Fair Resource Allocation in Active Networks

Vijay Ramachandran Raju Pandey S-H. Gary Chan
Computer Science Department
University of California, Davis, CA 95616
vramacha@cisco.com, {pandey, chan}@cs.ucdavis.edu

Abstract— **Packet scheduling scheme is an important component of a network node. The choice of a scheme dictates the allocation of network resources among contending flows of the network. In this paper we study packet scheduling in the context of active networks. Traditionally, packet scheduling schemes are used to fairly allocate a single resource. This cannot be directly applied to active nodes beacause active nodes contain multiple resources, such as CPU and bandwidth. Moreover, these resources are inter-dependent. Hence, fairly allocating one resource does not entail allocating the the other resources fairly. We describe a simple packet scheduling algorithm to fairly allocate multiple resources in an active node. Using simulations, we show that the algorithm effectively allocates both CPU and bandwidth resources fairly among the contending flows.**

I. Introduction

Traditional network nodes have mainly been data forwarding engines. Such nodes have very little intelligence and processing power that has led to a networking infrastructure in which network protocols cannot be easily modified or customized to suit applications. This has led to the introduction of an active network architecture [8] in which intermediate nodes, called *active nodes*, can perform customized computations on every packet. Programmability in active networks is achieved by having active packets carry programs that are executed at an active node [11] or having the packets carry references to programs that are injected into the active nodes offline [12]. As each packet arrives at an active node, the node first executes the program associated with the packet. It then routes the packet to the next node. In this network, each active node serves two kinds of resources to network packets: CPU and network bandwidth.

The ability to perform packet-specific computations within the network leads to an extensible network infrastructure that can be extended, modified, upgraded, and customized to suite application requirements. Networks services that were deployed primarily at the end points can now be implemented in interior nodes dynamically, offering richer network functionality to the application. Flexibility in protocol design is especially useful in a distributed system where there is massive interactions among collaborating parties. Distributed applications (such as online stock quote, online auction, web caching, and fusion applications [3], [10], [9]) can exploit the services offered by the network to improve its performance as well.

While active networks offer flexibility in tailoring network services to applications, one of the pitfalls in using active networks is the potential degradation in overall application performance when multiple applications use the active node services simultaneously. Since active packets from multiple sources contend for both CPU and networking resources at an active node, the node is specially susceptible to ill-behaved packet sources that can generate packets at high rates and and seize an unfair share of the bandwidth. Further, packets from ill-behaved sources can completely dominate other active packets by consuming too much of CPU, thereby denying them both CPU and networking resources.

While there has been significant research in developing suitable active node architectures, programming models for active software and developing efficient security architectures, the problem of efficient resource allocation in active nodes has not been sufficiently addressed. Support for efficient resource allocation models in active nodes is a fundamental step toward large scale deployment of active networks.

In this paper we explore the various issues in developing a resource allocation model that allocates resources fairly to all active flows. In traditional networks, several packet scheduling algorithms [1], [7], [5] exist that aim to isolate different network flows from ill-behaved flows. The basis for the isolation is derived through fair allocation of network resources among the contending flows. Network resources in traditional networks mainly refer to but are not limited to bandwidth in the core network nodes. Service discipline such as Fair Queueing (FQ) [5] provide perfect fairness among contending network flows. In this paper, we show that the traditional notion of fairness, which specifies a resource allocation constraint for a single resource, does not directly extend to active nodes. This is because allocation of resources in active networks involves two resources, CPU and network. Further, the allocation of the two is interdependent. Thus, fair allocation of one does not guarantee fair allocation of the other. We present a notion of fairness for active networks in which we use the total resource consumption of a flow as a basis for measuring fairness.

We have developed an active packet scheduling algorithm to achieve fairness as specified in our definition. The scheduling algorithm satisfies the resource allocation constraint by adjusting the share of CPU resource given to each flow based on the share of bandwidth given to the flow. We note that the CPU requirement of an active packet is not known a priori. That is, the number of CPU cycles required to execute the code carried in an active packet cannot be determined by examining the fields of the packet. Hence, the scheduling algorithm we propose is not dependent on the CPU requirements of active packets. Furthermore, the algorithm has the additional property of 100% CPU and bandwidth utilization at all times. We have measured the effectiveness of the algorithm through simulation and found that it achieves almost perfect fairness for all flows.

The rest of the paper is organized as follows: In Section II we describe in detail the issues involved in allocating resources in active networks. In Section III we outline a reference active node architecture, and describe the different components of the architecture. We then discuss our algorithm which uses these components to achieve fair resource allocation. In Section IV, we present the simulations results. In the final section, we conclude and present directions for future research in this area.

II. Problem definition

In this section, we develop the notion of fairness for active nodes. We begin by first presenting the notion of fairness in traditional network nodes. We then show that this notion of fairness does not extend to active nodes.

A. Fairness in traditional networks

We begin by first formalizing the notion of a flow using the definition in [7].

Definition 1: A flow denotes a stream of packets that traverses the same route from a source to a destination and that requires the same level of service at each network node in the path.

Each packet can be uniquely assigned to a flow using pre-specified fields in the packet header. A flow is **backlogged** during the time interval (t_1,t_2) if the queue for the flow is never empty during the interval. We next formalize a measure of fairness, also taken from [7].

Definition 2: Let $sent_i(t_1,t_2)$ be the number of bytes sent by flow i in the interval (t_1,t_2). Let f_i express the ideal share to be received by flow i. Let $FairnessMeasure(t_1,t_2)$ be the maximum, over all pairs $(i,\ j)$ of backlogged flows in the interval (t_1,t_2), of $(sent_i(t_1,t_2)/f_i - sent_j(t_1,t_2)/f_j)$. Define *FairnessMeasure* to be the maximum value of *FairnessMeasure*(t_1,t_2) over all possible executions of the fair queueing scheme and all possible intervals (t_1,t_2) in an execution.

In other words, $FairnessMeasure$ expresses the amount of deviation from the ideal case. In the ideal case, when there are N flows backlogged in the interval (t_1,t_2), $FairnessMeasure$ will be equal to 0, and the share of allocated resource for each flow will be equal to $\frac{1}{N}$.

B. Fairness in active nodes

We now develop the notion of fairness for active nodes. Active packets first get processed by the CPU before being deposited in the output queue for transmission. The scheduling of active packets at the network end is, thus, dependent on CPU scheduling. This means that guaranteeing fairness during CPU scheduling does not entail fairness in the output queue, regardless of the scheduling algorithm applied at the network end. We highlight this with a simple example:

Assume that there are two flows: Flow 1 contains packets, each requiring 10 CPU cycles and 2 units network bandwidth. Flow 2 contains packets, each requiring 2 CPU cycles and 10 units network bandwidth. A fair allocation of CPU would mean that for every packet from flow 1, CPU processes five packets from flow 2. Such an allocation of CPU means that in any given time interval (t_1,t_2), there are five times as many packets in the output queue from flow 2 than there are from flow 1. Hence, if the network bandwidth is to be conserved and if all packets are processed from the output queue, the active node allocates 250% more bandwidth to flow 2 than flow 1 in any given time interval. Bandwidth allocation in this case is not fair, even though the CPU allocation is fair to both flows. The primary reason for this anomaly is that the traditional notion of fair scheduling does not take inter-dependencies between the resources into account.

We, thus, develop a fairness measure that accommodates inter-dependencies between resources. Qualitatively, in the example above, we can make the system more fair by doing the following: Since flow 2 requires less CPU than flow 1, flow 2 can overwhelm the active node if it receives the same share of CPU as flow 1. We can hence *punish* flow 2 at the processing end by giving a larger share of the CPU to flow 1, thereby increasing the number of packets arriving at the output queue from flow 1. The question then is: How do we quantify the notion of punishment? We would like to formalize a fairness measure that allows a flow with a higher requirement for a resource to receive a higher share of the resource, but maintain the balance in resource consumption by allocating a lower share of the second resource. This is the basic premise of our definition of fairness.

Composite Fairness Measure (CFM)

We define a fairness measure for active networks that uses the total resource consumption by a flow in the active node as the basis for a measure of fairness.

Definition 3: Let N be the number of active flows in time interval (t_1,t_2). Let $Fairness_i^{cpu}(t_1,t_2)$ be defined as the fraction of time CPU spends servicing flow i in time interval (t_1,t_2). Similarly, let $Fairness_i^{bw}(t_1,t_2)$ be defined as the fraction of time given to transmit flow i at the output in time interval (t_1,t_2).

We define the *Ideal Operating Situation (IOS)* as the best case scenario for an active node. In an IOS, for every $1 \leq i \leq N$,

$$\begin{aligned} Fairness_i^{cpu}(t_1,t_2) &= \frac{1}{N} \\ Fairness_i^{bw}(t_1,t_2) &= \frac{1}{N} \end{aligned}$$

Hence, in the ideal case, the total amount of resources used by a flow is $(\frac{1}{N}+\frac{1}{N}=\frac{2}{N})$.

Definition 4: An active node achieves perfect fairness if it satisfies the total resource usage measure, $\frac{2}{N}$, for all flows. We define $FairnessMeasure_i$ to be $((Fairness_i^{cpu}+Fairness_i^{bw})-\frac{2}{N})^2$.

$FairnessMeasure_i$ measures the amount the deviation from the IOS. Note that the fairness measure given here does not require that both $Fairness_i^{cpu}$ and $Fairness_i^{bw}$ be individually satisfied; it only requires that the total resource usage measure be achieved. This is because, we would like to allocate resources proportional to the requirement of resources. Consider the previous example, where there are two flows with extreme requirements. As mentioned earlier, the flow with a higher demand for CPU should receive a greater share of CPU resource, but we maintain the balance by allocating a lower share of the bandwidth. Similarly, the flow with a lower demand for CPU should receive a lower share of CPU resource, and a higher share of the bandwidth.

III. Achieving fairness using CFM

In this section we describe an algorithm for achieving fairness in active nodes.

A. Reference active node architecture

We begin by first proposing a reference architecture for active nodes. In this model, an active node is composed of three main parts: CPU scheduler, Output scheduler and a feedback element (see Figure 1).

We assume that all active packets that traverse the node require both CPU and bandwidth requirements. On reception of a packet, the node performs several steps [2]: It first retrieves the packet from the network card. It enqueues the packet in the processing

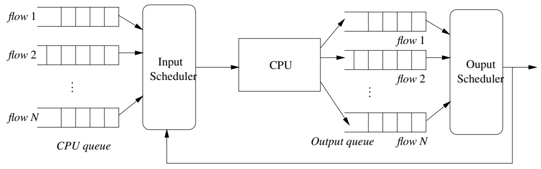

Fig. 1. Reference architecture for an active node

queue corresponding to its flow. The CPU scheduler then chooses a packet to process from the input queues. After processing the packet, it inserts the packet in the output queue corresponding to the flow. The output scheduler chooses a packet from the output queues for transmission. A novel aspect of the proposed architecture is the feedback element that the active node uses to periodically inform the CPU scheduler about output scheduling. The CPU scheduler uses the information to schedule packets for processing.

We note that the reference architecture contains multiple queues/flows scheme at both input and output ends. Another possible choice is to have a single queue at the input as well as the output. We did not chose this queuing scheme because it does not allow the node operating system to discriminate between different flows. This may invite an ill-behaved flow to trash the system by consuming more than its fair share of resource. Moreover, since the CPU requirement of an active packet cannot be determined from the fields in the packet, the exact arrival rate cannot be measured. Hence, we may not be able to use traffic shaping algorithms such as Leaky Bucket to guard against such an attack at the CPU end.

The three main components of the architecture are: CPU scheduling, packet scheduling for output, and feedback component from output to the CPU scheduler. They work in conjunction to meet our fairness criteria. We describe each in detail below.

B. Packet Scheduling

There are two scheduling algorithms used within an active node: one for CPU scheduling, and another for scheduling packet for transmission. The node operating system may execute these algorithms concurrently. Since the CPU requirements of active packets are not known a priori, the CPU scheduling algorithm should not only be fair, but also not depend on the knowledge of processing requirements of packets for scheduling. We have modified DRR [7] such that it can be used to schedule active packets fairly at the CPU end without any knowledge of the CPU requirements. We also use DRR for network scheduling. We chose DRR because it implements FQ efficiently.

CPU Scheduling

The algorithm is as follows: Each network node stores packets coming from different flows in different queues. The scheduling algorithm at the node selects a packet from the input queue, assigns it to CPU, runs the program associated with it until completion and then deposits it in the output queue. The CPU scheduling algorithm is shown in Figure 2.

```
while (true) {
  for each backlogged_flow(i) {
    Quantum(i) = get_quantum(i);
    DC(i) = Quantum(i) + DC(i)
    while (cpu_consumed(i) < DC(i)) {
      process_packet(p);
      CPUREQ_p = cpu_requirement(p);
      cpu_consumed(i) = CPUREQ_p +
         cpu_consumed(i);
    }
    DC(i) = DC(i) - cpu_consumed(i);
  }
  round = round + 1;
}
```

Fig. 2. CPU scheduling algorithm

As shown in the figure, the algorithm associates *Quantum* CPU cycles with with each flow i in each round. Each flow also maintains a state variable $DeficitCounter$, denoted by DC which is initialized to *Quantum* before the start of the each round of processing. Let the number of CPU cycles required for a packet p in a flow during a round be $CPUREQ_p$. The total number of CPU cycles consumed by a flow in a round is maintained in the variable $cpu\_consumed$. $CPUREQp$ is not known before processing the packet. During each round, after a packet is processed from a flow, $CPUREQp$ is added to $total\_cpu\_consumed$. The number of packets processed from each flow in a round is subject to the restriction that $DeficitCounter > total\_cpu\_consumed$. After all the packets are processed from a flow during a round, $DeficitCounter$ for that flow is reduced by $cpu\_consumed$. At the start of every new round, *DeficitCounter* is reinitialized to *DeficitCounter* of the previous round added to *Quantum*. Note that *DeficitCounter* can be negative during the start of a round. We note the following: The ratio of *Quantum* given to any two flows i, j is equal to the ratio of resource allocations for flows i, j. That is, resource is allocated proportional to the size of *Quantum* given to each flow. Also, the algorithm only examines non-empty and backlogged flows.

The algorithm is fair since the difference in total number of CPU cycles consumed between any two backlogged flows is bounded by a small constant. We omit the proof here due to lack of space. The proof can be found in [6].

C. Feedback function

An important component of the architecture is the feedback function that is used to control allocation of CPU resources so that overall fairness is maintained. The feedback function integrates the two resource allocation algorithm into a composite algorithm. We now describe how we apply the feedback.

In the active node, we cannot determine the arrival rate of a flow at the output queues because we do not know the CPU requirements of each packet. After packets are processed at the input, there are two possibilities at the output queue:

- All flows congested at the input queues are also congested at the output queues.
- Some flows which are congested at the input queues are not congested at the output queues.

In the first case, the CPU requirements for all flows were low enough such that the arrival rates at the output queues exceed the bandwidth of the outgoing link. In this case, the CPU scheduling algorithm in conjunction with the networking scheduling algorithm (DRR) allocate both CPU and Bandwidth fairly. In the second case, there are some flows at the input that have low CPU requirement such that the arrival rate at the output queue exceeds the link capacity and there are some flows with CPU requirements that are high enough such that the arrival rate at the output queue is less than the link capacity. Since DRR only services non-empty queues, bandwidth allocation depends on the arrival rate of packets at the output queues. Thus, bandwidth allocation per flow is fed back periodically to the CPU scheduler. The CPU scheduler adjusts *Quantum* Figure 2 for each flow based on the allocated bandwidth. This entails increasing or reducing the amount of CPU given to each flow. Below we describe how *Quantum* Figure 2 is adjusted to achieve fair resource allocation.

D. Algorithm

We would like a scheduling algorithm that allocates CPU and bandwidth resources proportionally, adaptively and fairly. The solution to meeting the resource allocation constraint of Definition 3 is to adjust the *Quantum* Figure 2 given to each flow at both the input and output queues in proportion to the requirement of the respective resources such that the fairness measure is met. This is the basis for our algorithm. The algorithm to assign the value of *Quantum* to each flow during the start of a new round is shown in Figure 3.

```
get_quantum(flow i)
{
  cpu_alloc = get_cpu_allocation(i)
  bw_alloc = get_bw_allocation(i)
  total_alloc = cpu_alloc + bw_alloc
  Quantum = 2/Nc - total_alloc
  return Quantum
}
```

Fig. 3. Algorithm to assign value of Quantum

In this paper, we look at the case where all the input queues are congested, and some or all of the output queues are congested. As mentioned earlier, the bandwidth allocation depends on the arrival rate at the output queues, and this depends on the amount of processing required by the CPU at the input. Let Nc be the number of queues that are congested at the input. At start, we use our CPU scheduling scheme with equal *Quantum* for each flow, for allocating the CPU resource. At start, CPU allocation for each flow will be fair. That is, the fraction of CPU allocated to each flow will be $\frac{1}{Nc}$. DRR is used for bandwidth allocation. After the first feedback period, the bandwidth allocation for each flow is fed back to the CPU scheduler. There are two possibilities:

- Bandwidth is fairly allocated among the flows. In this case, *bandwidth_alloc* is $\frac{1}{Nc}$ for all flows.

• Bandwidth allocation is not fair. That is, the bw_alloc for some flows will be greater than $\frac{1}{Nc}$, and for some flows will be less than $\frac{1}{Nc}$.

Note that the sum of the fraction of resource received by each flow is equal to 1 in both cases. In the former case, the value of *Quantum* for the flows will not change (i.e, $Quantum = 2/Nc - 1/Nc = 1/Nc$). In the latter case, let $bw\_alloc > \frac{1}{Nc}$. Let I be the absolute value of $(\frac{2}{Nc} - total\_alloc)$, where $total\_alloc$ is the sum of bw_alloc and cpu_alloc. I expresses the amount of deviation from our fairness requirement. For these flows, to meet the resource allocation requirement, the CPU allocation is decreased by I. That is, if the CPU allocation decreases by I, we guarantee that the total resource consumption for the flow is $\frac{2}{Nc}$ as required by our definition of fairness. Similarly, if $bw\_alloc < \frac{1}{Nc}$, the CPU allocation is increased by I. Note that since a variant of DRR is used for CPU scheduling and DRR is used for bandwidth scheduling, both CPU and bandwidth resources are conserved.

IV. Simulation results

We have analyzed the effectiveness of the algorithm through simulations. The primary goals of the experiments are to analyze the effectiveness of the algorithm in achieving overall fairness. We used a modified version of the *ns* network simulator [4] to simulate an active node-based network.

A. Default simulation settings

Unless otherwise noted, we present the default simulation setting for all the experiments. The network topology used in our study is as shown in Figure 1. We simulated 10 hosts each of which generated traffic corresponding to a flow. Hence, Host #1 generated flow 1 traffic, Host #2 generated flow 2 traffic and so on. Each host generated packets from the exponential on/off distribution with a burst time of 1000ms and an idle time of 1000ms. Each active packet contains two fields: code length and packet size, represented by the tuple $< CPU, BW >$. The number of cycles required to process a packet was randomly selected, with a uniform distribution, between Min_{cpu} and Max_{cpu}. Min_{cpu} and Max_{cpu} vary from 1 cycles to 35 cycles depending on the flow. Packet sizes were randomly selected, with a uniform distribution, between Min_{pkt} and Max_{pkt}. Min_{pkt} and Max_{pkt} vary from 400 bits to 4000 bits depending on the flow. To simulate extremities in the traffic pattern, flows 0 thru 3 had very high CPU requirement (in the range of 20 to 35 cycles), with varying packet sizes, while flows 4 thru 9 had lower CPU requirement (in the range of 1 to 5 cycles), with varying packet sizes. The output link capacity was set at 500Kbps, and the CPU processing power was set at 500 cycles per second. We simulated our topology for 200 seconds. We varied the packet generation for each flow experimentally such that the topology exhibited congestion at the node for the entire simulation time of 200 seconds.

B. Fairness Measures

In this section, we present the experiments related with the fairness measures.

B.1 Fairness measures comparison

First, we show that our algorithm achieves the *FairnessMeasure* as described in Definition 4. The parameters for this experiment are as outlined above. Figure 4 shows the result from the experiment.

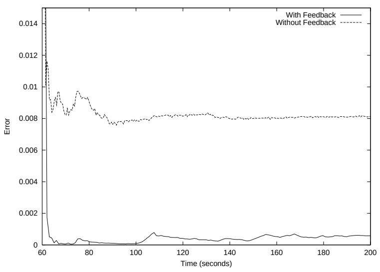

Fig. 4. FairnessMeasure with and without implementing algorithm with respect to time

It plots the *error* (i.e., deviation from Ideal Operating Situation) against *time* for a period of 200 seconds. We start the feedback at 60 seconds to let the traffic pattern reach steady state. We set the feedback period at 1.0 second. The graph shows that without feedback, i.e., without implementing the algorithm, the overall error is about 0.008. This means that on an average, since there are 10 flows, each flow deviates from the resource allocations constraint (0.2) by about 16%. When we use the fairness algorithm, the overall error reaches almost 0. Also, note from the graph that the time to reach an error of 0 is approximately 5.0 seconds. Hence the convergence is fast.

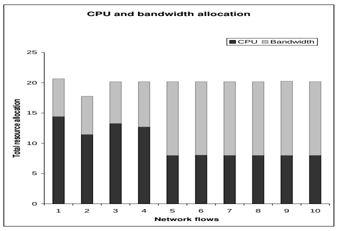

Fig. 5. CPU and Bandwidth allocation vs flow index

As mentioned earlier, our intention is to allocate CPU and bandwidth resources proportional to the requirement of Definition 4 for these resources. In the next experiment, we show that the algorithm indeed accomplishes this. As mentioned in Section IV-A, we chose our parameters such that flows 0 thru 3 have higher CPU requirement than the other flows, and have varying bandwidth requirement. In Figure 5, we show the percentages of CPU and bandwidth allocated for each flow. It can be seen from the graph that flows 0 thru 3 have higher percentage of CPU allocated. Flow 1 has the highest CPU requirement, and is allocated approximately 14% of CPU. Flows 1 thru 3 also have high CPU requirements and are allocated approximately 12 to 13 % of CPU. The rest of the flows have equal CPU requirement and are allocated approximately 8% of CPU. Bandwidth is allocated accordingly such that total resource usage constraint is satisfied.

B.2 Adaptivity

We next show that the algorithm can adapt to changes in network traffic patterns. In this experiment, there are extremities in traffic the pattern (i.e., in CPU and bandwidth requirements) as outlined in Section IV-A, for the first 100 seconds of the simulation. After 100 seconds, the traffic pattern changes such that the CPU and bandwidth requirements for all flows are statistically the same. The CPU requirement for all flows are randomly selected, with a uniform distribution, between 1 and 5 cycles. The packet sizes are randomly selected, with a uniform distribution, between 100 and 500 bits. Figure 6 shows the results of this experiment. The graph shows that at 100 seconds, the error increases to over 0.05. This is because the *Quantum* sizes prior to 100 seconds were adjusted for the traffic pattern before 100 seconds. The algorithm, however, adapts to the new traffic pattern quickly and the error term eventually reaches 0. The time to adapt is approximately 10 seconds. The time depends on the feedback period. Later we show convergence time for different feedback periods. We also discuss a good rule of thumb for choosing feedback periods.

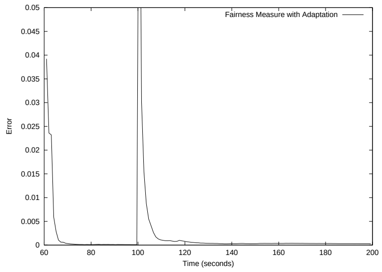

Fig. 6. FairnessMeasure versus time with new traffic pattern at t = 100s

B.3 Feedback

The feedback period forms an important parameter of the scheduling algorithm. In this experiment, we investigate the effect of the feedback period on the convergence of the algorithm. We conducted our experiment with the parameters as specified in Section IV-A, with feedback periods ranging the 0.5 seconds to 20 seconds.

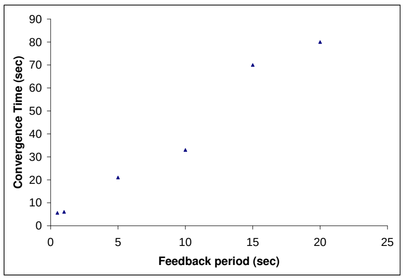

Fig. 7. Time to reach an error of 0.0004 for varying feedback periods

We illustrate the effect of feedback period using the the graph in Figure 7. Figure 7 plots the time to reach an *error* of 0.0004 for varying feedback periods. At one extreme feedback periods of 0.5 seconds and 1.0 seconds have the lowest convergence time (approximately 5 seconds). At the other extreme, for a feedback period of 20.0 seconds, the convergence time is 80 seconds. Since feedback is an expensive operation, a very low feedback period is not a good implementation choice. On the other hand, if the feedback period is very high, the convergence time is high and for bursty traffic patterns, timely convergence may not be achieved. We, therefore, need to find a middle ground for the feedback period. A good rule of thumb is the following: For any traffic pattern, we need to feed back the bandwidth allocation in order to change the *Quantum* size given to each flow. In order to have complete information about bandwidth allocation, all active flows (i.e., flows that have non-empty queues) must be serviced in a round. Similarly, for the CPU scheduler of have complete information about CPU allocation during every feedback period, all active flows must be serviced in a round. Hence,we use a feedback period equal to the time to finish one round of service at the CPU queues or Bandwidth queues whichever is bigger, i.e $FeedbackPeriod = Max\{(Max_{cpu}/CPU\_Speed) \times N, (Max_{pktsize}/LinkCapacity) \times N\}$.

C. Delay

We note that the underlying principles of DRR algorithm is unchanged. Hence, the delay bounds of DRR holds true for our algorithm as well.

V. Conclusion and Future Work

In this paper we have shown that the simple notion of Fairness used for traditional networks does not directly extend to active networks. We have formalized a definition of fairness for active networks and outlined an algorithm which achieves fairness as defined for active networks. The results from simulations confirm that the algorithm effectively allocates both CPU and bandwidth resources. The work presented here can be extended in a number of ways. A direct extension of this work is to incorporate multiple resources in the algorithm. Active packets may also require memory resources in addition to CPU and network resources. The algorithm presented here is the first step in this direction.

References

[1] Jon C.R. Bennet and Hui Zhang. WF^2Q: Worst-case fair weighted fair queueing. In *INFOCOM '96*, 1996.

[2] Dan E. Decasper, Bernhard Plattner, Guru M. Purulkar, Sumi Choi, John D. DeHart, and Tilman Wolf. A Scalable High-Performance Active Network Node. *Network*, pages 8–19, January 1999.

[3] Ulana Legedza, David Wetherall, and John Guttag. Improving Performance of Distributed Applications Using Active Networks. In *IEEE INFOCOM '98*, 1998.

[4] S. McCanne and S. Floyd. The LBNL network simulator. Lawrence Berekely Laboratory.

[5] A.K. Parekh and R.G. Gallagher. A generalized processor sharing approach in intergrated services networks. In *INFOCOM '93*, 1993.

[6] Vijay Ramachandran, Raju Pandey, and S-H. Gary Chan. Resource Allocation in Active Networks. Technical Report CSE-99-10, University of Califonia, Davis. Computer Science Department, 1999.

[7] M. Shreedhar and George Varghese. Efficient Fair Queueing using Deficit Round Robin. In *SIGCOMM '95*, aug 1995.

[8] David L. Tennenhouse, Jonathan M. Smith, W. David Sincoskie, David J. Wetherall, and Gary J. Minden. A Survey of Active Network Research. *IEEE Communications*, 35(1):80–86, January 1997.

[9] David L. Tennenhouse and David J. Wetherall. Towards an Active Network Architecture. *Computer Communication Review*, 26(2), April 1996.

[10] David Wetherall, Ulana Legedza, and John Guttag. Introducing New Internet Setvices: Why and How. *IEEE NETWORK Magazine Special Issue in Axtive and Programmable Networks*, July 1998.

[11] David J. Wetherall et al. ANTS: A Toolkit for Building and Dynamically Deploying Network Protocols. In *IEEE OPENARCH'98*, 1998.

[12] Yechiam Yemini and Sushil da Silva. Towards Programmable Networks. In *FIP/IEEE International Workshop on Distributed and Systems Operations and Management*, 1996.