ON THE IMPLEMENTATION OF AN AGENT MIGRATION PROTOCOL

A Thesis

Presented to

The Faculty of Graduate Studies

of

The University of Guelph

by

GUILLAUME AUTRAN

In partial fulfillment of requirements

for the degree of

Master of Science

May, 2003

©Guillaume Autran, 2003

National Library
of Canada

Acquisitions and
Bibliographic Services

395 Wellington Street
Ottawa ON K1A 0N4
Canada

Bibliothèque nationale
du Canada

Acquisitions et
services bibliographiques

395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file Votre référence

Our file Notre référence

The author has granted a non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of this thesis in microform, paper or electronic formats.

The author retains ownership of the copyright in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

L'auteur a accordé une licence non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de cette thèse sous la forme de microfiche/film, de reproduction sur papier ou sur format électronique.

L'auteur conserve la propriété du droit d'auteur qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

0-612-85025-0

ABSTRACT

ON THE IMPLEMENTATION OF AN AGENT MIGRATION PROTOCOL.

G. Autran. M.Sc., University of Guelph, May, 2003. Advisor: Professor Xining Li.

The world of Internet provides millions of computers and a great number of servers hosting an information retrieval system such as databases. The casual way used to search, extract and compile information contained in this gigantic pool of data, is based on a client-server architecture. A new approach to distributed information retrieval has seen the light of day a few decades ago with the merging of AI technology (for information retrieval) and Distributed Computing (to cope easily with distributed database).

Mobile-Agent systems, although well suited for distributed information retrieval, bring to the Internet world much more. Half way between Artificial Intelligence and Distributed Computing, the motto of Mobile Agents is: *“move the computations to the data rather than the data to the computations”*.

This thesis presents a novel agent-migration protocol developed in the IMAGO system. The IMAGO (Intelligent Mobile Agent Gliding On-line) system is an infrastructure for Mobile Agent applications. The major feature of this system is that it is capable of coping with both agent migration and inter-agent communication by deploying a simple, reliable agent migration protocol.

Our approach will focus on the protocol stack as well as its implementation - a multi-threaded, versatile scheduler that associates threads of control with the data flow rather than the instruction flow. Overall, the stack architecture of the IMAGO agent-migration protocol provides the flexibility, reliability and security required in the MA paradigm with a minimum footprint. It also implements some “cutting edge” technologies in multi-thread management.

ACKNOWLEDGMENTS

This whole project would have never been possible without the help of a few people. I would like to thank first, my supervisor, Dr. Xining Li, for his support. We have been working together for few years already and he has always been good advice and a perfect listener.

My friend and team mate, Hua Liang, deserves also many thanks for his patient regarding my constant requests. He always was a good support and helped to create a friendly working environment.

Dr. Wilson taught me many concepts in the *Networks and Distributed Systems* class. Some of the idea and algorithm used in the present thesis came straight from his teaching. He is the one responsible for introducing me to the *ADA rendez-vous* mechanism and I truly thank him for his clear and precise explanation of the algorithm.

As a cryptography advisor, Dr. Obimbo advice helped me validate my implementation and provided some very interesting insight on the topic.

Of course, this entire report would not be written in proper English without the help of my dear friend Denise Bellmore. She did a tremendous job reviewing this document and she did her best rewording my “Englofrench” enigmatic writing into good old English. Her promptitude and great deal of effort were exemplary and greatly appreciated.

To whom this all project would have never happen in the first place, I want to dedicate my last acknowledgment to my parents. Their constant support never cease even while my personal goal did not match their plan. I truly believe they did a great job raising their son and I hope to do as well as they did when my time comes.

TABLE OF CONTENTS

	Page
LIST OF TABLES	v
LIST OF FIGURES	vi
ABBREVIATIONS	viii
1 Introduction	1
2 Related Work	8
2.1 CBorg	8
2.2 Concordia	18
2.3 D'Agent	27
2.4 Messengers-M ϕ	36
2.5 Summary	41
3 Intelligent Mobile Messengers	43
3.1 Naming	43
3.2 Locating Mobile Workers	45
3.3 Messenger Behavior	47
4 The IMAGO System	51
4.1 IMAGO-Prolog	51
4.2 IMAGO Messengers	56
4.3 An Overview of MLVM	59
5 IMAGO Migration Protocol	68
5.1 Design Objectives	68
5.2 Multi-layered Migration Protocol Stack	72
5.2.1 Notation	73
5.2.2 Connection	73
5.2.3 Routing	77

	Page
5.2.4 Security	82
5.2.5 ADArendez-vous	91
5.2.6 Marshaling	96
5.3 Implementation	103
6 Performance Analysis	107
6.1 Simple Round Trip	110
6.2 One Way Trip	113
6.3 Round Trip With Growing Imago	116
7 Conclusion	119
LIST OF REFERENCES	121
A The project	125
A.1 What files to download ?	125
A.2 What to do with those files ?	126
A.3 Installation	126
A.3.1 Prerequisites	126
A.3.2 Building / Installing	127
A.4 Configuration	128
A.5 Problems?	128
A.6 Usage	128
A.7 Reporting Bugs	130
B Imago Migration Network Trace	131
B.1 Connection Layer	132
B.2 Routing Layer	133
B.3 Security Layer	136
B.3.1 RSA Key Exchange	136
B.3.2 Encrypted Packet	139
B.4 ADArendey-vous Layer	141
B.5 Marshaling Layer	142

C Performance Analysis Program	144
--	-----

LIST OF TABLES

Table	Page
2.1 Technical Details	9
2.2 CBorg <i>Sync-primitive</i>	15
4.1 Agent-oriented Predicates	54
4.2 Behavior of Accepting Predicates	58
5.1 Resolver Successive Requests.	79
5.2 Known Host Database File Format.	86
5.3 Meaning of the Security PDU Flags.	88
5.4 Supported Cipher Bit Mask.	88
5.5 The <i>PDU Type</i> (ADA Header).	96
5.6 Possible Imago Types.	99

LIST OF FIGURES

Figure	Page
2.1 Migrating An Agent Over The Network.	13
2.2 Concordia Architecture.	20
2.3 The Architecture Of Agent Tcl.	28
2.4 A Simple Tcl Agent.	31
2.5 Docking System.	33
2.6 Messenger Platform.	36
4.1 MLVM Architecture	61
4.2 Layout of an Imago Block.	63
4.3 IMAGO Server Architecture.	67
5.1 Migration Protocol Stack.	71
5.2 Middle-ware PDU.	72
5.3 Connection PDU.	76
5.4 Imago Routing Protocol Example.	79
5.5 Routing PDU.	81
5.6 Imago Gateway Minimum Stack.	82
5.7 Security Disabled PDU.	86
5.8 Security Key Exchange PDU.	87
5.9 Security PDU.	89
5.10 ADArendez-vous Three PDU Exchange.	93
5.11 Losing The Initial PDU.	93
5.12 Losing The <i>accept?</i> Message.	94
5.13 Losing The <i>ACK</i> Message.	94
5.14 Complete ADArendez-vous PDU.	95
5.15 Imago Block Compaction.	98

Figure	Page
5.16 Pointer Adjustment After De-compaction.	98
5.17 Marshaling PDU Header.	100
5.18 Marshaled Imago.	101
5.19 Migration Stack Simple State Diagram.	105
5.20 State Machine Structural Behavior.	106
6.1 Imago Lab Intranet.	108
6.2 Simple Round Trip Statistic Program.	111
6.3 12000 Round Trip With Encryption Enabled.	112
6.4 12000 Round Trip With Encryption Disabled.	112
6.5 One Way Timing With Clock Drift Estimation.	114
6.6 12000 One Ways Trip With Encryption Enabled.	115
6.7 12000 One Ways Trip With Encryption Disabled.	116
6.8 12000 Round Trip With Size Increase (Encryption Enabled).	117
6.9 12000 Round Trip With Size Increase (Encryption Disabled).	118

ABBREVIATIONS

ACK	ACKnowledgment
AIDL	Agent Interface Definition Language
API	Application Program Interface
ASCII	American Standard Code for Information Interchange
BSD	Berkeley System / Software Distribution
CGC	Chronological Garbage Collection
CRC	Cyclic Redundancy Check
CSP	Communicating Sequential Processes
CVS	Concurrent Versions System
DNS	Domain Name System
DSA	Digital Signature Algorithm
FIFO	First-In-First-Out
FTP	File Transfer Protocol
GCC	GNU C compiler
GNU	GNU's Not Unix!
HTTP	HyperText Transfer Protocol
ICB	Imago Control Block
IDE	Integrated Development Environment
IGW	Imago GateWay
IMAGO	Intelligent Mobile-Agent Gliding Online
IP	Internet Protocol
IPC	Inter-Process Communication
ISO	International Standards Organisation
LVM	Logic Virtual Machine
LWP	Light Weight Process

MA	Mobile-Agents (not 'Multi-Agents')
MAC	Media Access Control (ISO, OSI)
MIMD	Multiple Instruction [stream], Multiple Data [stream]
MLVM	Multi-thread Logic Virtual Machine
MTU	Maximum Transmission Unit
NACK	Negative ACKnowledgment
NTP	Network Time Protocol
OS	Operating System
OSI	Open Systems Interconnection
PDU	Protocol Data Unit
PID	Process IDentification number
POSIX	Portable Operating System for unIX
PPP	Point-to-Point Protocol
QoS	Quality of Service
RISC	Reduced Instruction Set Computer
RMI	Remote Method Invocation
RPC	Remote Procedure Call
RTOS	Real Time Operating System
SHA	Secure Hash Algorithm
SICP	Simple Imago Control Protocol
SITP	Simple Imago Transport Protocol
SMTP	Simple Mail Transfer Protocol
SNMP	Simple Network Management Protocol
SSH	Secure SHell
SSL	Secure Socket Layer
TCP	Transmission Control Protocol
TTL	Time To Live
ToS	Type of Service
UDP	User Datagram Protocol

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 an 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 MA systems have been implemented or are currently under development. Typical systems are ARA [1], Aglets [2], Ajanta [3], Concordia [4], D’Agent [5], Mole [6], Odyssey [7], Telescript [8], Tocomo [9], Voyager [10], *etc.*...

Fundamental issues related to the MA paradigm are 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 MA 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 MA 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 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 MA applications. Instead, 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 MA systems adopt some kind of communication models/protocols from traditional distributed systems while developing separate protocols for agent migration.

Processes in a distributed system must interact with each other using some kind of communication models to exchange data and coordinate their execution. 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/RMI)*, and *Distributed Event Handling*.

Message passing is used to support peer-to-peer communication patterns and is the most commonly adopted model in MA 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 which want to communicate with each other must establish a session before the actual communication can start. D'Agent supports text-based message passing. The location and identity of the receiver should be known by the sender. 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 which hides the remote location and acts as a reference of 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 MA systems support RPC/RMI paradigm such as Mole and Voyager. An argument against this paradigm is that under the new paradigm, mobile agents can move to any remote host for data and services. Why do we need RPC/RMI at all? Agents for Remote Action (ARA) attempt 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 are located at a specific place where it 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 MA systems have much in common with those event frameworks employed in GUI toolkits supported by Java and Tcl/Tk. Mobile agent systems such as Concordia, 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. Interprocess communication is dependent on the ability to locate the communication entities. This is the role of the naming services which 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 which 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 MA systems either do not provide the ability of automatically tracking moving agents, or overly constrain 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.

Briefly speaking, locating an agent is invoking a function of the form “where_is(A)” which should return the current address (or access point) of A . Researchers have recently proposed many schemes for designing such a function. Various approaches for storing, updating, and locating mobile agents are well addressed in [11,12].

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(A)” is not accurate if A moves to another host immediately after the answer is returned, which makes message loss unavoidable. To solve this problem, a snapshot broadcasting strategy [13] was proposed. However, this scheme requires agent communication and migration through a strict FIFO channel, inhibits the movement of the located agent during message passing, and involves a considerable system overhead.

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(*A*)” is ambiguous: *A* might still reside at that address, or *A* might have moved to another host and its location updating packet is on the way to home, or *A* might even have started to move at the same time a message is sent to its current location.

Forward pointers is a promising alternative for locating mobile agents. This scheme does not depend on a “where_is(*A*)” 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 a returned message has been received by the corresponding virtual object. 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.

Mailbox-based scheme [14] is a set of mobile agent tracking protocols proposed recently. In this approach, each mobile agent is assigned with a mailbox to buffer incoming messages. A mailbox is a mobile object and could be detached from its owner, but it is not an agent-like object in the sense that it can not make any decisions. Inter-agent communication follows the triangular manner such that the mailbox serves as the middle access point between sender and receiver. This scheme offers a 3D design space which includes *home* protocol, *forward pointer* protocol and *distributed registration* protocol. It also allows users to select a parameterized protocol ranging from unreliable communication to reliable communication. However, protocols which guarantee reliable message delivery require expensive synchronization and location registration.

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 MA 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, thereby avoiding remote communication. Furthermore, a communication model is usually implemented by a stack of communication protocols. It is also questionable whether such a protocol is actually necessary if we already have a simple, reliable agent migration protocol. In other words, can we utilize the mobility of agents to achieve powerful, reliable and flexible inter-agent communication?

Thesis Objectives

This thesis will investigate different existing MA systems based on a literature survey. We will present the IMAGO system and concentrate on the design and implementation of the Agent Migration Protocol. 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, our model is capable of coping with both agent migration and inter-agent communication.

The Imago Project is a self contained, fully functional mobile agent platform using an extended Prolog programming language to create agents. Those agents have mobility access according to some security rules. An Imago is a mobile agent designed to run on MLVMD platform. According to [15], there are three distinct kind of Imago: *stationary Imago*, *worker Imago*, and *messenger Imago*. Those Imago may freely evolve and reproduce within the boundary of the server. Compatibility with other Mobile Agent systems has never been an objective and the MLVM suite could easily be rename by *Yet Another Mobile Agent System* (YAMAS) if the project was

in need for a new name. In fact, MLVM is a new system totally self contained. The project includes a compiler, *imagoc*, an assembler, *masm*, a server daemon, *mlvmd* and a set of two client interfaces (a command line based interface named *mlvmc* and a Java based interface) to access the *mlvmd* server. The compiler has been designed and developed by my colleague Hua Liang. The assembler has been created by Dr. X. Li, our supervisor, as well as the LVM core. Initially, my role in the project was to design and implement a migration protocol to be used by the LVM core, but my task list kept on growing and quickly included the port of the LVM engine into a multi-threaded version (renamed MLVM) and many other tasks. I also hacked extensively in the assembler code, created the server along with its command line management interface, combined all components into a single package and source control, using CVS, on the entire project.

2 RELATED WORK

Numerous MA systems have been implemented or are currently under development. A short list is presented on table 2.1, along with a brief overview of the agent programming language and migration type for each individual case. From those, we retained four systems that implement many very important features of mobile agent computing. Namely; CBorg, Concordia, D'Agent and Messenger-M ϕ will be discussed in depth in this section as they contain some of the basic ideas we decided to implement and some other ideas that we found worth noting.

2.1 CBorg

CBorg agents ('Migrobs') are written in a high level Scheme-like programming language, called Pico [16]. Pico is an educational language used at the Vrije Universiteit, Brussels Belgium, to teach concepts of programming languages to first-year computer science students. Small and portable by design, Pico allowed us to experiment with new concepts and language constructs for mobile multi-agents¹. On top of Pico CBorg defines an integrated development environment (IDE) that supports the interactive writing and designing of agents. When the CBorg interpreter is started, it connects to another CBorg interpreter, which yields an interconnected network of agent systems. This interconnection of mobile multi-agent systems is called the mobile multi-agent infrastructure. Currently, the CBorg mobile multi-agent architecture features:

Strong mobility: Meaning that an agent can migrate between agent systems while it is executing. Strong mobility is seldom found in current, Java-based, agent

¹Because of the experimental nature of CBorg, there is no intention to be compatible with existing languages, such as Java.

	Java	C	C++	Prolog	Scripting	Migration
ARA	Yes	Yes	Yes	No	Tcl	Strong
Aglets	Yes	No	No	No	No	Weak
CBorg	No	No	No	No	Pico	Strong
Concordia	Yes	No	No	No	No	Weak
D'Agent	Yes	No	No	No	Tcl, Scheme	Strong
Grasshopper	Yes	No	No	No	No	Weak
Jini	Yes	No	No	Yes	No	Strong
JumpingBeans	Yes	No	No	No	No	Weak
MAGNET	Yes	No	No	No	No	Weak
Messengers-C	No	Yes	No	No	No	Strong
Messengers-M ϕ	No	No	No	No	M ϕ	Weak
MiLog	No	No	No	Yes	No	Strong
Mole	Yes	No	No	No	No	Weak
Odyssey	Yes	No	No	No	No	Weak
Telescript	No	No	No	No	Telescript	Strong
Voyager	Yes	No	No	No	No	Weak

Table 2.1
Technical Details

systems due to some technical drawbacks of Java. Because CBorg has the ability to recreate the complete computational state of a running process, including its runtime stack, strong mobility is one of its standard features. Furthermore, the agents have the ability to store and retrieve computations as variables (continuations) and pass these to other agents (remote continuations).

An easy to use agent communication layer: This communication layer, which consists of a serializer and an object call-like syntax, allows agents to pass messages to each other. Agents always communicate in an asynchronous fashion. The reasoning behind this design decision is the notion of being ‘autonomous’: an autonomous agent should be designed as a separate entity, sending messages to, and receiving messages from other agents, not as an entity which transfers its control flow to other agents.

A hierarchical naming system: Every agent has a human-readable name, which is always used to reference it. The naming system favors late binding, in the sense that agents bind to each other at execution time, not at compile time, as it is partially done with objects.

A high-performance location-transparent distribution layer: An agent can send messages to other agents, without having to know where the other agent resides. For example, if agent ‘Alice’ talks to agent ‘Bob’, and ‘Bob’ migrates to the agent system at the end of the universe, CBorg keeps on routing messages between Bob and Alice using the path between them.

Resource Transparency: All resources in the mobile multi-agent system (disks, user interfaces and so on) are represented as static agents (which cannot migrate). So whenever an intelligent agent migrates, it stays connected to the resources it was using.

Garbage Collection: CBorg has a state of the art, highly efficient garbage collector incorporated into the system. The garbage collector uses a 32-bit cell memory, with only 2 bits reserved for bookkeeping!

Agent Synchronization: This is performed by using a rendez-vous between multiple agents. This rendez-vous can be in time and/or in space (synchronize at a certain computer). The primitives themselves are based upon CSP [17], with the exception that CBorg uses unification instead of sequenced statements as guards.

We now will discuss four of the above items: the location-transparent distribution layer, strong mobility, synchronizing agents, and agent communication. As an illustration we use the example of a webserver throughout the discussion. The web server consists of a varying number of agents, depending on the required functionality. The main agent is called *WebDispatcher*, it dispatches incoming requests to appropriate secondary agents, which generate HTML for the given request. Incoming requests are transformed into a *WebRequest* agent, which announces itself to the *WebDispatcher*. The actual CGI-like processes (*UrlHandlerAgents*) announce themselves to the *WebDispatcher* by subscribing on a number of URLs.

One of the most immediate problems encountered was the total lack of support for mobility in existing distribution protocols. For example: if we implement an *UrlHandler* which can migrate from one machine to another, how does the *WebDispatcher* communicate with this ‘jumping *UrlHandler*’? A common solution to this is to let the programmer of the *WebDispatcher* solve the problem of localizing the *UrlHandler* and managing a communications channel to this agent. This solution is not acceptable because the problem can be solved by the distribution layer.

CBorg implements a distribution layer that is able to name agents uniquely, worldwide (for example, a counter *UrlHandler* called `prog/wernersCounter`), in a location-transparent fashion, while taking mobility into account (if the agent migrates to another machine, its name should remain the same). This naming system is

used to reference agents and to send messages to them. To provide this functionality, CBorg merges name server and router into one entity.

Sending Messages

The solution implemented in CBorg, is based on abolishing the distinction between the name of an agent and the address of an agent. Instead of resolving the name of an agent to find its place, the system immediately route messages to an agent based upon the receiver's name. There is no longer a statically interconnected routing infrastructure and a separate, statically interconnected naming infrastructure; instead there is one hierarchical infrastructure in which agents are named and messages routed between them.

To send a message there is no need for a lookup request to a name server. Instead, the agent sends the complete message to the name server, which 'routes' the messages further to the next name server. For example: the figure 2.1 contains a hierarchical interconnection of name servers/routers.

If the agent `prog.vub.ac.be/WebDispatcher` wants to send a message to an agent `belnet.ac.be/WernersUrlHandler`, it can pass the message to the local name server, which is in this case `vub.ac.be`. At the moment `vub.ac.be` receives a message, it sends the message through to `ac.be`. Then `ac.be` sees the message for `belnet.ac.be/WernersUrlHandler` and passes it to the right downlink, in this case, `belnet.ac.be`. There it is immediately delivered to `belnet.ac.be/WernersUrlHandler`.

Migrating Agents

Migrating an agent in this setup is similar to sending a message, with the difference that every router/name server should interpret the movement of an agent. This means that whenever an agent passes through a node, the node should update its routing-tables to point to the agent's new direction. Note that the routing table does not point to the new location, but instead show the link to where the agent has migrated. This

guarantees that no updates are needed in this node when the agent migrates further. If an agent moves through a node to its original position, the corresponding rules are deleted. To keep tables smaller, CBorg uses a system of wild cards to annotate groups and clusters of agents. For example, if we migrate `prog.vub.ac.be/WebDispatcher` to the system `belnet.ac.be`², as shown in figure 2.1, the system takes the agent's state and sends it as an agent message to the local node `prog.vub.ac.be`. This node sends the agent to its uplink, `vub.ac.be`, and updates its routing table to let `prog.vub.ac.be/WebDispatcher` point towards the uplink. Then `vub.ac.be` sends the agent to `ac.be` and keeps a rule for agent `prog.vub.ac.be/WebDispatcher` which points to its uplink. The node `ac.be` sends the agent to the right downlink, in this case `belnet.ac.be` and makes a rule to point out the agent's new location. If an agent migrates while messages are being sent to it, these messages follow the agent on its path and arrive at the right location.

Figure 2.1. Migrating An Agent Over The Network.

²We have to admit that a complete migration of the WebDispatcher agent is rather unlikely, instead we expect to see a migration of the UrlHandler Agents. Nevertheless the above is possible and still useful for demonstrative purposes.

Strong Mobility

After migration, the agent should proceed seamlessly with what it was doing before it was moved. In our example, we want to migrate the WebDispatcher agent. In order to obtain this result, three actions should be undertaken to prepare, guide and complete the actual migration. First the agent's complete state has to be encapsulated; next, this capsule has to be transferred and, finally the agent needs to be restored and re-activated in its new environment. The CBorg interpreter (originally called Pico) is written in a thunk-based way (meaning that every evaluation step is stored in the working memory of the running program) and has the capability of serializing the data store. CBorg uses serializing to implement strong migration.

Even if an agent is in the midst of being evaluated, it can simply have its evaluation process interrupted whenever the stack is consistent, i.e. after the current continuation thunk has stopped executing, and when all global variables are saved in the data store. The system can easily serialize the entire state of the interrupted agent. Which also includes the computational state, and send it to another location, after removing the process from the interpreter control. At the receiving end, the system de-serialize the agent and start a process which uses the freshly de-serialized computational state.

Serializing agents consists of traversing the data graph and storing everything encountered, including local objects. The one exception is the link with the operating system, which is simply marked as being the root environment. In this way it is possible to integrate agents into their new agent environment upon their arrival at a new location.

Synchronizing Agents

Now we can easily write WebDispatcher and UrlHandlers. We can name them and they can communicate and migrate freely. But there are still some problems left, such as synchronization between two agents. To use our example: how could the

WebDispatcher agent wait for an answer from one specific UrlHandler? To be able to synchronize processes, CBorg defines an underlying fundamental communication model including synchronization between processes. The models of [17], [18] and [19] are only valid in small-scale systems because they either use shared memory with locks on variables, or the granularity is too coarse. CBorg defines a synchronization mechanism which allows for synchronization between processes of different owners, for synchronization while using a network with unpredictable delays, and for data communication between the synchronizing processes. This *sync-primitive* and first class continuations are used in this effect. The *sync-primitive* behaves according to the table 2.2:

Bob	Alice
sync("Alice",[a,10])	...
Waiting for Sync	...
...	sync("Bob",[20,b])
Execution continues with a=20	Execution continues with b=10

Table 2.2
CBorg *Sync-primitive*.

The arguments of the sync are as follows:

- The first argument specify the agent with which we wish to sync. This can be done by explicitly naming that agent, by using a wild card, or by providing a table containing all possible agents.
- The second, optional, argument is a pattern, which places restrictions on which agents may synchronize. This way it is possible to choose the agent to synchronize to by other means than solely its name (ie: a message pattern).

Patterns are unified if, and only if they match and there are no free variables after the match. When unification occurs, the agents in question synchronize. The pattern

given to the *sync* operation can contain literals (numbers, strings, booleans, void), variables and tables. Numbers match when they are numerically equal. Strings match when they are equal using a *strcmp*. Bound variables match if their values match. Free variables match any expression (such as 12, 15, "johan", [10,20,30] and others) and are bound afterwards. Tables match if all the sub-expressions match. For example the table [10,20] matches with [10,20] and [10,a] but not with [20,20]. Finally there is a wild card called 'any' which matches with anything and forgets the result. The use of patterns and unification is clear:

- Agents immediately have two-way communication between them. In the example, when Alice continues after the *sync* operation, Alice knows that Bob continues with a=20.
- It allows for a certain selection of who can synchronize. There is no need to restrict synchronization between two agents only. For example, the WebDispatcher agent can synchronize with all its agents in one statement.
- Unification is a more dynamic guarding system than CSP; CSP even has more problems because of roll-backs. Unification does not need this.

Agent Communication

CBorg agents can send messages to each other using a fairly obvious mechanism. Inspired by the ease of use of RPC implementations, a CBorg agent refers to a remote agent using its name, and sending a message to an agent is similar to sending a message to an object. For example:

```
a:agent("belnet.ac.be/WernersUrlHandler")
    // creates a reference to the agent WernersUrlHandler
a.showHtml("~werner/index.html")
    // by sending this message the agent will display
    // page ~werner/index.html on the local screen.
```


Sending messages in this fashion allows for asynchronous delivery of a message from one agent to another. The message send command is asynchronous, execution of this instruction solely places the message in the message delivery system. The call immediately returns the value 'void' and the agent continues its execution. This implies that the sender does not wait for the message to actually arrive, which speeds up program execution through bypassing the large delays that can be possible in wide-area networks.

All parameters passed to a remote procedure are automatically serialized. In this process all local objects remain local and are passed as a reference. The main motivation for passing objects by reference is that it allows for easy implementation of call-back functions and objects, as is illustrated in the next small example. WernersUrlHandler is the receiving agent, which performs a call-back to the second agent, the WebDispatcher. After creating a call-back procedure (Display), WebDispatcher calls WernersUrlHandler. The result of the computation is sent back as a parameter of the Display procedure.

```

// in prog.vub.ac.be/WebDispatcher
Display(text):: [...] // Displays the given text
GenerateHtml(Url)::
{
    handler:findUrlHandler();
    // find the agent who will handle this URL.
    handler.generateHtml( Url, agentself())
    // ask the agent to generate HTML and send
    // it back to myself. (hence the agentself parameter)
}

// in belnet.ac.be/WernersUrlHandler
generateHtml(Url, Callback)::
    Callback.Display("<HTML >Online and kicking !</HTML>");

```

Comments

Because the CBorg designer felt that the current infrastructures do not provide adequate support for mobile components. Therefore, they implemented the CBorg mobile agent infrastructure, to allow them to experiment with mobile components. Some of the more notable features of CBorg which aid in implementing mobile components are a location-transparent distribution layer, strong mobility, the ability to synchronize agents, and easy agent communication. Also, a number of design rules created, aid in writing components: Asynchrony, Autonomy, Modularity, and using State-based components in the CBorg system.

2.2 Concordia

Simply speaking, a Concordia system includes: a Java Virtual Machine, a Concordia Server, and Concordia Agents(at least one). The Java Virtual Machine, which is a standard environment, can be on any machine. The Concordia Server is a Java program which runs there, and at any other nodes on the network where agents may need to travel. The agent is also a Java program which the Concordia Server manages, including its code, data, and movement.

Generally, there can be many Concordia Servers and they are distributed each of the various nodes of a network, both user and server nodes. The Concordia Servers are aware of one another and connect on demand to transfer their agents in a secure and reliable fashion. The agent initiates the transfer by invoking the Concordia Servers methods. The Concordia Server inspects an object called the Itinerary, created and owned by each agent, to determine the appropriate destination. That destination is contacted and the agent's image is transferred, where it is again stored persistently before being acknowledged. In this way the agent is given a reliable guarantee of transfer.

After being transferred, the agent is queued for execution on the receiving node. This happens promptly, but possibly subject to certain administrative constraints.

When the agent again begins executing, it is restarted on the new node according to the method specified in its itinerary, and it carries with it those objects which the programmer requested. Its security credentials are transferred with it automatically, and its access to services is under local administrative control at all times.

The work that the agent performs depends on its purpose, that is, the code which it was programmed to execute. Generally, agents have several components, just as any program has. An agent might start interactively, by prompting the user for search information, then may travel to a server to perform the query. As its methods complete, the itinerary causes the agent to be moved to other Concordia nodes. Therefore, agents with different purposes will typically have different itineraries.

In all cases, Concordia provides a robust and highly reliable framework for the development and execution of secure, intelligent, mobile agent applications. Furthermore, Concordia's collaboration and Service Bridge features provide an ideal framework for enabling Cooperative Information Gathering, a multi-agent based approach for combining information from multiple, heterogeneous, information sources. The need for an efficient approach to information gathering is becoming increasingly important because of the changing and growing amount of information available in the World Wide Web and other complex information spaces. the Concordia agent is autonomous and self-determining in its operation. In this way, it is unique since it is in control of its own itinerary.

Architecture of Concordia

The Concordia system is made up of numerous components, each of which integrates together to create the full Mobile Agent framework. It is similar to a number of existing agent infrastructures and tool kits with respect to its support for the basic communication plumbing that is required for agent mobility. The Concordia Server is the major building block, inside which the various Managers reside. Certain components have a user interface, such as the Administrator. In any case, each component

is responsible for a portion of the overall design, in a modular and extensible fashion. Concordia was developed based on Java language. It consists of the followings:

Figure 2.2. Concordia Architecture.

Concordia Server: The name of the complete Concordia component installed and running on a machine in the network. It serves as a communication server is responsible for agent transferring across the networks. At first, this transferring is activated by agent's invoking the server's methods. Then the agent travels the network until it reaches its destination. In the end, it brings the required data back.

Agent Manager: Provides the communications infrastructure that allows agents to be transmitted and received by nodes on the network. Moreover, agent mobility mechanism extends beyond the functionality provided in other Java-based agent systems. It offers a flexible scheme to dynamically invoke arbitrary method entry points within a common agent application. It abstracts the network interface so programmers do not need to know any network specifics nor to program any

network interfaces. The Agent Manager Server also manages the life cycle of the agent. It handles agent creation and destruction, and provides an environment in which the agents execute.

Administrator: Responsible for the network administration of Concordia. It resides on a separate Java virtual machine, and can cooperate with Concordia services which are running on the various nodes under administration. The Administration Manager manages all of the services provided by Concordia, including Agent Managers, Security Managers, Event Managers, etc. The Administration Manager supports remote administration from a central location, so only one Administration Manager is required in the Concordia network, although more can be employed as desired. The Administration Manager has a user interface component which is its primary means of use. The Administrator also monitors the progress of agents throughout the network and maintains agent and system statistics.

Security Manager: Security is one of the most important criteria of mobile computing. In concordia, the security manager is responsible for identifying users, authenticating their agents, protecting server resources and ensuring the security and integrity of agents and their accumulated data objects as they move. The Security Manager is also responsible for authorizing the use of dynamically loaded Java classes to satisfy the needs of agents. The Security Manager has a user interface component, in order to configure and monitor the security attributes of the various users and services known to Concordia. This user interface function is integrated into the Administration Manager interface. Security credentials used by the Security Manager may come from a number of sources. For secure, self-contained systems, no credentials may be needed. For systems that traverse public or semi-public networks, encryption may be required but credentials may need only reflect user identity, such as user name or group id. For fully fledged agent systems deployed on the Internet, strong authentication

and security can be provided from external authorities such as Version. All these security levels can be supported by Concordia's Security Manager.

Persistence Manager: Required to ensure that the Concordia agents can recover from a system crash successfully. It maintains the state of agents in transit around the network. As a side benefit, it allows for the checkpoint and restart of agents in the event of system failure. Additionally, it can checkpoint objects upon request by agents, to provide finer granularity of reliability guarantees for critical procedures. The Persistence Manager is completely transparent in its operation, that is, neither the agents nor the administrator need control or monitor its operation. However, management access is available if needed.

Event Manager: Concordia can provide two kinds of asynchronous distributed events: selected events and group oriented events. The Event Manager handles the registration, posting and notification of events to and from agents. The Event Manager can pass event notification to agents on any node in the Concordia network. The Event Manager works in conjunction with the Concordia Server to distribute events as needed. An important function of the Event Manager is to support Concordia agent collaboration.

Queue Manager: Queues are needed in the Concordia to store message produced by the Concordia agents. The Queue Manager is responsible for scheduling and possibly retrying the movement of agents between Concordia systems. These features include the maintenance of agents as they await the opportunity to perform their work, maintaining their persistent state as they enter and leave a system, and retrying as necessary when Concordia systems are disconnected from the network. The Queue Manager provides the mechanism for prioritizing and managing the execution of agents entering Concordia nodes.

Directory Manager: Enables mobile agents to locate the application servers they wish to interact with on each host. It maintains a registry of application services

available on each host it manages. A Concordia distributed system may be configured to include one or more Directory Managers (at most one per host). Application servers export their services to mobile agents by registering them with the Directory Manager. Mobile agents obtain references to application servers via the Directory Manager's lookup operation. The Directory Manager provides a naming service in the Concordia network. The administrator may configure the name service in a number of ways, chosen according to the needs of the programmer and services. The Directory Manager may consult a local name service or may be set up to pass requests to other, existing name servers.

Service Bridge: Provides the interface from Concordia agents to the services available at the various machines in the Concordia network. It comprises a set of programming extensions to provide access the native APIs as well as interfacing these to the Directory Manager and Security Manager.

Agent Tools Library: This library provides all the classes needed to develop Concordia Mobile Agents. This of course includes the Agent class, and others derived from Java base classes, with interfaces to the Concordia infrastructure.

Mobile-Agents in Concordia

A agent is defined as an independent software program which runs on behalf of a network user. It may run even if the user is disconnected from the network. Some agents run on specialized servers, others run on standard platforms. Concordia Mobile Agent can perform work on behalf of the users, such as collecting information or delivering requests. Agents are highly mobile and their mobility can extend to a number of local as well as wide area networks.

Concordia allows the creation of Mobile Agent programs written in the Java language. These programs use services to move about in a network of distributed machines and to access services available on them.

Security is an integral part of the Mobile Agent framework, and it provides for secure communications even over public networks. Agents carry users' credentials with them as they travel. These credentials are authenticated during execution at every point in the network. Agents and their data are fully encrypted as they traverse the network. All this occurs without programmer intervention.

Since the agent objects are composed of a combination of code and data, object mobility means the network transportation of both code and data. Concordia also provides interfaces allowing agents to create other agents and to clone themselves. As stated earlier, agent mobility is accomplished by the Queue Manager. In the Mobile Agent, the following information should be provided: (i) destinations, (ii) itineraries and (iii) methods.

How does an agent travel? It is described by its Itinerary. The Itinerary is a data structure which is stored and maintained outside of the agent object itself. The Itinerary is composed of multiple Destinations. Each Destination describes a location to which an agent is to travel and the work the agent is to accomplish at that location. In the current implementation, location is defined by the host-name of a machine on the network and the work to accomplish is by a particular method of the agent class.

There are some important characteristics of this Itinerary model worth noting. The Itinerary is a completely separate data structure from the agent itself. For flexibility reasons, the system allows agents to modify their Itineraries at runtime.

Mobility of an agent's data was accomplished using the Java Object Serialization facility. Transfer an agent state is a matter of serializing an agent's data down into a format suitable for network transmission, transmitting the data in this format, and then de-serializing the data back into the original agent. This is very similar to the mechanism used by Java Remote Method Invocation (RMI) for passing an object by value between distributed objects. RMI itself does not provide true object mobility as it does not serialize the object code. It requires that the code for any objects passed by value be pre-installed on both sides of the network connection. Java's Object Serialization features provide an almost transparent mechanism by which Java objects

can be serialized into data streams which provide suitable technology for implementing agent mobility.

Concordia's collaboration framework facilitates interactions by enabling multiple agents to work together to solve complex problems. Agents may perform computations, synchronously share results, and collaboratively determine any changes to future actions. Concordia employs a simple programming paradigm for this type of collaboration. The goals of the collaboration framework include: (i) A simple programming interface for synchronous collaboration. (ii) Asynchronous notification of exceptional conditions via events. (iii) Reliable and robust implementation utilizing proxy objects to shield agents from the effects of software failures within the collaboration framework. (iv) An infrastructure that enables location transparent inter-agent communication.

Agents within an application may form one or more collaboration units, known as agent groups. Concordia provides base classes for collaborating agents and agent groups. AgentGroups are implemented as distributed objects which export a simple interface to CollaboratorAgents. These agents hold remote references to AgentGroup distributed objects and access them via Java's Remote Method Invocation (RMI) facility.

AgentGroup collaboration is implemented as follows: The AgentGroup abstraction provides the distributed synchronization. Each application need only supply its own implementation of the interface to analyze the collective results of the agents in the group and to allow each agent to adapt its behavior based on those results. Both the synchronization point and invocation of `analyzeResults` are encapsulated within the AgentGroup's `collaborate` method. This distributed synchronization scheme requires that each agent "arrive" at the collaboration point before collaboration may commence. Hence, it is ideally suited to applications that subdivide a complex problem into sub-tasks then correlate their results. When each agent arrives at the collaboration point, it posts the results of its computation to the AgentGroup and blocks until all the agents in the group arrive. The AgentGroup class collects the results of

the computations. When all agents, in a group, arrive at a collaboration point, the AgentGroup invokes `analyzeResults` in each agent, passing it the collective result set. The AgentGroup abstraction supports both parallel and serialized execution of the analysis stage of collaboration.

The AgentGroup also uses time-outs to detect potential deadlocks. Note that since AgentGroup collaboration is designed for closely coordinated agents, deadlocks are generally caused by programming errors. Hence, the AgentGroup does not need to use a more sophisticated scheme for deadlock detection or avoidance. An additional benefit of AgentGroup collaboration is that it enables location-transparent inter-agent communication. As each agent migrates, it carries a remote reference to an AgentGroup distributed object and utilizes the AgentGroup as a gateway for communicating with the other members of the group.

As mentioned earlier, AgentGroups facilitate both synchronous collaboration and asynchronous notifications. This is possible because the AgentGroup object derives from the `PersistentEventGroup` object. AgentGroups forward any events they receive from their members to the remainder of the group. Occasionally, they may also initiate events that they deliver to the group.

Comments

Concordia offers a complete frame work for the development and management of network-efficient mobile agent applications. The design goals of Concordia have centered on providing support for flexible agent mobility, agent collaboration, agent persistence, reliable agent transmission, and agent security.

Concordia's agent mobility mechanism extends beyond the functionality found in current Java-based agent systems by offering a flexible scheme for dynamic invocation of arbitrary method entry points within a common agent application.

Through the use of proxies and object persistence, Concordia provides a highly robust environment where applications can gracefully recover from system or network

failures. Using a transaction message queuing system, Concordia provides for reliable network transmission of agents even over unreliable network connections.

The infrastructure provided by Concordia extends the standard security mechanisms of the Java language to provide an identity-based system where the rights given to an agent are determined from the identity of the user who launched the agent. Concordia also protects an agent and the information it carries from tampering when stored on disk or when transmitted over a network connection.

2.3 D'Agent

Like all mobile-agent systems, the main component of Agent Tcl is a server that runs on each machine. When an agent wants to migrate to a new machine, it calls a single function, `agent_jump`, which automatically captures the complete state of the agent and sends this state information to the server on the destination machine. The destination server starts up an appropriate execution environment (for example, a Tcl interpreter for an agent written in Tcl), loads the state information into this execution environment, and restarts the agent from the exact point at which it left off. Now the agent is on the destination machine and can interact with that machine's resources without any further network communication. In addition to reducing migration to a single instruction, Agent Tcl has several important features:

Simple architecture: The simple, layered architecture supports multiple languages and transport mechanisms. The main language is Tcl. The main transport mechanism is TCP/IP.

Security: Agent Tcl protects individual machines against malicious agents—agents that try to access or destroy restricted information or consume too many machine resources. It also protects groups of machines controlled by a single organization.

Docking system: The docking system lets an agent transparently jump off a partially connected computer (such as a mobile laptop) and return later, even if the computer is connected only briefly.

Interagent communication: Agents communicate with either low-level mechanisms (message passing and streams) or high-level mechanisms (RPC) that are implemented at the agent level atop the lower level mechanisms. All communication mechanisms work the same whether or not the communicating agents are on the same machine.

Figure 2.3. The architecture of Agent Tcl. (a) The core system has four levels: transport mechanisms, a server that runs on each machine, an interpreter for each supported agent language, and the agents themselves. (b) Support agents provide navigation, communication, and resource management services to other agents.

Core System

As figure 2.3 shows, Agent Tcl's architecture has a four-level core system and an agent-level support system. At the lowest level of the core system, Figure 2.3a, is an interface to each available transport mechanism. The next level is the server that runs on each machine. The server keeps track of the agents running on its machine, it provides the low level inter-agent communication facilities (message passing and

streams), receives and authenticates agents arriving from another host, and restarts an authenticated agent in an appropriate execution environment. The third level of the architecture consists of the execution environments, one for each supported agent language. Agent Tcl supports Tcl, Java, and Scheme, so its execution environments are simply a Tcl interpreter (Tcl 7.5), a Scheme interpreter (Scheme 48), and the Java virtual machine (Sun JDK 1.2). For each incoming agent, the server starts up the appropriate interpreter. The last level of the architecture comprises the agents themselves, which execute in the interpreters and use the facilities provided by the server to migrate from machine to machine and to communicate with other agents. Agents include both moving agents, which visit different machines to access needed resources, and stationary agents, which stay on a single machine and provide a specific service to either the user or other agents. From the system's point of view, there is no difference between the two kinds of agents, except that a stationary agent has authority to access more system resources.

To add a language to Agent Tcl, programmers simply extend the interpreter to provide

- state-capture routines that capture and restore the state of an executing program, and
- an interface to the agent servers.

In the Tcl interpreter, for example, the state-capture routines capture and restore all defined variables and procedures, the procedure-call stack, and the control stack. The interface to the servers is a set of Tcl commands, such as `agent_begin` and `agent_jump`, which are provided as a standard Tcl extension. The `agent_jump` command calls the state-capture routines to capture and restore the state of an executing Tcl agent. Similarly, in Java, the state-capture routines capture and restore the state of a single Java thread (including all accessible objects). The interface to the servers is a special Java class. Finally, in Scheme, the state-capture routines capture and

restore the current continuation (the rest of the program), and the interface to the servers is a set of Scheme functions.

Most of the interface between the interpreters and the servers is implemented in a C/C++ library and shared among all interpreters. The language-specific portion is just a set of stubs that call into this library.

Agent-Level Support

The agent servers provide low-level functionality. As Figure 2.3b shows, all other services are provided at the agent level by dedicated service agents. Such services include navigation, high-level communication protocols, and resource management. Both the docking system and Agent RPC (described later) are implemented entirely at the agent level.

Sample Agent

Figure 2.4 shows a simple Agent Tcl agent written in Tcl. The agent's task is to make a list of all users logged onto some Dartmouth machines and then show this list to its owner. The agent has several important parts:

agent_begin: The agent registers with Agent Tcl through the server on its home machine (Bald).

agent_jump \$machine: The agent migrates sequentially through the machines of interest (Muir, Tenaya, and others not shown). It continues executing from the point of the jump on each successive machine. On each machine, the agent executes the Unix who command (exec who) to obtain the user list.

agent_jump \$agent(home): Once the agent has migrated through all the machines, it migrates one last time to return to Bald.

display results: Once on Bald, the agent displays the complete user list to its owner (not shown).

agent_end: The agent tells Agent Tcl that it has finished.

Figure 2.4. A simple Tcl agent that figures out which users are logged onto some set of machines. Bald, Muir, and Tenaya are machines at Dartmouth College. The agent starts on Bald.

Although this agent performs a simple task, any agent that migrates sequentially through one or more machines has the same general form. The `exec who` command can be replaced with any desired local processing. Some agents will need learning and reasoning capabilities. Agent Tcl does not provide such capabilities directly, but an agent is just a program written in Tcl, Scheme, or Java, so it can include and use any existing libraries.

Security

Security, of course, is a critical issue in any mobile-code system. Agent Tcl currently protects machines from malicious agents (both individual machines and groups of machines under single administrative control), but does not protect agents from malicious machines. We give a brief summary of the current implementation here; a detailed description and the future plans are given elsewhere. [20] [21]

The security mechanism has three major features:

- Agents and messages sent between machines are encrypted, which maintains agent privacy.
- Agents and messages sent between machines are signed, which authenticates the agent to the new host.
- Resource managers control access to system resources.

Each resource (CPU, memory, file system, screen, network, and so on) has a stationary agent that acts as a manager. For Tcl agents, each visiting agent is run in an *untrusted* Tcl interpreter, and all resource accesses are trapped into a separate, *trusted* interpreter. The trusted interpreter asks the appropriate resource manager to determine if the visiting agent should have access to the resource. For example, the memory manager might limit an agent to 100 Kbytes of memory. The trusted interpreter then enforces the manager's policy decision, either proceeding with the resource access or throwing a security exception back to the untrusted interpreter. In addition to absolute limits on resource use, there are plans to use a currency-based model in which agents purchase resources from the managers, thus limiting their total resource use even across administrative domains [21]. The prices will vary according to supply and demand and the changing priorities of agents and servers.

The same resource managers are used for all agents; only the enforcement mechanism differs among agent languages. In Java, for example, a special security manager class contacts the resource managers and enforces the policy decisions, rather than a separate trusted interpreter.

Docking System

When a mobile agent tries to return to its home machine with final results, the machine might be disconnected. Thus, the agent must have some way to determine when the home machine reconnects. A simple approach is polling (try, timeout, sleep,

try again, and so on), but polling wastes network resources and will fail outright if the home machine reconnects for only brief periods. For this reason, AgentTCL devised a docking system, Figure 2.5, that pairs each laptop computer (“laptop” meaning any mobile device) with a permanently connected dock machine. When a mobile agent is unable to migrate to a laptop (laptopX), it waits at the laptop’s dock machine (laptopX_dock). When the laptop reconnects, it notifies the dock machine of its new network address, and the dock machine forwards all waiting agents. Although

Figure 2.5. Docking system. Each laptop is paired with a permanently connected machine, where agents wait for the laptop to reconnect. Here laptop refers to any partially connected machine.

only certain machines act as docks, all machines have a *dock master*, an agent that maintains a queue of waiting agents. The queue is always stored on disk rather than in memory. On the dock machine, the queue contains the agents waiting to visit the laptop; on the laptop itself, the queue contains the agents waiting to leave the laptop.

Inter-agent Communication

Agent Tcl provides message passing and byte streams at its lowest level. Higher-level communication mechanisms, which make many applications much easier, are implemented at the agent level using message passing or streams. Our most important high-level mechanism, Agent RPC [22] is similar to traditional RPC [23]. Agent RPC lets two agents communicate through the procedure-call abstraction. The agents can be on the same or different machines, but usually will be on the same machine, since most client agents jump to the same location as the desired service. Programmers using Agent RPC begin by writing an interface in AIDL (Agent Interface Definition Language). The interface specifies the procedures a server agent provides to its clients. The programmer presents the AIDL specification to a stub compiler, which generates the Tcl procedures (called stubs) that let the client and server agents communicate. The client and server agents simply include these stubs with their application-specific code. In addition to accepting client requests, the server stubs register the server with a name server agent, which client agents consult when searching for a needed service. Although this basic structure is no different from that of traditional RPC systems, Agent RPC offers two unique advantages:

Flexible interface language: AIDL allows both default and position-independent parameters in interface definitions.

Client-server bindings: Bindings (or connections between compatible client and server agents) are based on interface matching rather than on names. Thus, a client agent can obtain the desired service from any server that supports the appropriate interface, rather than only servers that have a particular service name and version. In addition, a client agent can have multiple, simultaneous bindings to the same or different servers. Finally, a server agent can accept or reject a bind request according to any security information the agent system provides, such as the authenticated identity of the client agent's owner.

Comments

Agent Tcl allows the rapid development of efficient, robust distributed applications, particularly when mobile computers are involved. Despite its current capabilities, there still are several areas for future work. The two most important are network sensing and service directories. To best plan its route through the Internet, an agent needs information about the network's current state, such as its bandwidth, latency, and connectivity. The Dartmouth College researchers are developing sensing agents that glean this information from recent past communications with remote hosts [24].

Once an agent can roam the network, it needs to know where to go to find relevant services. A distributed "yellow pages" infrastructure is in construction, in which agents can advertise their services and client agents can look for agents that meet their needs [24]. These "yellow pages" are similar to the RPC name servers except that they are hierarchical and can contain arbitrary service descriptions. Eventually the RPC interface definitions will be included in the yellow page entries, eliminating the need for the separate RPC name servers. Other areas of future work include rewriting some of the service agents in the much faster Java language, making the agent servers more efficient, and extending the security model to protect agents from malicious machines. The Dartmouth College researchers are also investigating multi-destination jump support, and are integrating their inter-agent message-passing with the docking system so that messages go through docks when necessary. They are adding a persistent store so that an agent may leave most of its data (such as the results of a database search) at one host, carry a small amount of its data along with it, and yet be able to remotely access the stored data if necessary. Finally, in cooperation with other groups, they are continuing to develop applications that demonstrate the effectiveness of mobile agents in different network environments.

2.4 Messengers-M ϕ

The messenger paradigm is born from a new way of thinking computer communications. Instead of the classical sender/receiver model based on protocol entities that exchange and interpret protocol specific messages, Tschudin proposed in [25] a communication model based on the exchange of protocol unspecific programs, called the messengers. Hosts receiving messengers will then execute the messengers code instead of interpreting them as messages.

Architecture

All hosts involved in a messenger based communication share a common representation of the messengers and provide a local execution environment, *the messenger platform*. A messenger is executed sequentially, but several messengers may execute in parallel inside the platform. Platforms are connected through unreliable channels

Figure 2.6. Messenger Platform.

through which messengers are sent as simple messages or data packets. Figure 2.6 shows the basic elements of a messenger platform:

- A set of *channels* enables a messenger platform to exchange messengers with other platforms. The channels provide an unreliable datagram service which enables messengers either to reach entirely and correctly the neighboring platform or to be discarded or lost;
- Integrally arriving messengers are turned into *independent messenger processes* or *threads of execution*, also called *messengers* for simplicity. The platform does not interfere with their execution except for programming errors or unavailable resources. Also do other messengers have no means to stop or kill another messenger process without its consent;
- The executing threads are able to share common data inside a given platform by the means of a *dictionary* which contains pairs of keys and data values;
- *Process queues* are the low-level mechanism offered by a platform in order to allow messengers to implement basic concurrency control functionality such as synchronization of threads, mutual exclusion, etc. Process queues are FIFO queues of messenger processes. All threads of the queue are blocked except the thread at the head of the queue.

Platforms share (1) a common messenger programming language, used to express messenger behavior; and (2) a common external representation of the messenger, used for the physical exchange messengers.

The Messenger Platform

Two prototype of messenger platforms have been implemented that allowed to make first experiences with tangible messengers and to study in more details the basic elements of such platforms. Three elements have to be distinguished:

Messenger data format: In order to exchange messengers, two platforms must agree about the allowed “channels” as well as the data format used to express the messenger. This channel must offer a datagram service, thus it relies on a datagram delimitation mechanism. For practical reasons there will be more information inside a messenger than just the instruction sequence: version number, possibly a checksum and, for convenience, an arbitrary data field etc.

Messenger language and execution model: After having agreed upon the placement of the instruction sequence(s) inside a messenger datagram, two platforms also must “speak the same language”. The messenger languages will probably be interpreted ones, but messengers may also contain or consist of native CPU code (machine language code).

Conventions about local data resources: Finally, the importance of *conventions* is required across all messenger platforms understanding a given messenger language: because each platform will offer varying services (number and type of channels, file system and other data bases, code libraries etc.), it is important to have conventions about the naming of such resources as well as the mechanism used to discover and access them. Such an infrastructure resembles very much *type systems* e.g., the one found in the distributed object execution environment CORBA [26].

In the following, we briefly describe the two messenger platforms.

The $M\phi$ messenger Platform

The $M\phi$ (M-Zero) project started in December 1993; a first version of the $M\phi$ platform that runs on various UNIX systems and that contains a full interpreter of the $M\phi$ language was distributed in June 1994 [27] and can be fetched by FTP at `cui.unige.ch` under `pub/m0`. The $M\phi$ language is ASCII-based and closely resem-

bles Postscript although it does not contain any graphics related operators. Operators usually have single character names, thus making messenger programs very compact. The platform's common variable space is represented as a dictionary, enabling messenger processes to persistently deposit and fetch data values. The implemented messenger channels are based on raw ETHERNET and UDP.

Different major types of protocols have been implemented in the $M\phi$ environment: network exploration using ETHERNET broadcasts ("wave protocol"), alternating bit and Stenning's sliding window protocols, self fragmentation and de-fragmentation as well as a "remote console".

Some of these protocols were implemented in variants e.g., using code downloading techniques.

Although useful for exploring messenger programming styles and techniques, the $M\phi$ platform is currently incomplete insofar as it has very limited links to the world of applications and other operating system services. Thus, more work on the third element, the conventions, is needed, as well as the integration of standard services (windowing environment, file system) into $M\phi$. Still under study are strategies for managing inside $M\phi$ the various resources offered by the platform (CPU time, memory, transmission bandwidth etc.)

A SCHEME-based Messenger Platform

A case study was made on a functional messenger language, namely one that would be based on SCHEME. It resulted in a SCHEME interpreter named MSGR-S [28] that supports parallel interpreter processes. The new functions `setglob` and `getglob` enable the access to the common variable space, the function `exist?` can be used to check if a name already is in use. Most notable is the way new messengers are generated and submitted. While in $M\phi$ the programmer can refer to a special `convert-to-messenger` operator in order to build a messenger datagram from an arbitrary (code) string, this step is hidden by the MSGR-S interpreter: only a closure

can be submitted i.e., a lambda expression that has no parameters. Hence, the `submit` operator takes such a closure, (re-)constructs an “equivalent” ASCII string and sends it inside a datagram through the requested channel. This string is simply a SCHEME lambda expression that, after reception at the destination platform, is evaluated in order to obtain the function to start.

Basing messengers on a LISP-like language is for some programmers an essential asset, especially when messengers are used to implement intelligent agents in the context of AI. Although elegant, this *implicit* back-transformation of an internal closure to an equivalent ASCII-string has its price: on one side in terms of computing time and the problem of limited datagram length, on the other side in terms of expressiveness, possibly excluding the implementation of some classes of protocols within this environment. Both environments, *M ϕ* and MSGR-S, have shown the importance and necessity of a new programming style. The first time we started to code messengers we found it difficult to “overcome” the thinking in terms of static execution flows. Later on, however, once one is more familiar with the (possibly unreliably) spreading execution flows of messengers, it is fun to express communication tasks with messengers. Messenger-*M ϕ* already see specialized “interaction techniques” popping up which will form the basic elements of a new communications programming style.

Comments

The messenger paradigm is a communication model relying on instruction-passing instead of message-passing. Messengers are mobile threads of execution that can travel across a network. The basic requirements needed to achieve the messenger paradigm are: (1) all hosts involved in the communication must be provided with the messenger platform, (2) all platforms are able to interpret the same messenger language; and (3) unreliable messenger “channels” link the different platforms.

Distributed systems are built on top of a communication mechanism. The proposed communication by messenger paradigm is a low-level communication mech-

anism, which can be used at different levels of abstraction. We have shown that all kind of protocols can be implemented using this new paradigm: functionality of classical protocols, based on entities exchanging PDUs, can be realized without entities and without PDU exchange, using messengers. Distributed operating systems, which depend on communication protocols, can then be implemented using messengers. We have proposed a three layer architecture populated at the second layer with messengers; this layer constitutes a control interface between processes sitting at the third layer and the messenger platform. This architecture allows the building of more generic micro-kernels, by removing from them any hard-wired protocol. Finally, at the highest level, the messenger paradigm seems to be suited for the implementation of multi-agent applications.

Going further, the classical view of data traveling the network to be exchanged is no longer available according to the messenger paradigm where it is the code which travels the network to search for the data. Following these ideas, the well-known models of server/client and sender/receiver are no longer available, since a messenger based client will ask no request to a server, but goes itself to take the information and a messenger based server will not know which clients are taking the information. Implementing distributed applications with messengers implies a new way of thinking: distributed applications will no more be seen as composed of static entities exchanging data, but as a set of mobile threads which can propel themselves across the network to look for resources.

2.5 Summary

The four systems presented above have brought specific features to the present Imago project. For example, the way CBorg messages and agents are routed using the server's network name, has inspired the routing algorithm implemented in our migration protocol. Similarly, the "Migrobs" routing table, in the same system, is a less generic variation of the method Imago servers uses to remember the immediate

destination of migrated agents. The fine granularity of the Concodia architecture makes the system easier to maintain. In this system, the implication of Java's lack of support for strong migration are clearly shown (with the Itinerary object) and therefore, has being an additional motivation in our choice for prolog programming language. The last two systems have been particularly useful when we designed our system. The single instruction used in agent migration is similar to D'Agent's "agent_jump(...)" function call. The Messengers-M ϕ paradigm defines protocol has being implemented using dedicated agents. We try to reuse this idea when we discuss our inter-agent communication mechanisms using "messengers".

3 INTELLIGENT MOBILE MESSENGERS

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 – which not only provide an agent-based solution for inter-agent communication but also make the solution efficient and feasible.

3.1 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 which it belongs to.

Secondly, we assume that each application is bound to a *home* location which always exists during the life cycle 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.

Our next assumption is that a stack of protocols exists to support efficient, reliable and strong agent migration. There is no constraint on the freedom of agent mobility. 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 which 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 MA systems.

3.2 Locating Mobile Workers

As discussed earlier, 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 which 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.

As we explained in the previous section, 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 new born worker, either by creating or cloning and regardless of being born at the home host or a remote host, must register its birth place 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 life cycle.

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 X to host Y, its cached record at X is modified with the new state `MOVED_TO(Y)`. 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 Y back to X, its record at X 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 X to Y and then to Z. From there, the worker dispatches a messenger to a receiver at X. When the messenger arrives, the updating operation will change the worker(sender)'s record from the old state `MOVED_TO(Y)` to the new state `MOVED_TO(Z)`. Therefore, subsequent communications to that worker will be dispatched to Z directly.

In our model, we do not intend to have a network-wise “`where_is(A)`” lookup function for locating the *current* address of A. Instead, we use a local lookup function which returns a *possible* location of A. 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(Y)`, there is no guarantee that the worker we are looking for is still working at Y, 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.

3.3 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 *dispatch*, *move*, *clone*, *dispose*,

etc. (see table 4.1), another primitive which plays a major role in between workers and messengers is *attach*. This primitive, only used by messengers, binds the caller with the receiver Imago. 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);
        // context switch to another ready agent
    }
    else {
        unlock(local_cache);
        return 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 either 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 runnable 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*-like 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 attached messengers will be accepted by its receiver 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 (this problem will be further discussed in the next section). 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 a messenger is resumed by the receiver it was attached to. 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 the receiver most likely resides, 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 it leaves the tracking-down job to the messenger.

4 THE IMAGO SYSTEM

In this section, we present the design and the current state of our agent system infrastructure—the IMAGO System. The origin of the word *Imago* means:

An insect in its final, adult sexually mature, and typically winged state, or an idealized mental image of another person or the self.

WEBSTER's Dictionary

While in the IMAGO system, Imago are intelligent mobile agents. 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). From computer terminology point of view, the term IMAGO is an abbreviation which stands for Intelligent Mobile Agents Gliding On-line.

The IMAGO system consists of two major projects: the IMAGO Prolog—an agent application development kit based on Prolog, and the MLVM—a Multi-threading Logic Virtual Machine which presents a logic-based framework for the implementation of mobile agent server.

4.1 IMAGO-Prolog

IMAGO Prolog is the Prolog language extended with a rich Application Programming Interface (API) for implementing intelligent mobile agent applications. An IMAGO Prolog program consists of a set of *Imago* definitions and *module* definitions. Imago definitions serve to specify autonomous entities. An Imago definition is a named program block containing one or more procedures as its body from which intelligent agents can be created. A procedure defined in an Imago must be complete and private to its name space. This means that procedures of an Imago are not

accessible anywhere outside of the Imago. On the other hand, 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.

Similar to the syntax of a module declaration, the IMAGO Prolog introduces special directives to enclose the body of an Imago. However, unlike a module which requires an interface declaration for exporting its public procedures, an Imago only has a body definition because all procedures in the body belong to the Imago privately. There are three kinds of Imago: *stationary Imago*, *worker Imago*, and *messenger Imago*. Corresponding, we supply three pairs of directives for Imago definitions:

- *stationary(Name)/end_stationary(Name)*
- *worker(Name)/end_worker(Name)*
- *messenger(Name)/end_messenger(Name)*

Consider the following example: the worker directive *:- worker(buyer)*, where *buyer* is the atomic name of the worker definition (somehow like the name of a Java class), specifies that the Prolog text bracketed between this directive and the matching closing directive *:- end_worker(buyer)* constitutes the implementation body of *buyer*.

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

The atomic name (such as *buyer*) of an Imago definition serves three purposes:

- It is used to specify the beginning and termination of an Imago text.
- It is a predicate indicator which gives the execution entry of the Imago. From Prolog convention, a predicate indicator is a compound term P/N where P is the procedure name and N indicates the number of arguments (arity of P). Here we

define that the *entry procedure* of an Imago must be identified by the name the same as the one used the Imago definition and its default arity is 1. For instance, procedure “*buyer(Arg):- ...*” in the above Imago text is the *entry procedure* of the worker. The entry procedure will be called by the IMAGO runtime system. In other words, the runtime engine automatically provides a goal towards the entry procedure of an Imago when it has been loaded for execution.

- It is used to form the file name of the Imago byte-code file. Even though several Imago definitions can be placed in a single source file, the IMAGO Prolog compiler will compile each Imago independently and save its byte-code into a separate file (the file name is composed by the Imago definition name with an extension, such as *buyer.lo*).

Like other logic programming systems, the IMAGO Prolog API is presented as a library of built-in predicates. This library includes predicates common to most Prolog-based systems and new predicates extended for mobile agent applications [15]. Some predicates, especially agent-oriented predicates, are context sensitive, *i.e.*, the eligibility and effect of such predicates depend on the calling context in which they are activated. For example, in order to protect local resources of remote servers from malicious mobile agents, I/O predicates are not allowed in any worker or messenger definition. In addition, the usage of agent-oriented predicates also depends on the type of Imago. Table 4.1 shows a brief list of predicates legal to each Imago type. In principle, all these predicates are not re-executable. Moreover, they can be used in Imago definitions only. That is, invocation of agent-oriented predicates is not allowed in any user-defined modules.

Somewhat like the way Java objects are instantiated from classes, Imago instances are created from Imago definitions. From now on, we simply use *Imago* to be synonymous with *Imago instance*. An IMAGO application starts from its stationary Imago. It looks like that the wings of a stationary Imago have degenerated, so that it has lost its mobility. Therefore, a stationary Imago always executes on the host where it

Imago Type	Built-in Predicates
<i>stationary</i>	create, accept, wait_accept, wait, dispatch, dispose
<i>worker</i>	move, clone, accept, dispatch, wait, wait_accept, dispose
<i>messenger</i>	move, clone, attach, dispose

Table 4.1
Agent-oriented Predicates

begins execution. The major reason of introducing stationary Imago came from the security consideration. A stationary Imago has privileges to access resources of its home machine, such as I/O, files, GUI manager, *etc*, whereas other Imago are strictly forbidden to access such critical resources. A stationary Imago can create workers or dispatch messengers, but it can not clone itself. We can find the similarity that there is only one queen in a colony of bees. Thus, an IMAGO Prolog application must contain one and only one stationary Imago in its context. Accordingly, the symbolic name *queen* is reserved to unambiguously refer to the stationary Imago instance in any application. The following code defines a minimum IMAGO application.

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

Workers are created by the stationary Imago. A unique symbolic name must be assigned to a worker at the time it is created, and be immutable throughout execution. For example, if the stationary Imago issues a call

```
create("buyer.lo", alice, toy(white_rabbit)),
```

a new worker is created with an identifier *alice*. After having loaded its byte-code from the file *buyer.lo*, the system will automatically generate a call *buyer(toy(write_rabbit))* to start the execution of *alice*.

During execution, workers are able to move to any location for data and services, as if working bees fly from place to place for honey and flowers. When a worker 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 remote hosts that they are visiting, the IMAGO system enforces tight access control on workers: they have no right to access any kind of system resources except legal services provided by remote servers.

Workers are able to launch system built-in messengers to deliver messages. They can also issue *accept* or *wait_accept* predicate to receive messages. The predicate which is similar to an unblocking receive is *accept(Sender, Msg)*. An invocation to this predicate 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 predicate which implements blocking receive is *wait_accept(Sender, Msg)*. A call to this predicate 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, the predicate will be automatically re-executed when a new messenger attaches to the caller's messenger queue.

The IMAGO system does not allow a worker to *create* other workers, but a worker can *clone* itself at any location. A cloned worker is an identical copy of its original except being assigned a different symbolic name. As the matter of fact, the behavior of *clone* resembles the *fork()* in C such that two identical Imago continue their execution at the instruction immediately following the *clone* predicate but each has a different return value (*clone* or *parent* for the clone or the parent respectively). As a consequence, the original Imago and the cloned Imago may take a different path to continue conditional upon the return value.

4.2 IMAGO Messengers

The IMAGO system provides a set of built-in messengers as a part of the IMAGO Prolog API. These messengers are robust and sufficient for most IMAGO applications. They may be dispatched by either a stationary Imago or a worker anywhere and anytime. For the sake of flexibility, the IMAGO system also allows users to define their own messengers. However, a user defined messenger must be dispatched by the stationary Imago because only the stationary Imago has the privilege to access the messenger code from the local file system of the home station.

The simple and flexible form of inter-agent communication is asynchronous communication. In this case, the worker continues immediately after it has dispatched a messenger. The following example shows an *oneway_messenger* which delivers a message asynchronously. It is worth to note that name *oneway_messenger* is the *Imago definition* name rather than the *Imago instance* name, because messengers are anonymous.

```
:- messenger(oneway_messenger).
oneway_messenger([Receiver, Msg]):-
    deliver(Receiver, Msg),
    dispose.
deliver(Receiver, Msg):-
    attach(Receiver, Msg, Result),
    check(Receiver, Msg, Result).
check(_, _, received):- !.
check(_, _, deceased):- !.
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).
:- end_messenger(oneway_messenger).

```

When the *oneway_messenger* starts execution, it tries to *attach* itself to the given receiver. Its behavior is mostly the same as what we have described in section 3.3 except that it is now able to handle the case of a cloned receiver. 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 worker. 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 re-attaches to the original receiver and the cloned messenger attaches to the cloned receiver. 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 (code, stack, messenger queue, *etc.*) to create the *identical* copy. Therefore, a cloned worker seems to have a messenger queue the same as its original, but the contents in the queue are new threads representing cloned messengers.

The *oneway_messenger* is the most basic system built-in messenger. 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 byte-code and execution stack is very small. In our current implementation, the total amount of bytes to be transferred is approximately 450 bytes (byte-code size of the *oneway_messenger*) plus the size of message.

An important aspect of our messenger mechanism is the format of messages and their interpretation. Messages can, in principle, contain any data. From IMAGO Prolog point of view, a message is a logic term which could be an atom, a list, a compound structure, or even a variable. When a receiver issues a call to the built-in predicate

wait_accept(Sender, Message)

this predicate will search the caller's messenger queue for a matching messenger. Pragmatically, the semantics of *matching messenger* is implemented by Prolog unification. Let (S, M) be the sender's identifier and message carried by a messenger, and (Sender, Message) be the arguments of an accept-like predicate, a messenger is the matching messenger accepted by the receiver if the general unification of (S, M) and (Sender, Message) succeeds. More precisely, the behavior of an accepting predicate is determined by table 4.2. The predicate succeeds if one of the cases in the table is satisfied (the matched messenger will be resumed with a return value *received*), or it fails/waits otherwise. For instance, a call *accept(alice, dating(Time))* will succeed only if there exists a messenger in the caller's messenger queue whose dispatcher is 'alice' and who carries a message which unifies with *dating(Time)*. An unsuccessful accepting predicate does not cause any side effect and the messenger queue remains unchanged.

<i>Sender</i>	<i>Message</i>	<i>Matching Messenger</i>
var	var	1st messenger (in the queue)
var	nonvar	1st messenger matching <i>Message</i>
nonvar	var	1st messenger matching <i>Sender</i>
nonvar	nonvar	1st messenger matching both <i>Sender</i> and <i>Message</i>

Table 4.2
Behavior of Accepting Predicates

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 workers, we can design a *paperboy* messenger to dispatch a copy of message (like a copy of newspaper) to different subscribers, we can design a *cod_messenger* (Cash On Delivery) 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. Those messengers follow the basic pattern of the *oneway-messenger* in design with minor differences.

4.3 An Overview of MLVM

The IMAGO system involves two kinds of agent servers: *stationary server* and *remote Imago server*. A stationary server is the home of an application. Imago are strictly forbidden to enter a stationary server if they are created from different homes. Conceptually, a stationary server coexists with its running application, *i.e.*, it terminates when its hosted application completes. This kind of server should provide rich services to allow its own Imago accessing local resources. Needless to say, it is the default name server to accept worker registrations. On the other hand, a remote Imago server hosts workers and messengers of different applications. Like a web server, a remote Imago server must have either a well known name or a name searchable through the Internet name binder. Also, it must be active all the time to host incoming Imago. This kind of server should provide services for network computing or interfaces to other Internet servers, such as WWW servers, database servers, *etc.*

In fact, an IMAGO server (no matter stationary or remote) is a machine running MLVM to host Imago and provide a protected Imago execution environment. In this section, we will discuss major issues of the MLVM implementation. The goal of MLVM is to present an efficient logic virtual machine architecture in the design space of Intelligent Mobile Agent server. To achieve this, our design has to cope with issues such as explicit concurrency, code autonomy, memory management policy, communication/synchronization and computation mobility.

In traditional operating systems, concurrent processes are the basic computational units which are asynchronous and run within independent logical address spaces. Unfortunately, the concept of process blends the concepts of resource ownership and

concurrent execution together. Modern systems try to distinguish these potentially independent concepts, and thus have led to the development of a computational concept known as the *thread*. Multi-threading refers to the ability of a system to support multiple threads of execution within a single process. In this paradigm, each concurrent process behaves like a virtual machine which emulates concurrent execution of threads. Consequently, it forms the second level concurrency within the same address space. In designing a multi-threading virtual machine, it is a critical factor to determine the strategy of multi-threading implementation. Generally speaking, there are two possible schemes:

- Threads handled and scheduled by system kernel are called *kernel level* threads. Kernel level multi-threading has the advantage that blocking and scheduling of threads are treated with greater flexibility and efficiency. For example, threads can be preempted easily. A thread issuing a system call can be blocked without blocking its host process, and each thread can get a fair share of processor time. The POSIX API is an example which provides preemptive kernel level threads. However, a major drawback is that the lighter context switching overhead of threads is lost.
- A thread package implemented as a software layer in the user space is called *user level* multi-threading. The implementation of a user level thread library is straightforward and portable without kernel modification. The key issues of this paradigm are how to handle blocking system calls from a thread and how to schedule threads for execution in a process. For example, the runtime system must trap blocking system calls to prevent blocking all threads in the process, and the processor time allocated to this process must be multiplexed among existing threads. The context switching of threads requires very little overhead because it involves saving and restoring only the program counter and stack pointers.

The MLVM goes one step further by combining both kernel level and user level multi-threading into its implementation. In brief, an MLVM-based server is a multi-threading process. Each MLVM-thread is a kernel level thread (sometimes called a light weight process) responsible for certain system function, such as memory manager, communication manager, and so on. Those MLVM-threads constitute the virtual machine to support concurrent execution of user level threads —Imago. Figure 4.1 gives a brief graphic view of the MLVM architecture.

Figure 4.1. MLVM Architecture

The advantage of exploring user level multi-threading is that switching threads is at least at an order of magnitude faster than trapping to the kernel. However, a problem related with this scheme is that a user thread can not invoke a blocking system call, such as waiting for a clock interrupt, or otherwise all Imago threads will be blocked. Thus, in the absence of clock interrupts, user level multi-threading has the difficulty of distributing processor time among runnable Imago. Clearly, the MLVM scheduler can preempt CPU from a running Imago if it invokes a built-in predicate, such as *move*, *dispose*, *etc.* In this case, the scheduler can easily switch CPU to another ready Imago. However, it is possible that a computation intensive Imago occupies CPU for an arbitrarily long time. In other words, if an Imago starts running, no other Imago in the MLVM process will ever run unless the running Imago voluntarily gives up the CPU. To solve this difficulty, the MLVM implements context switch by the cooperation of kernel level threads and the nature of our memory

management scheme. In brief, besides the context switch points caught at certain built-in predicates, the MLVM preempts CPU from an Imago thread at the points of garbage collection and memory expansion as well. It turns out that this context switch strategy is efficient and effective and it distributes CPU time quite fairly among runnable Imago threads.

When a runnable Imago is scheduled, the MLVM commits the thread to a virtual machine emulator —the MLVM engine. Essentially, the engine is a MLVM-thread which sequentially evaluates the Imago's byte-code instructions. Like the conventional procedure call paradigm, the control flow of an Imago corresponds to a depth-first traversal of the activation tree. An execution stack must be used to keep track of the chronological order of procedure calls and returns. In addition, data objects generated during execution must be properly stored. Traditional Prolog as well as most programming language implementations are based on the stack/heap memory architecture: the stack holds local variables and control information, whereas the heap stores data objects which outlive procedure activation.

Memory management is a central issue in the design of MLVM. It plays a determining role not only in shaping multi-threading strategy, but also in establishing computation mobile policy. The MLVM adopts a *single memory block* architecture [29] to manage its memory. This approach makes it much easier and simpler to control autonomous agent threads and implement strong agent migration by capturing execution state at the user thread level.

Ideally, a good memory management scheme should be efficient all the time and should use little extra space for its own bookkeeping. Unfortunately, no such scheme is known for the general problem of dynamic memory allocation. We compromise and use simple strategies to handle memory expanding/contracting as well as garbage collection, hoping that would be efficient enough in most cases.

To make this point clear, let us take a look of the *single memory block* architecture. The MLVM allocates a single contiguous memory block for each Imago thread. Figure 4.2 shows the layout of an Imago block.

The MLVM does not provide the programmer or the user control over the size of an Imago block. Instead, an Imago block is allocated by the MLVM Imago creator and can be automatically expanded or garbage collected with during the progress of the computation. During execution, an Imago thread might need more space to hold data objects, or it might move to a remote host and continue execution from the moving point. Thus, an Imago block should be able to be expanded (or possibly contracted) during run time, or to be moved to another (remote) virtual machine and then be restored at a different block space. The expandability and mobility of an Imago is in fact to manipulate the Imago block. Thus, an efficient algorithm is required to relocate block addresses from old to new.

Conceptually, the MLVM divides an Imago block into two separate segments: *program segment* and *environment segment*. The program segment contains program code, a procedure table and a symbol table. The reason of separating the program segment from the environment segment is because it exhibits different expansion and management pattern. First, the program segment is not subject to garbage collection. Secondly, it is unlikely to be expanded as often as the environment segment. For example, we do not support dynamic built-in predicates in our current implementation, therefore, the code table and the procedure table of an Imago remain static during execution. Moreover, the symbol table might never be expanded unless the table has no more empty space and new symbolic names are created during execution.

Figure 4.2. Layout of an Imago Block.

The management algorithms and strategies associated with the environment segment are critical to the system performance. In fact, the memory allocation strategy is very simple and straightforward. Any memory request is granted provided that the *Free Area* of the Imago block has sufficient space. On the other hand, the MLVM will suspend the running Imago and move it to the *memory manager* not only at the point that the *Free Area* has been exhausted but also at the point that the environment segment is subject to garbage collection.

As shown in figure 4.2, the environment segment of an Imago block is controlled by two crucial pointers: the current trail top (TT) and the current stack top (ST). The trail is used to save pointers for backtracking and garbage collection, and the stack contains all kinds of data and control information required by the computation. The trail and the stack grow towards each other. The environment segment is exhausted if TT and ST *threaten to cross*. Clearly, we should monitor the growth of these pointers at runtime. In addition, our simple memory allocation strategy makes the memory consumption worse. Stack frames are allocated on the fly with no concern for memory efficiency. Thus, we also need to monitor whether the environment segment should be garbage collected. For the sake of efficiency, we only check the *threaten to cross* and *garbage collection* condition at a few points —stack allocation instructions. When a stack allocation instruction is executed, the MLVM emulator will come up with one of the following cases:

1. There is a reasonable amount of accumulated garbage being detected, or
2. TT and ST are threaten to cross, or
3. The allocation is safe and no context switch can be made.

In case 3, the engine allocates a frame of stack to the running Imago, and then advances to the next instruction. However, if case 1 or case 2 occurs, the MLVM shifts the Imago from the engine to another MLVM-thread —the *memory manager*. The memory manager repeatedly accepts an Imago, works on the Imago's block according to the requirement, and then makes the Imago *ready* again.

As we mentioned before, an Imago block needs to be expanded if its TT and ST are threatening to cross. Generally speaking, there are two common ways of dynamic memory expanding: UNIX dynamic memory allocation functions, such as *malloc*, *realloc* and *free*, and UNIX memory mapping functions, such as *mmap*, *mremap* and *munmap*. The former scheme is adopted by XSB [30] —a Prolog emulator, and the latter is used in SableVM [31] —a Java virtual machine. The latter scheme has some advantages over the former. For example, using *mremap* to replace *realloc* may save many copying operations when the memory block needs to be relocated somewhere to grow. Unfortunately, *mremap* is not supported by some platforms. Thus, we borrow ideas from XSB for Imago block expansion. In brief, a small Imago block is allocated through *malloc* to a new born thread. When the Imago runs out of its space, the block is expanded by a call to *realloc* to meet the needs of the thread.

On the other hand, an Imago block is subject to garbage collection if a reasonable amount of garbage has been detected. The garbage detection and the Chronological Garbage Collection (CGC) algorithm have been described in [29]. Briefly speaking, CGC is mainly a generational copying collector. However, it has new features different from the traditional generational collectors [32]. First, rather than divide the memory block into a fixed number of generations, CGC introduces a concept of chronological generation —a dynamic way to divide generations. The size of a generation is bound by the machine cache size, and the number of generations vary from program to program. Second, CGC controls the frequency of collector invocations by capturing the continuation point (an approximation of local minima of stack) and a factor of cache size. Therefore it collects garbage much more frequently and (in most cases) incrementally with a trivial pause time. Third, CGC improves data locality. Most generational collectors promote survivors from the youngest generation into old generations. From the *weak generational hypothesis* that “most objects die young”, some survivors may very well become *tenured garbage* in the near future. On the other hand, CGC discards the young generations, and makes the survivors temporarily tenured as a part of the generation which becomes the youngest after

collection. This strategy definitely supports the *weak generational hypothesis*. Most importantly, the concept of chronological generation as well as the CGC detection provide us evenly distributed context switch points. From our experiments, a typical Imago is subject to garbage collection approximately after every 10K to 100K procedure calls. Therefore, the MLVM is able to fairly distribute processor time among runnable Imago without capturing clock interrupts, and conduct garbage collection in parallel.

Another important issue in MLVM design is the specification and implementation of the Imago migration protocol. This is the only protocol governing both Imago migration and inter-Imago communication. To make it easier to deal with the issues involved in different levels of communication, a multi-layer architecture has been chosen. In our current specification and implementation, five layers have been identified. They are, from bottom to top, *connection*, *routing*, *security*, *ADA rendez-vous* and *marshaling*. The lowest layer (*connection* layer) is built on the top of TCP/IP. It is responsible to establish links with remote hosts. For the sake of efficiency, established connections will be cached. A cached connection will be cleared if timeout. The next layer, *i.e.*, the *routing* layer, implements the Imago routing algorithm. Intermediate routers may be used to insure delivery of Protocol Data Units which is very similar to what the IP routing does. However, the routing algorithm at this layer is of a higher abstract order. The intended goal is to provide an automatic and user independent way for mobile agents to access any resource offered over the network, even if the destination host is located inside the boundary of a private network, behind a firewall. These two basic layers, namely, *connection* and *routing*, constitute the minimum protocol stack to support a fully functional agent-migration router.

Mobile agent security is an open research area. An important consideration in designing a security model is to specify at which layer a security mechanism should be placed. In our Imago migration protocol, security policies have been attached at the *connection* and the *security* layer. Furthermore, the MLVM has another kernel level thread, the *security manager*, to deal with privacy, physical access restrictions,

application availability, content integrity, and access policy. The purpose of the security manager is to protect Imago against a malicious host as well as protect hosts against malicious Imago. The implementation approach of the security manager is still under investigation.

Figure 4.3. IMAGO Server Architecture.

5 IMAGO MIGRATION PROTOCOL

In this chapter, I will describe the internal behavior of the migration stack designed for the Imago project. However, assumptions will be made that the reader is already familiar with network protocols (such as TCP, UDP, IP) and understands binary representation of data. The material contained in this chapter, can be found in more technical details, in appendix B. Also, the reader should keep in mind that the purpose of the entire protocol is a mobile agent migration mechanism with many strengths, but also some weaknesses. The aim of this part of the overall project was not to design and implement the most secure or the most reliable protocol, but to provide Imago agent with relative reliability and security suitable in a '*normal*', yet far from perfect, environment. The implementation of this part of the project was 95% completed at the time this report was written.

5.1 Design Objectives

At first, we had to establish a clear and concise design for our new protocol. This step took a few months (part time) of preparation to let the idea germinate and reach a mature state that could be, later on, implemented quickly and eventually extended. Because the architecture and design of the protocol was not yet clear at the time it all began, we could not easily use any of the modeling tools and techniques explained in [33]. Nevertheless, the final architecture slowly emerged from the preliminary research in a well organized manner. The first step concerned the *general* direction our protocol would take. There are probably as many migration protocol implementations as MA systems. Systems like Messenger-M ϕ [34], locate the communication module on top of the Ethernet layer, while others define it on top of the TCP/IP socket layer (CBorg [35]). Some implementations, such as Concordia [4], Aglets [2]

and many others (usually Java based), use the already existing Java serialization feature to transport data from computer to computer. And finally, systems such as D'Agent [36] implement a transport protocol relying on ASCII representation of the data contained in the transported agent. All those systems implement transportation over the network in a more or less creative way, but the general idea is that there are a minimum of four design policies to choose from.

1. Binary representation over low level hardware driver.
2. Binary representation over casual network stack (TCP/IP).
3. ASCII representation on top of casual network connections.
4. Language dependent representation (Java Serialization).

Since our transport protocol was intended to migrate a Prolog-based Imago agent, running on a C-based VM, it was not possible nor desirable to use existing tools in order to achieve communication. Because the standard TCP/IP network stack provides many features (reliable transmission, data CRC checking, ...), we did not see the purpose of completely redesigning the entire communication layers. Thus, we decided that our migration protocol would be based on the standard BSD Socket interface. We then only had to decide between two possible alternatives: Binary representation, or ASCII representation. The ASCII representation offers the advantage to be *human readable*, allowing easier debugging and maintenance process. However, this 'feature' gets taken away as soon as encryption enters into consideration, since encrypted data become unreadable to the human eye. Also, our Imago agents are binary blocks of data. Having to translate them into 7-bit ASCII representation would require implementing translation tools such as *uuencode* and the homologue, *uudecode*. These translations are quite painful to the CPU usage and, honestly, do not bring much advantage anyway.

Knowing this, the choice of the data representation over the network became quite straight forward. A binary representation on top of BSD Sockets was to be the

technology used. Once this preliminary process was completed, we had to decide on a general architecture for our new protocol. A flat architecture, involving a single layer between the Socket and the LVM engine (similar to FTP, SMTP, TELNET, HTTP, ...) would have been the first choice if it was not the best. At this stage, we had a general idea of the main architecture and it was time to move forward to a much more challenging task of enabling mobile agents.

After much consideration, we decided to compile a list of *User Requirements* that we suspected would be of most interest.

Speed: Nobody likes to wait and this item came up quite naturally.

Lightweight: Although, nowadays computers are quite powerful, we believed that a good usage of the memory and a low CPU load are still an important aspect.

Flexible: Internet being a huge jungle of disparate network topography, our protocol stack would have to be flexible enough to cope with atypical network architecture.

Reliable: As much as possible, the packets we send would have to reach their destinations, or at least try their best.

Security: Computer hackers and Internet “*junkies*” are unfortunately a reality. We do not intend to make their business easier.

Asynchronous: For optimum flexibility and control !

Multi-thread: As a starting point for a challenging implementation.

Many of these requirements have been deduced from studying the different implementations already existing. Some of the requirements are in direct relation to the problems many have, such as flexibility, or being lightweight. As far as we knew at the time, this ‘wish’ list had the features likely to attract potential users. Things got a little more complicated when we tried to match the above list with our original design. Flexible and reliable communication is hard to reach with a single layer. The

best example is the ISO OSI network stack that contains seven layers, each having been assigned a very specific task. Other design issues redirected us to a more flexible five layer stack architecture presented in figure 5.1.

4	Marshaling
3	ADArendez-vous
2	Security
1	Routing
0	Connection
	TCP/UDP
	IP

Figure 5.1. Migration Protocol Stack.

The design of the individual layers came naturally as soon as their task had been clearly defined. From bottom to top, we decided that the layer 0 (*Connection*) would be in charge for establishing and maintaining a direct TCP connection between two hosts. The layer number 1 (*Routing*) would resolve names and decide the route the PDU will take to reach its destination. Above these come the *Security* layer (number 2) in charge of the encryption/decryption of PDUs as well as detecting eventual attacks. The *ADA rendez-vous* layer (number 3) makes sure that everything sent arrives at destination or reports an error. The top of the stack would be the place for the adaptation layer that will allow Imago to be transformed and prepared for transport. This layer, namely the *Marshaling* (layer 4), was to transform and trim the Imago so only the required data would be transported to the destination machine. In this layer, extra preprocessing could also be applied (for example translation between Intel based CPU and RISC architecture based CPU) but would be the subject of later work. All these layers implement a middle-ware PDU been very similar to figure 5.2, which is the template used for all PDUs.

Figure 5.2. Middle-ware PDU.

Having the stack architecture decided, we now had to agreed upon an implementation and this was not an easy task. At first, a C++, object oriented, designed was 'laied on the table', so to speak, but did not come out as flexible as expected. No less than four different approaches were unsuccessfully explored before the idea of a totally asynchronous, C based, approach was found. Modularity was the characteristic of this particular design that made it better and made it easier to maintain and adapt. Two different (yet close) studies were conducted before the final layout of this asynchronous stack architecture was fixed, it has been untouched ever since.

5.2 Multi-layered Migration Protocol Stack

The final five layer stack is organized into modules. This modular design allowed us to work individually on each separate layer of the stack. It also made it possible to try different versions of single layers, helping to validate experimental implementations. A state machine, described in figure 5.19 and figure 5.20, was used to link the different modules into a single self-contain block accessed asynchronously from a stub interface. This section enumerates each layer in the stack describing its behavior, construction and involvement in the whole protocol.

5.2.1 Notation

In this section, the reader will be presented with a concrete example of a PDU at a very low level. Data dumps of TCP (or UDP) packets are shown to illustrate the implementation or to help the reader understand the argument. The data displays a specific syntax and formatting to simplify reading. First of all, all numbers in those dumps are hexadecimal numbers (except otherwise specified), and the different fields of interest are delimited using special characters. Those characters are: the '>' and '<' sign used to enclose the content to examine; the '*' field delimiter character is used to separate the different fields of a PDU header and the '|' header delimiter character is used to separate headers from each other within one PDU. The following is an example of a connection PDU header dump:

```
>01 23 45 67*01 02*00 00*00 00 02 2c<
```

Enclosed between > and < is the entire header which contains four fields separated by the * character. Most of the time, only the interesting area of the entire Ethernet frame is displayed so the text does not get clobbered by useless information. In any case, I will try to keep those dumps short and concise, because I know that many people do not like to read hexadecimal dumps, and only a few of us are interested in '*pushing bytes*'.

5.2.2 Connection

OSI seven layer model defines the lowest layer (*physical layer*) concerns with electrical and mechanical connections to the network. Our five layer model defines, in a similar way, the lowest layer (*connection layer*) concerns with *electrical* and *mechanical* connections to the network. Obviously, the terms *electrical* and *mechanical* connections are used in a figurative way and reflect more the BSD Socket connections than real physical hardware device connections (even if sockets may be thought of as hardware devices).

Similar in behavior, to the MAC layer on a standard Ethernet stack, but at a much higher abstraction level, the connection layer requires the IP address to an immediately reachable host to be provided by the upper layer. Depending on the length and nature of the PDU to transmit, a connected socket or a connectionless socket is instantiated and associated with the destination IP address. The connection layer offers two separate QoS levels for transmitting information over the network. An unreliable, small MTU with lighter and faster service is offered for exchanging control information between two different machines. This service is build on top of the UDP network protocol and does not guarantee delivery, nor CRC checking on forwarded data, but it does guarantee fast delivery (if successful) with minimum system resources used. This service is accessed through a specific PDU type called *SICP* PDU. The acronym stands for '*Simple Imago Control Protocol*' and is only used for inter-communications between corresponding stack layers of two different machines. The other ToS provided by the connection layer is a much more reliable, wider bandwidth, but slower and uses more resources; it is mainly used to transmit greater amount of data that should not be lost nor corrupted such as an entire mobile agent. The TCP layer used as a base for this service provides the reliability, integrity checking and delivery guarantee required for sending important data. This service is accessed with the PDU type called *SITP* (acronym which stand for '*Simple Imago Transport Protocol*'). Using those basic services, upper stack layers may freely communicate information with been guaranteed levels of the service.

As a point-to-point connection get established and information is forwarded through, the socket resource created is not destroyed right away. Instead, a caching mechanism insures that system resources are not wasted uselessly and are re-used as often as possible. The cache table is implemented as a hash table with a hash function based upon the IP address of the destination. A simple modulo operation is performed to translate the integer representing the IP address into a number in the range of the allocated hash table. Hash collisions are resolved by finding the next available slot in the table on a wrap around bases. This collision resolution algorithm has the ad-

vantage of simplicity but, however, introduce some potential irregularity in the hash table. Taking the example of two distinct IP address with the same hash value, the first IP encountered will be correctly located while the second one will be placed in the next available bucket (lets say immediately after the natural location). As long as both IP connection remain valid, both will be properly accessed using the hash collision algorithm described above. However, if the first connection is removed from the table, all entries are not re-hashed and replace accordingly. Thus, the next request made for the second IP will find an empty bucket forcing the system to allocate new resources for a new connection duplicating the previous one. This behavior is undesirable and may generate some side effects but is permitted. Cached connections are not maintained forever in the system and are associated with an idle timeout value. When the connection idle timeout expires, the socket is closed, the resources used are released to the system, and the hash table bucket is freed. In such a case, the lost connection will, sooner or later, be cleaned by the system and the cache will recover from such a collision. The same argument holds when multiple connections directed to the same host have to be established. Under heavy (usually temporary) traffic directed to a single machine, the connection protocol will try to establish parallel connections to the peer. Those connections having the same IP address (thus the same hash value) will definitely collide with each other in the hash table. However, the very first of those redundant connections will be the first selected (if not currently used) and the most exploited resetting every time its idle timeout preventing it from expiration. As soon as the traffic peak has passed, the redundant connections established to satisfy the demand, will, one after the other, timeout and get flushed out of the system. In any case, the hash table will automatically recover from colliding entries.

At a low level, this connection layer defines a three word long header containing sufficient information to identify the incoming PDU. One purpose is to validate the message as originating in an appropriate version of the protocol. The first four bytes (a long integer) contained in the header, represents an arbitrary magic number (currently set to: 0x01234567) used to filter out eventual corrupted frames or connections

Figure 5.3. Connection PDU.

establish from incompatible peers. If a miss-match of this magic number is detected, the connection layer will automatically drop the connection, freeing all resources used. This should allow the remote peer to recover quickly from the failure and does not endanger the integrity of the system by allocating an invalid resource. The following two bytes (a short), in the PDU header provides the protocol version number. The first byte is the version major number, while the second byte represents the version minor number. Currently, those two numbers are set to 1.2 for SICP packets and 1.2 for SITP packets as well. Only compatible versions are understood and all other packets are discarded (the connection remaining untouched). The one byte flag field following the version number indicates the PDU type (either SICP or SITP) for this packet using the bit marked "T" in figure 5.3. The last thirty two bits (an aligned integer) is set to the total length of the PDU excluding the connection header. This value is used to decide when the PDU has been completely received and when it should be released to the upper stack layers.

The following dump represents a connection header extracted from a SITP packet.

```
>01 23 45 67*01 02*80 00*00 00 06 d0<
```

The magic number is present, and the version number matches the current protocol

version. The first bit of the flags is set to 1 which indicates a transport PDU (SITP) and the length field indicates that 1744 (0x6d0) bytes of data are following this header.

Higher layers, using the connection layer services, always get at least the QoS specified by the server, but some implementation specific tricks often provide much better service than expected. The caching algorithm implemented in this layer helps providing much better usage of the system resources and faster delivery of SITP packets. It also offers the opportunity to improve SICP delivery by re-using cached TCP connections whenever possible rather using the UDP service every time.

5.2.3 Routing

But *why is routing important in a protocol located on top of a TCP/IP stack ?* To answer this question, we need to introduce the idea behind our migration protocol. In an inter-connected computer world, namely the Internet, hosts bear a human-defined name usually easier to remember than their resolved IP address. Using those names, Imago programs are given the opportunity to move from one place to another (migrate). Unfortunately, some symbolic names given to some machines may not be resolved into IP addresses, preventing them from being known to the world. Such configuration is an important way of protecting machines from attacks originating from other places on the Internet. However, in such cases, a migrating agent would be disallowed access to resources offered by such machines, which is not so desirable. The migration protocol, discussed here, offers an easy solution to this problem.

Migration, being a basic Mobile-Agents feature, should be made as robust and flexible as possible while hiding, as much as possible, all the irrelevant, underlying details from the end user. An important aspect of migration, that very few MA systems nowadays solve, is the total transparency of the underlying network configuration. Firewall computers, dedicated gateway machines with special security precautions on them, are used to service outside networks and are commonly used in a company network. The idea is to protect a cluster of more loosely administered

machines hidden behind it from crackers. By doing so the company establish a well monitored bottleneck point in which all network traffic must flow through. The intranet being isolated becomes directly unreachable from the outside world. In this case, it is impossible for agents to migrate to any machines within the boundary of the intranet. Instead, they may have the possibility to first jump to an intermediate machine (the firewall), from which it is then possible to migrate to the desired machine within the local network. Explicit, two step migration was quite undesirable in our migration protocol. Instead, we decided to provide a transparent way of coping with this issue, by designing a routing layer able to perform a high level routing between networks in a transparent manner.

The issue we have to solve in this *middle man* configuration is to find the gateway machine in charge of the forwarding task. The only input information provided by the agent programmer would be the complete name (including all the network components) of the destination server regardless of its intranet location. From this information, we had to find a way of discovering the gateway that would know eventually know about such machines in case a direct DNS request would fail. Using the well establish naming scheme of the internet, we came to the conclusion that some specific machines are named after the task they accomplish. For example, a World Wide Web server is most of the time named using the abbreviation '*www*', while a mail server would often be named '*smtp*', '*mail*' or perhaps '*mx*'. As a common sense rule, machines visible to the Internet are often named after the protocol or the function they implement. Knowing this, we decided to name the gateway, allowing Imago to be routed through them, after the acronym of the function those machines will accomplish. Since the task is to provide an Imago GateWay into an internal network, '*igw*' would be the name those machines would have to bear in order to allow the routing layer to discover them. Starting from this assumption, configurations, such as displayed in figure 5.4, would be easily achievable, letting agents to freely migrate from the outside to the inside of the protected cluster.

In more detail, a string substitution process takes place whenever a name resolution fails. At first, the original, programmer provided, destination symbolic name is submitted for lookup to the DNS. If this request fails, the original hostname component is stripped and replaced with the Imago gateway generic name, then the lookup is performed again. Each time a DNS lookup request fails, after that, the first component of the domain name is stripped and replaced by the Imago generic gateway name, as shown in table 5.1. Eventually, the constructed name will be successfully resolved or no more components will be left in the domain string. Either way, the name resolution terminates. If an IP address has been discovered from this process, it will be used by the connection layer to forward the PDU. Gateways receiving such PDU have the opportunity to perform the routing all over again to finally reach the destination.

Figure 5.4. Imago Routing Protocol Example.

Request	Reply
<i>perseus.imago_lab.cis.uoguelph.ca</i>	<i>Host not found</i>
<i>igw.imago_lab.cis.uoguelph.ca</i>	<i>Host not found</i>
<i>igw.cis.uoguelph.ca</i>	<i>131.104.127.127</i>

Table 5.1
Resolver Successive Requests.

Since name resolution requests are expansive (time consuming), an extremely simple cache algorithm is used in the internal lookup process in order to reduce redundant DNS requests. This cache is implemented with a hash table using the host name as hash key. No specific algorithm has been implemented to deal with collisions in the hash table. This is due to the fact that name caching is only used to improve speedup in name resolution without increasing the code complexity. In this case, a later conflicting entry insertion will overwrite the previous one. In fact, test results showed a quite significant improvement, compared with an earlier version of the code.

At a network level, the routing layer uses a variable length PDU header to carry information relative to routing. This header, as shown in figure 5.5, contains the symbolic names of both ends of the transmission along with a Time-To-Live value. The two machine names, namely the destination of the PDU and the source of the PDU, are nil terminated strings in the header. Their maximum length is not restricted within the PDU header instead, it is restricted by the internal name representation of the layer. In our implementation, the limit has been set to eighty characters per name. In the header, the destination machine name (as provided by the Imago programmer) is copied first, followed by the source machine name (the name of the machine sending this PDU) in its canonical form. Neither of those two fields are altered by the routing process. A one byte TTL field, initialized to the maximum value (0xff), is appended at the end. The gateway decrements this byte by one every time the PDU is forwarded, until it reaches zero (0). At this time, the PDU is automatically discarded, thus preventing infinite routing cycle to surcharge the network.

Concretely, the following is the dump of a routing header extracted from a SITP packet exchanged between two hosts.

```
>67 72 61 64 68 70 2d 30 33 2e 69 6d 61 67 6f 5f 6c 61 62 2e
63 69 73 2e 75 6f 67 75 65 6c 70 68 2e 63 61 00*6d 69 6e
67 2e 63 69 73 2e 75 6f 67 75 65 6c 70 68 2e 63 61 00*ff<
```

The first string spells *gradhp-03.imago_lab.cis.uoguelph.ca* in ASCII representation

Figure 5.5. Routing PDU.

while the second *ming.cis.uoguelph.ca*. The TTL has not been decremented, showing that no intermediate host has been used as a gateway.

The Imago migration protocol achieves flexible routing in a simple way while discharging the Imago itself of such a burden. The programmer does not need to know about the network configuration when creating the application. Only the full name (canonical name is preferred) of the place the agent needs to migrate to is necessary to achieve routing. As an implementation advantage, the server providing gateway services for Imago, does not need to implement the entire system. Shown in figure 5.6, the Imago gateway only needs to be able to route PDUs across networks and therefore, only needs to implement the lower two layers of the complete migration stack. Consequently, the memory required by the router program is reduced to a strict minimum and no agent will be allowed to execute on this machine preventing possible network attacks.

Figure 5.6. Imago Gateway Minimum Stack.

5.2.4 Security

On the Internet, never give your password, credit card number or any other personal information. This is a basic common safety precaution. When we first started studying the user requirements for the Imago migration protocol project, the notion of security came among the first. Because of the great variety of applications that may involve MA technology, ranging from distributed computing to distributed database searching or remote monitoring, we needed to provide the users with certain security assurances. Constructing a migration protocol (and not a whole server), we focused our effort toward security in the communications. Other common issues, such as server security or agent security, were left open for other graduate students. In the Imago migration protocol, we decided to provide secure and authenticated connection between the two end points of a migration.

To secure a connection between two ends, several elements have to be present. The most common, *encryption*, is only enough to insure privacy by preventing malicious people to read the content of the message. However, we also wanted to go further into security by creating the means for receivers to authenticate the original

sender of the message. This would involve a simple *authentication* scheme that would allow servers to detect possible fake identity and to drop such falsified PDUs. Using both, encryption and authentication, we hoped to provide enough security, at the communication level, without taking a toll on performances.

Symmetric Encryption

Using the pre-installed OpenSSL libraries, we could implement symmetric encryption in our Imago project. For greater flexibility, we decided to support several encryption algorithms. Sorting them by preference order, from the most secure to the least secure (according to our personal understanding of those ciphers). *3DES*, *Blow-Fish*, *IDEA*, *DES* and *RC4* will be tried in this preference order until one is found to be implemented by both sender and receiver. To achieve symmetric encryption, we have to generate a 32 byte long random session key, different for every PDU. Using this unique session key, we perform encryption on the data section of the security PDU according to the selected cipher. This encryption is done once and for all and will only be decrypted by the destination server (none of the intermediate router will try and read the secured content).

DSA Signature

After encryption, we had to implement some mechanisms allowing receivers to authenticate a particular sender. With this, we hoped to be able to detect and discard PDUs originating from an unauthenticated source or from a machine pretending being someone else. To reach this goal, the DSA signature scheme was the most logical tool to use since it has been designed for signing and verifying data. A design decision had to be made regarding the what would be used to generate a signature... The most logical option would have been to compute a digest upon the entire non-encrypted data, and to sign this digest. We could also achieve the same goal by using the encrypted data instead of the non-encrypted ones. Any of those scenarios would have

been acceptable and easy enough to implement. However, we feared the consequences (computational) of signing a considerable amount of data. In that case, we decided not to sign the whole PDU but merely a fraction of it. The amount and location of the information we would be using to compute a SHA digest was yet to be decided when we realized an easier, yet just as effective, method. Rather than signing an arbitrary piece of data, we decided to sign the non-encrypted session key SHA digest using our DSA key. By doing so, we would be sure of the truthfulness of the entire PDU since the session key was used in encrypting the whole data. Also, the length of the message to sign would be well known by both sides and no extra information (implicit or explicit) would have to be involved (such as the amount and the location of the signed message).

RSA Encryption

So, there we were with our encrypted data, our session key used to perform the encryption and the DSA signature made from the session key. If we did not think any further and sent, over the network, everything “AS IS”, we would have wasted our time and resources for nothing. We had to find a way to encrypt all vital information (such as the session key) so no-one else (not even us) but the intended receiver would be able to decrypt it. The solution to this problem has already been shown by the SSH process. When this program establishes a secure connection, it exchanges the session key by encrypting it using the server RSA public key. So, we decided to implement the same mechanism in our migration protocol stack. Both session key and signature would have to be asymmetrically encrypted using the RSA public key of the receiver such that only the destination would be able to perform decryption and recover the original tuple. However, we came across a new and interesting challenge: *how to get the destination's public RSA key ?*

Key Exchange

According to our algorithm of RSA encryption, the sender machine is required to use the destination's public RSA key to encrypt session key and signature. This requires that it has to get, by any means, the public RSA key of the destination server. It took us quite a while to figure out a way to make both servers exchange this bit of information. Every possible solution we envisaged were not suitable for the protocol as a whole, nor totally secure. At the end, we decided to select the easiest to implement, the most suitable for the protocol and the less unsecured solution we could think of. Unfortunately, there is no such thing that is 100% secure and attack proof but at this time, we agreed on settling for less. Maybe a later study will decide on a much more secure algorithm and implement it, but now was not the proper time for that. In our Imago migration protocol, public RSA key exchange is done when needed (first time the host is encounter) using a two SICP PDU exchange. The result of the security information exchange will have to be cached and reused ever after. Then, possible attacks made after the key exchange has completed, will be detected and dealt with accordingly in an efficient manner. The exchange in itself is done using the specific layer PDU (SICP) detailed in figure 5.8. The security information for the current server are packed and sent to the destination. This destination will reply using a similar packet (marked as a reply) toward this host. In the end, both servers should know about each other and use the information gathered to encrypt and verify the PDU.

All of this information, namely RSA public key and DSA public key, must be remembered in each server so they can be used to either encrypt the PDU or verify their authenticity. We decided to create a file format as described in table 5.2 and to store its content into a file name '*known_hosts*', and this file is saved into the configuration directory of the server. An incoming or outgoing PDU requiring RSA encryption or DSA signature verification would just load the appropriate key from the database and perform the action required.

Server Name	Cipher	RSA		DSA			
	Mask	n	e	p	q	g	pub_key

Table 5.2
Known Host Database File Format.

When the security is enabled in a PDU, the very first bit (bit 0) of the security header must be set to 1. A security header having the bit 0 cleared (ie: set to 0) does not contain any information and in this case, the header is only one byte long as shown in figure 5.7. In the other case, two different types of headers may be used

Figure 5.7. Security Disabled PDU.

to differentiate key exchange PDU and data transport PDU (respectively SICP and SITP PDU). The key exchange header shown in figure 5.8 starts by a one byte flag field describing what information are provided in this PDU (ie: RSA key and/or DSA key). The value of each separate bits in the flag area is explained by the table 5.3. The cipher mask byte, immediately following the flags, indicates, in the key exchange PDU, which of the ciphers listed in table 5.4, is supported by the server. This value is remembered by the server receiving this PDU and affects which cipher is going to be used whenever the receiver needs to send a transport PDU to the sender. On the first half word boundary, a two byte short number stores a unique identifier used

for this particular request. This ID must be copied *as is* in the reply PDU so the sender can identify the original request. Aligned on the first 32 bit word boundary starts the RSA Public host key information with its different components followed by its complement DSA public key components. The flag value must indicate the presence of the RSA key and DSA key if either one is provided for the PDU to be valid. By default, both keys are always provided and the flags always reflect this setting. Data is never transmitted using SICP packet so there is no data appended to the PDU. The second type of PDU header (the most often see), shown in figure 5.9,

Figure 5.8. Security Key Exchange PDU.

is a SITP PDU that contains the encrypted data and the information required for

Key Exchange Request	1
Key Exchange Response	0
PDU is signed	1.
RSA Key provided	...1....
DSA Key provided	..1.....
Security Enabled	1.....
Security Disabled	0.....

Table 5.3
Meaning of the Security PDU Flags.

IDEA CFB	1
DES CBC	1.
3DES CBC	1..
RC4	1...
Blowfish	...1....

Table 5.4
Supported Cipher Bit Mask.

a successful decryption. The header starts with a one byte flag having the same meaning as describe in table 5.3 and a cipher value. In this PDU, the cipher field must contain only one bit set to indicate which cipher has been used to encrypt the data. The second half word contains the length in byte of the encrypted session key stored after that. The encrypted signature length is next as a 16 bit short and the encrypted signature is the last in the PDU header. The header is prepended to the entire encrypted data and the whole makes the transport PDU.

Figure 5.9. Security PDU.

To illustrate those different headers, we present the trace dumps of each of the possible cases below. The security disabled case is not shown as it is only one byte long and this byte is set to 0x00. The following dump shows a key exchange request generated by a server:

```
>b1*1f*00 00*08 00*a2 58 40 05 c9 39 6c b8 94 f9 b4 3c 1e 77 cc bb 8f
```

...

```
58 2e b2 b9 e5 c1 cf be 83*00 06*23*02 00*87 4b 58 61 8b a5 f1 7f b4
```

...

```

37 15 4b be a1 57 98 24 97*00 a0*e6 c7 b5 af 77 82 05 1a 43 f3 dc e2
c9 b6 c4 38 b2 21 22 5b*02 00*80 70 45 35 ea 0b d7 99 db 9b e3 d3 e4
...
8e 0f 18 6e 1f*01 ff*54 32 88 a9 f0 73 8f 37 e5 68 65 24 84 e2 71 b2
...
3b ca<

```

The above example has been abbreviated with only the important information. The flag value (0xb1) indicates a key exchange request (0x01) containing both RSA public key and DSA public key components (0x20 for RSA and 0x10 for DSA). The server requesting the destination security information, supports all cipher (0x1f) and the request ID number is set to 0 (first request from the security layer). The short following the request ID indicate the length (in bits) for the big number representing the public modulus of the public RSA key for this server. The different components of both, RSA and DSA keys are given in order according to the figure 5.8. The next abbreviated dump is an example of a normal transport PDU.

```

>82*04*01 00*01 4e f0 af d6 0d 97 59 58 1c 81 44 d3 16 ad f2 98 aa 60
...
6b 09 68 ee 5e 04 14*01 00*3c 94 e6 de 58 80 7c 77 73 b8 81 84 38 b4
...
1b 80 1b 38 e9 b7 79 c9 cf 53 ee 89<

```

The bit 6 of the flags field is set indicating that the PDU has been signed, while the cipher field says that the *3DES-CBC* encryption (0x40) has been used on this PDU. The encrypted session key is 256 byte long (0x100) as well as the encrypted signature. After this header follows the encrypted data (not shown here).

This simple and flexible security scheme provides sufficient safety protection for migrating Imago while keeping the cost of encryption / decryption as low as possible. Also, designing the security layer on top of the routing layer allow for faster routing across subnetworks since no security processing is involved at the intermediate gateway.

5.2.5 ADArendez-vous

The French proverb, “*Aime la vérité mais pardonne à l’erreur.*”, Voltaire (1694-1778), could have the following English translation: “*Love the truth but forgive mistakes.*” It means that the true information should be enjoyed while leaving some room to recover from errors. With this idea in mind, we decided to implement some fault recovery procedure, in our migration protocol, that would not impact the overall performances. As said above, truthful information should be enjoyed at first, then, if and only if errors happen, we will initiate a recovery procedure. This, as a whole, is the purpose of the ADA rendez-vous layer discussed here.

When we first decided to implement a connection based layer in our network stack, it was because we were not quite clear how the actual transmission was to be carried out. With a direct TCP connection between the two peers (namely sender and receiver), such connection layer would be redundant with the connection based, TCP, layer. However, we knew that we had decided to implement a routing algorithm and from this, we assume that some routing would take place breaking the TCP connection into parts. In that case, we had to find a way to know if our PDU had been successfully delivered without taking a toll on the performance. The ADA rendez-vous protocol, described in [37], allow a high level connection to be established at a very low cost using a three message exchange. This message orientation is, also, very well adapted to our internal PDU representation.

The rendez-vous protocol defines a set of four different types of messages having specific meanings. Those messages, *request*, *accept?*, *ACK* and *NACK*, are implemented as PDU types in our migration stack. According to the protocol, a minimum number of three PDUs is sufficient (and required) to establish a reliable connection. An initial *request* containing the original data, is sent to the other peer, as shown in figure 5.10. This peer, analyzing the request, decides whether or not it can accept this new message, according to some security issues, memory requirements, ... If everything goes fine, the receiver will send back an *accept* confirmation to the source,

which, in turns, acknowledge the transaction using an *ACK* packet. This typical scenario does not require any extra exchange since the original request has reached the destination “on time”. The “on time” delivery simply means that the sender system (or the receiver), has not detected a timing violation within the transaction flow. A timing violation could be caused by an expected PDU not being received or by a PDU being received too late according to some timeout settings. The ADA rendez-vous protocol specifies a recovery procedures for any unexpected behavior, mainly consisting on re-sending the last sent PDU. The retransmission is expected to reach the destination and to generate the appropriate reply (which of course, has to be received on time). Three possible cases may be encountered while dealing with errors in the rendez-vous protocol. The first case, refers to figure 5.11, happens when the initial *request* PDU is lost during transmission. According to the fault recovery rules, this packet will be regenerated after a predefined amount of time if the *accept?* message has not received. Because this is the initial exchange, a retransmission count is incremented until a fixed limit triggers an abort procedure if crossed. This procedure immediately terminates the transmission and generates an error report forwarded back to the upper layers in the stack. In some cases, the *request* may successfully be received by the destination as shown in figure 5.12. However, the generated *accept?* sequence may be lost while been transmitted. In such case, it is more desirable to regenerate the *accept?* control message before the *request* transport message is re-sent. This can be easily done by fine tuning the different timeout values responsible for the two respective retransmissions. Finally, the last possible case shown in figure 5.13, may happen while the last acknowledgement is in transit over the network. The initiator of the exchange will have discarded its copy of the original data while the receiver will not be permitted to release those data to the upper layer while the exchange has not completed (to prevent duplication). This issue is easily resolved by simply reissuing the *accept?* message sent to the originator, as long as the *ACK* has not been received.

Figure 5.10. ADArendez-vous Three PDU Exchange.

Figure 5.11. Losing The Initial PDU.

Figure 5.12. Losing The *accept?* Message. Figure 5.13. Losing The *ACK* Message.

Internally, not much data structure is required for the ADA rendez-vous layer. Merely some timer facility (offered by the timer module) and some PDU backup tools (provided by the PDU backup module) are required to reproduce the rendez-vous. Timers are allocated on a per PDU basis and are initialized according to the direction of the packet. The timeout used for the *request* retransmission (outgoing PDU only) is set to 3 minutes while the timeout used to regenerate the *accept?* packet (incoming PDU only) is set to one minute. Those timeout values have been set arbitrarily but are believed to produce reasonable results in case of failure in most environments.

At a more detailed level, the ADA rendez-vous layer uses a two word (2 time 32 bits) PDU header described in figure 5.14, for all PDU types. The two bytes generation ID is located first. This ID remains constant throughout the life of the server. It is used in the crash recovery procedure. Following the generation ID, the one byte field PDU type indicates which of the four types, this particular packet is. A detail of the different values and their meaning is given in table 5.5. The second and last word in the header, contains a 32 bit integer representing the unique sequence ID used internally to identify the original PDU this message applies to. Original data from the upper layer of the stack are appended after the header.

Figure 5.14. Complete ADArendez-vous PDU.

REQUEST	0x01
ACCEPT?	0x02
ACK	0x04
NACK	0x08

Table 5.5
The *PDU Type* (ADA Header).

Concretely speaking, on the network, the ADA PDU header could look like the following example:

```
>f0 44*02 00*00 00 47 74<
```

By analyzing the different components, we can extract the generation ID (0xf044), and the sequence ID (0x4774) for this request. The generation ID tells us that at least 18292 transactions have been instantiated prior to this one. The field *PDU type* shows that this message is an *accept?* message (0x02).

No more than two 32 bit integers are sufficient to establish a reliable communication channel between two peers regardless of the underlying network architecture. Using this double word, this protocol stack layer is able to certify the success of the migration as well as its failure. This means that Imago will never be either lost or duplicated and they will not suffer any side effect from network failure. This insures very reliable communication between the two ends of the migration.

5.2.6 Marshaling

“To arrange, place, or set in methodical order: *marshal facts in preparation for an exam.*” (<http://dictionary.reference.com/search?q=marshaling>). This Web definition for the word “marshaling” seems custom fit for the goal we had set to this layer. When we designed our migration protocol, we designed it while remembering its end purpose (which would be migrating Imago). Then, we considered what processing an Imago

would required before being transmitted in an efficient manner. This layer executes the preprocessing and post processing of the Imago memory block prior and after migration.

Preparing an Imago for network transport is not an easy task. Every important fields of the ICB must be marshaled while the unused space is trimmed to reduce the size of the transmitted Imago. Two different processes must take place in the marshaling layer of our migration protocol stack. The first process, show in figure 5.15, does the compaction of the main Imago memory block while the second marshals the most significant fields of the ICB. These fields are then put into the PDU header while the compacted Imago is stored in the data field of the header.

The unmarshaling process undoes what the marshaling process did on the source machine: restore the ICB as before and unpack the Imago block to its original size. Using the sole information encapsulated within the PDU header, the unmarshaling process has to recover all the Imago registers and variables and regenerate the Imago block as before the move. Additionally, the Imago program code, stack and trail should be scanned and re-adjusted to fit the new memory block in which the Imago has been restored to. All old pointers must be translated as shown in figure 5.16, so they point to the new memory block area rather than the old one (from before the move).

The marshaling PDU contains a header, shown in figure 5.17, and a data area respecting the structure shown in figure 5.18. The header, is divided into three general parts. the *Imago Identity* contains the information relative to the Imago as an entity. The *Imago registers* contains all the runtime registers proper to the execution of this agent are stored, and the *all-In-One Block* pointers that define the boundaries of the different components of a healthy, running Imago. The identity of an Imago is defined by its unique application ID, its type, its nil terminated symbolic name, the full designation of the stationary server identity, its citizenship and its number of jumps. The *Imago Type* field indicates the nature of the Imago (stationary, worker, messenger) and can take any of the values found in the table 5.6. (Although, due

Figure 5.15. Imago Block Compaction.

Figure 5.16. Pointer Adjustment After De-compaction.

to the restriction policy, stationary Imago will never been moved nor marshaled...) The stationary server identity is a two element component with the full (canonical) server name hosting the queen Imago and the (unique in the world) host ID for this machine (from the '*hostid*' UNIX command). Clones and messengers may be spawned or dispatched by a moving worker anywhere on the web. Remembering their place of birth is worth while, and the information may be used later on. The *citizenship* field reflect the place of birth for this particular Imago (the machine on which the agent has been created) and is only meaningful for clones and messengers. The last integer of the Imago identity contains the exact count of jumps (or moves) this particular Imago executed successfully. The *Imago registers* area contains all registers used for the execution of the interpreted Prolog byte code, from the program pointer to the stack pointer and all other registers used by the virtual machine. The *All-In-One block* pointers indicates the boundaries of the actual memory block used to store the Imago program. This includes the block base, code base, name base, name top, stack base and block ceiling.

STATIONARY	0x00
WORKER	0x01
MESSENGER	0x02

Table 5.6
Possible Imago Types.

The data area of a marshaling layer PDU contains the compacted Imago *All-In-One block*. The procedure table comes first, then the program code, name table and finally, stack and trail last. The free memory kept for the name table expansion and the free memory kept for the stack expansion are removed by the compaction process.

To illustrate all the different fields of this complex PDU, the example below provides an incomplete marshaling layer PDU with its header and data components:

Figure 5.18. Marshaled Imago.

```

>75 15 00 00*01*77 61 6e 64 65 72 65 72 30 32 00*67 72 61 64 68 70 2d
 30 33 2e 69 6d 61 67 6f 5f 6c 61 62 2e 63 69 73 2e 75 6f 67 75 65 6c
 70 68 2e 63 61 00*a8 c0 83 00*00*00 00 00 00*00 00 00 00*40 be 36 e4
*40 b6 37 28*40 b6 36 d4*40 b6 37 28*40 b6 37 60*40 b6 36 f4*40 b6 30
40*40 b6 37 5c*40 b6 30 c0*40 b6 30 08*40 b6 30 40*40 b6 35 8c*40 b6
35 ec*40 b6 36 d4*40 be 36 f0|1f 04 80 00 1c 30 b6 40 1f 04 00 01 54
30 b6 40 4f 05 00 01 58 31 b6 40 7f 05 00 02 b4 31 b6 40 4f 05 80 01
80 32 b6 40 d7 05 00 02 54 33 b6 40 0f 06 00 02 04 34 b6 40*85 00 00
00 54 30 b6 40 10 00 00 00 87 00 00 00 1f 04 80 00 02 00 00 00 01 00
...
00 00 07 04 00 00 3c 00 00 00 07 04 00 00 3c 00 00 00 07 04 00 00 06
00 00 00*5b 5d 00 6d 79 5f 77 6f 72 6b 65 72 00 24 6f 6e 65 77 61 79
...
74 69 6f 6e 00 6d 69 6e 67 2e 63 69 73 2e 75 6f 67 75 65 6c 70 68 2e
63 61*00 d9 36 b6 40 2b 00 00 00 e1 36 b6 40 5f 06 00 00 07 04 00 00
...
00 00 00 00 00 00 00 00 00 00 00 e4 36 be 40 00 00 00 00 f4 36 b6 40*00
00 00 00 2a 37 b6 40 d6 36 01e0 b6 40<

```

The end of the PDU header is marked with the header delimiter special character “|”. The *Imago Symbolic Name* field reads: “wanderer02” and the *Imago Type* field (placed before) indicates that this particular Imago is a worker Imago.

Compacting and marshaling migration agent is a very important process. It is a way to provide efficient transmission by reducing by up to 99% the amount of data transmitted over the network. This incredible percentage comes from the removed free space between the name table top and the stack base plus the free space removed between the stack top and the trail top. With such efficient compaction, the critical task of migrating Imago becomes a lot easier and much, much faster than without compaction.

5.3 Implementation

In our final stack architecture, each layer had been made in a modular way. This design has proven to be, by experimentation, extremely well suited for a multi-threaded implementation. Thus, the asynchronous feature listed in the user requirements turned out to be extremely easy to achieve with very little effort.

The primary design of the migration protocol stack included a single thread of control migrating a single Imago at a time. This approach involved the widely implemented '*client-server*' architecture described in [12], with a server listening for requests, and one or many clients generating those requests. Such architecture is very well suited in an ideal and predictable world and forces us to assign roles to the different party involved, either '*server*' or '*client*'. In a mobile-agent world, the '*client-server*' paradigm does not fit that well since the nature of the data exchanged (mobile-agents) go way beyond such implementation. Instead, each node in the network should act as a server and a client at the same time indifferently and simultaneously. Such behavior involves multiple flows of control (client flow vs. server flow) as well as multiple flows of data (outgoing PDU vs. incoming PDU), increasing dramatically the architecture complexity.

To free the main instruction flow (also called *control flow*) from time consuming blocking waits, the migration stack layers all had to be made totally asynchronous. This was achieved in each layer individually and separately. However, from the moment an Imago is scheduled for migration until it actually is written onto the physical network media, it has to go through a series of states. Each state represents a specific layer in the protocol stack. Our first implementation (before the final asynchronous version), handled the whole process as one single step that had to be carried out integrally, from top to bottom. Therefore, it was not possible to achieve an asynchronous behavior, and this design had to be discarded in favor of a different one (later on discarded as well for similar reasons).

By the end of our research process, we had all our components laid out (namely the design of each single layer), but we were still missing the glue to link them together in a simple and flexible, whole. The idea of using a state machine associated with a MIMD stream of threads came from a work experience I had a few years ago. At this time, I was a member of a sustaining team that had for a main task to fix bugs and extend features on an entirely owned terminal server device. At many places in the embedded RTOS developed for this product, state machines were used to control various components. The SNMP embedded network stack, I had to work with, was using such a state machine oriented design in a quite effective and very flexible way.

The first step in the implementation of a state machine is to define the possible states, how those states interact with each other, and how the entire system interacts with each state individually and as a whole. Our modular implementation did answer many of these questions and most of the others were answered quite easily (see figure 5.19). This state diagram also showed us very clearly, the missing piece of the puzzle: having a pure asynchronous implementation. Then the structural behavior design made from the state diagram was a clear answer. States transition would be the perfect location to release and yield the system thread control to other tasks. In that case, the threads could run in a loop and would first poll for a PDU to process, then process the retrieved PDU through the state machine (regardless of its current state), store the PDU back and continue to some other, non migration protocol related, task. All this could be done in a single loop without any blocking steps. The later implementation of the design shown in figure 5.20, came up to our expectation in a single thread environment as well as multi-threaded. The difference between single thread and multiple threads was from a performance perspective (especially on multiprocessor machines).

When we implemented the state machine in our Imago project, we decided to create a very generic state machine code for re-usability purposes. The data structures used in this module are very simple. A dynamically constructed array of states entries is initialized using a registration function called at the initialization of the state. The

Figure 5.19. Migration Stack Simple State Diagram.

Figure 5.20. State Machine Structural Behavior.

index of the entry in the array is returned as an identifier used to trigger the transition between two states. A single linked list is used to store the PDU not being currently processed (while in transitional stage).

Using this highly reentrant stack implementation, we were able to incorporate the migration protocol with the multi threaded logic virtual machine in a very efficient way. All communication is done asynchronously, and threads of control are always available for different tasks.

6 PERFORMANCE ANALYSIS

The design for the high efficiency thread pool scheduling algorithm was yet to be proved effective when the migration protocol project started. Our previous experiences taught us the great advantages of such implementation in flexibility and speed but never actually showed us. Also, a great concern, while speaking of performance, was the SSL encryption mechanism. Most of the time, the servers are using the 3DES encryption algorithm to securely forward the packets to destinations. Although this encryption is quite secure, it is also extremely time consuming. Additionally, the RSA asymmetric encryption and the DSA signature are also reducing the overall transmission speed. The goal of this performance testing analysis was to define exactly how much time it takes to migrate an Imago according to the different stages of its life cycle (the Imago life time is define by the size of its running stack, trail and the size of its name table).

The analysis performed on the migration protocol stack is based upon a local network architecture representing to the best condition possible. The worst conditions are not considered here as they would not be easily bounded. The 100 Base-T internal network of our IMAGO lab, interconnects the servers *orion* and *perseus* used to perform the testing and retrieve the results. Those two servers are equipped with a set of two *XeonTM* processors each which helped us validate our high performance multi-threaded architecture. The Intel®*XeonTM* Processor is designed for dual-processor server and workstation platforms. Featuring innovative technologies such as the Hyper-Threading Technology, Intel®*XeonTM* processor-based systems double the number of logical CPUs within a single chip. Concretely, this means that our dual *XeonTM* processors are seen as quad processors by the Operating System. This gives us a unique chance to experiment with our multi-thread architecture design on a multi-processor platform. In addition, a firewall protects and filter the network

traffic coming from the Internet. This firewall is also running the Imago router program that allows the Imago agents to travel freely from the Internet into the internal network (using the firewall as a router). Figure 6.1 summarize the configuration in which the performance testing have been done.

Figure 6.1. Imago Lab Intranet.

The primary characteristic of a mobile agent platform is to provide the means for the agent to migrate from one computer to another computer. This process is integrally handled by our migration protocol stack explained in detail in the chapter 5. The different components of the stack make it robust, flexible and quite efficient as well. But do they make it fast? Even if some known improvements would increase the overall transmission performances, in its actual state, does our Imago migration protocol stack satisfy the speed requirement? To answer this question, we created a set of test programs capable of timing the migration of an Imago.

The intended purpose of the performance test bed was to show the interactions of the different components of the entire system. In order to determine the cost of migrating an agent, we have to consider several factors. Such factors include the network configuration, the Imago life cycle and the status of the different machines

involved in the migration process. To simplify the equation, we assume that protocol PDU are directly delivered to the destination without involving any intermediate routing.

As seen earlier, the network configuration in the IMAGO LAB is *shielded* from outside perturbations. Also, the distances involved between computers are not greater than a few meter. In such conditions, PDU have only an extremely thin chance to get lost or corrupted, and therefore, low communication layer retransmission on timeout or error recovery procedure will most likely never occur.

As Imagoes evolve in their environment, their stacks and trails grow or shrink according to the algorithm used in the program. Certainly, the transmission time on Ethernet is dictated by the transmission speed of the network interface as well as the total amount of data to be transmitted. Logically, the more information you send, the longer it will take to completely communicate it to the other side. According to this, the test programs are tailored to insure that Imago size is not constantly growing but remains stable as the garbage collector perform its task.

As we compute the transmission time between different machines, it is very important to consider the hardware differences between those machines. The most important hardware difference is in the internal clock for each computer. We do not intend to develop a distributed clock synchronization algorithm (even if our mobile agent system would be able to do such a thing easily), but only a testing program that would allow us to estimate the performance of our system regardless of such hardware differences.

To make our testing complete, three separate programs (merged into a single one for simplicity) are used. Each illustrates a specific characteristic of our migration protocol but also reflects some of the internal behavior of our Prolog virtual machine. The first test program involves a simple round trip agent while the second illustrates a one way timing estimation with a very simple distributed clock synchronization mechanism. The third program is designed in such a way that the Imago stack size keeps on growing. In this case, the amount of data involved in the migration is much

more than the previous two programs. The complete program can be found in the appendix C.

Since the IMAGO Server may be configured to either use or not use encryption and a signature scheme at a transport level, we decided to compare both cases (encrypted transmission vs. clear transmission). Results are provided for both cases to show the impact the security feature on the overall system.

All results gathered by agents are forwarded back to the queen via the dispatching of a *oneway_messenger*. The queen handles the printing of the result in a form that can be processed using the GNU plot tool.

6.1 Simple Round Trip

This first test is intended to give a rough idea about the system characteristics in general. We did not intend to highlight any specific feature (like garbage collection, memory expansion, *etc.*...) but rather get a first idea of the system behavior. To do so, we decided to write an Imago agent that would compute the time it will take to move to a specific machine and back again. The round trip procedure is shown below:

```
roundtrip(Home,Server,Time2):-
    move(Home),
    ctime(Ta1),
    move(Server),
    move(Home),
    ctime(Ta3),
    Time2 is (Ta3-Ta1).
```

The first statement in the procedure forces the Imago to move back to the starting position. By using the builtin '*ctime/1*' the Imago gather the current time on the machine it resides on, we named this time *Ta1*. Just after this, the agent starts executing a round trip to the server and back home using the '*move/1*' IMAGO-

Prolog specific builtin predicate. The arrival time is then, retrieved in the variable named *Ta3*, and the total round trip cost is computed and stored into the return value of the '*roundtrip/3*' procedure. A graphical representation of the process is showing in figure 6.2.

Figure 6.2. Simple Round Trip Statistic Program.

Running this agent on a 12000 round trips (24000 one-ways) with the encryption layer enabled generates the result shown in figure 6.3. From this graph, we can clearly see a saw-tooth pattern that can be explained by the garbage collection algorithm. As the Imago grows older, its stack and trail increase in size slowing down the migration. Since all intermediate results (previous round trip timings) are *tenured garbage*, they all get collected every now and then. The drastic stack size decrease, speeds up the migration process back to the original value. Every vertical line in the graph indicates a GC point. It is worth noting that as the Imago size linearly increases, its transmission time increases linearly as well.

To show the impact of encryption, the same 12000 round trips are executed but with the security feature been disabled on both servers. Due to the lack of encryption, the overall transmission is speeded up by a factor of 30%. Figure 6.4 shows the same pattern as seen earlier (with the encryption enabled) but at a smaller scale.

The simple round trip results remain consistent with the internal behavior of the virtual machine. Garbage collection points are shown clearly and are obviously

Figure 6.3. 12000 Round Trip With Encryption Enabled.

Figure 6.4. 12000 Round Trip With Encryption Disabled.

effective. The *parasites*, on both graphs, are to be blamed on the machine load, the system thread scheduling, and also the garbage collection process that may be executed on the remote machine (thus included in the round trip time). Other possible factors involved on the degradation of some migrations, may be due to the network traffic and the result messengers carrying the round trip value back to the queen.

6.2 One Way Trip

The second example makes good use of more background knowledge in distributed systems. After seeing the performance on a round trip basis, we decided to make an attempt to time a single one way trip between the two machines. The tentative was more perilous since the two computer clocks could drastically change our results and make a single one way trip going at the speed of light or at the speed of a turtle.

The problem of distributed clock is quite complex so rather than trying to solve it, we decided to overcome it using a simple clock drift estimation algorithm. To illustrate this idea, let us modify figure 6.2 and complete it until we get figure 6.5. From this new perspective, two different clock bases need to be defined and named Ta and Tb for the *Host A* and *Host B* respectively. At first, the Imago will execute a complete round trip to gather the time $Ta1$, $Tb2$ and $Ta3$. At this time, we make the following assumption that a round trip time is exactly twice as much as a single way. Such assumption is not valid in the sense that the overall system is not deterministic and the way back may take less or more time than the way forward. However, for the sake of simplicity, we do assume it is true in this case. Using the starting and ending time, we compute the theoretical value of $Ta2$ as follow: $Ta2 = (Ta1 + Ta3)/2$. This approximation represent the time $Tb2$ using the *Host A* clock as a base (rather than *Host B*'s clock). Then, we compute the clock drift between the two $Tb2$ and $Ta2$ such as: $Drift = Ta2 - Tb2$. This number is then added to all timing retrieved on the *Host B* to convert the clocks into the same base. This clock synchronization process is executed by the agent at the beginning by the procedure below.

`clocksync(Home,Server,Drift):-`

```

    move(Home),
    ctime(Ta1),
    move(Server),
    ctime(Tb2),
    move(Home),
    ctime(Ta3),
    Ta2 is (Ta3+Ta1)/2,
    Drift is (Ta2-Tb2).

```

Once the clock drift has been computed, the same value is used throughout the testing.

Figure 6.5. One Way Timing With Clock Drift Estimation.

Using this clock synchronization algorithm, we created an Imago that could retrieve the time of 12000 one way trip between *perseus* and *orion*. Two different run have been tried in order to show the effect of the encryption routine. Figure 6.6 represents the test result while the encryption was enabled. You can see once more, the exact same pattern as for a complete round trip; however, the times involved shrank by a factor two.

Figure 6.6. 12000 One Ways Trip With Encryption Enabled.

Similarly, we also executed the same testing with the encryption disabled, figure 6.7, on both servers, and once again, the results show a 30% improvement over the encrypted version.

However, both performance graphs exhibit an interesting behavior not seen before. It seems like the averaging value does not seem to be representing a line but rather a curve interrupted from times to times. The explanation can be found in the hardware clock differences between the two computers. As the Imago travels, it uses an estimation of the clock drift made at the very beginning of the entire process. As time goes, this estimation becomes less and less accurate since both computers' hardware clocks are out of synchronization. However, it is curious to see that at one point, the clocks of both computers suddenly synchronize again. To understand the issue, we would look closer into the configuration of each server and we should find that the NTP daemon server is running on both machines. This daemon has for objective to synchronize the local time with an external source (which would be *draco* in our case) causing the graph to suddenly recover its original level.

Figure 6.7. 12000 One Ways Trip With Encryption Disabled.

6.3 Round Trip With Growing Imago

Using the same idea as the very first test program, this new test takes into consideration the dynamically growing Imago stack. This program keeps a record of all the time values for the intermediate round trip in an array of floats (one float occupies three 32-bit words in the LVM implementation). In order to show the benefit of the garbage collection process, the 12000 round trips have been broken down into 4 sets of 3000 round trips. This means that the Imago will discard all previously computed round trip timings once it reaches the 3000 limit. Then, it restarts building up its array of values again and again. Consequently, the stack keeps on growing and growing until the array is discarded and garbage collected.

The resulting graph from figure 6.8, shows the Imago evolving with the encryption enabled. However, the pattern is slightly different from the first two results because our Imago program keeps on growing and growing. At one point, when the 3000 round trips limit is reached, the array of floating value is discarded and almost immediately

garbage collected by the memory manager. On the graph, it is seen as a brutal improvement of the transmission speed.

Figure 6.8. 12000 Round Trip With Size Increase (Encryption Enabled).

Similarly to the other tests, we also ran the same program disabling the security feature, as you can see on figure 6.9. Without any surprises now, we discovered the exact same graph as figure 6.8 but glided down by a 30% improvement factor.

As Imago travels the network, the time they will spend in transit is proportional to their runtime size. Fortunately enough, it seems that the relation *size*→*speed* is a linear function of the form ' $speed = a * size + b$ ' where a and b are constant factors. Unfortunately, at this early stage of implementation, the virtual machine does not provide the tools to retrieve the actual size of the Imago (stack, code and trail). Thus, it was not possible for us to study the performances of the migration process, according to the exact size of the Imago. However, we know by experiments that messengers travel much faster than their homologue workers. In this case, message delivery is usually done extremely quickly (according to the size of the message of course).

Figure 6.9. 12000 Round Trip With Size Increase (Encryption Disabled).

We know now that the encryption process introduces 30% slow down in the worst case (3DES encryption being the slowest). However, the different cryptography algorithms provided do have different effect on the performances.

As later work, we consider implementing a RSA public key caching algorithm as well as a more efficient PDU cloning mechanism (ADA rendez-vous layer). Also, some configuration options should be implemented to let the administrator select which encryption type is preferred among the ones provided.

7 CONCLUSION

In this thesis we investigated existing MA systems based on literal survey. Then, we introduced new design concepts in the project called “IMAGO” and concentrated on the design and implementation of the Agent Migration Protocol. The idea was to deploy intelligent mobile messengers for inter-agent communication. Imago Messengers are thin agents with the same capability as regular Imago Workers, but designed for one single purpose which is to establish reliable inter-agent communication. Those messengers are anonymous and carry the message to deliver. Using a simple, yet powerful, tracking system, they are able to establish a communication channel between two Imago regardless of their location on the network.

One of the strengths of the IMAGO system is that it defines an Agent Migration Protocol capable of coping with both agent migration and inter-agent communication. The agent migration is achieved using a five layer protocol stack which establishes an end-to-end encrypted channel. Using this basic feature of a MA system, Imago can transparently and reliably migrate to anywhere on the network in order to carry out their assigned task.

The implementation of the IMAGO Project brings the overall system to a very competitive level with many current MA platforms. With the set of features provided, many applications, such as “Smart Information Retrieval”, “Dynamic Monitoring” or “Reduced Bandwidth Wireless Browsing” can be made possible and easily implemented.

Since our system is fairly new, many issues related to Mobile-Agent technology, are yet to be solved. **At the current stage, most practical issues, such as multithreading, garbage collection, agent migration protocols, virtual machine engine and memory management, have been specified and their implementations are under testing, whereas some other issues, such as security,**

services, *etc*, will be investigated further. A signature based agent protection feature should provide strong authentication as well as detection of tampering for roaming Imago. As “e-shopping” grows and expands, Internet based applications are required to have some kind of user level authentication mechanism. In a later publication, we will discuss a “security record based” scheme in order to provide such security features.

Most of the MA systems implemented (commercial or experimental), use a quota based scheme for server resource protection. Mobile-Agents are assigned a fix amount of ressources on the hosting system. If the agent requests more ressources than allocated, it automatically get discarded in order to prevent any over use. The basic principle defined in our IMAGO project is that only the stationary agent has full hardware resource control. This is, however, not enough to prevent malicious programs from hurting the hosting system integrity by using an excessive amount of resources (CPU or memory). To solve this issue, we will propose in a later study, a “statistically based” server protection algorithm. This solution should provide security, based upon the study of the dynamic behavior of a particular agent. We will try to prove that the malicious agent behavior statistically differs from a “well behaved” agent, and therefore, by studying frequency rates and occurrences of particular actions, we can isolate and properly handle potentially harmful agents.

The BuzzBee family [38].

LIST OF REFERENCES

LIST OF REFERENCES

- [1] H. Peine. *Mobile Agents: Explanations and Examples* (eds. W. Cockayne and M. Zyda), chapter Ara - Agents for Remote Action. Manning/Prentice Hall, 1997.
- [2] D. B. Lange and M. Oshima. *Programming and Deploying Java Mobile Agents with Aglets*. Addison-Wesley, August 1998.
- [3] N. Karnik and A. Tripathi. Agent server architecture for the ajanta mobile-agent system. In *PDPTA'98*, pages 62–73. CSREA Press, 1998.
- [4] Mitsubishi Electric. Concordia. <http://www.meitca.com/HSL/Projects/Concordia>.
- [5] David Kotz, Robert Gray, Saurab Nog, Daniela Rus, Sumit Chawla, and George Cybenko. Agent Tcl: Targeting the needs of mobile computers. *IEEE Internet Computing*, 1(4):58–67, July/August 1997.
- [6] J. Baumann et al. *Mole - Concepts of Mobile Agent System*, volume 3 of 1, pages 123–137. World Wide Web, 1998.
- [7] General Magic Inc. Odyssey. <http://www.genmagic.com/agents>.
- [8] J. E. White. Mobile agents. Technical report, General Magic Inc., 1995.
- [9] D. Johansen, R. van Renesse, and F. B. Schneider. Operating system for mobile agents. In *HotOS-V'95*, pages 42–45. IEEE Computer Society Press, 1995.
- [10] ObjectSpace Inc. Objectspace: Objectspace voyager core package technical overview, 1997. <http://www.objectspace.com/>.
- [11] E. Pitoura and G. Samaras. Locating objects in mobile computing. *IEEE Trans. on Knowledge and Data Engineering*, 13(4):571–592, 2001.
- [12] Andrew S. Tanenbaum and Maarten van Steen. *Distributed Systems, Principles and Paradigms*. Prentice-Hall, Upper Saddle River, New Jersey, first edition, 2002.
- [13] A. Murphy and G. Picco. Reliable communication for highly mobile agents. In *ASA/MA'99*, pages 141–150, 1999.
- [14] J. Cao et al. Mailbox-based scheme for mobile agent communications. *IEEE Computer*, 35(9):54–60, 2002.
- [15] *IMAGO: A Prolog-based System for Intelligent Mobile Agents*, LNCS 2164. Springer-Verlag, 2001.
- [16] Theo D'Hondt. <http://pico.vub.ac.de>.
- [17] C.A.R Hoare. *Communicating Sequential Processes*. Prentice-Hall International Series in Computer Science, 1985.

- [18] Rogin Milner. *The polyadic pi-calculus: A tutorial*. LFCS report ECS-LFCS-91-180.
- [19] G. Agha. *Actors: A model of concurrent computation in distributed systems*. Technical Report 844, Massachusetts Institute of Technology, 1985.
- [20] Robert Gray. *Agent Tcl: A flexible and secure mobile-agent system*. PhD thesis, Dept. of Computer Science, Dartmouth College, June 1997. Available as Dartmouth Computer Science Technical Report TR98-327.
- [21] Robert S. Gray. Agent Tcl: A flexible and secure mobile-agent system. In *Proceedings of the 1996 Tcl/Tk Workshop*, pages 9–23. USENIX Association, July 1996.
- [22] Saurab Nog, Sumit Chawla, and David Kotz. An RPC mechanism for transportable agents. Technical Report PCS-TR96-280, Dept. of Computer Science, Dartmouth College, March 1996.
- [23] A. Birrell and B. Nelson. Implementing remote procedure calls. *ACM Trans. Computer Systems*, pages 39–59, Feb 1984.
- [24] Daniela Rus, Robert Gray, and David Kotz. Transportable information agents. In *Proceedings of the First International Conference on Autonomous Agents*, pages 228–236. ACM Press, February 1997.
- [25] C. F. Tschudin. *On the Structuring of Computer Communications*. PhD thesis, University of Geneva, 1993.
- [26] M. Betz. Omg's corba. *Dr. Dobb's Special Report*, 225:8–12, Winter 1994/1995.
- [27] C. F. Tschudin. An introduction to the m ϕ messengers language. Technical Report 86 (Cahier du CUI), University of Geneva, 1994.
- [28] R. Lino Valverde. *Msgr-s: Un environnement d'execution de messeagers base sur un interpreteur scheme paralle'le*. Diploma thesis, University of Geneva, 1994.
- [29] Xining 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)*, pages 245–256, 2000.
- [30] The xsb source code, the xsb research group. <http://xsb.sourceforge.net>.
- [31] E. Gagnon and L. Hendren. Sablevm: A research framework for the efficient execution of java bytecode. Technical Report 2000-3, Sable Research Group, McGill University, 2000.
- [32] R. Jones and R. Lins. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*. John Wiley and Sons, 1996.
- [33] James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy, and Williamk Lorensen. *Object-Oriented Modeling and Design*. Prentice Hall, Englewood Cliffs, New Jersey, 1991.
- [34] University of Geneva. Messenger-m ϕ . <http://cui.unige.ch/tios/msgr/>.

- [35] *Experiences in Mobile Computing: The CBorg Mobile Multi-Agent System*, 1530-2067. IEEE Computer Society Press, 2001.
- [36] Robert S. Gray. Transportable agents. Technical Report PCS-TR95-261, Dept. of Computer Science, Dartmouth College, 1995. Thesis proposal.
- [37] Department of Defense. Military standard ada programming language. Technical Report MIL-STD-1815, Naval Publications and Forms Center, Philadelphia, Pennsylvania, December 1980.
- [38] Mr. buzz bee tut. <http://www.psptoybox.com/tuts/psp7/bee.htm>.

Appendix

A THE PROJECT

The IMAGO (Intelligent Mobile Agent Gliding On-line) system is an infrastructure for Mobile Agent applications. It is based on the Prolog programming language.

The Imago distribution comes with the server daemon 'mlvmd', the Imago compiler 'imagoc' along with the assembler 'masm'. And a small unfriendly client software 'mlvmc' used to establish a low communication with the running server daemon. If you get scared by this unfriendly approach, you can also (recommended) download the Imago IDE software. This is a Java application providing point n' click feature used to edit, compile and run Imago programs. You can download the package at the same location than the main Imago distribution.

The official home of Imago is

<http://draco.cis.uoguelph.ca>

The latest distribution can be found in the subdirectory

<http://draco.cis.uoguelph.ca/download.html>

or

<http://draco.cis.uoguelph.ca/~gautran/download.shtml>

A.1 What files to download ?

You can download the latest, compressed tar archive for the entire source code at the URL provided above. Several compression types are provided for convenience. File with the extension '.gz' uses the GZip compression while files with the extension '.bz2' use the (better) BZip2 compression. The entire project is self contained and, except the IDE packed separately, you should not need to download multiple files.

A.2 What to do with those files ?

After you download the file, you will need to unpack it into a directory of your choice. Two compression format are provided. GZip compression and BZip2 compression. To uncompress the '.gz' file, run the command 'gunzip filename.gz' where 'filename' is the file name you want to uncompress. To uncompress the BZip2 file, run the similar bunzip2 command 'bunzip2 filename.bz2' where 'filename' is the file name you want to uncompress. After this, you will get a TAR archive you can unpack using the 'tar' command on all Linux system. The syntax is 'tar xf filename.tar' where the 'filename' is the file name you want to unpack. This will create the directory tree structure on your machine then you can build the entire project by following the steps in the "Installation" section. Below is an example of the command used to uncompress the bz2 file, then untar it into the final directory.

```
[gautran@gradhp-03 dir] bunzip2 mlvm-0.1B1.tar.bz2
[gautran@gradhp-03 dir]$ ls
mlvm-0.1B1.tar
[gautran@gradhp-03 dir]$ tar xf mlvm-0.1B1.tar
[gautran@gradhp-03 dir]$ ls
mlvm-0.1B1/
[gautran@gradhp-03 dir]$
```

A.3 Installation

A.3.1 Prerequisites

You will need a working installations of OpenSSL if you decide to use encryption and signature (default). OpenSSL 0.9.6 or greater: <http://www.openssl.org/>

A.3.2 Building / Installing

To install MLVM with default options:

```
./configure
make
(su root)
make install
```

This will install the MLVM binary in `/usr/local/bin`, configuration files in `/usr/local/etc/mlvm`, the server in `/usr/local/sbin`, etc. To specify a different installation prefix, use the `--prefix` option to configure:

```
./configure --prefix=/opt
make
(su root if required)
make install
```

Will install MLVM in `/opt/{bin,etc,lib,sbin}`. You can also override specific paths, for example:

```
./configure --prefix=/opt --sysconfdir=/etc
make
(su root)
make install
```

This will install the binaries in `/opt/{bin,lib,sbin}`, but will place the configuration files in `/etc/mlvm`.

If you need to pass special options to the compiler or linker, you can specify these as environment variables before running `./configure`. For example:

```
CFLAGS="-O -m486" LDFLAGS="-s" LIBS="-lrubbish" LD="/usr/foo/ld" ./configure
```


A.4 Configuration

The runtime configuration files are installed by in `${prefix}/etc/mlvm` or whatever you specified as your `--sysconffdir` (`/usr/local/etc/mlvm` by default).

The default configuration should be good in most cases, although you should review it to ensure that it matches your security requirements. The daemon configuration file is name `mlvmd.conf` and is located in the configuration file directory. This is the main configuration file for the MLVM Daemon.

The first time the server starts, it will automatically generate a set of RSA key and DSA key and store them into a file in the configuration directory. These keys are your private and public RSA and DSA key and should not be seen by anybody. It is best to keep them on the local filesystem. The data within the file is encrypted, but since the source code of the Imago project is publicly available, the encryption key and algorithm can easily be found among the source. For more information on configuration, please refer to the manual pages for `mlvmd` and `mlvmc`.

A.5 Problems?

If you experience problems compiling, installing or running MLVM. Please refer to the "reporting bugs" section of the webpage at <http://draco.cis.uoguelph.ca/>

A.6 Usage

In order to execute Imago programs on your machine, you must first install and start the server daemon. The daemon is usually started from the `/etc/rc` directory and a initialization file can be used to register the daemon with the `chkconfig`. In this case, the `initd` daemon will start `mlvmd` automatically at boot time. To compile Imago programs (outside of the IDE), use the `imagoc` compiler with your prolog source file. For more information on how to use the compiler, have a look at the manual pages

for 'imagoc'. You can generate byte code or assembler instructions that can be, in turn, assembled using the masm program to generate the byte code.

Once you have your program ready and compiled, try running it ! :) To do so, you must have the mlvmd server started on your machine, then run the mlvmc program. Once you are connected with the server you can load your new Imago by using the 'LOAD' command and providing the full path to your byte code. Below is an example of a full session involving compilation and loading of the Imago prolog.

```
[gautran@gradhp-03 qsort]$ pwd
/home/gautran/lvm/benchmarks/single/qsort
[gautran@gradhp-03 qsort]$ ls
CVS/  test1.imp  test2.imp  test3.imp
[gautran@gradhp-03 qsort]$ imagoc test1.imp
start compiling, wait...
```

IMAGOC Version 1.0

(c) IMAGO LAB -- University of Guelph

MLVM Assembler 1.3

Bytecode program generated

MLVM Assembler 1.3

Bytecode program generated

```
[gautran@gradhp-03 qsort]$ mlvmc
```

Reading configuration file: '/usr/local/etc/mlvm/mlvmd.conf'

Server contacted on local TCP port [8888]

```
550:      !!! Welcome to the MLVM Server !!!
```

```
550: MLVM version 0.1B1, Copyright (C) 2003 IMAGO LAB.
```

```
550: MLVM comes with ABSOLUTELY NO WARRANTY;
```

```
550: This is free software, and you are welcome to redistribute it
```

```
550: under certain conditions; type 'PLIC' for License details.
```

```
LOAD /home/gautran/lvm/benchmarks/single/qsart/test1.lo
```

```
550: Imago Queen Sucessfully loaded.
```

```
From=worker
```

```
Result=[1,1,1,1,2,3,3,3,5,5,9,10,10,11,11,12,14,15,15,16,16,17,18,18,
19,19,20,21,22,23,25,26,26,27,28,28,30,30,30,31,31,31,31,33,34,34,35,
37,37,38,38,39,40,41,41,41,43,46,47,47,48,48,49,50,51,52,53,53,54,54,
...
```

```
942,943,944,944,945,945,947,948,949,950,952,952,954,957,959,959,961,
963,963,967,969,970,970,971,972,973,974,974,975,976,980,981,982,982,
984,984,985,987,988,991,993,994,995,996,996,997,999,1000]
```

```
PRNT
```

```
660: ### LOG records begin:
```

```
Imago: queen, Jumps: 0, Status: Disposed
```

```
Imago: worker, Jumps: 0, Status: Disposed
```

```
660: ### LOG records end.
```

```
QUIT
```

```
550: Have a nice day !
```

```
[gautran@gradhp-03 qsart]$
```

Of course, you can always use the IDE to do all this by simply point'n click, but the process remains the same. Information regarding the usage and configuration options for every part of the project, can be found in the manual pages. However, feel free to contact us if you need assistance or help.

A.7 Reporting Bugs

If you find one of those hidden features called "bugs", please, let us know about it by sending an email to jimago-bugs@draco.cis.uoguelph.ca, including the detail explanation of the what you where doing when it happened. Attaching your Imago program is always a great idea and allow us to reproduce the issue rapidly...

concatenated into a two byte short, the PDU flags and finally the integer representing the total length of the PDU. Knowing all of this, I will show you the different component for each header of the migration stack.

B.1 Connection Layer

The twelve byte long connection PDU header provide a solid base for the point-to-point communication. The following example is an SICP PDU used to communicate between two ADA rendez-vous layer of two machine.

Frame 10 (145 bytes on wire, 145 bytes captured)

Ethernet II, Src: 00:00:00:00:00:00, Dst: 00:30:6e:26:69:04

Internet Protocol,

Src Addr: 131.104.48.208 (131.104.48.208),

Dst Addr: gradhp-03.imago_lab.cis.uoguelph.ca (192.168.0.131)

Transmission Control Protocol, Src Port: 8888 (8888), Dst Port: 56681

(56681), Seq: 1593469552, Ack: 1141322072, Len: 79

Data (79 bytes)

```

0000  00 30 6e 26 69 04 00 00 00 00 00 00 08 00 45 00  .0n&i.....E.
0010  00 83 71 e6 40 00 3e 06 55 2b 83 68 30 d0 c0 a8  ..q.@.>.U+.h0...
0020  00 83 22 b8 dd 69 5e fa 6a 70 44 07 31 58 80 18  .."...i^.jpD.1X..
0030  2d 40 2c 08 00 00 01 01 08 0a 38 56 1b b7 0e a9  -@,.....8V....
0040  7e fe>01 23 45 67*01 02*00 00*00 00 00 43<67 72  ~...#Eg.....Cgr
0050  61 64 68 70 2d 30 33 2e 69 6d 61 67 6f 5f 6c 61  adhp-03.imago_la
0060  62 2e 63 69 73 2e 75 6f 67 75 65 6c 70 68 2e 63  b.cis.uoguelph.c
0070  61 00 6d 69 6e 67 2e 63 69 73 2e 75 6f 67 75 65  a.ming.cis.uogue
0080  6c 70 68 2e 63 61 00 ff 00 40 cc 02 00 00 00 00  lph.ca...@.....
0090  00

```

The bit 16 in the second word has been cleared indicating a SICP packet with a short length of 67 bytes (0x43). The magic number (first word) is here to prevent corrupted header from misleading the connection layer. If the magic number does not match,

the packet is dropped and the connection is shutdown to allow the remote peer to recover from its failure. The following example show a SITP PDU.

Frame 14 (1514 bytes on wire, 1514 bytes captured)

Ethernet II, Src: 00:00:00:00:00:00, Dst: 00:30:6e:26:69:04

Internet Protocol,

Src Addr: 131.104.48.208 (131.104.48.208),

Dst Addr: gradhp-03.imago_lab.cis.uoguelph.ca (192.168.0.131)

Transmission Control Protocol, Src Port: 8888 (8888), Dst Port: 56681

(56681), Seq: 1593469631, Ack: 1141322151, Len: 1448

Data (1448 bytes)

```

0000 00 30 6e 26 69 04 00 00 00 00 00 08 00 45 00  .On&i.....E.
0010 05 dc 71 e8 40 00 3e 06 4f d0 83 68 30 d0 c0 a8  ..q.@.>.0..h0...
0020 00 83 22 b8 dd 69 5e fa 6a bf 44 07 31 a7 80 10  .."...i^..j.D.1...
0030 2d 40 ef c7 00 00 01 01 08 0a 38 56 1b ba 0e a9  -@.....8V....
0040 7f 05>01 23 45 67*01 02*80 00*00 00 06 d0<67 72  ...#Eg.....gr
0050 61 64 68 70 2d 30 33 2e 69 6d 61 67 6f 5f 6c 61  adhp-03.imago_la
0060 62 2e 63 69 73 2e 75 6f 67 75 65 6c 70 68 2e 63  b.cis.uoguelph.c
0070 61 00 6d 69 6e 67 2e 63 69 73 2e 75 6f 67 75 65  a.ming.cis.uogue
0080 6c 70 68 2e 63 61 00 ff 82 04 01 00 7a 1d 69 e9  lph.ca.....z.i.
0090 05 85 03 fa 44 9e 32 59 bd 4b f2 40 ce 7d 22 46  ....D.2Y.K.@.}"F
00a0 84 98 1e 1f 68 d8 7b 56 33 02 a5 cd f5 3a a2 16  ....h.{V3.....

```

...

B.2 Routing Layer

The routing layer PDU header is quite simple and can be easily analyzed in the following packet dump. This packet is a SICP frame exchange between two machines

ADA rendez-vous layer.

Frame 10 (158 bytes on wire, 158 bytes captured)

Ethernet II, Src: 00:30:48:12:1e:62, Dst: 00:30:6e:26:69:04

Internet Protocol,

Src Addr: perseus.imago_lab.cis.uoguelph.ca (192.168.0.5),

Dst Addr: gradhp-03.imago_lab.cis.uoguelph.ca (192.168.0.131)

Transmission Control Protocol, Src Port: 8888 (8888), Dst Port: 56603

(56603), Seq: 1278004196, Ack: 1274768620, Len: 92

Data (92 bytes)

```

0000  00 30 6e 26 69 04 00 30 48 12 1e 62 08 00 45 00  .On&i...0H..b..E.
0010  00 90 95 b5 40 00 40 06 22 da c0 a8 00 05 c0 a8  ....@.@.".....
0020  00 83 22 b8 dd 1b 4c 2c cb e4 4b fb 6c ec 80 18  .."...L,..K.l...
0030  2d 40 20 00 00 00 01 01 08 0a 27 c1 5e 5f 0e 9d  -@ .....'.^_...
0040  47 48 01 23 45 67 01 02 00 00 00 00 00 50 67 72  GH.#Eg.....Pgr
0050  61 64 68 70 2d 30 33 2e 69 6d 61 67 6f 5f 6c 61  adhdp-03.imago_la
0060  62 2e 63 69 73 2e 75 6f 67 75 65 6c 70 68 2e 63  b.cis.uoguelph.c
0070  61 00*70 65 72 73 65 75 73 2e 69 6d 61 67 6f 5f  a.perseus.imago_
0080  6c 61 62 2e 63 69 73 2e 75 6f 67 75 65 6c 70 68  lab.cis.uoguelph
0090  2e 63 61 00*ff<00 21 7c 02 00 00 00 00 00 00  .ca...!|.....

```

You can see the destination server symbolic name, null terminated string, immediately followed by the source server symbolic name also null terminated. The PDU TTL (last byte) is set to its initial value (255) showing a direct connection between the two machines 'perseus' and 'gradhp-03'. The following example show how the routing algorithm works as a security key exchange reply PDU has been routed through the Imago gateway machine shown in the IP header. The source server field of the routing header reflect the original server and not the gateway, and the PDU TTL has been decremented accordingly.

Frame 2 (597 bytes on wire, 597 bytes captured)

Ethernet II, Src: 00:00:00:00:00:00, Dst: 00:30:6e:26:69:04

Internet Protocol,

Src Addr: draco.imago_lab.cis.uoguelph.ca (192.168.0.1),

Dst Addr: gradhp-03.imago_lab.cis.uoguelph.ca (192.168.0.131)

User Datagram Protocol, Src Port: 40824 (40824), Dst Port: 8888 (8888)

Data (555 bytes)

```

0000  00 30 6e 26 69 04 00 00 00 00 00 08 00 45 00  .On&i.....E.
0010  02 47 0a 4b 40 00 40 11 ac 86 c0 a8 00 01 c0 a8  .G.K@.@.....
0020  00 83 9f 78 22 b8 02 33 00 10 01 23 45 67 01 02  ...x"...3...#Eg..
0030  00 00 00 00 02 1f>67 72 61 64 68 70 2d 30 33 2e  .....gradhp-03.
0040  69 6d 61 67 6f 5f 6c 61 62 2e 63 69 73 2e 75 6f  imago_lab.cis.uo
0050  67 75 65 6c 70 68 2e 63 61 00*6d 69 6e 67 2e 63  guelph.ca.ming.c
0060  69 73 2e 75 6f 67 75 65 6c 70 68 2e 63 61 00*fe<  is.uoguelph.ca..
0070  b0 1e 00 00 08 00 cd 04 25 42 01 41 ff 82 31 8b  .......%B.A..1.
0080  da 68 94 22 33 67 fc 30 ba 2b d0 27 2a 1d 1c d5  .h."3g.0.+.'*...
0090  24 1d 1a af 6f d3 20 2b e0 6e a8 ba 11 f9 b5 9a  $....o. +.n.....
00a0  4d 4d 78 a0 6b b2 68 b0 91 0e 66 48 56 8d 19 47  MMx.k.h...fHV..G
00b0  1f 93 75 23 c6 dc a1 ee d8 3c 90 01 b6 1b bd e9  ...u#.....<.....
00c0  f2 e2 63 24 6a ab c5 3d 52 bf 26 77 ab 94 92 95  ..c$j...=R.&w....
00d0  a3 87 7d 5c b5 ac 90 7c 7d ca 43 96 9a 43 b2 e7  ..}\...|}.C..C..
00e0  fa 64 8f fd b9 8c b8 25 a3 01 80 78 2e 1d f8 30  .d.....%...x...0
00f0  0e 87 93 57 10 86 92 5a 55 19 88 38 a8 6a 6a b3  ...W...ZU..8.jj.
0100  14 80 14 cb ab 23 02 7d ce 39 8b fd be 4d 4a 5e  ....#..}.9...MJ^
0110  6a 99 e9 2b ad e3 0a 0e 16 48 ca c0 a9 34 6b 87  j...+.....H...4k.
0120  87 0b 17 52 f7 6d 50 6e 1b cb 4a 66 33 ed 60 2d  ...R.mPn..Jf3.'-
0130  36 dc 8e 3e 0f 0c 65 f1 12 af 6d 0b 76 a2 c8 9a  6..>...e...m.v...
0140  14 25 c1 f3 f9 f0 17 9b c5 bb c7 20 3f 3c 4d 67  .%..... ?<Mg
0150  08 36 65 02 d5 7b ab f5 ca 72 9b f9 2a eb 4f 97  .6e...{...r...*.0.
0160  c5 f8 10 98 e7 13 80 63 fa f7 9c eb db e4 ab f6  ....c.....
0170  62 4b 8d 05 06 99 00 06 23 02 00 99 93 5f 84 23  bK.....#....._.#
0180  c9 a9 49 31 ad 87 1e fc 6f 10 b0 fd 02 a4 b2 0f  ..I1....o.....
0190  0e 6b 42 b0 b5 90 74 0b 11 1d 5c 8b 5c dc 9d 71  .kB...t...\..\..q
01a0  c8 fa bb 25 0a 55 9f 82 cb 30 11 93 c7 27 dc 60  ...%.U...0...'. '

```



```

01b0  1c ef c2 63 aa 7e ae 97 6d 13 6b 00 a0 d5 35 da  ...c.~...m.k...5.
01c0  7e 06 69 62 f6 d4 f8 d0 e0 dc 83 f2 c1 2a 3c 12  ~.ib.....*<.
01d0  df 01 fe 2f af 33 cf 90 2a f5 7d e5 6d 2d 82 95  .../.3...*}.m-..
01e0  ff 8e 79 ab 4d ce df d7 18 88 ad 97 76 2d ba 1e  ..y.M.....v-..
01f0  ff 4a 2a bd d8 6b c4 fb 0a 1a b6 91 48 fe f8 5b  .J*..k.....H..[
0200  b0 af ee 81 88 4a e7 7c 84 6b 87 92 96 71 eb 44  ....J.|.k...q.D
0210  f0 ff 0b 02 00 86 61 f5 40 f4 25 09 1f 1a b4 a5  .....a.@.%.....
0220  fa 53 9f 0a 55 f3 90 26 6f 84 60 24 ca 00 5e c8  .S..U...&o.'$..^.
0230  b5 91 0d ea 9b ea 42 e2 79 ef 68 2b bc e7 41 cb  ....B.y.h+...A.
0240  aa c4 88 f2 31 94 e6 1c 78 42 b1 63 90 15 05 97  ....1...xB.c....
0250  a8 c3 d6 52 97                                     ...R.

```

B.3 Security Layer

B.3.1 RSA Key Exchange

When two servers do not know each others, they must exchange their respective key in order to be able to encrypt data. The following is a trace of a key exchange request initiated by the 'gradhp-03.imago_lab.cis.uoguelph.ca' server toward the 'perseus.imago_lab.cis.uoguelph.ca' server. You can clearly see the actual UDP packet holding the RSA key and DSA key.

Frame 1 (610 bytes on wire, 610 bytes captured)

Ethernet II, Src: 00:30:6e:26:69:04, Dst: 00:30:48:12:1e:62

Internet Protocol,

Src Addr: gradhp-03.imago_lab.cis.uoguelph.ca (192.168.0.131),

Dst Addr: perseus.imago_lab.cis.uoguelph.ca (192.168.0.5)

User Datagram Protocol, Src Port: 33057 (33057), Dst Port: 8888 (8888)

Data (568 bytes)

```

0000  00 30 48 12 1e 62 00 30 6e 26 69 04 08 00 45 00  .0H..b.On&i...E.
0010  02 54 47 47 40 00 40 11 6f 79 c0 a8 00 83 c0 a8  .TGG@.@.oy.....

```

```

0020 00 05 81 21 22 b8 02 40 c4 9a 01 23 45 67 01 02 ....!"...@...#Eg..
0030 00 00 00 00 02 2c 70 65 72 73 65 75 73 2e 69 6d .....,perseus.im
0040 61 67 6f 5f 6c 61 62 2e 63 69 73 2e 75 6f 67 75 ago_lab.cis.uogu
0050 65 6c 70 68 2e 63 61 00 67 72 61 64 68 70 2d 30 elph.ca.gradhp-0
0060 33 2e 69 6d 61 67 6f 5f 6c 61 62 2e 63 69 73 2e 3.imago_lab.cis.
0070 75 6f 67 75 65 6c 70 68 2e 63 61 00 ff>b1*1f*00 uoguelph.ca.....
0080 00*08 00*a2 58 40 05 c9 39 6c b8 94 f9 b4 3c 1e ....X0..9l....<.
0090 77 cc bb 8f 0d e8 f1 f7 e1 82 4f 08 3d dd 46 be w.....0.=.F.
00a0 74 d3 1d 43 b9 f8 34 cf df bb 89 11 15 bd c0 af t..C..4.....
00b0 1b 68 e1 92 39 e8 e3 f8 76 5e 1e d2 4a dc 08 f6 .h..9...v^..J...
00c0 d0 9d 54 04 5a 9d cc 94 b7 91 ea 58 46 fc 4d fb ..T.Z.....XF.M.
00d0 a3 b7 82 55 b9 56 c3 a8 d5 11 b0 46 09 4d a0 b3 ...U.V.....F.M..
00e0 9b c1 fa eb 25 c8 c5 78 8b 2c f6 bd 73 19 b2 7d ....%..x...s..}
00f0 5e 56 c9 db bd 8d 99 b9 f0 56 e6 ed 64 5f 92 ec ^V.....V..d_..
0100 23 4c e2 ef 27 03 ab 2e 52 c1 be d5 73 37 b1 c8 #L...'...R....s7..
0110 93 de bb 54 aa d1 a5 17 97 87 53 60 d6 0f ce 2f ...T.....S'.../
0120 4d 81 4c f6 de af 92 0f 42 5d 17 0e 8f 6f be 43 M.L.....B]...o.C
0130 80 8a 09 8d 3f fb 78 c1 f3 ce 6f 99 03 62 c6 de ....?.x...o..b..
0140 56 23 e1 16 a2 13 6d 27 38 92 ab 64 10 66 9f 1a V#....m'8..d.f..
0150 dc 86 b9 fd 29 42 e2 2b 40 e0 d7 13 2c 34 da 5b ....)B.+@...,4.[
0160 e3 35 66 fb b6 89 0d 2f 09 80 c1 d7 6b 3b c8 7b .5f..../....k;.{
0170 e7 7f 03 a8 14 c2 8e 59 4f 33 58 2e b2 b9 e5 c1 .....Y03X.....
0180 cf be 83*00 06*23*02 00*87 4b 58 61 8b a5 f1 7f .....#...KXa....
0190 b4 2d db a5 0f 55 ab f3 b5 cd ba 35 67 54 82 6f .-...U.....5gT.o
01a0 c6 5f 5d ca ce 2e 62 8b 47 78 87 bf 09 46 9f b9 ._]...b.Gx...F..
01b0 7e 0e 4c 65 9d 23 df f2 56 72 66 64 72 a7 70 37 ~.Le.#..Vrfrdr.p7
01c0 15 4b be a1 57 98 24 97*00 a0*e6 c7 b5 af 77 82 .K..W.$.....w.
01d0 05 1a 43 f3 dc e2 c9 b6 c4 38 b2 21 22 5b*02 00 ..C.....8.!"[..
01e0 *80 70 45 35 ea 0b d7 99 db 9b e3 d3 e4 ea 95 de .pE5.....
01f0 7e c9 0e 14 9f 0f d2 38 06 fe 57 f8 fc 62 ea 4a ~.....8..W..b.J
0200 28 41 67 99 66 26 b4 4d d2 93 95 18 f2 c8 76 f3 (Ag.f&M.....v.

```

```

0210 68 13 5a a5 74 ee 01 79 2a bd 16 8e 0f 18 6e 1f  h.Z.t..y*.....n.
0220 *01 ff*54 32 88 a9 f0 73 8f 37 e5 68 65 24 84 e2  ..T2...s.7.he$..
0230 71 b2 c3 7a be 7d bc 2a 38 c5 40 75 db c0 f3 74  q..z.}*8.@u...t
0240 39 2d 80 ff f5 5c fc 67 18 7b 6b fc fe ac e8 ea  9-...\.g.{k.....
0250 4c a0 3d a7 de 5b 09 82 2a 41 7a 9b d9 23 60 b1  L.=...[*Az...#'.
0260 3b ca< ;.

```

Using the table 5.3 and the value of the flags in this security PDU (0xb1 or 10110001b), we can see that this is a key exchange request, that the PDU is not signed and both, RSA and DSA keys are provided inside the PDU. The following field is the supported cipher mask (0x1f) followed immediately with the request ID (0x00) and the different component of the two keys. The following is the reply made by the server.

Frame 2 (610 bytes on wire, 610 bytes captured)

Ethernet II, Src: 00:30:48:12:1e:62, Dst: 00:30:6e:26:69:04

Internet Protocol,

Src Addr: perseus.imago_lab.cis.uoguelph.ca (192.168.0.5),

Dst Addr: gradhp-03.imago_lab.cis.uoguelph.ca (192.168.0.131)

User Datagram Protocol, Src Port: 32874 (32874), Dst Port: 8888 (8888)

Data (568 bytes)

```

0000 00 30 6e 26 69 04 00 30 48 12 1e 62 08 00 45 00  .On&i..0H..b..E.
0010 02 54 5e 56 40 00 40 11 58 6a c0 a8 00 05 c0 a8  .T^V@.@.Xj.....
0020 00 83 80 6a 22 b8 02 40 a0 3e 01 23 45 67 01 02  ...j"...@.>.#Eg..
0030 00 00 00 00 02 2c 67 72 61 64 68 70 2d 30 33 2e  ....,gradhp-03.
0040 69 6d 61 67 6f 5f 6c 61 62 2e 63 69 73 2e 75 6f  imago_lab.cis.uo
0050 67 75 65 6c 70 68 2e 63 61 00 70 65 72 73 65 75  guelph.ca.perseu
0060 73 2e 69 6d 61 67 6f 5f 6c 61 62 2e 63 69 73 2e  s.imago_lab.cis.
0070 75 6f 67 75 65 6c 70 68 2e 63 61 00 ff>b0*1e*00  uoguelph.ca.....
0080 00*08 00*af b2 02 dd b4 cf f3 7b 02 58 85 44 45  .....{.X.DE
0090 72 7a 1e 43 8d 68 e1 40 c1 03 b8 86 ba d1 d2 71  rz.C.h.@.....q
00a0 76 a4 8a 6e 50 2a 88 33 ad 7e b3 e5 e9 f5 f5 84  v..nP*.3.~.....
00b0 b5 b9 73 86 59 09 d6 68 7a c3 04 95 79 1e b2 7f  ..s.Y..hz...y...
00c0 6d 2b 65 87 3d 1d 77 4e 1d f5 42 22 fe ee 9f 7f  m+e.=.wN..B"....
00d0 0d 1e 28 69 30 28 e4 30 84 e6 af 77 73 02 0f 66  ..(i0(.0...ws..f
00e0 06 e4 ec 01 79 ce fa 61 fd 2e 09 bb 03 24 be 2c  ....y..a.....$.

```

```

00f0 05 78 df 72 75 a3 3e fc ea e6 5b fe af 25 aa ee .x.ru.>...[.%.
0100 9c 63 7f 1e e1 d6 05 6a 37 29 71 bd ec c7 a4 2c .c.....j7)q....,
0110 fb 29 0a 67 7d 85 7f 20 0b 37 0c ac 89 87 ea c2 .).g}... .7.....
0120 5c 4c e3 53 f3 7f 03 7e a4 0c c9 b8 3e 50 c1 14 \L.S...~....>P..
0130 4f 53 cb e0 8c 23 a8 ce 95 28 e7 b2 ea 54 73 8b OS...#...(...Ts.
0140 be 2e d3 9e 9d cd 10 42 75 a3 96 97 70 21 8c 2f .....Bu...p!./
0150 34 31 7c a0 d4 29 09 59 c9 a3 5c 5f 0a 7c 93 cb 41|...)Y..\_|..
0160 1c e4 54 09 c2 32 59 3f 24 f2 b0 41 af 35 fd 5c ..T..2Y?$..A.5.\
0170 8b dc 72 fe 56 20 32 54 17 4f 05 38 8f e6 e0 94 ..r.V 2T.O.8....
0180 6b 2a 49*00 06*23*02 00*d3 c6 85 83 13 bf 0b a9 k*I.#.....
0190 03 b4 81 a7 a7 7b 7f dd 5f f5 1e b5 d5 a9 b7 57 .....{..._.....W
01a0 e8 19 8c 93 92 7f 82 db 7a 23 1e a8 90 b2 84 e4 .....z#.....
01b0 5f f1 35 a3 7d ab c4 ca 73 9e 2f e4 97 af 34 c4 _5.}...s./...4.
01c0 2e d6 27 f3 12 98 7a 4b*00 a0*e7 d9 bb b5 60 95 ..'....zK.....'.
01d0 62 56 9a cb 7c 03 7b 76 d1 eb 15 cc c9 db*01 ff bV...|.{v.....
01e0 *64 43 79 af 15 57 9a c0 d1 21 13 88 df 05 91 f2 dCy..W...!.....
01f0 cf d2 29 8f 96 1d 7e ae 00 ca 73 d2 c6 96 de f0 ..)....~....s.....
0200 72 9d d6 f7 12 ca 95 19 38 a4 92 4a c5 3f f6 ae r.....8..J?...
0210 b6 b3 78 eb 1a 85 1e ae f9 89 4d 30 f8 8a 11 93 ..x.....MO....
0220 *02 00*b6 5e e1 e8 55 d6 d2 32 8d 21 d0 19 fe 5a ...~..U..2.!...Z
0230 eb c2 41 f7 28 99 1c 6f 1c da 8c df d6 f7 94 3d ..A.(.o.....=
0240 78 dc 08 47 5e 52 06 43 f4 ee 47 3a 37 ca 9f 21 x..G~R.C..G:7..!
0250 87 39 ca 43 d0 a2 d7 a9 a5 fe 6b 01 fd 5c 9b 58 .9.C.....k..\X
0260 dc b4< ..

```

B.3.2 Encrypted Packet

This is an example of a security layer PDU with an encrypted PDU.

Frame 6 (1514 bytes on wire, 1514 bytes captured)

Ethernet II, Src: 00:30:6e:26:69:04, Dst: 00:30:48:12:1e:62

Internet Protocol,

Src Addr: gradhp-03.imago_lab.cis.uoguelph.ca (192.168.0.131),

Dst Addr: perseus.imago_lab.cis.uoguelph.ca (192.168.0.5)

Transmission Control Protocol, Src Port: 56603 (56603), Dst Port: 8888

(8888), Seq: 1274766171, Ack: 1278004196, Len: 1448

Data (1448 bytes)

```

0000 00 30 48 12 1e 62 00 30 6e 26 69 04 08 00 45 00 .OH..b.On&i...E.
0010 05 dc 24 14 40 00 40 06 8f 2f c0 a8 00 83 c0 a8 ..$.@.@../.....
0020 00 05 dd 1b 22 b8 4b fb 63 5b 4c 2c cb e4 80 10 ....".K.c[L,....
0030 16 d0 ca 05 00 00 01 01 08 0a 0e 9d 47 48 27 c1 .....GH'.
0040 5e 57 01 23 45 67 01 02 80 00 00 00 09 85 70 65 ^W.#Eg.....pe
0050 72 73 65 75 73 2e 69 6d 61 67 6f 5f 6c 61 62 2e rseus.imago_lab.
0060 63 69 73 2e 75 6f 67 75 65 6c 70 68 2e 63 61 00 cis.uoguelph.ca.
0070 67 72 61 64 68 70 2d 30 33 2e 69 6d 61 67 6f 5f gradhp-03.imago_
0080 6c 61 62 2e 63 69 73 2e 75 6f 67 75 65 6c 70 68 lab.cis.uoguelph
0090 2e 63 61 00 ff>82*04*01 00*01 4e f0 af d6 0d 97 .ca.....N.....
00a0 59 58 1c 81 44 d3 16 ad f2 98 aa 60 84 97 44 de YX..D.....'.D.
00b0 71 57 2e c1 68 48 6d 63 df a1 2b 9b d4 47 4e e2 qW..hHmc.+..GN.
00c0 24 79 2d 56 4f f1 6f ad f8 71 3f 20 ba f2 fb c8 $y-VU.o..q? ....
00d0 af 8b 36 79 0b b0 24 5e ba 99 b4 19 5f 28 d5 a1 ..6y..$^....._(...
00e0 2a 0a 0c 68 ae 6e f4 70 9b 4a 27 09 09 d9 a3 a9 *..h.n.p.J'.....
00f0 24 12 a3 31 68 fb fa 69 f8 e3 da 11 d0 c0 72 ef $.1h..i.....r.
0100 a8 f9 b1 19 2a ec 2e 82 67 3d 55 31 a4 d1 a1 4c ....*...g=U1...L
0110 5e 1d ee 4a eb 2c 5d ad 5c 34 97 71 8f bc 22 32 ^..J.,].\4.q.."2
0120 f5 44 90 36 55 86 c8 25 97 86 63 fa 0d 65 48 7c .D.6U..%...c..eH|
0130 96 d3 4d 3c 40 b2 a7 67 b3 38 37 ee f8 36 88 c0 ..M<@..g.87..6..
0140 dd 1c 8c 4a 7a 56 d2 02 13 04 e6 b6 a9 b8 52 0e ...JzV.....R.
0150 c4 11 ec 81 8f b7 81 34 e8 61 43 6c 83 d2 97 e8 .....4.aCl....
0160 b5 ee be e9 20 f8 2d 66 af ba 0c 4b c6 af 9f a7 ....-f...K....
0170 e0 48 bd 72 37 74 83 e0 c4 b9 89 4c 62 bc 9f 6d .H.r7t.....Lb..m
0180 96 00 1c ac 14 0b 48 e9 06 f0 c3 ed 58 34 42 53 .....H.....X4BS
0190 ab 76 6b 09 68 ee 5e 04 14*01 00*3c 94 e6 de 58 .vk.h.^....<...X
01a0 80 7c 77 73 b8 81 84 38 b4 1b ab 14 1c 68 95 48 .|ws...8.....h.H
01b0 bb d9 f4 6a 21 19 1a d6 fe 28 f3 11 9d af ac 0a ...j!.....(.....

```

```

01c0 b5 64 d0 c6 e2 5c f9 6a f3 71 49 ac 49 0e 1e be .d...\j.qI.I...
01d0 89 08 13 a7 d4 fe 73 d9 0f 16 88 4d 62 03 19 a2 .....s....Mb...
01e0 86 46 6e 9e 60 e6 59 81 b9 ec 38 ea 33 d8 9d 24 .Fn.'Y...8.3..$
01f0 48 05 b2 c3 76 e8 f0 14 03 d5 4d f9 07 c3 6e d5 H...v.....M...n.
0200 b6 5b 37 b7 36 a6 17 ee 9e 7b e7 87 14 32 b0 11 .[7.6....{...2..
0210 90 46 65 84 8b 3f dd 14 08 cb c1 17 b1 1d 91 89 .Fe..?.....
0220 5c 3c 4d ca 26 db b7 7c 3c 9d 2d fc 68 3f c2 93 \<M.&...|<.-.h?...
0230 33 a3 94 d2 e8 d6 a5 b2 1a 11 db d1 a8 5f a3 e7 3....._...
0240 3d f9 2c 9f 0c 64 f1 50 73 43 40 93 ea 41 46 68 =.,...d.PsC@..AFh
0250 b9 2e 08 e4 3c db 80 04 e8 46 8e 7a f1 64 ec 86 ....<....F.z.d..
0260 48 29 b9 52 2e 13 88 79 88 1f 37 10 90 9f 75 7b H).R...y..7...u{
0270 45 8f 45 9a e7 5a 73 1a 64 50 d2<d4 30 93 21 c5 E.E..Zs.dP..0.!.
0280 05 8b a1 81 ea ad 95 b4 8c 56 fe c6 0f 09 0f 1b .....V.....
0290 80 1b 38 e9 b7 79 c9 cf 53 ee 89 d5 ec dd 26 2e ..8..y..S.....&.

```

...

The remaining encrypted data are not shown in the packet output because they are irrelevant. From the header, we can see that the data have been encrypted using the 3DES-CBC encryption schema (cipher: 0x04) which is the one selected as a default. This security header is exactly 472 bytes long, but this may vary according to the selected RSA and DSA key length.

B.4 ADArendey-vous Layer

Frame 10 (145 bytes on wire, 145 bytes captured)

Ethernet II, Src: 00:30:6e:26:69:04, Dst: 00:00:00:00:00:00

Internet Protocol, Src Addr: gradhp-03.imago_lab.cis.uoguelph.ca (192.168.0.131), Dst

Transmission Control Protocol, Src Port: 57350 (57350), Dst Port: 8888 (8888), Seq: 26

Data (79 bytes)

```

0000 00 00 00 00 00 00 00 30 6e 26 69 04 08 00 45 00 .....0n&i...E.

```

```

0010 00 83 7e f3 40 00 40 06 46 1e c0 a8 00 83 83 68  ..~.@@.F.....h
0020 30 d0 e0 06 22 b8 9b ab a1 5c b6 a5 30 b9 80 18  0...".....\..0...
0030 16 d0 bc 3c 00 00 01 01 08 0a 0e ef e4 fe 39 be  ...<.....9.
0040 9c bf 01 23 45 67 01 02 00 00 00 00 00 43 6d 69  ...#Eg.....Cmi
0050 6e 67 2e 63 69 73 2e 75 6f 67 75 65 6c 70 68 2e  ng.cis.uoguelph.
0060 63 61 00 67 72 61 64 68 70 2d 30 33 2e 69 6d 61  ca.gradhp-03.ima
0070 67 6f 5f 6c 61 62 2e 63 69 73 2e 75 6f 67 75 65  go_lab.cis.uogue
0080 6c 70 68 2e 63 61 00 ff 00>f5 15*04*00*00 00 00  lph.ca.....
0090 00<

```

This SICP dump of an 'ACK' packet generated after the server 'gradhp-03' got a 'ACCEPT?' message. You can see the generation id (0xf515), packet type (0x04 == ACK) and the sequence ID (0x00000000). because this is only a SICP packet, no data are present and the security (encryption) is disabled for speedup.

B.5 Marshaling Layer

Frame 4 (1514 bytes on wire, 1514 bytes captured)

Ethernet II, Src: 00:30:6e:26:69:04, Dst: 00:00:00:00:00:00

Internet Protocol,

Src Addr: gradhp-03.imago_lab.cis.uoguelph.ca (192.168.0.131),

Dst Addr: 131.104.48.208 (131.104.48.208)

Transmission Control Protocol, Src Port: 57350 (57350), Dst Port: 8888

(8888), Seq: 2611714580, Ack: 3064279146, Len: 1448

Data (1448 bytes)

```

0000 00 00 00 00 00 00 00 30 6e 26 69 04 08 00 45 00  ....On&i...E.
0010 05 dc 7e f0 40 00 40 06 40 c8 c0 a8 00 83 83 68  ..~.@@.@.....h
0020 30 d0 e0 06 22 b8 9b ab 9a 14 b6 a5 30 6a 80 10  0...".....0j..
0030 16 d0 f1 27 00 00 01 01 08 0a 0e ef e4 fd 39 be  ...'.....9.
0040 9c bd 01 23 45 67 01 02 80 00 00 00 07 3c 6d 69  ...#Eg.....<mi
0050 6e 67 2e 63 69 73 2e 75 6f 67 75 65 6c 70 68 2e  ng.cis.uoguelph.

```

```

0060 63 61 00 67 72 61 64 68 70 2d 30 33 2e 69 6d 61 ca.gradhp-03.ima
0070 67 6f 5f 6c 61 62 2e 63 69 73 2e 75 6f 67 75 65 go_lab.cis.uogue
0080 6c 70 68 2e 63 61 00 ff 00 f5 15 01 00 00 00 00 lph.ca.....
0090 00>75 15 00 00*01*77 61 6e 64 65 72 65 72 30 32 .u....wanderer02
00a0 00*67 72 61 64 68 70 2d 30 33 2e 69 6d 61 67 6f .gradhp-03.imago
00b0 5f 6c 61 62 2e 63 69 73 2e 75 6f 67 75 65 6c 70 _lab.cis.uoguelp
00c0 68 2e 63 61 00*a8 c0 83 00*00*00 00 00 00*00 00 h.ca.....
00d0 00 00*40 be 36 e4*40 b6 37 28*40 b6 36 d4*40 b6 ..@.6.@.7(@.6.@.
00e0 37 28*40 b6 37 60*40 b6 36 f4*40 b6 30 40*40 b6 7(@.7'@.6.@.0@@.
00f0 37 5c*40 b6 30 c0*40 b6 30 08*40 b6 30 40*40 b6 7\@.0.@.0.@.0@@.
0100 35 8c*40 b6 35 ec*40 b6 36 d4*40 be 36 f0<1f 04 5.@.5.@.6.@.6...
0110 80 00 1c 30 b6 40 1f 04 00 01 54 30 b6 40 4f 05 ...0.@....T0.@.
0120 00 01 58 31 b6 40 7f 05 00 02 b4 31 b6 40 4f 05 ..X1.@.....1.@.

```

...

The remaining part of the Ethernet frame has been truncated as it is the actual content of the Imago code and is not required. The security has been disabled (security header set to 0x00) to show the actual content of the PDU in a non-encrypted version. You can see the application ID (0x75150000), the Imago type (0x01 for worker), the Imago name ("wanderer02") and all other information encapsulated in the marshaling header.

C PERFORMANCE ANALYSIS PROGRAM

This is the complete testing program used to get the performance analysis results.

```
// IMAGO Prolog performance testing program
// Three different schema: simple|oneway|growing
args(4,12000,[
    stat(simple,"perseus.imago_lab.cis.uoguelph.ca",
        "orion.imago_lab.cis.uoguelph.ca"),
    stat(oneway,"perseus.imago_lab.cis.uoguelph.ca",
        "orion.imago_lab.cis.uoguelph.ca"),
    stat(growing,"perseus.imago_lab.cis.uoguelph.ca",
        "orion.imago_lab.cis.uoguelph.ca")
]):-!.

// The Queen Imago start a worker and wait until it has completed
:- stationary(test1).

test1(_):-
    args(_,List),
    do_test1("GnuPlot Worker", List),
    write("Statistics Terminated"),nl,
    sleep(10),
    terminate.

do_test1(_,[ ]).

do_test1(Worker, [Head|Tail]):-
    atom_concat(Worker,"_aux",Worker1),
    create("stat_test1.lo", Worker1, Head),
    wait_accept(_, Msg),
```

```

do_result(0, Msg),
do_test1(Worker1, Tail).

do_result(N, done):- !.
do_result(N, result(Time,Specs)):-
    write(N),write(" "),
    write(Time),write(" #"),
    write(Specs), nl,
    wait_accept(_, Msg),
    N1 is N + 1,
    do_result(N1, Msg).
:-end_stationary(test1).

// The Worker Imago.
:-worker(stat_test1).
stat_test1(Serv):-
    args(Rounds,Count,_),
    write("# args("),write(Rounds),write(","),
    write(Count), write(","),
    write(Serv),write(")"),nl,
    do_main(Count,Serv),
    sleep(10),
    dispatch("$oneway_messenger","queen",done),
    dispose.

do_main(N,stat(simple,Home,Server)):-
    do_simple(N,Home,Server).
do_main(_,stat(growing,Home,Server)):-
    args(R,Counts,_),

```

```

    N is Counts / R,
    do_growing(R,N,Home,Server).
do_main(N,stat(oneway,Home,Server)):-
    clocksync(Home,Server,Drift),
    do_oneway(N,Home,Server,Drift).
do_main(_,Msg):-
    write("Unkown statistic: "), write(Msg), nl.

// Compute the time to and back from some host.
// Then, send the result to the queen
do_simple(0,_,_):- !.
do_simple(N,Home,Server):-
    do_simple_stat(N,Home,Server),
    N1 is N - 1,
    do_simple(N1,Home,Server).

do_simple_stat(N,Home,Server):-
    roundtrip(Home,Server,T2),
    dispatch("$oneway_messenger","queen",
        result(T2,simple(N,Home,Server,T2))).

// One way only with clock drift computation
do_oneway(0,_,_,_):- !.
do_oneway(N,Home,Server,Clock):-
    do_oneway_stat(N,Home,Server,Clock),
    N1 is N - 1,
    do_oneway(N1,Home,Server,Clock).

do_oneway_stat(N,Home,Server,Clock):-

```

```

oneway(Home,Server,Clock,T),
dispatch("$oneway_messenger","queen",
        result(T,oneway(N,Home,Server,Clock,T))).

// Roundtrip with an array of time value.
do_growing(0,_,_,_):- !.
do_growing(R,N,Home,Server):-
    do_growing_stat(R,N,Home,Server),
    R1 is R - 1,
    do_growing(R1,N,Home,Server).

do_growing_stat(R,N,Home,Server):-
    do_growing_round(R,N,Home,Server,List).

do_growing_round(_,0,_,_,[]):- !.
do_growing_round(R,N,Home,Server,[T2|L]):-
    roundtrip(Home,Server,T2),
    dispatch("$oneway_messenger","queen",
            result(T2,growing(R,N,Home,Server,T2))),
    N1 is N - 1,
    do_growing_round(R,N1,Home,Server,L).

// Tools used in the differents statistics cases.
roundtrip(Home,Server,Time2):-
    move(Home),           // First, make sure where we are.
    ctime(Ta1),           // Grab the time here.
    move(Server),         // Move to our destination
    move(Home),           // And back again as fast as we can
    ctime(Ta3),           // Time back to home.

```

Time2 is (Ta3-Ta1). // Compute the roundtrip time

oneway(Home,Server,Clock,Time):-

```

move(Home),           // First, go back home.
ctime(Ta1),           // Gather the Time at home.
move(Server),         // Move to the server.
ctime(Tb2),           // Get the time on the server.
Time is (Tb2+Clock)-Ta1. // Compute the adjusted time

```

clocksync(Home,Server,Drift):-

```

move(Home), // Make sure to know where we are.
ctime(Ta1), // grab the time on machine A
move(Server), // Move to Machine B
ctime(Tb2), // Arrival time on machine B
move(Home), // Move back Home
ctime(Ta3), // Arrival time on Machine A
Ta2 is (Ta3+Ta1)/2, // Get the Real time Ta2.
Drift is (Ta2-Tb2). // The Clock drift value.

```

:-end_worker(stat_test1).