Prolog goes Middleware: Java-based Embedding of Logic Servers

By Thomas Steiner

Introduction

Knowledge bases and logic engines have existed almost from the beginningof enterprise computing, and many developers have classified declarative languages - such as Prolog - as too scientific and not appropriate for modern environments, i.e. the Internet. The proponents of Prolog - I count myself as one - try to promote the value of logic bases to Internet programmers. Sometimes, our euphoric descriptions of Interactive Declarative Environments (IDEs) are heard and gain attention, but usually we earn a smile from Internet programmers. This situation has radically changed with the arrival of logic *servers* and the possibility to *embed* and *distribute* knowledge bases.

Early attempts to combine logic and the Internet focused on holding logic programs between meta-tags within HTML pages. More recent approaches promote the use and integration of logic components. Following this trend, a wrapper is typically built around traditional Prolog engines, allowing easy integration and distribution.

One way of combining knowledge bases with Internet technologies is to access knowledge bases via the WWW [1]. Such approaches use a gateway to generate queries and converted the results to HTML before it delivered. Some years ago, a server-side logic base, accessed through a query and implemented via a CGI-script, converted the results to HTML before it delivered them back to the client. We call this centralized approach the *integrated logic base*. Hence, other authors argue against this centralized conception of "intelligence". By observing a particular individual, most opponents to the centralized approach conclude that knowledge may be duplicated in many places, and may not be uniformly accessible.

An alternative - and still less deployed - method to implement WWW-logic bases consists of distributing the logic base itself. Such distributed logic bases reside within the client environments (such as an HTML page running in a Web Browser). This *distributed logic base* inspired by the ability of *collective information gathering* approaches and implements the *downsizing and distribution* of knowledge. This reduces, when combined with a pertinent *source and domain model*, the information overload.

We used Amzi's Logic Server 5.0 Prolog engine to integrate and distribute a (simple) logic base within the two dominant middleware architectures : RMI and CORBA.

"Good morning!" - A Sample Application

A very rudimentary Prolog program illustrates the embedding of the Prolog engine within the two middleware architectures. Consider the following code:

```
period(morning) :- time(Hour,Min,Sec,Hun),
    Hour < 12.
period(afternoon) :- time(Hour,Min,Sec,Hun),
    Hour >= 12,
    Hour =< 18.
period(evening) :- time(Hour,Min,Sec,Hun),
    Hour > 18.
```

The period(X) predicates check the period of the day and return a "Good morning!", "Good afternoon!" or "Good evening!" respectively. The time/4 predicate is built-in. Compiling and linking this code generates the .XPL file loaded by the Logic Server in the subsequent examples.

Extending Real World Objects with a Prolog Engine

Recently, more Prolog IDEs have begun to expose their logic engine through an Application Programming Interface (API). The advantage for the programmer is a simplified embedding of the prolog engine within traditional development tools (such as Delphi, C++, VB, Java etc.) We will concentrate on the *JDK 1.3* interface.

The Logic Server is available as a *Java class*. The main difference with Delphi and C++ is that in Java, *function pointers* are no longer required. As this feature is basically used to write extended predicates and to load them into the Prolog engine, we will not deal with those issues here.

A *Java application or applet* gains access to the Logic Server by importing the Amzi package

```
import amzi.ls.*;
```

Access to the Prolog functionality is then very easy. First, an instance of the Logic Server is created

```
LogicServer ls = new LogicServer();
```

and then, all the methods for effective Java-Prolog interoperation become available, such as:

```
ls.Init("Period");
ls.Load("Period");
long term = ls.ExecStr("period(X)");
System.out.println("Good
"+ls.GetStrArg(term,1)+"!");
ls.Close();
```

The code segment uses the .XPL file "Period" - a compiled prolog program. The results of running this application in a *terminal* are shown in figure 1.

```
java PeriodOfDay
Initializing the Logic Server.
Amzi! Prolog + Logic Server, Personal
Edition
Non-Commercial Runtime

Good afternoon!
```

Figure 1: The Amzi Logic Server Running in a Java Application (at 5:17 p.m.)

The full source codes of all samples in this article can be found on our Website (What is URL). When recompiling the programs, the *amzi.ls.* package* must be in the CLASSPATH. An easy way to avoid this issue is to copy the complete amzi subdirectory under the path that actually contains your .java source files.

Embedding Prolog in CORBA

The *Common Object Request Broker Architecture* (CORBA) is a complete abstraction over low-level network services. CORBA is based on broker agents (the ORBs), which provide a rich set of services (see figure 2):

We now concentrate on the CORBA *Naming service*, and we will leave aside all the other services, although some are of importance to distributed (Prolog) programmers - such as the distributed event service.

In order to use the potential of CORBA and Prolog, we first need to write the Logic Server interface in Interface Definition Language (IDL). A simplified version of this interface look like this:

```
interface CORBALogicServer
{
void corbaInitLS(in string xPLName);
void corbaLoadXPL(in string xPLName);
long corbaExecStr(in string aString);
string corbaGetStrArg(in long term, in long
argNum);
void corbaCloseLS();
};
```

We wrap five Logic Server methods into the interface. Note that the *types* in this IDL declaration are not identical with the types used in Java. The IDL syntax precisely allows the designation of *in and out parameters*.

This IDL file is now passed to the idlj compiler, which generates the CORBA middleware code. Our CORBA *implementation* of the Logic Server extends the idlj - generated class _CORBALogicServerImplBase:

```
public class CORBALogicServerImpl extends
_CORBALogicServerImplBase
```

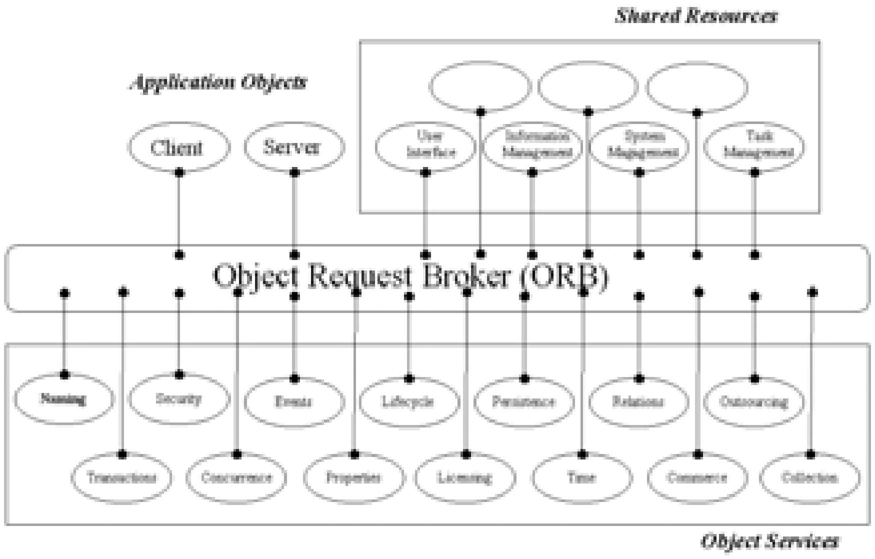


Figure 2 : CORBA Services ([Hoque98])

Initialization and wrapping of the LS engine methods are straightforward. They are identical to the pure Java solution presented above. Its much more interesting when we create an ORB:

```
ORB orb = ORB.init(argv, null);
```

Then we create an instance of the implementation and register it to the ORB. The naming service is also called in order to narrow the context to the freshly created object. This allows us to register the LS engine with the CORBA naming service.

```
CORBALogicServerImpl lsImpl = new
CORBALogicServerImpl();
orb.connect(lsImpl);
org.omg.CORBA.Object dObj = orb.resolve_initial_
references("NameService");
NamingContext dRefObj =
NamingContextHelper.narrow(dObj);
```

Like the Logic Server, the CORBA Naming Service comes as a package :

```
import org.omg.CORBA.*;
import org.omg.CosNaming.*;
```

The CORBA *client* to the Logic Server will make use of the idlj - generated classes CORBALogicServer (the interface) and CORBALogicServerHelper (the stub). The initializing CORBA code is almost identical to the server implementation above, except two lines, where we resolve the CORBA path to instantiate our Amzi! Logic Server Object, and then narrow it to its peer on the server by means of the CORBA Helper:

```
org.omg.CORBA.Object serverObject =
cd.resolve(path);
CORBALogicServer ls =
CORBALogicServerHelper.narrow(serverObject);
```

On the ls instance, we can invoke the five basic methods that we have wrapped in the IDL interface. To make this sample CORBA architecture operational, we need to run the *ORB* (Object Request Broker) on both the client and the server side in order to establish an IIOP communication between them. The JDK 1.3 is shipped with a basic CORBA Naming Service called tnameserv.

```
tnameserv
Initial Naming Context:
IOR:000000000000002849444c3a6 ...
...01010000000000
TransientNameServer: setting port for
initial object references to: 900
Ready.
```

Figure 3: The CORBA Naming Service (tnameserv) Running in a Terminal

The server connects to the CORBA naming service from within a terminal.

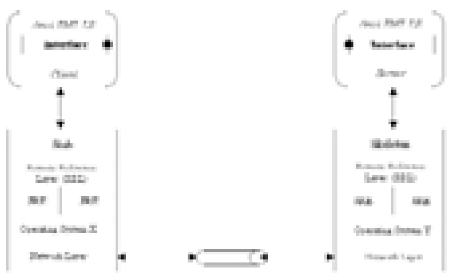


Figure 6 The RMI Architecture

```
java CORBALogicServerImpl
Initialisation of the ORB...
Registering a Logic Server with the ORB...
Calling the naming service...
Registering the Logic Server with the
Naming Service
Waiting for clients to connect...
```

Figure 4: Server Implementation Connecting to the Naming Service

The client then connects to the CORBA logic server through the abstraction provided by the Naming service.

```
java CORBALogicServerCli
Connecting through the ORB to the Logic
Server ...
Good afternoon!
```

Figure 5: A CORBA Logic Server Client Connecting through Localhost at 5:32 p.m.

Embedding Prolog in RMI

Remote Method Invocation (RMI) is Sun's middleware architecture for object distribution. RMI is based on URL references to distributed objects (figure 6):

The implementation of a RMI Logic Server is straightforward with the JDK1.3: first we write an *interface*, not in IDL as for the CORBA Logic Server, but directly in Java. The interface of the simplified RMI Logic Server looks like this:

```
import java.rmi.*;
public interface RMILogicServer extends
java.rmi.Remote
{
public void rmiInitLS(String xPLName) throws
java.rmi.RemoteException;
public void rmiLoadXPL(String xPLName) throws
java.rmi.RemoteException;
public long rmiExecStr(String aString) throws
java.rmi.RemoteException;
public String rmiGetStrArg(long term, int argNum)
throws java.rmi.RemoteException;
public void rmiCloseLS() throws
java.rmi.RemoteException;
}
```

Note that our RMILogicServer class must extend the java.rmi.remote class, and that the wrapper methods must throw a java.rmi.RemoteException.

Then we can write the *implementation* of the RMI Logic Server interface. We basically map incoming method calls to the original Logic Server methods as in the Java and CORBA codes above. A private attribute holds the Logic Server. The interesting part starts when we create an instance of the RMI Logic Server and register it with a unique name in the RMI registry.

```
String name = "RMILS";
RMILogicServerImpl ls = new RMILogicServerImpl();
Naming.rebind(name, ls);
```

The rest is done by the RMI architecture. This code must be compiled first with the javac compiler and then with the rmic compiler. As for CORBA, the RMI naming service is part of the JDK1.3 - it's called rmiregistry. If this rmiregistry is running in a terminal (see figure 7), you can then register the server implementation to the RMI architecture by running java on your .class file (see figure 8).

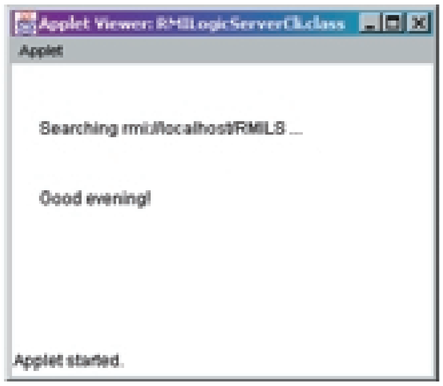
```
rmiregistry
```

Figure 7: The RMI Registry Running in a Terminal

```
java RMILogicServerImpl
Register RMILogicServerImpl as "RMILS"
RMI Logic Server ready...
```

Figure 8: The RMI Logic Server Running in a Terminal

Finally, we have to code the RMI *client*. The latter is very close to the pure Java code segment that we presented earlier in this article. The essential modifications concern



```
Searching rmi://localhost/RMILS ...

Good evening!

Applet started.
```

Figure 9: An RMI Logic Server Client (Applet) connecting through localhost at 11:39 p.m.

the RMI-specific referencing of a remote object (the Logic Server):

```
String name = "rmi://"+hostAdr+"/RMILS";
RMILogicServer ls =
(RMILogicServer)Naming.lookup(name);
```

The interesting part is the lookup of the RMI Naming service. Note that RMI is URL-based. This client program will be compiled with the javac compiler. When you run it hosted within the appletviewer, it may look similar to figure 9.

The RMI functionality comes as a package and is part of the JDK1.3:

```
import java.rmi.*;
```

Conclusions

We have demonstrated the embedding of a Prolog engine within the two major system-neutral middleware architectures, CORBA and RMI, and underlined the potential of logic programming in the context of the WWW. The best way to take advantage of the Logic Server would be an exchange between the Prolog engine and the host language in both ways – a feature that is required to program higher level software agents[2].

Thomas Steiner is a professor at the University of Applied Sciences Valais 3960 Sierre, Switzerland. ***thomas.steiner@hevs.ch http://www.netversity.ch***

References

1. Steiner T.: "Distributed Software Agents for WWW-based Destination Information Systems", PhD Thesis, The University of Lausanne, 1999.
2. Steiner T.: "Software Agents for the Tourism Industry - Prototypes and Perspectives", Informatik / Informatiques 01/2000.
3. Ben-Natan, R.: " CORBA on the Web", McGraw-Hill 1998.
4. Farley, J.: "Java Distributed Computing", O'Reilly 1998.
5. Hoque R.: "CORBA3", IDG Books 1998.
6. McCarty, B. & Cassady-Dorion, L.: "Java Distributed Objects – The Authorative Solution", SAMS 1999.
7. Messerschmitt, D.G: "Understanding Networked Applications – A first course", Morgan-Kaufmann 2000.
8. Ptak, R.L., Morgenthal JP & Forge, S.: "Manager's Guide to Distributed Environments", John Wiley & Sons 1999.
9. Serain, D.: "Le Middleware – Concepts et Technologies", Masson 1997.
10. Steiner T.: "From Objects to Agents - A bottom-up approach", to appear.
11. Watson, M.: "Intelligent Java Applications for the Internet and Intranets", Morgan-Kaufmann 1997.