An architectural view of distributed objects and components in CORBA, Java RMI and COM/DCOM

František Plášil1, Michael Stal2

1 Charles University, Faculty of Mathematics and Physics, Department of Software Engineering, Malostranské nám. 25, 118 00 Prague 1, Czech Republic; E-mail: plasil@nenya.ms.mff.cuni.cz, http://nenya.ms.mff.cuni.cz
2 Siemens AG, Corporate Technology, Dept. ZT SE 1, Wittelsbacherplatz 2, D-80312 Munich, Germany; E-mail: michael.stal@mchp.siemens.de

Abstract. The goal of this paper is to provide an architectural analysis of the existing distributed object-oriented platforms. Based on a relatively small number of design patterns, our analysis aims at a unified view of the platforms. We achieve this by articulating a series of key issues to be addressed in analyzing a particular platform. This approach is applied systematically to the CORBA, Java RMI and COM/DCOM platforms.

Key words: CORBA – Java RMI – COM/DCOM – Distributed computing – Distributed objects – Design patterns – Software architecture

1 Introduction

With the widespread utilization of object technology, it has become more and more important to employ the object-oriented paradigm in distributed environments as well. This raises several inherent issues, such as references spanning address spaces, the need to bridge heterogeneous architectures, etc. It is the main goal of this paper to provide an architectural analysis of current software platforms in this area. One of the obstacles to overcome in order to achieve this aim is the fact that the available descriptions of these platforms speak different languages. Thus to target the issue, we have decided to employ design patterns [3, 19] as a common denominator to help us provide a unified view on the platforms analyzed.

We focus on the following key distributed object platforms: CORBA, Java RMI and COM/DCOM. CORBA is specified by OMG [12], which is the largest consortium in the software industry. CORBA has undergone an evolution from CORBA 1.0 (1991) and CORBA 2.0 (1995) to CORBA 3.0, which is soon to be released. The Java environment, designed by Sun Microsystems, has probably experienced the greatest evolution recently. From the broad spectrum of the Java platform segments, we will focus on Java RMI [22], which targets working with distributed objects. The last platform analyzed is the Microsoft Component Object Model (COM). This platform has also been evolving gradually along the milestones OLE, COM, DCOM and COM+ [18]. In this paper, we will focus on COM/DCOM [6], as this is where Microsoft targets distributed objects.

The paper is structured as follows: Section 2 articulates the general principles of working with distributed objects. The division of the section reflects our approach to architectural analysis: basic principles, basic patterns, provision and employment of a service, and inherent issues. Using the same structuring as in Sect. 2, we offer analyses of CORBA (Sect. 3), Java RMI (Sect. 4) and COM/DCOM (Sect. 5).

Due to the limited size of the paper, we could not focus on security questions or benefits of code mobility over the Internet. Also, a thorough evaluation of each platform could not be provided. All of these areas have become very broad and each of them deserves at least a separate paper.

2 Distributed objects

2.1 Basic principles

2.1.1 Request and response

By distributed objects we usually mean objects that reside in separate address spaces and whose methods can be subject to remote method calls (a remote method call is issued in an address space separate from the address space where the target object resides). By convention, the code issuing the call is referred to as the client; the target object is referred to as the server object (or simply remote object);
the set of methods which implements one of the server object’s interfaces is sometimes designated as a service that this object provides. Similarly, the process in which the server object is located is referred to as a server.

An important goal of the client and server abstractions is to make it transparent how “far” the client and server spaces actually are: whether they reside on the same machine, are on different nodes of a local network, or even reside on different Internet URLs (thus being “intergalactic”); as a special case, the client and server may share the same address space.

Because it is inherently delivered over a network communication infrastructure, a remote method call is typically divided into the request (asking the service) and response (bringing results back to the client) parts. In principle, from the client’s view, the request and the response corresponding to a remote method call can be handled as one atomic action (synchronous call), or they can be separated, where the client issues the request and then, as a future action, issues a wait for the response (deferred-synchronous call). Sometimes the response part may be empty (no out parameters and no functional value). In this case, the corresponding method is usually termed a one-way method. A one-way method can be called asynchronously, where the client does not have to wait until the call is finished. In a distributed environment, the exactly-once semantics of remote calls is practically impossible to achieve; real distributed platforms ensure the at-most-once semantics of a synchronous and deferred-synchronous call (exactly-once semantics in case of a successful call, at-most-once semantics otherwise); best-effort semantics is ensured for a one-way method.

2.1.2 Remote reference

One of the key issues of remote method calls is referencing of remote objects. Classically, in a “local” case, in a method call \( \text{rro.m}(p_1, p_2, \ldots, p_n) \), \( \text{rro} \) contains a reference to the target object, \( m \) identifies the method called, and some of the parameters can contain references to other objects as well; let us suppose only one of the parameters, say \( p_j \), contains a reference. However, in a distributed environment we face the following issue: \( \text{rro} \) should identify a remote object over the network, and so should \( p_j \). It is obvious that classical addresses will not do as the references, at least for the following reasons: in addition to the data record of the target object, a reference has to identify also the node and the server in which the target object resides. Moreover, the target object may implement more (non-polymorphic) interfaces; thus, \( \text{rro} \) should also identify the particular interface which the target object implements and to which \( m \) belongs. By convention, a reference that contains all this information is termed a remote reference. Hence a remote reference identifies a service. In addition, representation of a remote reference has to span the differences in the hardware architectures of the nodes where the objects involved in a particular remote method call reside.

2.1.3 IDL interface

In principle, a client’s code and the server object that is subject to a remote call from the client can be implemented in different languages and can run on heterogeneous architectures. To span this kind of difference, the interfaces of a server object are specified in an architecture-neutral Interface Definition Language (IDL). Typically, IDL provides constructs for specification of types, interfaces, modules, and (in some cases) object states. However, there is no means for specifying the code of methods. Usually a mapping from IDL to standard programming languages, such as C++ and Java, is part of an IDL definition. CORBA IDL and Microsoft MIDL are examples of IDL languages.

2.1.4 Proxy: local representative

To bridge the conceptual gap between the remote and local style of references, both in the client and server code, the actual manipulation with remote references is typically encapsulated in wrapper-like objects known as client-side and server-side proxies. The client-side proxy and the corresponding server-side proxy communicate with each other to transmit requests and responses. Basically, the client-side proxy supports the same interface as the remote object does. The key idea behind proxies [17] is that the client calls a method \( m \) of the client-side proxy to achieve the effect of calling \( m \) of the remote object. Thus, the client-side proxy can be considered a local representative of the corresponding remote object. Similarly, the key task of a server-side proxy is to delegate and transform an incoming request into a local call form and to transform the result of the call to a form suitable for transmitting to the client-side proxy. Thus a server-side proxy can be considered as the representative of all potential clients of the remote object.

2.1.5 Marshalling: transmitting request and response

Both the request and response of a call are to be converted into a form suitable for transmitting over the network communication infrastructure (a message, or a TCP socket connection for transmitting streams might be an example of the infrastructure). Typically, serialization into a byte stream is the technical basis of such conversion. By convention, this conversion is referred to as marshalling (the reverse process is unmarshalling). Some authors use these concepts in a narrower sense, where marshalling/unmarshalling refers only to conversions of the parameters of a remote call.

The key issue of marshalling and unmarshalling is dealing with objects as parameters. The following