Modeling CPU Demand in Heterogeneous Active Networks

Virginie Galtier, Kevin Mills, and Yannick Carlinet
National Institute of Standards and Technology
Gaithersburg, MD USA 20899
kmills@nist.gov

Abstract

Active-network technology envisions deploying execution environments in network elements so that application-specific processing can be applied to network traffic. To provide safety and efficiency, individual nodes must include mechanisms to manage resource use. This implies nodes must understand resource demands associated with specific traffic. Well-accepted metrics exist for expressing bandwidth (bits per second) and memory (bytes) in units independent of particular nodes. Unfortunately, no well-accepted, platform-independent metric exists to express processing demands. This paper describes and evaluates an approach to model processing demand for active packets in a form interpretable among heterogeneous nodes in an active network. The paper applies the model in two applications: (1) controlling CPU use and (2) predicting CPU demand. The model yields improved performance when compared against the approach currently used in many execution environments. The paper also discusses the limits of the proposed model, and outlines future research that might lead to improved outcomes.

1. Introduction

In classical packet-switched communication networks, when a packet transits through an intermediate node along the path from source to destination, the intermediate node examines the destination address, consults a routing table for the next hop, and then forwards the packet on an appropriate link. The data transported within the packet remain opaque to the node. Since each intermediate node has a measured rating for per-message and per-byte throughput, a linear extrapolation from packet size and arrival rate should provide the node a reasonable estimate for the processor (typically called central-processing unit, or CPU) demand associated with individual packets or with sets of packets. Unfortunately, this simple approach cannot work for active networks because individual packets can require substantially different processing.

In active networks, when a packet arrives at an intermediate node, the data may include program code that can be accessed, interpreted, and executed by the node. This code describes how to process the packet, and perhaps subsequent, related packets. For instance, the code may specify a compression algorithm to be applied on the data if congestion has been detected in the area of the node, or may specify which packets to drop first, or may modify the destination address to route around congestion. This implies that identical packets can require different CPU time on assorted nodes and under various conditions. Thus, in active networks, a more sophisticated technique is needed to estimate CPU demand associated with active packets.

Inability to estimate the CPU demands of active packets can lead to some significant problems. First, a maliciously or erroneously programmed active packet might consume excessive CPU time at a node, causing the node to deny services to valid active packets. Alternatively, a node might terminate a valid active packet prematurely, wasting the CPU time used prior to termination, and ultimately denying service to a correctly programmed application. Second, an active node may be unable to schedule CPU resources to meet the performance requirements of packets. Third, an active packet may be unable to discover a path that can meet its performance requirements. This path selection problem occurs in part due to the node-scheduling problem, but also because the CPU time commitments of active nodes along a path cannot be determined. Devising a method for active packets to specify their CPU demands can help to resolve these problems, and can open up some new areas of research. Unfortunately, there exists no well-accepted metric for expressing CPU demands in a platform-independent form. This is the problem that motivated our research.

In Section 2, we discuss the problem in more detail, and we identify the outlines for a solution. In Section 3, we provide an overview and critique of some existing approaches to control CPU use in active applications.

Further, we examine some ideas from the literature that stimulated our thinking. In Section 4, we describe a statistical black-box model for specifying CPU demand associated with active packets, and we show how we can generate such models by tracing packet executions. In addition, we compare estimates from our models against real executions. In Section 5, we outline our strategy for calibrating active network nodes and for transforming CPU models for active packets among heterogeneous nodes in an active network. We also compare predictions made by our transformed models against measured executions on a variety of nodes in an active network topology. In Section 6, we show how our CPU models can be applied in two sample applications. One application controls CPU usage by active packets, where our models achieve improved performance over one of the existing techniques implemented within active-network nodes. The second application predicts CPU demand by active packets. Here, our models outperform estimators based on one of the simplest techniques used by a number of active-network execution environments. In Section 7, we discuss the limits of our current approach, and we suggest some future research that might yield improved outcomes. We close in Section 8 with our conclusions.

2. The Problem and Outlines of a Solution

The growing ubiquity of the Internet is changing the nature of software design and deployment. Increasingly, Internet-based system architectures employ distributed components and use mobile code, such as applets, scripts, servlets, and dynamically linked libraries, to deliver new software to millions of users. Absent an understanding of the processor (CPU) time required by such dynamically injected software, computer operating systems cannot effectively manage system resources or control the execution of mobile code. Unfortunately, since mobile code can be injected and executed on a variety of computer platforms with a wide range of capabilities, software developers cannot precisely specify CPU requirements a priori. We set out to improve the ability of software developers to quantify CPU time requirements of mobile code in a form that can be understood readily on heterogeneous computing platforms.

We conducted our research in the context of active networks, an emerging technology that exploits mobile code in an extreme form. Active-network technology augments traditional networking with the possibility that individual packets carry executable code, or references to executable code. Conventional (data-only) packets are forwarded on the so-called "fast path" of a router, while active packets, which invoke mobile code, are delivered to a higher-level execution environment that can identify and run a program specifically associated with the packet. Networking applications built with active packets are referred to as active applications. Figure 1 illustrates the architecture of an active-network node [1].

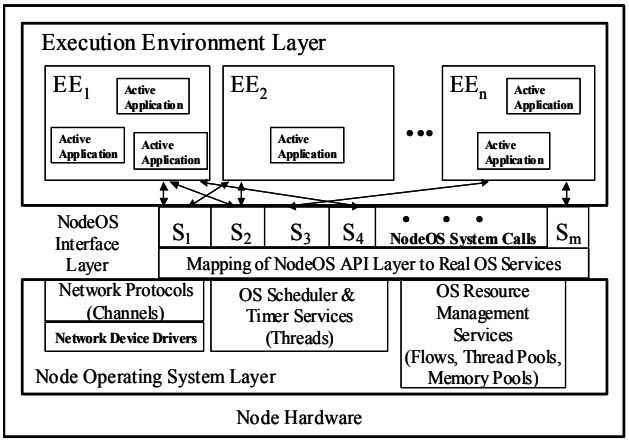

Fig. 1. Schematic representation of the active-network architecture, revealing five levels of abstraction, from bottom to top: (1) node hardware, (2) node operating system, (3) node operating system interface, (4) execution environment, and (5) active application.

Underlying each active-network node is a node operating system, which transforms the node hardware into a software abstraction that provides execution environments with controlled access to resources such as CPU cycles, memory, input and output channels, and timers. In order to allow many possible operating systems to provide services to many possible execution environments, the active-network node architecture includes a standard specification of system calls (the Node OS Interface Layer in Figure 1) [2]. Execution environments, similar to virtual machines, can be loaded onto an active node using ANETD [3], a daemon that implements a load-and-go protocol for execution environments. Each execution environment accepts active packets that can initiate the execution of packet-specific code. Each related code base and flow of active packets is known as an active application. During the course of the Active-Networks research program, funded by DARPA, researchers developed a number of node operating systems [4-7, 29-33], execution environments [8-12, 33, 34] and active applications [13-19].

While innovative and radical when considered for use inside networks, active-network execution environments share much in common with virtual machines used in Internet-based software architectures, and active applications appear quite similar to other forms of dynamically injected software, such as applets, mobile-agent scripts, and dynamically linked libraries.

These similarities encourage us to believe our ideas apply generally to the problem of specifying CPU demand in distributed applications that rely on the use of mobile code.

2.1. Analysis of Variability in CPU Demand

The amount of CPU time required by a computer program is a function of two factors: (1) the speed of the processor on which the program will execute and (2) the number of CPU cycles required for the program to complete its task. The first of these factors proves easy to measure on any computer platform because a computer operating system can readily determine the speed of the processor on which it executes. In general, processor speeds are specified in cycles per second, or Hertz, where the time taken to execute a single CPU cycle can be represented as the inverse of the processor speed. For example, a processor that operates at the rate of one billion Hertz (a Gigahertz, or GHz) will execute a single CPU cycle in one billionth of a second (a nanosecond, or ns). Unfortunately, the second factor, the number of CPU cycles required for a program to complete a task, proves very difficult to determine in any platform-independent manner. Below we consider some of the difficulties associated with providing an accurate measure of the count of CPU cycles required by a program.

As shown in Figure 1, a mobile program, or active application, executes at the highest level in a five-level architecture of abstractions. At this highest level of abstraction, the number of instructions required to complete the program is a function of the paths taken through the code, which can depend on various conditions that exist on the node at the time the program executes. The lowest level of abstraction consists of the node hardware, where program execution time is influenced directly by the raw speed of various components, such as the operating rate of the processor. Three additional levels of abstraction exist between the active application and the node hardware. Each of these levels of abstraction consists of its own set of computer code, and therefore CPU cycles, which may be traversed by specific executions of an active application. For example, consider the node operating system layer in Figure 1.

When an active application calls for a system service, such as a read or write to a disk device, control passes to a device driver that executes some CPU cycles. The number of CPU cycles required depends upon the specific device driver underlying the system call, and the device driver typically depends upon the specific hardware. So, for example, if the disk is accessed through a SCSI (small-computer system interface) controller, a particular device driver will be used, while if the disk is accessed through an IDE (integrated drive electronics) controller, then a different device driver is required. Further, from time-to-time device manufactures update their device drivers. This implies that the number of CPU cycles required to access a device can also vary based on the specific version of the device driver loaded on the computer platform. A similar analysis applies to other devices, such as network interface cards, codecs (encoder-decoders), and encryption hardware.

Similar reasoning applies at the other layers of abstraction. For example, the active-networks architecture (Figure 1) defines a standard node operating system interface, which permits any execution environment to run on any node operating system. This implies that some mapping may be required to enable various node operating systems to provide the standard interface. Each such mapping will introduce additional CPU cycles into the system calls made by an active application. The number of additional CPU cycles will depend upon the specific mapping code. Along similar lines, some code will be required to map an execution environment onto the standard interface provided by node operating systems. This mapping will introduce additional code, and therefore CPU cycles, that must be executed on behalf of the active application. The specific mapping will likely vary for each execution environment.

2.2. Some Supporting Measures

Above we argued that the processing demand of a mobile program, such as an active application, depends upon two factors: processor speed and number of CPU cycles that must be executed. Further, we suggested that the number of CPU cycles needed depends on a variety of factors that will vary as a program moves from node-to-node. In this section, we support our theoretical discussion with some concrete evidence obtained by measuring the operation of several computer platforms, described in Table 1.

To investigate our hypothesis that the same program will require different numbers of CPU cycles to execute on various computer platforms, we ran a small Java™[1] benchmark program on the three computers outlined in Table 1. The benchmark program simply makes a series of 10,000 repetitive invocations of various system calls through the Java virtual machine to the operating system.

In order to push our analysis to a limit, we installed the same versions of Linux (version 2.2.7) and the Java development kit (jdk 1.1.6) on three Pentium-based platforms. The three nodes differed only in the processor

[1] Commercial products are identified in this report to describe our study adequately. Such identification does not imply recommendation or endorsement by the National Institute of Standards and Technology.

architecture and speed, the amount of installed memory, and the characteristics of interface hardware, such as disk controllers, audio devices, and network cards. Using these three platforms, we investigated only the differences in CPU cycles that result from variations in node hardware, the lowest level of abstraction shown in Figure 1. Table 2 depicts the outcome from running our benchmark program repeatedly and averaging the results.

	Platform Description		
Node Name	Green	Black	Blue
Processor Speed (MHz)	199	333	450
Processor Architecture	Pentium Pro	Pentium II	Pentium II
Memory Size (Megabytes)	64	128	128
Operating System/Version	Linux / 2.2.7	Linux / 2.2.7	Linux / 2.2.7
Java Virtual Machine/Version	jdk / 1.1.6	jdk / 1.1.6	jdk / 1.1.6

Table 1. Characteristics of three computer platforms selected for investigation by measurements.

	Computer Node		
Java Benchmark Results	Green	Black	Blue
Average Number of CPU Cycles	167,830	159,412	240,269
Average CPU Time (microseconds)	843	479	534

Table 2. Results from executing the same Java benchmark program on three different Pentium-based platforms running Linux.

The results shown in Table 2 support our analysis that different computer platforms can require significantly varying numbers of CPU cycles to execute the same program, even when using identical versions of an operating system and virtual machine. In fact, as Table 2 illustrates, variation in the number of CPU cycles required may even lead to cases where a program takes longer to execute on a faster machine (Blue) than on a slower machine (Black). Our earlier analysis suggested that such results might be attributed to differences in the device drivers underlying the operating-system calls invoked by our benchmark program. Results given in Table 3 support the earlier analysis.

Table 3 shows that for four system calls (included in our benchmark) the slowest node (Green – 199 MHz) requires the fewest number of CPU cycles, while the fastest node (Blue – 450 MHz) requires the largest number of CPU cycles. The intermediate node (Black – 333 MHz) requires fewer CPU cycles than Blue but more than Green. As a result, Black executes the benchmark faster than either Green or Blue, while Green executes the benchmark only slightly slower than Blue, but certainly not twice as slow.

These results support our assertion that the CPU time required by a mobile program depends upon both the processor speed and the number of CPU cycles required to execute the program on a given computer platform. Since a platform operating system can easily determine processor speed, the main problem for modeling CPU demand in a mobile program, such as an active application, is to express the number of CPU cycles required to execute the program on a given platform. This is the problem that we attempted to solve.

	Computer Node					
	Green		Black		Blue	
System Call	ACC	ACT	ACC	ACT	ACC	ACT
Read	12,606	63	12,362	37	19,321	43
Socket Call	14,560	73	17,591	53	27,066	60
Stat	12,042	61	14,731	44	22,800	51
Write	12,362	62	14,394	43	22,609	50

Table 3. Results for selected system calls from executing the same Java benchmark program on three different Pentium-based platforms running the same version of Linux and identical versions of the Java development kit. Results measure average CPU cycles (ACC) and average CPU time (ACT) used to execute selected system calls on each node.

2.3. The Outlines of a Solution

The outlines of a solution seem clear. The number of CPU cycles needed to execute an active application on a particular platform depend upon: (1) the path taken through the application code, (2) the path taken through the execution environment, (3) the path taken through the mapping between the execution environment and the node operating system interface, and (4) the path taken through the system calls (and related device drivers) in the node operating system. Unfortunately, these factors can vary from platform-to-platform, based on the specific code implemented on each platform, and from node-to-node, based on various node-dependent conditions that exist at the time an active packet arrives. Regarding node-dependent conditions, two aspects seem relevant: (1) conditions at the node that affect the processing logic in an active application and (2) conditions at the node that affect resource sharing among multiple active applications and execution environments.

Any effective model of CPU demand by a mobile program, which we call an active-application model, seems likely to require delineating the processing paths through the program in terms of elements of a platform-independent abstraction that the program will invoke on

every node. We refer to such platform-independent abstractions as node models. In the context of active networks, two types of node model seem feasible: (1) white-box models and (2) black-box models. White-box models represent the functions offered to active applications by a specific execution environment. Black-box models hide execution-environment functions, and represent instead system calls offered to the execution environment by a standard node operating system interface. So, in a white-box model the execution environment is transparent, while in a black-box model the execution environment is opaque. While we are investigating both approaches, in this paper we focus mainly on a black-box model because, if successful, such models can work across the full range of execution environments being developed by active-network researchers. White-box models, on the other hand, must be developed for each execution environment that a node intends to support.

Whether specifying the logic of an active application in terms of a black-box or white-box node model, a means is also needed to characterize the performance of specific nodes with respect to the model elements. We refer to this aspect of the solution as node calibration. The main idea behind node calibration is to determine a node's performance in implementing the elements that compose the node model.

Through calibration, a node operating system can determine how many CPU cycles are required to execute each model element on the node. Then, given any active-application model expressed in terms of the elements of a node model, an active-network platform should be able to estimate the number of CPU cycles that the active application will require on the node.

Unfortunately, the processing logic in an active application does not consist solely of calls to elements in the node model. Instead, the active application also includes its own processing logic that is not carried out by functions in the execution environment or by system calls. This suggests that an active-application model must also express the number of CPU cycles used between calls to elements in the node model. As a result, some means is required to calibrate the performance of the active application on every node in the network. Such exhaustive calibration appears infeasible; however, the application-specific logic in an active application can certainly be measured on one node in the network. Such measurements provide an indication of the number of CPU cycles required by an active application on a specific node. Since an active application runs within an execution environment, we can imagine calibrating the ability of each execution environment to perform a representative workload on each node in the network. We call this process execution-environment calibration.

Given an active-application model expressed as a combination of: (1) elements of a node model and (2) the number of CPU cycles used between elements of a node model, a network node receiving such a model can conceivably transform the model into terms that might be meaningful on the node. The techniques for performing such a transformation make up another part of the solution.

To recap, the outlines of a solution to the problem of modeling CPU demand in mobile programs include at least the following components: (1) a node model expressed in terms of functions invoked by active applications, (2) an active-application model expressed in terms of paths through a node model and in terms of CPU cycles used between invocations of elements in the node model, (3) calibrations of a node with respect to the node model and the execution environments on the node, and (4) transformation techniques that can convert an active-application model sent between two nodes into terms meaningful on the destination. These are the portions of the solution that we investigated, and that we address in Sections 4 and 5 of this paper.

Other issues must also be resolved for a complete solution. For example, we have not tackled the problem of node-dependent conditions in this paper. This means that in our work the CPU demands of an active application are modeled from measuring the application in numerous scenarios in the development laboratory before we release the application (and its model) into a network. Should the application encounter conditions not seen in the development laboratory, our models have no means of adjusting to such new conditions. We have also not tackled the problem of adjusting node and execution-environment calibrations based on current conditions in a node or on new conditions that arise on a node over its lifetime. This implies our calibrations do not adapt to changes in a node or execution environment that arise after the calibration occurs. We discuss issues related to adaptation under future work.

3. A Critique of Selected Approaches

While the outlines of our solution appear complex, we believe that success along these lines will enable more effective control of CPU usage by mobile programs and will enable node operating systems to more efficiently manage CPU resources. Others also see a need to provide such capabilities. In this section we present and critique existing solutions to prevent excessive CPU resource consumption in active networks and in mobile-agent systems. Next we examine research conducted outside of active networks that could help to provide effective resource management in active-network nodes.

3.1. Existing Solutions to Control CPU Use

In order to prevent malicious or erroneous active packets from consuming excessive CPU time, most execution environments implement specific control mechanisms. In this section, we discuss the most common mechanisms (*a-d* below) and give our critique.

(a) Use a limit fixed by the packet. Some execution environments, such as ANTS [8], assign a time-to-live (TTL) to each active packet. An active node decreases this TTL as a packet transits the node, or whenever the node creates a new packet. In this way, each active packet can only consume resources on a limited number of nodes, but individual nodes receive no protection. The current TTL recommendation for the Internet protocol (IP) is 64 hops [20], which is supposed to roughly correspond to the maximum diameter of the Internet. Current implementations of Windows NT and Windows 2000 use 128 hops as the TTL in IP packets. This value might prove large enough for an active packet that propagates a configuration from node to node between two videoconferencing machines. But if the active packet creates numerous additional packets (to which it delegates a part of its own TTL), then the assigned TTL could prove insufficient. And it is usually difficult to predict how many new packets will be generated since these predictions might depend on network parameters, such as congestion and topology, which can rarely be known in advance. This TTL mechanism could contribute to protect individual nodes if the TTL is given in CPU time units instead of hop count. But the problem remains: how to choose the initial value for the TTL?

In the related context of mobile agents, Huber and Toutain [21] propose to enable packets that did not complete their "mission" to request additional credits. The decision to grant more credit would be taken by the originating node for its packets, or by the generating packet for packets created while moving among nodes. The decision must be made after examining a mission report included with the request for more credits. The proposed solution remains unimplemented, perhaps because the reports proved difficult to generate and evaluate. Even if implemented, a malicious application can be conceived, where the originating node will always grant more credits to any of its packets.

(b) Use a limit fixed by the node. In some execution environments (e.g., ANTS), a node limits the amount of CPU time any one packet can use. This solution protects the node but does not allow optimal management of resources. For instance, imagine that a node limits each packet to 10 CPU time units. Suppose that a packet requiring 11 CPU time units arrives when the node is not busy. In this case, the node will stop the execution of the packet just before it completes.

(c) Use a restricted language. The SNAP language [22] is designed with limited expressiveness so that a SNAP program uses CPU in linear proportion to the packet's length. While this approach supports effective management of resource usage, it could prove too restrictive for expressing arbitrary processing in active applications. For instance, only forward branches are allowed; as a result, if repetitive processing is required, the packet must be resent repeatedly in loop-back mode until the task is completed.

(d) Use a market-based approach. Yamamoto and Leduc [23] describe a model for trading resources inside an active-network node, based on the interaction between a "reactive user agent" included in the packet, and resource manager agents that reside in the network nodes. The manager agents propose resources (such as link bandwidth, memory, or CPU cycles) to the user agents at a price that varies as a function of the demand for the resource (the higher the demand, the higher the price). Packets carry a budget that allows them to afford resources on active nodes. Based on the posted price of the resources and on its remaining credit, the user agent of a packet makes decisions about the processing to apply. For instance, if the CPU is in high demand and thus expensive to use, then a packet may decide to apply a simple compression algorithm to its data, instead of a more efficient but more costly algorithm, which the packet would have applied if the resource were more affordable. This approach, which might prove appropriate for mobile-agent platforms, could increase the packet complexity too much to be used efficiently in active networks.

Our critique. The two most common approaches to resource control in active networks apply a fixed limit on the CPU time allocated to an active packet. In one approach, each node applies its own limit to each packet, while in the other approach each packet carries its own limit, a limit that might prove insufficient on some nodes a packet encounters and overly generous on other nodes. Neither approach provides a means to establish an appropriate limit for a variety of active packets, executing on a variety of nodes. Our research aims to solve this problem, while at the same time we intend to develop a solution that does not reduce the expressiveness of an active packet, nor make a packet too complex.

3.2. Attempts to Quantity CPU Demand

While we are unaware of any other projects aiming to quantify the CPU requirements of an active application in a heterogeneous network, we did survey several related research initiatives that could help us to devise an effective solution. The following sections (*a-d* below) outline and discuss some of the ideas we found.

(a) Use RISC cycles. The active-network architecture documents specify that a node is responsible to allocate and schedule its resources, and more particularly CPU time. Calvert [1] emphasizes the need to quantify the processing demands of an active application in a context where such demands can vary greatly from one node to another, and he suggests using RISC (Reduced Instruction Set Computer) cycles as a unit to express processing demands. He does not address two crucial questions. First, for a given active application, how can a programmer evaluate the number of RISC cycles required to execute a packet on a given node? Second, how can this number be converted into a meaningful unit for non-RISC machines?

(b) Use extra information provided by the programmer. In the AppLeS (application-level scheduling) project [24], the programmer provides information about the application that she wishes to execute on a distributed system. She must indicate for instance whether the application is more communication-oriented or computation-oriented or balanced, the type of communication (e.g., multicast or point-to-point), and the number of floating-point operations (in millions) performed on each data structure. Using this information, a scheduling program produces a schedule expected to lead to the best performance for the application. This method can yield acceptable predictions only if the programmer is both willing and able to provide the required characteristics of the program. Discussions with software performance experts led us to think this is rarely the case.

(c) Use combined node-program characterization. Saavedra-Barrera and colleagues [25] attempted to predict the execution time of a given program on various computers. To describe a specific computer, they used a vector to indicate the CPU time needed to execute 102 well-defined Fortran operations. In addition, they provided a means to analyze a Fortran program, reducing it to the set of well-defined operations. The program execution time can then be predicted by combining the computer model with the program model. The approach yielded good results for predicting the CPU time needed to execute one specific run of a program on different computer nodes. These results encouraged us to model platforms separately from applications; however, we need to capture multiple execution paths through each application, rather than a single path. We are pursuing a separate thread of research, discussed under future work, which aims to apply insights from Saavedra-Barrera to the active-network environment.

(d) Use acyclic path models. To measure, explain, or improve program performance, a common technique is to collect profile information summarizing how many times each instruction was executed during a run. Compact and inexpensive to collect, this information can be used to identify frequently executed code portions. Unfortunately, such profiles provide no detail on the dynamic behavior of the program (for instance, these techniques do not capture and report iterations). To solve this problem a detailed execution trace must be produced, listing all instructions as they are executed. But as program runs become longer, the trace becomes larger and more difficult to manipulate. Ball and Larus [26] propose an intermediate solution: to list only loop-free paths, along with their number of occurrences. Among other things, the authors demonstrate how the use of these acyclic paths can improve the performance of branch predictors. We might be able to exploit such algorithms to efficiently capture looping behaviors; however, to collect acyclic path information we would need to instrument the program code for each application to be modeled. Given the variety of execution environments and active applications being devised by researchers, we decided to first evaluate some simpler approaches.

4. A Black-Box Model of CPU Demand

Recall from Figure 1 that an active application executes in user mode within an execution environment, but requests services periodically from the node operating system through specific system calls. An observer, situated at the boundary between an execution environment and a node operating system, would view the behavior of an active application as a series of transitions between specific system calls: from an idle state, the application executes in user mode for some number of CPU cycles within its execution environment and then executes in kernel mode for some number of CPU cycles within a system call, then again in the execution environment before transitioning to another system call, and so on until the active packet is processed and the active application has returned to the idle state. Because the point of observation provides no insight into the logic of the active application or the execution environment, we can consider them to be a black box. We denote each observed transition-sequence as a black-box execution trace.

From a collection of execution traces, we can cluster together those that exhibit an identical sequence of system calls. We call each cluster a scenario. For example, Figure 2 depicts two scenarios discovered in an execution trace. The shorter scenario occurs 2/3 of the time, while the longer scenario occurs 1/3 of the time.

A black-box model for an active application consists of two types of information: scenario specifications and workload specifications. In the model, each scenario is specified by its sequence of system calls. Further, each system call is characterized by the distribution of the

number of CPU cycles spent in the system call, and each transition between system calls is characterized by the distribution of CPU cycles spent in the execution environment during similar transitions. The CPU cycle estimates are derived from an analysis of the same execution traces used to identify scenarios. In an earlier version of our model, we hoped to represent these distributions using classical probability distributions. Our goal was to produce an analytically tractable model. Unfortunately, the observed distributions exhibit a degree of discreteness and truncation not well represented by typical continuous distributions. For this reason, we chose to represent the distributions of CPU cycle estimates with histograms (see Figure 3). Note that this approach can require exchanging a large volume of information when active-application models are transferred among nodes in a network. (However, our experiments show that reasonably accurate results could be obtained with as few as five bins per histogram.)

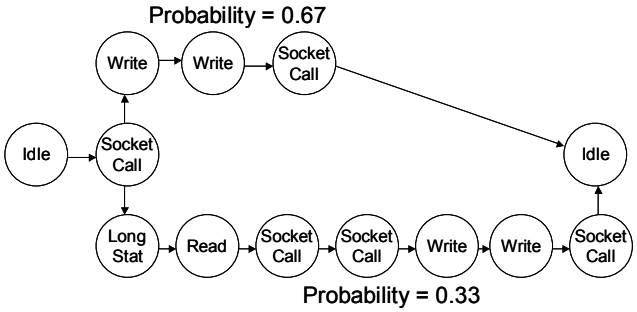

Fig. 2. An example of two scenarios discovered by clustering execution traces from a simple active application. The shorter trace occurs more frequently (probability is 0.67), while the longer trace occurs less frequently (probability is 0.33).

The remaining part of our model, the workload specification for an active application, consists of a list of the discovered scenarios, where each scenario is assigned a probability of occurrence (based on the frequency with which the scenario appeared in the execution trace). Taken together, the scenarios, their probability of occurrence, and the distributions of CPU cycles in user and system modes constitute a black-box model of CPU demand for an active application.

4.1. Generating Execution Traces

We based our models on measurements taken from execution traces. For various reasons, mainly arising from the relationship between Linux threads and multi-threading in the Java Virtual Machine, we could not use existing execution tracing programs available for typical operating systems, such as the Linux systems we used as our test platforms. (For more information on these issues, and on the tracing methods we considered, see a related technical report [35].) Instead, we designed our own kernel modifications to provide exactly the traces we needed, and at the granularity of individual CPU cycles. Table 4 provides an example trace from one execution cycle in one active application.

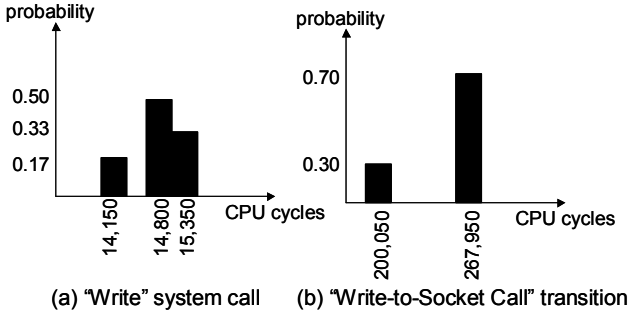

Fig. 3. Two example specifications of CPU cycle usage: (a) distribution of CPU cycles used by an active application in the "write" system call and (b) distribution of CPU cycles used by an active application in transitions between the "write" and "socket call" system calls. Each bin of a histogram is labeled with the mid-point of its value. The probabilities are the relative frequencies of observations falling within specific bins.

Transition		CPU Cycles Used	
Source	Sink	In Source	During Transition
Idle	Socket Call	0	17,703
Socket Call	Write	9,675	171,968
Write	Write	13,515	1,004,923
Write	Socket Call	13,247	1,057,165
Socket Call	Idle	9,569	1,056,127

Table 4. An example execution trace from idle-to-idle for a single path through an active application. Each row depicts a single instance of a transition between two system calls. The transition columns show the source and sink system calls for the transition. The first column of numbers counts the kernel-mode CPU cycles used in the source system call, while the second column of numbers counts the user-mode CPU cycles used in the execution environment between the return from the source system call and the beginning of the sink system call.

To generate execution traces of this granularity and accuracy, we used RDTSC (Read Time Stamp Counter),

a hardware instruction provided by Intel in their Pentium processors. This instruction records the number of CPU cycles used since the last reboot of the processor. The main difficulty we faced was attributing the use of CPU cycles to particular processes. To accomplish this, we designed and implemented modifications for insertion into the Linux scheduler.

To account for the number of CPU cycles spent by a process in user and in kernel modes, we added two fields to the process structure: "ucc" (number of CPU cycles spent in user mode) and "kcc" (for kernel mode). We also added two working fields: "e" to record the entry time into a new "state" (user or kernel) and "kflag" to indicate whether or not the process was sleeping in kernel mode when last exiting the scheduler. Indeed, with Linux, a process cannot be preempted while executing in the kernel. But when a process needs to wait for an event, it relinquishes the CPU for another process to run. This causes the waiting process to exit the scheduler. But it will exit the kernel only later, after having been rescheduled again and completing the execution of the suspended system call.

We used the following algorithm to update the “ucc” and “kcc” fields. On entering the scheduler: the “e” field of the entering process is set to RDTSC (the current value of the counter of CPU cycles since last reboot). On entering the kernel: the “kflag” is set and the value of “ucc” is updated: “ucc = ucc + RDTSC – e”, where “RDTSC – e” gives the number of clock cycles spent between the last time “e” was set (on entering the scheduler) and the current time. This represents time spent in user mode. Now the process is entering kernel mode, so “e” is set to RDTSC. On exiting the kernel the “kflag” is cleared and the value of “kcc” is updated: “kcc = kcc + RDTSC – e”. Then, “e” is set to RDTSC. On exiting the scheduler, if “kflag” is false, then “ucc = ucc + RDTSC – e”; otherwise “kcc = kcc + RDTSC – e”. Each time “ucc” or “kcc” are updated, their new value indicates how many CPU cycles the process has spent in user or kernel mode.

Our Linux kernel modifications enable us to trace a process in a very fine manner. Now we need to be able to retrieve the results. We found that the approaches typically used to capture trace information for Linux processes cause a traced process to run slowly, and also lead to inaccurate results at our required level of granularity. To avoid such problems, we implemented our own monitoring of the process to generate traces in a manner that was quite straightforward after the kernel modifications discussed previously. We simply print a message (using printk) at every entry and exit of the kernel. Of course, our tracing mechanism is not the only one using printk, so the resulting trace file must be pre-processed before analysis in order to extract only the trace lines of interest to us. To facilitate such pre-processing, we inserted tags into our trace lines to permit easy filtering.

4.2. Generating Models from Execution Traces

We wrote a model generator that can consume an execution trace and generate a black-box model for the program measured by the trace. First, the model generator clusters the traced executions into the scenarios contained, and assigns a probability of occurrence to each scenario. Then the model generator examines the CPU cycles used by each system call, and builds a corresponding histogram. Finally, the model generator examines the transitions between each pair of system calls and constructs a histogram describing the distribution of CPU cycles used. The model generator includes as an input parameter the number of bins to create in each histogram.

To generate estimates from a black-box model created by the model generator, we use Monte-Carlo simulation. Each pass through the simulator represents the processing of an active packet. Using the probability of occurrence contained in the black-box model, the simulator selects a scenario. For each component of the scenario (system calls and transitions in user mode between two system calls), the simulator runs another Monte-Carlo test to choose a bin of the histogram describing the count of CPU cycles. The sum of the CPU cycles of each component in the scenario yields a simulated number of CPU cycles, which can be easily converted into an estimate for CPU time by multiplying by the cycle time of the processor. After repeated scenario executions, we obtain a distribution of estimates for the CPU demands of the active application represented by the model. The distribution can be characterized with statistics, such as the mean or percentiles[2] (we used the 80^{th}, 85^{th}, 90^{th}, 95^{th}, and 99^{th}) of the CPU time demanded by the application. Of course, generating a large number of simulated executions can refine the estimates. Alternatively, selecting a small number of simulated executions can provide quick estimates.

4.3. Evaluating Models Against Measurements

To assess how well a particular model estimates the CPU demands of an active application, model predictions can be compared against measures for the relevant application. We conducted such measurements for numerous applications under a range of model conditions, including various bin granularities (from five

[2] Given a statistic, S, for a percentile, P, associated with a random variable, V, then P percent of the observed values for V will be less than S and 1-P percent will exceed S.

to 100) and simulation repetitions (from 100 to 20,000). Table 5 summarizes results comparing model predictions against measurements on various computing platforms for two execution environments, ANTS and Magician, and four active applications (two for each execution environment). The model histograms consist of five bins each. Each estimate is generated using 10,000 simulation iterations. Two comparisons are computed for each application: (1) error in predicting the mean and (2) average error in predicting the high percentiles (80^{th} through 95^{th}). Estimates for high percentiles can be useful in CPU control applications, while predictions of CPU demand can also benefit from estimates for the mean.

Execution Environment	Active Application	Node	% Error Mean	Average % Error High Percentiles
ANTS	Ping	Blue	0	7
		Green	1	6
		Black	1	5
	Multicast	Blue	11	10
		Green	11	9
		Black	8	8
Magician	Smart Ping	Blue	1	22
		Green	1	35
		Black	1	7
	Smart Route	Blue	1	30
		Green	1	46
		Black	0	8

Table 5. Comparing the percentage error in CPU demand between estimates from a black-box model and measurements of real applications for selected computer platforms, execution environments, and active applications. The black-box model was generated with five bins per histogram. Model estimates consist of 10,000 simulated executions. Percentage error computed as the absolute value of 100 * (prediction – actual) / actual. The errors for the high percentiles are averaged over the 80^{th}, 85^{th}, 90^{th}, 95^{th}, and 99^{th} percentiles.

Table 5 indicates very accurate predictions for mean CPU demand on most platforms, across execution environments and active applications. The prediction errors for mean CPU demand in ANTS Multicast are somewhat higher. Further, the predictions of high percentiles are less accurate than predictions for the mean, which can be expected because high percentiles represent extreme values that might not appear with great frequency. Still, for high percentiles, the predictions for the Magician execution environment appear significantly worse than the predictions for the ANTS execution environment.

The predictions and measurements compared in Table 5 consider each node running a mix of scenarios in all roles that a node might take on for an application. For example, in each application a node may serve as a source, a router, or a sink for active packets associated with the application. When we make comparisons between predictions and measurements while limiting a Magician node to hold one role (either source, router, or sink) for an active application, the predictions compare much more favorably with the measurements. Table 6 illustrates this point for three Magician applications: Smart Ping, Smart Route, and Active Audio. For the Active Audio application, where measurements were taken in the process of some sample applications (see section 6), each node assumed only one role.

Application Role ->		Source		Router		Sink	
Active Application	Node	% Error Mean	Avg. % Error High Perc.	% Error Mean	Avg. % Error High Perc.	% Error Mean	Avg. % Error High Prec.
Smart Ping	Blue	0	1	1	3	0	4
	Green	0	1	0	3	0	2
	Black	0	3	3	4	0	4
Smart Route	Blue	0	3	0	3	0	1
	Green	0	2	0	1	0	1
	Black	0	3	3	4	1	1
Active Audio	Blue	N/A	N/A	N/A	N/A	0	4
	Green	2	6	N/A	N/A	N/A	N/A
	Black	N/A	N/A	3	7	N/A	N/A

Table 6. Comparing the percentage error in CPU demand between estimates from a black-box model and measurements of real active applications for the Magician execution environment. In this case, predictions and measurements were compared when the role of each participant was restricted to that of source, router, or destination.

5. Transforming CPU Models

While the predictions made by our black-box models appear reasonably accurate in many situations, the more difficult part of our problem must still be solved. Particularly, given a model for the CPU demand of an active application running on one node, e.g., Green, can the model provide accurate estimates for the CPU demand of the application running on a different node, e.g., Black? To achieve this goal, we must transform the model generated on Green into a form that will be meaningful on Black. In this section, we address our approach to model transformation. First, we describe our model transformation algorithm. Second, we discuss our technique to calibrate nodes and execution environments. Finally, we evaluate how well our transformation technique works in a variety of tests.

5.1. Model Transformation Algorithm

Recall that our black-box model of an active application consists of two parts: scenario specifications and a workload specification. The workload specification assigns a probability of occurrence to each scenario in the model. We consider this information to be fixed on each node that encounters the model (though see Section 7 for a discussion of the need to adapt the workload specification). Each scenario specification delineates a sequence of transitions between system calls, where the number of CPU cycles consumed in each system call and in each transition is defined by histograms. The information in these histograms is based on measurements taken on a particular node; thus, this information will be meaningless on other nodes. The goal of our transformation algorithm is to convert the information in the histograms into a form meaningful on any node that receives the active-application model. Figure 4 shows the results of transforming the histogram discussed earlier in Figure 3.

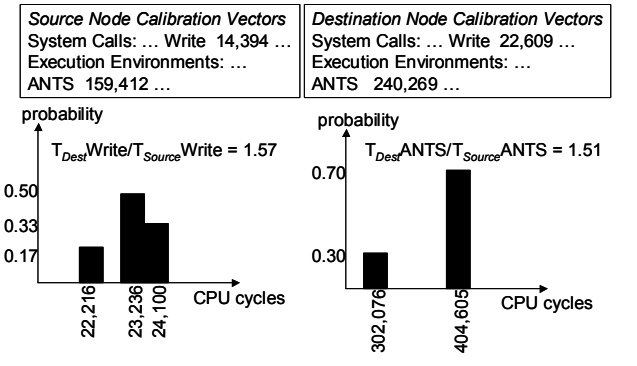

Fig. 4. This figure shows two histograms in an active-application model, shown earlier in Figure 3, after those histograms have been transformed from a form understood by the source node (Black) into a form meaningful on the destination node (Blue). The relevant parts of the calibration vectors are given for the source and destination nodes. Two scaling factors are computed: (1) 1.57 for the "Write" system call and (2) 1.51 for the "Write-to-Socket Call" transition.

We assume that each node has been calibrated with respect to its performance executing each system call and each execution environment. The calibration results are represented as two vectors. One vector, the system-call (SC) vector contains the average number of kernel-mode CPU cycles for the node to execute each system call. For example, Figure 4 shows the calibration for the "Write" system call on two nodes. A second vector, the execution-environment (EE) vector, contains the average number of user-mode CPU cycles for the node to execute a calibration benchmark for each execution environment that runs on the node. Figure 4 reveals the calibration information for the ANTS execution environment on two nodes.

For purposes of discussion, assume that a node (*Dest*) receiving an active-application model has access to its own calibration vectors as well as the calibration vectors of the source node (*Source*). Then, the destination can scale the contents of the histograms in the active-application model by multiplying each bin by T_{Dest}/T_{Source}, where T_{Dest} represents the number of CPU cycles taken from the appropriate element in the appropriate calibration vector for the destination node and T_{Source} represents the comparable value taken from a calibration vector for the source node. In Figure 4, applying this ratio for the "Write" system call yields a scaling factor of 1.57, while applying this ratio to the ANTS execution environment gives a scaling factor of 1.51. Applying these factors to each element of an active-application model has the effect of dilating or contracting the number of CPU cycles in each bin of each histogram. For example, Figure 4 shows the application of the appropriate scaling factor to the "Write" system call and to the "Write-to-Socket Call" transition.

Unfortunately, to enable our transformation algorithm, a destination node must have access to calibration vectors from the source node. This implies that the calibration vectors must be transmitted along with an already large model for the active-application. Instead, we can agree globally on an artificial node (*Ref*) as a reference, and deploy its calibration vectors at each node in the network. Then, before transmitting an active-application model between two nodes, *Source* and *Dest*, the model is subjected to a "Node-to-Reference transform": the values describing the number of CPU cycles required to execute each element in each histogram are dilated or contracted using the ratio T_{Ref}/T_{Source}, where T_{Ref} is the average number of CPU cycles taken to execute the histogram element on the reference node and T_{Source} is the average number of CPU cycles taken to execute the corresponding element on the source node. Upon arrival at the destination node, the model is subjected to an inverse (the ratio is T_{Dest}/T_{Ref}) "Reference-to-Node transform". The combination of these two transforms scales the CPU cycle values within an active-application model from a form meaningful on a source node to a form understood on a destination node.

5.2. Calibration Techniques

Obtaining the calibration vectors for specific nodes requires the execution of two calibration benchmarks,

one for system calls and one for execution environments. For system-call calibration, we execute a program that repeatedly invokes each system call under a range of parameter settings and then computes the average number of CPU cycles required to execute each system call. We make our calibration measurements using the same techniques we developed for tracing executions. Comparing our measurements against similar measurements taken with strace, using the –c option, we discovered that our measurement technique introduces a constant overhead into the system calls. We factor out this measurement overhead when creating our calibration vector for system calls.

Calibration of execution environments requires running a benchmark workload of active applications on each execution environment. By analogy with the classical process of computer-system benchmarking, we had two possible choices for a benchmark workload for calibrating execution environments. We could use a workload that includes a realistic mix of actual active-applications implemented for each execution environment, or we could define a workload of artificial applications whose behavior mimics the major classes of active-applications. The first option proved infeasible because active-network technology remains experimental, and few real applications exist. For now, we use an artificial mix of active-applications, executed with each node taking on a variety of roles, such as source, router, and sink. For example, for the Magician environment we use three applications (Smart Ping, Smart Route, and Active Audio) and for the ANTS execution environment we use two applications (Ping and Multicast). As the pool of applications grows, the calibration workload must be updated to reflect new functionality or roles. Even so, the ANTS Multicast application, while very basic, exercises all the major functions of active networking: to send and receive packets, and to store and modify information in nodes.

Since calibration is likely to require substantial computation on a node, we must consider appropriate means to perform the calibration. Several approaches should be investigated. In our case, we performed the calibration off-line, and then stored the results as parameters within a node operating system. This approach has the merit of requiring no resources from a node during operational execution. Of course, whenever a system configuration changes, the previously computed off-line calibration may no longer prove accurate.

As a second alternative, we could consider boot-time calibration. Here, the calibration programs would execute automatically as part of the startup process in the operating system. This approach has two advantages. First, since most operating systems must be re-booted after significant configuration changes, calibration at system boot is likely to account for the variability introduced by system alterations. Second, since calibration is completed prior to system execution, the calibration process will require no resources after the node becomes operational. One downside is that boot–time calibration could considerably lengthen system startup time. Additionally, future operating systems seem destined to include dynamic configuration through components downloaded during execution. Boot-time calibration could not account for such dynamic run-time changes in an operating system.

A third alternative is to execute an off-line calibration, and then to perform run-time calibration adjustments. Here boot-time would not be lengthened due to calibration requirements. In addition, configuration changes that affect the calibration can be accounted for during the run-time calibration adjustments. One might even consider altering automatically the frequency of run-time calibration adjustments depending on the variance computed between successive calibrations. As the variance diminishes between successive calibrations, the calibration adjustment interval could be lengthened. Conversely, increasing variance would stimulate more frequent calibration adjustments. The approach has two drawbacks. First, run-time calibration adjustments would subtract resources from operational uses of a node. Second, it might prove difficult to design and implement an effective run-time calibration adjustment mechanism.

5.3 Evaluating Transformed Models

In this section, we evaluate how well our transformed black-box models can predict the CPU demands for an active application measured on one node and then executed on another node. In effect, here we are evaluating how much additional error is introduced into a model by our transformation technique, and the associated calibration processes. We also compare our transformation technique against a more naïve approach that uses the ratio of processor speeds to scale models. To widen our base of platforms, we introduce two new nodes, Yellow and Red, to augment those described in Table 1. Both Yellow and Red use the same versions of Linux and Java as the nodes shown in Table 1; however, the platform hardware differs. Yellow embodies a Pentium 75 running at 100 MHz, and has 80 MB of memory, while Red includes a Pentium II running at 266 MHz with 128 MB of memory. Figure 5 shows all five nodes configured in a small active network in our laboratory.

We ran selected active applications repeatedly on each node, measuring the actual CPU time required for each execution. We then computed the mean CPU time and the high percentiles (80^{th}, 85^{th}, 90^{th}, 95^{th}, and 99^{th}) of CPU time used by each application on each node. These

served as baseline measurements against which we compared estimates obtained using our black-box model and transformation techniques. We also generated estimates using a more naïve approach that multiplies the observed execution times on a source node by the ratio of the processor speed of the source node to the processor speed of a destination node. Using this ratio, we scale the CPU time requirements to match the relative speed of the processors on each node. Table 7 provides a subset of the percentage (absolute) error we achieved when using each method to predict the mean and high percentiles of CPU usage when moving active application models between nodes. In this table, we average the error across the five high percentiles.

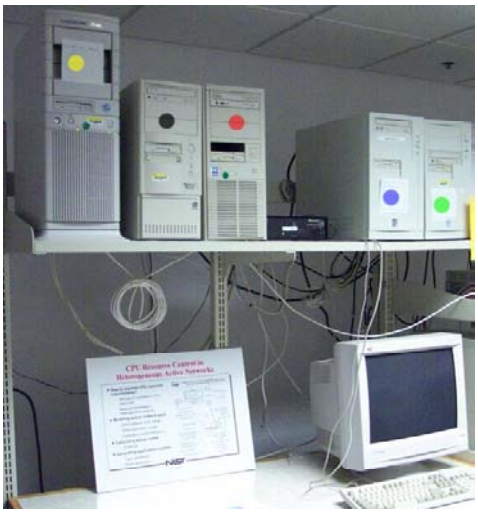

Fig. 5. The five-node active-network test bed we set up in our laboratory at NIST in order to conduct our experiments and to make measurements. The nodes from left to right: Yellow, Black, Red, Blue, and Green.

Table 8 shows comparative results for the percentage error in each statistic (mean and each of the high percentiles) when averaged over all runs. The table compares prediction error in three situations: (1) predicting performance on one node with our black-box models, (2) predicting performance on other nodes by scaling our black-box models, and (3) predicting performance on other nodes by scaling with processor speed ratios. Note that scaling the black-box models yielded a fourfold improvement in accuracy over scaling based solely on processor speed ratios. In addition, as Table 8 shows, scaling our black-box models did not introduce additional error beyond the error already present in the models.

EE is Execution Environment AA is Active Application				Scaled with Processor Speed Ratio		Scaled with Black-Box Model	
EE	AA	Source Node	Dest. Node	% Error Mean	% Error Avg. High Percentile	% Error Mean	% Error Avg. High Percentile
ANTS	Ping	Yellow	Green	2	6	0	8
		Red	Black	5	5	2	8
		Blue	Black	65	66	7	7
	Multicast	Green	Blue	46	47	2	12
		Black	Red	11	25	2	10
		Yellow	Black	2	8	5	10
Magician	Smart Ping	Blue	Black	81	74	5	9
		Green	Black	32	21	6	14
		Yellow	Red	45	43	9	10
	Smart Route	Blue	Red	92	86	2	24
		Red	Black	8	4	5	9
		Black	Yellow	94	94	3	10

Table 7. Reporting the percentage absolute error in estimating the mean and the average percentage absolute error in estimating the high percentiles (80^{th}, 85^{th}, 90^{th}, 95^{th}, and 99^{th}) using naïve scaling based on processor speed ratios and using scaling based on transformation of black-box models. The table presents a representative subset of the results we obtained.

		Average Percent Error Across All Runs for Selected Statistics					
		Mean	Percentile				
Prediction Method	Prediction Target		80th	85th	90th	95th	99th
Black-box Model	Same Node	3	17	22	14	10	17
Scaling Black-box Model	Different Node	4	14	12	8	10	10
Scaling with Speed Ratio	Different Node	40	43	39	39	40	38

Table 8. Comparing the absolute percent error for selected statistics, averaged across all runs, when predicting CPU demands in three situations: (1) using black-box models to predict CPU demands for the same node on which the model was generated, (2) scaling black-box models to predict CPU demands on different nodes from that on which the model was generated, and (3) scaling CPU demands based upon the ratio of processor speeds between pairs of nodes.

5.4 Anatomy of an Active-Packet Hop

Here, in way of summary, we describe an approach for using our black-box model when processing an active packet as it transits between two nodes in an active network. We assume that the code exists for an active application, *App*, and that a corresponding black-box model has been generated off-line on a node, *Source*. Further, we assume that the code and the black-box model for *App* have been loaded onto a code server.

Before loading the *App* black-box model onto the code sever, *Source* subjects the model to a "*Source-to-Reference* transform", so that the model is available throughout the network in its reference form.

When the first active packet related to *App* arrives at a node, *Next*, the execution environment extracts references (typically in the form of Uniform Resource Locators, or URLs) to the *App* code and its related black-box model. Using the URLs, *Next* retrieves the *App* code and black-box model from the code server, and subjects the model to a "*Reference-to-Next*" transform. Then, *Next* executes a Monte-Carlo simulation of the transformed *App* model to estimate relevant statistics (such as mean and 99th percentile) for the CPU time required on *Next* to process active packets associated with *App*. Using the estimated statistics, *Next* can decide whether or not to admit active packets associated with *App.* Each admitted *App* packet is executed using the retrieved *App* code. During execution, *Next* can monitor the CPU time used by *App* packets, and can terminate those that exceed their estimated demand by some threshold.

6. Sample Applications

In this section, we illustrate how our CPU demand models can be used in two sample applications. In one application, we decide when to terminate an active packet based on its consumption of CPU time. In a second application, we predict the CPU demand for nodes in an active network. In both applications, we compare results obtained using our black-box models against results obtained using CPU control and estimation techniques typically available in execution environments.

6.1. CPU Usage Control

As active packets traverse a series of nodes along a path from source to destination, each active node will wish to enforce CPU usage limits on each packet. This permits a node to protect itself from malicious or erroneously programmed active packets. Some execution environments provide a fixed maximum limit for any active packet, while some also permit each active packet to specify its own limit. In this way, should the active node choose to allow the packet to execute, the node will also have an idea when the packet should be terminated. In a small sample application, we show how the use of a fixed time-to-live (TTL) in each packet can lead to stolen and wasted CPU time in active nodes. We also show how our black-box models can be used to adjust the TTL on each active node; thus, saving CPU time and improving the quality of service in applications. Our sample active-audio application runs in the Magician [11] execution environment over the topology shown in Figure 6.

In this topology the source node (Green – 199 MHz) sends a stream of 2278 40-byte audio packets to the destination node (Black – 450 MHz) across two routers. The first router (Blue – 333 MHz) is faster than the source, and the second router (Yellow – 100 MHz) is slower than the source. Measurements of the application running on the source reveal that 8.29 ms is the 99th percentile of CPU time used to process active-audio packets. In our sample application, the source selects this value as the TTL for each active packet. Unfortunately, in our case study, an intruder on the source node manages to inject 455 malicious packets into the stream of valid active-audio packets. Each malicious packet is programmed to consume as much CPU time as possible on each node.

During the experiment, each malicious packet is allowed to use 8.29 ms on the first router before the packet is killed. However, all malicious packets are terminated on the first router. The CPU time allocated to the malicious packets is stolen from other users. Worse, as valid packets arrive at the second, slower, router, they are each given up to 8.29 ms of CPU time. Unfortunately, as Table 9 reveals, 23.99 ms is the actual 99th percentile required by active-audio packets executing on the second router. As a result of the poor TTL value, the second router kills 96% of the valid packets. The time spent processing the killed packets amounts to CPU time wasted on the part of the second router, and the end user receives an unacceptable quality of service.

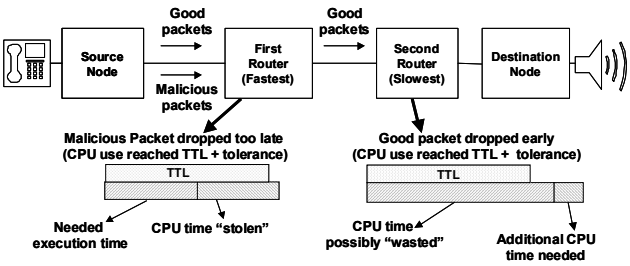

Fig. 6. A four-node active-network topology used to run an active-audio application that relays audio packets between a source and destination node over two intervening routers.

Table 10 provides a summary of results from running the active-audio application with two different approaches to TTL assignment. The first approach assigned a fixed TTL of 8.29 seconds, based on the observed 99th percentile on the source node. The second approach assigned a variable TTL on each node. In this case, the TTL on each node was determined by scaling a black-box model of the application that was generated on the source node (using the techniques discussed in Sections 4 and 5). During each run, the application

injected 2278 valid active-audio packets into the path, and also injected 455 malicious packets, about one for every five valid packets.

All values given in: milliseconds (CPU cycles)	99th Percentile CPU Usage		
	Source Node	First Router	Second Router
Measured Value	8.29 (1,650,084)	4.76 (1,589,382)	23.99 (2,398,702)
Fixed TTL	8.29 (1,650,084)	8.29 (2,769,487)	8.29 (829,187)
TTL Derived from Scaled Black-box Model	8.29 (1,650,084)	4.76 (1,589,382)	23.99 (2,398,702)

Table 9. Comparing the measured 99th percentile for the CPU time used by an active-audio application running on nodes in an active-network topology. The table shows the Time-To-Live (TTL) and the equivalent number of CPU cycles on each node for three cases: (1) measurements taken on each node, (2) a fixed TTL assigned based on measurements taken on the source node, and (3) predictions generated by scaling a black-box model generated from measurements taken on the source node.

2278 Valid Packets 455 Malicious Packets	CPU Time Stolen on First Router	CPU Time Wasted on Second Router
Fixed TTL	3,772 ms [455 * 8.29ms]	18,122 ms [2,186 * 8.29 ms]
TTL Derived from Scaled Black-box Model	2,166 ms [455 * 4.76 ms]	456 ms [19 * 23.99 ms]

Table 10. Comparing CPU time stolen or wasted on routers in an active-network topology when running an active-audio application. The table shows two situations: (1) fixed TTL and (2) variable TTL.

Using a fixed TTL, the malicious packets stole 3,772 ms (455 malicious packets X 8.29 ms TTL) from the first router. Using a variable TTL, the malicious packets stole only 2,166 ms (455 malicious packets X 4.76 ms) from the first router. This amounts to saving 3.53 ms per malicious packet, which could provide breathing room needed to activate defensive mechanisms in the router.

Our improved CPU-time estimation cannot combat malicious packets without also taking into account the topology of the deployed active application. For example, had the topology been inverted so that active packets traveled first to the slowest router and then to the fastest router, each malicious packet could consume 23.99 ms before being killed, resulting in 10,916 ms of stolen CPU time. This outcome indicates the need to deploy an active application in an appropriate topology to effectively combat injection of malicious packets. Specifically, ensuring that the active-packet stream transits a fast, first-hop router can lead to an outcome where malicious packets are filtered with a minimum of stolen CPU time. Then, only valid packets will be forwarded to subsequent hops along the route.

On the second router in our experiment, where only valid packets arrive, the use of a fixed TTL leads to an unfortunate outcome, where 96% (2,186 / 2,278) of the packets are terminated, each after consuming 8.29 ms of CPU time. This amounts to wastage of 18,122 CPU milliseconds, and presents an untenable audio channel to the end user. When a variable TTL is used, the situation improves greatly. First, the second router terminates less than 1% (19 / 2,278) of the valid packets. This improves the quality of service to an acceptable level, and limits the wasted CPU time to only 456 ms. These results confirm that improved models for CPU demand enable better control as mobile code traverses heterogeneous nodes in a network.

6.2. CPU Demand Prediction

In a second sample application, we demonstrate how improved models for CPU demand can lead to better predictions about the capacity available among nodes in an active-network topology. In this case, we concern ourselves with predictions for average CPU demand, rather than predictions for the 99th percentile. To conduct our case study, we use the Active Virtual Network Management Prediction (AVNMP) system [27] developed by researchers at the General Electric Corporate Research and Development Center. AVNMP applies active-network technology to inject simulation models into network nodes, and to run those models concurrently with corresponding applications. AVNMP then compares estimated performance against measured performance, and maintains predictions from the simulation within specified error bounds, when compared against measurements from the application.

To predict traffic load in a network, AVNMP constructs a shadow topology that overlays the operational network and then runs a simulation in the shadow topology. Figure 7 illustrates the relationship between the operational network and the shadow, prediction-overlay network. Using Magician as an execution environment, AVNMP deploys driving processes (DP) at each source node and logical processes (LP) at each intermediate and destination node in the topology of the operational network. DPs and LPs are deployed as active applications within an active virtual-overlay network (space dimension in Figure 7). Each DP

contains a model that simulates message sources, generating virtual messages that flow along links in the virtual-overlay network, which share physical links between nodes but remain logically isolated from operational traffic. As virtual messages arrive, the LP updates variables in the node's management information base (MIB) [28]. Each LP updates the future state of relevant MIB variables, providing the MIB with predicted state to complement the current and past state maintained for the operational network. After updating predicted MIB variables, the LP consults the node's routing table and forwards incoming virtual messages on to other LPs, if required.

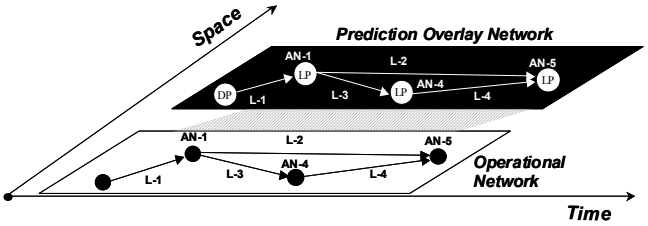

Fig. 7. An illustration of the time-space relationship between an operational network and a prediction-overlay network constructed by the Active Virtual Network Management Prediction (AVNMP) system.

The prediction-overlay network then generates and routes simulated network traffic that attempts to run ahead in virtual time of operational network traffic (time dimension in Figure 7). While the operational network advances in real time, the LP in the prediction-overlay network advances in virtual time, receiving virtual messages and estimating future load. Periodically, the LP compares the actual and predicted MIB values for corresponding intervals in real and virtual time. If the values agree within an error tolerance, then the simulation remains ahead of real time and continues to advance. If not, then the LP rolls virtual time back to the current real time, discarding predictions for future MIB state, and then simulation resumes. AVNMP contains some special processing to cancel virtual messages that might be in transit across the prediction-overlay network during a rollback, but we omit these details.

As shown in Figure 8, we constructed a four-node, heterogeneous active network, consisting of the same topology and nodes used for the active-audio case study (see Section 6.1). The operational active network comprised these nodes connected to a switched 10-Mbps Ethernet, which included a few other nodes that were not part of the experiment. We configured the experiment nodes to run the active-audio application discussed earlier; however, in this case we omitted the malicious packets. The prediction overlay network included AVNMP deployed as an active application on each node, with a DP injected into the source node and an LP injected into the destination and each intermediate node. The DP included a message model to generate virtual message traffic and a CPU model to estimate the processor demand associated with each virtual message. Each LP also included a copy of the CPU model to estimate processor demand for each arriving virtual message.

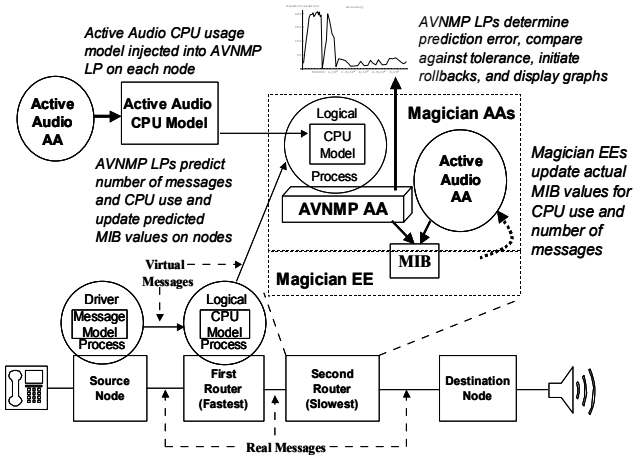

Fig. 8. A four-node topology supporting both an operational active network and a prediction-overlay network. The figure indicates how the various components of the AVNMP prediction-overlay network were deployed to estimate resource demand for an active-audio application.

We conducted two experiment runs. In the first run the DP and LPs predict a fixed average CPU time for each virtual message on every node. In the second run, the average CPU time predicted for each virtual message differs on each node, based on predictions made by scaling our black-box model of the active-audio application. Table 11 shows the relevant experiment parameters at each router node.

We assigned 7 ms per packet as the average CPU demand in the fixed prediction models. This figure was obtained by measuring the active-audio application executing on the source node. Note that 7 ms equates to a different number of CPU cycles on each node, depending on processor speed. By scaling our black-box model, we estimated 3 ms per packet as the average CPU demand on the first router and 16.5 ms on the second router. Our hypothesis: because our scaled black-box model more accurately represents CPU demand in the active-audio application, as compared against the fixed-time estimate, AVNMP should require fewer tolerance rollbacks; thus, the prediction-overlay network should provide better

look-ahead into virtual time. We ran two experiments to evaluate this hypothesis.

	Fixed Average CPU Time Scaled with Speed Ratio		Average CPU Time Scaled with Black-box Model	
Experiment Parameter	First Router	Second Router	First Router	Second Router
Avg. CPU Time ms (and CPU cycles)	7 (2,340,750)	7 (693,000)	3 (900,000)	16.5 (1,633,478)
Error Tolerance (+-10%) (CPU cycles)	234,075	69,300	90,000	163,347
Avg. Measurement Interval (s)	8.8	12.1	10.1	7

Table 11. The average CPU estimates used by AVNMP for each router in the prediction-overlay network, reported as milliseconds and as the equivalent CPU cycles. The table also indicates the number of CPU cycles that define the 10% error tolerance on each node, and the average interval at which measurements were sampled.

For each experiment run we fixed the relative error tolerance at 10 %, which means that AVNMP initiates tolerance rollbacks whenever the measured CPU use (averaged over 20 messages) differs from the predicted CPU use by more than 10 %. This tolerance, computed relative to predicted CPU use, equates to a different number of CPU cycles for each node and run. Using a wider error tolerance would likely mask any improvements from improved CPU demand predictions.

In conducting each run, the active-audio application emitted a stream of 91,105 bytes (2,277 40-byte packets followed by one 25-byte packet), and the intermediate nodes periodically measured the cumulative tolerance rollbacks and the virtual time. As shown in Table 11, the average measurement interval varied on each node due to the stochastic nature of thread scheduling in Java. Table 12 compares the results we obtained from our experiment runs.

Source Node is Green.
First Router is Black.
Second Router is Yellow.
Destination Node is Blue.

	Fixed Avg. CPU Time			Avg. CPU Time from Scaled Black-box Model		
	First Router	Second Router	Dest. Node	First Router	Second Router	Dest. Node
Maximum Look Ahead (s)	-101	-20	54	432	102	313
Cumulative Rollbacks	92	42	12	28	0	0

Table 12. Reports the results measured at two routers during two different experiment runs. The results include the cumulative number of rollbacks and the maximum look-ahead observed over all measurement intervals.

Over the audio streaming period, we can compare AVNMP performance for the same nodes when using the fixed CPU-demand model vs. the adapted CPU-demand model. For both the fastest and slowest intermediate node, the adapted CPU-demand model induces fewer tolerance rollbacks. This permits AVNMP to reach a greater maximum look ahead into virtual time on each intermediate node. Figures 9 and 10 provide a view of cumulative rollbacks and virtual time, respectively, on an interval-by-interval basis for the first router. The graphs compare progress with the fixed CPU-demand model against progress with the adapted CPU-demand model. These results support our hypothesis, suggesting that use of an adaptive CPU-demand model can improve the ability of AVNMP to predict resource usage in heterogeneous active networks.

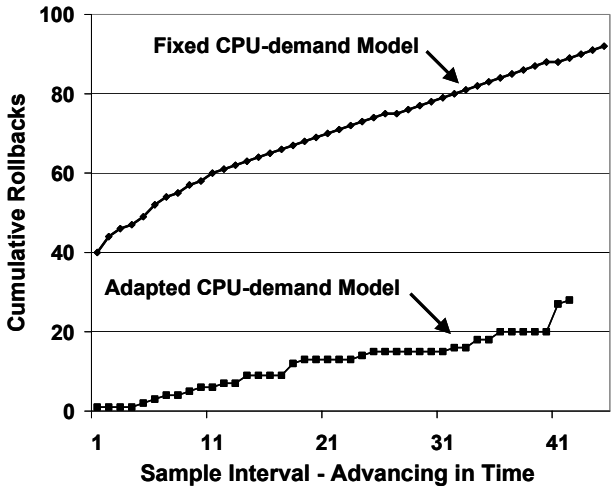

Fig. 9. An interval-by-interval comparison of the change in cumulative rollbacks for the fixed CPU-demand model versus the adapted CPU-demand model.

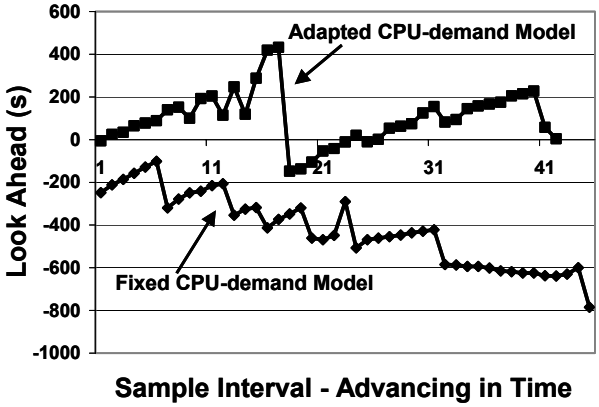

Fig. 10. An interval-by-interval comparison of the change in look-ahead in virtual time for the fixed CPU-demand model versus the adapted CPU-demand model.

7. On-Going and Future Work

While our black-box models of CPU demand, and the associated scaling techniques, appear promising, more research remains before the models can be practically applied. In this section, we outline some of the open issues in three main categories: (1) improving our existing black-box models, (2) investigating white-box models as an alternative to black-box models, and (3) exploring continuous improvement strategies that would enable models and node calibrations to monitor their own performance and to adapt to new conditions. We begin by considering the state of our black-box models.

7.1. Improving the Black-Box Model

The performance of our black-box models can be considered along three dimensions. Along the dimension of accuracy, our existing models assume that all application behavior can be measured prior to injecting a model into network nodes. Unfortunately, application behaviors often reflect conditions that cannot be known before a program reaches a node. For this reason, our application model must be enhanced to account for such node-dependent conditions. Two particular issues occur in this regard. First, some behaviors may appear more or less often on a particular node than the model would predict, based on the scenarios observed in the laboratory where a model is created. Given the restriction of black-box models, this becomes a statistical question surrounding whether or not the behavior measured in generating the model represents the behavior in actual use. Attacking this problem on a black-box basis requires some ability for continuous improvement (see Section 7.3 below). Removing the black-box restriction opens up the model and permits strategies more suited for white-box analysis (see Section 7.2 below). Second, selected scenarios in our black-box models might be repeated at a node, based on conditions at the time of execution. For example, in a multicast application a packet might be forwarded a number of times that depends on the current number of subscribers to a multicast group. We might be able to parameterize looping behavior in our black-box models (making them grey-box models, perhaps). If we can do this, then an arriving model might query the execution environment on a node for the current values of key behavioral parameters, and then could modify its CPU demand estimates accordingly.

Along the dimension of cost, our models consist of histograms, which must be exercised with Monte-Carlo simulations in order to predict CPU demand. As a result, specific application models can be large and could require substantial computation to produce predictions. To some degree the space-time properties of our model can be modulated; however, the prediction error also varies accordingly. We discuss these points further.

In our research, we found that the size of a model can vary depending in the first order on three parameters: the execution environment, the active application, and the granularity of the histograms. The execution environment, and its mapping to a node operating system, appears to affect the number of system calls made by an active application. Further, an active application may consist of a number of different roles (such as source, router, and sink), where a node may take on one or more of the available roles. The number and nature of roles in an active application affect (in the second order) the number of scenarios, and the number of scenarios can affect (in the third order) the number of transitions and the number of distinct system calls taken. To determine a model size, the number of bins in each histogram multiplies the number of transitions and system calls. Tables 13 and 14 provide, for ANTS and Magician, respectively, some statistics regarding the size of the models generated during our research.

Units are Bytes	Role	Average	Standard Deviation	Minimum	Maximum
Ping	Source	67,103	59,651	7,443	153,622
	Router	10,082	3,120	5,706	17,340
	Sink	18,277	40,843	4.,300	165,656
	All	80,162	59,266	16,913	165,656
Multicast	Source	56,433	14,452	38,744	80,163
	Router	9,466	3,558	5,090	17,190
	Router-Sink	63,487	25,128	42,414	112,528
	Sink	17,966	3,109	12,020	22,722
	All	120,325	42,583	65,287	213,679

Table 13. Some statistics about the size of black-box models generated for various possible roles that can be taken by two different active applications running in the ANTS execution environment. Note that both applications can adopt one or more of three roles. The role "All" denotes the application executing in all available roles. For the Multicast application, the table includes a row showing a combined role, "Router-Sink".

Tables 13 and 14 support the observation that the execution environment and the active application affect the size of the model. For example, notice that the various applications in each table require different numbers of bytes to describe a model. Further, note that two similar applications, "Ping" (Table 13) and "Smart Ping" (Table 14), required different sizes based on being written for different application environments. Tables 13 and 14 also provide some indication of the size of the

models that would be shipped among nodes. But model size is only part of the story.

Units are Bytes	Role	Average	Standard Deviation	Minimum	Maximum
Smart Ping	Source	12,557	6,187	5,182	25,518
	Router	13,001	7,257	3,084	25,180
	Sink	8,283	4,043	3,594	16,658
	All	27,641	13,765	12,307	53,651
Smart Route	Source	8,201	3,439	2,297	14,051
	Router	6,861	3,644	1,029	11,700
	Sink	2,610	1,031	824	4,683
	All	17,465	7,269	5,339	28,143

Table 14. Some statistics about the size of black-box models generated for various possible roles that can be taken by two different active applications running in the Magician execution environment. Note that both applications can adopt one or more of three roles. The role "All" denotes the application executing in all available roles.

Once a model arrives on a node, a Monte-Carlo simulation must execute to generate a sample population of CPU demands from which prediction statistics can be determined. In our experiments, we implemented the models in Java, which is not the most efficient choice. Table 15 shows the CPU seconds required to execute a number of our five-bin histogram models for varying repetition counts. Here, the models were executed on a Pentium Pro operating at 547 MHz. As shown in Table 16, the larger the number of repetitions, the better the accuracy of the predictions. Of course, the larger the number of repetitions, the more CPU time is needed to generate the predictions. Table 16 shows the increasing accuracy of the predictions as the number of repetitions increases from 500 to 20,000. On the other hand, Table 16 also shows that at 20,000 repetitions, increasing the number of bins from 50 to 100 does not appreciably improve the accuracy of the predictions.

Along the dimension of operational effectiveness, our models would benefit from inclusion of an associated error bounds. Before taking decisions based on predictions from CPU-time models, an operating system must consider the possible range of prediction error. While we have yet to characterize the error properties of our models, Table 17 provides another look at how scaling our black-box models compares against scaling predictions based on the ratio of processor speeds. For sake of discussion, assume that these results hold across all models of each type. Then, upon receiving predictions from a scaled black-box model, an operating system could realize that the predictions for the mean might be incorrect by up to 5% and that predictions for the higher percentiles might prove inaccurate by as much as 15%. On the other hand, when working with a model scaled based on the ratio of processor speeds, the operating system would realize that all predictions could be around 35% in error.

Models implemented in Java		CPU Seconds Required to Execute Model					
		100 Repetitions		1,000 Repetitions		10,000 Repetitions	
Execution Environment	Active Application	Mean	Standard Deviation	Mean	Standard Deviation	Mean	Standard Deviation
ANTS	Ping	0.69	0.12	0.86	0.14	4.51	1.55
	Multicast	0.75	0.11	0.91	0.14	5.48	1.17
Magician	Smart Ping	0.62	0.07	0.71	0.09	4.39	0.48
	Smart Route	0.64	0.10	0.74	0.08	3.82	0.82

Table 15. Shows the average number of CPU seconds (and the standard deviation) required to execute four different models through various repetitions. All models, implemented in Java and executed on a 547 MHz Pentium Pro, were composed of five-bin histograms.

Models of Active Applications Running in ANTS	50 Bins and 500 Repetitions		50 Bins and 20,000 Repetitions		100 Bins and 20,000 Repetitions	
	% Error Mean	% Error Avg. High Percentile	% Error Mean	% Error Avg. High Percentile	% Error Mean	% Error Avg. High Percentile
Ping	3	10	1	2	1	1
Multicast	5	16	0	3	0	2

Table 16. This table reports the percent absolute error predicting the mean and high percentiles for two active-applications running in the ANTS execution environment. The results give the error measured for three combinations of histogram granularity and simulation repetition count.

Standard Deviation in % Absolute Error for Selected Statistics	Scaling with Processor Speed Ratio	Scaling Black-box Model
Mean	36	3
80th Percentile	36	11
85th Percentile	34	12
90th Percentile	36	7
95th Percentile	37	5
99th Percentile	34	5

Table 17. Comparison of error bounds associated with predictions made from scaling models based on the ratio of processor speeds against predictions made from scaling black-box models.

7.2. Investigating White-Box Models

In addition to seeking techniques to improve black-box models, we have begun to investigate white-box models as an alternative approach. In our conception, white-box models represent the processing logic within an active application as it invokes services offered by an execution environment. Figure 11 (a), for example, shows mobile, an active application written for the ANTS execution environment, while Figure 11 (b) shows a corresponding white-box model for mobile.

```
Integer f = (Integer)n.getCache().get(getDst());
if (f  != null) { next = f.intValue(); }
if (n.getAddress() != getDst())
   { return n.routeForNode(this, next); }
else { return n.deliverToApp(this, dpt); }
```

(a) ANTS active application mobile

```
delay (t1);
if (c1) delay (t2);
if (c2) { delay (t3); }
else { delay (t4); }
```

(b) White-box model for mobile

Fig. 11. (a) Example code for an active application, ANTS mobile, and (b) a corresponding white-box model, where *c1* and *c2* represent distinct Boolean conditions and *t1* through *t4* represent distinct time delays.

As shown in Figure 11 (a), an active application consists of a combination of sequences, selections, and iterations that invoke specific primitives provided by an execution environment. In this example, such primitives include: *getCache*, *getDst*, *intValue*, *getAddress*, *routeForNode*, and *deliverToApp*. Given a specific active packet and a determinable state for relevant node-dependent conditions, prior to executing the packet, an active-network node can evaluate the state of relevant Boolean conditions (see *c1* and *c2* in Figure 11 (b)) to determine, the precise sequence of primitives that an active application will call to process the packet. Further, if the node can determine the time taken by the execution environment to execute each primitive, then the node can compute an estimate for the CPU time required to process the packet. To determine the amount of time taken to execute each primitive, an execution environment must be calibrated on the node. Calibration involves the execution of a synthetic workload that will repeatedly call the various primitives implemented by the execution environment. The calibration process yields estimates for various statistics (e.g., mean and variance) associated with CPU use by each primitive.

We imagine that an execution environment can generate a white-box model for an active application, once the source code arrives at a node. Figure 11 (b), for example, provides a possible white-box model derived from the source code for mobile. Then, assuming that each delay in the model (*t1* through *t4* in Figure 11 (b)) represents the CPU time required for an associated primitive, the model can be evaluated for each arriving active packet to estimate the CPU demand for that packet. In our preliminary work, the calibration process yields estimates for the first two moments (mean and variance) of CPU time used for each primitive in the execution environment. We estimate the mean execution time for a packet as the sum of the mean primitive times in the processing path for the packet. Similarly, we use an appropriate formula for summing the variance of random variables to derive an estimate for the variance in CPU demand by the active packet. Finally, assuming a normally distributed random variable, we use the mean plus an appropriate multiple of the standard deviation to estimate specific percentiles. While we already know through our experiments that CPU usage is not a normally distributed random variable, we used such an assumption in order to explore the effectiveness of a simple analytical approach to computing estimates for CPU demands.

Table 18 illustrates some results from applying this technique to predict CPU demand for five active applications running under the ANTS execution environment. The table compares predictions against measurements for three statistics: mean, standard deviation, and 99th percentile. The prediction errors are neither as accurate nor as well bounded as those obtained with our black-box models. We believe that this poor performance results from our assumption that CPU demand is normally distributed (which our measures demonstrate is clearly not the case). Regardless of these preliminary results, our work with black-box models leads us to believe that white-box models could be combined with histograms and Monte-Carlo simulations to yield reasonably accurate estimates. In the case of white-box models, the histograms would represent the CPU usage observed during calibration for each primitive provided by the execution environment. We have plans to investigate these ideas in the context of resource-management for mobile code loaded into call-processing servers.

7.3. Continuous Improvement Strategies

Regardless of the type of model chosen to provide estimates for CPU demand, strategies for continuous improvement will be required. We envision additional

work on techniques for continuous calibration of system calls and execution environments, for experiential improvements in active-application models, and, possibly, for real-time competition among various models. We discuss each of these topics below (*a-c*).

Predictions from white-box models	% Absolute Error in Prediction for Selected Statistics		
ANTS Active Application	Mean	Standard Deviation	99th Percentile
Mobile Update	37	5	33
Ping	5	60	0
Mobile	11	26	6
Multicast Subscribe	27	32	67
Multicast	0	52	67

Table 18. Prediction error for three different statistics (mean, standard deviation, and 99th percentile) estimated for five active applications. These predictions relied on white-box models, combined with analytical approximations appropriate for normally distributed random variables.

(a) Continuous Calibration. Calibration of a node and execution environment, even when carefully conducted, yields accurate information only so long as no change occurs in relevant elements of the calibrated system. Once a configuration changes, e.g., through introduction of new hardware or an updated version of some software component, a previous calibration might no longer prove accurate. In addition, to the extent that a calibration depends upon usage patterns associated with the calibrated components, the accuracy of a calibration might drift. For these reasons, research is needed to develop and validate techniques to recalibrate a system over time. In particular, techniques might be needed to track changes in calibration values, and then to vary the rate of calibration adjustment based on the rate of change in calibration accuracy.

(b) Learning Models. The accuracy of statistical models of program behavior depends upon successfully obtaining samples of representative behavior. Our black-box modeling approach assumes that representative application behavior can be measured sufficiently, during a tracing phase, prior to injecting a model into network nodes. Unfortunately, application behaviors often reflect conditions that cannot be known before a program reaches a node. Such conditions can alter the probability of executing various paths in a program, and can change the number of times particular paths are executed. For this reason, additional research is needed to investigate techniques to continuously improve the representation of statistical behaviors in black-box models. Can methods be found to enable a model to evolve as it gains experience while traveling through the network? Can new scenarios be identified and added to a model? Can the probability of execution and the distribution of the CPU times be adjusted as the application experiences more executions? Can models be parameterized based on conditions at a node? For example, to solve the problem of a loop executed an unpredictable number of times, can we design a holes-model, complete except for some parameters that would be included on arrival at the node where local conditions are known?

(c) Competitive Models. Our existing research assumes that we can develop one class of model that best predicts CPU demands for a mobile program. This assumption might prove wrong. We might be unable to find a single class of predictor that will yield the best estimates for all active applications. For example, one model might produce estimates through analytical computation, while another provides predictions using simulation. Perhaps one estimation technique gives better results than another under certain conditions. If so, then it could prove useful to continuously evaluate which of the available co-existing models or prediction systems is the most accurate. In this way, good predictors can be reinforced, and bad predictors can be de-emphasized, and the value of predictors can be assessed independently in time and space. Active-network technology provides a suitable basis to experiment with such competitive modeling techniques.

8. Conclusions

In this paper, we argued that some means is needed to accurately specify CPU demand in order to safely and efficiently deploy mobile code among heterogeneous platforms in a network. We showed that commonly used approaches, which are based on a fixed time-to-live, do not work effectively. We argued that CPU demand in a mobile program is a function of the speed of the processor on which the program runs and of the number of CPU cycles that must be executed. Further, we showed that it is quite difficult to estimate the number of CPU cycles demanded by a mobile program.

We proposed a class of statistical black-box models to estimate the number of CPU cycles required by a mobile program, and we evaluated how well the predictions from some instances of these models matched measured values. Further, we proposed mechanisms to transform instances of black-box models to provide estimates for CPU demand on a range of nodes. We

evaluated how well predictions made by transformed models matched measured values. We also compared the accuracy of our transformed black-box models against transformation techniques that take into account only the differences in processor speed among nodes. In most cases, the black-box models proved accurate within 15%, while the more naïve models proved accurate within 40%.

In addition to evaluating our black-box models, we applied one of them in two sample applications: CPU control and CPU prediction. In the control application, we demonstrated that better models of CPU demand could reduce the amount of CPU time stolen or wasted when malicious or erroneous code is injected into a node. We also showed that more accurate models of CPU demand can lead to better quality of service provided to end users. In the prediction application, we demonstrated that better models of CPU demand allowed AVNMP, a resource-usage prediction system, to estimate resource demand farther into the future with lower overhead.

Despite the successes reported in this paper, the problem of accurate prediction of CPU demand among heterogeneous nodes remains largely unsolved for mobile programs. We identified several open issues that require additional research. We hope that our analysis of the problem, our evaluation of results, and our demonstration of the benefits of an effective solution, will all encourage other researchers to tackle this important and difficult problem.

Acknowledgments

We thank Hilarie Orman for recognizing the potential impact of our wild ideas. We particularly appreciate the support and encouragement of Doug Maughan, DARPA's program manager for Active Networks, and Scott Shyne, from the Air Force Research Laboratory (AFRL). Our work benefited greatly from collaboration with colleagues, Stephen Bush and Amit Kulkarni, from the General Electric Corporate R&D Center. Working with Steve and Amit enabled us to demonstrate that better estimates for CPU demand can yield practical benefits. We also value the contributions of Stefan Leigh and Andrew Rukhin, colleagues who helped us early in the project to explore the potential accuracy of a wide range of statistical models. The work reported in this paper would not have been possible without funding from the National Institute of Standards and Technology (NIST) and from the Defense Advanced Research Projects Agency (DARPA).

9. References

[1] K. L. Calvert (ed), Architectural Framework for Active Networks, Version 1.0, Draft, July 27, 1999.

[2] L. Peterson (ed.), NodeOS Interface Specification, January 10, 2001.

[3] S. Dawson, M. Molteni, L. Ricculli, and S. Tsui, User Guide to ANETD 1.6.3, Sept. 28, 2000.

[4] S. Bhattacharjee, K. L. Calvert and E. W. Zegura. "An Architecture for Active Networking", *Proceedings High Performance Networking (HPN'97),* White Plains, NY, April 1997.

[5] D. Mosberger and L. L. Perterson, "Making Paths Explicit in the Scout OS", *Proceedings of the Second Symposium on Operating System Design and Implementation*, ACM Press, New York, 1997, pp. 153-168.

[6] F. Kaashoek at al., "Application Performance and Flexibility on Exokernel Systems", *16th Symposium on Operating System Principles*, ACM Press, New York, 1997, pp. 52-65.

[7] B. Ford, G. Back, G. Benson, J. Lepreau, A. Lin and O. Shivers, "The Flux OSKit: A Substrate for OS and Language Research", *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, ACM Press, October 1997.

[8] D. Wetherall, J. Guttag and D. Tennenhouse, "ANTS: Network Services Without the Red Tape", *IEEE Computer*, April 1999, pp. 42-48.

[9] Y. Yemini and S. da Silva, "Towards Programmable Networks", *IFIP/IEEE International Workshop on Distributed Systems: Operations and Management*, October 1996.

[10] M. Hicks, P. Kakkar, J. T. Moore, C. A. Gunter and S. Nettles, "PLAN: A Packet Language for Active Networks", *International Conference on Functional Programming (ICFP),* 1998.

[11] A. B. Kulkarni, G. J. Minden, R. Hill, Y. Wijata, A. Gopinath, S. Sheth, F. Wahhab, H. Pindi and A. Nagarajan, "Implementation of a Prototype Active Network", *Proceedings of OpenArch 98*, 1998.

[12] B. Schwartz, A. W. Jackson, W. T. Strayer, W. Zhou, D. Rockwell and C. Partridge, "Smart Packets for Active Networks", *Proceedings of OpenArch 99*, March 1999.

[13] D.C.Feldmeier, A.J. McAuley, J.M. Smith, D. Bakin, W.S. Marcus, T. Raleigh, "Protocol Boosters", *IEEE JSAC*, Special Issue on Protocol Architectures for 21st Century, vol. 16, no. 3, pp. 437-444, April 1998.

[14] S. Kasera, S. Bhattacharyya, M. Keaton, D. Kiwior, J. Kurose, D. Towsley, and S. Zabele, "Scalable Fair Reliable Multicast Using Active Services". *IEEE Network Magazine* (Special Issue on Multicast), January/February 2000.

[15] S. Zabele, T. Stanzione, J. Kurose, and D. Towsley, "Improving Distributed Simulation Performance Using Active Networks", Invited Paper, *Proceedings of World Multi Conference 2000*, January 23-27, 2000, San Diego, CA.

[16] S. Gribble, E. Brewer, J. Hellterstein, and D. Culler, "Scalable, Distributed Data Structures for Internet Service Construction", *Proceedings Fourth Symposium on Operating Systems Design and Implementation (OSDI 2000)*, 2000.

[17] S. Bhattacharjee, K. Calvert, and E. Zegura, "Self-Organizing Wide-Area Network Caches", *Infocom 98.*

[18] K. L. Calvert, J. Griffioen, B. Mullins, A. Sehgal and S. Wen. "Concast: Design and Implementation of an Active Network Service". *IEEE Journal on Selected Area in Communications* (JSAC). Volume 19, No. 3. March, 2001.

[19] T. Faber, "ACC: Active Congestion Control," *IEEE Network*, IEEE, May/June 1998, pp. 61-65.

[20] J. Reynolds and J. Postel. RFC 1700 Assigned Numbers, October 1994.

[21] O. J. Huber and L. Toutain, "Mobile Agents in Active Networks", *ECOOP'97 Workshop Mobile Object Systems*, June 1997.

[22] J. T. Moore, M. Hicks, and S. Nettles. "Practical programmable packets". *Proceedings of IEEE InfoCom 2001*, April 2001.

[23] L. Yamamoto and G. Leduc. "An agent-inspired active network resource-trading model applied to congestion control". *In MATA 2000*, pages 151–169, September 2000.

[24] F. Berman, R. Wolski, S. Figueira, J. Schopf, and G. Shao. "Application-level scheduling on distributed heterogeneous networks". In *Supercomputing '96*, September, 1996.

[25] R. H. Saavedra-Barrera, A. J. Smith, and E. Miya. "Machine characterization based on an abstract high-level language machine". IEEE Transactions on Computers, December 1989.

[26] T. Ball and J. R. Larus. "Using paths to measure, explain, and enhance program behavior". IEEE Computer, July 2000.

[27] S. F. Bush and A. B. Kulkarni, Active Networks and Active Virtual Network Management Prediction: A Proactive Management Framework. ISBN 0-306-46560-4. Kluwer Academic / Plenum Publishers, 2001.

[28] M. T. Rose, The Simple Book: An Introduction to the Management of TCP/IP Based Internets, Prentice-Hall, 1991.

[29] S. Merugu, S. Bhattacharjee, E. Zegura, and K. Calvert, "Bowman: A Node OS for Active Networks". *Proceedings of Infocom 2000*, IEEE, March 2000.

[30] P. Tullmann, M. Hibler, and J. Lepreau, "Janos" A Java-oriented OS for Active Networks". *IEEE Journal on Selected Areas in Communications*, Vol. 19, No. 3, March 2001.

[31] J. Hartman, L. Peterson, A. Bavier, P. Bigot, P. Bridges, B. Montz, R. Piltz, T. Proebsting, and O. Spatscheck, "Joust: A Platform for Liquid Software", IEEE Computer, 1999.

[32] S. Schwab, "AMP – Enabling Active Networks via Secure Exokernel Implementations". NAI Labs Advanced Research Project Profile, January 2001. This is a two-page NAI Labs glossy brochure. See: http://download.nai.com/products/media/pgp/pdf/NAI-Labs-AMP-1-5-01.pdf

[33] P. Menage, "RCANE: A Resource Controlled Framework for Active Network Services". *Proceedings of the First International Working Conference on Active Networks* (IWAN '99), July 1999.

[34] B. Braden, A. Cerpa, T. Faber, B. Lindell, G. Phillips, J. Kann, and V. Shenoy, "Introduction to the ASP Execution Environment (Release 1.5)". November 30, 2001. Apparently, this is a USC/ISI Technical Report that has no number assigned. See: http://www.isi.edu/active-signal/ARP/DOCUMENTS/ASP_EE.ps

[35] V. Galtier, C. Hunt, S. Leigh, K. Mills, D. Montgomery, M. Ranganathan, A. Rukhin, and D. Tang, "How Much CPU Time?", *Draft NIST Technical Report*. TR-ANTD-ANETS-111999, November 1999.
See:http://w3.antd.nist.gov/~mills/whitepapers/NISTanetsTR.pdf