Design And Implementation Distributed System Using Java-RMI Middleware

[1]Amira B. Sallow

[1] Department of Computer Science, College of Computer & I.T, Nawroz University, Iraq

ABSTRACT

Networks of computers are everywhere. The Internet is one, as are the many networks of which it is composed. Mobile phone networks, corporate networks, factory networks, campus networks, home networks, in-car networks, both separately and in combination, all share the essential characteristics that make them relevant subjects for study under the heading distributed systems. Most organizations use a wide variety of applications for the smooth functioning of their businesses that includes homogenous as well as heterogeneous systems. Heterogeneous systems run on different platforms, use different technologies or sometimes even run on a different network architecture altogether. The essential role of Middleware is to provide a simple environment to manage complex, heterogeneous and distributed infrastructures.

The main goal of this paper is to use Java-RMI middleware to build a distributed system for scheduling the threads. The system comprises two separate programs, a server, and a client. A typical server program creates some remote objects, makes references to these objects accessible, and waits for clients to invoke methods on these objects. A typical client program obtains a remote reference to one or more remote objects on a server and then invokes methods on them.

Keywords: Distributed system, Java-RMI, Middleware, Heterogeneous systems.

1. Introduction

The evolution of Internet-based computing from local area networks (LANs), after transitioning from unconnected computers to networks, is the hallmark of all business models today. The technological backbone of this evolution is the middleware. First connecting, then communicating, and finally seamlessly integrating the distributed systems to external sites, customers, suppliers, and trading partners across the world is the real challenge for the business world. It doesn't stop there. Also required is the talking between client and server over heterogeneous networks, systems architectures, databases, and other operating environments. All this is facilitated by the middleware technologies that offer undercover functions to seamlessly integrate various applications with information instantly to make it accessible across diverse architectures, protocols, and networks. Automation of back-end and front-end operations of the business is also affected by the middleware. [1]

the rest of this article is organized as follows. Section 2, related works. Section 3, mentions middleware. Section 4, explains positioning middleware in detail, Section 5 explains categories of middleware. Section 6 explains RMI and Java. Section 7 illustrates distributed RMI system design. Section 8 presents the conclusion.

2. Related Works

Weonjoon, Hyoungyuk and Park in [2] in 2001 presented object models and service models such without lots of considerations about data transmissions between as an I/O object, a control object, a broadcasting service and an event service for a

Academic Journal of Nawroz University
(AJNU) Volume 9, No 1 (2020).
Regular research paper : Published 19 Feb 2020
Corresponding author's e-mail : amira.bibo@nawroz.edu.krd

doi : 10.25007/ajnu.v9n1a550

distributed control system (DCS), and three types of distributed control systems based on CORBA, DCOM, and Tspace, are designed and implemented using the suggested object models and service models.

Yves, Frederic and Luc [3] in 2004 reported the development of the Concerto platform, which is dedicated to supporting the deployment of resource-aware parallel Java components on heterogeneous distributed platforms, such as pools of workstations in labs or offices. Their work aimed at proposing a basic model of a parallel Java component, together with mechanisms and tools for managing the deployment of such a component on a distributed platform. The Concerto platform was designed in order to allow the deployment of parallel components on a distributed platform.

Christopher et al. [4] in 2013 the use of Java RMI on mobile devices for peer-to-peer computing is presented. Detailed design and implementation of the artifact for peer-to-peer network using Java 2 platform programming language were carried out and java distributed programs have been developed with the same semantics and syntax used for non-distributed programs through mapping of java classes and objects to work in a distributed (multiple JVM) computing environment.

Andre et al. [5] in 2018 presented Java Ca&La (JCL), a distributed-shared-memory and task-oriented lightweight middleware for the Java community that separates business logic from distribution issues during the development process and incorporates several requirements that were presented separately in the GPDC middleware literature over the last few decades. JCL allows building distributed or parallel applications with only a few portable API calls, thus reducing the integration problems. Finally, it also runs on different platforms, including small single-board computers.

3. Middleware

The term middleware applies to a software layer that provides a programming abstraction as well as masking the heterogeneity of the underlying networks, hardware, operating systems and programming languages. The Common Object Request Broker Architecture (CORBA), And Java Remote Method Invocation (RMI) are middleware examples. Most middleware is implemented over the Internet protocols, which themselves mask the differences of the underlying networks, but all middleware deals with the differences in operating systems and hardware [6].

Middleware binds discrete applications, such as Web-based applications and older mainframe-based systems, to allow companies to hook up with the latest systems and developments that drive new applications without making their investments in legacy systems unyielding.[1]

4. Positioning Middleware

Many distributed applications make direct use of the programming interface offered by network operating systems. For example, communication is often expressed through operations on sockets, which allow processes on different machines to pass each other messages. Also, applications often make use of interfaces to the local file system. The problem with this approach is that distribution is hardly transparent. A solution is to place an additional layer of software between applications and the network operating system, offering a higher level of abstraction. Such a layer is accordingly called middleware. It sits in the middle between applications and the network operating system as shown in Figure (1). [6].

In addition to solving the problems of heterogeneity, middleware provides a uniform computational model for use by the programmers of servers and distributed applications. Possible models include remote object

doi : 10.25007/ajnu.v9n1a550

invocation, remote event notification, remote SQL access, and distributed transaction processing. For example, CORBA provides remote object invocation, which allows an object in a program running on one computer to invoke a method of an object in a program running on another computer. Its implementation hides the fact that messages are passed over a network to send the invocation request and its reply [6].

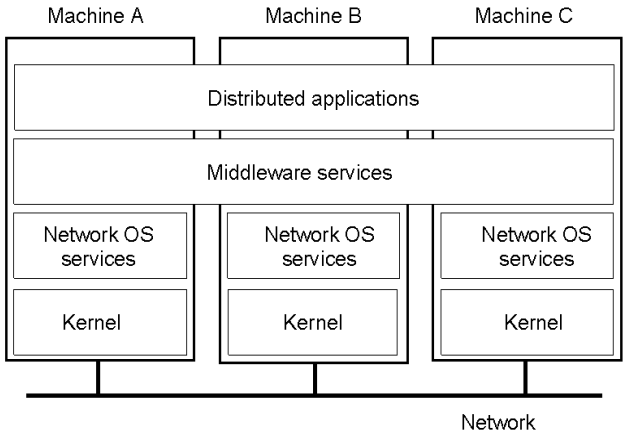

Figure 1. The general structure of a distributed system as middleware [6].

5. Categories of Middleware

There are a small number of different kinds of middleware that have been developed. These vary in terms of the programming abstractions they provide and the kinds of heterogeneity they provide beyond network and hardware [7].

5.1 Distributed Tuples

A distributed relational database offers the abstraction of distributed tuples, and are the most widely deployed kind of middleware today. Its Structured Query Language (SQL) allows programmers to manipulate sets of these tuples (a database) in an English-like language yet with intuitive semantics and rigorous mathematical foundations based on set theory and predicate calculus.

Distributed relational databases also offer the abstraction of a transaction. Distributed relational database products typically offer heterogeneity across programming languages, but most do not offer much if any, heterogeneity across vendor implementations. Transaction Processing Monitors (TPMs) are commonly used for end-to-end resource management of client queries, especially server-side process management and managing multi-database transactions [7].

Linda is a framework offering a distributed tuple abstraction called Tuple Space (TS). Linda's API provides associative access to TS, but without any relational semantics. Linda offers spatial decoupling by allowing depositing and withdrawing processes to be unaware of each other's identities. It offers temporal decoupling by allowing them to have non-overlapping lifetimes [7].

Jini is a Java framework for intelligent devices, especially in the home. Jini is built on top of JavaSpaces, which is very closely related to Linda's TS [8].

5.2 Remote Procedure Call

Remote procedure call middleware extends the procedure call interface familiar to virtually all programmers to offer the abstraction of being able to invoke a procedure whose body is across a network. RPC systems are usually synchronous, and thus offer no potential for parallelism without using multiple threads, and they typically have limited exception handling facilities [7][9].

5.3 Message-Oriented Middleware

Message-Oriented Middleware (MOM) provides the abstraction of a message queue that can be accessed across a network. It is a generalization of the well-known operating system that construct the mailbox. It is very flexible in how it can be configured with the topology of programs that deposit and withdraw messages from a given queue. Many MOM products offer queues with persistence, replication, or real-time performance. MOM offers the same kind of spatial and

doi : 10.25007/ajnu.v9n1a550

temporal decoupling that Linda does [7].

5.4 Distributed Object Middleware

Distributed object middleware provides the abstraction of an object that is remote yet whose methods can be invoked just like those of an object in the same address space as the caller. Distributed objects make all the software engineering benefits of object-oriented techniques: encapsulation, inheritance, and polymorphism [10].

The Common Object Request Broker Architecture (***CORBA***) is a standard for distributed object computing. It is part of the Object Management Architecture (OMA), developed by the Object Management Group (OMG), and is the broadest distributed object middleware available in terms of scope. CORBA offers heterogeneity across programming language and vendor implementations. CORBA (and the OMA) is considered by most experts to be the most advanced kind of middleware commercially available and the most faithful to classical object-oriented programming principles. Its standards are publicly available and well defined. ***DCOM*** is a distributed object technology from Microsoft that evolved from its Object Linking and Embedding (OLE) and Component Object Model (COM). DCOM provides heterogeneity across language but not across the operating system or tool vendor. ***COM+*** is the next-generation DCOM that greatly simplifies the programming of DCOM. Java has a facility called Remote Method Invocation (***RMI)*** that is similar to the distributed object abstraction of CORBA and DCOM. RMI provides heterogeneity across the operating system and Java vendor [10][11].

6. RMI and Java

The main components of RMI can be considered as follows:

- **The programming model,** where objects that can be accessed remotely are identified via a remote interface,
- **The implementation model,** which provides the transport mechanisms whereby one Java platform can talk to another to request access to its objects, and
- **The development tools** (e.g., rmic or its dynamic counterpart), which take server objects and generate the proxies required to facilitate the communication.

The key to developing RMI-based systems is defining the interfaces to remote objects. RMI requires that all objects that are to provide a remote interface must indicate so by extending the pre-defined interface Remote. Each method defined in an interface extending Remote must declare that it "throws RemoteException". Thus, one of the key design decisions of RMI is that distribution is not completely transparent to the programmer. The location of the remote objects may be transparent, but the fact that remote access may occur is not transparent [12].

7. Distributed RMI System design

Java-RMI is a Java-specific middleware that allows client Java programs to invoke server Java objects as if they were local. The Java Remote Method Invocation is a java API to perform the object-oriented equivalent of remote procedure calls. RMI applications often comprise two separate programs, a server, and a client. A typical server program creates some remote objects, makes references to these objects accessible, and waits for clients to invoke methods on these objects. A typical client program obtains a remote reference to one or more remote objects on a server and then invokes methods on them. RMI provides the mechanism by which the server and the client communicate and pass information back and forth. Such an application is sometimes referred to as a distributed object

doi : 10.25007/ajnu.v9n1a550

application figure (2). An RMI system must be composed of the following parts:

- An interface definition of the remote services;
- The implementations of the remote services;
- Stub and skeleton files;
- A server to host the remote services;
- An RMI Naming service
- A client program that uses remote services.

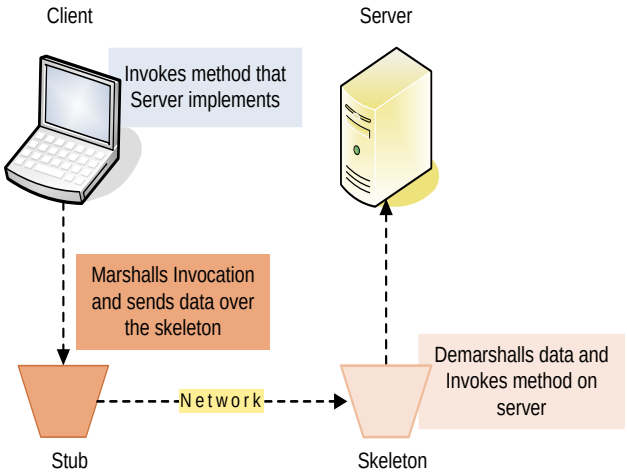

figure 2: A basic RMI call with a stub and skeleton

The RMI Distributed system was designed through several design steps Figure (3) summarize RMI system design steps, and they are:

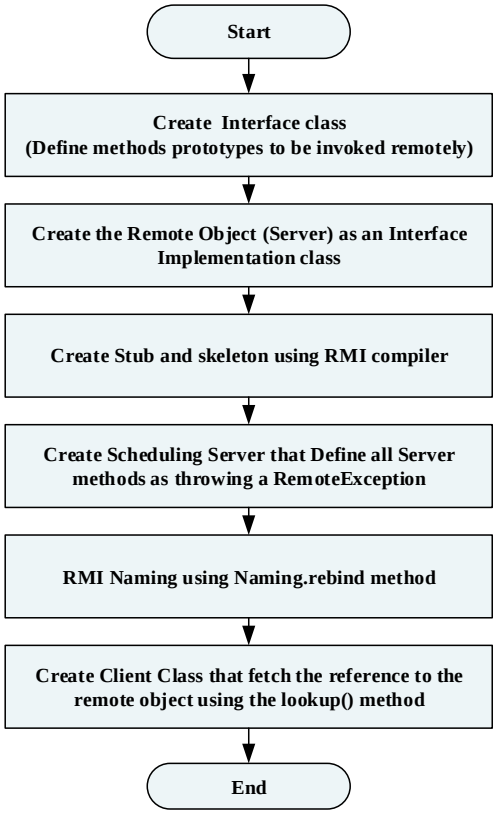

figure 3: RMI design steps.

Step One: Create Interface class

This interface will define the methods that will be available in a class for implementation. The interface is Scheduler.java, It contains only methods for scheduling the threads. The method throws a java.rmi.RemoteException (this exception is a superclass of all RMI Exception that can be thrown) as in Figure (4).

```
import java.rmi.*;

public interface Scheduler extends Remote
{
public int[] Schedule(int[] x,int[] y)throws RemoteException;
}
```

figure 4: Scheduler Interface java class.

Step Two: Create Remote Object (Server) that implements Scheduler interface:

The implementation is found in SchedulerRemote.java class. This class must extend

doi : 10.25007/ajnu.v9n1a550

java.rmi.UnicastRemoteObject and implement the schedule interface that has been created in step one as in Figure (5).

```
import java.rmi.server.*;
import java.math.*;

public class SchedulerRemote extends UnicastRemoteObject
implements Scheduler
{
SchedulerRemote()throws RemoteException
{   super();   }

public int[] Schedule(int[] x,int[] y)
{ ……
……}
}
```

figure 5: SchedulerRemote.java interface implementation java class.

Step Three: Create Stub and skeleton files

After writing and compiling the servers, there is a need to generate stubs and skeletons. It simply needs to invoke the RMI compiler (rmic). Here, for example, we use rmic instruction to generate stubs and skeletons for the Distributed system figure (6):

rmic SchedulerRemote: This instruction will generate two files:

- SchedulerRemote_Skel.Class
- SchedulerRemote_Stub.Class

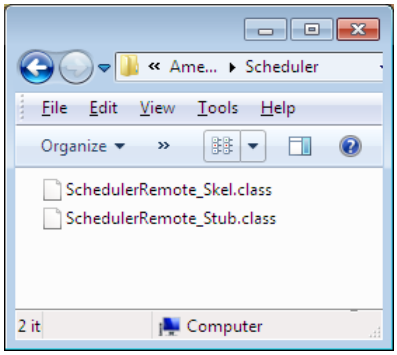

Figure 6: Stub and skeleton files

rmic works by generating Java source code for the stubs and skeletons and then compiling those Java files. The keep flag simply tells rmic to save the source code to java files, rmic takes a class file and creates a pair of companion files, a stub, and a skeleton, in the same package as the .class file. There's an interesting subtlety here rmic requires the actual implementation's class files. It seems, at first glance, that the compiled interface files might suffice. However, the designers of RMI decided that the stubs and skeletons should satisfy the following two requirements:

- There should be a unique stub/skeleton pair per server, so we can do things such as register the server in the naming service.
- The stubs and skeletons should implement all the remote interfaces that the server does so that casting when you retrieve objects from the naming service is a local operation.

Step Four: Create a server

The server class is "SchedulerServer.java". It is responsible for two things:

a) Installs a RMISecurityManager.
b) Creates an instance of " SchedulerRemote" class, and give it the name "SchedulerRemoteInstance".

Here "SchedulerRemoteInstance" object takes care of registering the object with the RMI Registry after this server run, the object will be available to the remote client this done using the Naming.rebind() method.

```
if (System.getSecurityManager() == null)
System.setSecurityManager(new RMISecurityManager() );
try {
Scheduler s=new SchedulerRemote();
// export server MyServer to RMI Registry at port NNNN
//Naming.rebind("//Hostname:NNNN/MyServer",server);
Naming.rebind("rmi://localhost:5000/MyServer",s);
}
catch(Exception e)
{System.out.println(e);}
```

figure 7: SchedulerServer java class.

Step Five: RMI Naming service

In RMI, the default naming service that ships with Sun

doi : 10.25007/ajnu.v9n1a550

Microsystem's version of the JDK is called the RMI registry as shown in figure (8). Messages are sent to the registry via static methods that are defined in the java.rmi.Naming class.

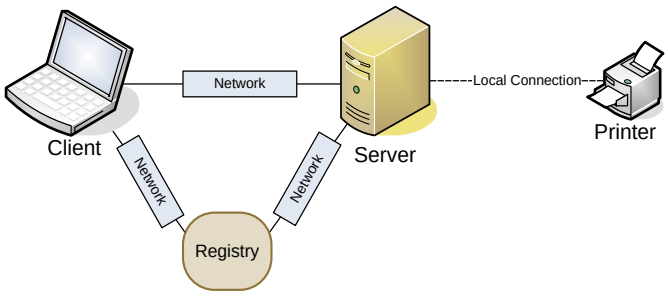

figure 8: RMI registry architecture diagram

Servers are bound into the registry using a string. More generally, names are formatted according to the following pattern of these three pieces of information, host-name and port-number describe the registry while human-readable-name is server-specific.

//host-name: port-number/human-readable-name

That is, host-name and port-number are used to find a running instance of the RMI registry (which should be listening on port-number of host-name). Human-readable-name, on the other hand, is the name used internally, by the registry, to identify the server being registered. Both host-name and port-number have default values. host-name defaults to "localhost" and port-number defaults to 1099.

Step six: Create a Client Class

The client is " SchedulerClient.java". The client first installs a new RMISecurityManager, then uses the static method Naming.lookup() to get the reference to the remote object. Note that the client is using the schedule interface to hold the reference and make method calls creating an interface which must be done before trying to build the client, or a "class is not found" occur.

8. Conclusion

In this paper, a Java-RMI middleware has used to build a distributed system that comprises two separate programs, a server, and a client. A typical server program creates some remote objects, makes references to these objects accessible, and waits for clients to invoke methods on these objects. A typical client program obtains a remote reference to one or more remote objects on a server and then invokes methods on them. The following conclusions can be marked

- Building the system with an RMI middleware minimizing the complexity of distributed programming.
- Building the system with an RMI middleware introduced A high degree of transparency.
- RMI middleware is easy to develop and use.

9. References

1. Qiyang Chen, James Yao, and Rubin Xing, 2006," Middleware Components for E-commerce Infrastructure: An Analytical Review", Department of Management and Information Systems Montclair State University, Upper Montclair, NJ, USA.
2. Weonjoon Kang, Hyoungyuk Kim and Hong Seong Park, "Design and performance analysis of middleware-based distributed control systems", ETFA 2001. 8th International Conference on Emerging Technologies and Factory Automation.
3. Y. Maheo, F. Guidec and L. Courtrai, "Middleware support for the deployment of resource-aware parallel Java components on heterogeneous distributed platforms", Proceedings. 30th Euromicro Conference, 2004., 2004.
4. Christopher I. Eke et al.," Use of Java RMI on Mobile Devices for Peer to Peer Computing", Department of Computer Science, Federal University, Lafia, Nigeria, iiste, vol. 3, no. 5, 2013.
5. [6]A. Almeida et al., "A general-purpose distributed computing Java middleware", Concurrency and Computation: Practice and Experience, vol. 31, no. 7, p. e4967, 2018.
6. Andrew S.Tanenbaum, Maarten Van Steen, 2002,

doi : 10.25007/ajnu.v9n1a550

"Distributed Systems principles and diagram", First Edition, Netherlands, Pearson Education, Inc., publishing as Addison-Wesley.

7. David E. Bakken, 2001, " Middleware", School of Electrical Engineering and Computer Science, Washington State University, Pullman, USA.
8. Bill McCarty , Luke Cassady-Dorion, 1999, "Java Distributed Objects" Sams, USA.
9. Richard E. Schantz , Douglas C. Schmidt,2001,"Middleware for Distributed Systems Evolving the Common Structure for Network-centric Applications", Electrical Engineering & Computer Science Dept, Cambridge University, USA.
10. Richard E. Schantz , Douglas C. Schmidt,2003,"Middleware for Distributed Systems" , Electrical Engineering & Computer Science Dept, Cambridge University, USA.
11. Johann Schlichter,2002,"Distributed Applications", Institut für Informatik TU München, Munich, Germany.
12. D. Abdullah and A. Sallow, "EOE-DRTSA: End-to-End Distributed Real-time System Scheduling Algorithm", *IJCSI International Journal of Computer Science Issues*, vol. 10, no. 2, pp. 407-413, 2013.

doi : 10.25007/ajnu.v9n1a550