance of sending 250000 copy of information to the clients is very poor, unless it uses some tool for broadcasting information. IP Multicast needs only one copy of the data to be sent from the host.

A Voyager Space is an IP Multicast-based communication mechanism that can send data to a group of multiple recipients. Objects are added to Space by invoking the add operation on Space. When a Space issues a multicast invocation on each participant, the arguments passed in the invocation are serialized and delivered to each participant. The Space is just any old Voyager place process.

8. RMI – Remote Method Invocation

RMI is product of JavaSoft. It is an ORB built into JDK 1.1 (*Java Development Kit version 1.1*) [1]. Two Java applications or applets can invoke each other's objects through RMI. RMI enables Java developer to declare the methods that are available for remote invocation using a normal Java interface rather than a separate language like IDL. RMI also has a notion of stub and skeleton. Their role is the same as in the other ORBs. RMI's requirements for interfaces that are used with distributed objects are: all interfaces to distributed objects must extend ***java.rmi.Remote*** and all operations on such interfaces must throw ***java.rmi.RemoteException***.

9. COM/DCOM

COM components integrate well with many development environments. The goal of COM is to enable the quick addition of components into new application.

Java and COM/DCOM are products of the opponent vendors, and have totally different philosophy. Microsoft virtual machine implements the Java standard to a point. Microsoft does not implement RMI or CORBA because it perceives technologies as competitive with DCOM: There is a way to use them together, but then Java software looses the benefits of "write once, run anywhere" principle and can be run only on the Windows operating systems. On the other hand, COM enables the quick addition of components into new applications.

COM and Java integration goes as follows: every COM components is mapped into a generic Java class called ***ActiveXControl***. This class is a handle to any COM component.

9.1 COM and Java

Microsoft has made COM very easy to use within Java programs. Creating a COM component within a Java program requires nothing but a constructor-like invocation. COM and Java integration has certain limitations, such as: COM components cannot accept Java objects as arguments. COM supports only limited set of types, and Java objects are not included in that set. COM's integration with Windows operating system is also a limitation: when a COM component is used within a Java program, the Microsoft virtual machine must use the Windows ***Registery*** to determine the location of the COM component.

Developing a Java object compatible with COM could not be easier: Microsoft supplies a tool called ***javareg.exe*** that can register any java object as a COM component.

9.2 DCOM

DCOM introduces distributed computing to COM. Unlike other distributed computer tools, which are responsible for communicating among applications that make up distributed system, activating new distributed objects on demand, and providing security to the distributed objects, DCOM leaves everything up to the operating system. DCOM uses the information compiled into the COM component to determine its interfaces at run time., and because of that stubs and skeletons are not always required.

Some system configuration is needed to use COM components with DCOM. The tool used for this operation is ***DCOMConf.exe***. For each COM component on the system, **DCOMConf.exe** controls when and how the component will be created. In DCOM components are referred to through a pointer to a special COM interface called ***IDispatch*** interface. The pointer to **IDispatch** interface may be sent to other computers. The value of the pointer itself is not sent because of the different physical memory address spaces involved in a distributed system.

10. Conclusion

Seen so far, the most complete tool for mobile objects, agents and components is Voyager. Microsoft has recently offered the .NET tool, the ancestor of the COM/DCOM. It has not yet been evaluated. There are a lot of experimental mobile agent tools, developed in university laboratories, as well as Java packages, which should help developers to build robust distributed software. Java's built in mechanism for serialization is of the greatest importance.

11. References

1. Jubin H., Friedrichs J.(2000), Enterprise Java Beans by Example, Prentice Hall
2. Java 2 Platform SE v1.3, www.java.sun.com
3. Nelson J. (1999), Programming Mobile Objects with Java, Wiley Computer Publishing
4. RFC, Network Working Group
5. Slama D., Garbis J., Russel P., (1999), Enterprise CORBA, Prentice Hall