Enhancing Ns-2 With DiffServ QoS Features

Ch. Bouras D. Primpas K. Stamos
Research Academic Computer Technology Institute, PO Box 1122, Patras, Greece and
Computer Engineering and Informatics Dept., Univ. of Patras, GR-26500 Patras, Greece
Tel:+30-2610-{960375, 960316, 960316}
Fax:+30-2610-{969016, 960358, 960358}
e-mail: {bouras, primpas, stamos}@cti.gr

Keywords: Quality of Service, DiffServ, ns-2, Bandwidth Brokers

Abstract

In this paper, we present work that we have carried out in extending the ns-2 simulator in order to study and validate Quality of Service issues and related architectures. In the case of the DiffServ framework, simulation is valuable since an analytical approach of mechanisms and services is infeasible due to the aggregation and multiplexing of flows. This paper covers work in extending ns-2 functionality towards the direction of realistic traffic generation and a series of mechanisms defined by the DiffServ architecture. We have also extended ns-2 with the functionality of Bandwidth Brokers, which are entities for managing the resources and negotiating end to end resource reservations between domains. The Bandwidth Broker ns-2 implementation is useful for studying the related architectures and admission control procedures.

1. INTRODUCTION

Because of the importance of simulating environments to research conducted in the area of computer and telecommunication systems, a lot of research work on telecommunications and more specifically in contemporary IP networks has been carried out in the last years. Two important trends have been the introduction and exploitation of the DiffServ framework towards the direction of building advanced networking services as an evolution to the traditional best-effort model, and the continuous penetration of wireless networks. Since ns-2 [1] has arisen as the most widely acknowledged simulator for packet switched networks, our work on studying the above issues has focused on extending the corresponding ns-2 functionality in these areas. ns-2 strength is that it is a powerful simulation tool that can simulate many kinds of networks and provide useful low-level insight in the operation of the networks.

The DiffServ framework [2] proposes the provision of service differentiation to traffic in a scalable manner, by suggesting the aggregation of individual application flows with similar quality needs. The modules presented in this paper have been built within the environment of ns-2 and comprise self-contained components, each one of which provides an additional traffic generation or DiffServ mechanism functionality.

In order to facilitate negotiations for automatic end to end QoS provisioning between domains, an additional mechanism has to be used. Such is the Bandwidth Broker [3], an entity that manages the resources within a specific DiffServ domain by controlling the network load and by accepting or rejecting bandwidth requests. For requests that span multiple domains (inter-domain requests), the Bandwidth Broker will have to communicate with Bandwidth Brokers in the adjacent domains that are traversed by the requested flow. Bandwidth Brokers only need to establish relationships of limited trust with their peers in adjacent domains, unlike schemes that require the setting of flow specifications in routers throughout an end-to-end path. Therefore, the Bandwidth Broker architecture makes it possible to keep state on an administrative domain basis, rather than at every router and the DiffServ architecture makes it possible to confine per flow state to just the leaf routers. Our work focuses on the implementation of several variations of the Bandwidth Broker module in ns-2, in order to compare the performance of different approaches, especially with regard to the important issue of admission control. The implementation and deployment complexity of such solutions makes it useful to be able to inexpensively study related research issues in a simulation environment.

The rest of this paper is structured as follows: Section 2 describes related work at the ns-2 and the issues discussed in this paper. Section 3 describes the implementation of the DiffServ functionality, while section 4 focuses on the implementation of Bandwidth Broker modules. Finally, section 5 concludes the paper.

2. RELATED WORK

There are a number of groups working on the ns-2 simulation platform worldwide. The DiffServ functionality supported by the current version of ns-2 is described in [1] [4]. In [5], a module for the functionality of Weighted Fair Queuing (WFQ) in the ns-2 environment is provided. In [6], an implementation of MPLS functionality is provided for ns-2. In [7], in the framework of testing a proposed

measurement based admission control algorithm, a number of realistic source models are implemented. The authors anticipate for long-range dependence (LRD) of network traffic. They present two types of LRD traffic simulation models, one based on Pareto ON/OFF processes, the superposition of which generates LRD series, and another one based on the fractional autoregressive integrated moving average process for the calculation of the number of fixed-sized packets to be sent back-to-back in each ON period of an ON/OFF source. According to relevant bibliography, the superposition of sources of the latter model effectively simulates aggregated VBR video traffic. Our approach differs from this one, because it anticipates for simulation of realistic aggregated background traffic, while at the same time providing modules for the generation of isolated flows in what we call 'foreground traffic'. Thus, we propose the use of distinguishable traffic flows that need to be individually monitored in a DiffServ environment, in order to observe how DiffServ mechanisms affect the characteristics of traffic and quality per application instead of per aggregate. Aggregated traffic in our model is only simulated to act in the background, in order to efficiently reproduce a realistic DiffServ environment where best-effort traffic co-exists with quality-demanding flows. In [1], several modules on DiffServ functionality are provided as a contribution by several research teams worldwide. Specific simulation on Bandwidth Brokers has taken place in the framework of the OPNET simulator [8].

3. DIFFSERV FUNCTIONALITY

3.1 Background traffic

Our module simulates the aggregated IP traffic existing in the backbone links of a MAN/WAN. It works as follows: A configurable number of peripheral nodes are created and connected on each of the two backbone link nodes. On each peripheral node, a configurable number of agents (TCP or UDP) for every type of traffic (SMTP, FTP etc) and a source for every agent are attached. Sources are modelled according to the ON/OFF model.

A number of input parameters are required for the module's operation. The major ones are the duration of the simulation (in seconds), the bandwidth (*bw*) and propagation delay (*pd*) attributes of the backbone ($bw_{backbone}$, $pd_{backbone}$) and access links ($bw_{access,i}$, $pd_{access,i}$) as well as the transmission rates of the TCP and UDP sources. In order for the load on the backbone link to be adjustable according to the transmission rates of the TCP and UDP sources and not limited by the access links' capacities, it is suggested that input parameters are such that $bw_{backbone} < bw_{access,i}$ *for every* i

The analysis for the characteristics of the produced background traffic can be found in [9].

Four procedures, one for each type of simulated traffic (SMTP, FTP, TELNET and HTTP), have been implemented. Each one of them is responsible for creating traffic sources for the corresponding traffic type, defining the transport protocol used (TCP or UDP) and the produced packets' size, creating the required agent to which the traffic source is attached and attaching the agent to a peripheral node of the topology. Finally, the traffic source is triggered to start generating packets towards a selected sink. The input parameters for these procedures are the pointers to the source and sink nodes and the duration of simulation.

Operation of sources is controlled by another set of procedures, one per traffic type, that are responsible for defining the details of transmission for each traffic source. More specifically, they define the idle and active state intervals for each source, according to the distributions and packet inter-arrival times as detailed in [9]. It is here that the destination node or sink for each source's transmission is randomly selected from the group of nodes on the other end of the backbone link to which the current source is attached. The selected sink differs between consecutive active states of the source. These two groups of procedures are mainly responsible for traffic generation over the already described topology.

The background traffic module is created so as to function according to the "cross traffic" model. As such, the module can be used on a simulation topology by making a call:

generate-backgr-traffic *<node1> <node2>* $<bw_{backbone}>$ $<pd_{backbone}>$ $<bw_{access,i}>$ $<pd_{access,i}>$ *<link>*

where node1, node2 are the nodes between which cross-traffic has to be created and link is the topology link between the two nodes. This call can be used to fill each link of the simulation with background traffic.

3.2 Foreground traffic

The foreground traffic simulation module has been created by the creation of flows belonging to isolated VoIP transmissions, streaming video and H.323 conferencing traffic flows. For the simulation of VoIP data transmission an exponential ON/OFF distribution was used, with an average duration of ON periods equal to 1.004 sec, average duration of OFF periods (idle-time) equal to 1.587 sec, packet size of 188 bytes (8 byte UDP header+20 byte IP header+160 byte voice data) and a transmission rate during the "on" period of 80Kbps. These values obey the principles of [10]. For the realistic reproduction of aggregated VoIP traffic multiple flows with these characteristics can be created simultaneously.

For each module of the foreground traffic simulation, a flow of a traffic type can be generated by making a call:

create-<type of traffic> *<source><sink>*

where source and sink are the source and sink nodes between which foreground traffic is generated.

3.3 Leaky Bucket Shaping

Leaky Bucket is a shaping algorithm, introduced in [11], according to which packets arriving with random rate are shaped to a configurable constant rate. The Leaky Bucket

anticipates for a buffering capacity with constant size. When packets arrive when the "bucket" is full, they are dropped. We have implemented the Leaky Bucket algorithm within the ns-2 environment, so as to provide the shaping functionality for DiffServ experiments.

The basic idea of our implementation is to delay each packet (if needed) for the appropriate time in order to shape traffic in a constant rate. The best way to do that is to delay the packets immediately when they enter a network node (the router), before the routing and scheduling processes take place. The critical point is to calculate the right delay time.

The existing implementation of ns-2 did not allow handling of packets when entering a router (edge or core) due to the fact that the implementation of the router itself lacks a receive function of its own and it inherits the receive function from an ancestor class. So we overloaded the receive function so as to call the Leaky Bucket function if configured to do so by the simulation script. In this way, the shaping functionality is inserted before the routing process.

The Leaky Bucket module requires the specification of three input parameters, namely the DSCP of the packets to shape, the rate to which traffic is shaped (in bits/sec) and the depth of the Leaky Bucket used to accumulate packets during shaping (in bytes). The Leaky Bucket module can be used by making the call:

<router_interface> Leaky-Bucket *<DSCP> <shaping_rate> <bucket_depth>*

where router_interface is the router port where the Leaky Bucket shaper is applied to all packets carrying the specified DSCP value.

The LeakyBucket module can be addressed via a simulation script during topology generation. A series of variables are then initialised for the DSCPs, the rate, the maximum size and the current size of Leaky Bucket. Each packet arriving at an interface for which a shaper is configured is examined regarding its DSCP value. If a shaping entry for the packet's DSCP exists, the available buffer space of the shaper is examined. If the packet cannot be accommodated in the buffer it is dropped, otherwise the available buffer space and number of packets in the buffer are updated. The interval for which each packet must be delayed in order for the required shaping to be achieved is also calculated. This calculation is based on the number of the packets that are already delayed in the shaper's buffer and the time of the last packet's departure. Each packet is scheduled to be released for transmission after the calculated period of time. At that time the available buffer space and number of packets in the buffer are updated once again.

The algorithm for the implementation of Leaky Bucket is presented in Figure 1. A function that prints the statistics of the Leaky Bucket shaper, namely the number of shaped and dropped packets has also been implemented and embedded in the Leaky Bucket module.

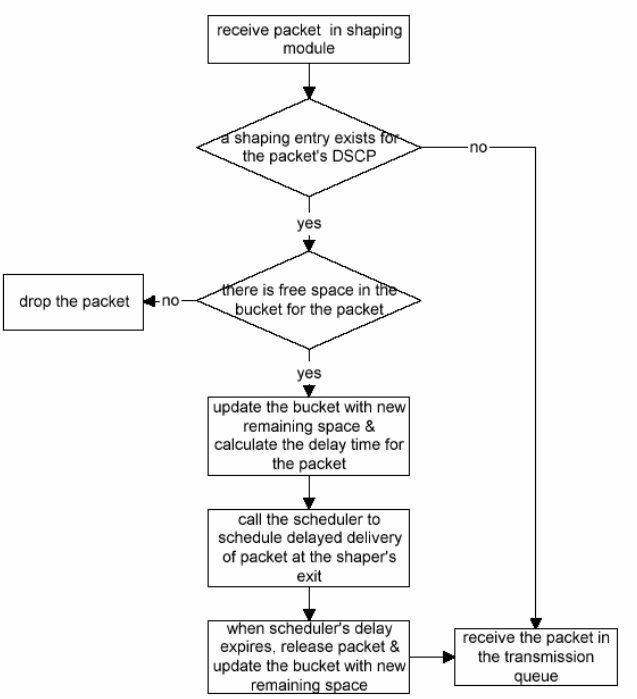

Figure 1. Flow chart of Leaky bucket shaping implementation

3.4 DiffServ-based tracing

In the original version of ns when trace-all for tracing of all events in the entire simulated topology is enabled, all packets are traced unconditionally. This leads to very large trace files. It therefore is desirable to only keep traces for only for packets that belong to a specific service class or to specific flows.

In our implementation, the individual flow identification numbers (flow-ids) and the value of the priority fields (or DSCPs) of the flows and classes of packets that we were interested in tracing are stored in appropriate data structures. The receive function of tracing objects has been updated so as to always consult the list of flow-ids and priority field values for which tracing has been explicitly enabled and forward to the tracing process only matching packets. The remaining packets are forwarded directly to the next downstream object of the ns topology, escaping the tracing process. Explicit definition of flow-ids and priority fields for which tracing must be enabled can be performed by:

set_DSCP_to_trace *<DSCP1><DSCP2>.. <DSCPn>*

and/or

set_fid_to_trace *<fid1> <fid2> ... <fidn>*

A packet the flow_id or DSCP value of which has been included in a command such as these is always traced.

3.5 Scheduling at the ingress interface

The purpose of the implementation of the module for packet scheduling at ingress interface queues is to create a queue mechanism at the routers, where the packets are put

in different queues depending on the incoming link that brought them there and are scheduled according to their priority towards a transmission queue of an egress interface. Ingress scheduling is a functionality that a series of Cisco Gigabit Switched Routers (GSRs [12]) offer and has been introduced to solve the head-of-line problem for high-priority traffic in large speeds.

The basic class of the module (Queue_input) is derived from the original Queue class of ns-2, inheriting all necessary functionality for a router queue. Queue_input was augmented with a list of queues (Queue_per_link objects) where the packets are enqueued depending on the incoming link that brought them to the router. This is achieved by checking the field of IP packet headers that denotes, when tracing is enabled, the upstream node for each packet. Scheduling between the "Queue_per_link" queues is performed in a round-robin fashion.

Each Queue_per_link object is a Random Early Detect (RED) Queue of the original ns-2 DiffServ architecture that simulates the queuing mechanism at an egress interface of a real router. Packets are enqued in different sub-queues depending on the priority field or DSCP value. For scheduling between the physical subqueues of the Queue_per_link object, any of the supported scheduling algorithms inherited from the original ns-2 RED Queue can be used. The architecture implemented is shown in Figure 2.

In the example of Figure 3, a sample configuration for ingress interface scheduling at a router with two upstream links and three outgoing links is presented.

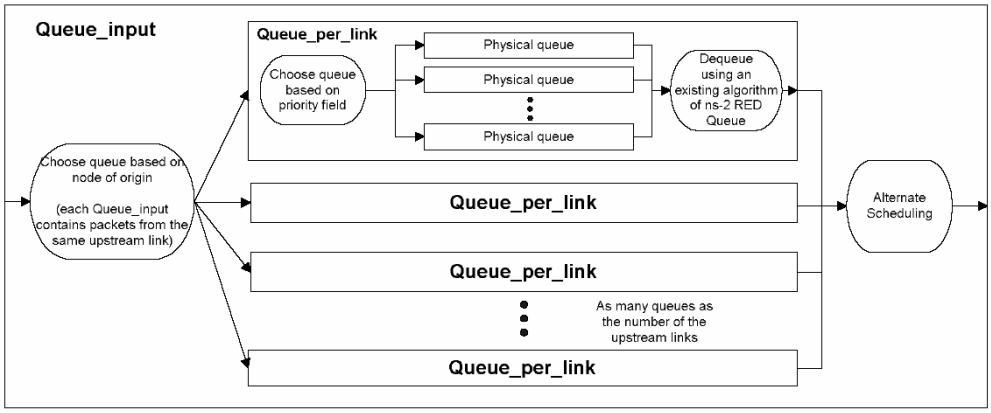

Figure 2. Architecture of the ingress interface scheduling module

```
for each outgoing link i of a simulated router {
  create a Queue_input(i) object
  set number of Queue_per_link of Queue_input(i) = 2
  set size of each Queue_per_link of Queue_input(i) = 5000 packets
  arrange so that packets coming from upstream node s(0) go to Queue_per_link 0
  arrange so that packets coming from upstream node s(1) go to Queue_per_link 1
  set number of physical queues of each Queue_per_link of Queue_input(i) = 3
  set scheduling algorithm between physical queues of each Queue_per_link of
                                                  Queue_input(i) to be e.g. Round Robin
  for each Queue_per_link of Queue_input(i) {
    for all possible DSCP values of packets entering Queue_per_link(j)
      arrange the packets with DSCP = k to be enqueued to an underlying physical
                                          queue, according to the local DiffServ policy
    }
  }
  connect Queue_input(i) to the ns topology before the router's outgoing_link(i)
}
```

Figure 3. Sample configuration required in simulation scripts for using the ingress interface scheduling module

3.6 MDRR scheduling

In the DiffServ module of ns-2 there are the following algorithms for scheduling at the edge and core routers: Round Robin (RR), Weighted RR (WRR), Weighted Interleaved RR (WIRR), and Priority (PRI). We implemented two more algorithms, Modified Deficit RR–Strict (MDRR_STR) and Modified Deficit RR–Alternate (MDRR_ALT), that are used by Cisco GSRs [14].

In the MDRR scheduling algorithm, all queues, except for the low latency one (LLQ), are served in a Deficit Round Robin (DRR) fashion. Each one of DRR served queues can be configured with a weight, according to which an initialisation quantum (in bytes) defining the maximum number of packets that can be uninterruptedly served by the queue is calculated. A deficit (initially equal to the corresponding quantum) is applied to each DRR queue and is decreased by the size of a packet, each time a packet exits the queue. The scheduler moves on to the next queue to be served when the current queue's deficit becomes zero or negative. When all DRR queues have been served, completing a round, all DRR queues' deficits are augmented by the corresponding quantum values, depending on the configured queues' weights.

There are two alternative modes for serving the LLQ:

- MDRR strict priority scheduling
- MDRR alternate priority scheduling

In both modes the weights for each physical queue on a router interface can be set by calling:

<router_interface> addQueueWeights *<physical queue> <weight>*

Both modes were implemented as additional modules to the scheduling alternatives of the Random Early Detect (RED) Queue of the original ns-2 DiffServ architecture.

In Strict Priority mode, the LLQ is always served if packets are queued. Thus, every time the queue to be served has to be elected, the LLQ is examined. If a packet is ready (at the head of the LLQ) to be transmitted, then it is immediately placed on the transmission medium. Otherwise, the round robin-fashion of serving the DRR queues picks up from where it was interrupted, when a previous packet appeared at the head of the LLQ queue. It can be activated by defining for each router's interface:

<router_interface> setSchedularMode MDRR_STR

In alternate priority mode, service alternates between the LLQ and the other DRR queues. The LLQ now obtains its own weight, quantum and deficit values, the latter of which is updated every time an LLQ packet is served and augmented by a value equal to the current queue's quantum at the end of each LLQ-DRR1-LLQ-DRR2-LLQ-DRRn round.

MDRR alternate priority scheduling algorithm can be activated by defining the scheduling mode for each router's interface during the simulation topology set-up:

<router_interface> setSchedularMode MDRR_ALT

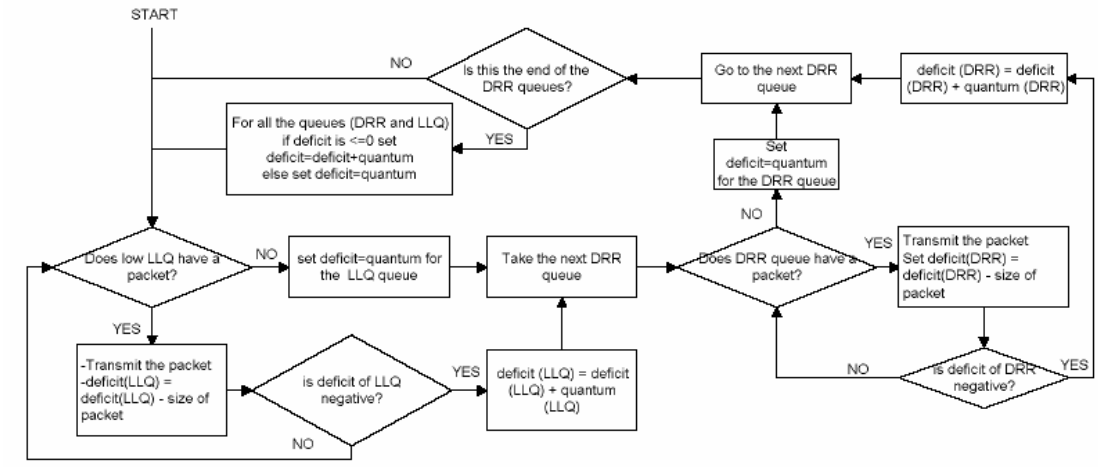

Figure 4. Flowchart of MDRR-Alternate scheduling algorithm

4. BANDWIDTH BROKERS

4.1 Implementation in ns-2

After the implementation in ns-2 of the basic QoS characteristics of the service, we extended our implementation in order to include modules for automated inter-domain provisioning of QoS services, such as the Bandwidth Broker (BB) concept [3]. In order to implement it correctly, it was necessary to make several changes and additions to the ns-2 structure and source code. An agent in ns-2 represents an endpoint where packets are consumed and constructed, using a specific protocol. The BB that was implemented is based on two new agents, the Edge BB and the Base BB. More specifically, we created the classes BBedgeAgent and BBbaseAgent, derived from class Agent. We also created two new packet types, BBB and BBE,

which are used for the BB interfaces (to simulate the BB messages) and have a size of 64 bytes. BBbaseAgent creates BBB packets and consumes BBE packets created by the BBedgeAgent. BBedgeAgent creates BBE packets and consumes BBB packets created by the BBbaseAgent. In order to create a new packet type, it is necessary to define the header of the new packet. The header fields that we defined for the BBB and the BBE packets were the address of the sender of the RAR, the address of the other end node, the type of the packet (RAR or RAA), the amount of the requested bandwidth and the final answer the BaseBB sends to the sender (Negative or Positive). The total bandwidth that the BB manages on each link is determined by a new tcl instruction "***set_bndw***". The syntax is *"BBedgeAgent set_bndw node_id bandwidth"*. This instruction informs the BBedgeAgent for the bandwidth that the BB will manage on the link that exists between the node where the BBedgeAgent is running and its adjacent node with node-id node_id. A BBedgeAgent, which represents a client (user / application), can send a RAR requesting guaranteed bandwidth between the node where it is running and another node with id *node_id* using the new tcl instruction "***sendto***". The syntax is *"BBedgeAgent sendto node_id bandwidth"*. The BBedgeAgent that exists on every node simulates a situation where a BB client is connected to a router on a real network. This agent operates as client that communicates with the base BB and updates its local router with the configuration modifications according to new admissions.

4.2 The supported QoS service

The Bandwidth Broker provides a QoS service with the characteristics of bandwidth guarantee as well as minimum delay and jitter. This service is the IP Premium and is currently supported by many network providers. The main characteristic of this service is that it follows the classic DiffServ architecture. It classifies the packets using the DSCP values for admitted and downgraded packets. The policing is performed at the edge of the network and high priority queuing is applied in the core and access routers at the outgoing interfaces.

Having already enhanced ns-2 so that the classification is done using the DSCP field of the IP header, enables packets that have the same source and destination nodes but belong to different applications to belong to different classes as well, and packets with different source and destination nodes to belong to the same class.

The QoS service has the responsibility of packet classification and policing. If the BBedge agent receives a positive answer about a request it has submitted, it configures through tcl all the edges that exist on the request path. After the configuration process has been completed, the BBedge agent can start using the requested and allocated network resources.

The QoS implementation starts with the insertion of the DSCP value into the packet headers for packets that use the requested service. When these packets are inserted into the network with the proper DSCP value, strict token bucket policy is applied to them, when they are in the first BBedge agent. This action guarantees that the transmitted rate matches the requested (admitted) rate. Next, the queue management mechanism is properly configured. The used queue management mechanism is a high priority queue on every node, which is used for all the admitted traffic classes.

4.3 Admission control algorithms

There is a large selection of admission control algorithms that can be followed for the BB. We have implemented and experimented with the performance of four different admission control modules. The first algorithm is simple admission control (SAC), the simplest type of admission control, with easy implementation and low complexity. Each incoming request is examined by itself, and is accepted if there is still available bandwidth for the service (that is, the total bandwidth available for the service minus the already reserved bandwidth). Therefore, this algorithm displays identical behaviour each time it is presented with the same set (and with the same temporal succession) of incoming requests.

Another approach to the admission control issue is taken by the Price-based algorithm (PBAC). This type of admission control is similar to the offline version of the algorithm presented in [14]. It makes a decision on which requests will be accepted trying to optimize the network utilization by gathering and evaluating multiple requests together. In order to solve the NP-complete problem that arises, an approximation algorithm is used.

The Adaptive admission control (AAC) [13] is an algorithm that tries to gather multiple requests and evaluate them together for purposes of increasing the resource utilization, but also uses an adaptation module in order to keep processing requirements low. The adaptation module is responsible for interrupting the process of solving the scheduling problem and for adjusting the size of subsequent instances of the scheduling problem based on constant monitoring of computation time. The algorithm includes a couple of parameters that can influence its behaviour. The first parameter is the adaptation parameter *a*, which takes values in the range from 0 to 1 and determines the aggressiveness of the adaptation. The second parameter is the threshold, which roughly determines the limit of the computational overhead that the algorithm incurs to the system.

Adaptive admission control with resubmissions (AACR) [15] is a variation of the AAC algorithm. It is enhanced with the capability to recognize previously rejected requests and increase their priority. Other than that, this algorithm is very similar to AAC. The basic idea is that the client will resubmit a rejected request only if the BB has indicated that the request should indeed be resubmitted, and if the user is willing to compromise for a delayed reservation. In order for the BB to utilize resubmitted requests, it needs to keep a list of the standby requests. Moreover, it actively prioritizes

such requests in expense of newly received requests, and the prioritization depends on the duration that a specific user has been waiting and resubmitting a request.

Implementation of the three advanced admission control modules was based on the utilization of the open source GLPK linear programming library [16], in order to solve instances of the optimization problems. A number of array and vector structures were used in order to keep the requests and categorize them according to the requirements of each algorithm.

4.4 Evaluation results

For each experiment we have measured the percentage of accepted requests, the delay that was required before the Bandwidth Broker would reply to a request and the percentage of network utilization achieved by each algorithm.

These results are summarized in Table 1.

Table 1. Summary of results

Averages per algorithm	Acceptance rate	Average delay (time slots)	Average network utilization (bytes x time slots)
SAC	29.60%	0	3920014
PBAC	21.79%	7.08	5243307
AAC thr=5	25.72%	5.44	4532672
AAC thr=10	24.77%	5.48	4780385
AACR	42.56%	5.58	5594577

The following figures display in more detail the behaviour of the algorithms under different experiments with different request frequencies, and they help reveal the features of each algorithm, its relative weaknesses and strengths.

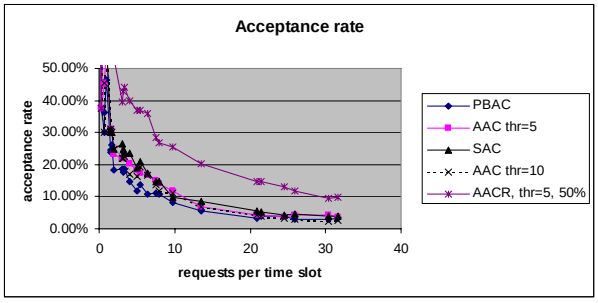

Figure 5. Acceptance rate

As Figure 5 demonstrates, the acceptance ratio of all algorithms except AACR remains fairly similar throughout the experiments. SAC is the algorithm that slightly achieves the highest acceptance rate, while PBAC is the one with the lowest, with AAC variations covering the middle. This is not a surprising result, since SAC will always accept a request if there are enough resources available, while PBAC is more oriented towards generating the maximum amount of resource utilization, rather than treating all requests alike. Because of the resubmission capability, AACR displays clearly better performance with regard to this metric. This result leads us to the conclusion that in environments where the most significant factor is the satisfaction of the maximum amount of users regardless of their relative weight, the good performance of the SAC algorithm combined with its simplicity make it the most suitable choice. If resubmissions are desirable and can be supported, AACR can then be used for its advantages.

In most cases however, all users will not generate the same revenue for the network provider and a cost scheme will most probably have to take into account both the relative weight of each request, and the effort to maximize the efficiency and utilization of currently available resources. We have tried to cover this aspect with Figure 6 and Figure 7, which display the total absolute profit generated for each experiment and the profit per request respectively. We have chosen to measure the provider's profit by calculating the product of a request's duration (in time slots used by the ns-2 simulator) times the resource allocation that a reservation requires.

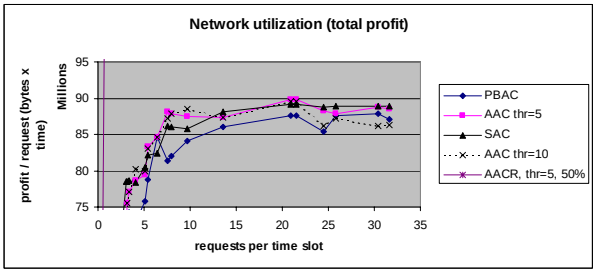

Figure 6. Network utilization

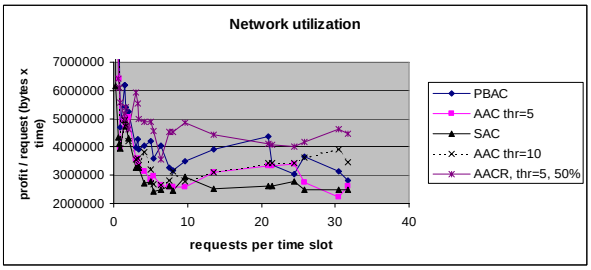

Figure 7. Network utilization per request

We have to mention that in Figure 6 AACR results are not displayed because they are far larger than all other results, in order to have better distinguishing capability for the rest of the algorithms. These results demonstrate the relative strengths of the price-based approaches, since PBAC is the most efficient algorithm in this regard, followed by AAC, with SAC displaying the worst performance. AAC even surpasses the PBAC performance in several cases when the request arrival ratio increases. The most plausible explanation for this result is that the increased arrival rate of new requests makes the larger size of the set examined by the PBAC algorithm unnecessary. Increasing the threshold for the AAC algorithm seems to

have a positive effect on its performance, but comparison with PBAC shows that a restrained increase in the threshold value is enough for obtaining equal or superior results. Therefore, the recommendation for fine-tuning the AAC algorithm is that it is beneficial to increase the threshold value as soon as the arrival rate of request increases. As expected, AACR again displays the best overall performance, which on the case of total profit exceeds several times the results of other algorithms.

In most real environments it is expected that a relatively quick response to a request will be essential. As Figure 8 demonstrates, SAC is extremely responsive as expected. This also means that there is room for a trade-off that can be used to improve performance in other areas such as the utilization of the network resources. PBAC is not efficient in that regard, as it demands the most time in order to respond to the reservation requests, a situation that in many real-world scenarios is unattainable. The adaptive variations prove to be attractive trade-offs, since for most of the experiments the additional delay they incur is minimal, while at the same time they manage to improve the utilization of the provider's resources, as demonstrated above.

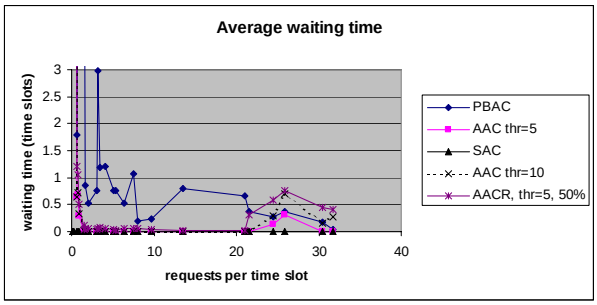

Figure 8. Average waiting time

5. CONCLUSION – FUTURE WORK

In this paper we have presented a series of modules implemented in order to extend the DiffServ functionality of the ns-2 simulator [18] that are provided free for use. We believe that such work is important because of the complexity and cost of implementing such mechanisms and architectures in real world environments. ns-2 can therefore be very useful in extracting conclusions for the performance and feasibility of these protocols and architectures.

Furthermore, our ns-2 implementations have helped us evaluate more sophisticated architectures for automatic provisioning of QoS services based on the Bandwidth Broker concept.

Our future work in this area will be focused on extending the simulated environments with more realistic characteristics, extending the scope and variations of the experiments including federated networks with independent bandwidth broker instances [17], and also compare the results from simulations with results from actual implementations.

6. REFERENCES

[1] S. McCanne and S. Floyd, "ns Network Simulator", available at: http://www.isi.edu/nsnam/ns/

[2] S. Blake et al., "An Architecture for Differentiated Services", RFC 2475, December 1998

[3] RFC 2638 "A Two-bit Differentiated Services Architecture for the Internet", K. Nichols, V. Jackobson, L. Zhang, July 1999

[4] Peter Pieda, Jeremy Ethridge, Mandeep Baines, and Farhan Shallwani. A Network Simulator, Differentiated Services Implementation. Open IP, Nortel Networks, 2000.

[5] R. Wielicki, "ns-2 ad-ons page", found at: http://thenut.eti.pg.gda.pl/~rafalw/wfq/ (Accessed May 2006)

[6] G. Ahn, "MPLS Network Simulator", found at: http://flower.ce.cnu.ac.kr/~fog1/mns/index.htm

[7] S. Jamin, P. B. Danzig, S. Shenker, and L. Zhang, "Measurement-Based Admission Control for Integrated Services Packet Networks", in proceedings of ACM SIGCOMM'95, pp. 2 – 13, Cambridge, USA, 1995

[8] Academic OPNET Research and Educational Projects, http://www.utdallas.edu/~lqian/opnet/ (Accessed May 2006)

[9] C. Bouras, D. Primpas, A. Sevasti, A. Varnavas, "Enhancing the DiffServ Architecture of a Simulation Environment", Sixth IEEE International Workshop on Distributed Simulation and Real Time Applications, Fort Worth, USA, 11-13 Oct. 2002, pp. 108-118

[10] ITU-T, P.59, 'Artificial conversational speech', (03/93)

[11] ATM Forum, "Traffic Management Specification", Version 4.0, af-tm-0056.00, April 1996

[12] Cisco 12000 Series Internet Router: Frequently Asked Questions, http://www.cisco.com/warp/public/63/gsrfaq_11085.shtml (Accessed May 2006)

[13] C. Bouras, K. Stamos, "An Adaptive Admission Control Algorithm for Bandwidth Brokers", 3rd IEEE International Symposium on Network Computing and Applications (NCA04), Cambridge, MA, USA, August 30-Sept. 1 2004, pp. 243-250

[14] C. Chhabra, T. Erlebach, B.Stiller, D. Vukadinovic "Price-based Call Admission Control in a Single DiffServ Domain", TIK-Report Nr. 135, May 2002

[15] C. Bouras, K. Stamos, "Resubmissions and Partly Defined Requests in an Adaptive Admission Control Algorithm for Bandwidth Brokers", In Proceedings of 5th International Conference on Networking (ICN 2006), 23-26 April

[16] GLPK (GNU Linear Programming Kit), http://www.gnu.org/software/glpk/glpk.html (Accessed May '06)

[17] C. Bouras, D. Primpas, "Pathfinding architectures for interdomain Bandwidth Broker operation", 14th IEEE International Conference on Networks (ICON 2006), Singapore, 13 - 15 September 2006

[18] CTI's NS-2 enhancements, http://ru6.cti.gr/ru6/ns_home.php (Accessed October 2006)