A Simulation Suite for Internet Nodes with the Ability to Integrate Arbitrary Quality of Service Behavior

Klaus Wehrle, Jochen Reber, Verena Kahmann

University of Karlsruhe – Institute of Telematics
Zirkel 2, D–76128 Karlsruhe, Germany
eMail: { Wehrle, Reber, Kahmann }@telematik.informatik.uni-karlsruhe.de

Keywords: Internet Protocol, Quality of Service, CPU- and Packet-Scheduling, Parallelism in protocol processing

Abstract

In the last few years, the Internet community spent a lot of research efforts in investigating several kinds of mechanisms to provide better services than the traditional best effort delivery. Simulating the behavior of Internet routers and hosts with modified QoS behavior is one of the most important ways to prove their guarantees. Existing models available for common simulation tools have not been implemented in such a modular way as to reuse them easily according to the principles of new Internet services.

In this paper, a modular simulation suite of complete Internet nodes with a realistic simulation of the Internet protocol and the underlying layers will be presented. The IP model is designed in a very detailed way, covering all features of the protocol, including mechanisms for ICMP and Multicast, unlike other network simulation models. The model considers hardware restrictions of IP processing computers as well. The second feature of the simulation suite is the ability to build any quality of service behavior. This is supported by elementary QoS models, which can be combined and integrated very flexibly. The simplicity to design this modular and reusable models is supported by the modular architecture of the chosen simulation environment – the OMNeT simulation tool.

1 Introduction

One of the reasons for using network simulation tools is the modeling of new networking concepts for validation of functionality, efficiency or scalability in global networks, such as the Internet. Quality of Service (QoS) is such a concept, particularly the Differentiated Services architecture [1] developed for the Internet in recent years.

As this architecture is designed of small building blocks called Per Hop Behaviors (PHB) composed to form Quality of Service behavior, an adequate simulation for the Differentiated Services model requires the feature to quickly build models from elementary entities. Additionally, the purpose of this paper is to analyze the behavior of QoS concepts in the Internet in a realistic way. For instance, real hardware has only a limited number of CPUs, which can only process a small number of packets at any one time.

For implementing the simulation suite, the OMNeT++ simulation environment [8] presented in section 2 has been chosen. It will be shown that this simulation environment is suitable to simulate complex systems like Internet nodes in a realistic way under consideration of the number of CPUs and the restriction to process more than one packet within one node. It also allows the design of modular simulation models, which can be combined and reused in a very flexible way. In section 3 we present the simulation model of Internet nodes. Besides that, OMNeT allows the composition of models with any granular hierarchy. Therefore, it is possible to build any Quality of Service behavior by reusing and concatenating basic QoS Behavior Elements. This concept will be shown in section 4, before we conclude the paper in the fifth section.

Currently, our notion of a realistic simulation suite of QoS behavior in the Internet is extended by integrating other aspects of Internet end systems like traffic generators for common applications (HTTP, FTP, multimedia applications) or transport protocols like TCP into the presented simulation model. However, for the sake of clarity, this paper concentrates on the network mechanisms.

2 The OMNeT++ simulation tool

OMNeT is a free object-oriented modular Discrete Event Simulation (DES) tool available under [8]. DES systems are especially suitable for the simulation of computer systems and communication protocols, because processing functionalities and protocol proceeding can be well modeled in discrete steps.

The OMNeT simulation system consists of C++ class libraries, forming the simulation kernel and the interface to the user environments. A model is compiled and linked with these libraries resulting in an executable file. The object-oriented approach allows the flexible extension of the base classes provided in the simulation kernel.

Simulated models are composed of hierarchically nested modules. There are two types of modules: First, so-called *Simple Modules*, which form the lowest hierarchy level and implement the activity of the module. Simple Modules can be arbitrarily put together to *Compound Modules*, the second module type.

From this distinction of module types, a distinction of the model description is made. The model topology, i.e. the hierarchy of modules and the connections between them, is defined in NED (Network Environment Description) – a simple and easy to understand C++ style language. A NED compiler transforms the NED definition of a module into C++ code. The Simple Module's activities are implemented by C++ code. The code implements both the module's algorithm as well as its communication with other modules via message sending.

OMNeT++ has been chosen as the simulation tool for the Internet simulation as it provides a flexible hierarchy where concepts and algorithms can be modeled in a modular and reusable fashion. Therefore, only the NED description of a new composed model has to be created – coding in C++ is not necessary. In other tools like ns-2, a lot of coding effort has to be spent on extending the module hierarchy, because several class definitions have to be implemented. In OMNeT++, a computer system like the presented Internet node can consist of a hierarchy of arbitrary granularity. Another advantage is that traffic generators may be included easily at each layer, so measurements of single layers can be taken independently of others. In contrast to other simulation tools, where only protocol processing is supported in most cases, it is possible to model packet processing as done in a real computer system. Since we wanted to get a realistic Internet simulation, we have modeled packet processing times in our Internet router model (cf. section 3). In ns-2 there is no processing time spent in the routing module, which we do not expect to be realistic behavior.

The modular object-oriented concept of OMNeT++ enables flexibility in the sense that building blocks can be exchanged and extended. For each building block, i.e. for each Simple Module, it is possible to choose between process-style or protocol-style modeling. Therefore, different parts of computing and communication systems can be connected easily. With this idea of flexibility, OMNeT++ allows to model new networking concepts and include them into a model – as it is done for the presented simulation of Quality of Service mechanisms (refer sec. 4).

3 Simulation of the Internet Protocol

The highest level of abstraction in the simulation of IP is the network itself, which consists of *IP Nodes*. IP nodes can represent routers or hosts. That organization allows the easy re-combining of different components to a new network.

3.1 IP Node structure and hierarchy

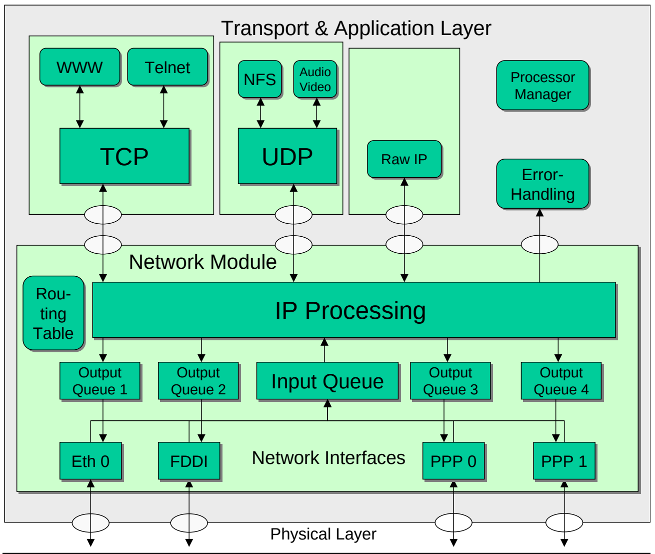

Figure 1 Components of the IP node simulation model

An IP Node in the simulation model is a computer capable of understanding the Internet Protocol, and is represented by one compound module.

The modules responsible for the simulation of the Internet Protocol are structured in a very similar way to the processing of IP datagrams in a real operation system, rather than organized to allow only a protocol processing, such as done in OPNET and ns-2. The structure of the Linux kernel has been chosen as model for the simulation of an IP Node. Individual changes in the Linux kernel would only result in local changes in the simulation model.

The *IP Network* module is mandatory in an IP Node, as it represents the network layers of the node. The upper layers for the transport protocols and other applications are optional.

They are usually present in hosts, but may be absent in routers. The IP Network module may have one or several connections out to other IP Nodes, either over point-to-point links or shared medium simulation entities.

The *Processor Manager* module is obligatory. It handles user and kernel process time scheduling and ensures that one IP Node can only have one kernel process running at any specific instance of time. The Processor Manager makes sure that the number of running user processes is limited by the number of processors of the IP Node. It is the Processor's Manager responsibility to not allow more than one IP datagram to be processed by the Kernel simultaneously. The number of processors in each system can be configured.

The system kernel and user processes can claim a processor or the kernel for an atomic operation. During the processing time of that operation, this processor and the kernel can not be used by any other process. However, if the IP Node has more than one processor, a second user process can claim another processor for an atomic operation on its own. Once the atomic operation of a process is finished, the kernel or processor gets released.

Claiming and releasing a processor or the kernel are done by sending a message to the Processor Manager and then waiting for an acknowledgement. If the kernel or a processor is free, then the Processor Manager sends an acknowledgement message back to the requesting module. Should the kernel or all processors be already in use, the event making the claim-request and all subsequent events are put into a waiting queue. Once the kernel or a processor has been released again, the next event from the queue gets to claim the processor or kernel.

With this mechanism, the simulation prevents a parallelism of the event handling system that would not be realistic in a real system. It ensures that two IP datagrams arriving from two network cards are processed subsequently by the kernel, rather than simultaneously. This feature is often not available in other event-based simulation systems. In [4], similar functionality has been modeled in OPNET, but as OPNET is based on Finite State Machines, this approach has not been as extensible with respect to the number of processors. A different approach in [9] presents a simulation suite with inherent modeling of kernel behavior, but does not allow the easy integration of new networking concepts. While the existence of the Processor Manager module is required, its functionality can be turned off optionally in our model.

The Network Module in an IP Node contains the handling of the MAC/PPP-layer and the IP layer. Between those two layers, the IP queues are located, which allow the kernel to store packets coming from or going to the network interfaces. The IP layer maintains one central input queue going into the IP Processing module and a separate output queue for each network interface connected to the IP Processing module. This architecture has been chosen, because it can be often found in real systems.

The *Routing Table* module acts like a database for all network interface and routing information. The routing and network information file is loaded at initialization time from a file for each node.

3.2 IP Processing and protocol Hooks

The *IP Processing* module contains all components of the Internet Protocol. It handles both packets from and to the transport layer, as well as datagrams from and to the IP queues above the network interfaces. The submodules of the IP Processing compound modules are structured in the such a way that the processing of a datagram in the simulation resembles closely the way the Linux kernel handles the processing of IP datagrams.

For instance, the submodule *Routing* determines the output port of a datagram or passes it to *Local Deliver*, if it should be delivered to the upper layers. *Local Deliver* removes the IP header and delivers the transport packet as well as the required control information (such as source and destination address) to the transport layer. The *ICMP* module handles ICMP messages both resulting from internal errors (for instance, if a bit error occurred over the transmission, or the TTL reaches 0) and ICMP queries such as ping and timestamp queries, given to them from the application layer. The *Fragmentation* module allows the fragmentation of one diagram into several fragments on the basis of the MTU of the assigned output network interface. The header of each fragment is equal to that of the original datagram, save for the fragmentation information. After the fragmentation, each fragment is forwarded sequentially to the *Output* module of the corresponding output port.

Each network interface has its own output module. The modules *Pre Routing, Routing, Local Deliver, IP Send* and *IP Output* each contain a hook for extending the protocols behavior with QoS modules. This will detailed be described in section 4.

3.3 Packet Structure

The OMNeT simulation environment offers the encapsulation of packets into other packets. This mechanism allows each layer to en- and decapsulate the packets from upper layers and add all the header and trailer information. The length can be adjusted, based on the information packet passed from the upper layer. That way, TCP and UDP packets, IP datagrams and MAC-frames can be represented in exactly the same fashion as in a real network. However, while the access to the fields yields the same results as in reality, the bit-wise representation of such a packet is not the same.

3.4 Processing of IP Datagrams

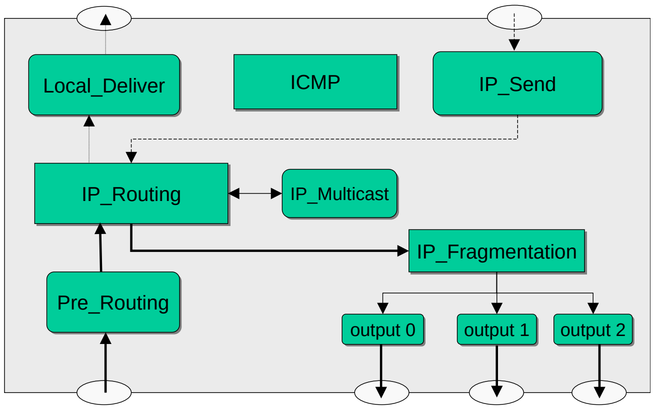

Figure 2 Processing of IP datagrams

The dotted connections on the IP Processing diagram describe the path of IP datagrams to and from the transport layer. The fat connections mark the incoming and outgoing IP datagrams to and from the network interfaces. Every datagram passes through the *Routing* module, which is the core of IP Processing.

1. IP datagrams which are sent originally from the IP Node are created in IP Send, passed on to Routing, then to the Fragmentation and finally to the Output modules. Afterwards, they are inserted into the output queue of the specific network interface.

2. IP datagrams destined at that IP Node arrive from the input queue at Pre-Routing, are passed to Routing, then to Local Deliver, where the IP header is stripped, before the packet is passed to the transport layer.

3. IP datagrams which are simply routed through the IP Node arrive again at Pre-Routing, passed on to Routing, then to the Fragmentation and Output modules.

3.5 Evaluation of the Processor Manager

The impact of the Processor Manager has been tested in the following scenario by generating packet bursts from several end points and measuring the time they take until arrival. The times of the two cases, one with the Processor Manager enables, the other with the Processor Manager disabled, will then be compared . It is expected that the second run will take longer and gives a more realistic time, as the Processor Manager prevents non-realistic parallel processing of multiple packets in the kernel.

The scenario is set up as follows:

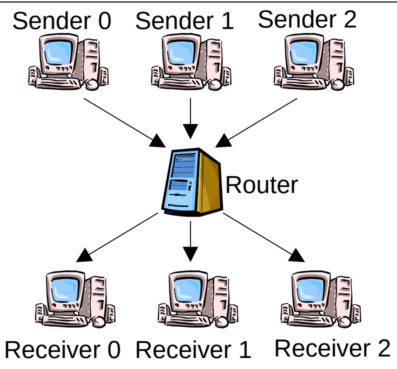

Figure 3 Scenario of the network

Each of the 3 sender nodes sends a burst of 1000 packets to the 3 receiver nodes.

The application layer and network produce no latency. Latency has been induced in the modules *Pre Routing*, *Routing* and the *Output Queue*s on the IP layer of the router. The delay has been set to $1\mu s$ in each module, amounting to a total delay of $3\mu s$ in the router. Two separate experiments have been tested out, one with a bandwidth of 10 Mbit/s and another with a bandwidth of 100 Mbit/s for each link. The end-to-end delay was in both experiments $3\mu s$.

The bottleneck of the scenario is the router. Since the three sender nodes send their datagrams at the same time, it has to handle multiple incoming datagrams simultaneously. In the version with the Processor Manager turned off, one datagram can be processed in Routing while another one can be processed in PreRouting. In the second run, the second datagram is queued until the first has left IP-Processing. Because the network interfaces are usually equipped with their own processors, the handling of frames from the physical network and insertion into the IP input queue can happen in parallel.

The result of the experiment are shown in figure 4.

Line speed:	10 Mbit/s	100 Mbit/s
Time Proc. Man. ON:	80.2021 ms	9.0330 ms
Time Proc. Man. OFF:	80.1951 ms	8.0330 ms
Time difference:	0.0070 ms	1.0000 ms
Relative time diff:	0.008 %	11.11 %

Figure 4 Results of simulation with and without considering parallelism in IP Processing

As expected, the time was in both cases lower for the case in which the Processor Manager was turned on. While the difference in the case of the 10 Mbit/s link is insignificant, the difference for the faster line is significant. The result can be explained that in case of a slow link, the router delay matters little. In the first case, the router was faster to process all arriving datagrams from the three lines than the line was capable of transmitting new ones. In the second case, the router was not fast enough to process all incoming packets as they arrived. As a result, new arriving datagrams had to wait in the input queue until older ones were completely processed. In that case, the parallel processing of datagrams in the three modules PreRouting, Routing and Output Queue allowed a much faster forwarding. However, such behavior is unrealistic for a software IP router, as it cannot process multiple datagrams at the same time. So the case with the Processor Manager enabled gives a more realistic simulation result, which marks after all a difference of 11 %.

4 Building any Quality of Service behavior from elementary QoS models

The goal of the simulation model presented in this paper is to offer elementary QoS modules for the Internet protocol stack, which can be combined and linked together to common QoS elements, like traffic shaper, token bucket, classifier, etc. The suite also offers a variety of queues and scheduling mechanisms like priority queueing, round robin, etc. In the following, these QoS elements are called Behavior Elements. In the next section a more detailed classification is given.

The main principle in building this simulation suite was the possibility to build new QoS behavior quickly from the existing pool of elementary Behavior Elements. This can mostly be done by varying the elementary models and connecting them in a special manner. The following example should motivate the architecture of the proposed model:

A token bucket is a widely used model to meter a certain network flow and to monitor its conformance to Service Level Agreements (SLAs). Traditionally a token bucket meters incoming flows to their conformance. If the negotiated rate is not exceeded, the packets will be forwarded – otherwise they will be discarded. This method of metering a flow is well known, but in some scenarios (i.e. AF PHB in DiffServ networks) the packets should not be discarded. They should either be marked with a lower priority and enqueued in an alternate queue with a lower priority. Or in another scenario, two token buckets should be combined to control the peak rate and the average rate. With existing simulations, new token bucket models, a marker a priority queueing model, etc. have to be developed.

Secondly, it is always a problem to integrate QoS elements into the right position within the protocol stack. For instance, it is important whether a token bucket is working on the IP layer – before the routing has been done – or at the output queue of a certain interface. In the first case all packets forwarded by IP will be considered in the token bucket meter, whereas in the latter case, only the packets leaving on one interface will be metered. In a third case only the packets leaving a host should be considered. It is obvious that in a protocol like IP, a lot of possible places to integrate QoS behavior can be identified.

As a result of this, a fast development of models for simulating new network behavior, as i.e. new QoS behaviors, is very time-consuming with existing simulation models, since they have mostly modeled complete architectures [6, 7]. Each time, new models have to be developed and implemented. The reuse of existing models is not very easy. The presented modular architecture with its elementary QoS models, and the individual linking of them, would solve such problems and allow to build immediately any QoS behavior for an Internet router or host – mostly without implementing new models. The existing pool of elementary QoS behaviors, and the complete implementation of the Internet Protocol, offer on the one hand a real IP behavior and on the other hand the possibility to build and evaluate rapidly new QoS behavior.

In the next few sections the basic architecture of our simulation environment will be presented. First the principle of `Hook`s which are strategic places for including QoS elements into protocol stacks, is explained. Subsequently the five different kinds of `Behavior` types and the rules to concatenate them will be introduced.

Figure 5 will illustrate the architecture by the following example (In appendix A the NED configuration file for the `IP_Forward`-Hook is listed). Three service classes should be distinguished: A Premium class, offering a high priority service with low delay. The flows of the Premium class will be metered by a Leaky Bucket and shaped at the output interface. A second class should offer a better service than Best Effort with a statistical guarantee of bandwidth. This will be achieved by a weighted fair queueing scheduler. The metering will be done by a token bucket. Non conforming packets will not be discarded, but degraded to the Best Effort service, which builds the third service class. The classification to the three service classes is done by a multifield classifier. This example is a possible implementation of the well-known 'Two bit architecture', which is described in details in [3]. To keep the example simple, only the L2-Hooks of interface `eth0` are shown.

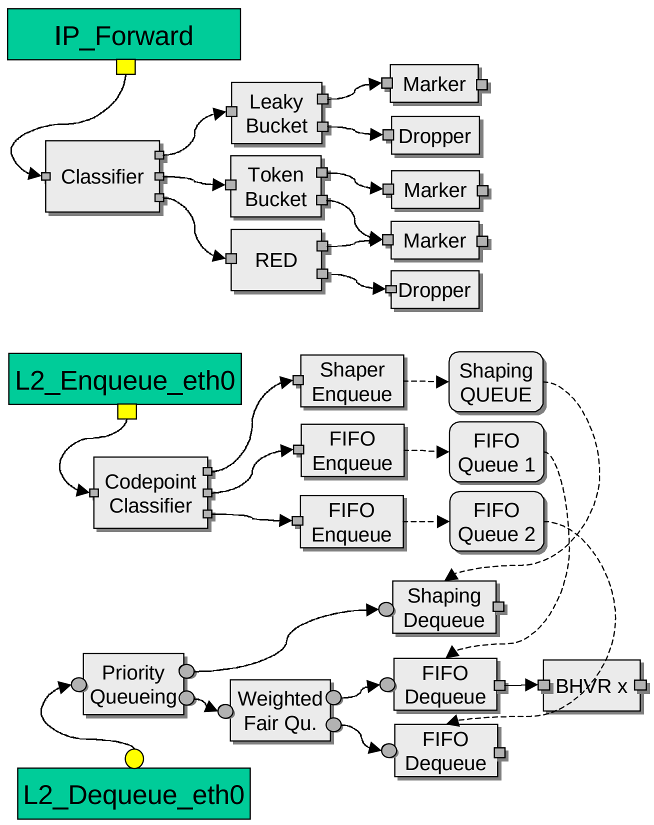

Figure 5 Example configuration of basic QoS Behavior Elements

4.1 Protocol Hooks

When QoS behavior should be introduced into an existing protocol, one basic problem is the place where the protocol is extended with the new Behavior Elements. Regarding the Internet Protocol, five strategic points can be identified. In the following this points will be called *Hooks*. They differ in the set of packets passing the point, i.e. the `Post_Routing`-Hook represents the set of packets leaving the IP node on an interface - whether they have been forwarded, or created from the host:

`Pre_Routing`: All packets arriving on a network interface will pass this hook before routing is processed

`Local_In`: All packets for the upper layers after routing is processed.

`Forward`: All forwarded packets will pass this hook after the routing.

`Local_Out`: Hook for all packets from upper layers, before routing is processed.

`Post_Routing`: Last Hook for all packets (forwarded and from upper layers) leaving on a network interface.

For each network interface, two additional Hooks can be identified: `L2_Enqueue_xx` and `L2_Dequeue_xx` (`xx` is the name of the interface). They are located around the output queue(s) of each network interface card (NIC), which would be the right point to add specific behaviors operating on the outgoing queues, like priority queueing, shaping, etc. (refer Fig. 2).

In this paper we only focus on the Internet Protocol and the underlying layer. But we are also building simulation models for the transport protocols UDP/TCP and Internet applications (HTTP-, FTP-, VoiceCall-traffic models). In these models, Hooks can be added easily at adequate places to integrate QoS behavior.

As described above, a Hook is a place within a protocol where QoS behavior can be added. The Behavior Elements included at such a hook are elementary models offering a certain behavior. They will be described in the following.

4.2 Behavior Elements

A Behavior Element (BE) is comparable to a black box, which offers a specific basic *behavior*. A BE consists of one in-gate, *n* out-gates and a certain processing behavior inside. At the in-gate packets enter the box and receive a certain manipulation inside the module. Dependent on the calculation within the box, a packet leaves on a certain out-gate. Behaviors can be concatenated after each other. Consequently, the treatment a packet receives within a BE decides which way it will proceed and which quality it receives.

Two kinds of gates of Behavior Elements and Hooks can be distinguished: *packet-gates* (abbreviated as □) and *non-packet-gates* (○). The main difference between them is, that between two packet-gates IP-packets are exchanged, and between non-packet-gates only messages to request packets are exchanged. One main rule is, that only gates from the same kind can be connected. The two different types of gates are described more detailed in section 4.3.

There can be five kinds of Behavior Elements distinguished (ref. Fig. 6):

(conventional) Behaviors (BHVR) are elementary QoS elements that operates on IP packets. As shown in Fig. 6, a BHVR has only one in-gate and up to *n* out-gates, where *n* depends on the particular Behavior Element. BHVRs can be interconnected between one another without fulfilling other requirements. Example BHVRs are Token Bucket, Marker, Dropper, Classifier, Random Early Detection (RED) etc.

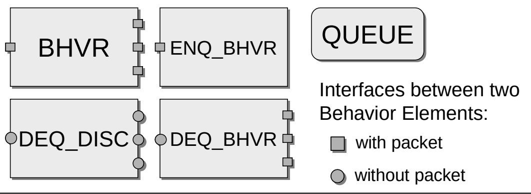

Figure 6 Five different classes of behavior elements

Queue (QUEUE): QUEUEs are well known packet queues. Packets can only be enqueued and dequeued with the two following kind of behaviors. Several types of QUEUEs exist, i.e. Fifo, Shaping, etc.

Enqueue Behavior (ENQ_BHVR) are specialized BHVRs for enqueueing a packet into a queue. The queue is identified by its name and an according Enqueue Behavior should be used. One special characteristic of an Enqueue Behavior is the missing out-gate. Whether the packet is inserted into the queue, or it has to be dropped. ENQ_BHVRs can be connected to out-gates of any BHVR-module. The detailed procedure of exchanging messages and packets between Behavior Elements is described in section 4.3.

Dequeue Behavior (DEQ_BHVR) can be used to dequeue a packet from a certain queue. I.e. a Fifo_Dequeue module removes the first packet from the named queue and sends it to its out-gate. DEQ_BHVR modules can only be connected to an ○-gate of a L2_Dequeue-Hook or Dequeue-Discipline. After a DEQ_BHVR all kinds of BHVRs can be connected to the packet-gate.

Dequeue Discipline (DEQ_DISC): A Dequeue Discipline is a strategy to choose the next Dequeue Behavior for serving a queue. Dequeue Disciplines are playing a very important role in reaching different service classes within a network. Examples for Dequeue Disciplines are Priority Queueing, Weighted Fair Queueing, Round Robin, etc.

4.3 Interactions between Behavior Elements

With the just presented BE types a simple mesh of Behavior Elements can be built and connected to several Hooks. But it is important to understand how a packet traverses this mesh and what other interactions can occur between the Behavior Elements. The following section describes the sequence of events by two example concatenations of Behavior Elements.

Interactions between BHVRs (□-junction): Several Behavior Elements with packet-gates can be arranged into one

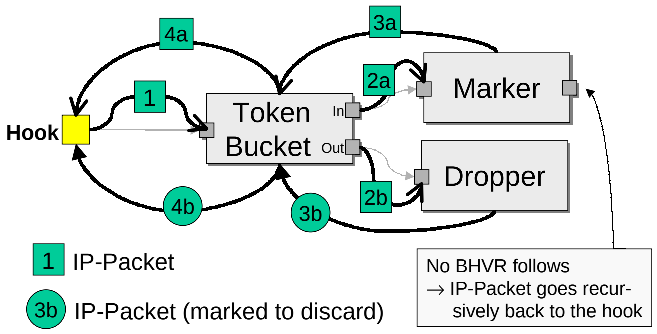
4a
3a
2a
1
Hook
Token Bucket
In
Out
Marker
Dropper
4b
3b
2b
1 IP-Packet
3b IP-Packet (marked to discard)
No BHVR follows
→ IP-Packet goes recursively back to the hook

Figure 7 Interactions on a □-junction between BHVRs

new model to create a new QoS Behavior. At the connections between the □-gates, IP packets are exchanged. Fig. 7 shows an example. The Hook sends an IP packet to the token bucket-BHVR. When the module has completed its operations on the packet it will send it further, if a module is connected on the dedicated port. That means in the example, that a SLA-conform packet (case *a*) will leave on the `In`-gate (In-Profile); otherwise it will leave on the `Out`-gate to the Dropper. If no module is connected to a BHVR on the dedicated port, or the BHVR has no port, the packet will be sent back to the previous module where the packet came from.

One can see, that a packet first traverses a chain of BHVRs and then recursively back to the Hook, where the normal protocol processing will be continued.

This is the normal procedure, but there are two possible exceptions. The first is when a packet reaches an ENQ_BHVR, which will enqueue it into a QUEUE. The second exception is a Dropper that marks the packet for discarding. In both cases, the modules won't send back an IP packet to the hook, but either a *Packet_Enqueued* message or a *Discard_Packet* message. The discarding of a packet will be done in the Hook, because all BHVRs between the Hook and the dropped have to be informed about the loss of packet. I.e. a token bucket has to put back the tokens of this packet into the bucket, because it will not be transmitted.

Interactions at non-packet-gates: On ○-junctions (between DEQ_DISCs or between DEQ_DISC and DEQ_BHVR) no packets will be exchanged. Dequeue Disciplines will first decide which DEQ_BHVR will be asked to dequeue a packet from a queue. This mechanism will be triggered from the Dequeue-Hook sending out a `Request_Packet` message to the first dequeue discipline or directly to a DEQ_BHVR, if no scheduling algorithm is used.

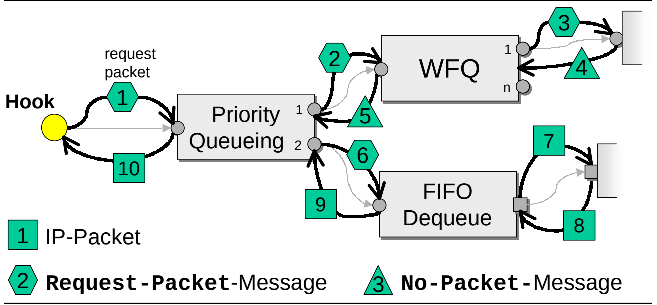

Figure 8 Interactions on ○-junctions between DEQ_DISC and DEQ_BHVR

A DEQ_DISC decides on which of its out-gates the `Packet_Request` will proceed. Any combination of DEQ_DISCs can be built, but finally a DEQ_BHVR has to be called. In Fig. 8, the Priority Queueing module first asks on the gate with the highest priority. The module connected to that gate proceeds with its own scheduling mechanism. In Fig. 8, the WFQ module asks on gate 1.

Each possible chain of DEQ_DISCs has to conclude with a Dequeue Discipline which executes the `Packet_Request` by dequeueing a packet from the QUEUE. On success and if a BHVR is connected, the packet will be sent out on the □-gate of the DEQ_BHVR. This follows the same procedure as described previously about BHVR-meshes.

When the DEQ_BHVR receives back the packet, it is sent recursively back trough the DEQ_DISCs to the Dequeue-Hook.

If no packet can be dequeued, the DEQ_BHVR returns a `No_Packet`-message to the previous DEQ_DISC, indicating that the dequeue-operation failed. The DEQ_DISC can choose then – according to its algorithm – another ○-gate to request a packet or it returns the `No_Packet`-message to its predecessor. On a successful dequeue, the hook starts the transmission of the packet on the interface.

As mentioned above, the dequeueing is triggered by the Dequeue-Hook. Normally, `Packet_Request`-messages will be initiated by the network interface, when it has finished the transmission of the previous packet and is now able to transmit the next packet. But if the interface has been idle for a while, it would not start a new request. Therefore the Enqueue-Hook can initiate a `Packet_Request`, when it just inserted a packet into one of the output queues of the interface. Such an indication only starts a `Packet_Request`, when the NIC is in idle state. This described mechanism is similar to the one in the Linux OS.

5 Conclusion

In this paper, a model for simulating the behavior of the Internet Protocol and a suite for simply creating any Quality of Service mechanism have been presented. The model is a partial aspect of the super model for Internet Protocol layers (TCP, UDP, Applications, etc.) to simulate the behavior of Internet routers and hosts. Special decisions in designing the model, like the consideration of CPUs and parallel processing of packets, have shown this to be an important topic to make realistic simulations of protocol behavior in the Internet.

The creation and evaluation of Quality of Service mechanisms can easily be done by using the elementary QoS models and concatenating them in the correct way. Common models for queue scheduling (priority queueing, weighted fair queueing, round robin, etc.), metering (token and leaky bucket), classifying (multi-header-field, DS-codepoint, etc.) and forming of network traffic are provided.

The simulation suite will be available for the public at [10]. That should everybody offer the possibility to evaluate our models and to increase the pool of available models.

References

[1] Steven Blake, David Black, Mark Carlson, Elwyn Davies, Zheng Wang, and Walter Weiss. An Architecture for Differentiated Services. RFC 2475, December 1998.

[2] Roland Bless and Klaus Wehrle. Evaluation of Differentiated Services using an Implementation under Linux. In *Proceedings of the 7th IFIP Workshop on Quality of Service, London, June 1999*. IEEE, 1999.

[3] Van Jacobson, Kathleen Nichols, and Lixia Zhang. A Two-bit Differentiated Services Architecture for the Internet. RFC 2638, July 1999.

[4] L. K. Lavu and Ravi Malghan. Quality-of-service ip simulation. http://bacon.gmu.edu/qosip, 1997.

[5] S. Lin and N. McKeown. A simulation study of IP switching. *ACM Computer Communication Review*, 27(4):15–24, October 1997. ACM SIGCOMM'97, Sept. 1997.

[6] S. Murphy. Diffserv additions to ns2. http://www.teltec.dcu.ie/∼murphys/ns-work/diffserv/index.html, July 2000.

[7] F. Shallwani et al. Diffserv model for ns2. http://www7.nortel.com/CTL/ns-release6.zip, 2000.

[8] Andrs Varga. OMNeT++ – Discrete Event Simulation System. http://www.hit.bme.hu/phd/vargaa/omnetpp.htm, 2000.

[9] S. Y. Wang and H. T. Kung. A simple methodology for constructing extensible and high-fidelity TCP/IP network simulators. In *Proceedings of the IEEE INFOCOM*, New York, March 1999.

[10] Klaus Wehrle, Verena Kahmann, and Ulrich Kaage. Internet Simulation Suite. http://www.uni-karlsruhe.de/∼omnet, 2001.