Transparent Remote Execution of Java Threads

by
Kenneth Blair Kent
B.Sc. (*hons*), Memorial University of Newfoundland, 1996

A Thesis Submitted in Partial Fulfillment of the
Requirements for the Degree of

MASTER OF SCIENCE

in the Department of Computer Science

We accept this thesis as conforming
to the required standard

_____
Dr. J. Muzio, Co-supervisor (Department of Computer Science)

_____
Dr. G. Shoja, Co-supervisor (Department of Computer Science)

_____
Dr. N. Dimopoulos, Outside Member (Department of Electrical and Computer Engineering)

_____
Dr. S. Atkins, External Examiner (Simon Fraser University, School of Computing Science)

Supervisors: Dr. J. Muzio and Dr. G. Shoja

ABSTRACT

One built-in feature of the Java programming language is the support for threads. With Java, programmers can easily write multi-threaded programs. Previous issues like portability, availability, and maintainability of thread packages are no longer a concern. Unfortunately, Java programs experience a decrease in speed of execution since Java is interpreted and not compiled. Multi-threaded Java programs are affected even more dramatically from the overhead involved in sharing a single CPU resource.

Previous thread packages for languages like C and C++ provided support for threads to migrate to different hosts and execute. This allowed programmers to take advantage of idle hosts, which increased the performance of their programs. This thesis discusses modifications to the Java virtual machine so that it can, transparently to the user, migrate threads to remote hosts. The migrated threads execute on the remote hosts and report back the state of the thread class upon completion. Through migration, the threads take advantage of idle processors on remote hosts to decrease processing time.

In this thesis, we address solutions to the many problems raised by migrating threads, including security, exception handling, load balancing, fault tolerance, and I/O control will be discussed. These solutions do not require changes to the Java Application Programming Interface (API). Consequently, programs written in Java have no dependence on any one virtual machine, thereby maintaining Java's full 100% portability. Experimental results to show the true performance gains are also included and discussed.

Examiners

Dr. J. Muzio, Co-supervisor (Department of Computer Science)

Dr. G. Shoja, Co-supervisor (Department of Computer Science)

Dr. N. Dimopoulos, Outside Member (Department of Electrical and Computer Engineering)

Dr. S. Atkins, External Examiner (Simon Fraser University, School of Computing Science)

Table of Contents

List of Figures

List of Tables

Acknowledgements

Many people contributed to the completion of this work. Special thanks to my supervisors, to Jon for paying the way and appreciating my strange sense of humor, and to Ali for taking the time to put me through a mini-defense every time to backup each step along the way. Thanks Ali.

Thanks also goes to the VLSI group which suffered through many a talk, and Labatt's breweries for producing the fine lager Blue Star that kept my body going. Thanks to Gord D. Brown and Stephen Goglin for acting as sounding boards to suggestions, and Gord for proof-reading the final product.

Last but not least, Mudder and Fadder for phoning each and every week asking how the thesis is going and lighting the fire beneath me, and Elizabeth who spent many a night by herself as I "worked" late in the office.

for Elizabeth

CHAPTER 1

Introduction

Since its advent, Java has been presented as a general-purpose programming language. Unfortunately, there exist many processing-intensive applications for which Java has not been suitable due to its slow execution speed. Many software solutions have been proposed to increase Javas processing capabilities by distributing the computation across multiple hosts [6][14][21][22][24]. All of these software solutions work in different ways but all, in essence, deliver faster processing than the Java virtual machine without any distribution. There is therefore evidence, that Java's performance can be improved with distributed execution.

1.1 Problem

Unfortunately, the software solutions currently available all have non-trivial weaknesses. The one major pitfall of all current solutions is added complexity for the programmer. Often it is required that the programmer provide multiple applications that work together to accomplish the overall goal. This gives the programmer flexibility in the design of the solution, but forces them to consider low-level details such as communication between applications. The software support therefore hides some detail, but it still leaves a lot of the detail for the programmer to contend with.

In addition, some solutions tie the application to the environment in which it is developed, compromising the portability that Java provides. The programmer can implement robust systems, but must do so explicitly for each application, requiring careful design and a high level of understanding. Even with a robust design, the application is not guaranteed to provide the optimal configuration on the hosts. This is evident by the software solutions static decision(s) on where to off-load some of the execution.

1.2 Proposed Solution

The solution proposed in this thesis keeps the distribution hidden from the programmer and maintains portability. In essence, the idea is to “scale” the resources available to the virtual machine. The primary goal of scalability with respect to the Java virtual machine is to enlarge the number of virtual machines. To realize this, a new virtual machine is created. This new virtual machine starts new virtual machines on remote hosts as needed and interacts with them to execute the application. Working together these multiple virtual machines can interact presenting a unified interface and a level of transparency. This is performed through message passing between the hosts collaborating on the execution. This allows for increasing the processing power of the virtual machine while still delivering the same results. All of this can be accomplished and still maintain roughly the same amount of effort from the programmer in development. The programmer is still responsible for dividing the work into tasks, but the actual task off-loading is hidden and automated. This leaves only the question as to how the programmer divides the work into tasks.

Java supports threads as a well defined part of the standard Java application programming interface (API). Unfortunately, when a program is interpreted in the Java environment, the Java Virtual Machine, the threads operate using only the resources available within the local host. Often the local host contains only a single processing unit and, as a result, the threads become serialized. Hence, these threads can be used by the programmer to attain concurrency of execution, but not parallelism [4]. To allow for parallelism, multiple processing resources must be provided. With the well defined interface of threads, they are an ideal solution to allow the programmer to split their application into distinct tasks. The proposed idea is to transparently export these threads to other virtual machines for execution.

The major benefit of this solution is the transparency to the programmer. All of the details are hidden from the programmer with the exception of dividing the application into distinct tasks. This much work on the programmer’s part is unavoidable, as trying to parallelize serial applications automatically has never historically worked well. In addition, the code provided by the programmer is still capable of running on a stand-alone

Java virtual machine, but runs faster on our new virtual machine.

1.3 Thesis Overview

In chapter 2 we discuss the currently available software solutions to speed up Java execution. In chapter 3 our proposed solution is discussed, which is designed to overcome the shortfalls of other solutions. The requirements of the proposed solution are outlined and the architecture of the solution is presented. Chapter 4 is the main chapter, we describe the implementation of the new virtual machine. Solutions are discussed for problems such as security, load balancing, input/output handling, class loading, exceptions, and fault tolerance. In chapter 5 we deliver the results of using the new virtual machine. Benchmark tests of both N-queens and computing the Mandelbrot set are executed and compared with the Sun Java virtual machine. Finally, in chapter 6 we summarize the thesis and discuss future work.

CHAPTER 2

Background

With the popularity of Java and the tremendous concern for speed, there are solutions for increasing performance. The most obvious performance increase is to use a multi-processor system. Unfortunately this requires an expensive computer, and not a "typical" development system. Various software solutions are also available to address the problem of slow execution for Java applications. Each of these software solutions has benefits and pitfalls that affect their suitability for particular applications. In this chapter we discuss and evaluate some of the more commonly known software solutions. These include the Remote Method Invocation (RMI) package, various integrations of the Parallel Virtual Machine (PVM) into the Java model, and to some extent the Common Object Request Broker Architecture (CORBA) and its influence on Java. Finally, a discussion of some shortcomings that the Sumatra virtual machine is designed to overcome.

2.1 Remote Method Invocation (RMI)

The Java Remote Method Invocation idea was developed by Sun Microsystems shortly after the original release of the Java programming language [14][22][23]. The drive behind the design was providing support for execution on a remote server. Such capabilities were available in other languages already. For example, there exist Remote Procedure Calls (RPC) in C that allow the execution of a procedure at a remote server; CORBA provides similar support to the C++ language for invoking objects at remote servers. Support was needed for these facilities within Java, so the developers decided to incorporate it directly into the language specification.

In RPC, the communication interface between the client and server is abstracted to the level of a procedure call. This hides the details of the data packaging and communication on both sides of the client-server, leaving the programmer with the impression of

calling a local procedure. This design is sufficient for an imperative language such as C. However it does not translate well into object-oriented languages. This is a result of required communication between program-level objects residing in different address spaces. In order to better fit the semantics of object invocation, a local surrogate, or stub, object manages the invocation of the method on a remote object.

The object model of the language greatly influences the design of the stub object that resides at the client. The Java RMI package has been specifically designed with the Java object model in mind. This is in contrast to CORBA that was designed with the idea of allowing inter-operability between languages in addition to a heterogeneous environment, thus having to take a generalized object model for all languages. Since Java RMI is expected to work for only the Java language, the design can assume the homogeneous environment of the Java virtual machine, and follow the Java object model whenever possible [11][12][25].

The implementation of the Java RMI has several system goals for supporting distributed objects, these include:

- Seamless remote invocation on objects in different virtual machines.
- Designing the new distributed model in such a way that it is a natural part of Java and maintains the semantics of the Java language's object.
- Making any differences that exist between the language and distributed models clear, and not hidden in the implementation.
- Minimizing the complexity on the client and server sides. If the complexity of using RMI is high, then it may not be accepted.
- Preserving the security of the Java virtual machine.
- Simplicity and ease of use.

The overall idea of RMI is for the user to break the application into multiple parts. Each part can be placed at a remote host with a server running at the host receiving requests to execute the piece of code. The application will execute at a client and make requests to a server(s) to execute a given part(s) of the application. After the execution of the code at the server, the result is passed back to the client.

The Java RMI is a solution to executing objects methods at a remote host. There are several drawbacks to the solution. Possibly the biggest pitfall is the client having to make an explicit connection with the server to gain access to the object. The need for explicit connection places the burden of managing issues such as the execution environment on the programmer. In addition, the programmer has to know where the object exists, which may not fit naturally into the design of the application.

The structure that is formed by the relationship of clients and servers is not indicative of the execution environment. That is, the distribution of execution is static. If the server provides some special resource that is only available at a particular host, then this makes sense, but if the service provided requires no special resources, then it is an unjustified requirement. If a node has a heavy load, the application is affected by a slow execution. Even worse, if a node is unavailable, the program cannot execute the application without changes to both the application and the setup. This requires the programmer to change the application to reflect the new host which executes the server, and additionally to start the server on the new host. In order for the user to accomplish this, the user must have access to start the server process on the new host. Quite often in a client-server relation, access by the user to start a server on a remote host is not available. This can cause many headaches when one is not working within a dedicated cluster.

2.2 Java and PVM

Before the advent of Java, the programming language of choice was C. Programmers that used C, commonly used a software package called the Parallel Virtual Machine (PVM) when distributing tasks in applications [8]. PVM is a software package that permits a heterogeneous collection of serial, parallel, and vector computers connected by a network appear as one large computing resource. This gives the programmer support for manipulating tasks at a remote host, and communicating between the tasks. PVM was originally written for procedural languages such as C and Fortran; hence it does not provide a natural direct transition to an object-oriented language. Nevertheless, the software library is good and has value in undergoing such a transition.

For PVM to give the impression of providing a single large computing resource, each of the individual resources must be connected in some way. This is accomplished by having a daemon at each host. Having a daemon at the host makes it eligible to be added to the host pool. The daemon then helps to control the messages being passed among the tasks. Once the PVM system is in place, the PVM library provides numerous functions that a programmer can use in their application to achieve distributed execution. Specifically, functions exist for manipulating tasks, i.e. creating and terminating, and message passing, i.e. packing and unpacking data as well as sending and receiving. Thus, the programmer is responsible for explicitly using the functions within the application.

To give the reader a better feel for the use of PVM, one need go no further than the programming paradigm. The programmer writes one or more sequential programs that contain embedded calls to the PVM library. Each of these programs corresponds to a task implementing part of the application. It is possible for multiple tasks to be generalized into one program, but it has to run several times in order to account for all tasks. Once the programs are written, they are compiled for each architecture in the host pool that makes up the virtual machine. The appropriate object files are placed at a location accessible from the appropriate machines in the host pool. To begin the application, the user manually starts one copy of the application known as the master or initiating task. This task subsequently starts other PVM tasks within the virtual machine. Eventually there are a number of active tasks computing and communicating to solve the problem. Once the tasks are started they can interact through the explicit message-passing functions provided by PVM using their task id (TID) to identify each other. Finally, once all tasks are finished they and the master task disassociate themselves by exiting the PVM system.

PVM has potential for allowing programmers to tackle large problems by using common hardware in a fraction of the time previously required. As such, it was immediately seen as a tool to tackle some of the problems that Java suffers from, namely execution speed, along with some support for off-loading tasks as opposed to using the Java standard IO classes. Two different approaches have been taken to incorporate PVM into Java. These approaches are discussed in detail in the following two subsections.

2.2.1 JPVM

The most obvious solution for incorporating PVM into Java is to translate the PVM library into a set of classes. This is the idea of JPVM [6]. This approach gives programmers the ability to use the PVM library within the Java environment. As with PVM in C or Fortran, the programmer includes explicit calls to the methods provided by the PVM class. The programming paradigm for PVM has not changed. Programmers must still write programs that correspond to tasks. The only difference is that now all the tasks are written in Java instead of the traditional C or Fortran.

One of the key reasons for bringing PVM into Java as a set of classes is to give the programmer the true feel of the Java language. This approach allows the programmer to use the set of classes in the same manner as standard classes in the Java API. Additionally, since PVM is written in Java, it can be used on any system that has a Java interpreter. PVM is available on a broad range of architectures, but it is not as architecture independent as Java. However, there are drawbacks to implementing PVM in Java. By bringing the PVM library into Java as a set of classes, complications are added which do not allow tasks to be given in combinations of languages. Instead, all tasks must be described in Java. Depending on the application, this may be a major problem if a particular task must run at a higher speed.

2.2.2 jPVM

jPVM, formerly known as JavaPVM, takes an approach that allows the combination of languages in the description of tasks [24]. The Java programmer uses a Java class called jPVM, similar to JPVM, but the underlying methods of this class are not written in Java. Instead, the methods are written in C. This is accomplished by using the Native Methods interface in Java.

The advantage of this is to gain faster performance from the jPVM class. It is well known that the execution of a native method is on average faster than the same method in Java. In addition, using the standard PVM software instead of a ported version has its advantages in maintenance, access to current features, and overall better support.

Since the class uses the standard PVM system, there is greater support but the sacri-

fice for this is the dependence on the PVM software. The PVM system does not work on all architectures that Java can operate on. Since jPVM requires both, only architectures supported by both are capable of utilizing the software.

2.3 Java and CORBA

The Common Object Request Broker Architecture (CORBA) is a standard for allowing applications to communicate with one another regardless of location or designer [13][14]. An Object Request Broker (ORB) is the middleware that establishes the client-server relationship between objects. How the ORBs interact with one another is also specified within the CORBA standard. Using an ORB, a client can transparently invoke a method on a server object, which can be on the same machine or across a network. The ORB intercepts the call and is responsible for finding an object that can implement the request, pass it the parameters, invoke its method, and return the results. The client does not have to be aware of where the object is located.

Within the Java Developer's Kit (JDK) there is support for CORBA. This support is in the form of the Java Interface Definition Language (IDL) [21]. Java IDL is an ORB, and together with the idltojava compiler can be used to define, implement, and access CORBA objects from the Java programming language.

So what is the benefit of the Java IDL over RMI? The main benefit is resolving location. When using RMI, the Java application needs explicit directions as to where to find the remote object for manipulation. Using CORBA, the object is located by the ORB, and thus is more robust and can adapt to the execution environment. As well, using CORBA allows the programmer to mix languages, and not be constrained to use Java for all objects. This robustness comes at a cost. There is an additional level of complexity by having an ORB at each host to direct communication and locate the remote objects. With the availability of an ORB, information must be posted to the ORB to notify it as to where objects exist. This posting of information to the ORB by the server objects increases the amount of difficulty for the programmer.

2.4 Proposed Solution: Transparent Remote Execution of Java Threads

It is clear, based on the preceding sections, that there exist several issues that make the programmers job much more difficult than if the program were written for a single Java virtual machine with no remote execution. In the case of RMI, the programmer has the added tasks of writing a server to handle the object at the remote location, as well as the complexity of explicitly establishing a connection with the remote server to gain access to the remote object(s). Likewise, with Java IDL there is the added complexity of interfacing with the ORB, making the remote execution far from transparent to the programmer. This also holds true for both implementations of PVM integrated with Java. Each of these solutions, requires explicit knowledge of the remote execution and extra effort from the programmer to carry it through.

The second major problem is the relationship between the application and the execution environment. When using either RMI or CORBA, the state of the environment does not affect the application. The onus is on the programmer to account for unavailable hosts. As well, since objects reside at servers determined by the programmer, the remote objects do not adjust to load changes within the environment. That is, the execution is static regardless of the environment's state.

Unlike RMI and CORBA, the PVM implementations can adjust to the state of the environment. In the event that a host is disabled or overloaded, tasks are executed on other hosts to gain better execution times. Instead, PVM suffers from having a poor design fit with the Java language. PVM has no concept of an object and as such must look at remote execution in the perspective of *tasks*. Having the programmer provide each of the processes as independent tasks conflicts with the object model, making the programmer's job more complex.

These are a few of the major points that our new approach of remote transparent execution of Java threads is designed to solve. To attain concurrent execution, there must be a certain amount of complexity that the programmer has to contend with. Within the Java language, the idea of threads is well developed, and they are relatively easy to use.

The programmer can use many threads within the same program and is free to use all the design and implementation features of the language. This eliminates the need for maintaining multiple programs, one for each task. Instead, the programmer writes the application just as if it were for a single virtual machine. The difficulties associated with these methods are addressed by our proposed solution: transparent remote execution of Java threads. The threads concept is well developed and fully supported in Java. There are no explicit calls to communicate between objects on different hosts. All of this communication and extra programming is not required and is handled by the virtual machine, thus making the programmer's job as simple as possible. This is the core contribution of this thesis, to provide the off-loading of thread objects to remote virtual machines for the purpose of gaining execution speed.

Java uses the idea of a thread as the basis of execution within its virtual machine. There are two basic thread types supported by the virtual machine, *Green* and *Native.* The green threads are simulated threads that have no dependencies on the host architecture. As such, these threads work in the same manner whether executing on a powerful Sun Sparc workstation, or an average personal computer. This only supports the idea that threads are the basis of remote execution. These Green threads provide the potential for moving threads across virtual machines just as if one were moving any other piece of data. Within Java, threads are themselves represented as objects with special capabilities, that is the ability to execute. In the distributed threads model, when a thread is requested to start execution, transparent to the user the thread will migrate to a remote host and begin execution. Upon completion of the execution, the finished thread object will migrate back to the client, again unknown to the user. This migration is totally hidden from the user and occurs only in the event that CPU resources can be gained by the migration. If resources are low at remote hosts, the execution is performed locally. Native threads are perceived as better since they are actual operating system threads, but using native threads ties the thread to the host architecture and imposes major constraints on moving the thread to another host.

Using Green threads allows for migrating threads to remote hosts, but does not provide support for taking advantage of multi-processing systems. For the threads to be exe-

cuted on multiple processors within the same host, the operating system must recognize the threads as having their own operating system execution environment. With Green threads, this is not the case as the threads all share the same single operating system execution environment. Hence, if multiple processors are available, only one processor is used. To use the multiple processors you must use Native threads, however this does not easily allow for migrating threads from host to host. With this in mind, Green threads is the thread of choice at the cost of efficiency.

Execution within the virtual machine is dynamic. Each execution of the program can result in a different distribution of the program across the hosts. In the event that a host is unavailable, or is heavily burdened by other processes, the virtual machine will account for this and off-load threads elsewhere. This dynamic layout also makes the program independent of the environment, adding to the overall portability.

2.5 Summary

In this chapter, the current software support for distributing Java threads is discussed. Each software solution is compared, and its advantages and disadvantages are exposed. This lead to the description of a new solution that is designed to overcome the downfalls of the current known support. The next chapter describes further the new approach to provide distributed support. The discussion includes the requirements of the distributed architecture and extra support needed.

CHAPTER 3

Architecture Overview

The architecture of the Sumatra virtual machine is very well designed and provides for an obvious distributed layout. From the original design, the requirements needed to scale the virtual machine can be seen, and from these requirements a new scalable architecture can be made.

3.1 Sun's Java Virtual Machine Architecture

Sun's Java virtual machine is written in C, however the design is very object based. All entities within the Java virtual machine are objects, both from the user's point of view and in actual implementation [7][11][12][25]. As such, for a Java thread object there exists a corresponding thread object in the implementation of the thread. This design provides a very clear-cut break-down on how each entity is defined and more importantly how each object interacts with the virtual machine.

Within the virtual machine there exist two flavors of threads, *user* threads and *daemon* threads as depicted in Figure 3.1. The daemon threads are system threads which control or maintain various properties of the system that are always present. This includes garbage collecting, time slicing, idling, and keeping track of time. These threads can be manipulated (to some extent) by the user, but they are always present within the virtual machine. User threads are threads of execution that the user specifies in their Java program. As the program is interpreted, the virtual machine will create, suspend, resume, halt, and destroy threads as the programmer indicates. These threads are totally application dependent, and all that can be said is that there is at least one user thread present at a given time, otherwise the virtual machine will halt.

Threads, whether user or daemon, are not very different within the virtual machine. The same thread structures are used by both kinds of threads, with the exception of a

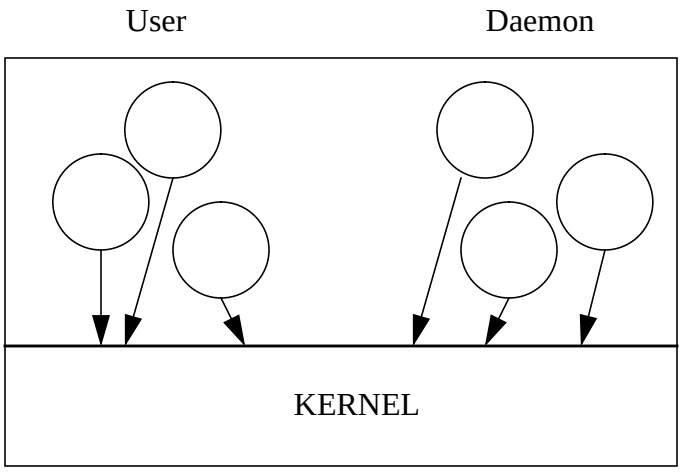

Figure 3.1 Java virtual machine overview

boolean flag set in daemon threads to indicate that it is not a thread belonging to the application. Both kinds of threads are in the same thread pool, both can make requests to the kernel of the virtual machine, and there is no differentiation between the types in context switching decisions. Context switching is on the basis of priority and runnability. The thread that has the highest priority and is runnable is the thread next chosen when a context switch occurs. Any differentiation between the threads is hidden in their priorities. For example, the clock thread within the virtual machine maintains time for processing time-outs, and time dependent operations. Thus, it has a greater priority than most other threads to guarantee that the time dependent requests are processed correctly. This is in contrast to the garbage collector thread that reclaims memory, and needs to execute periodically. Thus the garbage collector thread is given a low priority since its time deadlines are soft and infrequent. The programmer has the ability to disregard the priority scheme to an extent and make all user threads of the highest or lowest priority.

The virtual machine also has the context of a user level and a kernel level. The kernel level is very much like the kernel of an operating system. The kernel is protected and to gain entry, a call is made to ensure that no preempting of the thread within the kernel occurs. Preempting can occur through context switching or interrupts and with a thread in the kernel it can result in an unstable virtual machine. The user level of the virtual machine is where all threads reside. Threads temporarily enter the kernel solely for per-

forming special or system operations that require using special or system resources, then exit. When a thread enters the kernel, the remaining threads are effectively suspended until the thread exits the kernel. This is carried out by disabling all context switches until exiting the kernel.

3.2 Requirements of a Distributed Architecture

The Java virtual machine, due to its design, is very favorable towards scaling. This is evident in the design of the threads and their interactions. To obtain the scalability, a four step process is required.

- Decide to execute a thread remotely.
- Make a request to the remote host to execute the thread remotely.
- Start a virtual machine on the remote host to interpret the Java thread that is being migrated.
- Communicate with the remote virtual machine to relay the thread for execution.

A thread goes through a two step process before execution begins. The steps include *creation*, and then *runnable*. The creation step is the same, for threads, as well as for all objects. Since a thread is an object within the virtual machine, it must be created with the information concerning the thread. Only after a thread is created can it be added to the run queue for execution. However, it is not necessary that a thread be executed! It is probably not a good application design, but it is not required that all threads created must eventually be executed.

A special thread method, *run*, must be invoked to add the thread to the run queue. This is where the decision is made to execute the thread locally or to migrate the thread to a remote host for execution. This ensures that no thread is migrated unless it is guaranteed to enter the run queue; in addition it delays making the migration decision until the last moment. Waiting until the last possible moment ensures that the thread is migrated to the most suitable host at the time the thread begins execution.

Once the decision is made to migrate a thread to a remote host, the virtual machine must communicate with the remote host to determine if remote execution is appropriate. A communication primitive must be available within the virtual machine for this to take place [10][18][20]. In addition, a process is needed at the remote host to receive the request and reply. Hence, a communication primitive is needed in the remote process. If the thread request is accepted at the remote host, the process starts a virtual machine to execute the thread in question.

Once the remote virtual machine is started, the local and remote virtual machines must communicate to relay the information concerning the thread(s) that are to be executed remotely, and as well the results obtained from execution. This requires that the new remote virtual machine have access to a communication path to the master virtual machine. In addition, there must be some relay of information so that the new remote virtual machine can initiate contact with the original local virtual machine.

3.3 Our Proposed Distributed Architecture

From the above requirements, an understanding can be gained as to what is needed to make the virtual machine scalable. First, it requires a communication mechanism within the virtual machine that allows it to interact with other virtual machines. In addition, it requires some means by which this mechanism is monitored and controlled for incoming and outgoing communication. The second is a daemon process that runs on all hosts which the virtual machine can use to initiate contact with a remote host. These two topics are discussed in further detail below.

3.3.1 The Scale Thread (sthread)

When a thread is requested to start execution within the virtual machine, the decision is made as to whether to off-load the thread for remote execution or execute it locally. From the context of the thread’s parent, the thread starts execution immediately after it receives the request to execute. Unfortunately, it takes time to determine if the thread should be executed remotely or locally. If remotely, then the thread is accepted for execution at the remote host. To allow for this time delay there exists a daemon thread

within the virtual machine, namely *sthread*, which handles the off-loading duties and con-

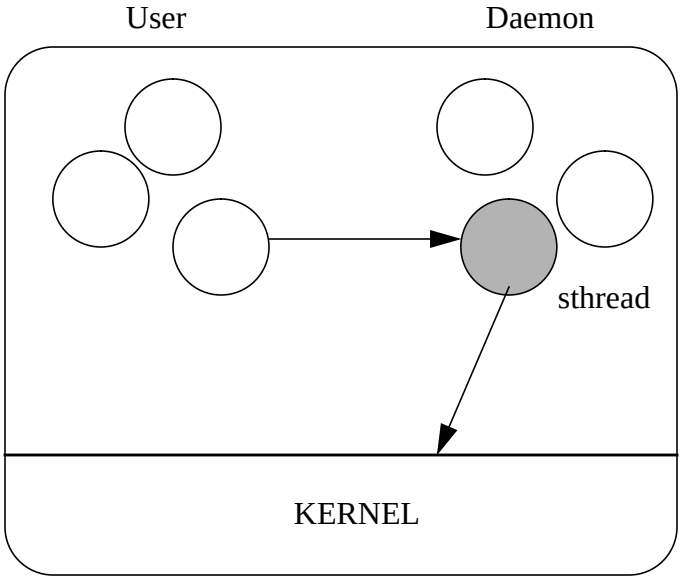

Figure 3.2 Illustration of sthread in virtual machine

firmation that the thread is executing remotely. Figure 3.2 shows the sthread (shaded) taking a user thread and off-loading it to another virtual machine using a communication mechanism in the kernel. Enabling the sthread to handle the off-loading allows the thread which requested the start of the new thread to continue execution without blocking and waiting for the thread to be off-loaded.

In the case when a thread is executed locally, the thread has already been instantiated and we need simply to create a new run-time data structure for the thread and add it to the runnable queue for a slice of execution time. In this case the thread is started in the same manner as the thread would be started for a stand-alone virtual machine. In the event that the thread is to be off-loaded, a host is chosen for executing the thread. The sthread then checks to see if a thread is already off-loaded to the remote host or if it is pending a response from the host to an earlier request. If this is the first off-loaded thread for a remote host, then a request for a thread is sent to the remote host's VMagent. The VMagent performs its duties, and, if the permissions are correct, a new remote virtual machine is created with the name of the thread object that is to be executed. This thread object is then started in the remote virtual machine, and added to the virtual machine's run queue for execution.

In the event that the virtual machine is pending a response to an earlier request for a virtual machine at the remote host, the request for a new thread is postponed until a response or time-out occurs on the pending previous request. If the thread is not the first thread that has been off-loaded to the remote host, the sthread looks up the address of the virtual machine that already exists on the host, and then directly forwards its request to the existing virtual machine. This prevents the overhead of duplicating virtual machines on the same host when multiple threads share the same virtual machine. In either case, the sthread sleeps, waiting for a response from the remote host indicating the acceptance or rejection of the request. After a maximum number of retries for off-loading the thread without receiving a response, the thread executes locally.

In the case that the virtual machine is already started on the remote host, and the sthread directly forwards its request to the virtual machine, it is the sthread within the remote virtual machine that handles the incoming request. If the request is accepted then the sthread that exists within the virtual machine is responsible for instantiating the thread, and starting the thread by creating the run-time data structures and adding the thread to the run queue. As each thread finishes execution, a message is sent by the sthread to the originating virtual machine indicating that the thread has terminated. In addition, a message is sent once all threads have completed to indicate the full completion of work by the virtual machine. The full thread off-loading algorithm is illustrated in Figure 3.3. As can be seen from the algorithm, the sthread is also responsible for relaying some of the load information for the host as used by the load balancing algorithm. This is discussed further in section 4.2.

3.3.2 The Virtual Machine Agent (VMagent)

To allow for easy remote process creation, the remote host runs a daemon process, VMagent, which can communicate with the local host. Once the remote machine is running, all intercommunication is handled within the virtual machine by the sthread as discussed in the previous subsection. To get into communication with the virtual machine (or start it) the local virtual machine can send a request to the remote VMagent for any vir-

```
while (1)
{
 if thread to off-load
 {
  determine optimum remote host;
  send request to remote host selected;
 }
 while (new message received)
 {
  switch (message type)
    LOAD: Record load of the replying message;
    ACCEPT: Replying VM accepted our thread;
    REJECT: Replying VM rejected thread;
          start thread locally;
    DEAD: Replying VM finished all off-loaded
       threads;
 }

 if not timed out
  re-send request;
 else
  start the thread locally;
}
```

Figure 3.3 The thread off-loading algorithm.

tual machine process. Once the VMagent receives the request it can easily create the processes and act as a mediator in establishing communication between the two virtual machines as depicted in Figure 3.4.

The VMagent process has many responsibilities in addition to starting the new virtual machine. These responsibilities include:

- Ensuring proper access to the computer. The system administrator of a computer must have the capability to restrict access.
- Communicating the information that allows the new process to communicate with the requesting virtual machine.

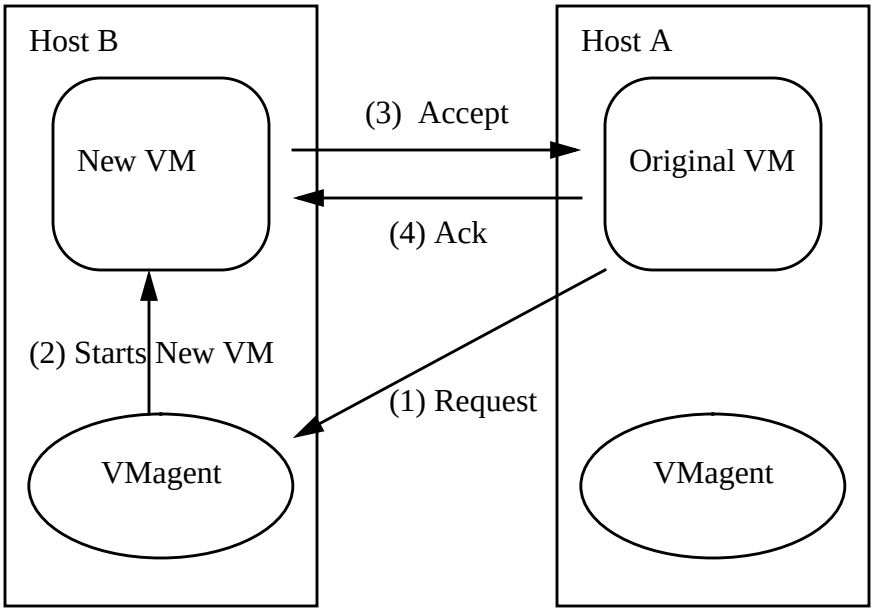

Figure 3.4 Creation cycle of a new remote virtual machine.

- Maintaining a log of the requests that have been serviced including the requesting source, time, communication information, and acceptance or denial of the request.
- Servicing multiple requests from various virtual machines simultaneously.

To initiate a new virtual machine, a message is sent to a dedicated port on the remote processor where the new virtual machine is desired. The daemon process, VMagent, executes in the background listening for any incoming requests, while providing support for handling multiple requests simultaneously. Upon receiving a request, the VMagent decides whether the request is to be accepted or declined. The request for a virtual machine is tagged with the identity of the user making the request. Then the VMagent proceeds to check its permission file to determine if the request should be accepted or declined. In the event that the request is declined, a rejection message is sent to the requesting virtual machine. In the event that the request is granted, a new virtual machine is invoked with the arguments desired from the request in addition to the special arguments “-user”, “-ip”, “-p”, “-mip”, and “-mp”. These arguments identify the user, internet address and port number of the initiating virtual machine, as well as the root virtual

machine's internet address and port. The request also contains the information required to instantiate the thread object to be executed. The slave virtual machine is now responsible for replying to the initiating virtual machine and performing any tasks communicated.

Multiple simultaneous requests are handled by the VMagent so that each request from a different application gets its own virtual machine. If there are 5 requests from 5 different applications, then there are 5 different independent virtual machines running at the remote host. There are two key reasons for this decision. The first is the settings of the virtual machine. Upon starting the virtual machine, several options are available that drastically affect the execution of the machine, so drastic that changing the options may cause the application to produce errors. For example, turning off garbage collection may cause one program to run out of memory. This also extends to environment interactions such as thread priorities, classpaths for resolving class definitions, and so on. The second is the issue of security. Within the virtual machine, it is normal for threads to interoperate. Attempting to control the threads to allow for some interoperation while not allowing for some other, is rather difficult. The security currently in the virtual machine used by security managers is aimed towards protecting the host from illegal actions, not towards protecting Java applications from one another. It is seen that having different virtual machines is the safest and simplest solution.

It is also worthy of note that the settings for the virtual machine are also passed along to the remote virtual machine. Some of these settings include the maximum and initial stack and heap sizes, and turning off the asynchronous garbage collection. Having these settings passed along to remote virtual machines allows remote threads to execute in the same environment as if they were executed locally. As for the operating system settings, the remote virtual machine is given the Unix process ownership of "nobody" since the initiator may or may not have an account on the remote machine. This regulates the actions of a Java program to prevent malicious actions. These include file manipulation and any other actions that have lasting or global effects on the system.

3.3.3 Thread Topology

The specification for the Sumatra virtual machine does not limit the number of threads that can exist on a virtual machine. Following this trend, neither the number of threads that can be distributed for execution, nor the topology of the distribution is restricted. It is possible for a virtual machine to remotely distribute threads of a thread process that was itself received as a result of load distribution. For instance, a thread executing on host A can create a new thread and off-load the thread to host B. It is quite possible that any thread created by the thread at B can be off-loaded to a new host C, or even back to the original host A. How distributed the threads become is solely based on the workloads of computers and not on implementation restrictions.

The distribution of threads is unrestricted and allows for a very interesting topology. With the possibility that threads at lower levels of the hierarchy can be created as "parent" hosts, the thread hierarchy is more representative of a web as opposed to a tree hierarchy. This reflects the freedom of threads to migrate to a near optimal host at the time of thread creation. The scalable virtual machine allows for this hierarchy and works with it, as opposed to other thread topologies that enforce restrictions upon the migration process.

3.3.4 Heterogeneity

When discussing distributed execution, the topic of heterogeneity is always a key issue in determining the success of distributed execution. Using the green thread model within the virtual machine where all threads are simulated in a neutral architecture, the application execution is independent of the host architecture. Likewise, the threads are independent of the operating system. This allows for multiple combinations of host architectures and operating systems to exist within the cluster and participate in the overall execution.

The only enforcement on the host platform is that there is support for communication with other hosts. Provided this is the case, the VMagent can operate on the host, sending and receiving the necessary messages to involve the host with the cluster. Once the remote virtual machine is started, additional support for remote class loading, syn-

chronization, and file input/output can all be supported through the communication primitives of the host. Using the standard network encoding for data, the physical architectures of the cluster are no longer an issue.

3.4 Summary

In this chapter, we discuss the architecture design that is necessary to accomplish transparent remote execution. The extra facilities required are discussed as to their purpose and how they operate within the current virtual machine. In the next chapter the implementation of this architecture is discussed in greater detail. Particular detail is spent on the needed support, and how it is provided.

CHAPTER 4

Implementation

In the previous chapter the architecture of the scalable virtual machine was discussed. In the implementation, many additional problems were encountered. Some of these problems are maintaining security, decision making on off-loading threads, providing consistent input and output control, loading classes for execution, handling exceptions and errors, and finally tolerating some network errors that can occur. This chapter discusses these problems and their solutions.

4.1 Security Issues

One of the emphasized features of the Java programming language is its built-in security as opposed to other object-oriented programming languages like C++ and Smalltalk that provide no security. This security feature must be maintained when scaling the virtual machine. Security is compromised in two main ways when scaling the virtual machine:

- Allowing only valid users to off-load threads to a host.
- Maintaining the application so that no other user can compromise its execution.

The second point is easily accomplished by ensuring that all users' applications run in a separate virtual machine at each host. This ensures the same security that the user has when executing on a shared UNIX machine. As discussed previously, having the applications share the same virtual machine would be more complex to implement. The first concern is a little more involved.

With the majority of computers being capable of running a Java virtual machine, the question arises as to how an individual or group can restrict the access to their computer(s) from other users. It is not sufficient to allow access only to those who possess an

account on the host. People to whom you wish to give access to a virtual machine are not necessarily the same group as those you wish to give full resource permissions. To allow for these different sets of users, the access procedure for a virtual machine must be independent of the normal resource access procedure on the host. We must provide an independent access system that implements the restrictions we would like to enforce. One restriction is the ability to explicitly deny all people access, as well as the ability to add individuals and specify the computers from which an individual can gain entry. This allows greater control over where a virtual machine is invoked, and can be used as a defense mechanism against malicious users. The proposed solution is for each computer to maintain a permissions file that holds the names of users who are granted access, as well as the machines from which they can gain entry. To allow for easier maintenance, the file can contain special characters that allow for generalizations to be made regarding users and hosts. Some sample entries from the file can be seen in Figure 4.1.

```
xyz.123.456.78    jadam
xyz.987.654.*     bcalm
*.*.*.*           gswift
xyz.987.654.321   *
```

Figure 4.1 Host Access file.

The first entry in the file allows access to the user “jadam” only from the machine with the IP address “xyz.123.456.789”. The wildcard character, ‘*’, can be used to denote generalizations of either hosts or IP address. In the second entry, the user “bcalm” can gain access from any machine that belongs to the internet class of “xyz.987.654.*”. This generalizes the number of hosts to any of 256 different possibilities. This generalization can be carried further in the third example in which the user “gswift” can gain access to a virtual machine from any IP address. The fourth entry depicts the generalization of the user. This allows access by any user who is connecting from the IP address “xyz.987.654.321”. It was decided to use IP addresses in the file as opposed to host names because hosts are more likely to have multiple names and host names are more likely to change. Using IP addresses also prevents the need for duplicate entries to allow

classes of IP addresses to be generalized in the file.

The administrator also has to provide some default permission in the event that the requesting host and user combination is not specified in the file. The default permission is generally to deny access to all unknown requests; however, if desired it can be set to provide access to all users. It is not suggested that one should allow any unknown requesting party access, but the practice is supported.

Each user has to maintain a file in their home directory listing the hosts where they have permission to execute Java programs. The filename is ".svmperm". With the list of hosts, the sthread, as described in section 3.3.1, can use this information to perform some load balancing of tasks. The virtual machine that is selected when scalability is desired is determined by a load-balancing algorithm, as explained in detail in the following section.

The problem arises in verifying that the process making the request is not attempting to gain unlawful entry. It is possible for a process to "fake" its identity and gain access for malicious purposes. The problem of authentication has been addressed many times in the field of cryptography [17]. By using a scheme for proof-of-identity the VMagent can ensure that the requesting process is legitimate.

All interactions through the VMagent are logged to provide a tracing mechanism for system administrators to protect against malicious users. After the VMagent has decided whether the request made by a virtual machine is accepted or declined, the VMagent adds a line to a log file containing the following information regarding the request:

- Whether the request was accepted or declined.
- The user id that was used in requesting access.
- The internet host (IP) address and port number from which the request was made.
- The time of the request.

With all of the above information recorded, it is possible for an administrator to determine the origin of a request and terminate certain access privileges. As discussed previously it is possible for a person to fake their identity when making a request. As a result, the logging scheme is only as strong as the security to ensure proof-of-identity.

4.2 Load Balancing

Whether a thread is created and executed locally or remotely is a decision which directly affects the performance benefits gained from scalability. The algorithm that is used for this purpose is derived from the “shortest” algorithm [3]. The algorithm chooses at random a distinct set of hosts. Each of the hosts in the set is probed to determine its load score. The task is transferred to the node with the lowest load score unless that load score is greater than or equal to the local load score. In that case, the originating node must process the task. The chosen node must process the task regardless of its state at the time the task actually arrives. This is designed with the intention of maintaining a fast start-up to a thread, while still attempting to gain scalability.

The load information obtained from each of the hosts includes the load averages for 1 minute, 5 minutes, and 15 minutes. The averages at these intervals are standard load averages provided by the UNIX system. Once the load information is obtained from each of the possible hosts, the formula for determining the best host is based on the 1 minute load average, with ties broken by the 5 minute load average, and further ties broken by the 15 minute load average. With the score computed for each host, the best host for handling the creation and execution of the new thread is contacted. This generic technique can be easily extended to other platforms.

The algorithm also provides several features that allow for greater efficiency in the scheme for finding the best host. In the virtual machine there exists an *idle* thread that executes when all other threads within the virtual machine are blocked due to input or waiting on some monitor. We have added a new thread called the *load* thread that has a priority one level higher than the *idle* thread. This new thread probes nodes that are available for executing a thread, and updates load information in the absence of the *sthread*. As hosts are probed, their load responses are stored for faster retrieval of load information when a thread is to be executed remotely. User threads make a request for the sthread to start a thread; after getting the load information from the load thread, the sthread makes a decision and communicates with the remote host if applicable through a shared socket. Each of the load responses that are obtained have a predefined life expectancy, after which the load information is considered “stale” and the host is probed again

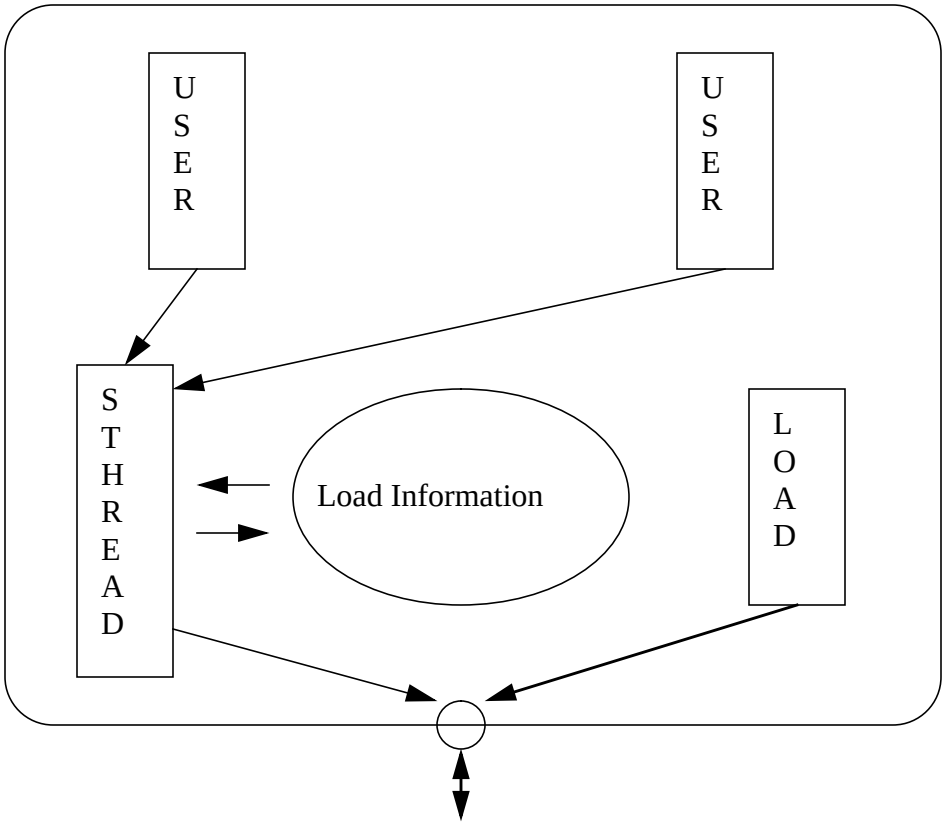

Figure 4.2 The load and sthread interaction within the virtual machine.

for a more recent workload. This reduces the amount of latency on a thread waiting for remote execution while sacrificing some processing time when the virtual machine is idle with all processes blocked on some condition. The lifetime of a load measurement is configurable, allowing greater control by the user. The interaction between the load and sthread threads can be seen in Figure 4.2.

A comparable approach for load balancing as discussed in Eager, Lazowska, and Zahorjan [3], is the “threshold” algorithm. The authors presented the algorithm as comparable to the “shortest” algorithm used here. Their conclusion is that the “shortest” algorithm is best in the event that the amount of information to be relayed between hosts is significantly large. In the Sumatra virtual machine, threads contain a significant amount of information and this information must be sent to a node when passing the thread for execution. Since the amount of information is rather large, it is considered that the extra work required for probing outweighs the costs of sending threads unnecessarily. From

this it is decided the “shortest” algorithm used is a better choice.

Currently, the basis of off-loading a thread is solely on the amount of processing resources available at the remote host. It is foreseen that the load balancing can be beneficial for the requisition of memory resources as well. The probing of information is easily scalable such that a probe is also performed at the same time on memory availability and is transmitted at the same time. With investigation, a relation can be deduced as to the amount of memory required to instantiate a class within a virtual machine. With this relation it can be determined whether a remote host can provide the necessary memory requirements to handle a thread that the original virtual machine cannot. This can be scaled further to include other resources or combinations thereof.

4.3 Input/Output

During remote execution of Java threads, the execution environment that surrounds the remote thread must be consistent with the environment the thread would have if locally executed. A significant part of the environment is the file system. To provide the same file system to the remote host that is also available at the local host, all of the system input/output (IO) actions at the remote host have been changed. These changes are such that when an IO action is performed, an IO request is sent to the root host to perform the action. This forces all the IO operations to be performed on the same filesystem as opposed to the possibly different filesystem at each remote host. To allow for this change, the virtual machine required a significant amount of work.

4.3.1 IO Operations

Within the Sumatra virtual machine, all system calls that are used in input/output are wrapped by functions that control the access to IO operations. These wrapper functions are used for all IO access including file, pipe, standard IO, and other communications. For the file system to be controlled, the functions that have to be changed are only those related to file manipulation. Clearly, it is not necessary for a thread executing on a

remote host to use a file descriptor at the root host if it is used for datagram communication. The functions that were determined to be used solely for file manipulation, were related to:

- Open/Closing files
- Read/Write operations to a file.
- Available to determine the status of a file for reading.
- Sync a file.
- Time-out to see if a file was ready for reading, within *time-out* milliseconds.
- Lseek to adjust the reading/writing position of the file.

Each of these functions were changed to direct all file IO to the root host. All communications between the remote and local hosts are handled by the sthread system thread at each host. Instead of the function making a system call to perform the operation the function creates a message to be sent to the sthread at the root host conveying the action to be performed. The sthread, upon receiving the message, performs the system action. Since the action is performed at the root host, the file actions are the same as if the thread were located on the root host. After execution of the request, a message is sent back to the requesting thread indicating the result of the request and any data that may have been generated as a result, such as a line of input that may have been read, or the number of bytes written to a file. In all situations, the message is sent to the root host and not the parent of the thread. For instance, if A starts a thread on B, and B starts a thread on C, C's I/O requests will go straight to A. This setup optimizes all remote I/O requests by reducing the amount of network traffic.

For each of the IO operations to provide the same service as if performed locally, the thread should be suspended until the result of the IO action is received. The result of the action may, in fact, change the next action of the user's Java thread to be performed. As such, when a thread makes a request, the action and the properties of the file descriptor are taken into consideration to determine whether the thread should be blocked until the result of the IO operation is received from the root sthread. If the thread should be

blocked, the thread is suspended upon sending the request to the root host. Once a response is received from the root host concerning the action which suspended the thread, the response will pass the result of the operation to the user thread and then notify the thread. This effectively awakens the thread with the result just as if the operation were performed locally.

4.3.2 File Handling

The file descriptor that is used as a handle to a file is no different from any other file descriptor that exists within the operating system of the hosts. It is a unique integer value that is mapped to a file. Since the file descriptor uniquely identifies a file, the file descriptor itself must be unique within the system. The operating system on each host guarantees that a file descriptor is unique, however, it cannot guarantee that the file descriptor is unique within the scalable virtual machine that spans multiple hosts. If the file descriptors were solely used for file manipulation, then this problem would not exist since all file manipulations are routed to the root host, and the operating system at that host can guarantee the uniqueness of the file descriptor. Unfortunately, file descriptors are used for all input/output manipulation including pipes and sockets. Clearly, we do not want or need to route *all* IO to the root host. If a user thread were to open a socket, then it is not necessary for the socket to be opened at the root host. It is better to have the socket totally at the remote host since this distributes some of the overall workload.

To provide unique file descriptors across the virtual machine for all file input/output, the child thread, upon opening a file, has a mapping for the file descriptor from its file descriptor to an associated file descriptor at the root host. This mapping allows remote hosts to map a file descriptor of their choice to the file descriptor known by the root. This gives the remote hosts the ability to be flexible as to how their file descriptors are used. A key part of this mapping is guaranteeing that the “dummy” file descriptor that maps to the real file descriptor will never be used by the operating system at the local host. To enforce this condition, the “dummy” file descriptor is chosen by making a request to the operating system for a file descriptor. This request reserves the file descriptor, and ensure that it is never used for any other purpose until the file descriptor is closed.

4.3.3 Synchronization

Within the Sumatra virtual machine, synchronization primitives exist for each file descriptor to control input/output access. When a thread performs an input/output action, the thread must first acquire the monitor for the file descriptor that the action is to be performed upon. It is possible that the monitor blocks the thread if another thread is in the monitor, and the requesting thread is notified when the monitor becomes available. Once inside the monitor the thread can perform its IO action, and upon leaving must exit the monitor to allow another thread to enter and perform its IO action on the file descriptor. These monitors provide a means of serializing requests so that operations performed upon file descriptors are atomic. The monitors are not to be confused with an ordering mechanism for concurrent requests.

Since all file descriptors need monitors to enforce IO operations as atomic, it was required that monitors for each file descriptor be held at both the root host where the system file descriptor is used, and as well at the remote host where the remote thread initiates the IO request. Since some of the work for each of the input/output operations is performed at both the root and remote host, for the operation to be atomic, it must be atomic at both sides. To accommodate this, a monitor for the file descriptor is held at the root host, and, as well, at each remote host that shares the file descriptor.

This architecture of monitors at both levels for each file descriptor has its advantages and disadvantages. The most notable disadvantage is the fact that all file IO is being routed through a central point and this central point can act as both a bottleneck, and a central point of failure within the scalable virtual machine. Unfortunately, there is nothing that can be done as regards the central point, since the virtual machine is not guaranteed to have the same file system available at any other host besides the root host. We can also take comfort from the fact that typically programs that use distributed resources for execution do not require a significant amount of file input/output. In addition, the multiple levels of monitors throttle requests to the host, reducing the possible bottleneck at the root. Since threads at a remote host have to gain access to a file descriptor through its monitor before the request is sent to the root host, this limits the number of requests sent to the root host to one request from each host. So at any single instant, the

root host has a maximum of *n* requests for each file descriptor, where *n* is the number of hosts. This is significantly less than if all threads were to send their request directly to the root node without any local monitor, resulting in a possible request for each thread. This results in a slow down for each of the requests since they have to go through two monitors as opposed to one, but it is necessary to keep the operations atomic.

4.3.4 Time-out

The input/output operation of “time-out” is a method that requires special consideration. The “time-out” method is given a file descriptor and a *time-out* value in milliseconds. The virtual machine is used to block the user thread making the request, and to poll the file descriptor. The file descriptor would be polled for up to *time-out* milliseconds to determine if there was something to be read, whether there was nothing to be read, or whether an error occurred in the polling process. Since the file descriptor may not exist on the local file system, several problems can arise. What if the number of milliseconds given by the user is too small a time interval to get a request to the root host and for the processing to occur and get a response back? What if the request message going to the root gets lost and a retransmission(s) is needed? What should happen if no response is received after *time-out* milliseconds have elapsed? All of these put at risk the possibility of getting a correct result to the function back to the remote thread in the time required.

The possibility of not getting a correct result from the root host in the amount of time available is not a significant problem. This can be safely said, since the Sumatra virtual machine does not guarantee to provide hard real-time functionality. Hence, any user program cannot assume that a response will be given in precisely *time-out* milliseconds or less. This does not mean that the problem can be ignored, but it is not as significant as it may be perceived. One solution is to perform the time-out operation in advance and buffer the solution. For instance, when the file descriptor is opened for reading or writing, perform the time-out operation and store the result until it is called. Since the result is only one of three possible integers, this is a viable solution. This solution however does not take into account the amount of effort that would be required to implement the propagation of the buffered result to all hosts within the virtual machine. In addition, the result also has to remain consistent among all of the virtual machines. For example, what if

user thread A performs a read operation as user thread B performs a time-out operation on a different host. The time-out operation will possibly have a different result after the read operation is performed. Attempting to buffer the time-out operation in advance is too complex and carries too much overhead for a solution.

The solution decided upon was to manipulate the amount of time that the user requested for a time-out value. When a user requests a time-out operation of *n* milliseconds, before the request is sent to the root host, the number of milliseconds is decreased by a network delay measurement, *delta*. *Delta,* is an average periodic sampling of *r* + *ra* as shown in Figure 4.3. This decrease takes into consideration the amount of time that is required for network delay. Thus, the time-out request now has a possibility of being completed by the root host before the requested number of milliseconds has elapsed. If no response is received after the reduced number of milliseconds has elapsed, then the time-out operation returns an error. This prevents the user thread from being suspended for an extended amount of time. In the special circumstance where the network delay is larger than the number of milliseconds to poll the file descriptor, the time-out value given by the user is increased to equal the network delay (*delta*) + 1. This gives the network time to get a response, and 1 millisecond to poll the file descriptor. This is a legitimate solution only because Java does not provide a hard real-time environment. Hence, the request does not necessarily need to be serviced in exactly *time-out* milliseconds.

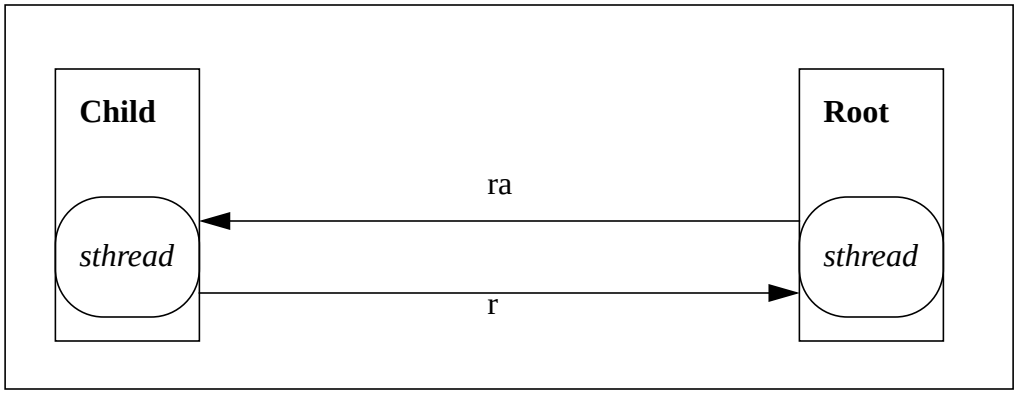

Figure 4.3 Network delay.

To provide positive results for this solution, the *delta* value needs to be chosen to give a good estimate of the network delay. Since the request and response are only ever going to be directed to the root host, the link between the child and root host is the sole

link that needs to be considered. To provide a good delta value, at various intervals, a message is sent to the root host, and a timer is started at the child. Upon receipt of a reply, the timer is stopped and the number of milliseconds is averaged with the previous known network delay. The interval between tests is dependent upon the amount of communication that occurs between the root and child hosts. The greater the number of messages, the shorter the time interval between samplings. This is indicative of the fact that the possibility of network delay increases with the increase of network use. In addition the averaging of previous network delays with the new measurement protects against extremities when sampling.

4.3.5 Input/Output Fault Tolerance

All of the input/output operations now use a datagram service to communicate to the root host for all file IO. Since the datagram service does not provide a guaranteed reliable service, each host within the virtual machine must itself provide some sort of fault tolerance to handle lost messages. Within the virtual machine that exists at each host, the sthread controls all communication that goes to another host within the overall virtual machine. This sthread maintains a data structure that keeps track of all remote IO requests that go out. The sthread is therefore able to resend requests for which it has not received a response. This looks after lost messages from the remote host. At the root host, the sthread controls all incoming IO from user threads dispersed to remote hosts. Again, since the requests all go to a central point, this point allows for easy recognition of duplicate messages, and keeps a record of unacknowledged responses.

4.3.6 Alternative I/O Solution

An alternative to providing I/O support at the remote hosts is to enhance the remote virtual machine to attempt direct access to the desired filesystem. Instead of the remote virtual machine sending an I/O request to the “root”, it could search for the origin of the filesystem to see if it was available over the network. If so, the remote virtual machine could open a connection with the filesystem’s server and directly interact with the files as opposed to going through the “root” host. This is particularly the case where the filesystem is on a host other than the “root”. The distribution of file control also reduces the

amount of network congestion at the “root”, and additionally makes the overall architecture more tolerant to faults. This can be seen in Figure 4.4 where host A is home of the

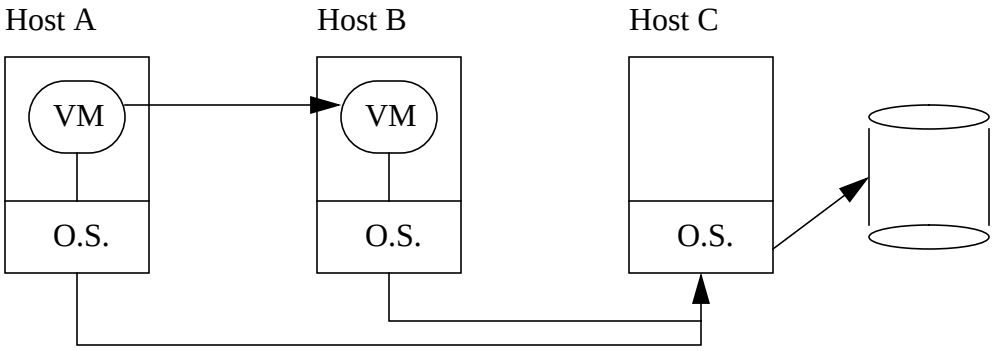

Figure 4.4 Cluster using a network file

remote virtual machine, and host B is the host of the original VM. Host C is the server for the network filesystem, and can possibly provide a more direct route for the remote virtual machine to any file manipulation.

There are several problems with this solution. A key problem is that the solution is only applicable in certain instances. No benefit is offered by using this solution unless the filesystem is not residing at the “root”. If the filesystem resides at the “root” host then all I/O requests are already forwarded directly to the filesystem. Likewise, this does not work if the remote virtual machine is not operating on a trusted host. For a host to gain access to another host’s filesystem it must be a trusted host by the filesystem host. All other solutions used keep in mind the idea of allowing remote execution at arbitrary hosts that are not providing any special or “trusted” status. The most important problem is that of management. Managing the file resources when they are controlled from multiple points is a rather tedious job. This is viewed as useless work since the majority of threaded applications use little or no file manipulation, resulting in less congestion at the root. Overall, the final decision came down to complexity of implementation and the benefits of the extra complexity, and the complexity far outweighed the potential minor gains.

4.4 Class Loading

In Java, the programmer defines classes and their interactions to make an application. After compilation, these defined classes exist in files named after the class with a ".class" extension, or a group of classes exist in a compressed file with a ".zip" extension. Each of these files contain the information necessary for the virtual machine to interpret the class in execution. To make interpretation simpler, each class is not read by the virtual machine until the class is needed. Once the class is required it is the job of the classloader to get the appropriate class definition.

The classloader first looks through a list of directories and zip files to find the desired class definition. The list of directories and zip files is defined by the environment variable CLASSPATH. Once the class is found, it is read into the virtual machine's memory and execution continues. With remote execution, it is possible that the class does not exist in a file accessible directly from a remote host. So what if the host needs this class to execute?

The only host that has guaranteed access to the file containing the class definition is the "root". As such, any remote host has to obtain the class definition from the root. With the availability of manipulating files at a root host, the classloader need only make its file Input/Output calls to the root instead of local Input/Output calls. It is not necessary or desirable for all classloading to be performed from the root host. Standard Java API classes are guaranteed to exist at each host and are consistent among hosts. To prevent unnecessary remote classloading, the classloader first searches through the standard classpath at the local host before attempting to load the class from the root host.

To ensure that a remote virtual machine only loads standard Java classes locally, the CLASSPATH environment variable is set to only contain the base Java API classes. If this is not true, correct execution cannot be guaranteed at the remote host since an incorrect class file can be located at the remote host.

4.5 Exception Handling

One problem in executing a thread remotely from the root host is the handling of errors and exceptions that occur during execution. In the Java programming language it is clearly defined how threads handle exceptions. The concern is to provide uniform handling of these exceptions whether they occur remotely or locally. The specific concern is when an exception can occur upon attempting to execute a thread. The Java language specifies that an exception can occur when a thread is starting in one of two cases:

1. The virtual machine is out of memory and cannot create the necessary structures to start a thread. The exception generated from this type of error is called “OutOfMemoryError”.
2. The virtual machine is attempting to start a thread that has been previously started and is still executing. This error generates an “IllegalThreadStateException”.

When a thread cannot be created due to a lack of memory, the virtual machine context switches to the garbage collector thread. The garbage collector attempts to reclaim as much memory as possible. Once all possible memory is reclaimed, then an attempt is made once again to create the thread. In the event that the thread cannot be created, then an “OutOfMemoryError” error occurs. When this error occurs the virtual machine is expected to simply halt. This is due to the fact that the Sumatra virtual machine has no way of recovering from this predicament. However, the remote virtual machine, upon encountering this error when attempting to create a remote thread, replies to the originating virtual machine with a **reject** message. This effectively sends the thread back to the original host for execution. If the original virtual machine encounters an “OutOfMemoryError”, then the entire scalable virtual machine will shutdown.

The rationale behind sending the reject back to the off-loading host instead of the child host attempting to off-load the thread is for two reasons. The first reason is that the off-loading host has just recently probed hosts in the process of off-loading. It knows the loads of other hosts that can accept the thread, as well it knows of any hosts that may have previously rejected the thread. For the child host to attempt to off-load the thread, it

may possibly have to update its probing information. More importantly, it may attempt to off-load the thread to a host that has already rejected the same thread previously. This lack of information by the child host may result in cycles of thread off-loading, where multiple hosts continue off-loading the thread to one another. The second reason for sending the thread back to the off-loading host for handling is book keeping. Having the thread off-loading information consistent at hosts to avoid these “cycles” results in a high amount of extra overhead work.

An “IllegalThreadStateException” only occurs when attempting to restart an already running thread. In the scalable virtual machine, each thread which is executed maintains a copy of the object at the originating virtual machine which made the request. Since the copy of the thread object can be checked for its state, whether running or not, it is possible to check for the occurrence of this exception before the thread is shipped for execution, thus, preventing this exception from happening at a remote host.

Many additional exceptions can occur within a thread. However, all other exceptions that occur within the thread are caught and dealt with appropriately at that point. There is no passing of an exception from a child thread to a parent, likewise from a parent to a child. This allows for normal processing of exceptions within the threads with the exception of terminal errors. When an error occurs that results in the virtual machine halting, the child or parent thread catching the exception notifies other virtual machines of the fatal error, and the entire virtual machine will halt.

4.6 Fault Tolerance

During the start-up process of a remote virtual machine many errors, such as duplicate thread objects executing can occur. Several steps have been taken to detect and handle them. From the perspective of the originating virtual machine, the only error that can occur is no response when a request is made for remote execution of a thread. To allow for detection of this fault, the requesting virtual machine maintains a timer and a count of the number of retries to communicate with the remote virtual machine or VMagent. In the

event that no response is received, the originating virtual machine simply resends the request. After *n* retries, the originating virtual machine stops attempting to off-load the thread and executes it locally.

It is still possible that the remote virtual machine has received the request for executing the thread, but its attempts to reply to the originating virtual machine were just unsuccessful. This results in multiple instances of the thread executing, one locally and one remotely. The remote thread object is referred to as a runaway (duplicate) thread since it was presumed to have failed on distribution. The remote thread does not know of any failure and begins execution immediately upon being received. This problem of runaway threads is prevented when the remote thread attempts to perform some global operation, such as file IO. When the global operation is performed, the remote virtual machine must communicate with the originating virtual machine and make a request for the global operation to take place. If the originating virtual machine receives a communication from an unknown thread object, then the thread can be terminated as it is a runaway object from a previous attempt to off-load a thread. This could be costly in that many thread objects might have duplicates executing on remote virtual machines due to failures in attempting to off-load threads. With the precautions taken to allow for lost messages, and time-outs, the number of runaway threads are rather low.

The remote virtual machine, upon receiving a request to execute a thread object must create the thread object and start execution of the object. To reduce the number of runaway threads, the remote virtual machine also keeps a timer and counter; resending the acknowledgment for the thread acceptance every *t* time units until a reply is received or it has resent the maximum number of times. It would be a waste of network resources for the remote virtual machine to keep sending the acceptance. In the event that no reply is received, the remote virtual machine still creates and executes the thread, in this case its acceptance has reached the originating virtual machine and its acknowledgment has been lost. Under these conditions the worst case scenario is for a runaway thread to occur. This is unfortunate, but it only slows down performance slightly while guaranteeing that all threads begin execution as quickly as possible.

4.7 Summary

In this chapter, the implementation of the distributed virtual machine, Sumatra, is discussed. Problems that arise are discussed in detail including the solution used to overcome each problem. In the next chapter, the topic is the performance of Sumatra. Two tests are used to generate timings of the performance and a comparison is made with the Sun's Java virtual machine.

CHAPTER 5

Performance Results

As with all system software, the heart of the matter is how well it works. In this chapter, we discuss some quantitative results that indicate how well our proposed solution performs. The discussion includes an overview of the testing environment in which the tests were performed, the two benchmarks that were used in testing, and an analysis of the results. These tests are not intended to be exhaustive, but to give an initial idea as to the practical performance of the virtual machine.

5.1 Testing Environment

The testing environment for our virtual machine consists of 3 Sparc workstations. These workstations had the following specifications:

- *HostA:* Sparc-5 workstation, running Solaris 2.6 operating system. 64Mb of RAM, with 198Mb of swap space. Connected to the network by a 10Mb ethernet card. MIPS rating of 91.21.
- *HostB:* Sparc-20 workstation, running Solaris 2.5.1 operating system. 128Mb of RAM, with 228Mb of swap space. Connected to the network by a 10Mb ethernet card. MIPS rating of 69.551.
- *HostC:* Sparc-10 workstation, running Solaris 2.5.1 operating system. 156Mb of RAM, with 295Mb of swap space. Connected to the network by a 10Mb ethernet card. MIPS rating of 108.189.

It is also important to note that these machines were not executing in a dedicated cluster. Each was connected to the local network and was in use by many users. This distorting effect was controlled to a certain extent in testing, by performing tests during off-hours to minimize the amount of usage by other users.

5.2 *n*-queens Problem

The *n*-queens problem is widely known in the field of distributing computing. The problem concerns placing *N* queens on a chessboard. Given a chessboard of size *n* x *n*, place *n* queens on the board such that no queen can "take" another queen. A queen can "take" a piece on the chessboard by moving horizontally, vertically, or diagonally any number of positions. The goal is to find all correct placements of the *n* queens.

For testing purposes, this problem is ideal since the problem is very easily expandable, and it requires a significant amount of processing time. If a more intense computation is desired, the size of *n* is increased to exponentially increase the size of the problem. In addition, the solution is easily implemented using a recursive backtracking technique.

5.2.1 The Algorithm

The algorithm used to implement the *n*-queens problem is deliberately rather simple. It is based on the observation that no correct solution can include multiple queens on the same row, since they would be able to attack each other. Knowing this, and that there are *n* queens, there must be exactly one queen on each row. The real problem is reduced to placing each queen on its row.

The algorithm follows from this in the following manner:

1. Place a queen in the first column of the row.
2. Check to see if the queens now positioned on the chess board are in such a way that no queen can take another queen.
3. If this is true and it was the *n*th queen, then print the positions of all queens.
4. If this is true but it was not the *n*th queen, move to the next row and start from step 1.
5. If this is false and the current queen is not at the last column position, advance the current queen to the next column position.
6. If this is false and the current queen is at the last column position, go back to the previous row and advance the queen one column.

This process is repeated starting at step 1 with the first queen on the first row, and is continued until there is no room to advance any queen on any row. This algorithm is effective in that it generates all correct placements. By generating all placements, it is guaranteed that exactly the same amount of work is required whether the program runs on a single host, or many hosts. The backtracking is quite good in that it effectively eliminates branches of possibilities immediately as soon as it detects that a correct solution cannot be obtained using the current layout. For instance, in Figure 5.1 it can be seen that

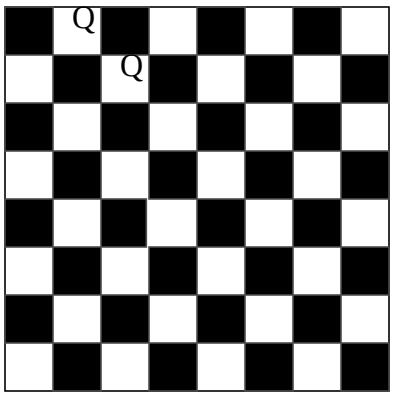

Figure 5.1 Example of benefit of using backtracking.

the two queens can take each other and that no matter what positions the other queens are placed in, a correct solution cannot be generated from this point. The algorithm detects this condition and advances the queen in the second row before proceeding further, thus immediately eliminating a large number of possibilities.

The implementation follows this algorithm very closely. The recursion is performed through the system stack using method calls recursively. A function is provided to test, given a chess board, whether two queens can “take” one another. The implementation is included in Appendix B. The implementation includes printing of the solutions to standard output. This is important since the standard output is redirected back to the “root” host and is not printed at the remote host where execution is performed. This adds complexity to the problem and results in a slower execution than if no output were produced. This adds to the value of the analysis of the results as it provides a way of determining the point at which the amount of processing outweighs the amount of communication overhead, making the remote execution worthwhile. This is also referred

to as determining the optimal grain size.

5.2.2 The Results

The *n*-queens problem is executed on clusters consisting of 1, 2, and 3 hosts. Figure 5.2 shows the average results from the test, and the actual timing values are in Table 5.1. Since the hosts used in the computation are not of the same configuration the timings given for executing the benchmark on a single host is the average of all three hosts computation. Likewise, the times shown for two hosts are the average of the combinations of two hosts. The times of execution for each test in single execution and in a cluster of 3 are

Figure 5.2 Results from N-queens benchmark.

included in the figure. Some times may be deceiving in the figure because the machines are not identical. In addition, since the machines are not dedicated, longer running process stand a higher chance of having to share the processor with other processes. The results from the test appear very promising and show roughly an even split in the compu-

tation time among the nodes involved in the execution. Hence, it is close to the theoretical maximum potential gain per processor added. This is an encouraging gain in performance, however, there are two points to remember:

- The single computation time is an average of each processor's time to perform the execution.
- The amount of communication between the threads in the execution is minimal, so there is very little overhead.

Cluster Configuration	**6 X 6**	**8 x 8**	**9 x 9**	**12 x 12**
host A	3.5	28.5	122.5	26355.5
host B	3.5	29.5	128.5	29656
host C	2	20	68	20308
Single host average	3	26	106.33	25439.83
host A and host B	5.5	12.5	42	8103.5
host A and host C	4.5	9.5	40.5	7077.5
host B and host C	4	9	34	7961.5
Two hosts average	4.67	10.33	38.83	7714.17
host A, host B, and host C	4.5	12.5	23	4895

Table 5.1. *N*-queens timings given for each configuration in seconds.

The results obtained from the test are extremely good, and in some instances, exceed a linear speed-up. There are several factors that contribute to this super-linear performance. Most notable is the cluster of computers is not a dedicated cluster. This provides fluctuation in both network and host activity that cannot be totally eliminated. Steps such as running the tests multiple times, and running during off hours have been taken, but not all of the interference can be eliminated. In addition, the Solaris operating system has a scheduling mechanism that incorporates aging into the priority. As such, the longer the process takes to execute, the lower the priority of the process becomes as time expires. Shorter processes will not be as drastically affected, if at all, by the aging process as longer processes. Finally, the division of work between the threads is not guaranteed to be equal. In various instances, it is not possible to split the chessboard evenly, such as the 8 x 8 board into 3 even pieces. Even when the chessboard can be divided evenly, it is not guaranteed that each division of the board will render the same amount of

work or results for that matter. As such, it is possible that the lower load threads are distributed to the lower power processors, making the computation on the low end nodes low. These factors combined add a reasonable amount of distortion to the numbers and result in cases of a super-linear speed-up.

5.3 Fractal Generation

Recently within the field of mathematics, it was discovered that the graphing of certain functions generated very interesting graphs. Some of these graphs display certain recursive features, and several others show striking resemblances to things found in nature such as leaves. The research of these functions has been pursued rather heavily in the past few years and have been tied to applications such as data compression [5][15][19].

5.3.1 The Mandelbrot Set

The Mandelbrot set, named after its discoverer Benoit Mandelbrot, is interesting in the fact that it displays recursive features as pictured in Figure 5.3. It is calculated by the

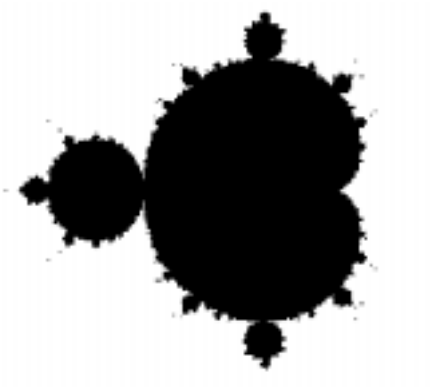

Figure 5.3 Graph of Mandelbrot set.

simple mathematical expression

$$Z_n = (Z_{n-1})^2 + C \tag{5.1}$$

where Z_n, Z_{n-1}, and C are complex numbers. For the base case, $Z_0 = 0$. The Mandelbrot set is defined only in the circle whose origin is (0,0) and radius 2. As such, the graph is selected to map some region of this circle. To graph the set, the color of each pixel is generated by iteratively calculating the function from 5.1. Initially the value of C is (a + b***i***) where a and b are the x and y coordinates respectively of the point to be calculated. This expression is calculated until a maximum number of iterations are performed, or when Z_n is infinite. The number of iterations performed before this condition occurs is the color of the pixel at that point on the graph.

Setting the maximum number of iterations is a matter of preference depending on the precision of the graph desired. Increasing the maximum number of iterations will result in the set's outline becoming sharper. Determining the point where Z_n becomes infinite is a more involved problem. The value of Z_n is the length of the vector given by it's real and imaginary parts. To calculate the length of a vector, the formula:

$$\text{Length} = \text{SQRT}(a^2 + b^2) \tag{5.2}$$

can be used. Knowing when this value is infinite is not possible, although, when the value exceeds 2 it is known that it will eventually reach infinity. This is due to the region in which the Mandelbrot set exists.[1]

5.3.2 The Algorithm

The objective is to graph some region of the Mandelbrot set. As such there is a grid for which all points require some amount of computation. Similar to the *n*-queens problem, it allows for easily breaking down the overall graph into sections and then computing the sections separately. Each section of the graph then corresponds to a thread of computation that will generate the colors of the pixel for each point in that section. As depicted in Figure 5.4, each of the pieces generated by the threads can be concatenated to form the final full graph. Once all sections have been computed, they can easily be

1. In the implementation, the square root is omitted and the value is compared to 4 instead of 2. This saves some computation time.

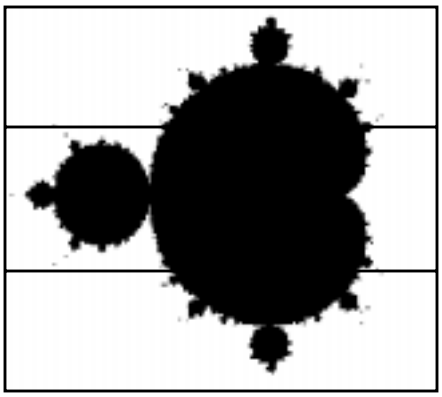

Figure 5.4 Mandelbrot set division by thread computation.

placed back together to form the entire graph. The amount of computation required for different regions will vary, so different threads will do different amounts of work, but they are close enough to be comparable.

The overall implementation strategy to solve the problem with multiple threads is very similar to the strategy used in the *n*-queens problem. However, this application requires that its results are stored in a file, and not just printed to the screen as in *n*-queens. The size of the graph that is generated is 1024 x 768, equalling 786,432 pixels of information that is to be written to files. For each pixel that is computed remotely, an IO request on the original virtual machine is needed to save it. This increases the amount of communication that occurs between the remote host and the root host, resulting in a decrease of performance, and shows the effects of high communication requirements together with remote execution. The benchmark application that is used is attached in Appendix C.

5.3.3 The Results

The test times generated for the Mandelbrot set are given in Table 5.2. Five different times are given to represent five combinations of clusters comprising of the three hosts available. As can be seen from the results, there is a gain in performance by adding processors to the computation. With the addition of each processor to the cluster, the overall computation time is reduced. However, one can see that the percentage gain in performance by adding the third host is less than the expected percentage gain with respect to the cluster of two hosts. The reason for this is the third host added to the cluster has less

raw computing capabilities and as such lowers the overall processing capabilities. Theoretically, when a cluster computes an application, the approximate time required is the time of the slowest processor to compute the entire application divided by the number of hosts in the cluster. In the case that the cluster contains a relatively slow host, the overall execution time increases. This shows that when off-loading threads, not only does the load at the remote host matter, but also the overall computing potential at the remote host.

Cluster Configuration	Time in Seconds
host A	3472
host B	3252
host C	2226
host A and host B	1435
host A and host C	1344
host B and host C	1396
All hosts	747

Table 5.2. Timing for Mandelbrot Set Calculation.

Another factor in the relatively poor performance when using three hosts is the increase in communication. In the case with two hosts, there is only one flow of messages to the “root” host handling I/O. However, when a third host is added to the cluster, there are then two hosts simultaneously directing I/O requests at the “root” host, thus resulting in a higher volume of network traffic, and a higher percentage of packet loss. This packet loss increases the overhead of maintaining the distributed threads, and reduces the gain of adding the extra host.

5.4 Results Analysis

The results from both tests combined can be used to formulate some overall conclusions concerning the software. As can be seen with the *n*-queens benchmark there exists a point where the overhead of off-loading a thread outweighs the amount of increased processing. This is a well known fact, but the key point is that granularity of the thread does not have to be large to gain performance benefits. In the case of *n*-queens, benefits were seen in off-loading even when a single host could perform the entire computation in 26 seconds. In the realm of distributed processing, applications tend to require long com-

putation times, and 26 seconds is considered a short interval of time, especially when the off-loading at this point reduces the computation time by 50%. Whether an application can benefit from the software is always dependent on the particular application, but this shows the threads do not have to be coarse grained.

As with all distributed systems, performance degrades with the increase of message passing between the threads. The purpose of the second test is to show how the new virtual machine performs when message passing is high. When both the results from *n*-queens and Mandelbrot are compared with respect to their percentage of gains, it can be seen that Mandelbrot does not have as high an overall percentage gain. Table 5.3 gives the percentages of time decrease by distributing the application.

Number of hosts	**N-Queens time**	**% gain for *n*-queens**	**Mandelbrot Set time**	**% gain for Mandelbrot Set**
1 host	25440 secs	N/A	2983 secs	N/A
2 hosts	7714 secs	330%	1392 secs	214%
3 hosts	4895 secs	520%	747 secs	399%

Table 5.3. Average percentage gains for *n*-queens and Mandelbrot Set.

The reason for *n*-queens showing greater decreases in time as opposed to the Mandelbrot benchmark is due to the communications overhead needed in the Mandelbrot application. The extra message passing reduces the amount of computation time gained. With the increase of messages, the increase in hosts has less of an effect, although there is still a significant gain in performance. This gives encouragement that despite the amount of message passing there are still performance gains to be had, even in a high message passing application.

5.5 Summary

The results produced by the benchmarks are encouraging and expected. Threaded applications with little or no dependence on one another demonstrate tremendous gains by adding hosts to the computation. The key points revealed by the tests show the threads making up the application do not need to be coarse grained. Obviously, the heavier the threads the better, but this is not required. Likewise, the architecture can tolerate a reason-

ably high level of message passing and still provide a performance increase. As expected, the performance gain decreases as more hosts are added to the cluster and concurrent message passing increases. The next chapter concludes the thesis with some future work insights.

CHAPTER 6

Conclusions and Further Research

6.1 Summary of Results

The idea of this thesis was to off-load Java threads to speed up execution time of Java applications. This was accomplished by manipulating the virtual machine when starting a thread to create a new virtual machine at a remote host, and then pass the thread to that host for execution. With this off-loading, several issues were encountered and dealt with including security, file input/output, exceptions, load balancing, and fault tolerance.

Using the new virtual machine, overall performance gains are observed by distributing the threads as opposed to executing the threads on a single host. Benchmark tests demonstrate this point and give some indication of the amount of potential gain. Likewise, these applications verify the transparency of the distribution without intervention by the developer.

The *n*-queens test contains a high amount of computation and demonstrates the minimum granularity of threads such that distribution is worthwhile. In this benchmark execution time was decreased to 30% of the original time required. The Mandelbrot set computation gives an indication of the effect of message passing on distribution. Both tests give positive results showing threads do not need to be of coarse granularity, nor does the application need to consist of completely independent threads for a high gain in performance to be observed. This was evident by the decrease in execution time to 38.6%.

The new virtual machine maintains the same usability of the stand-alone Java virtual machine from Sun. The application from the programmer's perspective is no different than that for a single host. All aspects of the distributed computation from the view of the user are hidden and carried out with no extra effort in development time, allowing an

increase in performance at zero cost to the developer.

6.2 Future Work

There are many aspects of this work in which further research can be performed. One of the restrictions of this research is that the benchmarks consisted of non-synchronized threads. An important topic is expanding the new virtual machine to support synchronization of threads. It is foreseen that an all-software distributed shared memory system would help ease the burden in providing this capability [1][2][9][16]. This may also prove to be useful in providing data marshalling for passing classes between hosts.

APPENDIX A

User Guide for Virtual Machine

A.1 Setup and Initialization

The installation of the new virtual machine is rather trivial. Due to an agreement with Sun Microsystems, the source code is not available. However, the binary is available for the new virtual machine. You can download the binary from *http://www.csc.uvic.ca/~ken/sumatra*. It is currently only available for the Sun Sparc running solaris 2.5.1 or 2.6 and comes tarred and gzipped.

Once the file is downloaded, you can extract the software using the command:

gzip -d sumatra.tar.gz | tar -xvf

A directory is created with the name Sumatra. Inside is located the following:

- a README file with installation instructions.
- a lib directory containing the dynamic library “libSumatra.so” for the software.
- the directories “mandelbrot” and “nqueens” containing the source for the benchmarks described in the thesis.
- a bin directory. The bin directory contains the binaries for the daemon at each remote host, “VMagent”, and the virtual machine, “Sumatra”.
- a “.ftsvm” config file.
- a “svmperm” config file.

To install the software you must first copy and edit the configuration files to the appropriate places. The “svmperm” file is copied to the directory /etc. Once there, the file is editted to include the names of all hosts and users that can gain access to a virtual machine from a remote host. Sample entries are included in the file to help the configuration. The “.ftsvm” configuration file is copied to the home directory of any user that wishes to use the virtual machine. Editting the file, is necessary to include the names of all hosts that the user has remote access to a virtual machine.

Once the configuration files are in place, it is now necessary to start the “VMagent”

running at each of the remote hosts to receive requests. For the “VMagent” to run correctly the user that starts the agent must have permission to read the UNIX device “/dev/kmem”. This is necessary so the agent can calculate host loads to report to virtual machines. This may require running the agent as the “root” user. The VMagents are daemon processes so it is conveinient to pipe their output to /dev/nul and place the process in the background. The software is now installed.

A.2 Example Running

Running the software is no different than running the standard Sun java virtual machine. To demonstrate the process, to run the n-queens benchmark test, first change into the nqueens directory. You must compile the source code into standard java classes using any java compiler you prefer that is Java version 1.1.x. Once the classes are compiled you can run the test using the command:

Sumatra Queen

The nqueens test should now start to execute, and off-load threads accordingly to the load. All applications follow this same basic procedure.

APPENDIX B

N-Queens Implementation

B.1 Queen Class

```
import java.lang.*;

public class Queen
{
  public static void main(String[] args)
  {
    SlaveOne a = new SlaveOne();
    SlaveTwo b = new SlaveTwo();
    SlaveThree c = new SlaveThree();
    String greeting[] = new String[3];
    int i;

    System.out.println("Starting the Four Slaves");
    a.start();
    b.start();
    c.start();
    System.out.println("Four Slaves are Processing");
  }
}
```

B.2 SlaveOne Class

```
import java.lang.*;
import java.io.*;

public class SlaveOne extends Thread
{
  private int MAXROW;
  private int MAXCOL;
  private int ChessBoard[][];

  SlaveOne()
  {
    MAXROW = 6;
```

```
  MAXCOL = 6;
  ChessBoard = new int[MAXROW][MAXCOL];
}

private void PrintBoard()
{
  int i, j, k = 1;
  StringBuffer s = new StringBuffer();
  String str;

  for (i = 0; i < MAXROW; i++)
    for (j = 0; j < MAXCOL; j++)
      if (ChessBoard[i][j] == 1)
      {
        s.append(j);
        s.append(" ");
      }

  str = s.toString();
  System.out.println(str);
  return;
}

private int PlaceQueen(int row)
{
  int col = 0;

  while (col < MAXCOL)
  {
    ChessBoard[row][col] = 1;
    if (CheckBoard() == 1)
    {
      if (row + 1 == MAXROW)
        PrintBoard();
      else if (row + 1 < MAXROW)
        if (PlaceQueen(row + 1) == 1)
          return 1;
    }
    ChessBoard[row][col] = 0;
    col++;
  }
return 0;
}

private int CheckBoard()
{
```

```
int i, j, x, y;
int temp[][] = new int[MAXROW][MAXCOL];

for (i = 0; i < MAXROW; i++)
  for (j = 0; j < MAXCOL; j++)
    temp[i][j] = ChessBoard[i][j];

for (i = 0; i < MAXROW; i++)
  for (j = 0; j < MAXCOL; j++)
    if (ChessBoard[i][j] == 1)
    {
      /* Mark the remaining col */
      for (x = 0; x < MAXCOL; x++)
        if ((ChessBoard[i][x] == 1) && (x != j))
          return 0;

      /* Mark the remaining row */
      for (x = 0; x < MAXROW; x++)
        if ((ChessBoard[x][j] == 1) && (x != i))
          return 0;

      x = 1;
     while (((i + x) < MAXROW)&&((j + x) < MAXCOL))
     {
       if (ChessBoard[i + x][j + x] == 1)
         return 0;
       x++;
     }

     x = 1;
     while (((i - x) > -1) && ((j - x) > -1))
     {
       if (ChessBoard[i - x][j - x] == 1)
         return 0;
       x++;
     }

     x = 1;
     while (((i - x) > -1) && ((j + x) < MAXCOL))
     {
       if (ChessBoard[i - x][j + x] == 1)
         return 0;
       x++;
     }

     x = 1;
```

```
        while (((i + x) < MAXROW) && ((j - x) > -1))
        {
          if (ChessBoard[i + x][j - x] == 1)
            return 0;
          x++;
        }
      }

    return 1;
  }

  public void run()
  {
    int i, j, k;

    for (k = 0; k < 2; k++)
    {
      for (i = 0; i < MAXROW; i++)
        for (j = 0; j < MAXCOL; j++)
          ChessBoard[i][j] = 0;
      ChessBoard[0][k] = 1;
      PlaceQueen(1);
    }
    return;
  }
}
```

B.3 SlaveTwo Class

```
import java.lang.*;
import java.io.*;

public class SlaveTwo extends Thread
{
  private int MAXROW;
  private int MAXCOL;
  private int ChessBoard[][];

  SlaveTwo()
  {
    MAXROW = 6;
    MAXCOL = 6;
    ChessBoard = new int[MAXROW][MAXCOL];
  }
```

```
private void PrintBoard()
{
  int i, j, k = 1;
  StringBuffer s = new StringBuffer();
  String str;

  for (i = 0; i < MAXROW; i++)
    for (j = 0; j < MAXCOL; j++)
      if (ChessBoard[i][j] == 1)
      {
        s.append(j);
        s.append(" ");
      }

  str = s.toString();
  System.out.println(str);
  return;
}

private int PlaceQueen(int row)
{
  int col = 0;

  while (col < MAXCOL)
  {
    ChessBoard[row][col] = 1;
    if (CheckBoard() == 1)
    {
      if (row + 1 == MAXROW)
        PrintBoard();
      else if (row + 1 < MAXROW)
        if (PlaceQueen(row + 1) == 1)
          return 1;
    }
    ChessBoard[row][col] = 0;
    col++;
  }
  return 0;
}

private int CheckBoard()
{
  int i, j, x, y;
  int temp[][] = new int[MAXROW][MAXCOL];

  for (i = 0; i < MAXROW; i++)
```

```
  for (j = 0; j < MAXCOL; j++)
    temp[i][j] = ChessBoard[i][j];

for (i = 0; i < MAXROW; i++)
  for (j = 0; j < MAXCOL; j++)
    if (ChessBoard[i][j] == 1)
    {
      /* Mark the remaining col */
      for (x = 0; x < MAXCOL; x++)
        if ((ChessBoard[i][x] == 1) && (x != j))
      return 0;

      /* Mark the remaining row */
     for (x = 0; x < MAXROW; x++)
       if ((ChessBoard[x][j] == 1) && (x != i))
     return 0;

     x = 1;
     while (((i + x) < MAXROW)&&((j + x) < MAXCOL))
     {
       if (ChessBoard[i + x][j + x] == 1)
         return 0;
       x++;
     }

     x = 1;
     while (((i - x) > -1) && ((j - x) > -1))
     {
       if (ChessBoard[i - x][j - x] == 1)
         return 0;
       x++;
     }

     x = 1;
     while (((i - x) > -1) && ((j + x) < MAXCOL))
     {
       if (ChessBoard[i - x][j + x] == 1)
         return 0;
       x++;
     }

     x = 1;
     while (((i + x) < MAXROW) && ((j - x) > -1))
     {
       if (ChessBoard[i + x][j - x] == 1)
         return 0;
```

```
        x++;
       }
      }

   return 1;
  }

  public void run()
  {
   int i, j, k;

   for (k = 2; k < 4; k++)
   {
    for (i = 0; i < MAXROW; i++)
     for (j = 0; j < MAXCOL; j++)
      ChessBoard[i][j] = 0;
    ChessBoard[0][k] = 1;
    PlaceQueen(1);
   }
   return;
  }
}
```

B.4 SlaveThree Class

```
import java.lang.*;
import java.io.*;

public class SlaveThree extends Thread
{
  private int MAXROW;
  private int MAXCOL;
  private int ChessBoard[][];

  SlaveThree()
  {
   MAXROW = 6;
   MAXCOL = 6;
   ChessBoard = new int[MAXROW][MAXCOL];
  }

  private void PrintBoard()
  {
   int i, j, k = 1;
   StringBuffer s = new StringBuffer();
```

```
  String str;

  for (i = 0; i < MAXROW; i++)
    for (j = 0; j < MAXCOL; j++)
      if (ChessBoard[i][j] == 1)
      {
        s.append(j);
        s.append(" ");
      }

  str = s.toString();
  System.out.println(str);
  return;
}

private int PlaceQueen(int row)
{
  int col = 0;

  while (col < MAXCOL)
  {
    ChessBoard[row][col] = 1;
    if (CheckBoard() == 1)
    {
      if (row + 1 == MAXROW)
        PrintBoard();
      else if (row + 1 < MAXROW)
        if (PlaceQueen(row + 1) == 1)
          return 1;
    }
    ChessBoard[row][col] = 0;
    col++;
  }
  return 0;
}

private int CheckBoard()
{
  int i, j, x, y;
  int temp[][] = new int[MAXROW][MAXCOL];

  for (i = 0; i < MAXROW; i++)
    for (j = 0; j < MAXCOL; j++)
      temp[i][j] = ChessBoard[i][j];

  for (i = 0; i < MAXROW; i++)
```

```
for (j = 0; j < MAXCOL; j++)
  if (ChessBoard[i][j] == 1)
  {
    /* Mark the remaining col */
    for (x = 0; x < MAXCOL; x++)
      if ((ChessBoard[i][x] == 1) && (x != j))
        return 0;

   /* Mark the remaining row */
   for (x = 0; x < MAXROW; x++)
     if ((ChessBoard[x][j] == 1) && (x != i))
       return 0;

   x = 1;
   while (((i + x) < MAXROW)&&((j + x) < MAXCOL))
   {
     if (ChessBoard[i + x][j + x] == 1)
       return 0;
     x++;
   }

   x = 1;
   while (((i - x) > -1) && ((j - x) > -1))
   {
     if (ChessBoard[i - x][j - x] == 1)
       return 0;
     x++;
   }

   x = 1;
   while (((i - x) > -1) && ((j + x) < MAXCOL))
   {
     if (ChessBoard[i - x][j + x] == 1)
       return 0;
     x++;
   }

   x = 1;
   while (((i + x) < MAXROW) && ((j - x) > -1))
   {
     if (ChessBoard[i + x][j - x] == 1)
       return 0;
     x++;
   }
 }
```

```
    return 1;
  }

  public void run()
  {
    int i, j, k;

    for (k = 4; k < 6; k++)
    {
      for (i = 0; i < MAXROW; i++)
        for (j = 0; j < MAXCOL; j++)
          ChessBoard[i][j] = 0;
      ChessBoard[0][k] = 1;
      PlaceQueen(1);
    }
    return;
  }
}
```

APPENDIX C

Mandelbrot Set Implementation

C.1 BitMapFileHeader Class

```
import java.lang.*;
import java.io.*;

class BitMapFileHeader
{
  public short  bfType;
  public int bfSize;
  public short  bfReserved1;
  public short  bfReserved2;
  public int bfOffBits;
};
```

C.2 BitMapInfoHeader Class

```
import java.lang.*;
import java.io.*;

class BitMapInfoHeader
{
  public int biSize;
  public int biWidth;
  public int biHeight;
  public short  biPlanes;
  public short  biBitCount;
  public int biCompression;
  public int biSizeImage;
  public int biXPelsPerMeter;
  public int biYPelsPerMeter;
  public int biClrUsed;
  public int biClrImportant;
};
```

C.3 Mandel Class

```
import java.lang.*;
import java.io.*;

public class Mandel
{
  Mandel()
  {
  }

  public static void main(String[] args)
  {
    MandelOne m1 = new MandelOne();
    MandelTwo m2 = new MandelTwo();
    MandelThree m3 = new MandelThree();

    System.out.println("Starting Fractal Generation");
    m1.start();
    m2.start();
    m3.start();
    System.out.println("Finished Fractal Generation");
  }
};
```

C.4 MandelOne Class

```
import java.lang.*;
import java.io.*;

public class MandelOne extends Thread
{
  public int cols = 1024;
  public int rows = 768;
  public int BI_RGB = 0;
  private double xmin = -1.75;
  private double xmax = 1;
  private double ymin = -1.15;
  private double ymax = 1.15;
  private int iter = 2000;
  private int radius = 2;
  private int[][] window = new int[rows][cols];
  private int temp[] = { 0, 0, 0, 255, 0, 0, 0, 255,
    0, 0, 0, 255, 255, 0, 255, 0, 255, 255, 255,
```

```
  255, 0, 188, 143, 143, 230, 232, 250, 79, 47,
  79, 205, 205, 205, 112, 219, 147, 255, 255, 255 };
private byte color[] = new byte[39];

MandelOne()
{
  int i, j;

  for (i = 0; i < 39; i++)
    color[i] = (byte)temp[i];

  for(i = 0; i < rows; i++)
    for(j = 0; j < cols; j++)
      window[i][j] = 0;
}

public void revwrite(FileOutputStream fp, int ptr,
  int bytes) throws IOException
{
  int i, d;
  byte[] tmp = new byte[4];

  d = ptr;
  for (i = 0; i < bytes; i++)
  {
    tmp[i] = (byte)(d % 256);
    d = d >> 8;
  }
  fp.write(tmp, 0, bytes);
  return;
}

public void revwrite(FileOutputStream fp, short ptr,
  int bytes) throws IOException
{
  int d;
  int i;
  byte[] tmp = new byte[4];

  d = (int)ptr;
  for (i = 0; i < bytes; i++)
  {
    tmp[i] = (byte)(d % 256);
    d = d / 256;
  }
  fp.write(tmp, 0, bytes);
```

```
  return;
}

public void mandelbrot()
{
  double dx;
  double dy;
  double x, y;
  double cr, ci;
  double zr, zi;
  double zsqr, zsqi;
  double radsqrd = radius * radius;
  int j, p, i, loop;

  dx = (xmax - xmin) / cols;
  dy = (ymax - ymin) / rows;
  x = xmin - dx / 2;
  loop = rows / 3;
  for (j = 0; j < cols; j++)
  {
    x = x + dx;
    y = ymin - dy / 2;

    for (i = 0; i < loop; i++)
    {
      y = y + dy;
      cr = x;
      ci = y;
      zi = zr = 0;

      p = 0;
      while ((p < iter)&&(zr*zr + zi*zi < radsqrd))
      {
        p++;
        zsqr = zr * zr - zi * zi;
        zsqi = 2 * zr * zi;
        zr = zsqr + cr;
        zi = zsqi + ci;
      }

      if (p == iter)
        window[i][j] = 0;
      else
        window[i][j] = (p / 100) + 1;
    }
  }
```

```
}

public void bitmapit()
{
 BitMapFileHeader bmfh = new BitMapFileHeader();
 BitMapInfoHeader bmih = new BitMapInfoHeader();
 int i, j, loop;
 FileOutputStream fp;

 try
 {
  fp = new FileOutputStream("Mandel1.bmp");

  bmfh.bfType = 19778; /* BM - Always this value. */
  bmfh.bfReserved1 = 0;
  bmfh.bfReserved2 = 0;
  bmfh.bfOffBits = (int)54;
  bmfh.bfSize = rows * cols * 3 + bmfh.bfOffBits;

  bmih.biSize = (int)40;
  bmih.biWidth = (int)cols;
  bmih.biHeight = (int)rows;
  bmih.biPlanes = (short)1;
  bmih.biBitCount = (short)24; /* bits per pixel */
  bmih.biCompression = BI_RGB; /* No Compression */
  bmih.biSizeImage = rows * cols * 3;
  bmih.biXPelsPerMeter = 3780;
  bmih.biYPelsPerMeter = 3780;
  bmih.biClrUsed = 0;
  bmih.biClrImportant = 0;

  revwrite(fp, bmfh.bfType, 2);
  revwrite(fp, bmfh.bfSize, 4);
  revwrite(fp, bmfh.bfReserved1, 2);
  revwrite(fp, bmfh.bfReserved2, 2);
  revwrite(fp, bmfh.bfOffBits, 4);
  revwrite(fp, bmih.biSize , 4);
  revwrite(fp, bmih.biWidth , 4);
  revwrite(fp, bmih.biHeight , 4);
  revwrite(fp, bmih.biPlanes , 2);
  revwrite(fp, bmih.biBitCount , 2);
  revwrite(fp, bmih.biCompression , 4);
  revwrite(fp, bmih.biSizeImage , 4);
  revwrite(fp, bmih.biXPelsPerMeter , 4);
  revwrite(fp, bmih.biYPelsPerMeter , 4);
  revwrite(fp, bmih.biClrUsed , 4);
```

```
    revwrite(fp, bmih.biClrImportant , 4);

    loop = rows / 3;
    for (i = 0; i < loop; i++)
      for (j = 0; j < cols; j++)
        if ((window[i][j] < 12)&&(window[i][j] > -1))
          fp.write(color, window[i][j] * 3, 3);
        else
          fp.write(color, 36, 3);

    fp.close();
    return;
   }
   catch (IOException io)
   {
    return;
   }
  }

  public void run()
  {
   System.out.println("MandelOne: Starting Fractal");
   mandelbrot();
   bitmapit();
   System.out.println("MandelOne: Finished Fractal");
  }
};
```

C.5 MandelTwo Class

```
import java.lang.*;
import java.io.*;

public class MandelTwo extends Thread
{
  public int cols = 1024;
  public int rows = 768;
  public int BI_RGB = 0;
  private double xmin = -1.75;
  private double xmax = 1;
  private double ymin = -1.15;
  private double ymax = 1.15;
  private int iter = 2000;
  private int radius = 2;
  private int[][] window = new int[rows][cols];
```

```
private int temp[] = { 0, 0, 0, 255, 0, 0, 0, 255, 0,
  0, 0, 255, 255, 0, 255, 0, 255, 255, 255, 255,
  0, 188, 143, 143, 230, 232, 250, 79, 47, 79,
  205, 205, 205, 112, 219, 147, 255, 255, 255 };
private byte color[] = new byte[39];

MandelTwo()
{
  int i, j;

  for (i = 0; i < 39; i++)
    color[i] = (byte)temp[i];

  for(i = 0; i < rows; i++)
    for(j = 0; j < cols; j++)
      window[i][j] = 0;
}

public void revwrite(FileOutputStream fp, int ptr,
  int bytes) throws IOException
{
  int i, d;
  byte[] tmp = new byte[4];

  d = ptr;
  for (i = 0; i < bytes; i++)
  {
    tmp[i] = (byte)(d % 256);
    d = d >> 8;
  }
  fp.write(tmp, 0, bytes);
  return;
}

public void revwrite(FileOutputStream fp, short ptr,
  int bytes) throws IOException
{
  int d;
  int i;
  byte[] tmp = new byte[4];

  d = (int)ptr;
  for (i = 0; i < bytes; i++)
  {
    tmp[i] = (byte)(d % 256);
    d = d / 256;
```

```
  }
  fp.write(tmp, 0, bytes);
  return;
}

public void mandelbrot()
{
  double dx;
  double dy;
  double x, y;
  double cr, ci;
  double zr, zi;
  double zsqr, zsqi;
  double radsqrd = radius * radius;
  int j, p, i, top, bottom;

  dx = (xmax - xmin) / cols;
  dy = (ymax - ymin) / rows;
  x = xmin - dx / 2;
  bottom = rows / 3;
  top = rows / 3 * 2;
  for (j = 0; j < cols; j++)
  {
    x = x + dx;
    y = ymin - dy / 2;

    for (i = bottom; i < top; i++)
    {
      y = y + dy;
      cr = x;
      ci = y;
      zi = zr = 0;

      p = 0;
      while ((p < iter)&&(zr*zr + zi*zi < radsqrd))
      {
        p++;
        zsqr = zr * zr - zi * zi;
        zsqi = 2 * zr * zi;
        zr = zsqr + cr;
        zi = zsqi + ci;
      }

      if (p == iter)
        window[i][j] = 0;
      else
```

```
        window[i][j] = (p / 100) + 1;
      }
    }
  }

  public void bitmapit()
  {
    BitMapFileHeader bmfh = new BitMapFileHeader();
    BitMapInfoHeader bmih = new BitMapInfoHeader();
    int i, j, bottom, top;
    FileOutputStream fp;

    try
    {
      fp = new FileOutputStream("Mandel2.bmp");

      bottom = rows / 3;
      top = rows / 3 * 2;
      for (i = top - 1; i >= bottom; i--)
        for (j = 0; j < cols; j++)
          if ((window[i][j] < 12)&&(window[i][j] > -1))
            fp.write(color, window[i][j] * 3, 3);
          else
            fp.write(color, 36, 3);

      fp.close();
      return;
    }
    catch (IOException io)
    {
      return;
    }
  }

  public void run()
  {
    System.out.println("MandelTwo: Starting Fractal");
    mandelbrot();
    bitmapit();
    System.out.println("MandelTwo: Finished Fractal");
  }
};
```

C.6 MandelThree Class

```
import java.lang.*;
import java.io.*;

public class MandelThree extends Thread
{
  public int cols = 1024;
  public int rows = 768;
  public int BI_RGB = 0;
  private double xmin = -1.75;
  private double xmax = 1;
  private double ymin = -1.15;
  private double ymax = 1.15;
  private int iter = 2000;
  private int radius = 2;
  private int[][] window = new int[rows][cols];
  private int temp[] = { 0, 0, 0, 255, 0, 0, 0, 255, 0,
    0, 0, 255, 255, 0, 255, 0, 255, 255, 255, 255,
    0, 188, 143, 143, 230, 232, 250, 79, 47, 79,
    205, 205, 205, 112, 219, 147, 255, 255, 255 };
  private byte color[] = new byte[39];

  MandelThree()
  {
    int i, j;

    for (i = 0; i < 39; i++)
      color[i] = (byte)temp[i];

    for(i = 0; i < rows; i++)
      for(j = 0; j < cols; j++)
        window[i][j] = 0;
  }

  public void revwrite(FileOutputStream fp, int ptr,
    int bytes) throws IOException
  {
    int i, d;
    byte[] tmp = new byte[4];

    d = ptr;
    for (i = 0; i < bytes; i++)
    {
      tmp[i] = (byte)(d % 256);
      d = d >> 8;
```

```
  }
  fp.write(tmp, 0, bytes);
  return;
}

public void revwrite(FileOutputStream fp, short ptr,
  int bytes) throws IOException
{
  int d;
  int i;
  byte[] tmp = new byte[4];

  d = (int)ptr;
  for (i = 0; i < bytes; i++)
  {
    tmp[i] = (byte)(d % 256);
    d = d / 256;
  }
  fp.write(tmp, 0, bytes);
  return;
}

public void mandelbrot()
{
  double dx;
  double dy;
  double x, y;
  double cr, ci;
  double zr, zi;
  double zsqr, zsqi;
  double radsqrd = radius * radius;
  int j, p, i, top, bottom;

  dx = (xmax - xmin) / cols;
  dy = (ymax - ymin) / rows;
  x = xmin - dx / 2;
  bottom = rows / 3 * 2;
  top = rows;
  for (j = 0; j < cols; j++)
  {
    x = x + dx;
    y = ymin - dy / 2 + ((ymax - ymin) / 3);

    for (i = bottom; i < top; i++)
    {
      y = y + dy;
```

```
      cr = x;
      ci = y;
      zi = zr = 0;

      p = 0;
      while ((p < iter)&&(zr*zr + zi*zi < radsqrd))
      {
        p++;
        zsqr = zr * zr - zi * zi;
        zsqi = 2 * zr * zi;
        zr = zsqr + cr;
        zi = zsqi + ci;
      }

      if (p == iter)
        window[i][j] = 0;
      else
        window[i][j] = (p / 100) + 1;
    }
  }
}

public void bitmapit()
{
  BitMapFileHeader bmfh = new BitMapFileHeader();
  BitMapInfoHeader bmih = new BitMapInfoHeader();
  int i, j, bottom, top;
  FileOutputStream fp;

  try
  {
    fp = new FileOutputStream("Mandel3.bmp");

    bottom = rows / 3 * 2;
    top = rows;
    for (i = top - 1; i >= bottom; i--)
      for (j = 0; j < cols; j++)
        if ((window[i][j] < 12)&&(window[i][j] > -1))
          fp.write(color, window[i][j] * 3, 3);
        else
          fp.write(color, 36, 3);

    fp.close();
    return;
  }
  catch (IOException io)
```

```
    {
      return;
    }
  }

  public void run()
  {
    System.out.println("MandelThree: Starting Fractal");
    mandelbrot();
    bitmapit();
    System.out.println("MandelThree: Finished Fractal");
  }
};
```

Bibliography

[1] Bal, Henri E., Bhoedjang, Raoul., Hofman, Rutger., Jacobs, Ceriel., Langendoen, Koen., Rühl, Tim., and Kaashoek, M. Frans. *Orca: a Portable User-Level Shared Object System.* Dept. of Mathematics and Computer Science, Vrije Universiteit, Amsterdam, The Netherlands. 1996.

[2] Bershad, Brian N., Zekauskas, Matthew J., and Sawdon, Wayne A. *The Midway Distributed Shared Memory System.* School of Computer Science, Carnegie Mellon University, USA, 1993.

[3] Eager, D. L., Lazowska, E. D., and Zahorjan, J. *Adaptive Load Sharing in Homogeneous Distributed Systems*. IEEE Transactions on Software Engineering. Vol SE-12, No 5., pp 662-675, 1986.

[4] Enslow, Philip H. *What is a "Distributed" Data Processing System?* Georgia Institute of Technology. IEEE Computer. Vol 11, pp 13-21, 1978.

[5] Falconer, K. J. *The Geometry of Fractal Sets.* Cambridge University Press, 1985.

[6] Ferrari, Adam J. *JPVM: Network Parallel Computing in Java.* Technical report CS-97-29, Department of Computer Science, University of Virginia, USA, 1997.

[7] Flanagan, David. *Java in a Nutshell.* O'Reilly & Associates, Inc., 1996.

[8] Geist, Al., Beguelin, Adam., Dongarra, Jack., Jiang, Weicheng., Manchek, Robert., and Sunderam, Vaidy. *PVM: Parallel Virtual Machine (A User's Guide and Tutorial for Networked Parallel Computing).* MIT Press, 1994.

[9] Johnson, Kirk L., Kaashoek, M. Frans, and Wallach, Deborah A. *CRL: High-Performance All-Software Distributed Shared Memory.* Operating Systems Review, Vol 29, No 5, pp 213-228, 1995.

[10] Leffler, S. J., Fabry, R. S., Joy, W. N., Lapsley, P., Miller, S., and Torek, C. *An Advanced 4.4BSD Interprocess Communication Tutorial.* Com-

puter Systems Research Group, Department of Electrical Engineering and Computer Science, University of California, Berkeley, 1986.

[11] Lindholm, Tim., and Yellin, Frank. *The Java Virtual Machine Specification.* Sun Microsystems Inc., 1997.

[12] Meyer, Jon., Downing, Troy. *Java Virtual Machine.* O'Reilly & Associates, Inc., 1997.

[13] Object Management Group. *CORBA: Common Object Request Broker Architecture.* http://www.omg.org, 1998.

[14] Object Management Group. *Java, RMI and CORBA.* http://www.omg.org/news/wpjava.htm, 1997.

[15] Peitgen, H.-O., and Richter, P. H. *The Beauty of Fractals: Images of Complex Dynamical Systems.* Springer-Verlag, 1986.

[16] Scales, Daniel J. and Gharachorloo, Kourosh. *Towards Transparent and Efficient Software Distributed Shared Memory.* Operating Systems Review, Vol 31, No 5, pp 157-169, 1997.

[17] Schneier, Bruce. *Applied Cryptography: Protocols, Algorithms, and source code in C.* John Wiley & Sons, Inc., 1994.

[18] Sechrest, Stuart. *An Introductory 4.4BSD Interprocess Communication Tutorial.* Computer Systems Research Group, Computer Science Division, Department of Electrical Engineering and Computer Science, University of California, Berkeley., 1988.

[19] Stauffer, D., and Stanley, H. E. *From Newton to Mandelbrot: A Primer in Theoretical Physics.* Springer-Verlag., 1985.

[20] Stevens, Richard W. *UNIX Network Programming.* Prentice Hall Inc., 1990.

[21] Sun Microsystems Inc. *Java IDL.* http://www.javasoft.com/products/jdk/1.2/docs/guide/idl/index.html, 1996.

[22] Sun Microsystems Inc. *Java Remote Method Invocation - Distributed Computing for Java.* http://www.javasoft.com/marketing/collateral/javarmi.html, 1998.

[23] Sun Microsystems Inc. *Java Remote Method Invocation Specification.* http://www.javasoft.com/products/jdk/1.2/docs/guide/rmi/spec/rmiTOC.doc.html, 1998.

[24] Thurman, Dave. *jPVM.* http://www.isye.gatech.edu/chmsr/jPVM, 1997.

[25] Venners, Bill. *Inside the Java Virtual Machine.* McGraw-Hill Inc., 1998.

VITA

Surname: Kent Given Names: Kenneth Blair
Place of Birth: Bell Island, Newfoundland Date of Birth: June 21, 1973

Educational Institutions Attended:

Memorial University of Newfoundland, 1991-1996.

University of Victoria, 1996 - 1999.

Degrees Awarded:

B.Sc (hons), Memorial University of Newfoundland, 1996.

Honours and Awards:

Dean's List, Memorial University of Newfoundland, 1996.

Government of Newfoundland Scholarship, 1996.

Publications:

Kenet: a Software Library for Designing and Testing Multicast Protocols. Hons. Dissertation, Memorial University of Newfoundland, 1996.