Evaluation of Differentiated Services using an Implementation under Linux

Roland Bless, Klaus Wehrle
Institute of Telematics, Universität Karlsruhe (TH)
Zirkel 2, 76128 Karlsruhe, F.R. of Germany
{bless,wehrle}@telematik.informatik.uni-karlsruhe.de

Abstract. Current efforts to provide distinct levels of quality-of-service in the Internet are concentrated on the Differentiated Services (DS) approach. In order to investigate the gain for users of those differentiated services, early experiences with implementations with respect to real applications are needed. Simulation models are often not sufficient if a judgement of the behavior under realistic traffic scenarios is desired. Because implementing new functionality into dedicated router hardware is difficult and time-consuming, we focused on a software implementation for standard PC hardware.

In this paper we present an implementation of Differentiated Services functions for a PC-based router running under the Linux operating system. Two per-hop forwarding behaviors for Assured Service and Premium Service were realized. Components for traffic conditioning such as traffic meter, token bucket, leaky bucket and traffic shaper were implemented as well as an efficient traffic classificator and queueing disciplines. We describe the design and implementation issues of these components, which were validated in detail by measurements. Evaluation of these measurements shows that the proposed forwarding behaviors work well for boundary and interior routers. But, it also becomes apparent that standard applications using short-lived TCP connections cannot always exploit the requested service completely whereas rate-controlled sending applications are able to take full advantage of it. Furthermore, it is planned to release the implementation to the public for research purposes.

1 Introduction

There is an increasing demand for different services other than the traditional best-effort delivery in the Internet. In order to meet the highly varying requirements of users and forthcoming applications, Internet service providers (ISPs) want to offer alternative levels of service to their customers. Additionally, competition between service providers is forced by providing various services at different pricing.

But, the actual benefit to users of Differentiated Services is not known a priori. Thus, early experiences with such new services are required. Most simulation scenarios don't have real application data as input, especially those of interactive applications. An implementation of new functionality for supplying Differentiated Services within 'real' routers, i.e., specialized hardware dedicated to routing functions, is often not possible, because development is in the hands of the hardware manufacturer. Therefore, we used a standard personal computer (PC) as a router to implement and evaluate new functionality. The router consists of a standard PC with a single CPU and several network adapters connected by the usual PCI or ISA bus. Naturally, this architecture has some performance limitations, but on the other hand it is very flexible, because routing functionality is completely done in software. Consequently, new mechanisms can be employed, tested and modified very fast. It can serve to build a first platform for testing new services in small testbeds using real applications.

2 Differentiated Services

After specification of an Integrated Services architecture for the Internet [FeHu98], some doubts about its scalability arose, because per flow state information has to be kept in every router on the path from sender to receiver(s). It is argued that identification of every single flow and management of very much state information would decrease performance dramatically, especially if considering heavily loaded routers in the core network of today's Internet. Thus, a new Internet working group was founded with the goal to develop a scalable architecture for supporting differentiated services in the Internet. Currently, there is much work in progress and the working group specified some draft documents of which two are already adopted as RFCs [BBBN98, BBCD+98] describing a basic architecture for Differentiated Services (DS).

Scalability is achieved by reducing complexity and state information in routers. This is mainly attained by aggregation of traffic classification state. Packets are classified and marked for a corresponding service category. Complex traffic conditioning functions such as classification, policing and shaping are only required at network boundaries or hosts. Consequently, nodes in the inner network only need to distinguish between different per-hop forwarding behaviors by looking at a mark in the packet header, and they need to provide accordant forwarding mechanisms. This mark is constituted by a part of the DS field (which replaces the IPv4 TOS octet or IPv6 Traffic Class octet), which is the so-called Differentiated Services codepoint (DSCP) [BBBN98]. A codepoint value corresponds to a particular per-hop forwarding behavior (PHB) in a *DS domain*, which is a contiguous set of nodes that support Differentiated Services as defined in the DS architecture [BBCD+98]. Hence, end-to-end services can be constructed by a sequence of (possibly identical) per-hop forwarding behaviors supplied by all DS domains that lie on the corresponding path.

A DS domain comprises *boundary nodes* that connect to other DS domains or non-DS domains and *interior nodes* which connect only boundary nodes or other interior nodes within this domain. Boundary nodes are ingress and egress points for traffic, and, consequently perform more complex traffic conditioning functions such as policing or shaping. Interior nodes only distinguish between different *behavior aggregates* which denote a collection of packets with the same DSCP crossing a link in a particular direction. Thus, there is no need to maintain state information per 'microflow' in interior nodes. In this context a *microflow* denotes a single instance of an application-to-application flow of packets which is identified by source address, source port, destination address, destination port and protocol id.

Our first implementation is mainly based on services (and their corresponding per-hop-behaviors) that were proposed in [JaNZ97]. Thus, we also distinguish between *first-hop routers*, *border routers* and *interior routers*. First-hop routers typically police incoming traffic from non-DS-aware end-systems and mark packets in accordance to a negotiated traffic profile. In addition, first-hop routers will often perform traffic shaping functions. Border routers typically connect different DS domains (often under administration of different ISPs) to each other, so they also perform policing for incoming traffic from other domains, but in their case, only aggregates are considered. Thus, classification is much simpler and mainly based on input link interface and DS codepoint. Border routers will also often shape outgoing traffic to enforce conformance to negotiated service level and traffic conditioning agreements with other service providers of DS domains. First-hop router and border router are both denoted as DS boundary nodes in the sense of the DS architecture [BBCD+98]. Consequently, interior routers are denoted as interior nodes in the same context.

The Premium Service (PS) described in [JaNZ97] and its corresponding Expedited Forwarding PHB in [JaNP99] respectively, provides a guaranteed bandwidth, low delay and low loss service that shows the same characteristics as a 'virtual leased line'. This is achieved by keeping PS queues very small or almost empty, which in turn can only be accomplished by guaranteeing that the maximum arrival rate of an aggregate is less than that aggregate's minimum departure rate. Thus, admission control and policing is needed with respect to the guaranteed rate as a required configuration parameter.

Assured Service (AS) as proposed in [JaNZ97] permits a statistical guaranteed rate only. It permits to use additional available capacity while providing a base rate. Packets that exceed the negotiated rate are either marked as best-effort traffic or dropped. The overall dropping probability of AS packets is considerably lower than for best-effort packets. Thus, Assured Service allows some amount of bursts that are also forwarded and even aggregated as bursts. Consequently, the firmness of its guarantee depends on how well links are provisioned for bursts, because packets that are marked as best-effort may be later dropped if congestion occurs. Furthermore, implementations of AS must assure that packets of microflows are not reordered. Moreover, delay characteristics would not be better than that of normal best-effort traffic. The current proposal for an Assured Forwarding PHB group [BHWW99] comprises the Assured Service as a special case and offers more degrees of freedom.

3 Implementation under Linux

A good basis for implementing a router with DS functionality is the Linux operating system. It runs on standard PC hardware and source code of the kernel is widely available. Additionally, since development kernel version 2.1 it supports a variety of queueing disciplines for output queues of network devices. Moreover, all functions for routing are already provided.

To get a deeper insight into our concrete realization of Differentiated Services mechanisms, understanding how Linux handles basic network functions is important. Hence, a short overview of the standard Linux network implementation is given next. Afterwards, the implementation of Assured Service and Premium Service is presented.

3.1 Network implementation

To give a short overview of the Linux IP network implementation, the course of an IP-packet through the system is described first (cf. figure 1).

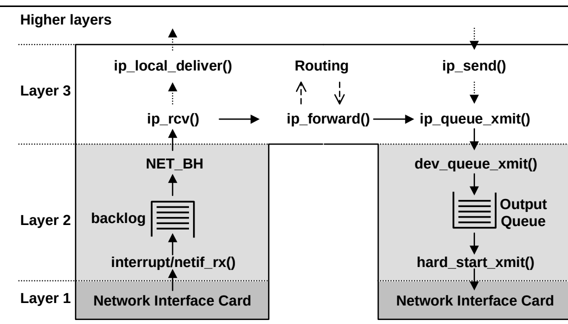


Figure 1 The course of a packet through the system

After a packet is received by a network interface card, a hardware interrupt is triggered. As a consequence, an interrupt handling routine (`ei_interrupt()`) is invoked and determines the type of interrupt. For interrupts caused by incoming packets a further handling routine is called (`ei_receive()`) which simply copies the packet from the network card into an internal socket buffer structure (`sk_buff`) and calls a procedure named `netif_rx()`. The latter queues the packet (represented by a socket buffer structure) into a central queue (`backlog`) consisting of all packets that arrived on any network adapter of the system. The first time-critical part of the interrupt routine, called 'top-half', is finished at this time.

The necessary second part, called 'bottom-half', is handled by the network bottom-half routine (`NET_BH`) which is regularly invoked by the kernel scheduler. At first, this procedure checks whether there are still packets waiting for transmission in an output queue of any network adapter. If there are any packets waiting they are processed for a limited period. Subsequently, `NET_BH` proceeds with the next packet from the backlog queue and determines the appropriate protocol to handle the packet which is in our case the Internet Protocol IP. `ip_rcv()` checks for correctness of the IP header and then processes any existing options. It also reassembles the original IP packet from fragments if necessary and if the packet has reached its final destination. In the latter case, the packet is delivered locally, otherwise it is routed and forwarded towards its destination. `ip_forward()` tries to find the right network adapter this packet is forwarded to next by use of a routing table. If there is a valid entry in the routing table `ip_queue_xmit()` is subsequently invoked, performing some final operations such as decrementing time-to-live values and recalculating IP header checksums. `dev_queue_xmit()` queues the packet into the output queue of the corresponding network device. At this point a special queueing discipline can be invoked. One can imagine that the output queue is probably realized by some sophisticated queueing discipline that manages several virtual queues in order to treat packets in different ways (cf. fig. 2). Thus, each queueing discipline constitutes one output queue for a device that is not necessarily served in FIFO order. Within a queueing discipline, transfer of a packet onto network media is initiated by calling `hard_start_xmit()`, which instructs the network device to send the packet.

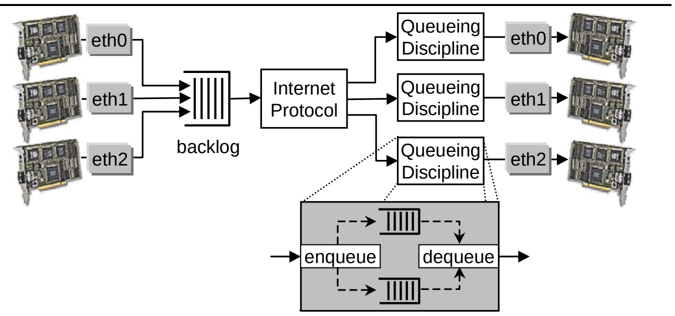


Figure 2 Routing with different queuing disciplines

The Linux kernel already contains various queueing disciplines such as Class Based Queueing, Weighted Fair Queueing or Random Early Detection. We found that the existing RED implementation contained some errors and was not precise enough for our purposes. Thus, implementation of the RIO algorithm was done from scratch.

The problem to put configuration data (e.g., traffic profiles) for Differentiated Services into the kernel memory easily was solved by using device drivers. Naturally, there is no real physical device that is controlled by those drivers, but the new DS functions are driven by them. Thus, one can configure new parameters for services by simply writing data structures into the corresponding device. Otherwise, new kernel system calls had to be introduced leading to more complexity. Furthermore, some status data of the modules is

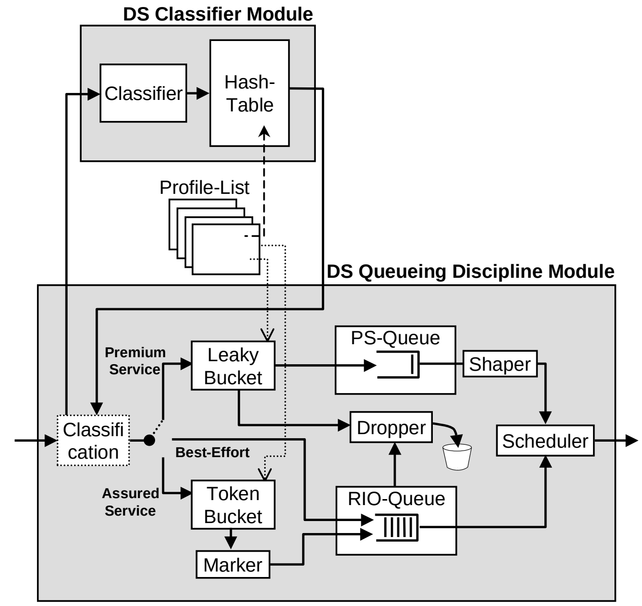


Figure 3 Implementation architecture

supplied by files in the `/proc` filesystem and can be obtained by simply reading these files.

Another design decision was to use kernel modules for implementation of DS functionality. Kernel modules need not to be present all the time in the kernel, so that the kernel can run without them if they are not actually used. Particularly, instead of recompiling the whole kernel and restarting the system everytime a part of the module's code was changed, one can simply reload the newly coded module. This shortens development time dramatically.

The implementation comprises two kernel modules (see figure 3): a Differentiated Services Classifier (DSC) module and a Differentiated Services Queueing Discipline (DSQD) module. The DSC module classifies packets based on a value of a combination of one or more header fields (Multi-Field Classifier) and assigns the corresponding profiles to those packets. In addition, for border routers classification can be restricted to evaluation of a combination of input port and DS codepoint. The DSQD module includes the following components for traffic conditioning: a traffic meter realized by a token bucket and a leaky bucket, a marker, a traffic shaper and a dropper. Moreover, queueing disciplines and scheduling mechanisms for both forwarding behaviors are located in this module. In the following, realization of both modules is described more detailed.

3.2 Differentiated Services Classifier Module

This module is responsible to classify any incoming packets in first-hop-routers or border routers and to find the corresponding traffic profiles for them. Multi-field classification is used in first-hop-routers and is restricted in the current implementation to a combination of the fields source end-system address, source port number, destination end-system address, destination port number and protocol id. Which combination of header fields is to be considered significant for comparison is determined by a flag field, thus giving flexibility to identify all flows to a certain destination port (e.g., port 80 for WWW-traffic) or even individual microflows. Classification in border routers is done on identification of the incoming network link and DS codepoint in order to allow traffic policing according to existing service level contracts with other providers.

```
struct profile_t {
 unsigned int id;
 unsigned long timeout;
 struct ds_addr_t addr;
 unsigned char type;
 union {
  struct as_data_t as;
  struct ps_data_t ps;
 } service;
}
```

(a) profile data structure

```
struct ds_addr_t {
 unsigned char flags;
 struct in_addr s_addr;
 struct in_addr d_addr;
 unsigned short s_port;
 unsigned short d_port;
 unsigned char protocol;
}
```

(b) classifier data structure

Figure 4 An internal DS profile structure

Profile data (see fig. 4(a)) comprises information about traffic parameters (e.g., token rate and token bucket size) for a specific service (in the union `service`). In our implementation, classification information is also stored within the corresponding profile (member `addr`). Profiles are identified by a unique identification number (member `id`) and also contain the type of service they belong to (member `type`). All profiles for Premium and Assured services are organized in a linked list. For packets belonging to the traditional best-effort service no profile data is needed.

An efficient implementation of this classificator is crucial for the performance of the whole system, because every incoming packet has to be classified except in interior routers. Therefore, profiles are stored in the kernel memory to shorten access time, but because kernel memory cannot be paged out, data structures have to be compact. A hash table is used to find a profile related to an incoming packet. If there is no entry in the hash table for a packet, the list of active profiles is searched. On an unsuccessful search the packet only gets best-effort forwarding as default behavior. In any case a new hash entry is created either containing a reference to the profile or a null pointer indicating best-effort traffic. This guarantees that best-effort traffic is not treated disadvantageously. In order to reduce memory usage every entry in the hash table expires if no packets related to it were received within a certain period, which is configurable by setting a value in the variable `timeout` of the `profile_t` structure.

A reference to the profile is stored in the `sk_buff` structure, because it must be available before a packet is queued and if a packet is removed from the queue. A reference for best-effort packets is constituted by a null pointer.

3.3 Differentiated Services Queueing Discipline Module

The queueing discipline module contains the main functions for traffic conditioning and forwarding (cf. figure 3) for Assured Service, Premium Service and traditional best-effort service. Incoming traffic is classified either by the DSC module (in boundary routers) or simply by the DS codepoint (in interior routers). If we look at boundary routers, incoming traffic is checked against traffic profiles by a token bucket for AS flows and by a leaky bucket for PS flows. In interior routers traffic is directly put into one of both queues without policing. Non-conforming PS packets are dropped, whereas non-conforming AS packets are marked as out-of-profile (in our implementation as best-effort). The PS queueing discipline is a simple FIFO (first-in-first-out) strategy, whereas the AS queue is served according to a RIO algorithm (Random Early Detection with distinction of in-profile and out-of-profile packets) [FlJa93, JaNZ97]. The RED algorithm is used for in-profile and out-of-profile packets with different parameter values, resulting in different drop probabilities. The parameter values for out-of-profile packets are also applied to best-effort traffic. A simple priority scheduler is used between PS queue and AS queue, servicing all packets waiting in the PS queue before any packet from the AS queue. Additionally, traffic from the PS queue is shaped in first-hop or border routers.

Implementation of token bucket, RIO queueing discipline, leaky bucket and traffic shaper is now discussed in depth by looking at implementation of Assured Service and Premium Service.

3.3.1 Assured Service Implementation

To test conformance of observed AS traffic with its specification in the traffic conditioning agreement, a token bucket is used [FeHu98]. A token bucket can describe characteristics of a traffic source, which is allowed to emit bursts of packets to some extent. A token bucket is filled with tokens at a constant rate and has a fixed size for holding these tokens, so the bucket cannot contain more tokens than specified by this size. Every time a packet arrives, it consumes a corresponding number of tokens. If there are enough tokens available in the bucket, the packet is allowed to pass by and the content of the bucket is reduced by the number of used tokens. Otherwise the packet is considered to be non-compliant with the traffic profile and can be discarded or marked accordingly. The service-specific traffic profile data is depicted in figure 5(a). The first two parameters specify token rate (in bits per second) and token bucket size (in bytes) accordingly.

We decided to use a byte-oriented token bucket (one token corresponds to one byte) to achieve a high accuracy and common base for policing. Considering only IP packets as units is not precise enough, because their length can vary, consequently yielding varying throughput (measured in bits or bytes per second) even at constant packet rates.

Some limitations of the hardware must be considered before implementing a byte-oriented token bucket. The internal

```
struct as_data_t {
 ulong rate;
 ulong bucket_size;
 ulong bucket;
 ulong packets;
 ulong cycles_per_byte;
 CPU_STAMP last_arvl;
};
```

(a) AS profile structure

```
struct ps_data_t {
 ulong rate;
 ulong bucket_size;
 ulong pq_bytes;
 ulong packets;
 ulong packets_enq;
 ulong packets_deq;
 ulong packets_dropped;
 ulong cycles_per_byte;
 CPU_STAMP next_kick;
};
```

(b) PS profile structure

Figure 5 Internal AS and PS specific profile structures

system clock is incremented every 10 ms by a timer interrupt, and, for that reason not accurate enough in order to measure time between arrival of two consecutive packets even at 'slow' speeds of a 10 Mbit/s Ethernet (within 10 ms more than 8 packets can arrive if sent at full rate). The solution is to use a special processor register provided by most modern CPUs that is incremented with every processor cycle for improvement of the system's clock precision. For example a processor running at 200 MHz gives a resolution of 5 ns. But the achievable accuracy is reduced, because generating a timestamp typically consumes more than 1 cycle (in our case 10 cycles) due to storing the register contents into main memory which is not clocked at the same rate as the CPU.

The token bucket is realized as follows: the packet arrival time is taken immediately when it is queued into the backlog by the interrupt handling routine `netif_rx`. The amount of tokens that were filled into the bucket within the period between arrival of the previous and this packet is calculated by building the difference between arrival time of the last packet belonging to this profile (`last_arvl`) and arrival time of this packet multiplied by the token rate (`(netif_rx - last_arvl)/cycles_per_byte` with `cycles_per_byte:=` $8 \cdot$`cpu_clock/rate`). If addition of this amount to the current content (`bucket`) lets the bucket overflow, the content is set to `bucket_size`. In case the bucket holds enough tokens corresponding to actual packet size, the bucket content is reduced by this amount and the packet is passed as in-profile. Otherwise the packet is marked as out-of-profile. In any case, `last_arvl` is updated.

All packets belonging to Assured Service (in-profile as well as out-of-profile packets) and best-effort service are queued into one RIO-queue. Because the RED algorithm calculates the drop probability in dependence on average queue length q_{avg}, this value has to be calculated and updated. A new packet is dropped with increasing probability (up to $p_{\max}$) if q_{avg} is between lower threshold $q_{\min}$ and upper threshold $q_{\max}$. Below $q_{\min}$ the packet is never dropped and above $q_{\max}$ it is always dropped. The RIO algorithm simply keeps track of separate queue lengths for in-profile (only in-profile packets are considered) and out-of-profile packets (in-profile, out-of-profile and best-effort packets are considered) as well as different thresholds $q_{\min}, q_{\max}$ and drop probabilities $p_{\max}$. In order to avoid floating point arithmetic, some simplifications are used. q_{avg} is periodically updated by $q_{\text{avg}} := q_{\text{avg}} + w_q \cdot (q_{\text{cur}} - q_{\text{avg}})$ with current queue length q_{cur} and weight w_q. Instead of multiplying with w_q we divide by the integer `1_Wq` which is set to $\lfloor 1/w_q \rfloor$. If q_{avg} is updated only at times when a packet arrives one has to take the period into account during which no packet arrived. In order to avoid complexity introduced by this extra calculation such as presented in [FlJa93], we simply update q_{avg} periodically (256 times a second) by using a timer interrupt. Similarly, checking for the condition $\texttt{random} \cdot \texttt{1\_Pmax} \leq q_{\text{avg}} - q_{\min}$ requires only integer arithmetic (without division) to decide whether the current packet has to be dropped if $q_{\min} \leq q_{\text{avg}} \leq q_{\max}$ holds. `random` is a pseudo-random integer value between 0 and $q_{\max} - q_{\min}$, and, `1_Pmax` is set to $\lfloor 1/p_{\max} \rfloor$. Packets remain in the RIO-queue until they are removed for transmission by the `NET_BH` routine that is regularly called by the kernel scheduler.

3.3.2 Premium Service Implementation

First of all, packets belonging to PS flows are checked for conformance according to their related profile (except in interior routers). This is accomplished by using a byte-oriented leaky bucket [FeHu98] that also shapes outgoing traffic. At arrival of a packet a check is performed whether it still 'fits' into the bucket. This can be accomplished by evaluating the condition $\texttt{packetlen} + \texttt{pq\_bytes} \leq \texttt{bucket\_size}$, with `packetlen` denoting the length of the current packet in bytes, `pq_bytes` representing the current amount of bytes held in the leaky bucket, and, `bucket_size` an upper bound for its contents (the latter two are contained in the PS profile structure that is depicted in fig. 5(b)). In case there is not enough space left, the packet is discarded. Otherwise transmission of the packet is delayed if necessary (traffic shaping is applied). The earliest instant for transmission of this packet is stored in the variable `next_kick`. As described above, arrival time of packets is recorded in the variable `netif_rx` of the `sk_buff` structure. If the packet arrived too early ($\texttt{netif\_rx} < \texttt{next\_kick}$) its theoretical instant of transmission is stored into the variable `ps_stamp` of the `sk_buff` structure and it is queued into the PS queue. Packets in this queue are sorted by their theoretical moment of transmission (`ps_stamp`), so new packets are simply inserted from the end of the queue until they fulfill this sorting condition. In case a packet arrived too 'late' ($\texttt{netif\_rx} \geq \texttt{next\_kick}$) it is immediately scheduled for transmission by setting `ps_stamp` to `netif_rx`. In order to avoid exceeding the agreed rate contained in the profile (the first variable `rate` [bits per second] of the PS profile structure), transmission of the next packet belonging to this profile is not allowed until the time is elapsed that would be necessary if the current packet will be transmitted at speed `rate`. Thus, conforming transmission of the next packet can be started earliest at `next_kick` := `ps_stamp` + `packetlen` · `cycles_per_byte`.

Transmission of packets in this queue is initiated by a timer interrupt, because they have to be sent at the right time `ps_stamp` for generating shaped traffic, and, transmission initiated by `NET_BH` occurs too indeterministically. The available default resolution of this system timer (100 Hz on standard PCs, 1024 Hz on workstations with an Alpha CPU) is not accurate enough for traffic shaping. But one can increase this frequency up to 8192 Hz. We use a value of 4096 Hz to achieve a resolution of 244 μs. Although the number of timer interrupts per second is now much higher than before, a degradation of system performance could not be observed at our systems. If a timer interrupt calls `premium_send()`, `ps_stamp` of the first packet in the queue is checked. Transmission of the packet is started immediately by calling `hard_start_xmit` if the instant is reached or already passed. In case the network adapter is currently busy transmitting another packet, `hard_start_xmit` is called until forwarding is possible. If there are further packets in the PS queue with an elapsed `ps_stamp`, they are also transmitted.

4 Evaluation

In order to evaluate effective gain for users of Differentiated Services we accomplished some tests. A first series of tests had the objective to validate our implementation of traffic conditioning and forwarding mechanisms. Subsequently, a second series of tests focused examination of characteristics of both services.

4.1 Test configuration

We used the following configuration that is shown in figure 6. The testbed comprises a PC as DS router (Pentium CPU at 200 MHz, 64 MB Ram, 3 network cards 3Com 3c509 Etherlink III) two PCs as sender (end-systems A and B both Pentium CPU at 90 MHz) and one PC as receiver (end-system C, Pentium CPU at 133 MHz). The network was completely build of separate 10Base2 Ethernet segments, because we currently had no 10BaseT network cards available for this testbed. Using 10Base2 simplex Ethernet instead of duplex 10BaseT also induced a problem for tests using TCP: because acknowledgements have to be sent back to the sender, collisions would occur and disturb our measurements. Thus, we used a separate 10BaseT network (dashed line in fig. 6) for all traffic that is sent back from C to A, especially TCP acknowledgements.

I/O performance and CPU power of the router are crucial for a successful operation, because if the PC is too slow, protocol processing for a packet is never finished before a new packet arrives. As a consequence, the backlog is filled up to its maximum and all queued packets are discarded until the backlog is empty again. Thus, implemented mechanisms such as the RIO algorithm are never used in this case, because packets are dropped already earlier. Nevertheless, a PC with a Pentium CPU running at 200 MHz is sufficient to route incoming traffic of 20 Mbit/s at least.

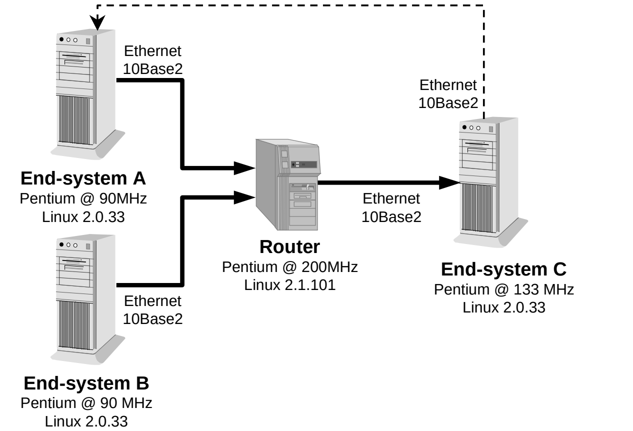


Figure 6 Configuration of the test network

Sender A and B generate accurate shaped UDP traffic flows for a certain period. All packets have a constant UDP payload length of 1000 bytes. Every packet gets a timestamp that contains its correct instant for transmission. To achieve high accuracy, the processor cycle counter register (PCC register) is used for all timing purposes. Packets for distinct flows are queued for transmission into a sending queue. An endless loop checks whether the instant of the first packet in this queue has already passed (causing immediate transmission of this packet) or whether the specified execution period of this test has already expired. Before a packet is sent, a timestamp containing the current time is put into the UDP payload of the packet. The router also writes timestamps of certain instants into transmitted packets (only for packets with a specific port number): when a packet is received (`netif_rx`), when it is dequeued from the backlog (`net_bh`), when it enters the DS queueing discipline before classification (`before_dsc`), after classification (`after_dsc`), before packets are enqueued into the PS queue or RIO queue (`dsqd_enqueue_ps`,`dsqd_enqueue_rio`) and when they are dequeued (`dsqd_dequeue_ps`, `dsqd_dequeue_rio`). Additionally, for Premium Service packets the aforementioned theoretical packet send-off time `ps_stamp` is recorded as well as the instant `dsqd_premium_sent` when a packet is actually delivered to the network adapter. In order to influence the router as little as possible, recording of those timestamps is done at the receiver. Otherwise, accesses to the file system would disturb our measurements.

The receiver C also records the arrival time of a packet, again using its PCC register. All timestamps included in the packet are extracted at the receiver and dumped into a file, that can be evaluated later with a standard spreadsheet program.

All tests were executed for a fixed duration: tests with UDP packets took 10 s whereas tests with TCP lasted 100 s. Furthermore, we used fixed packet sizes in every experiment.

4.2 Tests and Results

We did all tests, if applicable, for every type of router: first-hop router, border router and interior router. Thus, we can conclude how services are provided end-to-end. First of all, token bucket, leaky bucket and the traffic shaper were validated by tests.

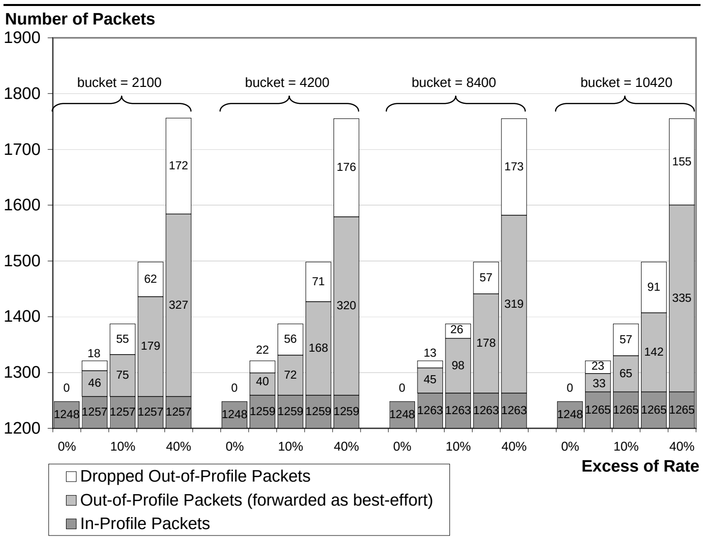


Figure 7 Results for validation of token bucket behavior

The token bucket should determine which packets of an Assured Service flow are conforming and which are not. For this test sender A sent 1 Mbit/s Assured Service traffic, 1 Mbit/s Premium Service traffic and 5 Mbit/s Best-Effort traffic to Receiver C. Sender B simply sent 1 Mbit/s Premium Service and 5 Mbit/s best-effort traffic to C. The actual sent rate for Assured Service was varied to excess the allowed rate in the profile about 0%, 5%, 10%, 20% and 40%. Each test was repeated with different bucket sizes. The result is shown in figure 7 (note that the Y-axis starts at 1200): the part of in-profile packets is colored dark-gray, the part of out-of-profile packets gray and the portion of dropped packets white. In every case the rate of in-profile packets was strictly bounded independent of bucket size or rate excess. Flows that exceeded their profile rate transmitted a fixed amount of additional packets, because they exploited their allowed portion of bursty traffic. 60%-70% of all out-of-profile packets still gets through to the receiver. This portion depends heavily on the current traffic situation in the router. The probability that an out-of-profile packet is not dropped on a specific interface of a route can be estimated by $(\mu - \lambda_{\mathrm{PS}} - \lambda_{\mathrm{In}})/(\lambda_{\mathrm{Out}} + \lambda_{\mathrm{BE}})$ with μ denoting the total available netto transmission rate of the physical medium and λ_{X} denoting the rate of service X (PS=Premium Service, BE=Best-Effort, In=Assured Service In-Profile traffic, Out=Assured Service Out-Profile traffic) loaded on this interface. Additional actions must be provided to prevent unfairness caused by intentional excess of the contracted AS rate.

The same tests were applied to the leaky bucket that is used as a meter for Premium Service. Consequently, sender A and B sended each 1 Mbit/s Assured Service traffic and 5 Mbit/s best-effort traffic. In addition, Sender A transmitted 1 Mbit/s Premium Service traffic which was subsequently increased about 5%, 10%, 20% and 40%. The test results confirmed the effective restriction to the allowed rate for Premium Service traffic of 1 Mbit/s.

The next presented series of measurements is based on the following configuration: End-system A sends 5 Mbit/s best-effort traffic and 10 Premium Service flows (5·100 kbit/s, 4·25 kbit/s, 1·400 kbit/s). Sender B also sended 5 Mbit/s best-effort traffic, 6 Premium Service flows (5·100 kbit/s and 1·500 kbit/s).

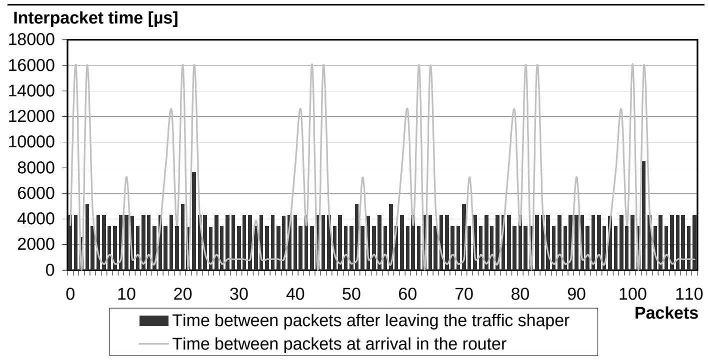


Figure 8 Validation of the traffic shaper

Figure 8 shows interpacket times (i.e., the time distance between two consecutive packets passing the same measurement point) of incoming PS traffic at the router (timestamp `netif_rx`) and of received traffic at end-system C. The traffic is shaped to an aggregated output rate of 2 Mbit/s for Premium Service. The observed rate-jitter (variation in interpacket time) has two sources. First, if a packet has reached its theoretical time for transmission (`ps_stamp`) the network adapter can still be busy with transmission of previous frames. Because some network cards provide a transmission buffer for caching several frames, we reduced the size of this buffer to two packets, so only one frame can be currently in transmission if a new frame is stored into the buffer for transmission. Thus, the interpacket time can vary by one packetizing time (i.e., the time a packet needs to be put completely onto the physical medium). A second source for extra delay is inaccuracy of the timer. Although its resolution was increased to 244 μs we get an additional inexactness of this resolution in the worst case, because within this period transmission of a new frame from other service classes could have been just initiated. Furthermore, if packets have varying length (which is the usual case) the jitter would be increased. Finally, the two peaks at packets 22 and 102 are caused by senders which do not fully exploit their negotiated traffic rate at these instances.

4.2.1 Evaluation of Assured Service

Although the Assured Service offers only statistical guarantees, a minimum throughput should be guaranteed if the sender does not exceed his specified traffic profile. Obviously, parameters of the RIO algorithm determine the amount of in-profile AS traffic that is forwarded unchanged, i.e., not re-marked as out-of-profile or dropped. In-profile packets are not dropped by the RIO algorithm if $q_{\mathrm{In,avg}} < q_{\mathrm{In,min}}$ holds. One can assume that the average queue length for in-profile packets ($q_{\mathrm{In,avg}}$) is proportional to $\lambda_{\mathrm{In}}/(\mu - \lambda_{\mathrm{PS}})$. Thus, we have $q_{\mathrm{In,avg}} \leq \lambda_{\mathrm{In}}/(\mu - \lambda_{\mathrm{PS}}) \cdot q_{\mathrm{Out,max}}$ which constitutes an upper bound for in-profile packets in the RIO queue. Consequently, if we set the lower threshold $q_{\mathrm{In,min}}$ at least to $(\lambda_{\mathrm{In}}/\mu - \lambda_{\mathrm{PS}}) \cdot q_{\mathrm{Out,max}}$ then no in-profile packets are dropped. Now we have a relation between the minimum threshold $q_{\mathrm{In,min}}$ and λ_{In} which defines the minimum guaranteed throughput for Assured Service. This formula was also empirically verified. Sender B generated 9.7 Mbit/s best-effort traffic and sender A varied rates of Premium Service and Assured Service traffic as shown in fig. 9. It shows calculated (light gray) and measured values (dark gray) of $q_{\mathrm{In,avg}}$.

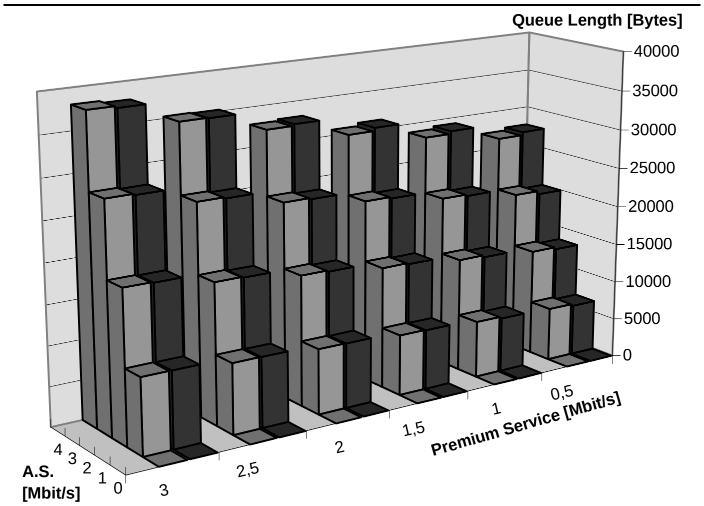


Figure 9 Validation of queue length estimation for Assured Service

Although Assured Service gives no guarantees related to delay, we want to have a closer look on it. Two main influences can be distinguished: the current queue length of the RIO queue (which is $q_{\mathrm{Out,max}}$ in the worst case) and current portion of PS traffic (see fig. 10. Thus, average delay of an AS packet in a fully loaded router can be estimated by $q_{\mathrm{Out,max}}/(\mu - \lambda_{\mathrm{PS}})$. Consequently, this justifies again restricting the share of PS traffic to a small portion of overall link bandwidth.

Because queue length also varies with best-effort traffic, jitter of Assured Service traffic cannot be determined in advance. Furthermore, AS traffic tends to become more bursty because of the RIO queue, which leads to higher drop probability in subsequent routers. Consequently, it is recommended to also shape AS traffic at boundary routers.

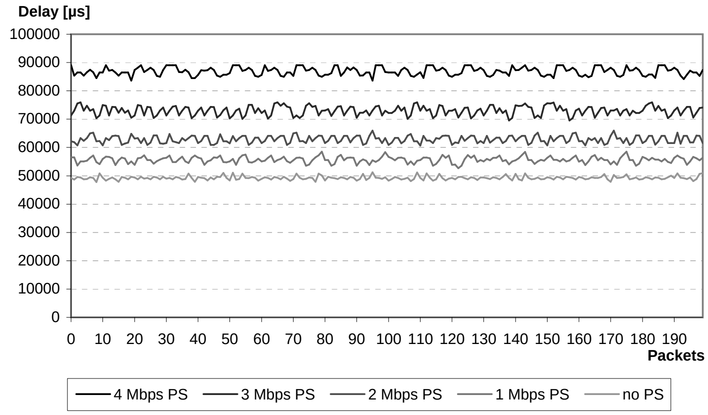


Figure 10 Delay of Assured Service packets in dependence on Premium Service traffic

4.2.2 Evaluation of Premium Service

Premium Service should offer a low loss, low latency, low jitter and assured bandwidth end-to-end behavior [JaNP99]. We executed some tests in order to validate those characteristics.

According to loss, we did not observe any packet loss if the sender generated traffic conforming to its profile. It is crucial that admission control is applied for Premium Service traffic in order to guarantee low loss, low latency and assured-bandwidth. Additionally, policing at boundaries of the DS domain is required to prevent any excess of the negotiated rate. Thus, non-conforming traffic can only evolve from packet clumping caused by aggregation of (possibly already aggregated) traffic flows in interior routers. Consequently, one must have a closer look at evolution of jitter in interior routers.

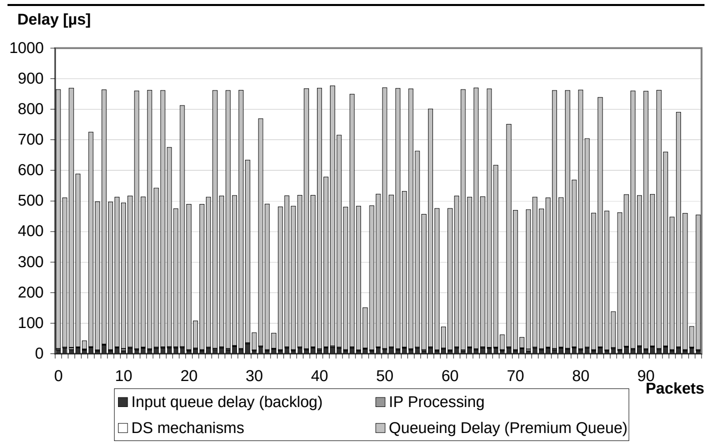
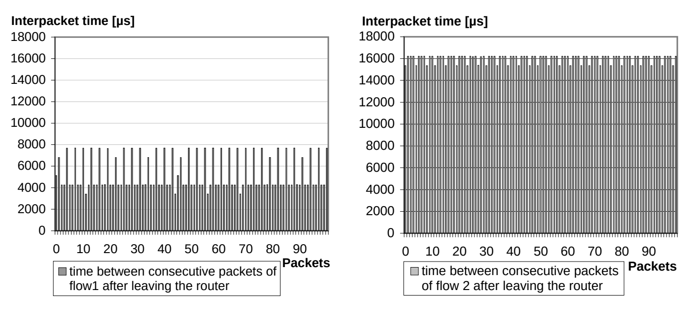


Figure 11 Delay of PS packets in an interior router

Delay of Premium Service packets in interior routers is mainly determined by waiting for an already started transmission of a frame to end (cf. fig. 11). Thus, even in interior routers where no traffic shaping is applied we often get a rate-jitter of one packetizing time (852.8 μs for an Ethernet frame carrying a UDP payload of 1000 bytes and considering interframe gap). On the one hand, this jitter can be multiplied by n if some packets are arriving simultaneously at n interfaces being forwarded to the same output link. On the other hand, this situation is very unlikely in reality, because Premium Service packets consume only a small portion of overall bandwidth, and, in addition this jitter can be reduced if the speed of the physical output link is increased. In our tests we did not observe any significant change of traffic characteristics caused by interior routers. Furthermore, we observed from some delay measurements that delay caused by the backlog queue is $30\,\mu$s at most (the queue is served with a rate of approximately $1/30\,\mu$s). Protocol processing time for the IP protocol stack with Differentiated Services enhancements is below $10\,\mu$s. Naturally, both values depend mainly on CPU power of the router.



(a) Flow 1, 1.5 Mbit/s (b) Flow 2, 0.5 Mbit/s

Figure 12 Rate-Jitter after aggregation and shaping of single flows in a border router

Border routers will often shape outgoing PS traffic after aggregation to the configured aggregated rate for a particular outgoing link in order to forward a conforming stream to the next upstream DS domain. Because many different incoming PS traffic streams have to be multiplexed into one stream at a fixed rate, a border router is a serious source for rate-jitter that is mainly experienced by microflows. The behavior aggregate will not show any serious amount of rate-jitter, because it is shaped. But rate-jitter of microflows (or incoming aggregated traffic streams) within this aggregate is increased. It is caused by the fact that most packets of individual streams must be delayed in order to fit into the periodical traffic pattern of the shaped traffic aggregate. In consequence, packets of a single stream have to be delayed more or less, resulting in jitter.

We observed this effect also by measurements. In figure 12 the interpacket times of two individual flows within an aggregate after passing a border router are presented. Sender A and Sender B each generated one Assured Service traffic stream at 1 Mbit/s and 5 Mbit/s best-effort traffic. In addition, sender A sent a Premium Service flow at 0.5 Mbit/s and sender B one flow of 1.5 Mbit/s Premium Service. In figure 12(a) one can clearly identify the jitter that is caused by traffic shaping of the aggregate.

In previous tests UDP was used to examine effects of different forwarding behaviors on characteristics of single packets, while excluding hidden side-effects and interactions between functions of another transport protocol and our implementation of DS mechanisms. After validation of our implementation, we finally accomplished some tests using TCP. As mentionted before, because we had only 10Base2 technology available, acknowledgements of TCP would cause collisions on the segments. For that reason, we used an extra 10Base2 segment for transmission of TCP acknowledgements and all 10Base2 segments were used unidirectional only. Moreover, we also used the socket option `TCP_NODELAY` to disable the latency of the Nagle algorithm, and, we also removed the checksum calculation to allow the aforementionted insertion of timestamps by the router.

Although these constraints are not representative for most TCP connections in the 'real world', we wanted to get first impressions how most standard network-based application software could profit from Differentiated Services.

The duration of each test for behavior with TCP was increased to 100 s. In all tests sender A transmitted 1 Mbit/s Premium Service to C. There was no additional load necessary, because it would not influence the measurements. We also varied the round trip time (RTT) by using our traffic shaper for generation of an artificial delay. Results are plotted in figures 14 and 13 which show packet transmission time in dependence on packet number. As one would expect from theoretical considerations, TCP's mechanisms for congestion and flow control prevent it from performing well with some Differentiated Services.

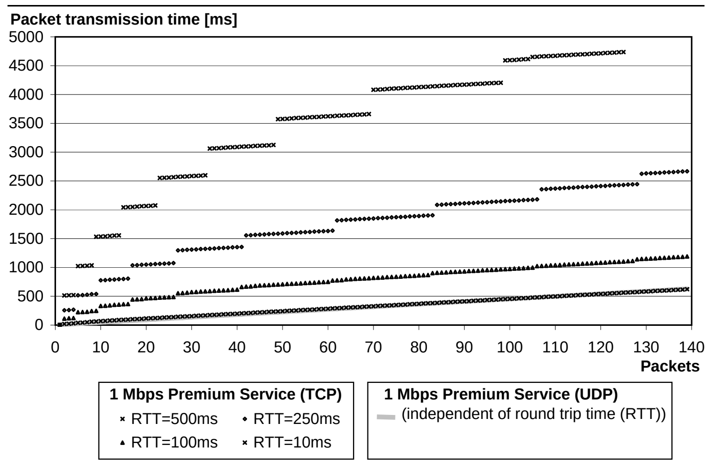


Figure 13 TCP behavior using Premium Service with bucket size 20000

One reason is the slow start algorithm. It prevents short-lived TCP connections (which are most HTTP connections) from exploiting the available bandwidth that was supplied by the Premium Service. As illustrated by figure 13 the shorter the round trip time (e.g., 10 ms), the better the performance of TCP. The optimal throughput was achieved at 10 ms RTT which is identical to shaped UDP traffic. If the round trip time increases, performance decreases.

Furthermore, if the size of the leaky bucket is big enough to hold a complete content of the sending window, no packets are discarded. What happens otherwise is shown in fig. 14: If a packet is lost TCP assumes that congestion occured somewhere in the network and reduces its sending window drastically to resolve the network contention and increases slowly the window again.

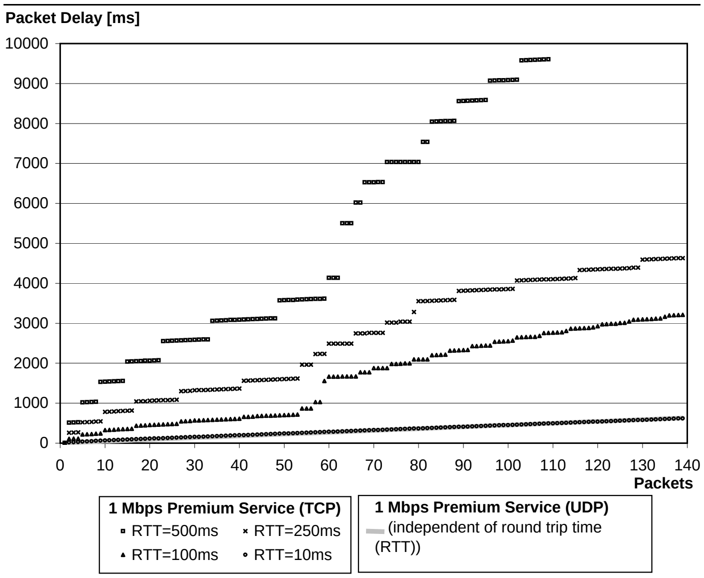


Figure 14 TCP behavior using Premium Service with bucket size 10000

There can be thought of some solutions to solve the observed problems with TCP. First of all, slow start is no longer needed for Premium Service connections, because packets cannot be congested on their way to the receiver. Another possibility is to let TCP begin with a higher sending rate at the very first time. Rate control or traffic shaping enhancements are also easy to implement into the current TCP protocol stack, but they will require a new interface option to propagate the value for the target rate from an application to the transport protocol.

5 Conclusions and further work

The implemented forwarding behaviors worked as expected. Premium Service can give a hard guarantee for throughput, but only if admission control, policing and shaping are applied consequently. Shaping of aggregated traffic increases rate-jitter of microflows contained in this aggregate. If the receiving end-system or application is unable to compensate for this jitter, the last-hop router could shape the traffic again.

Quality of Assured Service traffic can be improved by also applying traffic shaping to it, because otherwise bursts are aggregated and lead to much higher drop probabilities of packets on their way downstream. Delay of Assured Service depends much on the current amount of Premium Service.

Currently, a management architecture of Differentiated Services is lacking. Most work is concentrated on forwarding behavior. Only if end-users could easily order and use services on demand, Differentiated Services will be accepted by them. Moreover, DS is mainly sender-based. Support of required admission control, accounting, receiver initiated services, multicast and even mobility could be accomplished by using a service management architecture. In further research we focus on this topic.

Additional development of our implementation is planned in order to integrate and investigate new services or forwarding behaviors. Currently, a new testbed with a Fast-Ethernet (100 Mbit/s) DS capable infrastructure is build. New and more comfortable programs for performing tests are going to be developed. We also plan to release the current implementation to the public for research purposes [KIDS99].

Nearly all executed measurements used a fixed packet size to identify the presented effects uniquely. As mentioned before, experiences with 'real' applications are missing. Thus, a next series of tests is planned to measure performance and quality gain for standard applications (e.g., telnet, ftp, vic, vat, etc.) using Differentiated Services. Especially, behavior of interactive applications will be examined.

References

[BBBN98] F. Baker, D. Black, S. Blake and K. Nichols. Definition of the Differentiated Services Field (DS Field) in the IPv4 and IPv6 Headers. RFC 2474, December 1998.

[BBCD+98] S. Blake, D. Black, M. Carlson, E. Davies, Z. Wang and W. Weiss. An Architecture for Differentiated Services. RFC 2475, December 1998.

[BHWW99] F. Baker, J. Heinanen, W. Weiss and J. Wroclawski. Assured Forwarding PHB Group. Internet draft – draft-ietf-diffserv-af-06.txt, February 1999.

[FeHu98] P. Ferguson and G. Huston. *Quality of Service*. Wiley, 1998.

[FlJa93] S. Floyd and V. Jacobson. Random Early Detection Gateways for Congestion Avoidance. *IEEE/ACM Transactions on Networking*, 1(4):397–413, August 1993.

[JaNP99] V. Jacobson, K. Nichols and K. Poduri. An Expedited Forwarding PHB. Internet draft – draft-ietf-diffserv-phb-ef-02.txt, February 1999.

[JaNZ97] V. Jacobson, K. Nichols and L. Zhang. A Two-bit Differentiated Services Architecture for the Internet. Internet draft – draft-nichols-diff-svc-arch-00.txt, November 1997.

[KIDS99] Karlsruhe Implementation of Differentiated Services (KIDS) homepage. http://www.telematik.informatik.uni-karlsruhe.de/forschung/diffserv/KIDS/, April 1999.