On the Implementation of IMAGO System

Xining Li

University of Guelph, Guelph, Canada

Summary
Mobile Agents are mainly intended to be used for network computing - applications distributed over large-scale computer networks. An intelligent mobile agent is a self-contained process, dispatched by its principal, roaming the internet to access data and services, and carrying out its assigned decision-making and problem-solving tasks remotely. In this paper, the author will present the design and implementation of the IMAGO (Intelligent Mobile Agents Gliding On-line) system. The goal of the project is to build a logic-based framework in the design space of intelligent mobile agent systems. To achieve this, we need to cope with design issues, such as explicit concurrency, code autonomy, security, communication/synchronization, service discovery, computation mobility, as well as implementation issues, such as multithreading, garbage collection, code migration, communication mechanism and database access. The unique feature of IMAGO system is that it deploys intelligent mobile messengers for inter-agent communication. Messengers are anonymous, thin agents dedicated to deliver messages. Like other agents, messengers can move, clone, and make decisions for their assigned task: track down the receiving agents and reliably deliver messages in a dynamic, changing environment.

Key words:
Mobile Agents, Inter-agent Communication, Messengers, Virtual Machine, Migration

1. Introduction

Mobile agent systems are generalized distributed systems in the sense that they are mainly intended to be used for network computing - applications distributed over large-scale computer networks. An agent is autonomous process acting on behalf of a user. A mobile agent roams the Internet to access data and services, and carries out its assigned task remotely. Numerous mobile agent systems have been implemented or are currently under development. Typical systems are Aglets[1], Voyager[2], Grasshopper[3], ARA[4], Mole[5], D'Agent[6], *etc*.

Fundamental issues related to the mobile agent paradigm are support of agent mobility and inter-agent communication. As the primary identifying characteristic of a mobile agent is its ability to migrate from host to host, a protocol to support agent mobility is an essential requirement of a mobile agent system. An agent is normally composed of three parts: code, execution state (stack), and data (heap). All these parts should move with the agent whenever and wherever it moves. However, the majority of mobile agent systems (especially those built on top of Java) only support *weak migration* - an agent moves with its code and data, but without the stack of its execution thread. Thus, the moving agent has to direct the control flow appropriately when its state is restored at the destination. In contrast to the weak migration, *strong migration* requires that the execution thread of an agent must be transferred as well, so that the moving agent can resume its execution at a new host from the point it left off. Clearly, strong migration is more suitable to the fact that agents should be able to interact with their environment autonomously and effectively.

Agents are not working alone in most mobile agent applications. They need to communicate with each other for cooperation and synchronization. Therefore, another essential issue is the design of communication models and inter-agent communication protocols. Generally speaking, communication models are concerned with conceptual paradigms such as RPC/RMI, message-based, or event-based, whereas communication protocols deal with problems such as how to name mobile agents, how to establish communication relationships, how to track moving agents, and how to guarantee reliable message delivery. Most existing mobile agent systems adopt some kind of communication models/protocols from traditional distributed systems while developing separate protocols for agent migration. However, the IMAGO system adopts a different strategy to cope with this issue. The idea is to deploy intelligent mobile messengers for inter-agent communication. Messengers are thin agents dedicated to deliver messages. Like normal agents, a messenger can move, clone, and make decisions. Unlike normal agents, a messenger is anonymous and its special task is to track down the receiving agent and reliably deliver messages in a dynamic, changing environment. As a consequence, having only a simple agent migration protocol, the IMAGO system is capable of coping with both agent migration and inter-agent communication.

Manuscript received February 25, 2006.
Manuscript revised February 28 , 2006.

The IMAGO system is an infrastructure that implements the agent paradigm. An IMAGO server resides at a host machine intending to host mobile agents and provide a protected agent execution environment. From the application point of view, the IMAGO system consists of two kinds of agent servers: stationary server and remote server. The stationary server of an application is the home server where the application is invoked. On the other hand, agents of an application are able to migrate to remote servers. Like a web server, a remote server must have either a well-known name or a name searchable through the service discovery mechanism. Remote servers should provide services for network computing, resource sharing, or interfaces to other Internet servers, such as web servers, database servers, *etc*.

This paper presents the design and implementation of the IMAGO system and is organized as follows. Section 2 gives a briefly review of recent works related to this research, focusing on the agent communication models and different mechanisms for tracking mobile agents. Section 3 presents the architecture of the IMAGO system. We will briefly discuss the virtual machine design of the IMAGO server as well as the functionality of each module. In section 4, we will discuss the intelligent mobile messenger model and feasible solutions for problems such as agent naming, agent tracking, and inter-agent communication language. At this moment, the IMAGO system only supports the IMAGO Prolog – an extended Prolog with rich Application Programming Interface supporting mobile agent applications. More programming languages are currently under investigation and will be added to the system. The IMAGO system has been implemented and is currently under benchmark testing. Finally, we present the conclusion and outline of future work.

2. Related work

Most of the mobile agents systems are based on scripting or interpreted programming languages that offer portable virtual machines for executing agent code, as well as a controlled execution environment featuring a security mechanism that restricts access to the host's private resources. Clearly, agents in a network application must interact with each other using some kind of communication models to exchange data and coordinate their execution. Typical models are message passing, RPC/RMI, and distributed event handling.

Message passing is used to support peer-to-peer communication patterns and is the most adopted model in mobile agent systems such as Aglet, Mole, D'Agent, Voyager, *etc*. Aglet supports an object-based messaging framework that is flexible, extensible, rich, and both synchronous and asynchronous. Mole deploys the (global) exchange of messages through a session-oriented mechanism. Agents that want to communicate with each other must establish a session before the actual communication can start. D'Agent supports text-based message passing. The sender should know the location and identity of the receiver. There is no guarantee for reliable message delivery because communication is lost as soon as one peer jumps to another location. Voyager implements message passing through the concept of virtual objects. Agents are a special type of object in a Voyager application. Communication with a remote object is handled by its virtual object that hides the remote location and acts as a proxy to the remote object. When messages are being sent to the remote agent, the virtual object forwards the message to the remote object and returns messages back if necessary.

RPC and RMI are commonly used paradigms in today's distributed programming. Since there is no distinction in syntax between an RPC and a local procedure call, it provides access transparency to remote operations. Several mobile agent systems support RPC/RMI paradigm such as Mole and Voyager. An argument against this paradigm is that under the new paradigm where mobile agents can move to any remote host for data and services, why we need RPC/RMI at all. Agents for Remote Action (ARA) attempts to minimize the remote communication through a 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.

The concept of event based communication and synchronization can be viewed as a sophisticated paradigm of meeting oriented agent coordination. Some mobile agent systems have much in common with those event frameworks employed in GUI toolkits supported by Java and some scripting languages. Mobile agent systems such as D'Agent, Mole, *etc*., extend the event-driven programming technique to coordinate groups of mobile agents. 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 managers. It is their responsibility to accept event registration, listen and receive events, and notify interested parties when an event arises. On the other hand, an agent participating in such groups is responsible to register a list of event types

it is interested in as well as the location it wishes events to be sent. Certainly, this model requires the static binding of agents with their registered locations, or otherwise event notification becomes unreliable.

Nevertheless, no matter which inter-agent communication model is selected, the model must be implemented through a stack of dedicated communication protocols. Inter-process communication is dependent on the ability to locate the communication entities. This is the role of the naming services that primarily map each entity in its name space to a fixed location in traditional distributed systems. Mobile agents are distributed processes. However, once they are invoked they will autonomously decide the hosts 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 that is usually modifiable at runtime. As a result, the mobility of agents makes it much harder to provide such kind of name resolution service because there is virtually no way to bind a moving agent with a static (fixed) location. Thus, existing mobile agent systems either do not provide the ability of automatically tracking moving agents, or overly constraint the mobility of agents. For example, Aglet API does not support agent tracking. Instead, it leaves this problem to applications. To avoid tracking agents during communication, Mole prevents agents from moving if they are involved in a session.

Although the RPC/RMI model offers access transparency, it turns out that general purpose, high level message-based models are more convenient, and often adopted by most mobile agent systems. However, it can always be argued whether agent communication should be remote or restricted to local, considering that the most attractive motivation of mobile agents is that they are able to migrate between locations to locate data and services as well as their peers, and therefore avoiding remote communication. Furthermore, a stack of communication protocols usually implements a communication model. It is also questionable whether such a protocol is actually necessary if we already have a simple, reliable agent migration protocol. In response to these questions, the IMAGO system utilizes the mobility of agents to achieve powerful, reliable and flexible inter-agent communication.

Research in the area of mobile agents looked at languages that are suitable for mobile agent programming, and languages for agent communication. Much effort was put into security issues [7], control issues, and design issues. However, few research groups have paid attention to offering an environment to combine the concept of service discovery and mobile agents to build dynamic distributed systems.

A variety of Service Discovery Protocols (SDPs) are currently under development by some companies and research groups. The most well known schemes are Sun's Java based Jini™[8], Microsoft's UPP[9], IETF's SLP[10] and OASIS UDDI[11]. Some of these SDPs are extended and applied by several mobile agent systems to solve the service discovery problem. Though SLP provides a flexible and scalable framework for enabling users to access service information about existence, location, and configuration, it only possesses a local function for service discovery and is not scalable up to global Internet domain. After a study of different SDPs and mobile agent systems that are adopting these methods, we found that several problems cannot be easily solved by the existing protocols due to their limitations.

3. The Architecture of the IMAGO System

The IMAGO system is an infrastructure for mobile agent applications[12]. It includes the IMAGO server - the specification and implementation of Multi-threading Logic Virtual Machine (MLVM), the IMAGO Prolog - a Prolog-like programming language extended with a rich API for implementing mobile agent applications, and the IMAGO IDE, a Java-GUI based program from which users can perform editing, compiling, and invoking an agent application.

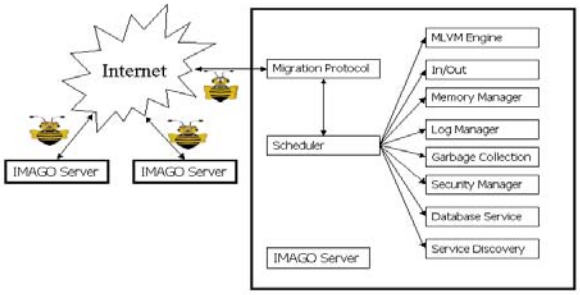

Fig. 1 The Architecture of the IMAGO System.

IMAGO Prolog is a simplified Prolog with an extended API to support mobile agent paradigm[13]. An IMAGO Prolog program consists of a set of agent definitions and module definitions. Agent definitions serve to specify autonomous entities from which mobile agents can be created, whereas modules serve to partition the name space and support encapsulation for the purpose of constructing large applications from a library of smaller components. Like most Prolog implementations that are based on different kinds of virtual machines, the

implementation of IMAGO Prolog is based on MLVM - an efficient Multi-threading Logic Virtual Machine. The most significant differences from traditional Prolog virtual machines are that MLVM adopts a novel memory management approach and fully supports distributed as well as mobile agent applications. Fig. 1 shows the architecture of the IMAGO system as well as the organization of MLVM internal modules.

Unlike the traditional client/server model, the IMAGO system exhibits a server/server model where each server is installed with the MLVM. The goal of MLVM is to present a secure, consistent and efficient execution environment for mobile agents. The MLVM combines both kernel level and user level multi-threading into its implementation. Kernel level threads are responsible for system functionalities, such as engine, memory manager, security manager, service discovery module, *etc*. Those kernel level threads constitute the virtual machine to support concurrent execution and migration of user level threads, namely, mobile agents.

Versatility in the threading algorithm is that a hardware configuration replacing threads with CPUs would accomplish the exact same behavior with little changes. To achieve such goal, the pool is initialized with a pre-defined number of general-purpose system threads. A highest-priority-first scheduling algorithm is then applied on a sorted set of queues. Each thread will scan those queues and process any task available according to the priority setting until no more task is to be handled. At this time, and at this time only, the unused thread will be put into a blocking state until new tasks are generated. The scheduler task list contains entries which match different stages of the life cycle of an agent, such as, creation, execution, and memory related processing (expansion, contraction or garbage collection), termination and migration (outgoing and incoming). The sorting of the priority list is such that the outgoing migration related tasks are of a highest priority followed by agent creation and incoming migration request tasks. The last entry in the list is, respectively, memory related tasks and the virtual machine engine, because they will be the most frequently requested cycle in the life of an agent. Such ordering prevents starvation by simply using the natural life cycle of the agent itself. The synchronization issues that arise from a multi-threaded environment have been resolved by the use of mutual exclusion primitives.

Memory manager handles the tasks associated with memory expansion and contraction. The manager adopts a merged stack/heap scheme for memory allocation, and automatically conducts garbage collection [14]. The MLVL engine is an emulator of IMAGO Prolog. Not only it executes byte-code instructions of an agent, but also it dispatches an agent to different queues for further services, such as agent creation, cloning, moving, *etc*. Agents can be either created on the fly or from a file stored on a hard disk. The In/Out module interacts with the migration protocol to handle incoming and outgoing agents.

3.1 Agent Migration Protocol

To make it easier to deal with issues involved in different layers of communication, stack architecture has been chosen for the IMAGO agent-migration protocol. The design goal of the protocol stack is to provide the flexibility, reliability and security required in the mobile agent migration. The protocol stack also implements some cutting-edge technologies in multi-thread management. The current implementation consists of five layers, from top to bottom, Marshaling, Rendezvous, Security, Routing and Connection. The whole stack sits on top of the TCP/IP layer.

In brief, the Connection layer is in charge of establishing and maintaining a direct TCP connection between two hosts. The connection layer offers two mechanisms for transmitting information over the network. One provides unreliable, lighter and faster service for exchanging control information between different servers. The other offers a much more reliable, wider bandwidth, but slower and using more resources. It is mainly used to transmit greater amount of data that should not be lost or corrupted, such as an entire mobile agent.

The Routing layer resolves names and decides the route that a Protocol Data Unit (PDU) will take to reach its destination. Migration, being a basic feature of mobile agents, should be made as robust and flexible as possible while hiding, as much as possible, all the irrelevant, underlying details from the end user. Since name resolution is quite time consuming, an extremely simple caching algorithm is used in the internal lookup process in order to reduce redundant DNS requests.

The Security layer is in charge of the encryption and decryption of PDUs as well as detecting tampering attacks. To secure a connection between two servers, encryption is only enough to insure privacy by preventing malicious people to read the content of the message. In order prevent masquerading attacks, this layer provides a simple authentication scheme for servers to detect possible fake identity and to drop such falsified PDUs.

The Rendezvous layer makes sure that everything sent arrives at destination or reports an error. The purpose of

this layer is to establish a reliable agent migration channel between two servers regardless of the underlying network architecture. Using the shaking-hands technology, this protocol is able to certify the success of an agent migration as well as its failure. This means that a moving agent will never be lost or duplicated and its migration will not suffer any side effect from network failure.

The top of the stack would be the place for the adaptation layer that will allow mobile agents to be transformed and prepared for transport. This layer, namely the Marshaling, is responsible to transform and trim a moving agent so only the required data would be transported to the destination server.

3.2 Security

Clearly, the potential benefits of mobile agent technology must be weighted against the very real security threads. Security threads can be visually lumped into three categories: (1) comprised hosts and mobile agents that attempt represent different parties that may exhibit malicious behavior toward one another, (2) exposure of mobile agents to third party intruders through the network, and (3) agents interfere with each other or gain unauthorized access to internal state. Security problems of the last two categories have been handled by the IMAGO migration protocol. The first category can be further divided into two cases: protecting agents from malicious hosts, or vice verse, protecting hosts from malicious agents.
The first case, in which a malicious server attacks a visiting mobile agent, is the most difficult issue. In order for an agent to function, the server on which the agent executes must be able to interact with each other. Most mobile agent systems rely at least on the concept of virtual machine to standardize the execution environment. However, agents written in scripting or interpreted languages are easily de-compiled. Consequently, a malicious host may easily steal private information, modify the agent code, or mislead the agent.

The IMAGO system assumes that IMAGO servers are trustworthy. The security mechanism will focus on the second case, *i.e.*, protecting hosts from malicious agents. The security manager, one of the kernel threads in MLVM, is designed to deal with the system privacy issues, such as physical access restrictions, application availability, content integrity, and access policies.

Although the migration protocol provides a secure communication channel for moving agents, whether or not an agent has tampered with is uncertain. It is the security manager's responsibility to verify the integrity of agent code using a digit signature, such as optimized MD5. In addition, IMAGO system adopts a two-stage process to check if a system access issued by an agent is valid. The first stage is conducted during compilation. A so-called static semantic check will prevent a mobile agent to use illegal system calls. When an agent is loaded to execute, the second stage, namely, dynamic semantic check is required to ensure that the agent code had not been tampered with during transit.

Certainly, the semantic checking technique only protects the server from the illegal usage of system resources. However, for some types of abnormal mobile agent activity, such as unlimited cloning itself or constantly expanding memory, the security manager must provide further assurance. The technique being adopted is called limitation, which controls the persistent survivability of mobile agents and prevents them from "running wild". The typical limitations controlled by the security manager are span of lifetime, number of migrations, number of cloning, amount of memory, *etc*. In order to facilitate different configurations of agent servers, a customizable security policy generator is equipped with the installation procedure of the IMAGO system. System installer can incorporate security manager comprised of various choices of security policies, refined from a set of system recommended defaults.

3.3 Service Discovery

Mobile agents must interact with their hosts in order to use their services or to negotiate services with other agents. Discovering services for mobile agents comes from two considerations. First, the agents possess local knowledge of the network and have a limited functionality, since only agents of limited size and complexity can efficiently migrate in a network and have little overhead. Hence specific services are required which aim at deploying mobile agents efficiently in the system and the network. Secondly, mobile agents are subject to strong security restrictions, which are enforced by the security manager. Thus, mobile agents should find services that help to complete security-critical tasks, other than execute code that might jeopardize remote servers. Following this trend, it becomes increasingly important to give agents the ability of finding and making use of services that are available in a network.

In the IMAGO system, we have implemented a new service discovery model DSSEM (Discovery Service via Search Engine Model) for mobile agents. DSSEM is based on a search engine, a global Web search tool with centralized index and fuzzy retrieval. This model especially aims at solving the database service location

problem and is integrated with the IMAGO system. Service providers manually register their services in a service discovery server. A mobile agent locates a specific service by submitting requests to the service discovery server with the description of required services. Web pages are used to advertise services. The design goal of DSSEM is to provide a flexible and efficient service discovery protocol in a mobile agent environment.

Before a service can be discovered, it should make itself public. This process is called service advertisement. The work can be done when services are initialized, or every time they change their states via broadcasting to anyone who is listening. A service advertisement should consist of the service identifier, plus a simple string saying what the service is, or a set of strings for specifications and attributes.

The most significant feature of DSSEM is that we enrich the service description by using web page's URL (later the search engine will index the content referenced by this URL) to replace the traditional string-set service description in mobile agent systems. Because of their specific characteristics, such as containing rich media information (text, sound, image, *etc.*), working with the standard HTTP protocol and being able to reference each other, web pages may play a key role acting as the template of the service description. On the other hand, since the search engine is a mature technology and offers an automated indexing tool that can provide a highly efficient ranking mechanism for the collected information, it is also useful for acting as the directory server in our model. Of course, DSSEM also benefits from previous service discovery research in selected areas but is endowed with a new concept by combining some special features of mobile agents as well as integrating service discovery tool with agent servers. Fig. 2 shows the steps of the IMAGO service discovery process.

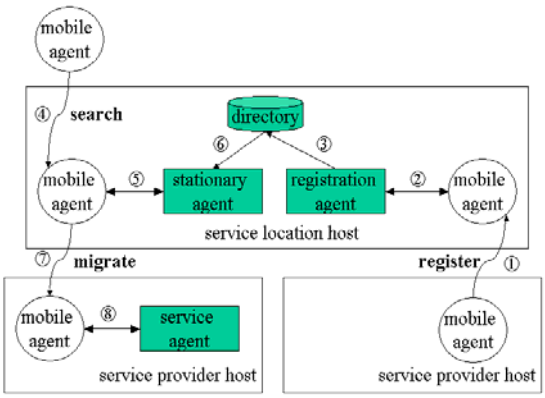

Fig. 2 Process of the IMAGO Service Discovery Module

3.4 Database Management

In order to endow agents with the ability of accessing remote data resources, the IMAGO system provides a database interface between mobile agents and remote DBMS. To the database interface coupled with mobile agent system and remote DBMS, efficiency has become an important issue, because heavy-duty agent database operations may easily turn it into bottleneck. Two levels of system efficiency have been introduced: the technical level and logic level. On the technical level, the most commonly used designs are Database Connection Management [15] and Local Cache Management [16].

In IMAGO system, each agent appears in the form of a logic program written in Prolog. Furthermore, a set of pre-defined system predicates can be used in the agent program to require certain services provided by agent servers. For example, several sample predicates of database accessing are shown in the following code segment:

```
// Database connection predicate.
db_connection(
   connection('131.104.127.113','Guelph','student
   ','1234','readonly'), Handler),

// Database searching predicate with SQL query.
db_search("select * from student", Handler),

// Database disconnection predicate.
db_disconnection(Handler).
```

Generally an agent will stay inside of the engine module and its byte code will be executed until a pre-defined system predicate is hit. Then the engine will postpone the current process, and transfer the agent to a proper service module according to the type of the system predicate. For example if the predicate is *db_connection*, the agent will be relocated to database module. After the required service being done, it will be returned back to the engine module for continuous execution.

Whenever an agent is transferred into Database Module, the connection management will be able to receive proper request of the database operation based on the type of the current database predicate in the agent program. A corresponding kernel thread will be picked up from the Database Thread pool under some pre-defined thread assignment strategies. After finishing the current database operation, the agent will be returned back to the engine.

By introducing special database module threads, it is possible to unload those heavy resource consuming database operations from system threads. Thus system threads can run quickly and smoothly without being delayed or blocked and the performance of the whole

system will not be affected by executing heavy-duty database jobs. The architecture of multi-threading database connection management is shown in Fig. 3.

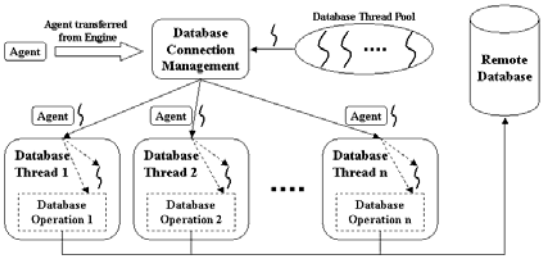

Fig. 3 Architecture of Multi-threading Database Connection

4. Intelligent Mobile Messengers

The IMAGO system utilizes the mobility of agents to achieve powerful, reliable and flexible inter-agent communication. To better understand how our model works, we shall first distinguish messengers from normal mobile agents, and we shall call them as workers in the rest of this paper. A worker is a normal mobile agent created by its owner for some specific task, whereas a messenger is an anonymous thin agent dispatched by a worker to deliver messages. Generally speaking, a worker is purposely separated from the location of its owner, and best equipped with as much intelligence as possible in order to autonomously carry out the assigned task on behalf of its owner. Unfortunately, adding more intelligence to a worker will make some sacrifices of mobility. Thus, to deploy these thick workers directly for inter-agent communication is neither economical nor practical. This is the reason for introducing messengers – specialized thin agents - that not only provide an agent-based solution for inter-agent communication but also make the solution efficient and feasible.

4.1 Naming

Clearly, each worker must have a unique name so that its owner and other workers can communicate with it over the network. Some systems, such as Voyager, adopt location-transparent names at the application level. In contrast, systems such as Aglets and D'Agent, assign location-dependent names. For example, an agent in Aglets is associated with a unique identifier so that every agent in the network can be uniquely addressed by combining its identifier with its context URL.

Obviously, identifying an agent by the combination of its identifier and its current location does not fit well to the mobility of agents. Since a worker may move any time to an arbitrary remote server, its current location is uncertain. For this reason, our model adopts a location-transparent, closed-world naming scheme to identify workers. First, we assume that a user-friendly, symbolic name is assigned to each worker. Such a name must be unique in the application (a closed world) where the worker is created, and immutable throughout the worker's lifetime. This user-friendly symbolic name is used to unambiguously refer to the worker inside the application that it belongs to.

Secondly, we assume that each application is bound to a home location that always exists during the lifecycle of the application. Consequently, workers of an application are all originated from the same home. If several applications are concurrently running at the same home, we shall use different sequencing numbers to distinguish them. Therefore, by concatenating the user-friendly symbolic name, the home URL and the application sequencing number, we have a location-transparent, globally unique identifier for each worker. It is worthwhile to note that the home URL embedded in a worker's identifier is independent from the worker's current location. By using such identifiers, it is sufficient to unambiguously refer to a worker over the Internet.

Both workers and messengers are allowed to move freely from one host to another. That is, they may decide where to go based on their own will or the information they have gathered. At any stage of execution, a worker can dispatch a messenger to deliver a message to another worker. Deployment of messengers is the only way to achieve inter-agent communication in our model. Since messengers are in fact mobile agents, they can be designed to serve different purposes such as asynchronous messaging, synchronous messaging, broadcasting or multicasting, *etc*. To accommodate incoming messengers, we assume that each worker is associated with a messenger queue that holds all messengers destined to this worker and waiting for delivery of messages. The assumptions we made in this section are reasonable because they are already satisfied by our implementation as well as some other mobile agent systems.

4.2 Locating Mobile Agents

Briefly speaking, locating an agent is invoking a function of the form “where_is(X)” which should return the current address (or access point) of agent X. Researchers have recently proposed many schemes for designing such a function. Various approaches for storing, updating, and locating mobile agents are well addressed in [17].

Some communication protocols use *broadcasting* or *multicasting* approaches to locate mobile agents. In this paradigm, location queries are broadcast to the entire network, or multicast to a specific group of hosts. Upon receiving such a request, each machine must check the names of hosting agents and give a response if the requested name is found. This paradigm is mainly applicable to intranets. It becomes inefficient in a large-scale network. Furthermore, the answer to "where_is(X)" is not accurate if X moves to another host immediately after the answer is returned, which makes message loss unavoidable.

Another popular approach is to use a fixed location server, called home, to keep track of locations of mobile agents. In this scheme, agents follow a triangular routing to communicate with each other, that is, a message is sent to home first, which looks up the destination address and then simply forwards the message to the receiving agent. Unfortunately, the same problem remains. The address returned from the lookup function "where_is(X)" is ambiguous: X might still reside at that address, or X might have moved to another host and its location updating packet is on the way to home, or X might even have started to move at the same time a message is sent to its current location.

Forward-pointer is a promising alternative for locating mobile agents. This scheme does not depend on a "where_is(X)" lookup function. Instead, each mobile agent host keeps a reference (forward pointer) for each moving agent. For example, in Voyager, 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 the corresponding virtual object has received a returned message. The advantage of this approach 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. Furthermore, a theoretical flaw is that messages might forever chase a frequently moving receiver, even though this hardly occurs in practice.

In the IMAGO system, the only way that workers cooperate with each other is by the means of dispatching messengers. Therefore, each messenger is responsible for locating the receiving worker. In order to locate a moving worker, agent servers should maintain enough information to keep track of current location of every worker. However, we have indicated that it is virtually impossible to have the precise information about a changing environment, because an application may involve workers that are creating, cloning or moving all the time. To cope with such a dynamic configuration, our model maintains heuristic location information through distributed registration and local updating operations, and employs a variant of forward-pointer-based approach plus a home-based mechanism as the backup.

Based on the naming scheme, identifiers of workers have an embedded static home location, although these workers might spread and roam over the network. This home is the default server for workers to send their registration. A newborn worker, either by creating or cloning and regardless have born at the home host or a remote host, must register its birthplace with the home automatically. Even though registrations take a distributed manner, *i.e.*, registration messages might flow to the home from different remote hosts, it does not cause much network traffic because each worker registers only once in its whole lifecycle.

A registration message is stored as a worker record in the local cache of the home server. A worker record is a structure of the form {*worker_id, timestamp, status*} where *worker_id* is the globally unique identifier of the worker, *timestamp* gives the time the record last been modified, and status indicates the current state of the worker. For the sake of simplicity, we assume that a worker must be in one of three possible states: ALIVE, DISPOSED, or MOVED_TO(url).

Like the home server, a remote agent server also remembers a collection of worker records per application basis. However, it maintains caching information through the local updating operation. Such an operation is very efficient because it is done completely in the local system layer. In general, a worker record is inserted into the local cache when the worker is created or cloned locally, or the first time it moves into this server. To make the local caching more effective in locating a worker, a remote server should also cache sender's information carried by a messenger. Obviously, caching sender's information exploits locality. For instance, a receiving worker is most likely to reply to its sender in the near future, and the sender's location can be found immediately from the local cache.

An updating operation is also applied to a worker record whenever the worker changes its state. For example, when a worker moves from host S1 to host S2, its cached record at S1 is modified with the new state MOVED_TO(S2). This is very similar to the forward pointer scheme which leaves behind a forwarding reference whenever an entity moves to a new location. Likewise, if the worker moves from S2 back to S1, its

record at S1 will be simply changed back to the state ALIVE. Furthermore, updating operations can be used to short cut a forwarding chain. For example, suppose that a worker moves from S1 to S2 and then to S3. From there, the worker dispatches a messenger to a receiver at S1. When the messenger arrives, the updating operation will change the worker (sender)'s record from the old state MOVED_TO(S2) to the new state MOVED_TO(S3). Therefore, subsequent communications to that worker will be dispatched to S3 directly.

In the IMAGO system, we do not intend to have a network-wise "where_is(X)" lookup function for locating the current address of X. Instead, we use a local lookup function that returns a possible location of X. The reason for saying possible is that the information recorded in a server's local cache is heuristic. For instance, if the status of a worker is recorded as MOVED_TO(S2), there is no guarantee that the worker we are looking for is still working at S2, because a worker is never bound to an absolute host address - it may very well have moved on to another location. However, it is guaranteed that successive lookup's at subsequently forwarded heuristic hosts will eventually trap the worker if the worker really wants to accept the messenger.

Now, let us consider the general lookup facility for remote servers. The principle is very simple. We only search the local cache to find where the worker possibly resides in. This lookup function will never return something like WORKER_NOT_FOUND. Instead, it either returns the value of the current status from the located worker record, or MOVED_TO(home) if a cache miss occurs. Since a remote server might host multiple concurrent agents (workers and messengers), the lookup operation and the updating operation must be mutual exclusive. That is, when a messenger has to locate a worker and deliver message, it must lock the cached worker record (critical region) to achieve mutual exclusion and ensure that the worker is not able to change its state at the same time.

The lookup function on the home server is analogous to the above description. In principle, there is no cache miss because the home should hold a complete set of worker records. However, what possibly happens in practice is that a messenger is dispatched to a worker who might have not been created yet or whose registration message might be on the way home. To solve this problem, the lookup function simply blocks this messenger. A blocked messenger will be resumed if a new registration with a matching receiver arrives.

4.3 Messenger Behavior

A messenger is an agent. It has its own code to be executed. There are many ways to design messengers for different purposes. To make it easier to understand, we will start out by discussing a very important system primitive. Then we will look at a simple messenger and discuss its behavior in some detail. A more concrete example will be given in the next section.

System primitives serve as the interface between agents and the underlying system. In addition to the commonly used primitives such as *create*, *move*, *clone*, *etc.*, another primitive that plays a major role in between workers and messengers is *attach*. The following code segment shows the skeleton of the *attach* primitive.

```
attach(receiver){
        lock(local_cache);
        r = lookup(receiver);
        if (r == ALIVE){
          // insert this messenger into
          // the receiver's messenger queue
          unlock(local_cache);
          // switch to another ready agent
        }
         else {
          unlock(local_cache);
          retrun r;
        }
}
```

The basic idea behind a messenger is try to track down the receiver until its message is accepted. To achieve such behavior, a messenger simply invokes the following recursively defined *deliver* function.

```
deliver(receiver, message){
        r = attach(receiver);
        if (r == RECEIVED || r == DISPOSED)
                 dispose();
        else { // r == MOVED_TO(url)
                move_to(url);
                deliver(receiver, message);
        }
}
```

A messenger starts by invoking a call to *attach* which will issue a lookup mutual exclusively. Only two possible cases make the *attach* return immediately: the receiver has deceased locally, or the receiver has moved to another host. Recall that MOVED_TO(home) will be returned if a cache miss occurs in a lookup, so that it seems as if the receiver has migrated to home. Therefore, the messenger will follow the receiver by calling *move* and then try to deliver at the new host, or simply dispose itself if the receiver no longer exists.

On the other hand, if the receiver is ALIVE at the current host, the *attach* primitive will insert the caller into the receiver's messenger queue and then make it suspended.

The underlying system is free to schedule another ready agent to execute. It is now the receiver's responsibility to resume an attached messenger based on different actions it is going to perform. Certainly, the receiver might invoke an *accept*-type primitive. If this happens, the accepted messenger is resumed as soon as its carried message has been transferred to the worker's working space. Unfortunately, whether its receiver will accept attached messengers is unknown, because the receiving worker might not be ready to accept any messenger yet. For instance, it is possible that the worker moves to another host while there are pending messengers. It is possible that the worker disposes itself without accepting messengers. It is also possible that the worker clones itself while it has a non-empty messenger queue. Nevertheless, these possible cases are facing with the same problem, namely, how the receiver deals with pending messengers.

From the well-known semantics of strong migration, a mobile agent should take its code, data and execution state together when it moves to a new location. Unfortunately, such semantics have a flaw of ignoring messages. In fact, messages to an agent should also be a part of the agent. If they have been received, they become a portion of data. If they have not been received (either buffered by the underlying system or still in transmission), then they should go with the agent together whenever the agent moves. Therefore, the highest degree of strong migration is to take four parts of an agent, *i.e.*, code, data, state and messages, into consideration.

Although it sounds more difficult, the solution in our model is straightforward. A worker simply resumes all attached messengers if it moves. Likewise, a worker resumes all pending messengers if it disposes itself. Now consider what happens, for example, when the receiver it was attached to resumes a messenger. At this point, it seems as if the call to *attach* has just returned. However, the returned value might be one of the three possible cases now: RECEIVED, DISPOSED or MOVED_TO(url). Therefore, the 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. This is why a messenger will invoke *attach* each time it moves to a new place. A messenger claims that “I can track the receiver down provided I have the trail of the receiver”, whereas our lookup facility says that “the location I found is where most likely the receiver resides at, or at least the receiver has lived”. In other words, the heuristic location from the lookup facility provides the trail of the receiving worker while leaves the tracking-down job to the messenger.

4.4 Agent Communication Language

Agent Communication Language (ACL) is in fact a high-level communication protocol that allows the sending agent and receiving agent mutually understanding each other. In an ACL, a message consists of two separate aspects, namely, performative and content. The performative shows the purpose of a message while the content gives a concrete description for achieving the purpose. Of course, the sending agent and the receiving agent must agree with their ACL, so that they have at least the same understanding of the purpose and the same interpretation of the content of a message.

An ACL developed by FIPA [18] has defined several performatives of messages, such as IFORM, QUERY-IF, CFP, PROPOSE, and so on. For example, performative INFORM indicates that the content of a message is a *true* proposition, whereas QUERY-IF asks if the proposition given as the content of a message is *true*. On the other hand, FIPA does not prescribe the language used to express the message content. Instead, it specifies the ACL Protocol Data Unit (PDU) as a data structure, which contains a set of one or more message elements, such as performative, sender, receiver, language, content, *etc*. Precisely which elements are needed for an ACL message is application dependent, except that the performative element is mandatory in all ACL messages. Certainly, most ACL messages will also include sender, receiver, and content elements. A simple example of a FIPA ACL message is given in Table 1.

Table 1: An Example of FIPA ACL

Element	**Value**
Performative	INFORM
Sender	alice@simp://where.alice.resides.at
Receiver	white_rabbit@simp://to.be.located
Language	Prolog
Content	invite(mad_tea_party)

As the matter of fact, our agent-based communication model is in compliance with the FIPA ACL message structure specification. The example in Table 1 is a typical message carried by a mobile messenger. In the IMAGO system, the messenger type determines performatives of messages. For example, a messenger created from the one-way messenger carries a message with a default performative INFORM. To identify the sender, *alice@simp://where.alice.resides.at* is used to refer to an agent called *alice* residing on a mobile agent server with the DNS name *where.alice.resides.at* and relying upon the Simple Imago Migration Protocol. However, the receiver *white_rabbit* is at an unknown location at this moment. It is the messenger's responsibility to locate the receiving

agent. In general, the receiver may be a single agent name, or a non-empty list of agent names. The language element specifies that the message content be expressed as a Prolog term that could be an atom, a variable, a list, or a compound structure. Consequently, upon receiving this message, the receiver *white_rabbit* should know that *alice* invites him to a *mad_tea_party*.

Obviously, the performative and content of a message often determine the reaction of the receiver. In addition to the various types of system messengers for sending agents, the IMAGO system provides a set of primitives for receiving agents. The primitive which is similar to an unblocking receive is *accept(Sender, Msg)*. An invocation to this primitive succeeds if a matching messenger is found, or fails if either the caller's messenger queue is empty or there is no matching messenger in the queue. Likewise, the primitive which implements blocking receive is *wait_accept(Sender, Msg)*. A call to this primitive succeeds immediately if a matching messenger is found. However, it will cause its caller to be blocked if either the caller's messenger queue is empty, or no matching messenger can be found. In this case, it will be automatically re-executed when a new messenger attaches to the caller's messenger queue. Pragmatically, the semantics of matching messengers is implemented by Prolog unification. Let *(S, M)* be the sender and content element carried by a messenger, and *(Sender, Msg)* be the arguments of an accept-like primitive, the messenger is a matching messenger of the accept-like primitive if the general unification of *(S, M)* and *(Sender, Msg)* succeeds.

5. Conclusion

In this paper, we discussed the design issues of mobile agent systems and concepts for inter-agent communication, and investigated these issues with respect to existing mobile agent systems. We presented the design of IMAGO system, and discussed implementation issues such as virtual machine, code migration, security, service discovery, communication mechanism and database access. The major concern of IMAGO system is how to track down agents and deliver messages in a dynamic, changing environment. We proposed an agent-based model that deploys intelligent mobile messengers for inter-agent communication. The advantage of the messenger model is that a simple, reliable agent migration protocol is sufficient to support both agent migration and inter-agent communication. The IMAGO system has been implemented and is currently under benchmark testing. An evaluation release of IMAGO is available at the IMAGO Lab Web site (http://draco.cis.uoguelph.ca/main.html).

Research on this subject involves further extensions of API and investigation of adding more programming languages to the system. 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 and mobile computing community.

Acknowledgments

I would like to thank members in the IMAGO Lab for their contributions to the IMAGO project. I would also like to express my appreciation to the Natural Science and Engineering Council of Canada for supporting this research.

References

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

[2] http://www.objectspace.com/, Voyager: *Application Development Platform for Distributed Java Applications*, 2005

[3] C. Baumer, M. Breugst & S. Choy, 1999, Grasshopper - a universal agent platform based on OMG MASIF and FIPA standards, *In Proc. of MATA*, 1999, pp. 1–18

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

[5] J. Baumann *et al*., "Mole - Concepts of Mobile Agent System", *World Wide Web*, (1)3, 1998, pp. 123-137

[6] R. Gray, G. Cybenko & D. Kotz, "D'Agents: Applications and Performance of a Mobile-Agent System", *Software – Practice and Experience*, 32(6), 2002. pp. 543-573

[7] G. Vigna, "Mobile Agents and Security", LNCSD9, Vol. 1419, *Springer-Verlag Inc*., 1998

[8] http://www.sun.com/jini/, Sun. Technical, "Jini Architectural Overview", *White Paper*, 1999

[9] Universal Plug and Play Forum, "Universal Plug and Play Device Architecture", Version 0.91, *White Paper*, 2000

[10] E. Guttman, C. Perkins & J. Veizades, "Service Location Protocol", Version 2, *White Paper*, *IETF*, *RFC* 2608, 1999

[11] http://www.uddi.org, OASIS UDDI, UDDI *White Paper*, 2005

[12] X. Li, "IMAGO: A Prolog-based System for Intelligent Mobile Agents", *In Proceedings of MATA*, 2001, pp. 21-30

[13] X. Li, "IMAGO Prolog User's Manual version 1.0", *Technical Report*, University of Guelph, 2003

[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*, 2000, pp. 245-256

[15] D. Mckay, T. Finin and A. O'Hare, "The Intelligent Database Interface: Integrating AI and Database Systems", *In Proceedings of the 8th National Conference on Artificial Intelligence*, 1990, pp.677-684

[16] A. Sheth and A. O'Hare, "The Architecture of BrAID: A System for Bridging AI/DB Systems", *In Proceedings of the Seventh International Conference on Data Engineering*, 1991, pp. 570-581.

[17] A. Tanenbaum and M. van Steen, "Distributed Systems", *Prentice Hall Inc*., 2002

[18] http://www.fipa.org, "Agent Communication Language Specifications", *FIPA*, 2005

Xining Li is a professor of computing and information science at the University of Guelph and the director of the IMAGO Lab. His research interests include mobile agent system, logic programming, and virtual machine implementation. Li received a PhD in computer science from the University of Calgary, Canada.