Expressing Meaningful Processing Requirements among Heterogeneous Nodes in an Active Network

Virginie Galtier
National Institute of Standards and Technology
Building 820, Room 445
Gaithersburg, Maryland 20899
1-301-975-3613
vgaltier@nist.gov

Kevin L. Mills, Yannick Carlinet, Stefan Leigh, Andrew Rukhin

{kmills, carlinet}@nist.gov {sleigh, rukhin}@nist.gov

ABSTRACT

Active Network technology envisions deployment of virtual execution environments within network elements, such as switches and routers. As a result, nonhomogeneous processing can be applied to network traffic associated with services, flows, or even individual packets. To use such a technology safely and efficiently, individual nodes must provide mechanisms to enforce resource limits. To provide effective enforcement mechanisms, each node must have a meaningful understanding of the resource requirements for specific network traffic. In Active Network nodes, resource requirements typically come in three categories: bandwidth, memory, and processing. Well-accepted metrics exist for expressing bandwidth (bits per second) and memory (bytes) in units independent of the capabilities of particular nodes. Unfortunately, no well-accepted metric exists for expressing processing (i.e., CPU time) requirements in a platform-independent form. This paper investigates a method to express the CPU time requirements of Active Applications (similar to distributed, mobile agents) in a form that can be meaningfully interpreted among heterogeneous nodes in an Active Network. The model consists of two parts: a node model and an application model. For modeling applications, the paper describes and evaluates a semi-stochastic state-transition model intended to represent the CPU usage requirements of Active Applications. Using measurement data, the general model is instantiated for two Active Applications, ping and multicast. The model instances are simulated, and the simulation results are compared against real measurements. For both Active Applications, the simulated and measured CPU time usage compare within 5% for the mean and the 90th and 95th percentiles. The 99th percentiles compare within 7%. The paper also evaluates three different scaling factors that might be used to transform a model accurate on one node into terms that prove accurate on another node.

Keywords

Active Networks, Resource Management, CPU Usage.

1. INTRODUCTION

Active Network technology envisions deployment of virtual execution environments within network elements, such as switches and routers. As a result, nonhomogeneous processing can be applied to network traffic associated with services, flows, or even individual packets. To use such a technology safely and efficiently, individual nodes must provide mechanisms to enforce resource limits. In order to provide effective enforcement mechanisms, each node must have a meaningful understanding of the resource requirements for specific network traffic. In Active Network nodes, resource requirements typically come in three categories: bandwidth, memory, and processing. Well-accepted metrics exist for expressing bandwidth (bits per second) and memory (bytes) in units independent of the capabilities of particular nodes. Unfortunately, no well-accepted metric exists for expressing processing (i.e., CPU time) requirements in a platform-independent form. This paper investigates a method to express the CPU time requirements of Active Applications in a form that can be interpreted among heterogeneous nodes in an Active Network.

Absent the capabilities we propose, Active Nets researchers have taken two main approaches to the problem. One approach assigns a time-to-live (TTL) to each packet, and then decreases the TTL at each hop, or each time a packet creates another packet. This TTL approach limits packets from excessively consuming global CPU resources in a network, but does not allow a particular node to protect itself. The second approach enforces an arbitrary limit on the CPU time that a packet can use at each node. Using an arbitrary limit ensures only a worse case upper bound, and can also allocate less CPU time than an application needs. The ideas proposed in this paper aim to provide Active Nets researchers with better information to allocate and control CPU usage for individual applications. These ideas might also be applied more generally, for example to heterogeneous distributed systems, where processes execute on nodes that exhibit a wide range of computational capabilities. Such heterogeneous systems could include systems based on mobile code, such as mobile agent applications, or systems based on code loaded from disk, such as distributed parallel processors. Wherever applied, metrics for CPU time requirements among distributed nodes will open the door to some essential uses.

CPU time requirements can be used to inform resource management decisions on a node. For example, a node might reject a new packet class if the processing requirements cannot be supported. Or, having accepted a new packet class, the node's operating system can monitor the CPU time used by each arriving

packet of the new class in order to halt the execution of packets that exceed the stated CPU resource requirements. Beyond admission control, new questions can be imagined relating to path routing decisions in an Active Network. For example, can a path be found through the network that can provide the CPU time required by an application, while also meeting the application's throughput, delay, and jitter requirements? As another example, given both application performance constraints and resource requirements (CPU time, memory, and bandwidth), can an application query an Active Network for multiple viable paths, each with an associated cost? The resource management questions in nodes become very interesting under these circumstances because taking on new processing loads can have a negative effect on the ability of nodes to meet the requirements promised for other applications. Nodes might need to condition such promises on a certain level of CPU utilization dedicated to an application. Then, resource management algorithms must police CPU utilization for each application, in addition to total CPU time used. This discussion illustrates that dynamically programmable networks will provide a range of interesting research questions that cannot be addressed unless CPU time can be interpreted among heterogeneous nodes.

The paper is organized into six main sections. We begin in Section 2 by considering the sources of variability in CPU time usage in an Active Network node. To the degree feasible any model proposed must account for these sources of variability. In Section 3, we review briefly the state of practice with regard to computer system performance benchmarks, and we provide a brief summary of research related to the problem addressed in this paper. In Section 4, we describe a semi-stochastic state-transition model to represent Active Applications, and we explain how the model can be used within a simulator to predict the mean and high percentiles of the CPU requirements for specific applications. Next, we explain how to scale this model from one Active Node to another, using scaling factors intended to capture the performance differences of each node. Section 5 compares results for two real Active Applications, ping and multicast, against results from simulating their parameterized models. Further, we compare measured results against simulated results obtained using three different factors to scale application models. Section 6 presents a critical view of our approach. Section 7 introduces ideas we need to explore to improve our work.

2. VARIABILITY IN CPU TIME USAGE

Any reasonable metric for an application's CPU time requirements must account for the major sources of variability affecting the application. In this section we identify and discuss the major sources of variability likely to affect the CPU time requirements of an active application. A proposed architecture for an Active Network node [1] identifies several components and the relationships among them. Figure 1 gives a conceptual overview of the major components and relationships. The components can be viewed as four layers: the hardware, the node operating system (Node OS) and interface, the execution environment (EE), and the active application (AA). Each EE provides a virtual execution environment (similar for example to a Java Virtual Machine [2]) in which AAs can execute. Several EEs have been defined and implemented within the Active Networks research community [e.g., see 3, 4, 5, 6, and 7]. In addition, several implementations of a Node OS are being developed [e.g., see 8, 9, and 10]. To enable any EE to run over any implementation of a Node OS, a standard application-programming interface, in the form of system calls, is defined within a separate Node OS specification [11].

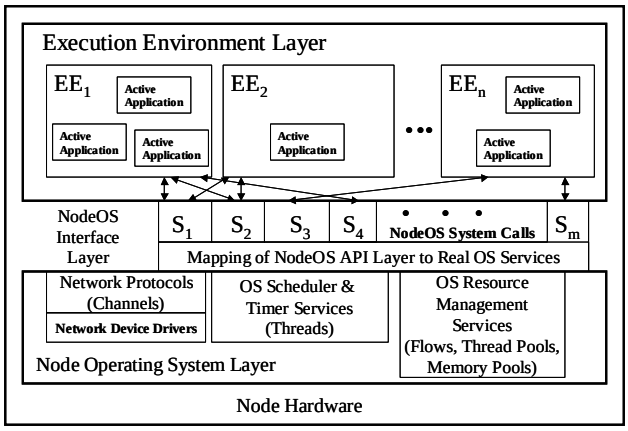

Figure 1. Conceptual Architecture of an Active Network Node

Our analysis of this model, and of real systems, reveals the main sources of variability affecting the CPU time requirements of an AA. In the hardware layer, the main factors that influence the execution time of an application include: the frequency of the processor, the architecture of the processor (e.g., Pentium, Pentium II, K6, Sparc, and so on), the amount of memory available on the host, the speed of the different buses (e.g., memory, I/O, and system), the technology of the persistent storage (SCSI or IDE hard drive, for instance), and the type of network card (e.g., 10 or 100 Mbps Ethernet). Within the node operating system (OS) and node OS interface layer, the main sources of variability include: the performance of the device drivers, the performance in managing processes and memory, and the nature and performance of the system calls provided by the operating system. In the case of networking system calls, performance of reads and writes also varies with the specific protocol stacks that are buried beneath the systems calls, and with the implementation of those protocols. Within the EE layer, performance can be affected by the mapping between the EE system calls and the OS system calls (a mapping usually defined by a library), as well as the compiler and options used to compile the EE. For example for a Linux system, if the EE uses the Java Virtual Machine (JVM), then we must consider the performance of the C library and the results from the C compiler used to compile the JVM. Finally, the execution of a specific AA can go through many paths in the code of the program. The path of execution taken can depend on many things, such as the state of the node (e.g., whether data is cached or not), the data carried (e.g., the length of the data to be processed), and even the state of other nodes (e.g., in an active multicast application an intermediate node creates and sends as many new capsules as the number of subscribed nodes). A detailed discussion of these issues appears elsewhere [48].

To corroborate our analysis, we measured the variability of a single EE running on three different hardware platforms. Table 1 gives the details of the platforms. We used an Active Application, ping, as a calibration workload to assess EE performance. We implemented a small calibration driver to measure the performance of the Linux system calls. The systems were measured in the absence of background load. The results appear in Tables 1 and 2.

Table 1. Average CPU Time Required for Calibration Workload on Three Platforms

Trait	Node A	Node B	Node C
CPU seconds	1.00	0.46	0.53
CPU speed	200 MHz	450 MHz	333 MHz
Processor	PentiumPro	Pentium II	Pentium II
Memory	64 Mbytes	128 Mbytes	128 Mbytes
OS	Linux 2.2.7	Linux 2.0.36	Linux 2.2.7
JVM	jdk 1.1.7B	jdk 1.1.7B	jdk 1.1.6

Table 1 shows the average number of user-mode CPU seconds taken for three different platforms to execute a calibration workload: acting as intermediate nodes for 3,000 ping capsules. Using as a predictor only the measured workload performance on Node B and Node C and their CPU speeds relative to Node A, given in rows one and two of Table 1, we might have expected Node A to take either 1.04 CPU seconds [based on Node B, Node A = (450 MHz / 200 MHz) * 0.46 CPU seconds] or 0.88 CPU seconds [based on Node C, Node A = (333 MHz / 200 MHz) * 0.53 CPU seconds] instead of the measured 1.00 CPU second. Clearly, factors other than CPU speed affect application performance. Table 2 shows that variability can also occur in the execution of system calls, perhaps depending on how the system call is implemented from OS-to-OS (or in this case even from version-to-version of the same OS). Other factors might include differences in devices, and thus device drivers, underlying specific system calls. An effective metric for CPU time usage in an Active Network node must account for these sources of variability. In Section 4, we investigate a model designed to respond to this challenge. First, we present a brief survey of some existing computer system performance benchmarks used by industry, followed by references to other related work.

Table 2. Average CPU Time Required for System Calls on Three Platforms

System Call	Microseconds		
	Node A	Node B	Node C
_llseek	6	4	3
close	11	9	7
fstat	6	4	2
link	31	32	22
open	19	32	11
read	13	7	8
socket	21	13	14
stat	15	12	9
unlink	28	19	20
write	6	21	3

3. RELATED WORK

Over the years, the computer industry has developed many benchmarks to evaluate the performance of computer systems. The most successful among these evolve over time to account for changes in underlying hardware performance and for shifts in user interest. While these industry benchmarks have different characteristics, they share a common purpose: to compare performance among computer systems. None of these expresses the CPU time requirements for specific applications. Still, measurements of performance differences among computer systems provide one essential ingredient in any design that attempts to meaningfully express an application's CPU time requirements among heterogeneous nodes. For this reason, examining existing industry benchmarks has proven valuable to us. We have also considered related research that attempts to model the performance of computer programs. In addition, we have surveyed the current state-of-the-art with respect to CPU resource control in Active Networks. Readers uninterested in this background can move directly to Section 4.

3.1 Performance Benchmarks

While a few benchmarks, such as composite theoretical performance (CTP) calculated for purposes of export control [24], use static computations, most benchmarks entail dynamic execution of a workload. As a main division, most dynamic benchmarks are either synthetic or real, while a few hybrid benchmarks mix both elements. Synthetic benchmarks contain artificial programs designed to mimic characteristics that the designers believe would be exhibited by real programs from a population of interest [12-23]. These programs are artificial in that they produce no useful results outside the benchmark. Real benchmarks contain programs from a population that the designers either: (1) know is the population of interest or (2) believe behave similarly to programs within the population of interest [25-30]. The programs used in real benchmarks produce useful results when used outside the context of the benchmark. Some hybrid benchmarks consist of a mix of synthetic and real benchmarks [31, 32].

3.2 Program Models

Computer science researchers and practitioners have explored the use of graph theory and Markov models to represent computer systems and programs and to predict performance characteristics at both micro and macro levels [35-42, 45, 49]. These previous explorations have guided our work. Other researchers have investigated additional techniques to analyze program code in order to predict the performance of programs on various computer systems [33-34, 43-44, 46]. Of particular interest, Saavedra and Smith [33, 34] developed a machine-independent model of program execution that formed the basis for characterizing both machine performance and program profile. While this is an appealing approach, the workload characterization, and thus the resulting performance prediction, is valid for only a particular instance of execution behavior, that is, a single run of a program. In our case, we need a model that will capture many possible executions of a program.

3.3 CPU Usage Control in Active Networks

The Active Network architecture document [1] proposes to use RISC (reduced instruction set computer) cycles as a unit of CPU resource measurement; however, the document does not explain how to convert RISC cycles into a meaningful figure on an actual node. The document also does not explain how to determine the number of RISC cycles needed to execute specific active applications. In light of this situation, most execution environments implement their own mechanism to protect CPU resources. For example, some systems, such as ANTS [3] and PLAN [50], assign a time-to-live (TTL) to each active packet, while other systems, such as Magician [51], Smart Packets [52], and ANTSv1.3 [53], limit that amount of CPU time any packet can use at each node. In both approaches, limits are assigned arbitrarily, regardless of an application's requirements.

4. MODELING CPU USAGE

In order for a node operating system (Node OS) to efficiently manage the utilization of its CPU, each active application (AA) must declare the CPU time required for its execution. In a simple approach, an application might declare the average CPU time required and the variance. We believe statistics for the higher percentiles of CPU time required by an application could prove more useful for resource enforcement and estimation. With an aim to characterize both the average and high percentile CPU time requirements of applications, we investigated a fine-grained model, as presented in this section. First, we devised a general model for the CPU requirements of AAs, and then we used measurements to instantiate our model for two specific AAs, ping and multicast, executing on Node A (see Table 1). Second, we simulated our model instances; validating them against results obtained from measurements on Node A. Third, we used several different techniques to scale our model instances to reflect the CPU time usage we expected to see on Node C. We then compared the results from simulating our scaled models against the CPU time usage obtained from measuring real executions of the same applications on Node C. The details follow.

4.1 Modeling Active Applications

Recall from Figure 1 that an AA executes in user mode within an execution environment (EE), but requests services periodically from the node operating system through specific system calls. An observer, situated at the boundary between an EE and the Node OS, would view the behavior of an AA as a series of transitions between specific system calls: from an initial state, the application executes in user mode for some amount of CPU time within its EE and then executes in kernel mode for some amount of CPU time within a system call, then again in the EE before transitioning to another system call, and so on until the active packet is processed and the AA has returned to its initial state. We call each execution cycle a scenario. We are interested in modeling the total CPU time taken for each scenario, as well as the distribution of CPU time used by an application when executing a typical mix of scenarios.

To generate models from real applications, we built a software monitor that can observe the transitions between system calls. The monitor logs a trace of system calls made to process an active packet, along with CPU time information. Each transition lists the previous system call and the upcoming system call, along with the CPU time used by the previous system call, and the CPU time used by the EE between the calls. The sequence of system calls may vary from one execution to another because several paths are possible within the processing code. We label each different sequence as an execution scenario. To ensure that the model captures a representative coverage of the AA, the AA developer is responsible for running the AA through the expected mix of scenarios. Using the resulting execution trace, we construct the model of an AA.

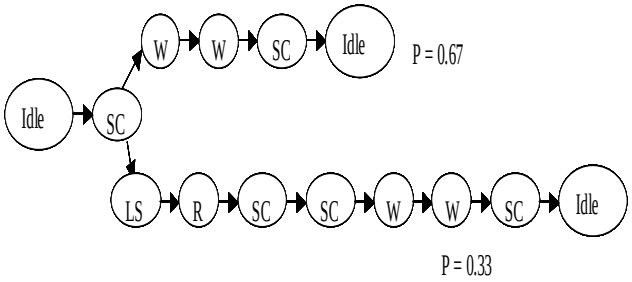

Figure 2. Stochastic State-Transition Graph Representing Two Scenarios in an Active Application (System Calls: LS = _llseek; R = read; SC = socket call; W = write) (P = the probability of executing a specific scenario)

An AA model consists of two types of information: scenario representation and workload representation. Each scenario (see Figure 2) is represented by its sequence of system calls. Further, each system call is characterized by the distribution of the CPU time spent in the system call, as measured from the execution trace. In addition, each transition is characterized by the measured distribution of CPU time spent in the EE during the transition, also as measured from the execution trace. 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 usage with histograms (see Figure 3). Note that this approach increases significantly the volume of information that must be exchanged when AA models are transferred among nodes in a network. (Our experiments show that at least 25 bins per histogram are required to obtain reasonable results.)

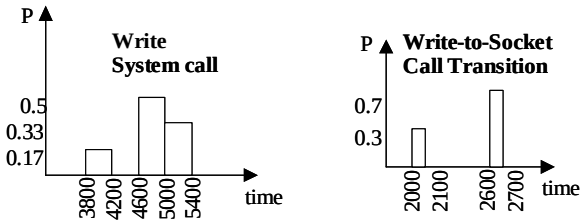

Figure 3. Sample Histograms Representing the CPU Time Requirements for the Write System Call and a Write-to-Socket-Call Transition (P is the Probability of a Specific Bin in a Histogram)

The AA workload is represented as a list of the possible scenarios, where each scenario is assigned a probability of occurrence (based on the frequency with which the scenario appeared in the execution trace). The scenarios, their probability of occurrence, and the distributions of user and system CPU time usage constitute the AA model, which is transmitted among nodes in the network, and used to estimate the execution time needed at each node

4.2 Simulating CPU Requirements

To validate the predictive ability of our AA models, we used Monte-Carlo simulation. Each pass through the simulator represents the processing of an active packet. Using the probability of occurrence contained in the AA 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 CPU time distribution. The sum of the CPU times of each component in the scenario yields a simulated execution time. After repeated scenario executions, we obtain an estimate for the mean and the high percentiles of the CPU time requirements of the AA.

4.3 Overcoming Node Heterogeneity

Recall that the CPU time values included in the model were generated from an execution trace measured on a specific node. For this reason, these values likely have no meaning on other nodes in the network. We need to scale the values in the model to a form that will have meaning on these other nodes. Since the model consists of CPU time values related to system calls (kernel mode) and EE execution (user mode), we need to devise transformation factors that can scale independently the kernel-mode and user-mode times. (The reasons for this were discussed earlier in Section 2.)

Our approach is to provide workloads that each node can use to calibrate itself with respect to performance of EEs contained on the node, and also with respect to performance of system calls. After executing the calibration workloads, a node obtains two vectors. The EE vector gives the average user-mode CPU time taken by the node to execute a representative workload for each type of EE. The system-call vector gives the average CPU time taken to execute each Node OS system call. (We discuss in another paper [55] our progress on node calibration.) To allow the transformation of AA models to scale across a large population of nodes, we select a specific node as a reference.

The calibration vectors for the reference node are distributed to all nodes in the network. Prior to transferring the model of an AA (running on a specific execution environment, EE1) between two nodes x and y, the model is subjected to a "Node-to-Reference transform": the histograms describing the time spent between two system calls are dilated or contracted using the ratio Tref/Tx, where Tref is the average time taken by EE1 to execute the calibration workload on the reference node and Tx is the average time taken by EE1 to execute the calibration workload on node x. The histograms describing the time spent in each system call are transformed in the same manner using the system-call vectors. The model, with its CPU time requirements expressed in terms of a reference node, is then transmitted across the network. Upon arrival at node y, the model is subjected to an inverse (the ratio is Ty/Tref) "Reference-to-Node transform". The combination of these two transforms scales the CPU times within an application model from a form meaningful on node x into a form meaningful on node y.

5. Results

We used our approach to construct models for two active applications (ping and multicast) written for the EE ANTS. First, we validated our models by comparing simulation results against measured results obtained on Node C (see Table 1). Second, we investigated three scaling factors to transform our models for use on Node A. Then we simulated the scaled models, and compared the simulation results against measured results obtained when running the real applications on Node A. The details follow.

5.1 Validating the Models

We selected the relevant scenarios for each of the applications, and then exercised each application on Node C until 20,000 active packets were processed. The mean and high percentiles of execution time (expressed in CPU clock cycles) measured from these test sequences appear in Table 3 below. Row one provides the measured values for ping and row three provides the measured values for multicast.

We used these same execution traces to construct the models. When constructing and using a model, two parameters can influence the results: (1) the number of bins chosen for the histograms and (2) the number of simulation runs. Our experiments showed that, independent of the number of runs, 25 bins per histogram appears to be the minimum needed to obtain results within 5% of the real values. With 100 bins per histogram, 1,000 runs of the simulator proved sufficient to provide estimated values for mean and high percentiles with the accuracy shown in Table 3. For both applications, the simulated and measured CPU time usage compare within 5% for the mean and the 90^{th} and 95^{th} percentiles. The 99^{th} percentiles compare within 7%.

Table 3. Validation Results for Active Applications Ping and Multicast on Node C

	Thousands of CPU Cycles			
Experiment	**Mean**	**90%**	**95%**	**99%**
Measured Ping	176	197	211	263
Simulated Ping	170	190	202	269
Measured Multicast	143	169	180	201
Simulated Multicast	144	165	173	187

5.2 Validating the Scaled Models

Next we tried to scale our Node C models for ping and multicast for use on Node A. Scaling was accomplished using the transformations described above (see Section 4.3). First, we compared two different scaling factors: (1) MHz scaling, using the ratio of CPU speeds on the two nodes (as given in Table 1) and (2) Workload scaling, using the ratio of the times taken by the two nodes to execute the preliminary calibration workloads (as given in Tables 1 and 2). Our working hypothesis was that the first ratio would yield poor results, but that the second ratio would yield better results.

Table 4 gives the results we obtained. As expected, scaling the model using the ratio of CPU speeds (results in rows two and six of Table 4) yields poor correspondence with the measured results (rows one and five in Table 4). The errors range from about 40% to as much as 60%. Unfortunately, scaling the model using the ratio of CPU time needed to execute the calibration workloads yielded equally poor results (row three of Table 4) for the ping application (errors range from 30% to 80%), and for multicast (results not included in the table). One possible reason is that the preliminary calibration workload does not provide a good representative workload. This issue is under investigation. Given the poor results from both MHz and Workload scaling, we adopted a third approach to obtain a scaling factor. We call this approach Inspection scaling. We simply varied the value corresponding to the average time to execute a calibration workload on ANTS on each node until we found a ratio giving better results for ping (see row four in Table 4). We achieved accuracy within 2% for the mean and the 90th and 95th percentiles, and within 9% for the 99th percentile. The same ratio turns out to give reasonable results for multicast (see row seven in Table 4). In the case of multicast, all the statistics correspond within 8%.

Table 4. Validation Results after Scaling Ping and Multicast Models to Node A (M = MHz scaling; W = Workload scaling, I = Inspection scaling)

	Thousands of CPU Cycles			
Experiment	**Mean**	**90%**	**95%**	**99%**
Measured Ping	206	229	237	274
Simulated Ping-M	293	320	342	440
Simulated Ping-W	367	358	327	357
Simulated Ping-I	210	229	241	298
Measured Multicast	171	186	198	239
Simulated Multicast-M	236	276	289	311
Simulated Multicast-I	172	200	210	228

6. DISCUSSION

Two topics warrant further discussion. First, the approach outlined in this paper has some limitations. Second, the issue of node calibration requires further exploration.

6.1 Limits of the Approach

The limits of our current approach fall into two main categories. First, our model lacks space and time efficiency. A specific model can be huge: if a lot of different paths exist through the processing code of an active packet, if the paths are long, and if the histograms are fine-grained. These issues can be overlooked to some degree because the envisaged AAs are designed for processing network packets, rather than performing arbitrary distributed computation. Thus, we can reasonably expect an AA to have a small number of scenarios, and each scenario to be short. That said, since AA CPU usage profiles do not match well the known statistical distributions, we have had to resort to simulation in order to obtain estimates of CPU time usage. Using simulation to estimate the high percentile of CPU usage can be costly. For this reason, simulation is interesting only for AAs that will execute for a long period on a node. An analytical solution would be much faster. To date, the analytical approximations that we have investigated yield much less accurate results than simulation.

The second limitation of our approach is more conceptual. Our model captures only the behavior of the application as exercised during the generation of execution traces. Unless the AA developer exercises good judgment, the execution traces can give a distorted representation of the application. Even when the AA developer exercises good judgment, the conditions existing on a node or in a network of nodes might well prove different than the conditions present when the execution trace was generated. Perhaps this limitation could be overcome if a model is allowed to evolve as it travels from node to node and gains experience (an active model!). Such an active model should also be parameterizable to capture the fact that an AA may execute loops a varying number of times based on parameter values local to the node. For example, a multicast application might perform some operations once per each multicast user known to a node. Information such as this cannot be known in advance, but would be available on each node.

6.2 Calibration Issues

The results of our transformation attempts led us to consider several alternative approaches. First, we could try to generate a different calibration workload that better represents AA processing. But we might instead want to rethink the calibration process. We could deploy a calibration application on each node, and then measure the mean and 95th percentile of the application's CPU usage. Subsequently, we choose one node as a reference and then instantiate a simulation model of the calibration application, as it executes on the reference node. This model would be distributed to all nodes. Node calibration would comprise finding a scaling factor that yields a close correspondence for the calibration application between simulated and measured CPU usage for the mean and 95th percentile statistics.

Another solution to the model transformation would be to calibrate by node and by application. In that way, each node would have its own model for a given application, and the model would not be transmitted through the network. This solution reduces network traffic at the cost of memory at each node. In addition a long initialization step would be required, and of course the approach does not scale to large and highly dynamic networks. Still, this approach could be used for AAs deployed by a network operator for a long period of time. In that case, static limits could be enforced for other classes of AA.

7. FUTURE WORK

Our future work on this project will evolve along two lines: (1) confirming and extending our current results and (2) investigating techniques to overcome the limitations of our existing approach. We have discovered a scaling factor that successfully transforms models for two AAs, ping and multicast, between two different nodes. We must confirm that this same scaling factor applies for other AAs. Further, we must show the existence of scaling factors that can successfully transform AA models among a variety of nodes. If such scaling factors exist, then we must develop a calibration technique aimed at discovering the scaling factors for specific nodes. Finally, since some aspects of AA processing depend on conditions present on particular nodes, we must extend

our model to enable parameterization that can account for node-dependent conditions. Such parameterization would likely require us to include the possibility of loops within our transition graphs.

Beyond confirming and extending our current approach, we must investigate AA models that exhibit improved space and time efficiency. We are exploring a nonparametric technique using the Epanechnikov kernel [54] to estimate a quantile function, given a set of representative data points. To date, we have applied the technique to analyze nineteen data sets drawn from five applications running on four nodes. In general, we have found that this technique yields a good correspondence between its estimates and the input data set, provided two conditions hold: (1) the input data set contains at least 200 sample points and (2) the distribution of data points is not overly discrete. We imagine that this technique might be applied to represent the CPU usage of an application with as few as 1,000 data points. Such a representation would be much more compact than our current model. In addition, using the estimated quantile function for high percentiles might well require less CPU time than needed to simulate 1,000 data points using our current approach. Further, modeling CPU time as a flat set of observations could provide a nice basis on which to evolve the model over time, based on actual measurements as the modeled application moves throughout the network. Even if these hypotheses are confirmed, we still need to develop suitable scaling factors to transform the sample data points into a form meaningful on various nodes, and we still need to account for node-dependent conditions.

8. CONCLUSIONS

The work reported in this paper is motivated by the search for an effective technique for Active Network (mobile code) applications to express CPU time requirements in a form that can be meaningfully interpreted on heterogeneous nodes distributed in a network. The paper identifies and discusses the major sources of variability likely to affect the CPU time requirements of an Active Application. To support that analysis, the paper reports results from measuring CPU time usage for virtual machine executions and for system calls on three Linux nodes. The paper describes a semi-stochastic model that can be used to represent the CPU time requirements of an Active Application. Measured execution traces from real Active Applications (ping and multicast) were used to construct instances of this model. Simulation results show that the model can give a reasonably accurate representation of the behavior of an Active Application, provided the data used to construct the model contain a faithful representation of the mix of scenarios composing the application.

We also present a model capturing variations in node performance. We planned to use this model in conjunction with algorithms to transform Active Application models among heterogeneous nodes in a network. The poor results obtained with our proposed scaling factor illustrate the difficulty of generating a representative calibration workload for Active Applications. But we show that a ratio leading to better results exists, and we discuss other techniques to calibrate an active node.

While the model investigated in this paper still has strong limitations, we plan to continue our research, searching for more effective models. Without such models, it will prove impossible to effectively manage CPU resources in distributed applications based on mobile code. The urgency to develop models of CPU time requirements for software applications will increase with the increasing use of mobile code in distributed software systems. In such systems, models must provide reasonably accurate representations of the expected CPU time usage of an application. Such models must also be capable of meaningful interpretation among heterogeneous nodes in a network. In our research to date, these requirements have proven difficult to meet. We plan to continue our search. We urge other researchers to join us.

9. ACKNOWLEDGMENTS

The work discussed in this paper was conducted under joint funding from the Defense Advanced Research Projects Agency (DARPA) and the National Institute for Standards and Technology (NIST). The authors thank Hilary Orman for her insightful comments during discussions that led to the beginning of this project. In addition, the authors thank Doug Maughan for his continued support of the work. Finally, we thank the anonymous reviewers for taking time from their own research to constructively criticize ours.

10. REFERENCES

[1] K. Calvert (ed.), Architectural Framework for Active Networks, Version 0.9, Active Networks Working Group, August 31, 1998.

[2] T. Lindholm and F. Yelling, The Java Virtual Machine Specification, Addison-Wesley, Reading, Mass., 1997.

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

[4] D. S. Alexander, W. A. Arbaugh, M. W. Hicks, P. Kakkar, A. D. Keromytis, J. T. Moore, C. A. Gunter, S. M. Nettles, and J. M. Smith, "The SwitchWare Active Network Architecture", *IEEE Network Special Issue on Active and Controllable Networks*, vol. 12 no. 3, pp. 29 - 36.

[5] 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.

[6] Samrat Bhattacharjee, Kenneth L. Calvert and Ellen W. Zegura. "An Architecture for Active Networking", *Proceedings High Performance Networking (HPN'97)*, White Plains, NY, April 1997.

[7] 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.

[8] 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.

[9] D. Decasper, G. Parulkar, S. Choi, J. DeHart, T. Wolf, and B. Plattner, "A Scalable, High Performance Active

Network Node", *IEEE Network*, January/February 1999.

[10] B. Ford, G. Back, G. Benson, J. Lepreau, A. Lin, 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.

[11] L. Peterson (ed.), NodeOS Interface Specification, Active Networks Node OS Working Group, February 2, 1999.

[12] R. P. Weicker, "A detailed look at some popular benchmarks", *Parallel Computing*, No. 17, 1991, pp. 1153-1172.

[13] R. P. Weicker, "Dhrystone Benchmark: Rationale for Version 2 and Measurement Rules", *SIGPLAN Notices*, 23, 8, August 1988, pp. 49-62.

[14] H. J. Curnow and B. A. Wichmann, "A Synthetic Benchmark", *The Computer Journal*, 19, 1, 1976, pp. 43-49.

[15] TPC Benchmark A Standard Specification, Revision 2.0, Transaction Processing Performance Council, June 7, 1994.

[16] TPC Benchmark B Standard Specification, Revision 2.0, Transaction Processing Performance Council, June 7, 1994.

[17] TPC Benchmark C Standard Specification, Revision 3.4, Transaction Processing Performance Council, August 25, 1998.

[18] TPC Benchmark D Standard Specification, (Decision Support) Revision 2.1, Transaction Processing Performance Council.

[19] K. Shanely, History and Overview of the TPC, Transaction Processing Performance Council, February 1998.

[20] TPC Benchmark H Standard Specification, (Decision Support), Revision 1.1.0, Transaction Processing Performance Council.

[21] TPC Benchmark R Standard Specification, (Decision Support), Revision 1.0.1, Transaction Processing Performance Council.

[22] TPC Benchmark W (Web Commerce), Public Review Draft Specification, Revision D 5.0, July 12, 1999.

[23] "WinTune98", *Windows Magazine*, Version 1.0.39, June 24, 1999, 1.75 Mbyte download.

[24] Export Administration Regulations: Technical Note to Category 4, Computers: Supplement No.1 to Part 774.

[25] SPEC CPU95 Benchmarks, Standard Performance Evaluation Corporation, June 24, 1999.

[26] "BAPCo Debuts First Benchmarking Software for Computers Running Windows*98" press release from Business Applications Performance Corporation, Santa Clara, CA, August 26, 1998.

[27] "PC Magazine Labs Benchmarks Tests: Winstone99", *PC Magazine On-line*, ZDNet, December 1, 1998.

[28] "NetBench 6.0", *ZDNet*, 1999.

[29] "WebBench 3.0", *ZDNe*t, 1999.

[30] "SPEC Announces SPECweb96, Industry's First Standardized Benchmark for Measuring Web Server Performance", press release from Standard Performance Evaluation Corporation, July 22, 1996.

[31] "WinBench 99", *ZDNet*, 1999.

[32] "PC Magazine Labs Benchmark Tests: CPUmark99", *PC Magazine On-line*, ZDNet, December 1, 1998.

[33] R. H. Saavedra-Barrera, A. J. Smith, and E. Miya, "Machine Characterization Based on an Abstract High-Level Language Machine", *IEEE Transactions on Computers*, Vol. 38, No. 12, December 1989, pp. 1659-1679.

[34] R. H. Saavedra and A. J. Smith, "Analysis of Benchmark Characteristics and Benchmark Performance Prediction", *ACM Transactions on Computer Systems*, Vol. 14, No. 4, November 1996, pp. 344-384.

[35] Boris Beizer, Micro-Analysis of Computer System Performance, Van Nostrand Reinhold Company, 1978.

[36] William S. Bowie, "Applications of Graph Theory in Computer Systems", *International Journal of Computer and Information Sciences*, Vol. 5, No. 1, 1976, pp. 9- 31.

[37] R. M. Karp, "A note on the application of graph theory to digital computer programming", *Information and Control*, Vol. 3, No. 6, 1960, pp. 179-190.

[38] C. V. Ramamoorthy, "Discrete Markov analysis of computer programs", in *Proceedings of the 20th ACM National Conference*, 1965, pp. 386-392.

[39] J. D. Foley, "A Markovian model of the University of Michigan execution system", *Communications of the ACM*, Vol. 10, No. 9, 1967, pp. 584-589.

[40] W. S. Bowie, Towards a distributed architecture for OS/360, PhD Thesis, Department of Applied Analysis and Computer Science, University of Waterloo, 1974.

[41] M. A. Franklin and R.K. Gupta, "Computation of Page Fault Probability from Program Transition Diagram", *Communications of the ACM*, Vol. 17, No. 4, 1974, pp. 186-191.

[42] P. Hoschka, "Compact and Efficient Presentation of Conversion Code", *IEEE Transactions on Networking*, Vol. 6, No. 4, 1998, pp. 389-396.

[43] B. Wiegbreit, "Mechanical Program Analysis", *Communications of the ACM*, Vol. 18, No. 9, 1975, pp. 528-539.

[44] T. Hickey and J. Cohen, "Automating Program Analysis", *Journal of the ACM*, Vol. 35, No. 1, 1988, pp. 185-220.

[45] G. Ramalingam, "Data Flow Frequency Analysis", *ACM SIGPLAN NOTICES*, Vol. 31, No. 5, 1996, pp. 267-277.

[46] V. Balasundaram, G. Fox, K. Kennedy, and U. Kremer, "A Static Performance Estimator to Guide Data Partitioning Decisions", *Proceedings of the Third ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming*, 1991, pp. 267-277.

[47] M. Kijima, Markov Processes for Stochastic Modeling, Chapman and Hall, London, 1997, especially pages 168-229.

[48] Virginie Galtier, Craig Hunt, Stefan Leigh, Kevin L. Mills, Doug Montgomery, Mudumbai Ranganathan, Andrew Rukhin, and Debra Tang, "How Much CPU Time?", Draft NIST Technical Report TR-ANTD-ANETS-111999, November 1999.

[49] Connie U. Smith, Performance Engineering of Software Systems, Addison-Wesley, Reading, Mass., 1990.

[50] Michael Hicks, Pankaj Kakkar, Jonathan T. Moore, Carl A. Gunter and Scott Nettles, "PLAN: A Packet Language for Active Networks", *Proceedings of the International Conference on Functional Programming (ICFP)'98.*

[51] 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*.

[52] Beverly Schwartz, Alden W. Jackson, W. Timothy Strayer, Wenyi Zhou, Dennis Rockwell and Craig Partridge, "Smart Packets for Active Networks", Proceedings of OpenArch 99, March 1999.

[53] While inspecting the code for ANTS v1.3, we discovered the addition of a "ChannelWatchDog" thread, which halts long-running ANTS capsules.

[54] R.-D. Reiss, Approximate Distributions of Order Statistics With Applications to Nonparameteric Statistics, Springer-Verlag, New York, 1988 (see specification Chapter 8).

[55] Y. Carlinet, V. Galtier, K. Mills, S. Leigh, A. Rukhin, "Calibrating an Active Network Node", *Proceedings of the 2nd Workshop on Active Middleware Services*, August 2000.