Resource Control and Estimation Based Fair Allocation (EBFA) in Heterogeneous Active Networks

K. Vimala Devi
Department of Computer Science and Engineering
Anna University, Trichirappalli,
Trichirappalli, 620024, India.
E-mail: k_vimadevi@yahoo.co.in

C. Thangaraj
Kalasalingam University,
Krishnankoil – 626190, India.
E-mail: thangaraj@akce.ac.in

K.M. Mehata
Anna University,
Chennai - 600025, India
E-mail: mehata@annauniv.edu

Keywords: active network management, resource control, fair allocation, resource estimation, estimation based allocation

Received: July 11, 2009

Active networks perform customized computation on the messages flowing through them. Individual packets carry executable code, or references to executable code. Active networks are changing considerably the scenery of computer networks and consequently, affect the way network management is conducted. In a heterogeneous networking environment, each node must understand the varying resource demands associated with specific network traffic. This paper describes and evaluates an approach to control the CPU utilization of malicious packets and to estimate the CPU demand for good packets in a heterogeneous active network environment. We also describe a new approximation for estimation based fair allocation. The proposed algorithm called Estimation Based Fair Allocation Algorithm (EBFAA) avoids the ill-behaved flows to utilize more CPU time and achieves perfect fairness for all flows during allocation.

Povzetek: Prispevek opisuje obravnavo zlonamernih paketov v aktivnih heterogenih mrežah.

1 Introduction

In classical packet-switched communication networks, when a packet transits through an intermediate node along the path from source to destination, each intermediate node has a measured rating for per-message and per-byte throughput. Thus a linear extrapolation from packet size and arrival rate should provide the node a reasonable estimate for the 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 [15], when a packet arrives at an intermediate node, the data may include program code that can be accessed, interpreted, and executed by the node. 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. Thus, in active networks, some more sophisticated technique is needed to estimate CPU demand associated with active packets.

1.1 Active network architecture

Active networks [15] allow individual user, or groups of users, to inject customized programs into the nodes of the network. "Active" architectures enable a massive increase in the complexity and customization of the computation that is performed within the network

- Node operating system (node os)

A NodeOS [7,9] is a special-purpose operating system that runs on the routers of an active network and supports active network execution environments (A router in an active network is called an **active node**, and hence the name NodeOS). In order to prevent active applications

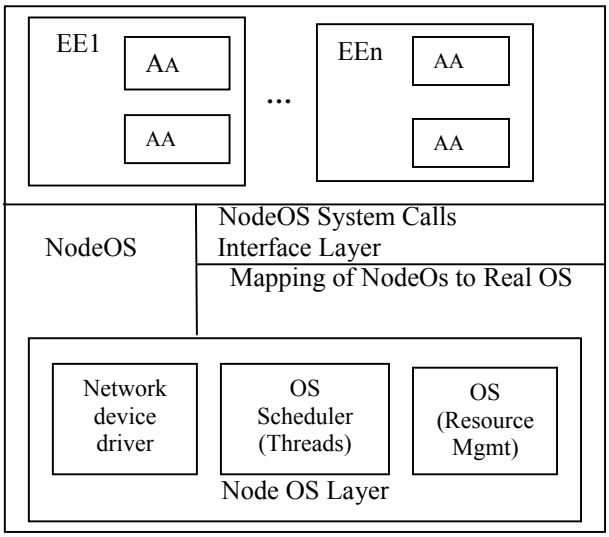

Figure 1: Active-network architecture [3, 5].

from misbehaving, active network execution environments enforce fine-grained control over the resources consumed by active applications. For example, an execution environment may restrict the number of CPU cycles an active application can consume, or it may enforce a limit on the number and type of packets an active application can receive and send. The interface/API provided by traditional operating systems is inadequate for such needs of execution environments. For example, in traditional Unix, where all resources are associated with a **process**, it is very difficult to enforce an absolute limit on resources consumed by an active application if it is not a process, and active networks would be very slow if each active application is run in a separate Unix process. A NodeOS provides the exact interface needed by active network execution environments. A NodeOS is also different from a traditional OS in terms of the overhead it imposes to do its job. Further, a NodeOS should be capable of handling as many network packets per second as possible. Therefore, the NodeOS should impose minimum overhead to perform operating system functions. The above requirements raise interesting operating system design issues, primarily in the areas of API design and effificent resource control.

The Node operating System (NodeOS) provides the basic functions from which Execution Environments build the abstractions that make up the network APIs. The NodeOS isolates EEs from details of resource management and the existence of other EEs. The EEs in turn, hide most of the details of their interaction with the end user from the NodeOS. The NodeOS defines four primary abstractions: threads pools, memory pools, channels and flows. The first three encapsulate a system's three types of resources: computation, storage and communication. The fourth is used to aggregate control and scheduling of the other three in a form that more closely resembles network application programs[11].

Examples of NodeOS: -Scout and Amp. A number of other NodeOS implementations, such as xbind and EROS, are also under development and testing.

- ***Execution environment (ee)***

Active networks rely on the ability to add programs easily to the network infrastructure, so the choice of the Execution Environment's runtime environment and programming language is critical. Below are some of the Execution Environments for setting up of Active Networks.

1. ***Ants: Active Node Transfer System***

The Massachusetts Institute of Technology's ANTS aims at standardizing on a communication model rather than individual communication protocols, such as IP, UDP etc. The major design goal is to build a system that allows rapid transfer and deployment of protocol code across the network. ANTS uses Java as its programming language, and the Java Virtual Machine as its runtime environment. Java's features make ANTS suitable for a variety of applications [19].

2. ***Magician***

Magician [1], a toolkit for creating a prototype Active Network was developed at the University of Kansas. In an Active Network, program code and data is placed inside specialized packets called SmartPackets. The nodes of an Active Network are called active nodes and they are programmable in the sense that when a SmartPacket reaches an active node, the code inside the SmartPacket is extracted and executed. Depending on the nature of the code inside the SmartPacket, the SmartPacket either modifies the behavior of the active node or transforms the data it is carrying. The basic implementation uses UDP/IP combination for transport and routing.

1.2 Current network management and its limitations

Currently, networks are monitored and controlled mainly through SNMP commands that read or set variables in the MIBs of the elements. Current MIB implementations, which defined by their manufacturers, have several significant limitations.

1. A well-known limitation of SNMP is related to its inability to handle high volumes of processed network data.
2. Another limitation of the current management techniques is that all management decisions are usually made centrally. This approach is inefficient when the network is congested, or when a part of it is inefficient when the network is congested, or when a part of it is unreachable, since the management commands may arrive late or get lost. Active nodes can be programmed to make such decisions, thus allowing the distribution of the decision centers across the network [8, 12].

1.3 Suitability of active networks for network management

The use of Active Networks technologies to network management has the following advantages:

- The information returned can be controlled and managed according to needs.
- The management rules can be shifted from the management centers to the active nodes.
- The monitoring and control loop is shortened.

Active networks for the functional areas of network management

The functional areas of Network Management are Fault Management, Configuration Management, Accounting Management, Performance Management and Security Management (FCAPS).

- ***Fault management***

In fault management as well as other areas of Network Management such as configuration and performance management, predicting and preventing undesirable situations is important. Current predictive algorithms take into consideration only a few parameters. However active network technologies enable deployment of efficient predictive management, since the computations can be distributed to the whole network. Each node predicts and transmits to its neighbors its future state and also the prediction of each node depends on its current state and the predictions of its neighbors. Congestion can also be predicted with satisfactory accuracy [12].

- ***Configuration management***

Configuration management techniques may be enhanced in an AN environment. For instance, MAs can be used for inventory management. Those MAs can be used to discover and report changes to the existing configuration. For example agents could be programmed to propagate DNS updates to the entire network.

AN can also facilitate VPN deployment. VPNs are independent private networks built over a shared public network. Practically this means that network resources are partitioned and allocated (dynamically or statically) to each group. In AN access to the resources of active nodes can be controlled; hence partitioning of resources can easily be implemented. An attempt in this direction is Virtual Active Network (VAN) architecture [12].

- **Accounting management**

One of the important tasks accounting management tools carry out is monitoring network usage. Most AN architectures, for security and safety reasons, authenticate the users before any resources are allocated to them to access any service. Thus the monitoring of the resources is integrated to the network architecture, rather than being an additional function. With AN, all resource usage, such as bandwidth, CPU, memory, or scheduling priorities, can be accounted.

Finally, AN may be manageable even when some areas cannot be reached by the management stations. This is crucial for accounting management, because those situations usually lead to unreported network use, and therefore loss of profit. Such situations can be prevented in active environments [12].

- **Performance management**

With AN, the way devices handle traffic can easily be customized on a per-device and per-user basis. Hence scheduling and routing, traffic shaping, admission control, and priorities can easily be controlled in order to manipulate traffic. The deployment of QoS services can easily be achieved in AN, since protocols that perform the necessary reservations and computation can be installed on active nodes. AN is also flexible in installing protocols. Complex QoS protocols, such as Resource Reservation Protocol (RSVP) or qGSMP, could be deployed easily too: the reservation of resources and scheduling algorithms of active nodes can be manipulated in any desired way. The ability to implement QoS protocols without relying on legacy and rigid protocols (e.g., IP) makes those protocols lightweight and efficient [12, 19].

- **Security management**

The AN architectures implement modules that relate to security and safety. These modules authenticate access to resources hence; several of the current security management tasks are architecturally integrated in to these modules. This relieves NM tools from the enforcement of policies and SLAs. Apart form traditional policing; intrusion detection can become much easier and effective by agents that reside on sensitive nodes. Attacks such as TCP SYN attack can also be effectively detected and prevented. For instance Phonix framework allows the existence of MAs that are programmed to perform specific tasks, such as safeguarding the network. [12]

1.4 The resource demand in active networks

Performance management aims to keep network performance within predefined levels. It is strongly related to resource management and QoS provisioning [12, 17] and to the parameter, resource utilization. Performance management tools measure various parameters, such as network throughput, delays, and CPU and bandwidth utilization, and attempt to control them. To use the Active Network technology safely and efficiently, individual nodes must understand the varying resource demands associated with specific network traffic.

Three types of resources in active networks: Computation, Storage and Communication (Network)

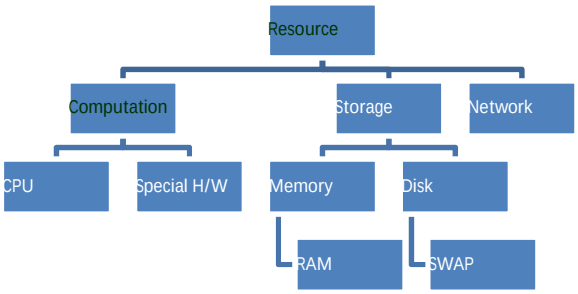

Figure 2: Resource management (QoS- parameters).

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. Devising a method for active packets to specify their CPU demands and fair resource allocation 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.

The paper is organized as follows: In section 2, the existing solutions to the problem are presented. Section 3 discusses the problem with the applied models. The implementation of a heterogeneous active network setup using Magician, a tool for active networks is discussed in section 4 and the evaluation and comparison of the results with various active applications is also presented. The resource estimation methods are presented in section 5. Section 6 proposes the Estimation Based Fair Allocation algorithm (EBFAA) and evaluates the performance of the algorithm. Section 7 draws conclusions on the effectiveness of our solution and suggests some possible future work.

2 A survey of existing 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 the existing solutions to prevent excessive CPU resource consumption in active networks and in mobile agent systems. Next we examine the research conducted outside of active networks that could help to provide effective resource management in active-network nodes.

2.1 Existing solutions to control the CPU usage

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.

Limit fixed by the packet

Some execution environments, such as ANTS [23], assign a timeto-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 [13], which is supposed to roughly correspond to the maximum diameter of the Internet. 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 [7] 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.

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.

Use a restricted language

The SNAP language [10] 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.

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 agents" 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.

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.

2.2 Existing attempts to quantify the CPU demand units

The survey of research related to quantify the CPU requirements initiates us to devise an effective solution. The following sections outline and discuss some of the ideas we found.

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 [4] 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?

Extra information provided by the programmer

In the AppLeS (application-level scheduling) project [3], 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.

Combined node-program characterization

Saavedra-Barrera and colleagues [14] 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.

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 [2] 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.

3 CPU control and demand prediction models

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 specify the functions offered to active applications by a specific execution environment. Black-box models specify the system calls offered to execution environments by a standard node operating system interface. While we are investigating both approaches, in this paper we focus mainly on a white-box model because, if successful, such models can be developed for each execution environment that a node intends to support. 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.

3.1 Proposed approach and significance

Now, 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 white-box models (without considering the system calls) against results obtained using CPU control and estimation techniques typically available in execution environments. 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.

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, scripts, servlets, and dynamically linked libraries. These similarities encourage us to believe that our model can be applied generally to the problem of specifying CPU demand in distributed applications that rely on the use of mobile code.

4 Implementation of white-box model

4.1 A heterogeneous active network setup

For the test setup, a three node heterogeneous active network is constructed: the machine *"AH-1"* is the sending node and *"AN-1"* is the destination. The following figure (Figure 3) represents this topology:

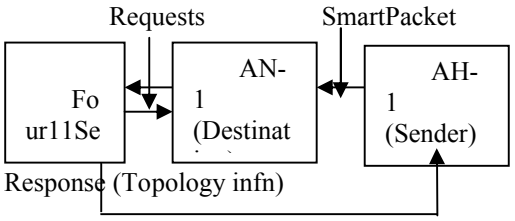

Figure 3: Test network setup.

The Smartpacket is transmitted from the Sender AH-1 to the Destination AN-1.

Network setup using MAGICIAN:
MAGICIAN was loaded in Linux environment. The tool provides an additional Execution Environment for setting up the Active Network and for sending the active packets using ANEP (Active Network Encapsulation Protocol). A Network Environment was created by giving specific host names to the machines forming the network.

One server (Four11) and two nodes have been setup:

Host name	**IP address**
magicserver	192.168.1.60 - Server
magicclient1	192.168.1.168 – (AN-1)
magicclient2	192.168.1.169 – (AH-1)

The tool was installed in all nodes. A network configuration (topology) file was created with the filename as similar to netname:magicserver

magicserver.conf consists of the following topology information:

```
(net:magicserver
      (node: AN-1
       host:magicserver
       IP:192.168.1.60
       gateway: (AN-2)
       ports: (3325 3322 3324)
      nbors; ((AN-2 192.168.1.168 3325 10000))

      node: AN-2
      host:magicclient1
      IP:192.168.1.168
      gateway: (AN-2)
      ports: (3323 3324)
      nbors: ((AN-1 192.168.1.60 3324 10000))
      ……..
```

A configuration file is created for Four11 server which is to be read by all the clients to know where the server is

running and to setup a connection from the client side to the server.

Four11.conf consists of:

host: magicserver port: 9411

Starting and testing the active network:

Starting the Four11 server:

The server has to be started from the directory where the tool has been stored

Java magician.Four11.Four11 magicserver

arguments:

magicserver - the netname-which in turn the name of the network config file-magicserver.conf

Starting the active node from the magicclient1 (host m/c):

The Node has also to be started from the directory where the tool has been stored in client machine.

Java magician.Node.NetworkNode magicclient1 magicserver log.txt

arguments:

magicclient1- host name where the node has to be setup – must be present in the n/w config file

magicserver - netname

log.txt - trace filename is the name of some file where we want the results to be stored

4.2 Evaluations and comparison

The white box model is implemented over *Magician*, a tool for implementing the Active networks. Magician is modified in order to incorporate CPU usage control. The CPU time needed for the execution of each packet in the first node is found out and stored. Here, the execution time of each packet in the first node is taken as the predicted value. SmartPing and SmartRoute are the applications which send the active packets. One malicious packet is intruded in between 5 good packets. Each malicious packet is programmed to consume as much CPU time as possible on each node. The EE monitors the execution of each active packet, interrupting them on a regular basis to query their execution time. If this execution time is below the predicted, then the packet continues its execution. Otherwise the EE kills it. Once the packet completes its execution, or when it's killed, the EE writes the information about the packet in the MIB (trace file): increments the number of packets killed or completed, and modifies the average CPU time used (computed over the last 20 packets) and all these information about the packet are stored. The average CPU time, the mean time and the variance CPU time are calculated and also stored in the trace file. The percentage of error, (i.e) the difference between the predicted time and the executed time is found out. The CPU time wasted for identifying and killing the malicious packet is also stored in MIB.

The characteristics of the heterogeneous platform selected for the control demo is presented in TABLE I.

Characteristics of three computer platform selected:

Table 1: CPU control and prediction demo-platform.

	Platform Description		
Node Name	Server	Node1 – AN-1	Node2 – AH-1
Processor Speed	3 GHz	2.4 GHz	1.2 GHz
Processor Architecture	Pentium IV	Pentium III	Celeron
O.S / Version	Red hat Linux / 7.0	Red hat Linux / 7.0	Red hat Linux / 7.0
Java Virtual Machine / Version	Jdk1.3.1_02	Jdk1.3.1_02	Jdk1.3.1_02
Memory size(Mega bytes)	512 MB	256 MB	128 MB

The results from the control demo and prediction are again analyzed and compared against two applications and two nodes. TABLE II gives the node-wise CPU utilization between two applications. The time taken for executing the two applications like Smartping and SmartRoute are given in the table II. The CPU time spent in Node1 and Node2 resembles the predicted value, which is shown in Table III. CPU predicted timings-report is presented in TABLE III. The predicted value is the time taken for executing the packet in the first node. If any packet with the active application executes beyond the predicted time in other nodes, the packet is identified as the malicious packet and it is killed. The average CPU time and variance in CPU time, calculated and stored in a log file by the Execution Environment is presented in TABLE III. The average CPU time and the variance in CPU time almost resembles the predicted time.

Comparison between applications

Table 2: CPU time usage-node wise report.

	CPU time Used (instruction cycles)		
Application	In Source	During Transition	
		Node1	Node2
SmartPing	13,500	13,700	13,740
SmartRoute	65,084	65,110	65,169

Table 3: CPU predicted timings-report (instruction cycles).

EE EE	AA	Predicted value	Avg CPU Time	Var CPU time
Magician	SmartPing	13,750	13,587	13,587
	SmartRoute	65,200	65,104	65,104

The percentage of error is presented in TABLE IV and the wasted CPU time for identifying and killing the malicious packets is also found. Percentage error is calculated as: Percentage Error = 100 * (prediction – actual) / actual. The actual CPU time is the one measured

in the first node. The percentage error between the predicted and actual CPU time is presented in TABLE IV. The variation is minimum between the predicted and actual timings.

Table 4: Error report.

EE	AA	Node	Percentage Error
Magician	SmartPing	Source	1.85
		Node1	0.36
		Node2	0.07
	SmartRoute	Source	0.18
		Node1	0.14
		Node2	0.05

Control demo – results: - 30 packets were sent and out of which 6 malicious packets were identified between 5 good packets and discarded.

The malicious packets were distinguished by evaluating their execution time, which goes beyond the estimated. It was found that the avg-wasted time for identifying and killing the malicious packets is 8.29 ms per packet. The total time taken per node is 49.74 ms. The CPU demand is calculated and reported for a heterogeneous setup.

5 Estimation of CPU time

The simplest estimation scheme is to measure the actual computation time offline as done in the above models, and include this value in all packets. The Estimation Based Fair Allocation algorithm can use this value for the estimation. This scheme has some drawbacks. The execution time of a program is dependent of the data and also dependent on the particular machine where it is executed. Different cache sizes, for example, can cause a program to take different amount of times, although the same sequence of instructions is executed. Additionally, a protocol is required to include the estimates in the packets, which is a considerable overhead. To avoid these problems, we have focused on estimation schemes [16, 18] that use local results to predict the next packet's execution time. We identified the estimation techniques for CPU estimation:

Constant

The constant estimate is the simplest estimator. The estimated computation time for queue i in round n, $estimate_{i,n}$, is always the same for all packets. If queues correspond to different traffic classes, this information can be used to select the constant.

$$estimate_{i,n} = estimate_{i,n-1} = const.$$

Exponential average

The exponential average is a common method for an adaptive estimation [15, 18] that combines the most current execution time, $actual_n$, with the previous results. It is defined as:

$$estimate_{i,n} = \alpha \,.\, actual_{i,n} + (1-\alpha) \,.\, estimate_{i,n-1}$$

The parameter α specifies how much of the previous history is preserved. This scheme is used in many of practical applications, e.g TCP round-trip delay estimation.

Packet size dependent estimate

While the exponential average works well in practice, it ignores the size of the packet [14] that is going to be processed. The packet size dependent estimate is defined as:

$$estimate_{i,n} = f_n(size(P_n));$$

where the function f_n maps the packet size $p_{n,}$ to a processing time. The function f_n is adopted by the estimator E as

$$f_n = E(f_{n-1}, actual_{i,n})$$

The estimator E maps a packet size dependent estimation function to a new estimation function under consideration of the actual processing time. Any function can be used for estimation but polynomial functions seem to be most suitable, especially since a polynomial of order 0 can be represented with 0+1 variables. Depending on the precision of the required estimation, higher or lower order polynomials can be used.

6 Estimation based fair allocation

6.1 Estimation based fair allocation algorithm

We propose an allocation algorithm based on Adaptive estimations and DRR for servicing flows (queues) in an active node. For each queue, a deficit counter and an estimate is maintained. The deficit represents the amount of processing that this queue can use. The estimate represents the amount of processing that is expected for the next packet of this queue. The scheduler forwards the packets of a queue to the processor as long as the deficit is larger than the estimate of the next packet. When a packet uses excessive processing, the packet is interrupted by the timer. When the processing is finished or terminated, the actual processing time is used to adjust the deficit, as well as the estimate that is used for the next packet. The architecture is shown in Figure 4.

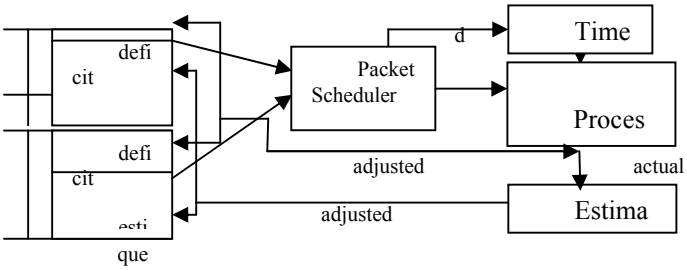

Figure 4: Estimation based fair allocation architecture.

Each network node stores packets coming from different flows in different queues. There are n queues. Each queue has initially no deficit and the estimated

processing time set to a default. The calculation of the estimation time is done using the exponential average method given under section V. The scheduling algorithm at the node selects a packet from the input queue, assigns it to CPU and runs the program associated with it until completion and then deposits it in the output queue. The algorithm (Figure 5) defines *round* to be a state in which the maximum number of packets allowable has been processed from all flows. The algorithm associates *Quantum* units with each flow *i* in each round. Each flow maintains a state variable d*eficit* that is initialized to *Quantum* before the start of each round of processing. The variable *cpu_estimate* maintains the number of CPU cycles required for a packet *p* in a flow during a round.

The main loop checks whether the deficit is positive, the deficit and the estimated processing time for the next packet is compared. If the cpu_estimate is less than the deficit, the packet is processed by p*rocess_packet_p*. If the current packet was previously interrupted, the old state is restored. A timer is also set to the deficit and started. The processing ends by means of two possibilities:

- If the packet used more time than it was permitted (the timer expires), the processing is preempted. The state of the processing is saved and the packet is pushed back into the head of the queue. The processing can then continue with that packet in the next round.
- If the processing is finished before the time expires, the packet is sent on and the next packet in the queue is considered.

The processing function returns the actual time that was used by the packet. The actual time is subtracted from the deficit. The estimator uses the actual time in adjust_estimate(), to adjust the estimation for the queue. If there is remaining deficit for the computation, then the next packet is considered for processing, otherwise the next queue is considered.

The total number of CPU cycles consumed by a flow in a round is maintained in a variable *total_cpu_con*. During each round, after a packet is processed from each flow, *cpu_estimate* is added to *total_cpu_con*. The number of packets processed from each flow in a round is within the restriction that d*eficit* > *total_cpu_con.* At the start of every new round, d*eficit* of the previous round is added to *quantum*. The ratio of q*uantum* given to any flows i, j is equal to the ratio of resource allocations for flows i, j. Also the algorithm only examines non-empty and backlogged flows.

6.2 Fairness

The algorithm is fair based on the following properties:

- The deficit counter is increased only once per round by the allotted quantum.
- If a queue does not make use of its entire share in a round, the amount is carried over to the next round in the deficit counter.
- The difference in total number of CPU cycles consumed between any two backlogged flows is bounded by a small constant.
- No queue receives more processor time than the deficit counter indicates. If the processing is interrupted by the timer, then the queue used its whole deficit and has to wait for the next round to receive more. If the processing terminates earlier, the deficit was not exceeded either.
- The deficit is charged only for the actual time that the processor was used.

```
For each flow i, get quantum
Assign Deficit  for flow i as Quantum of i + Deficitof i
Calculate cpu_estimate_i
While (deficit_i >0 && deficit > total_cpu_con_i )
    If (deficit_i >=cpu_estimate_i),
       Start_timer(deficit_i);
          P = head(queue);
          If ( is_interrupted_packet(p)) then
            Restore_state(i);
          end if;
            Start_timer(deficit);
            actual_time = process_packet_p;
           if (process_interrupted) then
              save_state(i);
              enqueue_at_head(p,queue);
          end if;
         deficit_i = deficit_i - actual_time;
         Cpu_estimate =
                adjust_estimate(estimate,actual_time);
     else
      break;
     end if;
end while;
 if (empty(queue)) then
   deficit_i = 0;
end if;
total_cpu_con[i]= cpu_estimate+total_cpu_con[i];
increment totalround with total_cpu_con[i]
increment the round by one
assign prevround as totalround
end For
```

Figure 5: Estiamtion based fair allocation algorithm.

6.3 Results of EBFAA

To find out the performance of the EBFAA, we implemented the algorithm. In Figure 6, we used a single active node and one host configuration with twenty flows sending packets. The only exception is that Flow 10 is a misbehaving flow. While the ill behaved flow grabs an arbitrary share of bandwidth when the EBFAA is not used. While in EBFAA, there is nearly perfect fairness.

The quantum / timeslice ranges from .1 times to 100 times the average processing time of a packet. The algorithm is implemented using three estimates ‘e’ (e = .1, e = 1 or e = 10 times the average actual processing time). Here ‘e’ is the scalar constant chosen as the

estimates. The estimation time is closer to actual processing time when e = 1. The algorithm incurs fewer context switches for quantum sizes in the range of the actual processing time.

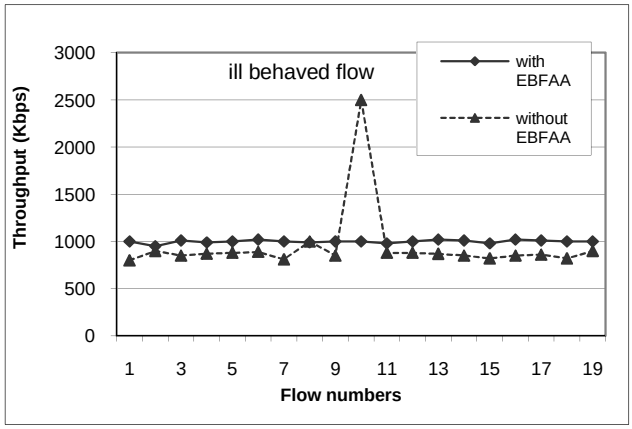

Figure 6: Control of the malicious flow.

We measured the delay rates for 20 flows. Each flow reserves the same processing rate, and sends packets randomly at specified average time intervals to just saturate its share. The sum of average processing rates of all flows is just under the processor capacity. Therefore, the delays measured are mainly due to scheduling not due to queuing backlog. The results in Figure 7 show that EBFAA provides lower maximum delays to all flows, when compared with WFQ, SFQ and SWFQ. EBFAA also gives smaller delay standard deviations than SFQ and SWFQ for all flows. Reduction in delay standard deviations would reduce the delay jitters. We expect that EBFAA would give better delay behavior due to its more accurate system virtual time, especially, where variations in processing requirements of packets are large. Figure 6 shows that WFQ provide smaller maximum delays than SWFQ, SFQ and EBFAA for application flows that have low processing time per packet to reserved rate ratio. However, EBFAA can provide lower maximum delays for all packets in flows when compared with WFQ. We propose to use EBFAA for processing resource scheduling in programmable networks to support QoS in two categories: processing resource reservation, and best-effort. We believe that EBFAA is also applicable for processor scheduling in operating systems in general.

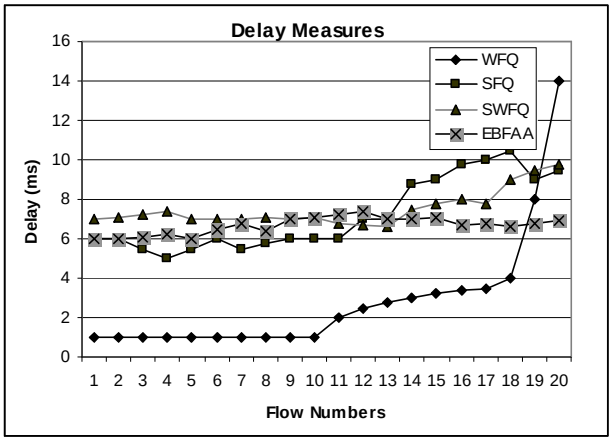

Figure 7: Delay measures

7 Conclusion and future work

This paper examines a way to analyze the CPU resource control and Fair Resource Allocation to improve the Quality of Service (QoS) in a heterogeneous active network environment**.** It is discussed that some means are needed to accurately specify the CPU demand in order to safely and efficiently deploy mobile code among heterogeneous platforms in a network. This paper has described an approach (White Box model) and an algorithm (EBFAA) to control the CPU utilization of malicious packets and to estimate the CPU demand for good packets in a heterogeneous active network environment and evaluated the approach**.** In the control application, it is demonstrated to identify the malicious packets, when malicious or erroneous code is injected in to a node and that the amount of CPU time stolen or wasted has been found out and can be reduced. The results from the control demo and prediction model are again analyzed and compared against various applications and nodes. The percentage error between the predicted and the actual CPU time is also less for this prediction model. The algorithm (EBFAA) also provides near-perfect fairness during resource control and allocation. Thus using this resource control model, the network management systems can allocate the capacity better by anticipating varying demands and the network operators can better estimate the quality of service (QoS) that customers can expect.

This work can be extended and can be compared with different Execution environments. 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 future plans to investigate these ideas in the context of resource management for mobile code loaded into call-processing servers. The issue of determining the CPU requirement for active packet can also be resolved by introducing a policy base [6, 18] at the active node. Combined scheduling algorithms which could schedule both CPU and bandwidth resources adaptively and fairly among all the competing flows can be applied. This work can also be extended for the prediction of the resources in wireless and sensor networks.

Acknowledgment

This work is supported by Technology Information Forecasting and Assessment Council (TIFAC) under the Department of Science and Technology (DST), Government of India and the Centre of Relevance and Excellence (CORE) in Network Engineering at Kalasalingam University, Krishnankoil-626190, Tamil Nadu, India.

References

[1] Amit. B. Kulkarni, “Magician–An Active NetworkingToolkit”, http://ittc.ukans.edu/projects/-Magician, 2000.

[2] Ball. T and Larus. J.R, "Using paths to measure, explain, and enhance program behavior", IEEE Computer, July 2000.
[3] Berman. F, Wolski. R., Figueira. S, Schopf. J, and Shao. G, "Application-level scheduling on distributed heterogeneous networks", in Supercomputing '96, September, 1996.
[4] Calvert K. L., Griffioen J., Mullins B., Sehgal A. and Wen S., "Concast: "Design and Implementation of an Active Network Service", IEEE Journal on Selected Area in Communications (JSAC), Volume 19, Issue 3, 2001.
[5] Calvert. K. L., "Architectural Framework for Active Networks", Version 1.0 Draft July 27, 1999.
[6] Christos Tsarouchis et.al. , "A Policy-Based Management Architecture for Active and Programmable Networks", IEEE Network, May/June 2003.
[7] Huber O.r J. and Toutain. L, "Mobile Agents in Active Networks", ECOOP'97, Workshop on Mobile Object Systems, June 1997.
[8] Rohan De Silva, ," A Security Architecture for Active Networks", Proceedings of the 4th ACM WSEAS International Conference on Applied Informatics and Communications, 2004.
[9] Larry Peterson and AN Node OS Working Group, "NodeOS Interface Specification", January 2000.
[10] Moore J.T., Hicks M., and Nettles S. "Practical programmable packets", *In IEEE InfoCom 2001*.
[11] Peterson. L, Gottlieb. Y, Schwab. S, Rho. S, Hibler. M, Tullmann. P, Lepreau. J, and Hartman. J, "An OS Interface for Active Routers", IEEE Journal on Selected Areas in Communications, 2001.
[12] Rauf Bautaba, University of Waterloo, Andreas Polyrakis, University of Toronto, "Projecting Advanced Enterprise Network and Service Management to Active Networks", IEEE Network, Jan/Feb 2002.
[13] Reynolds. J and Postel. J. "RFC 1700 Assigned Numbers", October 1994.
[14] Saavedra-Barrera R. H. Smith A. J., and Miya. E., "Machine characterization based on an abstract high-level language machine". *IEEE Transactions on Computers*, December 1989.
[15] Sohil Munir, "A survey of Active Network Research", IEEE Communications Magazine: vol. 35, no.1, pp 80-86, Jan 1997.
[16] VimalaDevi K. & Mehata K.M., "Resource Estimation and Policy Based Allocation for Qualityof Service in Active Networks", IEEE workshop on Coordinated Quality of Service in Distributed Systems (COQODS-II), held in conjunction with 14th IEEE ICON2006, in Singapore from September 13 to 15, 2006.
[17] VimalaDevi K. &. Mehata, K.M "Advancing Performance Management using Active Network Technology", NCCN'03, S.R.M.Engg. College, Chennai, India, Feb 2003.
[18] Cen, et al. "A distributed real-time MPEG video audio player", In Proceedings of Internal Workshop on Network and Operating System support for Digital and video (NOSSDAV), Lecture Notes in Computer Science, pages 151-162, Durham, New Hampshire, April 1995, Springer.
[19] Wetherall,. D, Guttag. J and Tennenhouse. D, "ANTS: Network Services without the Red Tape", *IEEE Computer*, pp. 42-48. (8), April 1999.
[20] Yamamoto. L and Leduc. G, "An agent-inspired active network resource-trading model applied to congestion control". *In MATA 2000*, pages 151–169, Sep 2000.