Multicast Protocols for Jinni Agents

Satyam Tyagi, Paul Tarau, and Armin Mikler

Department of Computer Science
University of North Texas
P.O. Box 311366
Denton, Texas 76203
E-mail: {*tyagi,tarau,mikler*}*@cs.unt.edu*

Abstract. We extend the Jinni Agent Programming Infrastructure with a multicast network transport layer. We show that the resulting protocol emulates client/server exchanges while providing high performance multicasting of remote calls to multiple sites. To ensure that our agent infrastructure runs efficiently independently of router-level multicast support, we also describe a blackboard based algorithm for locating a randomly roaming agent for message delivery. Jinni's orthogonal design, separating blackboards from networking and multi-threading, turns out to be easy to adapt to support a generalization of Jinni's Linda based coordination model. The resulting system is particularly well suited for building large scale, agent based, IP transparent, fault tolerant, tele-teaching and shared virtual reality applications.

Keywords: *Mobile Computations, Intelligent Mobile Agents, Logic Programming, Blackboard based Coordination, Networking Software Infrastructure, Multicast, Java3D based Shared Virtual Reality, IP Transparent Communication*

1 Introduction

The advent of networked, mobile, ubiquitous computing has brought a number of challenges, which require new ways to deal with increasingly complex patterns of interaction: autonomous, reactive and mobile computational entities are needed to take care of unforeseen problems, to optimize the flow of communication, to offer a simplified, and personalized view to end users. These requirements naturally lead towards the need for *agent programs* with increasingly sophisticated inference capabilities, as well as autonomy and self-reliance. To host them with minimal interference to their main mission, we need a software infrastructure providing a minimal basic ontology - ideally as simple and self contained as the IP packet universe on top of which the Internet is built.

Conveniently encapsulated as *agents*, software artifacts enabled with autonomy, dynamic knowledge exchange and network transparent mobility (as key features) have emerged.

An important number of early software agent applications are described in [3] and, in the context of new generation networking software, in [30, 14].

Mobile code/mobile computation technologies are pioneered by General Magic's Telescript (see [17] for their Java based *mobile agent* product) and IBM's Java based Aglets [20]. Other mobile agent and mobile object related work illustrate the rapid growth of the field: [24, 18, 19, 21, 31, 32, 25]

Implementation technologies for mobile code are studied in [1]. Early work on the Linda coordination framework [7, 8, 4] 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 [13, 11, 22, 33, 10, 12, 23, 9].

Jinni 2000 is a multi-threaded Java based Prolog system with modular, plugin networking layer, designed as an agent programming library, through a combination of Java and Prolog components. It supports mobile agents either directly through a mobile thread abstraction or as a combination of remote predicate calls, client/server components, multiple networking layers and blackboards.

1.1 Overview of the paper

The paper is divided into 8 sections. Section 2 describes the Jinni's Mobile Agent Architecture briefly and some of Jinni's functionalities and advantages. Section 3 introduces us to the recently developed Multicast layer for Jinni and some of its advantages. Section 4 outlines the Synchronization of Multicast Agents with Blackboards. Section 5 explores some important properties of multicast networks, which are significant for Jinni in particular, and mobile agents and mobile computing in general. In section 6 we describe two applications for Multicast Agents using the Jinni Agent Programming Infrastructure. Section 7 discusses some of the problems and related areas to be explored in future. Finally we conclude with section 8 stating current achievements, ongoing applications and new possibilities.

2 The Jinni Architecture

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 [26, 27].

2.3 Data, Code and Computation Mobility

While *data* and *code* mobility present no challenge in a Java environment, migrating the state of the computation from one machine or process to another still requires a separate set of tools. Java's remote method invocations (RMI) add transparent control mobility and a (partially) automated form of *object mobility* i.e. integrated code (class) and data (state) mobility.

Mobility of live code is called *computation mobility* [5]. It requires interrupting execution, moving the state of a runtime system (stacks, for instance) from one site to another and then resuming execution. Clearly, for some languages, this can be hard or completely impossible to achieve (C/C++) while in other languages like Java it still requires class specific serialization methods (providing writing and reading of objects to/from byte streams).

Conventional mobile code systems like IBM's Aglets [20] require serialization hints from the programmer and do not implement a fully generic reflective computation mobility infrastructure. Aglets do not provide code mobility as they assume that code is already available at the destination site. In practice this means that the mobile code/mobile computation layer is not really transparent to the programmer.

In contrast, our architecture is based on building an autonomous layer consisting of a reflective interpreter, which provides the equivalent of implicit serialization and supports orthogonal transport mechanisms for data, code and computation state. The key idea is simply that by introducing interpreters spawned as threads by a server at each networked site, *computation mobility* at object-level is mapped to *data mobility* at meta-level in a very straightforward way. A nice consequence is transport independence coming from the uniform representation of data, code and computation state (in practice this means that Corba, RMI, HLA or plain/multicast sockets can be used interchangeably as a transport mechanism).

2.4 Advantages of Computation Mobility

Current agent programming infrastructures use message passing as the key communication mechanism. Existing Concurrent Constraint Logic Programming languages (and multi-paradigm languages like Distributed Oz) support distributed programming and coordination through monotonic stores and shared logical variables.

By contrast, we want to explore the impact of mobile live computations, lightweight multi-engine/multi-threaded script interpreters, blackboard constraints and multicast based coordination on building mobile multi-agent systems. Our distributed Linda blackboards generalize concurrent constraint programming stores by allowing non-monotonic updates of assertional constraints [27]. With our multicast layer we implement transparently replication of shared blackboards distributed over different places.

We refer to [26, 15] for more detailed discussion on the advantages of computation mobility.

3 A Multicast Layer for Jinni

3.1 Multicast

Multicast is a technique that allows data, including packet form, to be simultaneously transmitted to a selected set of destinations on a network. 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 [29].

The key concepts of Multicast Networks are described in [16]. 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.

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).

3.2 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.

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_server, which was there for unicast servers is extended to
run_mul_server defaults to **Host = 224.1.1.1, Port = 7001**
run_mul_server(Port) defaults to **Host = 224.1.1.1**
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_run(G), which was there for unicast requests is extended to
remote_mul_run(G) which defaults to **Host = 224.1.1.1, Port = 7001**
remote_mul_run(Host, Port, AnswerPattern, Goal, Password, Result)

Our set of multicast servers runs by listening for packets on their group and by responding back on the group's multicast address. Clients listening on this *multicast 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.

4 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[7]

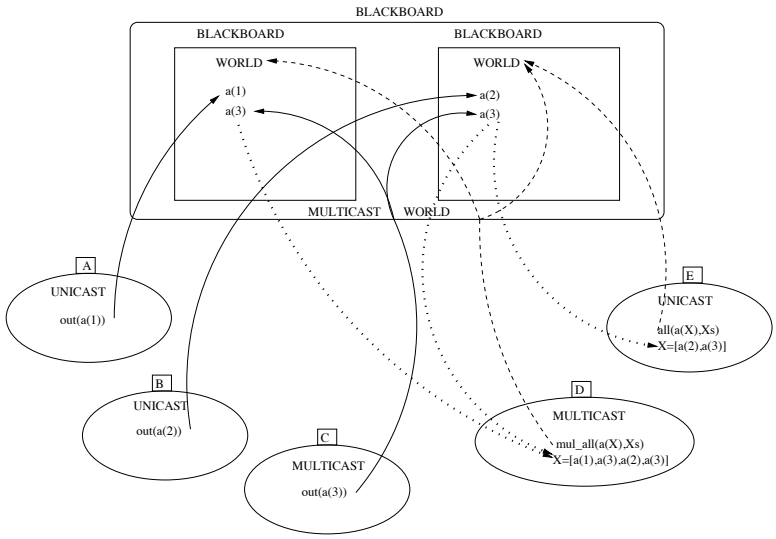

Fig. 1. Basic Linda operations with Multicast

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.

5 Some Properties and their 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.

5.1 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

[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."

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 [15].

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**[28]. 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),
  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 [28].

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 [6].

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.

5.2 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 [2] may be used.

5.3 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 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 in case we assume a home site associated with an agent. In this case, when looking for a particular agent we simply check at the home site where the agent is currently located and deliver the message to it, while the agent updates it's 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 it's messages.

The real problem arises in large multi-agent systems in which agents have different capabilities and agents receive tasks dynamically and an agent can a receive a task which it can not perform due to incompatible capability or expertise and needs to locate another agent in it's vicinity who can. In our approach as an agent moves from site to site minimally it posts it's presence message on the blackboard (**out(i_am_here(MyID))**). As the agent is about to move it removes it's 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 it's trace. This trace may have mechanism 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. Another agent could do an associative lookup on the blackboard for agents with a particular capability get the **ID** of any agent with a trace on the blackboard and start following the trace to catchup with the agent to delegate a service or deliver a message.

There could be two extremes in the task of locating an agent depending on how the agent performs updates as it moves. The agent could update whole of it's path on a movement, which makes movement an expensive operation. Another extreme is that the agent updates only it's local blackboard, which requires a traversal of it's 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 agents and analyze the cost of the update operation, and search for delivery of a message.

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

Let's now analyse 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 (Processing=large + movement)
(Tm) Time for movement of message from one site to another (Processing =0 + movement)
Assumption: Tn ¿ Tm
(E) Extra Nodes traversed by agent once message starts tracking
Tn * E = Tm * (E + N)
E = N/(Tn - Tm)
Total nodes traversed by the message
= E + N
= N(1 + 1/(Tn - Tm)
Tu Update time of one node Tu = Tm (Processing =0)
Total nodes searched = E + N
Total nodes updated = E + N
Total Time to search and Update = 2 * Tm * N * (1 + 1/(Tn - Tm))

After 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 it's
Total Time to search and Update = 2 * Tm * K * (1 + 1/(Tn - Tm)) order of K

Now if a message comes after agent has moved another K places. then we have two possibilities either it is from one of the places updated in the current step or previous step. In the worst case
Total Time to search and Update = 2 * Tm * (K + 1) * (1 + 1/(Tn - Tm)) order of K + 1

After S such steps the worst case is
Total Time to search and Update = 2 * Tm * (K + S) * (1 + 1/(Tn - Tm)) order of K + S

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

K*S < N
if we have a good case K = S (approx.) then K+S ¡ 2*sqrt(N)
Time to search and update = 4 * Tm * sqrt(N) * (1 + 1(Tn - Tm))

The worst case messages are sent after **N** moves **(K=N)** of the agent 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)** and we are sending a message from just updated node the nexthop information is only at the previous node if we have to search for the agent from any node the complexity becomes **N** (but the probabilty of this should become low as **S** increases).

The code for experimentation with the agent location problem appears in the appendix. It also shows how simple it is to experiment with mobile agent algorithms with the Jinni's Mobile Agent Infrastructure.

7 Applications

We will now overview a number of applications likely to benefit from a multicast agent infrastructure.

7.1 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.

7.2 Java3D based Shared Virtual Reality

Fig. 2. *Synchronized Java3d worlds using multicast.*

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.

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.

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.

8 Some Problems and Future work

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. 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.

9 Conclusion

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 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 [2] will necessitate further exploration, to achieve greater insights on mobile agent technology and realize its full potential and possible impact.

References

1. A.-R. Adl-Tabatabai, G. Langdale, S. Lucco, and R. Wahbe. Efficient and Language-independent Mobile Programs. In *Proceedings of the ACM SIGPLAN '96 Conference on Programming Language Design and Implementation (PLDI)*, pages 127–136, Philadelphia, Pa., May 1996.
2. K. P. Birman. Reliable Multicast Goes Mainstream. Technical report, Dept. of Computer Science, Cornell University, 1999. Available from http://www.cs.odu.edu/ mukka/tcos/e_bulletin/vol9no2/birman.html.
3. J. Bradshaw, editor. *Software Agents*. AAAI Press/MIT Press, Menlo Park, Cal., 1996.
4. A. Brogi and P. Ciancarini. The Concurrent Language, Shared Prolog. *ACM Trans. Prog. Lang. Syst.*, 13(1):99–123, 1991.
5. L. Cardelli. Mobile Computation. In J. Vitek and C. Tschudin, editors, *Mobile Object Systems - Towards the Programmable Internet*, pages 3–6. Springer-Verlag, LNCS 1228, 1997.

6. L. Cardelli. Abstractions for Mobile Computation. Technical report, Microsoft Research, Apr. 1999.
7. N. Carriero and D. Gelernter. Linda in Context. *CACM*, 32(4):444–458, 1989.
8. S. Castellani and P. Ciancarini. Enhancing Coordination and Modularity Mechanisms for a Language with Objects-as-Multisets. In P. Ciancarini and C. Hankin, editors, *Proc. 1st Int. Conf. on Coordination Models and Languages*, volume 1061 of *LNCS*, pages 89–106, Cesena, Italy, April 1996. Springer.
9. B. Chaib-draa and P. Levesque. Hierarchical Models and Communication in Multi-Agent Environments. In *Proceedings of the Sixth European Workshop on Modelling Autonomous Agents and Multi-Agent Worlds (MAAMAW-94)*, pages 119–134, Odense, Denmark, Aug. 1994.
10. P. R. Cohen and A. Cheyer. An Open Agent Architecture. In O. Etzioni, editor, *Software Agents — Papers from the 1994 Spring Symposium (Technical Report SS–94–03)*, pages 1–8. AAAIP, Mar. 1994.
11. P. R. Cohen, M. L. Greenberg, D. M. Hart, and A. E. Howe. Trial by Fire: Understanding the Design Requirements for Agents in Complex Environments. *AI Magazine*, 10(3):32–48, 1989.
12. P. R. Cohen and H. J. Levesque. Communicative Actions for Artificial Agents. In *Proceedings of the First International Conference on Multi-Agent Systems (ICMAS-95)*, pages 65–72, San Francisco, CA, June 1995.
13. P. R. Cohen and C. R. Perrault. Elements of a Plan Based Theory of Speech Acts. *Cognitive Science*, 3:177–212, 1979.
14. G. Cugola, C. Ghezzi, G. P. Picco, and G. Vigna. A characterization of mobility and state distribution in mobile code languages. In *2nd ECOOP Workshop on Mobile Object Systems*, pages 10–19, Linz, Austria, July 1996.
15. A. K. David Chess, Colin Harrison. Mobile agents: Are they a good idea? Technical report, IBM Research Division, T. J. Watson Research Center, 1995.
16. S. Deering. *Multicast routing in a datagram internetwork*. PhD thesis, Stanford University, Dec. 1991.
17. GeneralMagicInc. Odissey. Technical report, 1997. available at http://www.genmagic.com/agents.
18. R. S. Gray. Agent tcl: A flexible and secure mobile agent system. In *Proceedings of the Fourth Annual Tcl/Tk Workshop*, pages 9–23, July 1996. http://www.cs.dartmouth.edu/ agent/papers/tcl96.ps.Z.
19. R. S. Gray. Agent tcl: A flexible and secure mobile-agent system. Technical Report PCS-TR98-327, Dartmouth College, Computer Science, Hanover, NH, Jan. 1998. Ph.D. Thesis, June 1997.
20. IBM. Aglets. Technical report, 1999. http://www.trl.ibm.co.jp/aglets.
21. D. Kotz, R. Gray, S. Nog, D. Rus, S. Chawla, and G. Cybenko. Agent TCL: Targeting the needs of mobile computers. *IEEE Internet Computing*, 1(4):58–67, July/August 1997.
22. R. Kowalski and J.-S. Kim. A Metalogic Programming Approach to Multi-Agent Knowledge and Belief. In V. Lifschitz, editor, *AI and Mathematical Theory of Computation: Papers in Honour of John McCarthy*. Academic Press, 1991.
23. Y. Lésperance, H. J. Levesque, F. Lin, D. Marcu, R. Reiter, and R. B. Scherl. Foundations of a Logical Approach to Agent Programming. In M. Wooldridge, J. P. Müller, and M. Tambe, editors, *Intelligent Agents II (LNAI 1037)*, pages 331–346. Springer-Verlag: Heidelberg, Germany, 1996.
24. B. Steensbaard and E. Jul. Object and native code thread mobility among heterogeneous computers. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, pages 68–78, Copper Moutain, Co., Dec. 1995.

25. M. Straer, J. Baumann, and F. Hohl. Mole – a Java based mobile agent system. In *2nd ECOOP Workshop on Mobile Object Systems*, pages 28–35, Linz, Austria, July 1996.
26. P. Tarau. A Logic Programming Based Software Architecture for Reactive Intelligent Mobile Agents. In P. Van Roy and P. Tarau, editors, *Proceedings of DIPLCL'99*, Las Cruces, NM, Nov. 1999. http://www.binnetcorp.com/wshops/ICLP99DistInetWshop.html.
27. P. Tarau. Intelligent Mobile Agent Programming at the Intersection of Java and Prolog. In *Proceedings of The Fourth International Conference on The Practical Application of Intelligent Agents and Multi-Agents*, pages 109–123, London, U.K., 1999.
28. P. Tarau and V. Dahl. High-Level Networking with Mobile Code and First Order AND-Continuations. *Theory and Practice of Logic Programming*, 1(1), Mar. 2001. Cambridge University Press.
29. N. C. S. Technology and S. Division. Telecommunications:Glossary of TeleCommunication terms:Federal Standard 1037C. Technical report, General Service Administration Information Technology Sevice, Aug. 1996. Available from http://www.its.bldrdoc.gov/fs-1037/dir-023/_3404.htm.
30. D. L. Tennenhouse and D. J. Wetherall. Towards an active network architecture. *Computer Communication Review*, 26(2), Apr. 1996.
31. D. E. White. A comparison of mobile agent migration mechanisms. Senior Honors Thesis, Dartmouth College, June 1998.
32. J. E. White. Telescript technology: Mobile agents. In Bradshaw [3]. Also available as General Magic White Paper.
33. M. Wooldridge. *The Logical Modelling of Computational Multi-Agent Systems*. PhD thesis, 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).

Appendix A:Prolog code for agent location problem

```
%the randomly moving agent
next_step(ID,Ports,Port):-
  %select randomly a port where I wish to go next
  select_port(Ports,Port,SelectPort),
  println(here(ID)),
  %if I have been here before I need to clean up the old value of next port
  (rd(been_here(ID))->
    in(nextport(ID,_Val))
    ;
    println(firstvisit(ID)),
    out(been_here(ID))
  ),
  out(i_here(ID)),
  %relax do processing
  sleep(10),
  %if i_here(ID) held by some tracking agent then wait for it
```

```
  in(i_here(ID)),
  % now leave the port of the next place to be visited
  out(nextport(ID,SelectPort)),
  println(gone(ID)),
  % goto nextport and do the same
  bg(remote_run(localhost,SelectPort,_,next_step(ID,Ports,SelectPort),none,_)).

%selects a randomly generated port
select_port(Ports,Port,SelectPort):-
  remove_item(Port,Ports,RealPorts),
  random(X),
  (X<0->
    X1 is X * -1
    ;
    X1 is X
  ),
  length(RealPorts,N),
  Nex is X1 mod N,
  Next is Nex + 1,
  nth_member(SelectPort,RealPorts,Next).

searchagent(ID,Port,List,Sorted):-
  sort(List,Sorted),
  (cin(i_here(ID))->
    %if agent is here hold it
    println(found(ID)),
    [Currport|Rest]=List,
    sort(Rest,UpdateList),
    %update all the sites traversed till now
    update(ID,Currport,UpdateList),
    %release the agent
    out(i_here(ID)),
    %actually track ends here
    sleep(50)
    ;
    println(searching(ID))
  ),
  println(Sorted),
  in(nextport(ID,Port)),
  out(nextport(ID,Port)),
  println(thenextport(Port)),
  sleep(1).

%tracks the agent with given ID and builds the list of nodes it has travelled
trackagent(ID,List):-
```

```
  searchagent(ID,Port,List,Sorted),
  bg(remote_run(localhost,Port,_,trackagent(ID,[Port|Sorted]),none,_)).

update(ID,Port,Sorted):-
  %update all ports found till now except the current place where we are
  member(P,Sorted),
  not(P=Port),
  println(P),
  bg(remote_run(localhost,P,_,the_update(ID,Port),none,_)),
  fail;true.

the_update(ID,Port):-
  in(nextport(ID,OldPort)),
  println(updated(OldPort,Port)),
  out(nextport(ID,Port)).
```