CSRP 499
Implementing a Java Virtual Machine for Network Simulation

Rory Graves and Ian Wakeman *†

October 1998

Abstract

In this paper we discuss the development of a controllable virtual machine for use within network simulations. We describe general problems in integrating virtual machines with simulation environments, illustrated through our experiences in building a Java Virtual Machine.

1 Introduction

A burgeoning area of network research is in Active Networks [11]. In active networks, packets contain both date and *code*, which can be executed on intermediate switches. In order to evaluate the effectiveness of this decision, simulations of network algorithms and protocols need to be able to measure the effect of processing load on switches as well as on the more traditional resources of buffers and bandwidth. We have thus designed a simulation environment building upon the *ns* network simulator [8], in which we integrate virtual machines on which to run the packetised code.

Most Active Network research uses the Java Virtual Machine as the computational substrate. We therefore decided upon a Java Virtual Machine (JVM) as our initial target machine.

Many JVMs already exist (see section 2.1) but we have found none that are suitable for simulation work. This is because JVMs are written for speed, not clarity or controllability. To use a JVM in simulation requires the ability to "step" the JVM and keep it synchronized with "simulation time", and to allow arbitrary instrumentation of code within the JVM. We also need to experiment with various choices within the design of the JVM, such as the scheduler, the class loader and the garbage collection system, requiring the JVM to be amenable to accepting new implementations of these services.

*This work is sponsored by EPSRC grant GR/L06072, BT Laboratories and Hewlett-Packard.

†School of Cognitive and Computing Sciences, University of Sussex, Falmer, Brighton, BN1 9QH, UK

In the rest of this paper, we first describe our requirements in detail, and describe how other implementations of JVMs are inadequate. We then outline the major problems in building our JVM, and how we have overcome them, concluding with a discussion of how the JVM is used within our simulation environment.

2 Virtual Machine Requirements

A machine that is steppable The VM should be controllable down to the level of individual instruction execution. We must define an indivisible unit of time (a step).

useful information so were ignored. The only exception to this was a vaguely

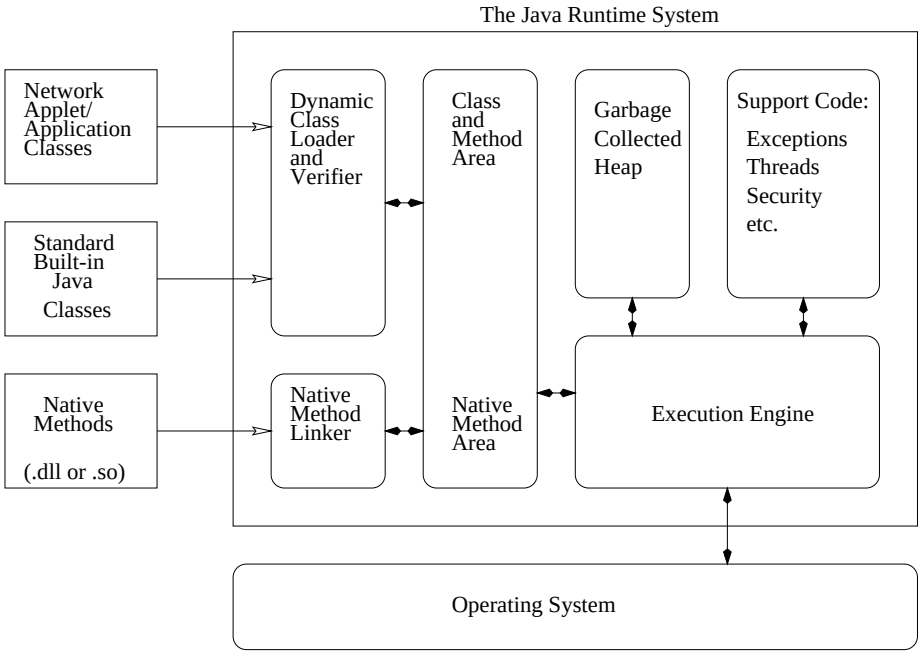
The Java Runtime System
Network Applet/ Application Classes
Standard Built-in Java Classes
Native Methods
(.dll or .so)
Dynamic Class Loader and Verifier
Native Method Linker
Class and Method Area
Native Method Area
Garbage Collected Heap
Support Code:
Exceptions
Threads
Security
etc.
Execution Engine
Operating System

Figure 1:

6

Java provides a very simple mechanism for programmers to synchronize threads. Each object in Java has an associated lock. A thread may attempt to gain the lock on an object. If the lock is already held by another thread the thread is forced to wait. Synchronization can occur in three places. Firstly there are two JVM instruction `MONITOR_ENTER` and `MONITOR_EXIT` (generated by synchronized blocks). Secondly synchronized methods implicitly call these instructions when the method is entered and exited. Lastly synchronization also occurs with class loading. If a thread begins loading an initialising a class when another thread is already doing so it is forced to wait. The second thread is restarted and given the requested class when the first thread has finished initialising it.

Within this JVM there is no way for a separate clock to interrupt a thread running. Instead the JVM asks the scheduler to pass a certain amount of simulation time. This translates into the number of steps that can occur in a given period of simulation time (see 7.1). A thread will run for a given number of steps (a certain time period) just as a normal system. The scheduler is currently a fairly simple prioritised round-robin scheduler. Each thread is allowed to run for a defined number of steps before control is handed to another thread. This mimics reality fairly well.

The scheduler is also responsible for dealing with locks and synchronization issues. The lock manager and scheduler interact but are designed in a modular fashion. Each makes requests to the other as needed to add or remove threads from the running queues.

Little is specified about what the normal scheduling method is. It is declared "implementation dependent" and ignored. This gives the implementor freedom and flexibility so that an appropriate algorithm for their needs can be used. As stated the scheduler is currently a prioritised round robin scheduler but it is interchangeable with any other required scheduling module. There would be no problem changing the current implementation for a different scheduling algorithm.

7.1 Ste ing

We required a large degree of flexibility in step control and granularity. We first considered using a finite state machine over the instructions but the ensuing complexity made this infeasible. The alternative design used interlocking threads. To run an instruction on the JVM the control thread wakes the VM thread and goes to sleep on a common lock. The VM thread executes a single step defined by us, wakes the control thread and then goes to sleep on the com-

class

Th.7(to)-55412Td(thet)]TJ-187.4412Td-30.712.5602Td[(W)2999.9638.64020Td(fo20Td[(This)-14000(tr9.119

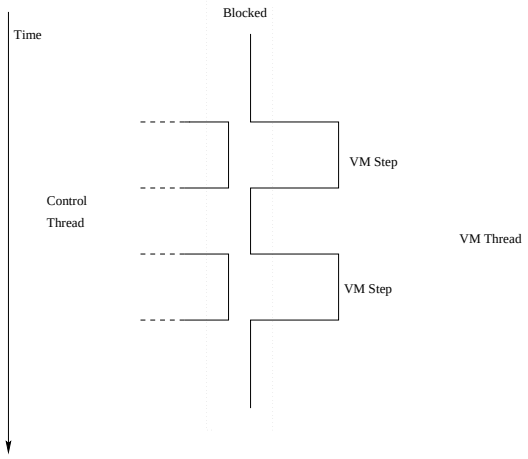

Figure 2: Thread interlocking for stepping

8 The Heap

True simulation of the heap has been ignored in this implementation for several reasons. Java is strongly typed which means that it is hard to use a block of memory to represent a flat memory area. The only obvious way visible to achieve this is to use a byte or int array as a heap and do data conversions to transfer to and from the represented types as needed.

Reference Counting For reference counting each object has a counter associated with it. This counter represents the number of different references to the object. Whenever a reference within the running program is modified the counters for the affected objects are modified. For example, if a reference is duplicated on the stack the reference counter would be incremented. When a reference is written over or destroyed the counter is decremented. When the counter reaches zero the object can no longer be referenced from the running program and can thus be garbage collected.

The only place that this algorithm falls over is when we have a cyclic data structure. If we create two objects A and B which refer to each other. When the running program can no longer reference each other the reference count is not zero. This is because A still holds a reference to B and vice-versa. Reference counting must be supplemented with another algorithm to guarantee that all the garbage is collected.

Mark and Sweep The concept behind mark and sweep garbage collection is simple. Take all of the reference on the stack and the local variable area. We will call these the root set. From each root object we take all of the references it contains and visit these recursively. As we travel through each object we mark it as 'visited'. Any object that is not marked as 'visited' when we finish is not reachable from the running program and can thus be garbage collected.

9.1 Problems

The two main versions of simple garbage collection detailed above have some problems. Reference counting, as described, cannot deal with circular references and must be supplemented with some other garbage collection method. Simple mark and sweep is fine as long as nothing changes during the collection. This means that the program must stop for garbage collection to take place. In many cases this is unacceptable.

9.2 Incremental Garbage collection

Incremental garbage collection allows collection during program execution. i.e rather than collecting when there is no more memory, collection can be done 'on the fly'. The most common technique for this is tri-colour marking as described below.

9.2.1 Tri-Colour Marking

Tri-colour marking works in a very similar way to a normal search. Every object on the heap has one of three colours:

White Objects that have not yet been visited.

Grey Objects that have been visited, but whose children have not all been visited.

Black Objects that have been visited along with all their children.

All objects start white, as they are visited they are changed from white to grey. When all of an objects have been visited they are coloured black. When there are no grey objects left all of the white objects left are unreachable and therefore may be garbage collected.

For this to work we must maintain two invariants:

1. No black object points to a white object
2. All grey objects are in a list of objects yet to be explored

The running program may still be creating and modifying objects. To keep the invariants valid we may recolor different objects so that the invariants hold. Different versions of the algorithm use different colouring mechanisms to do this.

9.3 Does no memory mean no memory?

When a garbage collecting memory management system reports that it is out of memory, it may not be accurate. If resource became free during the current pass the GC algorithm may not realise until the completion of its next GC cycle. This raises some interesting issues about when, and how much time should be spent garbage collecting. It may be better to allocate a percentage ofy1000.l5980Td[(It)-14000.os4n199Td(memorfn)Tj52.a27.119905237(v)1000.67(e)-14(completion)Tj500.im

13 Integration within the network Simulator

14 Conclusions

Determinism One of the nicest features of this JVM is that it is completely

Where Next? The implemention is a good clean implemention of a basic JVM. We would recommend taking another implementation and converting it to your needs, if possible, rather than starting from scratch. To truly finish this JVM there is much work that needs to be done but it has been taken to a point where it meets our needs for experimentation. This project will be publically released shortly and work will probably continue with the help of others to make it a really complete project.

[12] Unknown. An implementation of the java virtual machine. Master's thesis, USA, Unknown. Implementation in C without GC.

[13] Bill Verner. *Inside the Java Virtual Machine*. McGraw-Hill, 1998. Excellent introductory book and the example applets are great.

[14] Paul R. Wilson. Uniprocessor garbage collection techniques. In *1992 International Workshop on Memory Management*, St. Malo, France, September 1992. Springer-Verlag Lecture Notes in Computer Science.