An Alternative Framework for Intelligent Mobile Agents

Xining Li
Department of Computer Science
Lakehead University
Thunder Bay, Canada, P7B 5E1

Abstract The primary characteristic of a Mobile Agents system is the ability to move agent threads over the network. Support for agent mobility is a fundamental requirement in the design of a virtual machine which executes agent code. This paper will discuss some design issues in the development of MLVM - a Multithreaded Logic Virtual Machine for Intelligent Mobile Agents based on an extended Prolog. I will focus on the memory management consideration because it plays a determining role not only in shaping multithreading strategy, but also in establishing computation mobile policy. The objective of this research is to present an alternative model in the design space of Intelligent Mobile Agents.

Keywords: Intelligent Mobile Agents, Virtual Machine, Multithreads, Code Migration

1 Introduction

Mobile Agents are mainly intended to be used for network computing - applications distributed over large scale computer networks. The motto of Mobile Agents is: move the computations to the data rather than the data to the computations. In general, a mobile agent is a self-contained process that can autonomously migrate from host to host in order to perform its task on behalf of a (human) user. Numerous Mobile Agents systems have been implemented or are currently under development. System-level issues and language-level requirements that arise in the design of Mobile Agents systems are well discussed in [1].

Most of the Mobile Agents systems are based on scripting or interpreted programming languages that offer portable virtual machines for executing agent code, as well as a controlled execution environment featuring a security mechanism that restricts access to the host’s private resources. Some Mobile Agents systems are based on Java [2, 3, 4, 5, 6], and some are based on other object oriented programming languages or scripting languages [7, 8, 9]. As the primary identifying characteristic of a mobile agent is its ability to migrate from host to host, support for agent mobility is a fundamental requirement of a Mobile Agents system. An agent is normally composed of three parts: code, execution thread (stack), and data (heap). All these parts move with the agent whenever it moves. However, most of the Mobile Agents systems (especially those built on top of Java) only support weak migration - an agent moves with its code and data without its stack of the execution thread. Thus, the agent has to direct the control flow appropriately when its state is restored at the destination. For example, a Java-based agent captures/restores its execution state through the Java’s serialising/de-serialising feature which provides a means for translating a graph of objects into a byte-stream and thus achieves migration at a coarse granularity. This implies that an agent restarts execution from the beginning each time it moves to another host. As a result, the agent has to include some tracing code in order to find its continuation point upon each migration.
Another mobile agent framework which embeds a logic programming component is pioneered by Distributed Oz\cite{11} - a multiparadigm language (functional, logic, object-oriented, and constraint), and Jinni\cite{10} - a lightweight, multithreaded, Prolog-based language (supporting mobile agents through a combination of Java and Prolog components). Distributed Oz does not support thread-level mobility, instead, it provides protocols to implement mobility control for objects. In a user program, the mobility of an object must be well defined under the illusion of a single network-wide address space for all entities (include threads, objects and procedures). Jinni implements computation mobility by capturing continuations (describing future computations to be performed at a given point) at the thread-level. A live thread will migrate from Jinni to a faster remote BinProlog engine, do some CPU intensive work and then come back with the results.

This paper will discuss the design of a Multithreaded Logic Virtual Machine (MLVM) for Intelligent Mobile Agents based on an extended Prolog. First, I give the overview of MLVM. Section 2 will discuss the considerations of multithreading. I will focus on the memory management issues in section 3 because it plays a determining role not only in shaping multithreading strategy, but also in establishing computation mobile policy.

2 An Overview of MLVM

The MLVM design is based on the research of a sequential Logic Virtual Machine (LVM) for Prolog\cite{12}. The LVM bears a strong resemblance to the well known WAM\cite{13, 14}. However, it is simpler with a small, clean instruction set. Most importantly, it adopts a merged heap/stack for all dynamic memory allocations and embeds an efficient garbage collection algorithm, the Chronological Garbage Collection (CGC), to reclaim useless memory space. CGC exploits a new policy in the spectrum of generational garbage collection. An experimental LVM emulator has been implemented. Experimental results show that this approach has low runtime overhead, good virtual memory and cache performance, and very short, evenly distributed pause times during garbage collection. Some benchmarks show that the improved program's cache performance makes the LVM emulator competitive with existing commercial systems in speed.

The goal of MLVM is to present a logic-based framework in the design space of Intelligent Mobile Agents. To achieve this, we need to extended the LVM to cope with new issues, such as explicit concurrency, code autonomy, communication/synchronization and computation mobility. A long term goal is to complete an Intelligent Mobile Agents System which will be an infrastructure for a wide range of distributed applications, to examine the possibility of integrating the MLVM with existing Mobile Agents systems, and to explore the potential application areas, such as distributed data mining, electronic commerce, network information retrieval, mobile monitoring and notification, etc.

In designing the MLVM, some practical issues, such as multithreading, garbage collection, code migration, communication primitives, etc, have been specified, and some issues, such as security, services, etc, will be investigated further.

- Multithreading and thread-level mobility. Allowing multiple threads of control in a process introduces a second level of concurrency in a system. In our design, each MLVM-based agent server behaves like a virtual machine that supports concurrent execution of agents. An agent is represented by a thread which is a lightweight process that runs strictly sequentially. Memory management considerations play a determining role not only in shaping multithreading strategy, but also in establishing computation mobile policy. The merged stack/heap architecture of the MLVM makes it much easier and simpler to control autonomous agent threads and implement agent migration by capturing execution state at thread-level. Thus, the destination host can reanimate the...
thread at precisely the point where migration was initiated, which can be useful for automatic load-balancing or for fault-tolerant programs.

- Naming. There are two layers of naming services: the server layer and the application layer. To contact a server, the server must be identifiable. Generally speaking, three possible ways could be considered: 1) identifying by symbolic name of a server; 2) identifying by either URL or physical address; or 3) identifying by service that the servers provide. Although we have not decided yet which scheme to use for defining or looking for servers, we intend to accept the commonly used method such as those adopted for web servers, i.e., identifying servers by URL’s (the Uniform Resource Locator). In addition, we intend to use the method proposed by the Aglet project, namely, this URL should provide the host and domain names plus the protocol (Agent Transfer Protocol) to be used for transferring agents over the network. On the other hand, each agent must have a unique name so that its owner and other agents can communicate with it over the networks. Some systems, such as Distributed Oz and Voyager, adopt location-transparent names at the application level. Systems such as Aglets and Agent Tcl, assign location-dependent names. For example, an aglet is associated with a unique identifier so that every aglet in the network can be uniquely addressed by combining its identifier with its context URL. I believe, however, that identifying an agent by the combination of its identifier and its location does not suit well with the mobility of agents. As an agent may move any time to an arbitrary host server, making the location uncertain. Therefore, I intend to use the location-transparent scheme because it not only avoids the conflict between fixed locations and moving agents, but also allows easier agent programming and portability of agent applications.

- Programming primitives. In general, a Mobile Agents system provides primitives for agent management (creation, dispatching, migration), agent communication/synchronization, agent monitoring (query, recall, termination), etc. Like other logic programming systems, the MLVM Application Programming Interface is presented as a set of builtin predicates. This set consists of builtin predicates common to most Prolog-based systems and new builtin predicates extended for mobile agent applications. In most concurrent programming languages, communication primitives take the form of message passing, remote procedure calls, or blackboard-based. For example, SICStus MT[15] uses the asynchronous message-passing mechanism whereas Bin-Prolog[10] adopts the blackboard-based model. On the other hand, agents communication in MLVM are conducted through messengers - special agents dedicated to deliver messages. The reason of introducing such special purpose agents is that the peer to peer communication mechanism in traditional concurrent (distributed) programming languages does not fit the paradigm of mobile agents. This is because mobile agents are autonomous - they may decide where to go based on their own will or the information they have gathered. Most mobile agents systems either do not provide the ability of automatically tracing moving agents, or try to avoid discussing this issue. For example, Aglet API does not support agent tracking, instead, it leaves this problem to applications. On the other hand, the MLVM system allows messengers to track down receiving agents and therefore achieves reliable message delivery.

- Security issues. Mobile agent security is an open research area. Several security related problems, such as agent privacy and integrity, authentication, authorization, and access control, are to be investigated.

- Services. An agent server must provide services to its hosted agents. One scheme is to offer a generic service by exporting a parametric interface which facilitates a large spectrum of requirements from all the potential agents. Another approach is to provide a set of primitives which can be invoked by agents for a desired service. This issue will be studied further.

Three projects are currently under develop-
The MLVM originates from the LVM which, on the outside, is just a simplified WAM. Hence, it is expected that most WAM-based compilation techniques and optimizations can be directly adopted. In the development of the MLVM compiler, however, two garbage collection related issues might require further investigation: how to minimize the initial root set and how to prevent CGC from collecting data which will never be used in the future execution. We are also going to investigate the possibility of Just In Time compiler approach.

The MLVM emulator will be designed to test Mobile Agents programs. The intended system consists of a number of networked Unix(linux)-based servers running MLVM’s - each MLVM provides a protected execution environment to host mobile agents. Integrating security into the basic agent infrastructure will be considered at this stage. The system will incorporate robustness and fault-tolerance mechanisms. The system will also be used in quantifying the performance trade-offs of the mobile agent paradigm.

The cache performance emulator will be used to verify and validate the MLVM design and our memory management policy. It will gather information and statistics of instruction frequency, memory usage, and cache performance. To cover typical cache implementations, a large number of cache parameters should be considered. This emulator will do the trace-driven simulation for directly-mapped and set associative caches with different write miss policies.

3 Multithreading and thread-level mobility

The LVM is a sequential logic virtual machine for Prolog-like languages. To accommodate the LVM to the mobile agents paradigm, we must introduce concurrency into the LVM model. Concurrent processes are the basic computational units in most operating systems. They are asynchronous and each has its own logical address space. The concept of process blends the concepts of resource ownership and concurrent execution together. Modern systems tries to distinguish these potentially independent concepts, and thus has led to the development of a computational concept known as the thread.

Multithreading 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 that supports concurrent execution of threads, forming the second level concurrency within the same address space.

A MLVM-based agent server is a multithreading process. Each MLVM-based server resides at a host machine intending to host mobile agents and provides a protected agent execution environment. An agent is represented by a thread which is a lightweight process which runs strictly sequentially.

In designing a MLVM emulator, it is a critical factor to determine the strategy of multithreading implementation. Generally speaking, there are two possible schemes:

Kernel level multithreading: kernel level multithreading has the advantage that blocking and scheduling of threads are treated with greater flexibility and efficiency. For example, threads can be pre-empted 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. Solaris is an instance which introduces the kernel level multithreading called a Light Weight Process (LWP). LWP’s are recognized by the Solaris kernel as the basic unit that can be scheduled. The POSIX application programming interface (API) is another example which provides preemptive operating system level threads. However, the shortcomings are that the lighter context switching overhead of threads is lost, and this paradigm does not fit with the idea that mobile agent servers should be able
to run on different operating systems.

**User level multithreading:** thread packages implemented as a software layer in the user space are 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 run time procedure 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.

I intend to explore user level multithreading because switching threads is at least an order of magnitude faster than trapping to the kernel. However, in the absence of clock interrupts, user level multithreading has the difficulty of distributing cpu time among runnable threads. In other words, if a thread starts running, no other threads in the MLVM process will ever run unless the running thread voluntarily gives up the cpu. There are four possible solutions to solve this problem:

- MLVM requests a clock signal and captures this signal to switch threads.
- MLVM counts procedure calls of each running thread and uses a certain call limit to switch threads.
- MLVM preempts cpu from a thread at the point of garbage collection.
- Or let a thread run and believe that most threads will block often.

At this moment, we have not decided which solution or a combination of solutions to use. The above possible solutions will be investigated further and be experimented in the MLVM implementation.

### 4 Memory Management

Memory management is a central issue in the design of the MLVM. 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.

The MLVM allocates two memory blocks for each thread: a program block and an environment block. The memory regions for these blocks does not need to be contiguous. The MLVM does not provide the programmer or the user control over the initial sizes of these blocks, instead, they are dynamically expanded or possibly contracted as required by the computation. During execution, a thread might need more program space to cope with dynamically generated clauses or symbolic constants, or more environment space to hold dynamic data. Further, for a Mobile Agents application, a thread might move to a remote host and continue execution from the moving point. Thus, both blocks should be able to be expanded (or possibly contracted) during run time, or to be moved together to another (remote) virtual machine and then be restored at different block spaces. The expandability and mobility of a thread require efficient algorithms to relocate addresses from old to new.

The program block of a thread contains program code and a symbol table. The MLVM separates the program block from the environment block because it exhibits different expansion and management patterns. First, this block is not subject to garbage collection. Secondly, it is unlikely to be expanded as often as the environment block. The code portion might be expanded only if the directive *dynamic* occurs in the program text. A symbol hash table is used for symbolic *resolution* - a unique entity (offset) produced by the initial resolution of an atom entry will be shared by
all the later resolutions of the same atom entry. The symbol table and its associated hash table might be expanded if new atoms are created during execution (through input or atom constructing predicates). The basic rule is that all existing atoms will remain in their original offsets to the symbol table base after expansion. However, they must be rehashed with respect to an expanded hash table.

The management algorithms and strategies associated with the environment block is critical to the system performance. The environment block is controlled by two crucial pointers: the current trail top and the current stack top, where the trail is used to save pointers for backtracking and garbage collection, and the stack contains all kinds of data during execution. The environment block is exhausted if these two pointers threaten to cross. Clearly, monitoring the growth of both pointers must be very expensive. We only check the threaten to cross condition at a few points - stack allocation instructions. When an stack allocation instruction is executed, the MLVM emulator will first check if a round of garbage collection can be invoked and thus come up with the following cases:

1. no GC and two pointers are not threaten to cross: allocate the frame and continue.
2. no GC, but two pointers are threaten to cross: expand the environment block.
3. GC and have enough copying area: do garbage collection.
4. GC but no enough copying area: expand the environment block and do GC.

The environment block needs to be expanded in cases 2 and 4. 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 [16] - a Prolog emulator, and the latter is used in SableVM [17] - 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 environment expansion. In brief, a small environment block is allocated through malloc to a new born thread, the block might be expanded by realloc dynamically to meet the needs of the thread, and finally, the block will be returned to the system pool at the thread death.

If there is a reasonable amount of accumulated garbage being detected, an inline CGC function will be invoked. CGC is mainly a generational copying collector[12]. However, it has new features different from the traditional generational collectors[18]. First, 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 (in most cases) incrementally with a trivial pause time. Third, most generational collectors promote survivors from the youngest generation into old generations. 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. Contracting only applies to the environment block. A possible point to reduce the size of the environment block is a big chunk of memory being reclaimed after a round of garbage collection.

5 Conclusion

In this paper, we discussed the preliminary design of the MLVM - a logic-based framework for an Intelligent Mobile Agents server - focusing on the aspects of multithreading and
memory management. The major feature of the MLVM is its novel memory management model - to deploy an automatically garbage-collected, merged stack/heap architecture to accommodate dynamic memory allocation and the thread level mobility.

Although this study concentrates on the design and implementation of an Intelligent Mobile Agents system based on logic programming, the results will be also useful in related disciplines of network/mobile computing and functional/logic programming community.

Finally, I would like to express my appreciation to the Natural Science and Engineering Council of Canada for supporting this research.

References


