An Efficient Resource Management Framework for Programmable and Active Networks.

Fariza Sabrina and Sanjay Jha
School of Computer Science and Engineering
The University of New South Wales, NSW 2052, Australia
Email: {farizas,Sjha}@cse.unsw.edu.au

UNSW-CSE-TR-0305
May 2003

THE UNIVERSITY OF NEW SOUTH WALES

SYDNEY · AUSTRALIA

Abstract

This report presents a framework for resource management in highly dynamic active and programmable networks. The goal is to allocate and manage node resources in an efficient way while ensuring effective utilization of network and supporting load balancing. The framework supports co-existence of active and non-active nodes and proposes a novel Directory Service (DS) architecture that can be used to discover the suitable active nodes in the Internet and for selecting best network path (end-to-end) and reserving the resources along the selected path. Intra-node and inter-node resource management are facilitated through the DS, while within an active node the framework implements a composite scheduling algorithm to schedule CPU and bandwidth resources to resolve the combined resource scheduling problems. In addition, a flexible active node database system has been introduced in order to resolve the challenging problem of determining the CPU requirement of the incoming packets. Through simulation we show the improved performance of our scheduling algorithm in achieving overall fairness in allocating active node resources.

1. INTRODUCTION

Active network is a new framework where network nodes not only forward packets, but also perform customized computation on the packet flowing through them. The primary goal of active networks is to make the network a more general computation engine. It provides a programmable interface to the user where users dynamically inject services into the intermediate nodes. Intermediate nodes that provide these services are called active nodes and the network packets that invoke the services are called active packets. The behaviors of active packets are controlled by the users through the programmable interface or via the packets they send. As the packets arrive at an active node, the node first executes the program associated with the packet. It then routes the packet to the next node. Active network technology envisions deployment of virtual execution environment within network elements, such as switches and routers. To use such technology safely and efficiently, individual nodes must provide mechanisms to enforce resource limits. This implies that each node must understand the varying resource requirements for specific network traffic.

In an Active network one of the fundamental problem is the development of efficient resource distribution model [3]. While there has been significant research in developing suitable active node architectures [11, 12, 13,15], programming models for active software, and developing efficient security architectures [2,16], the problem of efficient resource allocation in active nodes has not been sufficiently addressed.

In traditional network, several packet scheduling algorithms exist that aim to isolate different network flows from ill-behaved flows. The basis for isolation is derived through fair allocation of network resources among the contending flows. Network resources in traditional networks mainly refer to bandwidth in the core network nodes. Service discipline such as Fair Queuing (FQ) provides perfect fairness among contending network flows. However, traditional notion of fair queuing that specifies a resource allocation constraint for a single resource does not directly extend to active nodes since allocation of resources in active networks involves more than one resource such as, CPU, bandwidth, and memory. Moreover, the allocation of the three is interdependent. Thus fair allocation of one does not guarantee fair allocation of other. Some work has been done [5,6,7,8] on scheduling network resources but those allocate resource for a single resource. This does not directly extend to active networks. It has been identified that CPU requirement of active packets on multiple platforms cannot be determined a priori and it is a major obstacle in managing CPU resource in active nodes [3, 4, 20].

The limited Resource management model in ANTS [12, 13, 17] network does not ensure that resource allocation between competing packet flows would be fair. Another recent active networking research PANTS [15] provides only bandwidth sharing policies over links. Ramachandran et al [3] have proposed a fair scheduling scheme with a separate CPU scheduler (based on DRR) and a bandwidth scheduler where bandwidth scheduler periodically provides feedback to the CPU scheduler about the bandwidth allocations in order to adjust the CPU allocations. In this scheme the accuracy of maintaining fairness depends on the frequency of the feedback. Also their work lacks the notion of inter-node communications and an active node's awareness of the other active nodes within the network and it does not support resource reservations.

It is apparent that for large-scale deployment of active networks, researchers not only need to address the issues of managing multiple resources within a node but also need to provide some efficient mechanisms for intra-node and inter-node communications in order to exchange information regarding node capabilities and resource usage. Moreover, as more and more active nodes become available, the users would require an efficient way to discover an active node in the Internet is capable of providing the desired services.

In order to resolve the resource management issues of active networks this report presents an Active Networking framework for hybrid network, where a hierarchical Directory Service (DS) architecture is used to publish the existence of the active nodes and their capabilities. The DS provides the services such as discovering an active node, determining the best end-to-end path making sure that the selected active node falls within the path, and reserving resources along the path. Active Nodes within the network can

communicate with each other through the DS. This framework also provides a detailed architecture for an individual Active Node and presents an efficient resource management model. A new "Composite Bandwidth and CPU Scheduler" has been developed that can schedule both CPU and bandwidth while maintaining fairness for all the competing flows. The challenging problem of determining the CPU requirement of an active packet is resolved by introducing an Active Node Database (ANDB) within the node. Once an active code/program is referenced for the first time by an active packet in the node, the CPU requirement for the active packet is evaluated and then added in the ANDB. Therefore, the content of the ANDB is built "on-the-fly" and used for allocating resources for the subsequent incoming packets.

2. ACTIVE NODE RESOURCE MANAGEMENT ARCHITECTURE

We propose an active networking framework that can be suited to both programmable and active networks and it is suitable for both large intranet and Internet. The detailed Architecture of intra-node, inter-node, and an individual resource management are discussed in this section.

2.1 Resource Management In Intranet Environment

Here, we present a scenario where the client and the destination are within same intranet. It is assumed that a Directory Service (DS) is a well-known server (Figure 1), which is known to all of the nodes within an intra-net. An Active Node sends a request to the DS to register it as an Active Service Provider (i.e., Active Node) specifying its resource capabilities. An Active Node may un-register itself at any time by sending a message to the DS. Directory service sends a multicast message to all the active nodes in order to collect the resource utilization information whenever it is necessary.

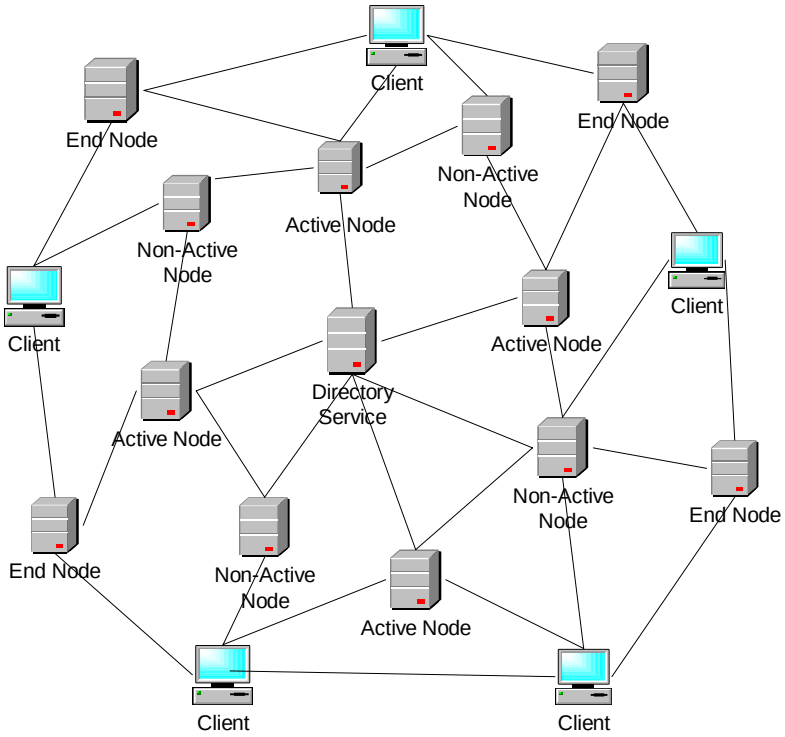


Figure1 : Intranode Communication using a Directory Service

Whenever a client wishes to send a flow of active packets to a destination, and wishes to process the packets on the way in an active node, the client sends a query along with a packet carrying code (or

reference to a code) and data (for evaluation) to the directory service specifying its resource requirements, and destination address (including reservation requirements) in order to discover an Active Node that is suitable and available to provide the service at that time. Also the client expects that the DS would provide an end-to-end path and would reserve resources on behalf of the client. On receiving a request, the DS analyzes its Active database in order to find out a number of active nodes currently registered under the DS and that can provide the resources.

After finding the a set of active nodes, DS multicasts a message to all the Active Nodes requesting the Current Snapshot of the Resource Utilization of the nodes and also sends the packet to all the nodes for evaluation. On receipt of such a request, the active node evaluates the packet and sends a reply to the DS. DS then analyzes the messages and determines the most suitable node (i.e., the node that is less busy). Now if the destination address is within this intranet, DS finds the best path (from client to destination) that includes the selected active node on the path and does the resource reservation along the path and then sends the path information to the client. If multiple active nodes show the same capabilities then DS chooses the active node based on the shortest path. If none of the active nodes could provide the requested service then the DS will notify the client so. At this point the client decides whether it should send the traffic to the end node hoping that an active node on the way may provide the service, if not then it will go to the end node.

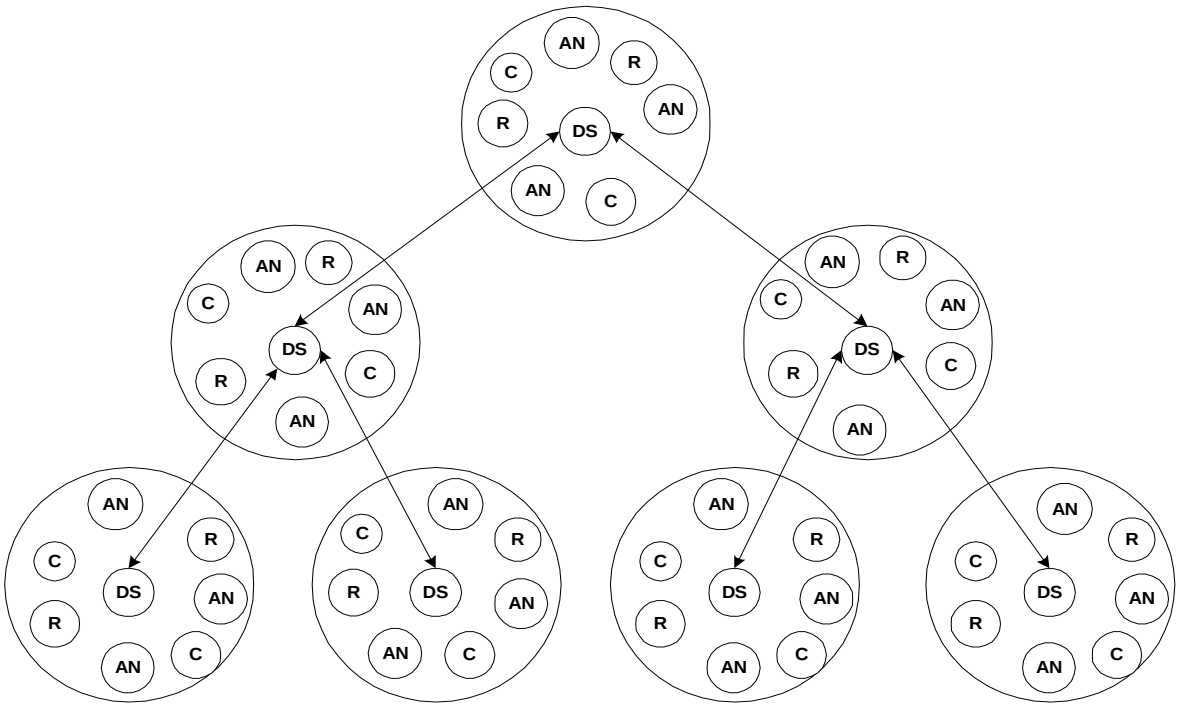


Figure 2: Inter-node Communications - Hierarchical Structure of the Directory Services

2.2 Resource Management in Internet Environment

Figure 2 shows the inter-node resource management using a hierarchical directory service structure. In inter-node scenario a client first connects to the DS within its own intranet specifying the destination address and resource requirements in order to find a suitable active node on the way. The DS identifies that the destination address is outside its own intranet and then the DS sends a request specifying the destination address and the requirements to the DSs that are one level above or below its current level. The Directory Services communicate with each other in order to resolve the issue of node and path selection, and resource reservation. After the path selection and reservation is completed the path information is sent back to the client.

2.3 Active Node Resource management

The detail of the active node resource management architecture is shown in Figure 3. The architecture can be applied for both intra-node and inter-node scenarios. The architecture supports code-carrying active packets and active packets that only carry references to certain codes or programs (assuming they were pre-installed on the node) and also non-active packets. The main objects in the Active Node are:

Active Node Resource manager (ANRM): This is the main object in the active node. This object receives packets, communicates with directory service and clients, builds and maintains the Active Database, manages the Composite Bandwidth and CPU Scheduler, Scheduler Queues, and the Active Code Evaluator Object, and maintains the Code Cache and Active Packet cache. In short, the ANRM oversees and manages the entire activity within an active node.

Active Node Data Base (ANDB): It contains the information of CPU Requirements for all the evaluated Active Code. It also contains some other information such as reservation information, information about any ill behaved code etc.

AN Composite Bandwidth and CPU Scheduler: The composite bandwidth and CPU scheduler fairly allocates both CPU and bandwidth for all the competing flows as described in section 3.

Scheduler Queues: These are the underlying FIFO queues used by the scheduler. Packets from the each individual flow are en-queued in different queues.

Active Code Evaluator: This object is responsible for evaluating a newly arrived active code (i.e., the ANDB does not have any record for this code).

CPU Handler: This object is responsible for processing an active packet.

Code Cache: When a code comes to the node after evaluation this code is saved in code cache.

Active Packet Cache: This is a temporary buffer for storing and forwarding of active packets under special circumstances, such as the node has become extremely busy and cannot cope with the volume of traffic. There is another buffer in this cache for data packets that come without the code and the referenced code is not found in the code cache.

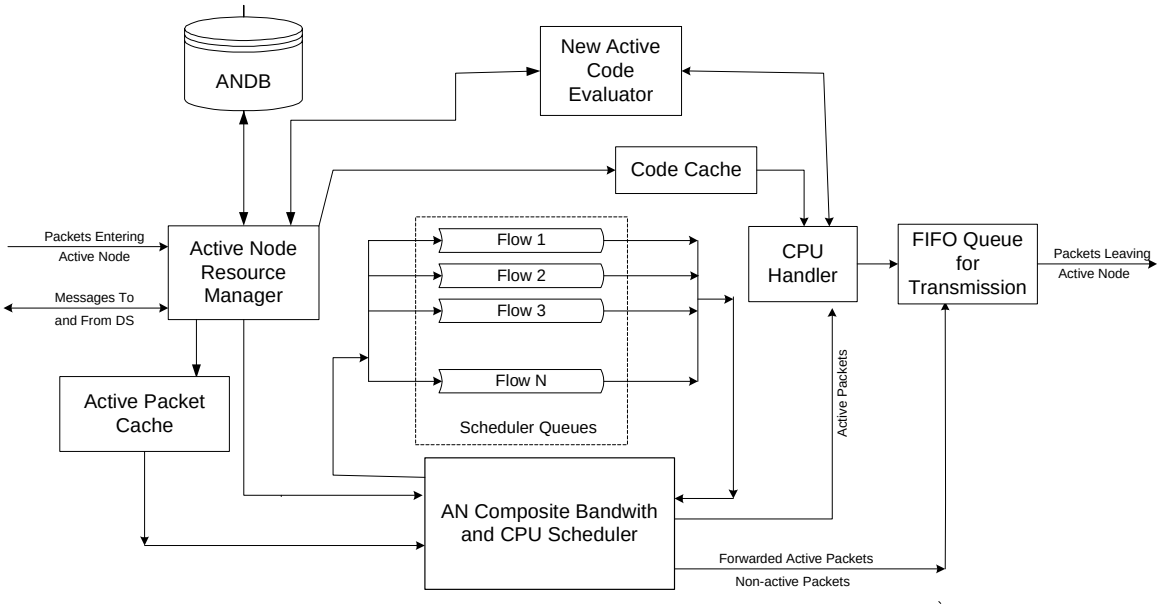


Figure 3: Proposed Architecture for an Active Node

An active packet contains active code that is identified by a GUID (as in the PANTS [13] and ANTS [10] packet formats). The active packet header contains fields that are used to specify the referenced code, monitor the processing status, and identify the type of service required, and so on.. We consider that in most cases an active packet flow would reference a limited number of different programs or code that the source may have already evaluated in the node prior to the flow's commencing. It may be noted that considering a limited number of referenced codes within a flow maps well to most active and programmable networking applications (e.g., video encoding, fusion, online brokering, data filtering) that require executing the same specific code on many packets within a flow. However, our system also deals with the situation where a packet comes with reference to an unknown code.

Under normal operations scenario (i.e., processing and forwarding) when a packet enters an active node, the Active Node Resource Manager (ANRM) reads the packet header to determine the service requirement of the packet. For an active packet that must be evaluated, ANRM uses the Active Code Evaluator (ACE) to evaluate the CPU requirements of the packet against the referenced code. If the packet does not carry the referenced code and the code is not available in the code cache, then ANRM sends a request to the packet stream's source node to send the referenced code and puts the packet in a waiting queue in the active packet cache.

It may be noted that processing of packets on an active or programmable node can also affect the size of packets after processing is completed. To take this packet size change into account for bandwidth consumption, we define and calculate an Expansion Factor that is the packet size after processing divided by the packet size before processing. After evaluation of the packet is done, ANRM updates the ANDB with the CPU requirement (against evaluated packet size) and the Expansion Factor for the referenced code.

For an active packet that must be processed in the node, the ANRM searches the ANDB for a record corresponding to the referenced code. If a record is found and was not recorded in the ANDB as ill-behaved code, then it determines the CPU requirement based on the recorded information in the ANDB and passes the packet to the scheduler object, specifying the CPU requirement and the Expansion Factor. The CPU requirement of the packet may differ from the CPU requirement of the evaluated packet as recorded in the ANDB. Pappu et al. [7] have shown experimentally that for header processing

applications (e.g., IP forwarding) the processing time does not depend on the packet size, while for payload processing applications (e.g., compression) the processing time varies linearly with the packet size. ANRM takes the current packet size and the evaluated packet size as recorded in ANDB into account for determining the CPU requirement if the referenced code were classified as a payload processing application in the ANDB record.

If a record was not found in ANDB for the referenced code, the ANRM checks whether the referenced code is available (either carried by the packet or in the code cache). If it is available, ANRM uses the ACE object to evaluate the packet, updates the ANDB and then passes the packet to the scheduler object, specifying the CPU requirement and Expansion Factor. If the referenced code is not available then the ANRM puts the packet in a waiting queue in the active packet cache and sends a request to the source node to send the referenced code. Note that this request may not traverse up to the source node; it may be intercepted by an active node on the way and the code found in that node. When the requested referenced code arrives in the node, ANRM evaluates the CPU requirement and updates the ANDB, and at this point all packets from the waiting queue corresponding to the referenced code are handed to the scheduler object. The non-active packets and the active packets that do not require processing in this node are handed to the scheduler object specifying zero CPU processing time and 1.0 as the Expansion Factor.

If ANRM receives a packet from the DS for evaluation along with the query for resource capability and current utilization, ANRM hands the packets to ACE, updates the ANDB after the evaluation is done, updates the code cache, and it then responds to the DS. ANRM also reserves resources upon request from the DS to do so.

3. COMPOSITE BANDWIDTH AND CPU SCHEDULER

In an active node, active packets first get processed by the CPU before they are transmitted through the network link. It has been identified that maintaining fairness during CPU scheduling does not entail fairness in the bandwidth allocation [3]. In this project we have developed 2 new Composite Bandwidth and CPU scheduling algorithms that can allocate both the CPU and bandwidth resources proportionally, adaptively and fairly among all the competing flows. The first algorithm (named $CBCS^{DRR}$) is based on DRR (Deficit Round Robin) principle and the second one (named $CBCS^{WFQ}$) is based on WFQ (Weighted Fair Queuing) principle. The $CBCS^{WFQ}$ has the ability to allocate both CPU and Bandwidth resources based on reservation. ANRM is designed to use either one based on the set-up parameter of the Active Node. In this report we describe only $CBCS^{DRR}$. Detail design and performance analysis of $CBCS^{WFQ}$ algorithm have presented in the technical report UNSW-CSE-TR-0306.

The scheduler object en-queues packets in the corresponding queues and de-queues packets using its fair-scheduling algorithm $CBCS^{DRR}$, which guarantees fair shares of both CPU and bandwidth between all the competing flows. After de-queuing a packet, the scheduler hands the packet to the CPU handler to run the code in the CPU i.e., to process the packet. The packet is sent to its next destination after processing. The scheduler also continuously monitors the individual queue lengths in terms of the total CPU requirement for all packets in each queue. If the individual queue length becomes greater than a pre-defined threshold (which is configurable for each flow) then the scheduler stops accepting packets from the corresponding flow. ANRM then puts these packets into the Active Packet Cache (which includes a separate FIFO queue for each flow) and they will be handed to the scheduler when the queue length gets smaller than another predefined threshold. The details of the algorithm are discussed below.

3.1. Definition of fairness

The total resource consumption by a flow in an active node is used as the basis for measuring fairness. We introduce the following notations.

Let,

$CPU_i^t(t_2 - t_1)$ = the CPU time consumed by packets in flow *i* within the time period $(t_2 - t_1)$.
$BWi^t(t_2 - t_1)$ = the transmission time consumed by packets in flow *i* within the time period $(t_2 - t_1)$.
$CPU_{tot}^t(t_2 - t_1)$ = the total CPU time consumed by packets in all flows within the time period $(t_2 - t_1)$.
$BW_{tot}^t(t_2 - t_1)$= total transmission time consumed by packets in all flows within the time period $(t_2 - t_1)$.
N = number of competing flows within the time period $(t_2 - t_1)$.

Definition 1: A flow is *backlogged* during the time interval $(t_2 - t_1)$ if the queue for the flow is never empty during the interval. Our fairness measure specifies that for any time period of $(t_2 - t_1)$ the total resource allocation (i.e., CPU plus bandwidth) for each individual backlogged flow is a constant:

$$\{CPU_1^t(t_2 - t_1) + BW_1^t(t_2 - t_1)\} = \{CPU_2^t(t_2 - t_1) + BW_2^t(t_2 - t_1)\} = \{CPUi^t(t_2 - t_1) + BWi^t(t_2 - t_1)\}$$
$$\ldots\ldots\ldots\ldots = \{CPU_N^t(t_2 - t_1) + BW_N^t(t_2 - t_1)\} = C, \text{ where } 0 < C <= 2(t_2 - t_1)/N$$

Definition 2: An active node achieves *perfect fairness* if total resource allocation for a time period of $(t_2 - t_1)$ for all the backlogged flows is the same, ideally (1/N + 1/N = 2/N). We define the deviation from the ideal allocation for any flow *i* as:

$$Dev_i^{(t_2 - t_1)} = \left\{ \frac{2}{N} - \left(\frac{CPU_i^t(t_2 - t_1)}{CPU_{tot}^t(t_2 - t_1)} + \frac{BW_i^t(t_2 - t_1)}{BW_{tot}^t(t_2 - t_1)} \right) \right\}$$

Definition 3: We measure the error in maintaining fairness in resource allocation for any backlogged flow *i* for the time period $(t_2 - t_1)$ as:

$$Error_i^{(t2 - t1)} = \{Dev_i^{(t2 - t1)}\}^2$$

We measure the average error for time period as:

$$Error_{avrg}^{(t_2 - t_1)} = \sum_{i=1}^{N} \frac{Error_i^{(t_2 - t_1)}}{N}$$

3.2 CBCS^{DRR} – A Composite Bandwidth and CPU Scheduling algorithm based on the DRR principle

CBCS^{DRR} enforces fairness using the Deficit Round Robin (DRR) principle. The algorithm uses the following parameters and equations:

Quantum = A variable that represent a time slice that is used to serve packets from each flow queue (which includes both CPU processing time and network transmission time) (sec)
DeficitCounter [i] = A state variable that represents a time slice for which the flow *i* queue can be serviced within a specific round of scheduling (sec).
BW = Bandwidth of the transmission link in Mbps.
P^{Size} = Size of a specific packet within a competing flow in Bytes.
P^{EF} = Expansion factor corresponding to the code referenced by a packet.
P^{CPUReq} = CPU time requirement to process a specific packet (sec).
$CPU_{t[i]}^{UL}$ = Upper limit of the total CPU Queue in terms of CPU processing time requirement for all packets in flow *i*.

$CPU_{t[i]}^{LL}$ = Lower limit of the total CPU Queue in terms of CPU processing time requirement for all packets in flow *i*.
P^{ts} = Time slice required for processing and transmitting a packet (sec).
$P^{ts}_{(Max)}$ = The maximum allowable time slice that a packet can have to cover both CPU processing and network transmission. It is configurable, and should depend on the capability of an active node.

Therefore, $P^{ts} = (8 * P^{Size} * P^{EF}) / (BW * 1000000) + P^{CPUReq}$

When the scheduler receives a packet, it looks at the header of the packet to determine the flow-id for the packet, notes the CPU requirement and Expansion Factor, and then stores the packet in the corresponding flow queue. However, the scheduler continues to monitor the queue length for all individual flows in terms of CPU time requirement and stops accepting packets from a flow if its queue length becomes greater than $CPU_{t[i]}^{UL}$. In this case the scheduler continues to refuse new packets from flow *i* until its queue length becomes smaller than $CPU_{t[i]}^{LL}$.

When the scheduler starts, the *Quantum* is set to $P^{ts}_{(Max)}$, and the *DeficitCounter* for all flows is set to zero. The scheduler continues to serve all non-empty queues within each round of processing. When it starts to serve a queue within a round, the *DeficitCounter* is set to *Quantum* plus the *Deficit Counter* of the previous round. The scheduler then dequeues a packet from the head of the queue and calculates the P^{ts} of the packet. It sets the Deficit Counter to (*DeficitCounter* – P^{ts}) and hands the packet to the CPU Handler object for execution. The packet is sent to its next destination after processing. The scheduler stops serving a queue once the queue is empty or the deficit counter becomes zero or negative. It may be noted that the *DeficitCounter* for a non-active flow (i.e., a flow having no packets in the queue) is reset to zero.

Also, processing and transmissions of different packets happen in parallel, i.e., after processing the packets enter in a FIFO queue, which is served by a separate thread for transmitting the packets.

The pseudo code of $CBCS^{DRR}$ is given below.

```
Quantum = P^ts_(Max);
For each flow(i) { DeficitCounter[i] = 0;}
while(true)
{
  for each flow(i)
  {
    if (the queue is not empty)
    {
        DeficitCounter[i] = DeficitCounter[i] + Quantum;
        while(DeficitCounter[i] > 0 and the Queue is not empty)
        {
            De-queue a packet;
            Read the packet header and calculate the P^ts ;
            DeficitCounter[i] = DeficitCounter[i] - P^ts ;
            Hand the packet to CPU Handler;
        }
        if (the queue is empty)
      {
            //Don't Accumulate DeficitCounter for
          //a non Active flow
            DeficitCounter[i] = 0;
        }
    }
```

```
    else
    {
        //Don't Accumulate DeficitCounter for a non Active flow
        DeficitCounter[i] = 0;
     }
  }
}
```

3.3 How parallelism is implemented in Composite Scheduler

Our composite scheduler is a two-staged scheduler. This is how parallelism is ensured – the scheduler de-queues a packet from an individual queue, calculates the P^{ts}, updates the deficit counter by deducting the P^{ts} and then gives the packet to the CPU Handler object. CPU Handler process the packet and then the packet is en-queued in a FIFO QUEUE for Bandwidth transmission which is served by a separate thread. i.e. while a packet is being processed in the CPU, it does not block the transmission of other packets from the transmission queue. Lets take an example. Say, scheduler is serving two packets from input queues. First packet has P^{ts} = 10 ms (CPU = 2 ms and Bandwidth = 8 ms), and the second packet also has P^{ts} = 10 ms (CPU = 8 ms and Bandwidth = 2 ms). The scheduler picks the first packet, and hands it to CPU handler. CPU handler process the packet for 2 ms and then it puts the packet in the transmission queue. After putting the packet in the transmission queue, scheduler picks the second packet and hands it to CPU handler for Processing. So after 2 ms, the scenario is: The second packet is being processed in the CPU, and the first packet is being transmitted through the link.

Since we are already considering the bandwidth transmission time and CPU time within P^{ts} for de-queuing the packets from the individual input queues, we are using a single FIFO queue for Bandwidth transmission for all the packets. I.e., after processing packets in CPU, they enter in this FIFO queue. Parallelism is also clearly visible in our simulation results presented in the later section.

4. SIMULATION RESULTS

We analysed the effectiveness of the $CBCS^{DRR}$ algorithm in achieving overall fairness through simulations. We have modified the NS2 network simulator [14] to implement the described active node components to simulate an active node-based network. The simulation was performed on a PC having a 1.5 GHZ Pentium 4 processor and 512 MB memory and running under the Linux operating system (RedHat 7.2). The experimental results achieved using $CBCS^{DRR}$ are compared with the results using DRR for scheduling CPU and bandwidth independently.

Note that the processing overhead and memory requirement for the proposed system (e.g., to classify and manage individual flows) are minimal. The state information of each individual active flow is kept in the memory (in arrays of integer and double variables).

4.1 Default Simulation Settings

We used 10 hosts to generate network for an active node, where each host generated CBR traffic corresponding to a flow. Host #1 generated active packets containing MPEG2 video data and referencing MPEG2 encoder code. The other hosts generated active packets with different

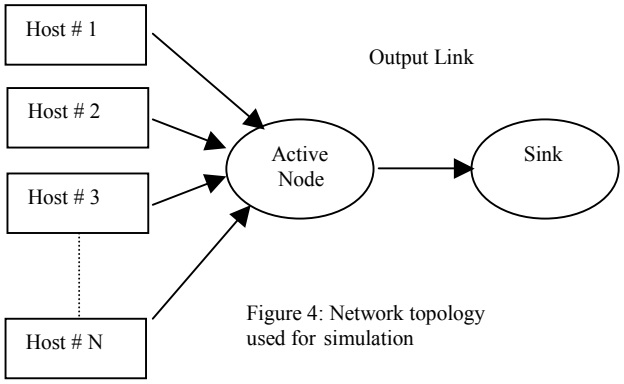


Figure 4: Network topology used for simulation

packet sizes and referencing different code GUIDs. MPEG2 code was implemented in C++ within the NS2 environment and the evaluator object used the code to evaluate the CPU requirement and expansion factor of MPEG2 data packet the first time one arrived at the active node. For the other hosts, the packets generated from a given host carried the same GUID, and the simulated evaluator object randomly evaluated the CPU requirements for the code (when the first packet arrived in the active node) between 1 and 10 milliseconds and the expansion factor was set to 1. The output link capacity was set to 5 Mbps. The simulation settings of the individual flows are given in Table 1. Packet generation rates for each flow is varied based on the CPU requirements so that the topology exhibited congestion or backlog for each flow at the active node for the entire simulation time of 50 seconds.

Each simulated active packet header contained some additional fields (or parameters) to signify the packet as an active packet and to facilitate the handling of the packet by the active node. In reality, the communication between active nodes could be done by exchanging messages, similar to the PANTS [13] architecture. However, for our simulation in NS2 simulator we added some parameters in the common header (i.e., hdr_cmn structure) of the NS2 packet. The active packet parameters are explained below:

Anbit_ (Boolean): Signifies a packet as an active packet, set by the Hosts.
Code_GUID (String, 16 bytes): Code Identifier, set by the source Hosts
ANPdone_ (Boolean): CPU Handler object sets its value to true after simulating the execution of the packet (i.e. holding the packet for the specified time)

The simulations were run for 50 seconds and measurements were taken at 5-second intervals. Packet generation rates for all the flows were adjusted such that the topology exhibited congestion or backlog for each flow at the active node for the entire simulation time of 50 seconds.

Flow#	1	2	3	4	5	6	7	8	9	10
Packet size (KB)	24	3	3	3	3	3	3	3	3	3
CPU Req. (msec)	20	10	10	8	6	4	3	2	2	1
Expansion Factor	.11	1	1	1	1	1	1	1	1	1

Table 1: Default Simulation settings for individual flow

4.2 Fairness Measurement

As mentioned earlier, the objective of our composite scheduling algorithm is to allocate CPU and bandwidth resources proportionately and fairly according to Definition 1. CPU allocation and bandwidth allocation were measured every 5 seconds. We found that while using CBCS^{DRR}, the total allocation of resources for each flow remained more or less constant at any time. As sample results, the measurements at the 45th second are shown in Figure 5.

Figure 5: Total resource allocation at 45th sec using CBCS^{DRR}

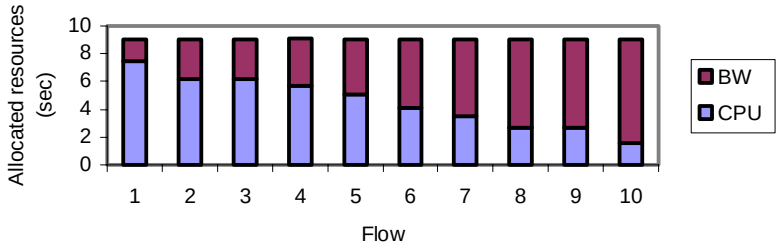
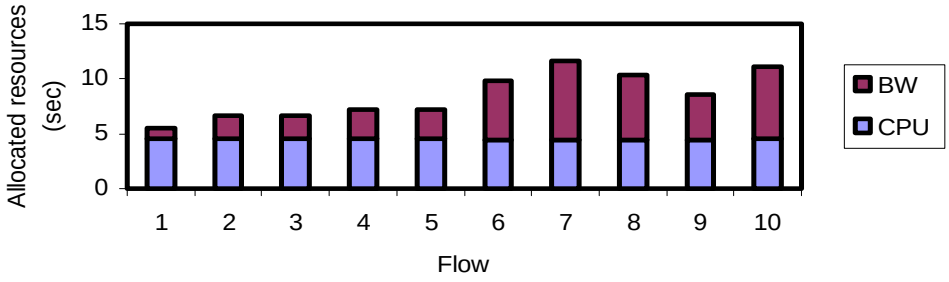


Figure 5 demonstrates that CBCS^{DRR} maintained perfect fairness by allocating more CPU to flows1–4 and more bandwidth to flows 7–10 while keeping the total allocation almost constant (which is ≤ 45x2/10 or 9 seconds). For instance, the total allocation for individual flows varied between 8.97 and 8.99 seconds. The measured CPU utilization was 100% and bandwidth utilization was 99.97%.

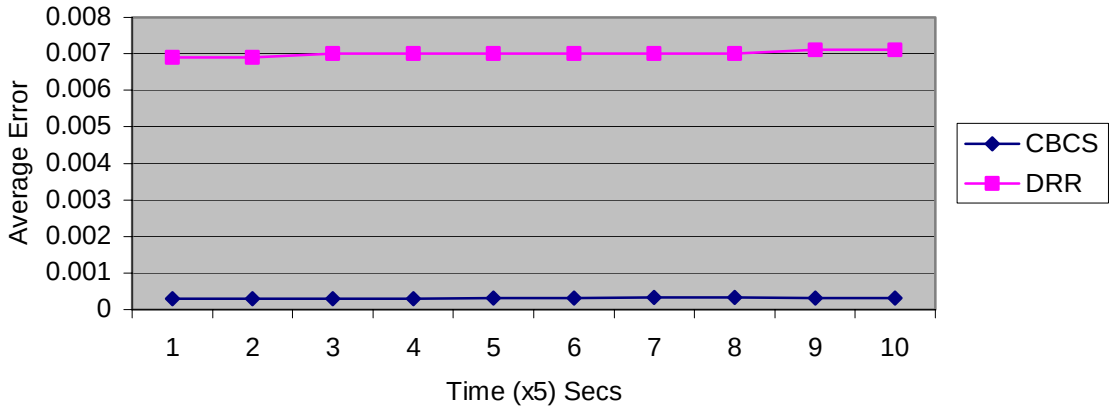
Figure 6 CPU and Bandwidth Allocation at 45th sec using DRR



Figures 6 show the resource allocation results using DRR separately for CPU scheduling and bandwidth scheduling. It demonstrates that maintaining fairness in CPU scheduling does not entail fairness in bandwidth scheduling. Though the CPU allocation between active flows remains fair, the

bandwidth allocation varies significantly and so does the total allocation per flow. The CPU scheduler dispatched more packets for the flows with lower CPU requirements (i.e., flows 6–10) compared to the flows with higher CPU requirements, resulting in a total bandwidth requirement of 149.48%. Since DRR serves only non-empty flows, the bandwidth scheduler allocated more bandwidth to flows 6–10. However, a significant number of packets from flows 8–10 were dropped in the DRR link queue, resulting in actual bandwidth utilization of 90.48% while the CPU utilization remained at 100%.

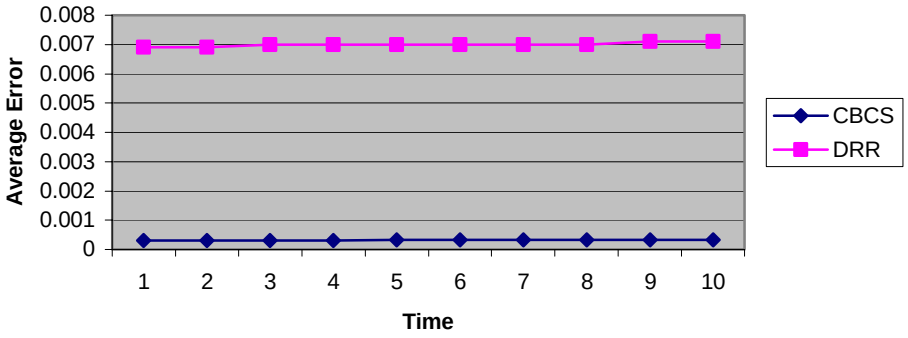
Figure 7 Average Error



We also measured the error (i.e., deviation from the ideal situation) as defined in Definition 2 at 5-second intervals for all the flows. Average errors are shown in Figure 7. The average error for each flow was recorded as 0.007 while using DRR. Thus, on average each flow deviates from the fair resource allocation constraints of 0.2 by 0.0837 or by 41.83%. When we use $CBCS^{DRR}$, the average error reaches almost zero. Thus, using $CBCS^{DRR}$ the achieved results did not deviate noticeably from the expected ideal situation.

4.3 Adaptivity to Highly Dynamic Traffic Patterns

Figure 8 Average Error under highly dynamic traffic patterns



In this experiment (carried under a different simulation settings) we show that CBCS^{DRR} is totally adaptive with the changing conditions of the incoming traffic pattern and with the varying resource requirement for each flow. We used a scenario where all the hosts generates active packet, and each hosts sends the packet with 50 different Code GUIDs. The code evaluator randomly selected the CPU requirements for the 500 different code within the range of 1 to 10 ms. All the hosts generated packets with varying packet size ranging from 1000 to 2000 bytes. Under this scenario, the average errors and total resource allocations of the flows are shown in figure 8. The figures again show that under highly dynamic environment, CBCS maintained the fairness in allocating resources.

4.4 Fairness under mixed traffic scenario

We have also performed experiments under a mixed traffic scenario where the Hosts 6 and 9 generated non-active packets and all other nodes generated active packets. The settings of the individual flows under mixed traffic scenario are shown in table 2 below.

Figure 9.a demonstrates that the CBCS^{DRR} maintained perfect fairness by allocating more CPU to flow 1-4 and more bandwidth to flow 7-10 in mixed traffic scenario while keeping the total allocation more or less constant. Also Figure 9.a shows that in mixed traffic scenario, the flow 6 and 9 received their fair total share all in bandwidth resource. Figure 9.b shows the total resource allocation results using DRR separately for CPU scheduling and bandwidth scheduling and demonstrates that maintaining fairness in CPU scheduling does not entails fairness in bandwidth scheduling. Though the CPU allocation among active flows remains fair, the bandwidth allocation varies significantly and so does the total allocation per flow.

Flow Number	**CPU Requirement**	**CBR Interval**
1	10 ms	5 ms
2	9 ms	5 ms
3	9 ms	5 ms
4	8 ms	5 ms
5	6 ms	2 ms
6	0	2 ms
7	3 ms	1 ms
8	2 ms	1 ms
9	0	1 ms
10	1 ms	1 ms

Table 2: Simulation settings under mixed traffic scenario

We have also measured the error (i.e. deviation from the ideal operating situation) as defined in the definition 2 at every 5 seconds for all the flows and the average errors are shown in figure 10. It shows that with mixed traffic scenario the CBCS^{DRR} records virtually zero errors.

Figure 9.a: CPU and Bandwidth Allocation for mixed traffic scenario scenario at 45 sec using CBCS^{DRR}

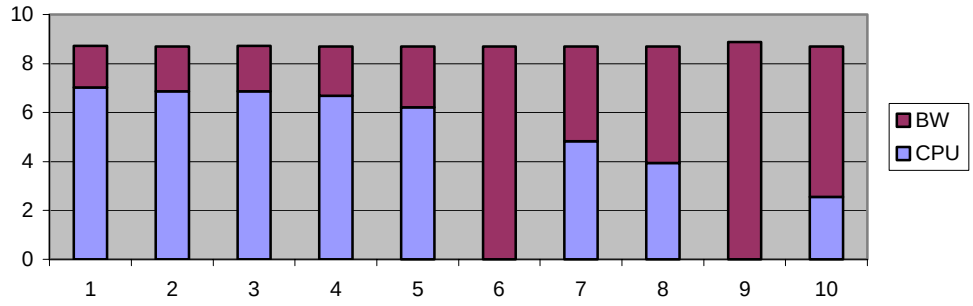


Figure 9.b : CPU and Bandwidth Allocation for mixed traffic scenario scenario at 45 sec using DRR

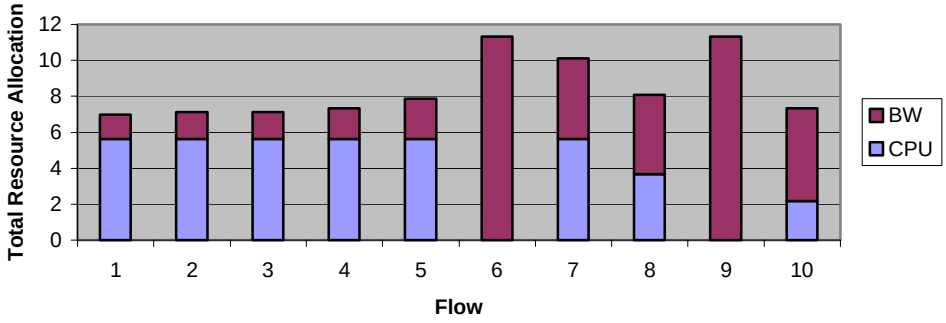
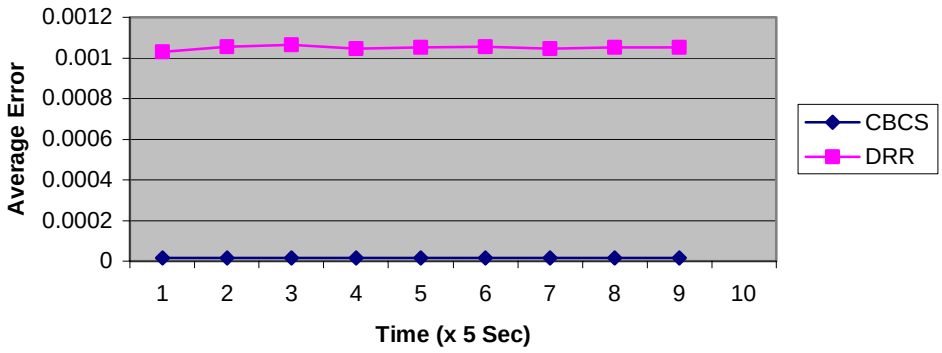


Figure 10: Average Error for mixed traffic scenario both CBCS^{DRR} & DRR



4.5 Delays and Throughput

We measured delays and throughput under another different simulation settings, where all the flows required 97.8% of the CPU resources and 98.9% of the bandwidth resources (i.e., resource utilizations were just below 100%) so that the measured delays are because of scheduling and not because of queuing backlog. Though the measured throughput of each flow remained the same under DRR and CBCS^{DRR}, the measured delays showed that CBCS^{DRR} could provide somewhat better delay guarantees. For example, the delay plots for the MPEG data traffic are shown in Figures 11.a and 11.b. Using DRR the delays range from 0.142 to 0.316 Sec whereas using CBCS^{DRR} they vary from 0.142 to 0.250 Sec.

Figure 11.a Delay for MPEG data traffic using DRR

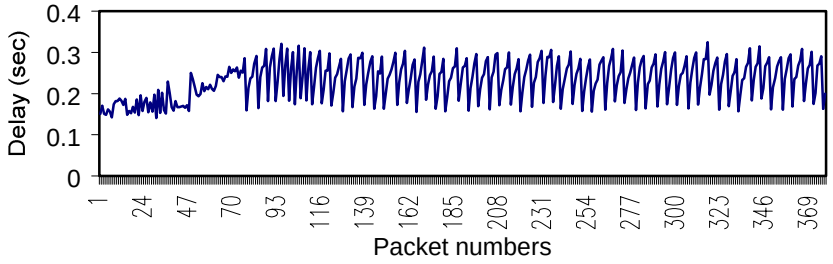
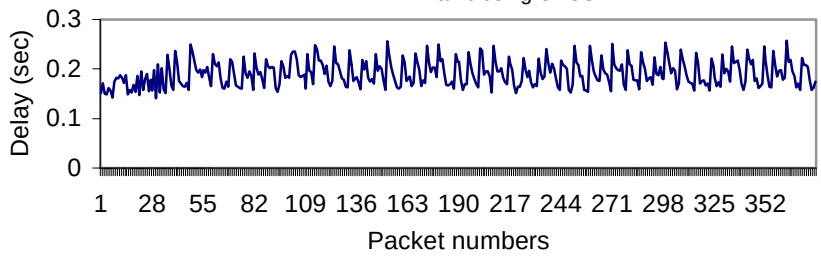


Figure 11.b Delay for MPEG data Traffic using CBCS^{DRR}



Flow #	Packet entering active node	Packets processed at active node		Packets received at sink	
		CBCS^{DRR}	DRR	CBCS^{DRR}	DRR
1	445	373	226	371	226
2	750	614	451	611	451
3	750	614	451	611	451
4	938	711	565	707	565
5	1050	841	751	838	751
6	1332	1031	1121	1028	1121
7	1623	1164	1495	1161	1495
8	2355	1335	2242	1332	1227
9	2352	1335	2243	1328	838
10	4820	1565	4491	1556	1378

Table 3: Throughput measurements under congestion

We also measured the throughput when all the flows were highly congested or backlogged, and the results are shown in Table 3. It shows that under highly congested scenario $CBCS^{DRR}$ can provide significant improvement on throughput and proper utilization of resources compared to using DRR separately for CPU and bandwidth scheduling. With DRR, a significant number of packets from flows 8–10 were dropped after being processed, thus wasting the consumed CPU resources.

5. CONCLUSION

Resource management is a major challenge for active networks. This work is aimed at improving network performance by managing resources efficiently. We have provided an efficient resource management architecture for intra-node and inter-node using directory service. We have resolved the issue of determining the CPU requirement for active packets by introducing a database at the active node. Our framework also enables a client to select an active node in the network, select the best end-to-end path that includes the active node, and reserving resources along the selected path. We have developed combined scheduling algorithms that could schedule both CPU and bandwidth resources adaptively and fairly between all the competing flows. We evaluated the performance of our combined scheduling algorithm $CBCS^{DRR}$ for fair service between all the competing flows through simulation. The simulation result shows that $CBCS^{DRR}$ offers better fairness, throughput, and delays than the traditional DRR algorithm used separately for CPU and bandwidth scheduling.

ACKNOWLEDGEMENT

This research work is supported by Co-Operative Research Centre (CRC) for smart Internet Technology, Australia.

REFERENCES

1. Campbell, H.D. Meer, M. Kounavis, K. Miki, J. Vicente and D. Villela, “A survey of programmable networks,” SIGCOMM Computer Communications Review, vol. 29, no. 2, April 1999
2. A.K. Parekh and R.G. Gallagher. “A generalized processor sharing approach in integrated services networks”, INFOCOMM ’93, 1993.
3. Ramachandran, R. Pandey and S.-H. Chan, “Fair Resource Allocation in Active Networks,” Proceedings of the IEEE International Conference on Computer Communications and Networks (ICCCN), pp. 468–475,Las Vegas, Nevada, Oct 16–18, 2000.
4. Galtier, K. Mills and Y. Carlinet, “Predicting and Controlling Resource Usage in a Heterogeneous Active Network (2001),” National Institute of Standards 2001.
5. Ion Stoica, Scott Shenker, and Hui Zhang. Core-stateless Fair queueing: A Scalable Architecture to Approximate Fair Bandwidth Allocations in High Speed Networks.”, SIGCOMM ’98.
6. M. Shreedhar and George Varghese. “Efficient Fair queueing using Deficit round robin.”, SIGCOMM ’95, August 1995.
7. P. Pappu and T. Wolf, “Scheduling Processing Resources in Programmable Routers”, Department of computer science, Washington University in St. Louis, MO, USA, WUCS-01-32, July 25, 2001.
8. F. Sabrina and Sanjay Jha, “A Novel Architecture for Resource Management in Active Networks Using a Directory Service” ICT03, February 2003.
9. V.y Ramachandran and R. Pandey, “Resource allocation in active networks”, Technical report TR-99-10, Parallel and Distributed Computing Laboratory, Computer Science Department, University of California, Davis, CA 95616, August 28, 2000.
10. Wetherall, J. Guttang and D. Tennenhouse, “Ants: A toolkit for building and dynamically deploying network protocols,” IEEE OPENARCH 98, San Francisco, CA, April 1998.
11. Tennenhouse, J.M. Smith, W.D. Sincoskie, D.J. Wetherall and G.J. Minden, “A Survey of active network research,” IEEE communications Magazine, pp.80–86, January 1997.
12. Wetherall, U. Legedza and J. Guttang, “Introducing new internet services:why and how,” IEEE Network Magazine, July/August 1998.
13. Fernando and B Kummerfeld, “A New Dynamic Architecture for an Active Network”, IEEE Openarch 2000.

14. NS2 Simulator, The LBLN Network simulator, University of California, Berkley.
15. Xiaohu Qie, Andy Bavier, Larry Peterson, and Scott and Karlin, " Scheduling computations on a software-based router," in proc. IEEE Joint International Conference on Measurement & Modeling of Computer Systems (SIGMETRICS), Cambridge, MA, June 2001, IEEE.
16. A.Demers, S. Keshav, and S, Shenker. Analysis and simulation of a fair queueing algorithm. In SIGCOMM '89, 1989
17. D. Wetherall, U. Legedza, and J. Guttang, "Introducing new internet services:why and how," IEEE Network Magazine, July/August 1998.
18. A. Fernando and B Kummerfeld, "Pants: Python active node transfer system," Tech Rep., University of Sydney, Australia, 1998
19. D.S.Alexander, W.A. Arbaugh, A.D. Keromytis, and J.M. Smith, "A Secure active network environment architecture: Realization in SwitchWare," IEEE network magazine, special issue on Active and programmable networks, pp.29-36, 1998
20. D. J Wetherall, "Service Introduction in An Active networks", PhD thesis, MIT, Feb.1999