EXTENSIONS TO JINNI MOBILE AGENTS ARCHITECTURE

Satyam Tyagi, B. Tech.

Thesis Prepared for the Degree of

MASTER OF SCIENCE

UNIVERSITY OF NORTH TEXAS

May 2001

APPROVED:

Paul Tarau, Major Professor

Tom Jacob, Comittee Member and Chair of the
Department of Computer Science

Armin Mikler, Comittee Member

C. Neal Tate, Dean of the Robert B. Toulouse
School of Graduate Studies

Tyagi,Satyam, Extensions to Jinni Mobile Agent Architecture. Master of Science (Computer Science), December 2000, 57 pp., 1 table, 4 figures, 46 references.

We extend the Jinni mobile agent architecture with a multicast network transport layer, an agent-to-agent delegation mechanism and a reflection based Prolog-to-Java interface. To ensure that our agent infrastructure runs efficiently, independently of router-level multicast support, we describe a blackboard based algorithm for locating a randomly roaming agent. As part of the agent-to-agent delegation mechanism, we describe an alternative to code-fetching mechanism for stronger mobility of mobile agents with less network overhead. In the context of direct and reflection based extension mechanisms for Jinni, we describe the design and the implementation of a reflection based Prolog-to-Java interface. The presence of subtyping and method overloading makes finding the most specific method corresponding to a Prolog call pattern fairly difficult. We describe a run-time algorithm which provides accurate handling of overloaded methods beyond Java's reflection package's limitations.

ACKNOWLEDGEMENT

I, Satyam Tyagi, would like to acknowledge all those, who made this thesis a reality. I would like to acknowledge my family. I am grateful to Dr. Paul Tarau for his invaluable advice, guidance and patience at all stages of this thesis.

Lastly, I wish to acknowledge God.

CONTENTS

ACKNOWLEDGEMENT . . . iii

1 INTRODUCTION . . . 1

1.1 What are the problems? . . . 1

1.1.1 Multicast . . . 1

1.1.2 Fallback Mechanism . . . 2

1.1.3 Reflection based Java API for Prolog . . . 2

1.2 Why they are significant? . . . 2

1.3 Overview . . . 3

2 OVERVIEW OF JINNI . . . 6

2.1 Ontology . . . 6

2.2 Basic Features . . . 6

3 EXTENDING JINNI WITH JAVA BASED BUILTINS . . . 8

3.1 The Term Hierarchy . . . 8

3.2 The Builtin Registration Mechanism . . . 9

3.3 The Builtin Execution Mechanism . . . 10

4 EXTENDING JINNI WITH JAVA REFLECTION . . . 11

4.1 The Method Signature Problem . . . 11

4.2 The Prolog predicate API for reflection based Java Interface . . . 12

4.3 The details of implementation . . . 14

4.3.1 Creating a class . . . 14

4.3.2 Instantiating an Object . . . 14

4.3.3 Invoking a method . . . 15

4.4 Limitations of Reflection . . . 16

4.5 Java Compile Time Solution 17
4.5.1 Finding the Applicable Methods 17
4.5.2 Finding the Most Specific Method 18
4.5.3 Overloading Ambiguity 18
4.5.4 Example: Method Resolution at Compile time 19
4.6 Finding a Most Specific Method at Runtime 21
4.7 An example application/GUI using reflection API 24

5 DELEGATION MECHANISM 25
5.1 Local Delegation Mechanism 25
5.2 Remote Agent Delegation Mechanism 25
5.2.1 Case for Fall Back Mechanism 25
5.2.2 Implementation 26
5.2.3 Algorithm 26

6 MULTICAST NETWORKING LAYER FOR JINNI 29
6.1 Multicast 29
6.2 Multicast in Jinni 30
6.2.1 The API 30
6.3 Synchronizing Multicast Agents with Blackboard Constraints 31
6.4 Some Properties and Consequences 33
6.4.1 Impact on mobile Computers and transient IP-address systems 34
6.4.2 IP Transparent Architecture for a Platform of Mobile Agents. 36
6.4.3 Fault tolerant computing 37
6.5 Overcoming Agent Location Problem in absence of Multicast enabled routers 38
6.6 Java3D based Shared Virtual Reality 42

6.6.1 The User's interface 42

6.6.2 The basic API 42

6.7 Tele-teaching 43

7 RELATED WORK 45

7.1 Mobile Agents Architectures 45

7.2 Reflection based Language Extensions 46

8 FUTURE WORK AND CONCLUSION 48

8.1 Future Work 48

8.1.1 Reflection based Prolog Extension 48

8.1.2 Multicast Protocols 48

8.2 Conclusion 49

BIBLIOGRAPHY 51

LIST OF TABLES

4.1 Data Conversion 13

LIST OF FIGURES

3.1 Java Classes of Prolog Term Hierarchy 8

4.1 Screenshot of Prolog IDE written in Prolog 24

6.1 Basic Linda operations with Multicast 32

6.2 Synchronized Java3d worlds using multicast. 41

CHAPTER 1

INTRODUCTION

1.1 What are the problems?

The problems we deal with can be divided into three sub-categories based on how they were dealt with:

1.1.1 Multicast

A major concern with distributed agent programming is to separate as much as possible the distributed aspects and application functionality. We want to make sure even with the advantages and problems with distributed programming the view presented to the programmer is that the application is running on a single machine.

Some of the problems we tried tackling with multicast were :

- Simultaneous propagation of agents to different sites specially on a LAN.
- The agents when they propagate are unable to carry TCP/IP ports with them. We tried to tackle this with location independent property of multicast.
- The need to know the location or the IP address of an agent to communicate with the agent. This is also solved by location independent multicast addressing.
- Sharing of Blackboards for flexible and easier agent communication.
- Fault tolerant computing.
- Synchronization of distributed network applications(Java3D games, Teleteaching)

- We also address the problem of agent location tracking in absence of multicast enabled routers.

1.1.2 Fallback Mechanism

Mobile agents when they move need to execute there code at different sites. Sometimes the code is available at different sites and sometimes it is not under such situations agents typically fetch code from there home site or some sort of a code server. Sometimes this code maybe large or compiled and not so easy to fetch due to network and security constraints. We try to address this problem in a novel way in which agent can fallback to the homesite and execute code there and only carry results to remote site.

1.1.3 Reflection based Java API for Prolog

As our prolog is written in Java we already had an extension mechanism in place to extend our Prolog by mapping new predicates to Java function calls. This mechanism was quite difficult and cumbersome. When providing a simple interface to Java we came across the problem of method-overloading and selection of most specific method at runtime for accurate Java compile time implementation. We address this problem with our own algorithm.

1.2 Why they are significant?

The significance of the problems we deal with lies in various areas of distributed agent programming. We deal with separating the concerns of distributed agent programming from application programming and therefore making distributed agent programming conceptualy simpler. We deal with reducing network overhead and speeding up of agent applications. Also, we deal with extending our agent architecture with the latest Java libraries and in the process solving object oriented programming concerns arising due to run time execution of reflected code.

We are able to parallelize searches and make agent propagation to a group of sites a single step operation with multicast. Also communication ports and communication are made site (address) independent. We are able to synchronize real time distributed applications and make applications fault tolerant with multicast. The sharing of blackboards through multicast channels makes agent communication more flexible and powerful.

We address the agent location tracking problem in the absence of multicast using linda blackboards. The blackboards alleviate the need for code fetching with our fallback mechanism.

The Java extension mechanism with Java reflection allows for inclusion of Java libraries for our Prolog based agent infrastructure. The architecture can use any of the Java libraries with the provided API almost completely in Prolog as shown by our example application. In making the Java reflection interface with Prolog we realised certain shortcomings of the reflection package in method overloading. We describe a new run-time algorithm which closely mimics Java's own compile-time method dispatching mechanism and provides accurate handling of overloaded methods beyond the reflection package's limitations. The algorithm is not specific to our interface but makes Java's reflection package more powerful.

1.3 Overview

The thesis is divided into several chapters. We start of with describing the mobile agent architecture of Jinni (Java Inference Network Engine). The architecture briefly describes the ontology to support mobile agents and some basic features which help us to execute mobile agent Prolog scripts at remote sites and to help mobile agents to coordinate and communicate.

Next we describe mechanism for extending Jinni with new Java based builtins and the mapping between Prolog predicates and Java methods, and mapping between Prolog terms and other Prolog types to Java classes. The reflection based Java extension method is explained next, which makes the process of extending Jinni a much simpler process with automated data-type conversion and our runtime algorithm for finding most specific method amongst overloaded methods. We end the chapter by explaining an example application, which helps us to understand the power and usage of this extension mechanism.

Local delegation mechanism from Jinni's fast compiler and easy to extend interpreter are briefly stated. Our alternative to code fetching - remote agent delegation mechanism to an agent's homesite or a code-server are described. We also discuss the cases where this maybe more advantageous as compared to code fetching and how it can be used more flexibly.

The multicast networking layer for Jinni is described next. We discuss the API and implementation briefly. We then describes its impact on Linda blackboards the way each agents view to the blackboard world depends on which group it belongs to. Protocol for multicasting and collecting data and code from groups of blackboards at remote sites. The impact of properties of multicast on mobile and transient IP address systems is discussed with emphasis on new possibilities with location independent property of multicast addresses and groups. We describe an architecture for an IP transparent mobile agent architecture using new multicast layer and a fault tolerant protocol. We also discuss an algorithm and it's complexity for agent location tracking in absence of multicast enabled routers. Finally we describe two possible applications: Teleteaching and Java3D games and their implementations on our enhanced mobile agent system.

We discuss other work taking place in related fields; mainly in Java reflection based language extensions and Prolog based distributed mobile agent architectures. We explain the related directions and highlight the differences in our approach. Finally we conclude with the current achievements and possible directions for future research.

CHAPTER 2

OVERVIEW OF JINNI

2.1 Ontology

Jinni is based on simple **Things, Places, Agents** ontology.

Things are Prolog terms (trees containing constants and variables, which can be unified and other compound sub-terms).

Places are processes with at least one server and a blackboard allowing synchronized multi-user Linda and Remote Predicate Call transactions. The blackboard stores Prolog terms, which can be retrieved by agents.

Agents are collections of threads executing various goals at various places. Each thread is mobile, may visit multiple places and may bring back results.

2.2 Basic Features

Jinni is a Prolog interpreter written in Java, which provides an infrastructure for mobile logic programming (Prolog) based agents. It spawns interpreters as threads over various network sites and each interpreter has its own state. Computation mobility is mapped to data mobility (through use of meta-interpreters, data can be treated as code). Mobile threads can capture first order "AND"-continuations (as "OR" continuations would cause backtracking, which is not a good idea over the network) and resume execution at remote site by fetching code as needed.

Shared blackboards are used for communication and coordination of agents. Jinni has an orthogonal design and separates high level networking operations (supporting remote predicate calls and code mobility), from Linda coordination code. It has various plugins for GUI, different Network layers (in particular the multicast layer described in this paper) and a Java3d interface, which can be plugged in as an extension. Jinni 2000 embeds a fast incremental compiler, which provides Prolog processing within a factor of 5-10 from the fastest C-based implementations around.

For more details on Jinni see [8, 9].

CHAPTER 3

EXTENDING JINNI WITH JAVA BASED BUILTINS

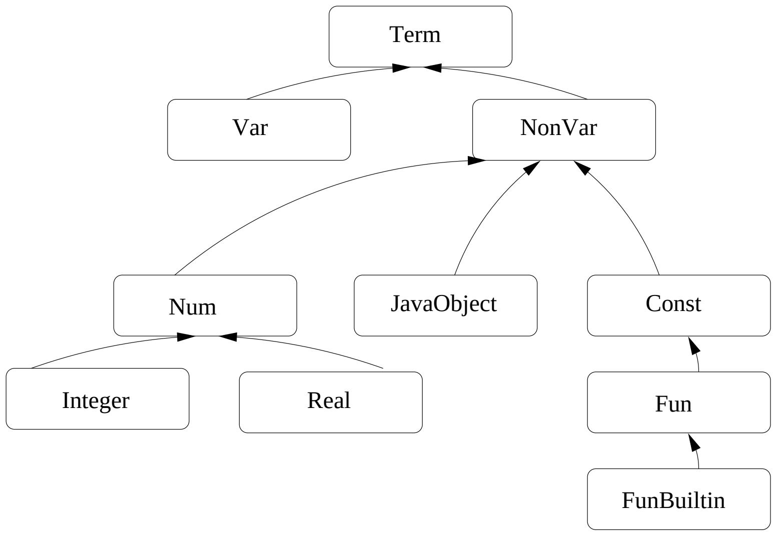


Figure 3.1: Java Classes of Prolog Term Hierarchy

3.1 The Term Hierarchy

The base class is **Term** which has two subclasses: **Var** and **NonVar**. The **NonVar** class is in turn extended by **Num**, **JavaObject** and **Const**. Num is extended by **Integer** and **Real**. **Term** represents the generic Prolog term which is a finite tree with unification operation distributed across data types - in a truly object oriented style [36]. The **Var** class represents a Prolog variable. The **Integer** and **Real** are the Prolog Numbers. **Const** represents all symbolic Prolog constants, with the compound term (called *functor* in Prolog) constructor class **Fun** designed as an extension of **Const**.

JavaObject is also a subclass of **Const** which unifies only with itself[1] and is used like a wrapper around Objects in Java to represent Prolog predicates.

3.2 The Builtin Registration Mechanism

Jinni's Builtins class is a specialized subclass of Java's Hashtable class. Every new component we add to Jinni 2000 can provide its own builtin predicates as a subclass of the Builtins class. Each added component will have many predicates, which are to be stored in this Hashtable mapping their Prolog representation to their Java code, for fast lookup. Let us assume the Prolog predicate's name is *prologName* and the corresponding Java classes name is *javaName*. We make a class called *javaName* which extends FunBuiltin (a descendant of Term with which we represent a Prolog functor (compound term). It accepts a string (the functor's name) and an integer in its constructor (arity). When we call the **register** method of the appropriate descendant of the Builtins class, a new Hashtable entry is generated with the supplied *prologName* and the arity as key and *javaName* as its value. Whenever the Prolog predicate *prologName* with appropriate arity is called, we can look up in constant time which class (*javaName*) actually implements the**exec** method of the builtin predicate in Java. Each component extending the Builtins class will bring new such predicates and they will be added to the inherited Hashtable with the mechanism described above - an therefore will be usable as Prolog builtins as if they were part of the Jinni 2000 kernel.

[1]Modulo Java's *equals* relation.

3.3 The Builtin Execution Mechanism

The descendents of the **FunBuiltin** class implement builtins which pass parameters, while the descendents of the **ConstBuiltin** class implement parameterless builtins. Both **FunBuiltin** and **ConstBuiltin** have an abstract method called **exec** to be be implemented by the descendent *javaName* class. This is the method that is actually mapped to the Prolog builtin predicate with *prologName* and gets invoked on execution of the predicate. The **exec** method implemented by the *javaName* class will get arguments (**Term** and its subclasses) from the predicate instance using **getArg** methods and will discover their dynamic through a specialized method. Once we have the arguments and know their types we can do the required processing. The **putArg** method, used to return or check values, uses the **unify** method of Terms and its subclasses to communicate with the actual (possible variable) predicate arguments. On success this method returns **1**. If putArg does not fail for any argument the exec method returns **1**, which is interpreted as a **success** by Prolog. If at least one unification fails we return **0**, which is interpreted as a **failure** by Prolog. We call this mapping a *conventional builtin* as this looks like a builtin from Prolog side, which is known at compile time and can be seen as part of the Prolog kernel.

CHAPTER 4

EXTENDING JINNI WITH JAVA REFLECTION

4.1 The Method Signature Problem

Most modern languages support method overloading (the practice of having more than one method with same name). In Java this also interacts with the possibility of having some methods located in super classes on the inheritance chain. On a call to an overloaded method, the resolution of which method is to be invoked is based on the method signature. Method signature is defined as the name of the method, its parameter types and its return type[1].

The problem initially seems simple: just look for the methods with the same name as call, number and type of parameters as the arguments in the call and pick that method.

The actual problem arises because Java allows *method invocation type conversion.* In other words this means that we are not looking for an exact match in the type of a parameter and the corresponding argument, but we say it is a match if the type of argument can be converted to the type of a corresponding parameter by method invocation conversion [35]. Apparently, this also does not seem to be very complicated: we just check if the argument type converts to the corresponding parameter type or not. The problem arises because we may find several such matches and we have to search among these matches the **most specific method** - as Java does through compile time analysis. If such a method exists, then that is the one we invoke. However, should this search fail, an error has to be reported stating that no single method can be classified as the most specific method.

[1]In resolving the method call Java ignores the return type.

This chapter will propose a comprehensive solution to this problem, in the context of the automation of type conversions in Jinni 2000's bidirectional Prolog to Java interface.

4.2 The Prolog predicate API for reflection based Java Interface

Our reflection based Jinni 2000 Java Interface API is provided through a surprisingly small number of conventional Jinni builtins. This property is shared with the JIPL [42] interface from C-based Prologs to Java. The similarity comes ultimately from the fact that Java's reflection package exhibits to Java the same view provided to C functions by JNI - the Java Native Interface:

new_java_class(+'ClassName',-Class).

This takes in as first argument the name of the class as a constant. If a class with that name is found it loads the class and a handle to this class is returned in the second argument wrapped inside our *JavaObject*. Now this handle can be used to instantiate objects.

new_java_obj(+Class,-Obj):-new_java_obj(Class,new,Obj).
new_java_obj(+Class,+new(parameters),-Obj).

This takes in as the first argument a Java **class** wrapped inside our *JavaObject*. In the case of a *constructor* with parameters, the second argument consists of **new** and parameters (Prolog numeric or string constants or other objects wrapped as *JavaObjects*) for the constructor. As with ordinary methods, the (most specific constructor) corresponding to the argument types is searched and invoked. This returns a handle

Java	**Prolog**
int	
maybe (short,long)	Integer
double	
maybe (float)	Real
java.lang.String	Const
any other Object	JavaObject is a bound variable, which unifies only with itself

Table 4.1: Data Conversion

to the new object thus created again wrapped in *JavaObject* in the last argument of the predicate. If the second parameter is missing then the void constructor is invoked. The handle returned can be used to invoke methods.

invoke_java_method(+Object,+methodname(parameters),-ReturnValue).

This takes in as the first argument a Java class's instantiated object (wrapped in *JavaObject*), and the methodname with parameters (these can again be numerical or,string constants or objects wrapped as *JavaObjects*) in the second argument. If we find such an (accessible and unambiguously most specific) method for the given object, then that method is invoked and the return value is put in the last argument. If the return value is a number or a string constant it is returned as a Prolog number or constant else it is returned wrapped as a *JavaObject*.

If we wish to invoke static methods the first argument needs to be a class wrapped in JavaObject - otherwise the calling mechanism is the same

The mapping of datatypes between Prolog and Java looks like this:

4.3 The details of implementation

4.3.1 Creating a class

The reflection package uses the Java reflection API to load Java classes at runtime, instantiate their objects and invoke methods of both classes and objects. The Java Reflection *Class.forName("classname")* method is used to create a class at runtime. In case an exception occurs, an error message stating the exception is printed out and a **0** is returned, which is interpreted as a **failure** by Prolog. The error message printing can be switched on/off by using a flag.

This is interfaced with Prolog using the conventional Builtin extension mechanism getting the first argument passed as a Prolog constant seen by Java as a String. After this, the Java side processing is done and the handle to the required class is obtained. Finally this handle wrapped as a *JavaObject* is returned in the second argument.

Example:

new_java_class('java.lang.String',S)

Output:

S=JavaObject(java.lang.Class_623467)

4.3.2 Instantiating an Object

First of all, the arguments of a constructor are converted into a list, then parsed in Prolog and provided to Java as *JavaObjects*. Then each one is extracted individually. If the parameter list is empty then a special token is passed instead of the JavaObject, which tells the program, that a void constructor is to be used to instantiate a new object from the class. This is done by invoking the given class' *newInstance()* method, which returns the required object.

If the argument list is not empty, the class (dynamic type) of the objects on the argument list is determined using the *getClass()* method and stored in an array. This array is used to search the required constructor for the given class using the *getConstructor(parameterTypes)* method. Once the constructor is obtained, its *newInstance(parameterList)* method is invoked to obtain the required object. The exception mechanism is exactly the same as for creating a new class as explained above.

This also uses the conventional Builtin extension mechanism to interface with Java, therefore Objects are wrapped as *JavaObjects*. Prolog **Integers** are mapped to Java **int** and Prolog's **Real** type becomes Java **double**. The reverse mapping from Java is slightly different as **long**, **int**, **short** are mapped to Prolog's **Int**, which holds its data in a **long** field and the **float** and **double** types are mapped to Prolog's Real (which holds its data in a double field). Java Strings are mapped to Prolog constants and vice versa (this is symmetric).

Example:

new_java_obj(S,new(hello),Mystring)

Output:

MyString=JavaObject(java.lang.String_924598)

4.3.3 Invoking a method

The method invoking mechanism is very similar to the object instantiation mechanism. The mapping of datatypes remains the same. The exception mechanism is also exactly same as that of constructing objects and classes.

First we determine the class of the given object. The getConstructor method is replaced by *getMethod(methodName, parameterTypes)* except that it takes in as the first argument a method name. Once the method is determined, its return type is determined using the *getReturnType().getName()* for the mapping of Prolog and Java datatypes following the convention described earlier. If the return type is void the value returned to Prolog will be the constant 'void'. To invoke the required method (*the method we wish to invoke*) we call the obtained method's *invoke.(Object, parameterList)* method and will return after conversion the return value for the given method.

To invoke static methods, first we determine whether the object passed as the first argument is an instance of the class **Class**. If so, this is taken to be the class whose method is to be searched, and the call to invoke looks like *invoke.(null, parameterList)*

Example

invoke_java_method(Mystring,length,R)

Output:

R=5

Example

invoke_java_method(Mystring,toString,NewR)

Output:

NewR=hello

4.4 Limitations of Reflection

An important limitation of the reflection mechanism is that when we are searching for a method or a constructor for a given class using the given parameter types. The reflection package looks for exact matches. That means if we have an object of class Sub and we pass it to a method, which accepts as argument an object of class Super, which is Sub's super-class, we are able to invoke such a method in normal Java, but in case of reflection our search for such a method would fail and we would be unable to invoke this method. The same situation occurs in the case for an accepting an **interface** method, which actually means accepting all objects implementing the **interface**. The problem arises in the first place because method could be overloaded and Java decides, which method to call amongst overloaded methods at compile-time and not at runtime. We discuss in the next section how the Java compiler decides, which method to call at compile time.

4.5 Java Compile Time Solution

The steps involved in the determination of which method to call once we supply the object whose method is to be invoked and the argument types.

4.5.1 Finding the Applicable Methods

The methods that are applicable have the following two properties:

- The name of the method is same as the call and the number of parameters is same as the arguments in the method call.
- The type of each argument can be converted to the type of corresponding parameter by method invocation conversion.

This broadly means that either the parameter's class is the same as the corresponding argument's Class, or that it is on the inheritance chain built from the argument's class upto **Object**. If parameter is an **interface**, the argument implements that interface. We refer to [35] for a detailed description of this mechanism.

4.5.2 Finding the Most Specific Method

Informally, **method1** is more specific than **method2** if any invocation handled by **method1** can also be handled by **method2**.

More precisely, if the parameter types of method1 are M_{11} to M_{1n} and parameter types of method2 are M_{21} to M_{2n} method1 is more specific then method2 if M_{1j} can be converted to M_{2j} for all**j** from **1** to **n** by method invocation conversion.

4.5.3 Overloading Ambiguity

In case no method is found to be most specific then method invocation is ambiguous and a compile time error occurs.

Example:
Consider class A superclass of B and two methods with name m.

m(A,B)
m(B,A)

Now an invocation which can cause the ambiguity is.

m(instance of B, instance of B)

In this case both method are applicable but neither is the most specific as *m(instance of A,instance of B)* can be handled only by first one while *m(instance of B,instance of A)* can be handled only by second one i.e. either of the method's all parameters can not be converted to other's by method invocation conversion.

4.5.4 Example: Method Resolution at Compile time

Method resolution takes place at compile time in Java and is dependent on the code for call to the method. This becomes clear from the following example.

Consider two classes Super and Sub where Super is superclass of Sub. Also consider class A with a method m and class Test with a method test, the code for the classes looks like this:

Super.java

```
public class Super {}
```

Sub.java

```
public class Sub extends Super{}
```

A.java

```
public class A {
  public void m(Super s) { System.out.println("super");}
}
```

Test.java

```
public class Test {
  public static void test(){
    A a=new A();
    a.m(new Sub());
  }
}
```

On invocation of method *test()* of class **Test**, method *m(Super)* of class **A** is invoked and **super** is printed out. Let's assume that we change the definition of the class **A** and overload the method *m(Super)* with method *m(Sub)* such that **A** looks like this:

A.java

```
public class A {
  public void m(Super s) {System.out.println("super");}
  public void m(Sub s) {System.out.println("sub");}
}
```

On invocation of method *test()* of class **Test**, method *m(Super)* of class **A** is invoked and **super** is printed out. Let's assume that we change the definition of the class **A** and overload the method *m(Super)* with method *m(Sub)* such that **A** looks like this:

A.java

```
public class A {
  public void m(Super s) {System.out.println("super");}
  public void m(Sub s) {System.out.println("sub");}
}
```

If we recompile, and run our test method again, we expect **sub** to be printed out since *m(Sub)* is more specific than *m(Super)* but actually **super** is printed out. The fact is method resolution is done when we are compiling the file containing the method call and when we compiled the class Test we had the older version of class A and Java had done resolution based on that class A. We can get the expected output by recompiling class Test, which now views the newer version of class A and does the resolution according to that, and hence we get the expected output **sub**.

4.6 Finding a Most Specific Method at Runtime

We will follow a simple algorithm. Let's assume that the number of methods which are accessible and have same name and number of parameters as the call is **M** (small constant) and the number of arguments in the call is **A** (a small constant). Let us assume that the maximum inheritance depth of a the class of an argument from Object down to itself in the class hierarchy tree is **D** (a small constant) It can be trivially shown that the complexity of our algorithm is bounded by **O(M * A * D)**. Our algorithm mimics exactly the functionality of Java and the following example would run exactly the same on both Java and our interface, the only difference being that since Java does the resolution at compile time, in case of an ambiguous call Java would report a compile time error while we do the same thing at runtime and hence, throw an exception with appropriate error message. So if class A looks like this:

A.java

```
public class A {
  public void m(Super s1,Sub s2) {System.out.println("super");}
  public void m(Sub s1, Super s2) {System.out.println("sub");}
}
```

and the class Test looks like this: **Test.java**

```
public class Test {
  public static void test(){
    A a=new A();
    a.m(new Sub(),new Sub());
  }
}
```

then Java will not compile class **Test** and give an error message. In our case there is no such thing as the class **Test**, but the equivalent of the above code would look like follows:

```
new_java_class('A',Aclass),
new_java_object(Aclass,Aobject),
new_java_class('Sub',Subclass),
new_java_obj(Subclass,Subobject1),
new_java_obj(Subclass,Subobject2),
invoke_java_method(Aobject,m(Subobject1,Subobject2),Return).
```

The result will be an ambiguous exception message and the goal failing with **no**.

Our Algorithm   We will now describe our algorithm in detail:

- **1.** If the method is an exact match and hence reflection gives a valid method object, call the method and stop, else:
- **2.** Get the accessible methods with same name and number of arguments as the call and store in an array (MethodArray). (Size M)
- **3.** Declare a corresponding MethodPropertyarray (Size M*A)
- **4.** Get the parameter types for each member of MethodArray in an array of ParameterTypes (ParameterArray). (Size M*A)
- **5.** For Arguments $\mathbf{a} = 0$ to $\mathbf{A}$ do (6,7,8,9,10,11,12).
- **6.** Depthcounter = 0.
- **7.** While Arguments[a].type != null do (8,9,10,11).
- **8.** For methods $\mathbf{m} = 0$ to $\mathbf{M}$ do (9).

- **9.** If ParameterArray[m][a] = the Argument[a].type

  MethodProperty[m][a] = Depthcounter*Loops over D*

  end For (9)

- **10.** Argument[a].type = Super(Argument[a].type).

- **11.** Increment depthcounter.

  end While (7)

- **12.** For methods **m** = 0 to **M** do

  If MethodProperty[m][a]=no value then MethodProperty[m][a]=infinity. (*Since, this does not come in argument types hierarchy chain to object, it is not an acceptable method*)

  end For (5).

- **13.** For **m** = 0 to **M** do

  Find a unique **m** such that Sum of MethodProperty[m][a] over all **a** is minimum. Store this in **mChosen**.

- **14.** For **m** = 0 to **M** and **m** != **mChosen** do (15,16)

- **15.** For **a** = 0 to **A** do (16)

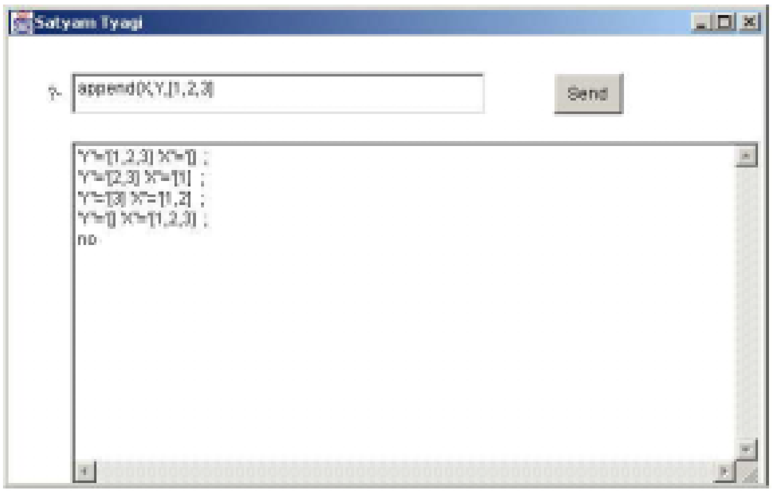
- **16.** If(MethodProperty[m][a] > MethodProperty[mChosen][a])

  Throw ambiguous exception

  end For (14)

  end For (15)

- **17.** Call the Method[mChosen] with given arguments.



```
append(X,Y,[1,2,3])
```

```
Y'=[1,2,3] X'=[] ;
Y'=[2,3] X'=[1] ;
Y'=[3] X'=[1,2] ;
Y'=[] X'=[1,2,3] ;
no
```

Figure 4.1: Screenshot of Prolog IDE written in Prolog

4.7 An example application/GUI using reflection API

This GUI has almost completely been implemented in Prolog using the reflection API. A special builtin which, allows us to redirect output to a string is used to interface default Prolog i/o to textfield/textarea etc. The total Java code is less than 10 lines. Jinni provides, on the Java side, a simple mechanism to call Prolog *Init.jinni("Prolog command")*. Since we do not have references to different objects in the Java code, but everything is in the Prolog code, we need a mechanism to communicate between Java's action-listener and Prolog. Jinni's Linda blackboards have been used for this purpose [8, 9]. Whenever an action takes place the Java side calls Jinni and does a out with a number for type of action on the blackboard by calling something like *Init.jinni("out(a(1))")*. On the Prolog side we have a thread waiting on the blackboard for input by doing an *in(a(X))*. After the out variable **X** gets unified with **1** and depending on this value, Prolog takes the appropriate action and again waits for a new input. Hence, we can make action events such as button clicks communicate with Prolog.

CHAPTER 5

DELEGATION MECHANISM

5.1 Local Delegation Mechanism

Jinni has two parts a fast incremental compiler and an interpreter. Some of the features are on the compiler while some are on the interpreter. By default Jinni is supposed to be running in compiler mode.
Whenever the compiler is unable to find definition of some predicate in it's compiled code it falls back on the interpreter.
The interpreter in turn falls back on the compiler but as is obvious this can go on in an infinite loop. This is handled by the predicate *iscompiled(undefined predicate)*. The interpreter before falling back on the compiler checks if the compiled definition already exists else it simply fails with an error message.

5.2 Remote Agent Delegation Mechanism

5.2.1 Case for Fall Back Mechanism

The Fall Back Mechanism is proposed as an alternative to mobile live code. When an agent moves from one place to another with prolog predicates it may not carry the complete definitions of predicates. Now if the code is available at the target site it is executed there else the agent falls back to it's home site and executes the code for the undefined predicate and carries the results and bindings to the target site.

The main advantage of this is that the code never moves. This is an advantage when we have proprietary algorithms and do not wish to reveal our algorithm. Secondly when the network is of low bandwidth and predicates have large definitions and code. Also if there is compiled code it can not be executed remotely, thus extending the concept of mobile code from interpreted code to compiled code as well.

5.2.2 Implementation

The implementation of the mechanism has been achieved utilizing the existing infrastructure of Jinni. Jinni has Blackboards at each place to which clauses can be asserted where the definition of currently available prolog clauses are already there. Next we explain the algorithm followed by the agent and the interpreter at each site.

5.2.3 Algorithm

The agent on arriving at the destination asserts it's hosts IP with it's current thread id as it's unique identifier at that site. Now whenever the interpreter fails to find definition for a predicate in it's database (and the predicate is not found in the compiled code) it searches for the IP address of the agent and sends request for results to it's home machine at a fixed port. At the home machine a server is running on this fixed port which responds to such request.
The prolog implementation looks like this.

```
undefined(G):-
 not(iscompiled(G)),
 current\_thread(ID),
 ip(Host,ID), /*searches for the IP using the current thread as key*/
 remote\_run(Host,8001,Gs,findall(G,G,Gs),none,the(Gs),
 member(G,Gs).
```

Code for Agent

```
out_name(Y):-
```

```
 current_thread(X), /*gives identifier for current thread*/
 assert(ip(Y,X)). /*asserts to the data base*/
in_name(Y):-
 current_thread(X), /*gives identifier for current thread*/
 retract(ip(Y,X)). /*retracts from the data base*/
remote_do(Name,G):-
 remote_run((out_name(Name),G,in_name(Name))). /*executes predicate remotely*/
remote_exec(G):-
 my_host_name(Name), /*gives the current host name*/
 remote_do(Name,G). /*executes predicate remotely*/
```

Properties There are some properties for this mechanism which make it even more interesting

- No trail or trace of agent is left on the destination machines.
- There is complete transparency as to where the undefined predicates are executed.
- If the agent moves from one machine to another and so on the encapsulation is such that the undefined predicate is always executed at the place where it originated.

The encapsulation looks like this:

```
X:-
 remote_exec(Host1,(G1,remote_exec(Host2,G2),G3))
```

The expanded encapsulated code looks like this:

```
X:-
```

```
 my_host_name(Name), /*where Name is Initial Host*/
 remote_run(Host1,
 (out_name(Name),
 G1,remote_exec(Host2,G2),
 G3,in_name(Name)
 )
 ).
remote_exec(Host2,G2):-
 my_host_name(Name), /*where Name is Host1*/
 remote_run(Host2,(out_name(Name),G2,in_name(Name))).
```

This makes sure the 'Name' gets bound at the home machine and hence the delegation is directly to home machine. Since the name is specified we have an option to specify a special code server where we want all our agents undefined predicate to be executed.

CHAPTER 6

MULTICAST NETWORKING LAYER FOR JINNI

6.1 Multicast

Definition: 1. In a network, a technique that allows data, including packet form, to be simultaneously transmitted to a selected set of destinations. Some networks, such as Ethernet, support multicast by allowing a network interface to belong to one or more multicast groups.

2. To transmit identical data simultaneously to a selected set of destinations in a network, usually without obtaining acknowledgment of receipt of the transmission. [6].

The key concepts of Multicast Networks are described in [4]. On an Ethernet (Most popular LAN architecture) each message is broadcasted and the machine seeing its own address grabs the message. The multicast packets are also sent in a similar way except that more than one interface picks them up.

The multicast packets received at the interface but not subscribed to are rejected at interface level, if the interface has hardware and software to support it. Interfaces are able to handle often 64 but up-to 512 groups. On overloading interfaces go into "multicast promiscuous" mode and send all packets to the TCP/IP stack. Here after searching through the list of groups a host is subscribed to packets are accepted or rejected. This requires some CPU cycles [2].

We consider 512 is a pretty significant number presently and even if it does go into the "multicast promiscuous" mode the penalty is insignificant. Thus the spreading and cloning of agent threads/agents themselves on the whole network or a subset multicast group is now a single step operation. This leads to important speed up especially when multiple copies of same code need to be executed at different places (like for parallel searches or for simultaneous display in shared virtual reality applications).

6.2 Multicast in Jinni

Multicasting has various interesting properties, which make it well suited for an agent platform like Jinni. An important advantage of multicasting agents is that, the same code can be run in parallel at the same time in one single operation at different remote sites, retrieving different data available at different sites.

6.2.1 The API

A minimal API consists of two predicates one to run multicast servers, which service requests for multicast clients and second to send multicast requests to these servers:

run_mul_server This joins a multicast group with an address and port. The server now is listening on this port and can receive remote requests for local execution. (When we join a group we are telling the kernel, "I am interested in this multicast group. So, deliver (to any process interested in them, not only to me) any datagram that you see in this network interface with this multicast group in its destination field.")

run_mul_server(Host, Port)

Notice that **run_mul_server** has a Host field as well, because it does not run on the local IP but on a multicast address i.e. 224.x.x.x 239.x.x.x

remote_mul_run(G) This is kind of a goal (G), which is multi-casted to the group to be run remotely on all the multicast servers accepting requests for this group.

remote_mul_run(Host, Port, Answer, Goal, Password, Result)

A set of multicast servers is run, which listen for packets on their group and respond back on the group multicast address. All the clients listening on this group receive these results. An important issue here is that the server should be able to distinguish between a request and a reply. Otherwise it would keep responding back to its own replies. This is solved by introducing a simple header distinguishing the two types of messages, which are stripped of when appropriate.

6.3 Synchronizing Multicast Agents with Blackboard Constraints

The synergy between mobile code and Linda coordination allows an elegant, component wise implementation. *Blackboard operations are implemented only between local threads and their (shared) local blackboard.* If interaction with remote blackboard is needed, the thread simply moves to the place where it is located and proceeds through local interaction. The interesting thing with multicast is that the thread can be multi-casted to a set of places and can interact at all these places locally. This gives an appearance that all these blackboards are one to the members of this multicast groups. For example the **remote_mul_run(mul_all(a(X),Xs))** operation is multi-casted to all servers in the group. It collects lists of matching results at these remote servers and the output is unicast-ed from all these remote sites to the local blackboard.

This can be achieved as follows:

mul_all(X,Xs):-mul_all('localhost',7001,X,Xs).

mul_all(Port,X,Xs):-mul_all('localhost',Port,X,Xs). - (the defaults where our server, which receives the results is running)

mul_all(Host,Port,X,Xs):-all(X,Xs), - executes on remote servers.
remote_run(Host,Port,forall(member(Y,Xs),out(Y))) - executes back on our server.

Host and Port determine the address we want the answers to come back on and the answers are written on the local blackboard from where they can be collected.

Note: in/1 and out/1 are basic Linda blackboard predicates[15]

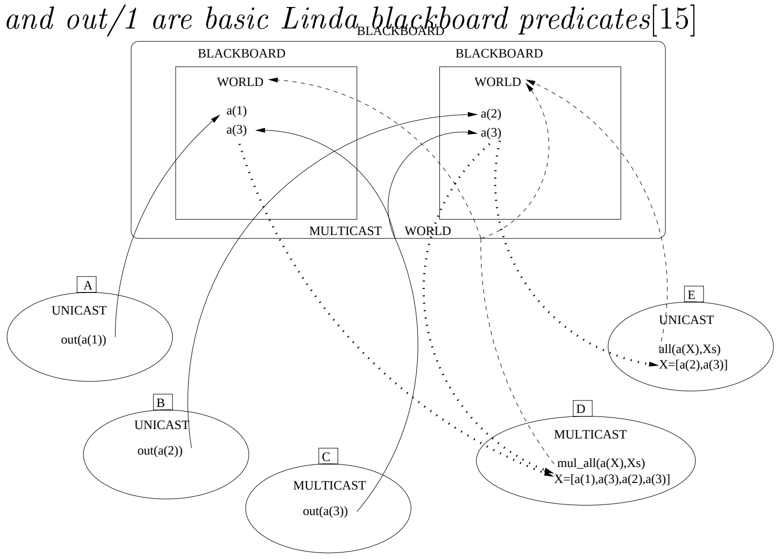


Figure 6.1: Basic Linda operations with Multicast

Places are melted into peer-to-peer network layer forming a 'web of interconnected worlds'. Now different worlds can be grouped into multicast groups. The member of multicast groups viewed as these groups as one world. The concept extends further as groups can intersect and be subsets or supersets and basically have combination of unicast and multicast worlds. The point is that each agents views the world it is interacting with depends on how it is moving its threads to different places.

In the Fig. 1 A, B unicast outputs to two different worlds while C multicasts output to both. The 'unicast all' in E is able to collect from only one blackboard while 'multicast all' in D collects from both. The **all** operations collect multiple copies of the term **a(3)**, but a sort operation could remove such duplication.

6.4 Some Properties and Consequences

There are various interesting properties of multicast networks, which open up many possibilities for future work especially regarding mobile agents and new architectures for their implementation.

As previously discussed there are three types of mobility in a network software environment:*data mobility, code mobility and computation or thread mobility.* An important shortcoming of computation mobility was that if the thread was providing a service or listening on a particular **(host, port)** it could no longer do so once it moved. In other words, ports are not mobile.

Some properties of multicast addresses and ports overcome exactly this shortcoming. These properties are:

- multicast address and port are same for a multicast group and are independent of host or IP address of the host(*IP Transparent*)

- it is possible to have two servers with same multicast address and port running on the same machine. (*In other words we do not need to investigate if a server* *with same port and IP is already running.*) Both servers can respond to the request, a client can chose if it wants one or all answers. Also a header can be put, which helps servers discard requests not intended for them.

This means that when live code or a thread migrates it can just does a *joingroup* [1] on the same group it belonged to and start listening or providing service on the same multicast address and port.

6.4.1 Impact on mobile Computers and transient IP-address systems

A mobile computer like a laptop, palmtop etc. does not have a permanent IP-address because one may connect to one's office, home, in an airplane etc. The transient IP address can also come from one connecting through a dialup connection to an ISP. Such systems can launch mobile agents and receive results when connected and hence can be clients [3].

An important impact of the proposed multicast agent architecture on such transient IP-address systems is that they can also *provide a service* and *listen* on a known multicast address and port whenever they are connected to the Internet. This is possible because to listen on a multicast port one's IP address is not needed. One can have any IP address and still listen on the same Multicast address and port.

Another concept in the Jinni architecture is that of **mobile Linda servants**[7]. A **servant** is an agent, which is launched and on reaching the destination can pull commands from the launching site or other clients, and run them locally.

```
servant:-
  in(todo(Task)),
  call(Task),
```

[1]When we join a group we are telling the kernel, "I am interested in this multicast group. So, deliver (to any process interested in them, not only to me) any datagram that you see in this network interface with this multicast group in its destination field."

`servant.`

Note that the servant is a background thread and blocks when none of the clients in the group have a task to be done i.e. no 'busy wait' is involved [7].

We will try to expand on these two concepts of multicast and servants to generalize the client/server architecture especially for mobile and transient IP-address systems.

Case 1 Mobile Servers

Even when the mobile server is disconnected it can have **servant agents** running on it, doing the processing for its clients and posting results or queries on the local blackboard. In the mean time, the clients keep making lists of the tasks they want to get done on the server. When the server comes up, the servant can pull the list of tasks to be done by clients and run them. Also the server can have a new IP address but the same multicast address, when the server reconnects. The clients having knowledge of this can collect the required responses from the servers' blackboard.

Case 2 Mobile Clients

Even when disconnected, the mobile client can launch an agent on a different machine, which can do the listening for it. Whenever the client reconnects it can pull list of tasks and results (*the agents do processing*) from the agent and destroy the agent. Whenever the client is disconnecting it can launch the agent again.

This concept can also be extended, as both clients and servers can be made mobile or with unknown or transient IP-addresses with multicasting. As we discussed before, to communicate on a multicast channel we do not need to know the IP. We explore this concept of IP transparent communication further in the next subsection. Some ideas of this mixed mobility of computers and agents are discussed in [1].

Applets as servers

Another possible use can be in the case of applets, which can not listen on ports but can connect to the server they are coming from. This means they can connect to their server pull tasks from it and execute them locally on their current machine (*hence pretend to be servers*).

This architecture could possibly be three-tier. The applets connect to their server, which offers services to other clients. The server now posts the requests on the local blackboard to be pulled by the applets and executed locally on their machine.

6.4.2 IP Transparent Architecture for a Platform of Mobile Agents.

Consider a group of agents moving freely in the network. Let's assume each agent is a member of two multicast groups: a common shared group address between all agents and a unique personal multicast address. Whenever they propagate they do a *joingroup* on both these multicast groups.

The analogy for private address is that of a cell phone. Each agent can communicate with the others on its private multicast address **being completely unaware about the location of one it is sending messages to.** The best analogy for the shared common address is that of a broadcast radio channel. Whenever an existing agent spawns a new agent it gives the new agent the set of addresses known to it and the new agent chooses a new private multicast address and communicates to the rest of the agents (via the shared common group address) its private address. Metaphorically, this is like (**broadcasting on the radio channel its cell phone number to communicate with it**)

The main advantage of this architecture is that it carries out communication amongst different agents without any knowledge of each others current location, i.e. no agent requires the knowledge of other's IP address to communicate whether they want the communication to be public within a group or private.

Among the application of a such an architecture could be found in situations where the agents need to communicate with each other but do not have a fixed itinerary or the itinerary changes with the decision the agent makes. The address need not be communicated on each hop but only when a new agent is spawned it needs to make its address known.

For now, the important question of lost messages during transportation of agents remains unanswered. One must not forget multicast is based on UDP and messages can be lost. However, real-time applications like video games, which consider late messages as lost messages could be target applications. Also one of the reliable multicast protocols [5] may be used.

6.4.3 Fault tolerant computing

Here we suggest a protocol, which tries to make sure that even when some agents crash the number of agents is preserved.

Consider the above architecture with each agent having **k** (**k** is a small number greater than 1 say 2 or 3) copies and its own number. Each agent issues a heart beat message on its private channel with its number with a fixed beat time interval (T_{beat}). The agents will have an expiry timeout in case the beat is not received for a particular amount of time from a particular agent (T_{exp}). The agent with the next number in cyclic [**(n+1) mod k**] order generates a new agent with **id** number **n** on expiry of timer (T_{exp}). Although, each agent is listening for only the previous agent in the cyclic order but even if one agent is left the whole agent group will grow back again. Consider a group of three agents **1**, **2** and **3**. If any two die let's say **2** and **3** then **1** will spawn **3**, which will in turn eventually spawn **2** and hence the group will grow back again.

This makes sure with a good probability that we always have approximately **k** agents on each private channel.

6.5 Overcoming Agent Location Problem in absence of Multicast enabled routers

In large mobile agent systems agents frequently need to delegate services to other agents. This ability of mobile agents comes from their ability to communicate. This gives arise to the problem of locating a randomly roaming agent for message delivery. In the previous section we proposed an approach based on multicast which is highly efficient within a LAN but outside is dependent on network routers being capable of multicast.

In this section we propose another approach which is based on Linda blackboards and does not require any network router support.

The problem can be trivially solved if a home site is associated with an agent. As, when looking for a particular agent one can simply check at the home site where the agent is currently located and deliver the message to it, while the agent updates only its home site, every time before leaving for a new site. In this way even if the message arrives before the agent leaves the message is simply posted on the blackboard and when the agent arrives it can check for its messages.

The real problem arises in large multi-agent systems in which agents have different capabilities, agents receive tasks dynamically. In such a system agents can receive a task which they can not perform due to incompatible capability or expertise and need to locate other agents in their vicinity with required capability. In our approach as an agent moves from site to site minimally it posts its presence message on the blackboard (**out(i_am_here(MyID))**). As the agent is about to move it removes its entry (**in(i_am_here(MyID))**) and posts the next location it is going to (**out(nexthop(MyID, NextAdd))**). Hence, the agent leaves in some sense its trace. This trace may have mechanisms such as timeouts to insure that it is not greater than a definite length. The **ID** of the agent has certain capabilities associated with it. An agent would do an associative lookup on the blackboard for agents with a particular capability, obtain the **ID** of a capable agent with its trace on the blackboard, and start following the trace to catch-up with the capable agent to delegate a service or deliver a message.

There are two extreme approaches in the task of locating an agent based on how the agent performs updates as it moves. The agent could update whole of its path on a movement, which makes movement an expensive operation. Another extreme is that the agent updates only its local blackboard, which requires a traversal of its whole path by every message making message delivery an expensive operation. In our approach we hybridize the two extremes and share update operation task among agents and messages (message agents) and analyze the cost of update and search for delivery of a message.

In our proposal the agent only updates its local blackboard with its next hop information. The message (message agent) executes a (**cin(i_am_here(ID_other))**) which is an advanced Linda construct translating to an atomic (**rd(i_am_here(ID_other)) -> in(i_am_here(ID_other));false**). This makes sure message agent never blocks. In case it fails to find the agent it executes an (**in(nexthop(ID_other, NextAdd))**) and (**out(nexthop(ID_other, NextAdd))**) and departs to the next hop. If the message agent is able to find the other agent it holds the found agent (*as this agent can not move unless its ID is kept back on the blackboard*), communicates with the agent, and updates all the sites it had visited till now with the location information.

Let's now analyze the complexity of our approach.

(N) maximum distance of agent from current site (maximum possible length of trace of path).
(Tn) Time for movement of agent from one site to another (Large processing time is + time for movement)
(Tm) Time for movement of message from one site to another (Negligible processing

time + time for movement)

Assumption: $Tn > Tm$

(E) Extra Nodes traversed by agent once tracking starts:

$Tn * E = Tm * (E + N)$

$E = N/(Tn - Tm)$

Total nodes traversed by the message:

$= E + N$

$= N(1 + 1/(Tn - Tm)$

Update time of one node $Tu = Tm$ (Negligible processing time)

Total nodes searched $= E + N$

Total nodes updated $= E + N$

Total Time to search and Update $Tsu = 2 * Tm * N * (1 + 1/(Tn - Tm))$

After the update operation if the agent has moved K places and a message originating at one of the updated places wants to find the agent then its

$Tsu = 2 * Tm * K * (1 + 1/(Tn - Tm)) order of K$

If a message comes after the agent has moved another K places then there are two possibilities either the message is from one of the places updated in the current step or in the previous step. In the worst case

$Tsu = 2 * Tm * (K + 1) * (1 + 1/(Tn - Tm))$ order of $K + 1$

After S such steps the worst case

$Tsu = 2 * Tm * (K + S) * (1 + 1/(Tn - Tm))$ order of $K + S$

If N is the maximum number of nodes (Path Length) then we know that K is the number of nodes it visits in every step and S is the number of steps thus

$K * S < N$.

In the best case $K = S$ (approx.) then $K + S < 2 * sqrt(N)$ and

$Tsu = 4 * Tm * sqrt(N) * (1 + 1(Tn - Tm))$.

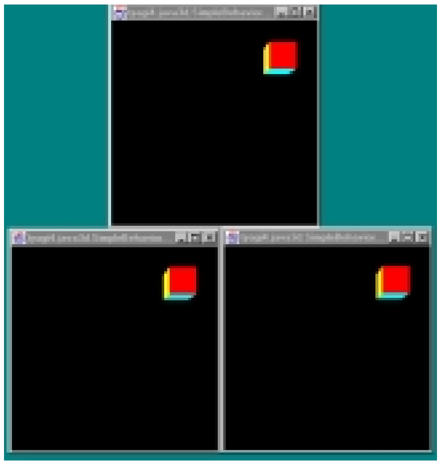


Figure 6.2: Synchronized Java3d worlds using multicast.

In the worst case messages are sent after every **N** moves of the agent **(K=N)**, they have to trace all **N** nodes to find the agent (In this case messages are so infrequent that it maybe unimportant). Also, if messages are sent after every move **(K=1)**, from a recently updated node, the nexthop information does not propagate and is only at the previous node. This means a search beginning at the start of chain will need to go through all **N** nodes (but the probability of this should become low as **S** increases).

The code for experimentation with the agent location problem is available from [45]. It also shows how simple it is to experiment with mobile agent algorithms with the Jinni's Mobile Agent Infrastructure.

6.6 Java3D based Shared Virtual Reality

The three concepts of *intelligent logic programmable agents*, *multicast single step synchronization* and *Java3D visualization* provide an interesting synergy for game programming. We will now explore an implementation architecture we have prototyped on Jinni 2000's multicast layer.

6.6.1 The User's interface

The user interface is based on shared Java3D virtual worlds.
Each user can join at anytime by joining a given multicast group.
Each user can create and own objects, which he/she/it can manipulate.
The user is opaque to the knowledge if he/she/it is playing against/with another user or an intelligent agent.

The implementation The main advantage we have in our implementation is that there is no centralized server. The application is completely distributed. If one user's machine goes down only the objects controlled by him/her go down. This is achieved by having the state being multi-casted to all users and stored only on the local blackboards from where it is to be collected when a new user logs in. The next subsection describes a basic Java3D API on which the virtual world is built and the interface is provided.

6.6.2 The basic API

java3d_init initializes and opens Java3d window and joins the multicast group and collects (**remote_mul_run(mul_all(a(X)))**) current state from the blackboards of other users.
new_obj creates a new object and puts the state on local blackboard (**out(a(X))**)

and multicasts the predicate call to currently subscribed users.
move_obj moves the object if owned and modifies (**in(a(X)),out(a(Y))**) the state on local blackboard and multicasts the predicate call to currently subscribed users.
remove_obj removes the current object clears entry (**in(a(X))**) from local blackboard and multicasts the change to currently subscribed users.

The blackboards preserve the current state. The multicasting makes sure all updates are single step. The agent scripts are written in Prolog and the visualization is based on Java3D. The logic of agents can be changed and different agents can have different personalities as per the learning logic, algorithm and experience of the agent. The agents generate keyboard and mouse events to play with humans (making them more similar to human interaction).

The Fig. 2 shows three multicast synchronized Java3D worlds, running under our prototype, in three process windows. In a real game they are distributed over the network.

6.7 Tele-teaching

A set of intelligent agents on student machines, join the multicast group of a teaching server (**run_mul_server**).

The agents can always be given required information or 'todo' tasks from the server as needed on the multicast channel (**remote_mul_run(out(a(X)))**).

The server can collect responses posted on the local blackboards by the agents with the extended blackboard concept (**remote_mul_run(mul_all(a(X))))** .

The application is more suited to present circumstances as most routers are incapable of multicast. It is however easy to ensure that classrooms are on a single LAN capable of multicast. The main advantage here is that even though the system is interactive the model is not *message based* - *query/response*. The agents are reactive and intelligent and the responses/queries are posted on the local blackboard from which the server can collect periodically or be informed to collect after a certain time. The model is flexible and can be extended and made more flexible by adding unicast channels and subset multicast groups for teamwork in students.

CHAPTER 7

RELATED WORK

7.1 Mobile Agents Architectures

Here we discuss certain other mobile agent architectures, their approaches to agent mobility, distibuted computing, code and computation mobility etc.

An interesting distributed mobile agent architecture we consider is Mozart [46]. The Mozart system also provides network transparency. It allows remote references to objects, functions and variables and other entities. Different entities have different protocols for maintaining consistency of remote access. It has certain similar concepts such as holding or locking of an entity unless its state is updated at remote site or multicasting of updates at remote sites [46].

An interesting abstraction for mobile computations with regards to agents, channels and mobile network components is described in [1]. This explores a unified framework for various difficulties in mobile computing and computations.

An important number of early software agent applications are described in [23] and, in the context of new generation networking software, in [24, 25].

Mobile code/mobile computation technologies are pioneered by General Magic's Telescript (see [11] for their Java based *mobile agent* product) and IBM's Java based Aglets [10]. Other mobile agent and mobile object related work illustrate the rapid growth of the field: [26, 14, 27, 28, 29, 30, 31]

Implementation technologies for mobile code are studied in [32]. Early work on the Linda coordination framework [15, 12, 33] has shown its potential for coordination of multi-agent systems. The logical modeling and planning aspects of computational Multi-Agent systems have been pioneered by [17, 18, 21, 22, 19, 20, 34, 16].

7.2 Reflection based Language Extensions

Here we discuss a few other approaches followed for interfacing Prolog to Java using reflection or the Java Native Interface (JNI), and also a Scheme interface to Java using reflection. First, Kprolog's JIPL package provides an interesting API from Java to a C-based Prolog and has a more extensive API for getting and setting fields. It also maps C-arrays to lists. The Kprolog's JIPL has dynamic type inference for objects, but the problem of method signature determination and overloading has not been considered in the package [42].

SICStus Prolog actually provides two interfaces for calling Java from Prolog. One is the JASPER interface which uses JNI to call Java from a C-based Prolog. To obtain a method handle from the Java Native Interface requires to specify the signature of the method explicitly. So JASPER requires the user to specify as a string constant the signature of the method that the user wishes to call. This transfers the burden of finding the correct method to the user [43], who therefore needs to know how to specify (sometimes intricate) method signatures as Strings.

SICStus Prolog also has another interesting interface for calling Java from Prolog as a Foreign Resource. When using this interface the user is required to first declare the method which he wants to call and only then can the user invoke it. Declaring a method requires the user to explicitly state the ClassName, MethodName, Flags, and its Return Type and Argument Types and map it to a Prolog predicate. Now the Prolog predicate can be used directly. This feature makes the Java method call look exactly like a Prolog builtin predicate at runtime - which keeps the underlying Java interface transparent to, for instance, a user of a library. (*This is very much similar to our old Builtin Registration and Execution mechanism, with one difference: here registration or declaration is on the Prolog side, while we were doing the same on Java side - for catching all errors at compile time.*) The interface still requires the programmer to explicitly specify types and other details as the exact method signature [43].

Kawa Scheme also uses Java reflection to call Java from Scheme. To invoke a method in Kawa Scheme one needs to specify the class, method, return type and argument types. This gives a handle to call the method. Now the user can supply arguments and can call this method. Again, the burden of selecting the method is left to the user as he specifies the method signature [41].

In our case, like JIPL and unlike other interfaces, we infer Java types from Prolog's dynamic types. But unlike JIPL, and like with approaches explicitly specifying signatures, we are able to call methods where the argument type is not exactly same as the parameter type. Hence, our approach mimics Java exactly. The functionality is complete and the burden of specifying argument types is taken away from the user.

CHAPTER 8

FUTURE WORK AND CONCLUSION

8.1 Future Work

8.1.1 Reflection based Prolog Extension

Future work includes extending our API, as currently we do not support getting and setting fields and arrays. Another interesting direction which is a consequence of the development of a reflection based API, is the ability to quickly integrate Java applications. We have shown the power of the API with the simple GUI application. Such applications can be built either completely in Java with an API based on methods to be called from Prolog, or almost completely in Prolog using only the standard packages of Java.

Jinni 2000 has support for *plugins* such as different Network Layers (TCP/multicast/CORBA) and a number applications such as Teleteaching, Java3D animation toolkit developed with its conventional builtin interface. New applications and plugins can now be added by writing everything in Prolog while using various Java libraries. Arguably, the main advantage of such an interface is that it requires a minimal learning effort from the programmer.

8.1.2 Multicast Protocols

There are some inherent problems with multicast. The protocol is UDP based (probably because it is not a great idea for each receiver to send an acknowledgment to each sender and flood the network). Also one can never know how many users are currently subscribed to a group. This makes blocking reads (**in(a(X))**) impossible, as we do not know how many responses to loop for. Currently we have implemented multicast **outs**, which do not require responses and non blocking multicast reads (**mul_all**), which collect responses from remote sites and respond on a unicast channel. Some possible scenarios for experimentation would be first matching response or first (k) matching response. Also currently multicast application remains untested on the Internet, as we are confined to the Ethernet LAN. We have overcome the problem of locating randomly moving mobile agents without multicast enabled routers, but with the new generation routers capable of multicast, it would be interesting to test the applications and protocols over larger domains. The unreliability in the protocol makes it unsuitable for some applications. Our multicast agents work well in real time applications for which a delayed packet is equivalent to a lost packet. Some future work would depend on implementation of reliable multicast protocols and its impact assuming that Internet routers will become more and more multicast aware.

8.2 Conclusion

We have described a new reflection based Prolog to Java interface which takes advantage of implicit dynamic type information on both the Prolog and the Java sides. Our interface has allowed to automate data conversion between overloaded method parameters, through a new algorithm which finds the most specific method corresponding to a Prolog call. The resulting run-time reflective method dispatching mechanism provides accurate handling of overloaded methods beyond the reflection package's limitations, and is powerful enough to support building a complete GUI library mostly in Prolog, with only a few lines of application specific Java code.

The ideas behind our interfacing technique are not specific to Jinni 2000 - they can be reused in improving C-based Prolog-to-Java interfaces like JIPL or Jasper or even Kawa's Scheme interface. Actually our work is reusable for any languages with dynamic types, interfacing to Java, as our work can be seen as just making Java's own Reflection package more powerful.

In the agent-to-agent delegation mechanism we proposed a simple alternative to code fetching mechanism for stronger mobility of agents. The alternative is specially well suited for low bandwidth networks and systems with security restrictions on execution of code. The concept of delegating code to home-site can be further explored with delegation to other agents and automated delegation of heavy computational jobs to more powerful servers.

We have outlined here an extension of Jinni with a transport layer using multicast sockets. We have also shown some interesting properties of multicast, which have opened various new possibilities for mobile agents and mobile computers. We have also described a blackboard based algorithm for locating a randomly roaming agent for message delivery. We are currently working on certain applications, which we have shown can be greatly simplified, speeded up and improved with multicast extended version of Jinni. We suppose that the possibilities and applications we have shown here is only a starting point for an unusual niche for Logic Programming based software tools. The spread of multicast technology from simple LANs to the complete Internet and the development of reliable protocols for multicast [5] will necessitate further exploration, to achieve greater insights on mobile agent technology and realize its full potential and possible impact.

BIBLIOGRAPHY

[1] "Abstractions for Mobile Computation", Luca Cardelli, Technical Report, Microsoft Research, apr, 1999

[2] "CPU Performance and Multicast", Don Mcgregor, Technical Report, Naval Postgraduate School Computer Science Department,
http://www.stl.nps.navy.mil/ mcgredo/projectNotebook/mcast/ethernet.html

[3] "Mobile Agents: Are They a Good Idea?", David Chess, Colin Harrison, Aaron Kershenbaum, Technical Report, IBM Research Division, T. J. Watson Research Center", 1995

[4] "Multicast routing in a datagram internetwork", Deering, S., PhD Thesis, Stanford University, Dec, 1991

[5] "Reliable Multicast Goes Mainstream", Kenneth P. Birman, Technical Report, Dept. of Computer Science, Cornell University, 1999,
http://www.cs.odu.edu/ mukka/tcos/e_bulletin/vol9no2/birman.html

[6] "Telecommunications:Glossary of TeleCommunication terms:Federal", National Communication System Technology and Standard Division, Standard 1037C, Technical Report, General Service Administration Information Technology Service, aug, 1996,
http://www.its.bldrdoc.gov/fs-1037/dir-023/_3404.htm

[7] "High-Level Networking with Mobile Code and First Order AND-Continuations", Paul Tarau and Veronica Dahl, Theory and Practice of Logic Programming, jan, 2001, Cambridge University Press

[8] ”A Logic Programming Based Software Architecture for Reactive Intelligent Mobile Agents”, Paul Tarau, Editors: Van Roy, P. and Tarau, P., Proceedings of DIPLCL’99, nov, 1999,
http://www.binnetcorp.com/wshops/ICLP99DistInetWshop.html

[9] ”Intelligent Mobile Agent Programming at the Intersection of Java and Prolog”, Paul Tarau, Proceedings of The Fourth International Conference on The Practical Application of Intelligent Agents and Multi-Agents, London, U.K.”, 1999, pg 109-123

[10] ”Aglets”, IBM, 1999,
http://www.trl.ibm.co.jp/aglets

[11] ”Odissey”, GeneralMagicInc., 1997,
http://www.genmagic.com/agents

[12] ”Enhancing Coordination and Modularity Mechanisms for a Language with Objects-as-Multisets”, S. Castellani and P. Ciancarini, Proc. 1st Int. Conf. on Coordination Models and Languages, April, 1996, Cesena, Italy, LNCS, 1061, pg 89-106, Editor:P. Ciancarini and C. Hankin, Springer, coordination language, mine

[13] ”Mobile Computation”, Luca Cardelli, Mobile Object Systems - Towards the Programmable Internet, 1997, Editor: Vitek, J. and Tschudin, C., pg 3-6, Springer-Verlag, LNCS, 1228

[14] ”Agent Tcl: A flexible and secure mobile agent system”, R. S. Gray, Proceedings of the Fourth Annual Tcl/Tk Workshop, pg 9-23, jul, 1996,
http://www.cs.dartmouth.edu/ agent/papers/tcl96.ps.Z

[15] "Linda in Context", N. Carriero and D. Gelernter, CACM, 32, 4, 1989, pg 444-458

[16] "Hierarchical Models and Communication in Multi-Agent Environments", B. Chaib-draa and P. Levesque, MAAMAW94, 119-134, aug, 1994, Odense, Denmark

[17] "Elements of a Plan Based Theory of Speech Acts", P. R. Cohen and C. R. Perrault, CogSci, 3, pg 177-212, 1979

[18] "Trial by Fire: Understanding the Design Requirements for Agents in Complex Environments", P. R. Cohen and M. L. Greenberg and D. M. Hart and A. E. Howe, AIMag, 10, 3, pg 32-48, 1989

[19] "An Open Agent Architecture", P. R. Cohen and A. Cheyer, Software Agents — Papers from the 1994 Spring, Symposium (Technical Report SS–94–03), Editor:O. Etzioni, AAAIP, pg 1-8, mar, 1994

[20] "Communicative Actions for Artificial Agents", P. R. Cohen and H. J. Levesque, ICMAS95, pg 65-72, jun, 1995

[21] "A Metalogic Programming Approach to Multi-Agent Knowledge and Belief", R. Kowalski and J.-S. Kim, AI and Mathematical Theory of Computation: Papers in Honour of John McCarthy, Editor:V. Lifschitz, AP, 1991

[22] "The Logical Modelling of Computational Multi-Agent Systems", M. Wooldridge, Department of Computation, UMIST, Manchester, UK, oct, 1992, Also available as Technical Report MMU–DOC–94–01, Department of Computing, Manchester Metropolitan University, Chester St., Manchester, UK

[23] "Software Agents", Editor:Jeffrey Bradshaw, AAAI Press/MIT Press, 1996, Menlo Park, Cal.

[24] "Towards an Active Network Architecture", David L. Tennenhouse and David J. Wetherall, Computer Communication Review, apr, 1996, 26, 2, ftp://ftp.tns.lcs.mit.edu/pub/papers/ccr96.ps.gz

[25] "A Characterization of Mobility and State Distribution in Mobile Code Languages", Gianpaolo Cugola and Carlo Ghezzi and Gian Pietro Picco and Giovanni Vigna", pg 10-19, ecoop-mos2, jul, 1996, Linz, Austria, http://www.polito.it/˜picco/papers/dpunkt.ps.gz

[26] "Object and Native Code Thread Mobility Among Heterogeneous Computers", Bjarne Steensbaard and Eric Jul", pg 68-78, sosp15, dec, 1995, Copper Moutain, Co., ftp://ftp.research.microsoft.com/users/rusa/sosp95.ps

[27] "Agent Tcl: A flexible and secure mobile-agent system", Robert S. Gray, Dartmouth College, Computer Science, Hanover, NH, PCS-TR98-327, jan, 1998, also Ph.D. Thesis, June 1997, ftp://ftp.cs.dartmouth.edu/TR/TR98-327.ps.Z

[28] "Agent TCL: Targeting the Needs of Mobile Computers", David Kotz and Robert Gray and Saurab Nog and Daniela Rus and Sumit Chawla and George Cybenko, IC, jul/aug 1997, 1, 4, pg 58-67, http://computer.org/internet/ic1997/w4058abs.htm,

[29] "A Comparison of Mobile Agent Migration Mechanisms", D. Eric White, jun 1998, mobile agent, Senior Honors Thesis, Dartmouth College,

[30] "Telescript Technology: Mobile Agents", James E. White, Also available as General Magic White Paper, http://www.genmagic.com/agents/Whitepaper/whitepaper.html

[31] "Mole – A Java Based Mobile Agent System", Markus Straer and Joachim Baumann and Fritz Hohl, pg 28-35, ecoop-mos2, jul, 1996, Linz, Austria, http://www.informatik.uni-stuttgart.de/ipvr/vs/Publications/1996-strasser-01.ps.gz

[32] "Efficient and Language-independent Mobile Programs", Ali-Reza Adl-Tabatabai and Geoff Langdale and Steven Lucco and Robert Wahbe, pg 127-136, Proceedings of the ACM SIGPLAN '96 Conference on Programming Language Design and Implementation (PLDI), Philadelphia, Pa., http://www.cs.cmu.edu/afs/cs.cmu.edu/user/ali/www/pldi96-omniware.ps

[33] "The Concurrent Language, Shared Prolog", A. Brogi and P. Ciancarini, 1991, pg 99-123, TOPLAS, 13, 1

[34] "Foundations of a Logical Approach to Agent Programming", Y. L'esperance and H. J. Levesque and F. Lin and D. Marcu and R. Reiter and R. B. Scherl, ATAL95, Editor: ATAL95editors, pg 331-346", 1996

[35] "The Java Language Specification", James Gosling and Bill Joy and Guy Steele, July 1996
http://java.sun.com/docs/books/jls

[36] "Inference and Computation Mobility with Jinni", P. Tarau, Editor: K. Apt and V. Marek and M. Truszczynski, The Logic Programming Paradigm: a 25 Year Perspective, pg 33–48. Springer, 1999
ISBN 3-540-65463-1

[37] "Partial Translation: Towards a Portable and Efficient Prolog Implementation Technology", P. Tarau and K. De Bosschere and B. Demoen, Journal of Logic Programming, 29(1–3):65–83, Nov. 1996

[38] "The Java Virtual Machine Specification", F. Yellin and T. Lindholm, Java Series, Publisher: Addison-Wesley, 1999
ISBN:0-201-43294-3

[39] "Fluents: A Refactoring of Prolog for Uniform Reflection and Interoperation with External Objects", P. Tarau, Editor: J. Lloyd, Proceedings of CL'2000, London, July 2000

[40] "Overloading and Inheritance in java", D. Ancona and E. Zucca, and S. Drossopolou, Technical report, DISI University of Genova, Department of Computing Imperial College, 1999
ftp://ftp.disi.unige.it/pub/person/AnaconaD/DISI-TR-99-14.ps.gz

[41] "Kawa, the Java based Scheme system", P. Bothner, Technical report, 1999
http://www.delorie.com/gnu/docs/kawa/kawa_toc.html

[42] "Plc.java: JIPL class source", N. Kino, KLS Research Labs, 1997
http://www.kprolog.com/jipl/index_e.html

[43] "Manual document of SICStus Prolog", SICStus Support, Swedish Institute of Computer Science, May 2000
http://www.sics.se/isl/sicstus/docs/latest/html/docs.html

[44] "Java 2 platform, standard edition, v 1.3, api specification", Sun, Technical report, 2000
http://java.sun.com/j2se/1.3/docs/api/index.html.

[45] "Source code for agents"
http://www.cs.unt.edu/home/tyagi/agent.pl

[46] "Mobile Objects in Distributed Oz" P. Van Roy and S. Haridi and P. Brand and G. Samolka and M. Mehl and R. Scheidhauer, ACM Transactions on Programming Language Systems, Vol. 19, No. 5, Sept 1997, Pg 805-852