ON THE IMPLEMENTATION OF AN AGENT-MIGRATION PROTOCOL

Guillaume Autran and Xining Li

Department of Computing and Information Science
University of Guelph
Guelph, ON, Canada N1G 2W1
gautran@uoguelph.ca xli@cis.uoguelph.ca

ABSTRACT
This paper 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 (MA) 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. This paper 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.

Keywords: Imago, Agent-migration protocol, Multiple threading, Thread pool

1. INTRODUCTION
In general, communication of data over the network is done in two distinct ways. A low level approach uses a binary representation of data to transmit messages. In this case, we agree upon a universal formatting such as Big-endian/Little-endian and everything is transmitted "AS IS". Java serialization based MA systems, such as Concordia [1] and Aglets [2], rely on a binary representation of their data. On another hand, ASCII representation is often used in transmitting text oriented data. The SMTP protocol (email) is based on the 7-bits ASCII code. D’Agents [3] also uses an ASCII oriented communication. However, this approach brings several drawbacks. For instance, we need to add control information on top of the raw data, and we should provide functions to transform the binary to ASCII format, etc. Solutions to these problems are usually time consuming and often increase the size of the original data (base64 translation). Since the binary-oriented scheme has advantages to be more flexible and usually faster than the ASCII-oriented scheme, the IMAGO server will use binary stream transmission in the implementation of agent-migration protocol.

To make it easier to deal with issues involved in different layers of communication, a stack architecture has been chosen for the IMAGO agent-migration protocol. In our current implementation, five layers have been identified. They are, from bottom to top: Connection, Routing, Security, ADA’rendezvous and Marshaling. The major considerations of the Imago migration protocol are:

Speed Communication speed is an important factor in most network application.

Lightweight Memory and CPU usage should remain low even under heavy load.

Flexible Adapt to variable network configuration and insure delivery even when direct IP connection is not possible.

Reliable Packet loss should be prevented and network error reported in order to certify the success/failure semantic.

Security Data protection and source authentication should be one of the major security concern while moving an agent to the destination.

Multi-thread MLVM server requires multiple threads for multiple data flow (MIMD) while handling Imago Processes.

Unfortunately, some design considerations are in conflict with one another. For example, a high level of security comes with a cost. The CPU intensive encryption, decryption, as well as their memory requirements, must be taken into consideration while designing a fast speed communication model. 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.
2. SERVER IMPLEMENTATION

Generally speaking, an Imago, as described in [4], is composed of three parts: its identifier which is unique to distinguish with others, its code which corresponds to a certain algorithm, its execution thread which is maintained by a single memory block (a merged stack/heap with automatic garbage collection) [8]. Those information segments represent themselves a self contained control block and are very well suited to be ran by a thread of control such as a processes due to their structure similarity. Threads (or Light Weight Processes) hold many advantages compared to regular Operating System processes as they share the same code area, system resource table (descriptors, signals), identity information and heap area, with one another as described in [7]. Also, information sharing between threads is greatly simplify and scaled down to simple memory access (to shared variables) without requiring any operating system intervention. Imago threads may be associated with operating system threads. However, a direct one to one mapping has many disadvantages, such as a scalability weakness. In general, the slow down factor introduced by spawning threads in an anarchic manner is not desirable. Besides, The life cycle of an Imago is an alternation of processing time with waiting time: waiting for messengers, waiting for completion of a database query and so on. In that case, the resources held by the system thread are either wasted or unused. In our thread implementation algorithm, no thread is allowed to be switch into a blocking state as long as there is processing to be done.

Versatility in the threading algorithm is that a hardware configuration replacing threads with CPUs would accomplish the exact same behavior with little changes. To achieve such goal, a thread pool is been created and initialized with a user defined number (typically 5 to 10). A highest priority first scheduling algorithm is then applied on a sorted set of queues. Each thread will scan those queues and process any task available according to the priority setting until no more task is to be handled. At this time, and at this time only, the unused thread will be put into a blocking state until new tasks are generated. The scheduler task list contains entries which match different stages of the life cycle of an Imago, such as, creation, execution, memory related processing (expansion, contraction or collection), termination and migration (outgoing and incoming). The sorting of the priority list is such that the outgoing migration related tasks are of a highest priority followed by Imago creation and incoming migration request tasks. The last entry in the list are, respectively, memory related tasks and the Imago virtual machine engine, because they will be the most frequently requested cycle in the life of an Imago. Such ordering prevent starvation by simply using the natural life cycle of the Imago itself. The synchronization issues, that arise from a multi-threaded environment, have been resolved by the use of mutual exclusion primitives. Special care is taken as too much mutual exclusion would hurt the concurrency by translating a multi-threaded application into a single thread program (all other threads are blocked while waiting for the current thread to release locks).

Practically, the versatile thread pool algorithm used in this project revealed to be of a great efficiency and low response time, especially on multi-processor machines where concurrent execution of system thread becomes possible.

The Imago Prolog Logic Virtual Machine core is invoked by the thread pool scheduler. The engine picks from the ready queue a new Imago to execute. From the Imago Control Block (ICB), the engine loads the registers saved by the previous run and start executing the program where it has left off, until a new context switch is requested or required. At this time, all prolog registers are saved in the ICB structure and the Imago is put at the back of the ready queue. From an Imago point of view, the context switch is totally transparent. Unlike many other virtual machine implementations, MLVM does not implement a time slicing (or time sharing) algorithm to dictate the context switching. The complexity introduced would be greater than its gain and context switch decided by the expiration of a asynchronous timer may break the atomicity of the execution of the Imago prolog instruction. Instead, context switch occur at predefined locations called context switch point. Those context switch points are located in such way that it becomes virtually impossible for a program to hold the thread resources indefinitely (or for very long period of time). All transition in the life of our Imago agent are perfect candidates for context switching and are used in this purpose. Those transitions include migration requests, cloning and creation request as well as messenger dispatching or accept request. As well as the more prolog general life transitions such as garbage collection, memory expansion or contraction and the like.

The Memory manager module manipulates memory expansion and garbage collection requirements. Memory is not an expandable resource, extra care must be taken while allocating room for an Imago to grow in. Therefore, the Imago will, at first, grows exponentially up to a limit to reduce the number of memory expansion, then linearly thereafter. In case of memory allocation failure, the Imago is placed into a waiting for memory queue and be released (along with all other waiting Imagoes) to compete for memory as soon as some becomes available.

The creator module handle all the Imago creation requests, such as cloning, dispatching a messenger and creating a new Imago from a file out of the hard drive storage. First, a new Imago Control Block is allocated, then, the program code is loaded into memory. Registers and stack must be initialized in the same way as a prolog procedure call.
would do. Once the ICB is ready, it is passed to the engine queue and set ready to run.

Imago migration is handled by the migration protocol stack. The in and out modules are interfacing with this stack, receiving and sending agents. Also, the out module takes care of releasing the resources used by an Imago when this one is on a termination stage. As soon as an Imago moves out, all its messengers presently attached to it are released and notified of the new location for the specific destination.

3. MIGRATION PROTOCOL DESIGN

An Imago block contains five segments. At the bottom comes the procedure table along with the program code. Immediately after follows the name table with some free space to allow dynamic name expansion. On top of the name table, goes the execution stack plus a large amount of free space to allow the stack to expand upward. Last at the top of the memory block is the trail used to remember the roots in different garbage collection generations as well as roots for backtracking. The trail grows downward and uses the same free space as the stack.

The marshaling process takes an Imago memory block as input and compact the block as the output. The compaction process does not alter any of the data contained within the Imago block as it does not know how to interpret them. This compacted block is then passed to the lower layer in the migration stack. On the other hand, the unmarshaling process restores the original block from a compact block.

Reliable migration must involve some overhead. Like most of the reliable transmission protocols, such as TCP, the Imago migration stack is based on a connection oriented schema. However, rather than using a strong (or direct) connection, our migration algorithm implements a weak connection, such as it remains at a higher abstract level. The protocol expects lower layer to accomplish proper delivery through connection oriented or connectionless algorithm but makes no assumptions about the status of the underlying network. The ADA rendezvous algorithm is used to insure a proper fault recovery and insure a reliable migration through the true/false semantics. A detail description of the ADA rendezvous algorithm can be found at the Department of Defense for reference, however, our implementation will be described in this paper as it differs slightly from the original one and has been adapted to fit the mobile agent migration.

At first, a copy of the upper layer PDU is transmitted along with the layer encapsulation datad which includes a sequencing number, a generation number and some PDU flags. PDU flags indicate the type of packet transmitted or received. If all goes fine, this request PDU is received at the destination side which generates an accept offer packet. Upon receiving the accept offer, the sender assures that the PDU has safely reached the destination. It thus generates an acknowledgment response and then, safely discards its own copy of the original PDU. Receiving the acknowledgment instructs the destination peer that it should reload its copy of the PDU and release it up the stack to the marshaling layer. If at any given time, one peer fails to receive the expected response within a specific amount of time, it will initiate a fault recovery procedure. For example, if the sender does not receive the accept offer following its request, it will retransmit the request up to a predefined amount of time. Failure for all retransmission will result in a failure of the Imago migration and the original PDU will be notified of the error and forwarded back up the stack. Eventually, the Imago, originating the move, will be reinserted back into the engine queue and the built-in predicate procedure call would have failed. This scheme insures a true/false semantics proper to the Prolog implementation. Similarly, if the acknowledgment packet is missing, the receiver side will retry transmitting the accept offer until a response is received (either positive or negative acknowledgment) and take action accordingly.

The problem of security in the mobile agent world is far beyond a simple case. Therefore, this paper does not pretend to bring a solution to all issues related to agent mobility, but rather discuss a small, simple, yet effective, way of securing migrating agents. The following assumptions are made that neither the sender, nor the receiver host (namely Alice and Bob in this paper) are malicious and that Imago may be routed through some unknown number of intermediate hosts before they can reach their final destination. Using these assumptions, a malicious host (namely Oscar), may find a way to get a hold on the Imago in transit and attempt to read or alter the content of its data. The security layer discussed here is an attempt to prevent Oscar from doing so.

Initially, Alice and Bob hold a set of RSA and DSA keys. When Alice wants to communicate with Bob, she first searches her local database to see either she finds Bob’s public RSA key. If Bob is not known, she will generate a control packet requesting Bob’s security information, i.e., public RSA key and public DSA key along with all Bob’s supported ciphers. She will, at the same time, include her own security information within the request so Bob can cache it and use it later on (while replying to Alice for example). Upon receiving this request, Bob will send its own security information to Alice. Then both sides will use, from now on, the cached RSA key and DSA key, from the other host, to encrypt and verify the information received.

Upon receiving the security PDU, Bob first decrypts session key and signature using its own RSA key. Then, using Alice public DSA key, Bob verifies the authenticity of the session key. If this authenticity can be established, Bob will
trust this PDU and he can start decrypting the remaining data using the original session key, then forward it to the upper layer. This simple security solution provides a fairly safe and flexible environment and makes the encryption and decryption overhead to a minimum.

As often as possible, a direct connection between hosts is used to migrate Imagoes. However, in some particular cases, name resolution may not succeed. For example, nodes within an internal private network protected by a firewall, may not be accessed directly from another computer on the Internet. In such case, the only alternative, a traditional mobile agent would have, is to jump to intermediate nodes prior to reach destination. Those intermediate nodes would have to be programmed (or discovered by the agent) and should have a full installation of the environment able to run the mobile agent. The algorithm used in the Imago Project achieves routing across unreachable sub network in a totally transparent and flexible way.

As soon as an Imago invokes the move(1) built-in predicate, the destination server name is extracted from the stack and passed, along with the ICB and memory block, to the migration stack. After going through the upper layer of the stack, the resulting PDU is passed to the routing layer. An attempt to directly resolve the destination server name is done at first, and the retrieved IP address is used if the DNS query succeed. In case of failure, the host name component is stripped and replaced with the generic Imago gateway host name (`igw' for Imago Gateway). Then, the name resolution is performed again, using the newly created name. There after, each time a request to the resolver fails, one component of the original domain name is removed and the remaining part appended to the Imago Gateway generic name. The process keeps on retrying until either the resolver request succeed, or the original domain name becomes empty. If an IP address is found, the routing encapsulated PDU is forwarded down the stack with the resolved IP address to the connection layer. In the other case, an error is generated and reported up the stack. At the intermediate gateway, the routing layer analysis the PDU header of an incoming packet, to find out the final destination. According to this destination, the packet is either forwarded to the next host or up the stack for local reception.

The Imago migration protocol achieve flexible routing in a simple way while discharging the Imago of such burden. The Imago does not need to know about the network configuration. Also, the gateway machine is not required to implement the entire MLVM server but mainly the last two layers of the migration stack (routing and connection) for intermediate routing.

The connection layer links two peers together and is in charge of delivering the data to the other side. Control PDU been usually much smaller than regular (or data) PDU’s, they often get transmitted using the UDP connectionless protocol. Data PDU, however, been much bigger and usual too big for UDP to handle, are always carried over a TCP connection. To improve efficiency, opened connections are cached for a predefined amount of time and can be reused without having to go through the connection establishment phase all over again.

4. CONCLUSION

In this paper, we discussed the design of the Imago migration protocol focusing on the aspect of network transport. The major feature of the migration protocol is that it does not distinguish messengers from worker and migrate both in the same uniform way - everything is an Imago. As a result, the migration support required for the IMAGO project could be achieve in a very generic way. Also, the thread pool technology, used for the migration protocol, has been easily ported and successfully adapted to the MLVM server. The Imago Project has been implemented and are under benchmark testing. Finally, we would like to express our appreciation to the Natural Science and Engineering Council of Canada for supporting this research.

5. REFERENCES


