COMPARISON OF COMMUNICATION MODELS FOR MOBILE AGENTS

Xining Li

Department of Computing and Information Science
University of Guelph
Guelph, ON, Canada N1G 2W1

ABSTRACT

An agent is a self-contained process being acting on behalf of a user. A Mobile Agent is an agent roaming the internet to access data and services, and carry out its assigned task remotely. This paper will focus on the communication models for Mobile Agents. Generally speaking, communication models concern with problems of how to name Mobile Agents, how to establish communication relationships, how to trace moving agents, and how to guarantee reliable communication. Some existing MA systems are purely based on RPC-style communication, whereas some adopts asynchronous message passing, or event registration/handling. Different communication concepts suitable for Mobile Agents are well discussed in [1]. However, we will investigate these concepts and existing models from a different point view: how to track down agents and deliver messages in a dynamic, changing world.

Keywords: Mobile Agents, Communication models, Messenger, Logic Programming

1. INTRODUCTION

Mobile Agents are mainly intended to be used for network computing - applications distributed over large scale computer networks. An agent is a self-contained process being acting on behalf of a user. A Mobile Agent is an agent roaming the internet to access data and services, and carry out its assigned task remotely. Numerous Mobile Agents systems have been implemented or are currently under development. Typical systems are Aglets, Voyager, Odyssey, Concordia, Ajanta, ARA, Telescript, Tocoma, Mole, D'Agent, *etc.* Some MA systems are based on Java [3, 4, 5, 6, 7], and some are based on other object oriented programming languages or scripting languages [8, 9, 10]. Another mobile agent framework which embeds a logic programming component is pioneered by Distributed Oz[11] - a multi-paradigm language (functional, logic, object-oriented, and constraint), Jinni[12] - a lightweight, multi-threaded, Prolog-based language (supporting mobile agents through a combination of Java and Prolog components), and MiLog - a logic programming framework for mobile agents (implemented on top of JVM)[13].

Fundamental questions related to Mobile Agents paradigm are the support of agent mobility, communication and synchronization among agents, and security issues such as agent privacy and integrity, authentication, authorization, and access control.

This paper will focus on the communication models for Mobile Agents. Generally speaking, mobile agent communication include how to name Mobile Agents, how to establish communication relationships, how to trace moving agents, and how to guarantee reliable communication. Some existing MA systems are purely based on RPC-style communication, whereas some adopts asynchronous message passing, or event based communication. Different communication paradigms suitable for Mobile Agents are well discussed in [1][2]. However, we will investigate these paradigms and existing models from a different point view: how to track down agents and deliver messages in a dynamic, changing world.

Based on our survey, we will propose a new communication mechanism for Mobile Agents. This paradigm has been used in our on-going project IMAGO: Intelligent Mobile Agents Gliding On-line. IMAGO project consists of two major parts: MLVM - a multithreading logic virtual machine which presents a logic-based framework in the design space of Intelligent agent server framework and the IMAGO Prolog - an agent development kit based on Prolog which implements agent communication through *messengers* - special mobile agents dedicated to deliver messages on the network.

2. MODELS OF MA COMMUNICATION

Processes in a distributed system must interact with each other using some kinds of communication models to exchange data and coordinate their execution. Mobile agents are distributed processes, however, once they are invoked they will autonomously decide the host(s) they will visit and the tasks they have to perform. Their behavior is either defined explicitly through

the agent code or alternatively defined by an itinerary which is usually modifiable at runtime. Interprocess communication is dependent on the ability to locate the communication entities. This is the role of the name services in distributed systems. However, the mobility of agents makes it harder to provide such kind of service, because there is virtually no way to return a static location of a moving agent.

Several communication models have been widely used in distributed systems as well as most MA systems. Typical models are *message passing*, *remote procedure call (RPC)*, and *distributed event handling*.

Message passing is used to support peer-to-peer communication pattern and is the mostly adopted model in MA systems, such as Aglet, Mole, D'Agent, Voyager and *etc.* Aglet supports an object-based messaging framework that is location independent, extensible, rich, and both synchronous and asynchronous. However, Aglet API does not support agent tracking, instead, it leaves this problem to applications. Mole supports the (global) exchange of messages through a session-oriented mechanism. Agents that wants to communicate with each other, must establish a session before the actual communication can started. To avoid tracking agents during communication, Mole does not allow agents to move if they are involved in a session. D'Agent support text-based message passing. The location and identity (symbolic name or system assigned number) of the receiver should be known by the sender. Therefore, communication is lost as soon as one peer jumps to another location. Voyager implements message passing through the concept of virtual objects. Agents are special type of objects in a Voyager application. Communication with a remote object is handled by its virtual object which hides the remote location and acts as a reference of the remote object. When messages are being sent to the remote agent, virtual object forwards the message to the remote object and returns messages back if necessary. A virtual object keeps track of the remote object by its last known address. If the remote object moves from its last location, it will leave a *secretary* object behind to forward messages to its new location. The secretary object will be removed only if a returned message has been received by the corresponding virtual object. The advantage of Voyager is that it could automatically track down moving agents, however, it could cause a lot of overhead and delay if remote objects involve frequent movements.

RPC or RMI are commonly used paradigm in today's distributed programming. Since there is no distinction in syntax between an RPC and a local procedure call, the RPC provides access transparency to remote operations. Several MA systems support RPC/RMI paradigm, such as Mole and Voyager. It can be always argued whether agent communication should be remote or restricted to local, considering that the most attractive motivation of mobile agents is to move computation to the data rather than the data to the computation, and therefore avoid remote communication. A similar argument is that under the new paradigm of mobile agents, why we need RPC/RMI at all. Agents for Remote Action (ARA) [18] attempts to minimize the remote communication through meeting oriented paradigm. Ara provides client/server style interaction between agents. The core provides the concept of a service point which is the meeting point with a well known name where agents located at a specific place can interact as clients and servers through an RPC-like invocation on a local host.

Some MA systems have much in common with those event frameworks employed in GUI toolkits supported by Java and Tcl/Tk. The concept of event based communication and synchronization can be viewed as a sophisticated paradigm of meeting oriented agent coordination. Mobile agent systems such as Concordia, D'Agent, Mole, *etc.*, extends the event-driven programming technique to coordinate groups of mobile agents and achieve agents synchronization. In this paradigm, agent synchronization is achieved by the objects that are defined as active entities responsible for the coordination of an entire application or parts of it. These synchronization objects could be user defined objects or system implemented event manager. They are responsible to accept event registration, listens and receives events, and notifies interested parties of each event it receives. In this paradigm, agents participating in such groups is responsible to register a list of event types it is interested as well as the location it wishes events to be sent.

Another problem related with agent communication is that an agent developed in one system generally can not *talk* with an agent in another system. A solution is to provide a universal agent communication language and standards for the mobile agent community. Typical researches in this area are MASIF [16] and FIPA [17]. MASIF specifies a set of standards for agent naming, agent management, agent tracking, *etc*, mainly for homogeneous agents. Some existing MA systems, such as Aglets, will adopt MASIF in their future implementations. FIPA is a forum of 70+ international telecommunication companies and research institutes which specifies open standards focusing on languages and protocols for communication, coordination, and management of heterogeneous agents. In other words, FIPA is more like an abstract architecture which makes all FIPA-compliant systems to communicate and interact with each other through a common language: Agent Communication Language (ACL). Several (open source) im-

plementations have emerged from both industries and academic research groups.

3. OVERVIEW OF IMAGO PROLOG

IMAGO project consists of two major parts: MLVM - a multithreading logic virtual machine which presents a logic-based framework in the design of mobile agent server, and the IMAGO Prolog - an agent development kit based on Prolog which implements agent communication through *messengers* - special mobile agents dedicated to deliver messages on the network.

The goal of MLVM is to present an efficient logic virtual machine architecture in the design space of Intelligent Mobile Agents server. To achieve this, our design has to cope with new issues, such as explicit concurrency, code autonomy, communication/synchronization and computation mobility. At current stage, some practical issues, such as multithreading, garbage collection, code migration, communication mechanism, etc, have been specified, whereas some other issues, such as security, services, etc, will be investigated further.

IMAGO Prolog is a simplified Prolog with an extended Application Programming Interface (API) for Intelligent Mobile Agents. Briefly, imagoes are programs written in a variant of Prolog that can fly from one host on the Internet to another. That is, an imago is characterized as an entity which is mature (autonomous and self-contained), has wings (mobility), and bears the mental image of the programmer (intelligent agent).

An IMAGO Prolog program consists of a set of imago definitions and module definitions. Imago definitions serve to specify autonomous entities. An imago definition provides an implementation framework from which intelligent (mobile) agents can be created. A procedure defined in an imago must be a complete procedure and private to its name space. This means that such procedures are not accessable anywhere outside of the imago. Modules serve to partition the name space and support encapsulation for the purpose of constructing large applications from a library of smaller components. A module definition in IMAGO Prolog follows the specification of standard Prolog.

The IMAGO Prolog introduces imago directives to specify the body of an imago. Since all procedures defined in an imago belong to the imago privately, there is no interface required. There are three kinds of imagoes: *stationary imago*, *worker imago*, and *messenger imago*. An imago definition provides the framework (like a Java class) for creating instances of the imago (like Java objects of the class). For example, the worker directive *worker(buyer)*, where *buyer* is an atom giving the name of an imago definition, specifies that the Prolog text bracketed between this directive and the matching closing directive *end_worker(buyer)* belongs to the imago definition *buyer*.

```
:- worker(buyer).
        buyer(Arg) :-
              buyer_body, ...
:- end_worker(buyer).
```

Imago instances, *i.e.*, (mobile) agents, are created from imago definitions. From now on, we simply use *imago* to be synonymous with *imago instance*. Generally speaking, an imago is composed of three parts: its identifier which is unique to distinguish with others, its code which corresponds to a certain algorithm, its execution thread which is maintained by a single memory block (a merged stack/heap with automatic garbage collection)[14].

An agent application starts from a stationary imago. It looks like that the wings of a stationary imago have degenerated, so that it has lost its mobility. In other words, a stationary imago always executes on the host where it begins execution. However, a stationary imago has the privileges to access resources of its host machine, such as I/O, files, GUI manager, *etc.* A stationary imago can create worker or messenger imagoes, but it can not clone itself. We can find the similarity that there is only one queen in a colony of bees. An imago Prolog application must contain one and only one stationary imago in its context. The following gives a minimum Imago application.

```
:- stationary(foo).
        foo(_).
:- end_stationary(foo).
```

Worker imagoes are created by the stationary imago of an application. A worker imago is able to move such that it looks like a worker bee flying from place to place. A worker imago can clone itself. A cloned worker imago is an identical copy of the original imago but with a different identifier. A worker imago can not *create* other worker imagoes, however, it may launch messenger imagoes (system built-in imagoes) to deliver messages. When a worker imago moves from one host to another, it continues its execution on the destination host at the instruction which immediately follows the invocation of the move predicate. As mobile agents are a potential threat to harm the remote hosts that they are visiting, the IMAGO system enforces a tight access control on worker imagoes: they have no right to access any kind of system resources except the legal services provided by the server. A messenger queue is associated with each worker imago which holds all attached messenger imagoes waiting to deliver messages. Names (identifiers) of worker imagoes must be presented at the

time they are created, and are immutable throughout execution.

The IMAGO Prolog API consists of a set of primitives that allows programmer to create mobile agent applications [15]. Like other mobile agent systems, we provides primitives for agent management (creation, dispatching, migration), agent communication and synchronization, agent monitoring (query, recall, termination), *etc.* In addition, IMAGO system explores a novel communication model: instead of passing messages among agents through simple send/receive primitives, the IMAGO implements agent communication through "messengers" - special mobile agents dedicated to deliver messages on the network.

Like other logic programming systems, IMAGO Prolog Application Programming Interface is presented as a set of builtin predicates. This set consists of builtin predicates common to most Prolog-based systems and new builtin predicates extended for mobile agent applications. IMAGO Prolog API predicates are context sensitive, *i.e.*, the eligibility and effect of such predicates depend on the calling context in which they are activated. In general, the usage of agent management predicates depends on the type of imagoes. Table 1 shows a brief list of predicates legal to each imago type.
.

Imago Type	**Builtin Predicates**
stationary imago	create, accept, wait_accept, dispatch, terminate, workers
worker imago	move, clone, accept, dispatch, wait_accept, dispose
messenger imago	move, clone, attach, dispose

Table 1: Builtin Predicates for Imagoes

In principle, all these predicates are not re-executable. Furthermore, they can be used in imago definitions only, that is, invocation of agent predicates is not allowed in any modules.

4. MESSENGERS

Messenger imagoes are agents dedicated to deliver messages. The reason of introducing such special purpose imagoes is that the peer to peer communication mechanism in traditional concurrent (distributed) programming languages does not fit the paradigm of mobile agents. This is because mobile agents are autonomous - they may decide where to go based on their own will or the information they have gathered. Most mobile agents systems either do not provide the ability of automatically tracing moving agents, or try to avoid discussing this issue. On the other hand, the IMAGO system allows messenger imagoes to trace worker imagoes and therefore achieves reliable message delivery. The system provides several builtin messenger imagoes. Programmer designed messenger imagoes are possible but this kind of imagoes can only be created by the stationary imago. A messenger imago is anonymous so that there is no way to trace a messenger imago. However, it can move or even clone itself if necessary.

The IMAGO system provides a set of builtin messenger imagoes as a part of the IMAGO API. These messengers should be robust and sufficient for most imago applications. They may be dispatched by either a stationary imago or a worker imago. For the sake of flexibility, a stationary imago may also dispatch user designed messengers. In this case, the system will load the user designed messenger code from the local host, create a thread and add the messenger thread into the ready queue for execution.

In this section, we will discuss the design pattern of system builtin messengers. Each system builtin messenger has a given code name. The following example shows an asynchronous messenger: *oneway_messenger*. It is worth to note that this name is the *imago definition* name, rather than the imago instance name, because messenger imagoes are anonymous.

```
:- messenger(oneway_messenger).
oneway_messenger([Receiver, Msg]):-
        deliver(Receiver, Msg).
deliver(Receiver, Msg):-
        attach(Receiver, Msg, Result),
        check(Receiver, Msg, Result).
check(_, _, received):- !,
        dispose.
check(_, _, deceased):-!,
        dispose.
check(Receiver, Msg, cloned(Clone)):- !,
        clone(_, R),
        R == clone →
              deliver(Clone, Msg);
              deliver(Receiver, Msg).
check(Receiver, Msg, moved(Server)):-!,
        move(Server), !,
        deliver(Receiver, Msg).
check(Receiver, Msg, _):-
        deliver(Receiver, Msg).
:- end_messenger(oneway_messenger).
```

When the *oneway_messenger* is started, it tries to *attach* itself to the given receiver. Only two possible cases make the *attach* succeeds immediately: either the receiver has moved or the receiver has deceased (here we consider the receiver dead if it could not be found through the IMAGO name resolution). For the former case, this messenger will follow the receiver by calling

move and then try to deliver its message at the new host; for the later case, the messenger simply disposes itself. Otherwise, the receiver must be alive at the current host, thus the messenger attaches to this receiver and makes the receiver ready if the receiver was blocked by a *wait_accept*.

After having attached to its receiver, the messenger is suspended. There is no guarantee that the receiver will release this attached messenger by calling an accept-type predicate, because the receiver is free to do anything, such as *move*, *back* or *clone* before issuing an accept, or even *dispose* without accepting messengers. For this reason, a resumed messenger must be able to cope with different cases and try to re-deliver the message if the message has not been received yet and the receiver is still alive.

An interesting case is when the receiver imago clones itself while it has pending messengers. In order to follow the principle that a cloned imago must be an identical copy of its original, all attached messengers must also clone themselves and then attach to the cloned imago. From the *oneway_messenger* program, we can find that after knowing that the receiver has been cloned, the resumed messenger invokes *clone* and then an *if-then-else* goal is executed: the original messenger reattaches to the original receiver and the cloned messenger attaches to the cloned imago. The word *identical copy* refers to the "as is" semantics, that is, at the time an imago issues a *clone* predicate, it takes a snapshot (stack, messenger queue, *etc.*) to create the *identical* copy. Therefore, a cloned imago will have the same messenger queue as its original, but messengers pending in the queue are new threads representing cloned messengers.

The *oneway_messenger* is the most basic system builtin messenger imago. It is simple and easy to understand. The overhead of its migration from host to host is only slight higher than the cost of peer to peer message communication, because the amount of its bytecode and execution stack is very small. It implements asynchronous communication between a sending imago and a receiving imago. It has the ability to automatically trace a moving receiver.

Clearly, the concept of messengers offers flexibility to simulate different patterns of agent communication. For example, we can define a *postman* messenger to deliver mails to a list of addressed users (agents), we can design a *paperboy* messenger to dispatch a copy of message (like a copy of newspaper) to different subscribers, we can design a *round_trip* messenger which delivers a message to the receiver and carries a reply back to the sender, we can also define a *multicasting* messenger which will generate multiple clones to dispatch a message to a group of workers, as show in the following example.

```
:- messenger(multicasting_messenger).
multicasting_messenger([Receiver, Msg]):-
        back, // go back to stationary server
        multicast(Receiver, Msg).
multicast(Receiver, Msg) :-
        workers(Receiver, alive, L),
        spawn(L, Msg).
spawn([], _) :-
        dispose.
spawn([Receiver], Msg) :- !,
        deliver(Receiver, Msg).
spawn([Receiver | L], Msg) :-
        clone(_, R),
        R == clone →
              deliver(Receiver, Msg);
              spawn(L, Msg).
deliver(Receiver, Msg):-
        attach(Receiver, Msg, Result),
        check(Receiver, Msg, Result).
check(_, _, received):- !,
        dispose.
check(Receiver, Msg, cloned(Clone)):- !,
        clone(_, R),
        R == clone →
              deliver(Clone, Msg);
              deliver(Receiver, Msg).
check(_, _, deceased):-!,
        dispose.
check(Receiver, Msg, moved(Server)):-
        move(Server), !,
        deliver(Receiver, Msg).
check(Receiver, Msg, _):-
        deliver(Receiver, Msg).
:- end_messenger(multicasting_messenger).
```

Those messengers follow the basic pattern of the *oneway_messenger* in design with minor differences. Most importantly, each of these messengers has the intelligence to deliver a message to its receiver in a changing, dynamic mobile world.

5. CONCLUSION

In this paper, we discussed different communication concepts suitable for Mobile Agents systems, and investigated these concepts with respect to existing MA systems. The major concern of our survey is how to track down agents and deliver messages in a dynamic, changing world. Based on our survey, we have proposed a new communication mechanism for Mobile Agents. This paradigm has been used in our on-going project IMAGO.

Research on this subject involves two ongoing projects: a compiler of IMAGO-Prolog and the implementation of MLVM. Although this study concentrates on the design of intelligent mobile agents based on logic programming, results will be also useful in related disciplines of network/mobile computing and functional/logic programming community.

We would like to express our appreciation to the Natural Science and Engineering Council of Canada for supporting this research.

6. REFERENCES

[1] J. Baumann *et al.*, "Communication Concepts for Mobile Agent Systems", *First Int. Workshop on Mobile Agents (MA'97), LNCS1219*, Springer-Verlag, 1997, pp. 123-135.

[2] N. M. Karnik and A. R. Tripathi, "Design Issues in Mobile-Agent Programming Systems", *IEEE Concurrency*, July-Sept., 1998, pp. 53-61.

[3] D. B. Lange and M. Oshima, "Programming and Deploying Java Mobile Agents with Aglets", *Addison-Wesley*, August, 1998.

[4] "ObjectSpace: ObjectSpace Voyager Core Package Technical Overview", *Technical Report, ObjectSpace Inc.*, 1997, http://www.objectspace.com/.

[5] "Odyssey", *Technical Report, General Magic Inc.*, http://www.genmagic.com/agents.

[6] "Concordia", *Mitsubishi Electric*, http://www.meitca.com/HSL/Projects/Concordia.

[7] N. Karnik and A. Tripathi, "Agent Server Architecture for the Ajanta Mobile-Agent System", *In Proc. of PDPTA'98, CSREA Press*, 1998, pp. 62-73.

[8] J. E. White, "Mobile Agents", *Technical Report, General Magic Inc.* , 1995.

[9] D. Johansen, R. van Renesse, and F. B. Schnelder, "Operating System for Mobile Agents", *In Proc. of HotOS-V'95, IEEE Computer Society Press*, 1995, pp. 42-45.

[10] R. S.. Gray, "Agent Tcl: A Flexible and Secure Mobile-Agent System", *In Proc. Fouth Ann. Tcl/Tk Workshop*, 1996, pp. 9-23.

[11] P. van Roy *et al.*, "Mobile Objects in Distributed Oz", *ACM Trans. on Programming Languages and Systems*, (19)5, 1997, pp. 805-852.

[12] P. Tarau, "Jinni: Intelligent Mobile Agent Programming at the Intersection of Java and Prolog", *In Proc. of PAAM'99*, 1999, pp. 109-123.

[13] N. Fukuta, T. Ito and T. Shintani, "MiLog: A Mobile Agent Framework for Implementing Intelligent Information Agents with Logic Programming", *In Proc. of the First Pacific Rim Intl. Workshop on Intelligent Information Agents (PRIIA'2000)*, 2000, pp. 113-123.

[14] X. Li, "Efficient Memory Management in a Merged Heap/Stack Prolog Machine" *ACM-SIGPLAN 2nd International Conference on Principles and Practice of Declarative Programming (PPDP'00)*, 2000, pp. 245-256.

[15] X. Li, "IMAGO: A Prolog-based System for Intelligent Mobile Agents", *Proceedings of Mobile Agents for Telecommunication Applications (MATA'01), LNCS 2164*, Springer-Verlag, 2001, pp. 21-30.

[16] D. Milojicic *et al.*, "MASIF: The OMG Mobile Agent System Interoperability Facility", *LNCS 1477*, Springer-Verlag, 1998, pp. 50-67.

[17] "Foundation for Intelligent Physical Agents - FIPA'99 Version 0.2",
FIPA, http://www.fipa.org/.

[18] H. Peine, "Ara - Agents for Remote Action", *in Mobile Agents: Explanations and Examples (eds. W. Cockayne and M. Zyda)*, Manning/Prentice Hall, 1997