

An Implementation of A Two Rate Three Color Marker Marking Scheme for Differentiated Services

Scott Thomas
CSC 564, Winter 2003

March 20, 2003

Abstract

Quality of Service is an area of much research in the networking domain. Specifically, much thought and effort has gone into the Differentiated Services method of providing Quality of Service on the Internet. Differentiated Services consists of three phases: Filtering, Marking, and Queuing. The three components are separate and each can be implemented using a number of different schemes.

The Two Rate Three Color Marker marking scheme is a simple, effective marking scheme for differentiated services. The marking scheme works on a stream of traffic and is based on two rates, the committed information rate and the peak information rate. A packet is marked green if it does not exceed the committed information rate. The packet is marked yellow if it exceeds the committed rate, but not the peak rate and red otherwise. The mark is used to determine how the packet will be queued at subsequent hops on the Internet.

This paper covers a background on Quality of Service including the current areas of research in differentiated services. It then goes into detail on my implementation of a two rate three color marking scheme. Included in the description are design decisions made, and the results of running the marking scheme. I conclude by discussing future work that could be performed in this area.

1. Introduction

Quality of Service (QoS) on the Internet is a major area of research effort. The standard Internet is a realm of best-effort service, meaning that all traffic is treated the same and has equal chance of being thrown away or delayed, possibly causing the data never to reach its final destination. This kind of service can wreak havoc on applications such as video conferencing and voice over IP. Video conferencing is a bandwidth intensive application in which the data needs to move fast, but some loss is acceptable. The video codecs in use today can handle some loss and still display a video stream that is acceptable to the viewer. Voice over IP, allows voice to be transmitted over an IP network. Voice compresses well and therefore does not require much throughput, however it is very sensitive to delay and loss [1]. Quality of Service assurance is not only helpful in providing specific data a certain level of service, but is also useful in restricting a data stream to provide fair service to all. As it is now, the “friendly” connections, which watch out for others and share the communications link, are getting cheated out of their fair share by the high throughput video applications. QoS can be used to restrict the amount of bandwidth a videoconference can take allowing other Internet traffic to use a link. As applications like video conferencing and Internet telephony become more and more prevalent, the Internet protocol’s standard best effort service will no longer be acceptable. Many standards of QoS have been proposed, but the two main ideas are presented in the following sections.

Integrated Services

Integrated Services (intserv) provides three different levels of service. The first level is called the best effort service class, which provides ordinary IP best effort service. All QoS functionality is turned off, even though it is still using intserv. The second level is the extreme opposite of the first. This guaranteed service class provides a guaranteed amount of bandwidth available to the user at any time. A virtual circuit path is set up between the source and destination and all traffic traverses the same path. No one else can use this bandwidth even if the owner is not using it. While this provides a guaranteed level of service, the overhead involved is extreme. This setup requires some kind of call admission control, signaling, and policing. The last level of intserv is called the controlled load class. This class provides service that approximates IP’s best effort service on a lightly loaded link no matter what the actual condition of the link is.

Differentiated Services

Differentiated Services (diffserv) was designed to provide different levels of service over the Internet. It was published as a proposed standard request-for-comments in December 1998 [2]. The authors of diffserv kept three main goals in mind during the design of diffserv. First, the QoS option must be protocol layer independent. In other words, it must work with TCP, UDP, and any other transport protocol that may be used. In order to achieve this goal, the authors used fields in the IP header to indicate QoS information. Therefore, no matter which transport level protocol is being used, the routers will be able to determine the QoS appropriate to the stream from the IP header. The second goal the authors had was that diffserv was to be easily scalable. This means that the bulk of the work needs to be concentrated on a small number of routers. It was determined that the ingress and egress routers should perform these timely operations. Packets will be analyzed and marked upon entry to a differentiated services cloud, and this marking will be used as the data progresses through the network. At the egress router, the marking is stripped off and the packet is passed into the next cloud. Figure 1 shows this process in detail.

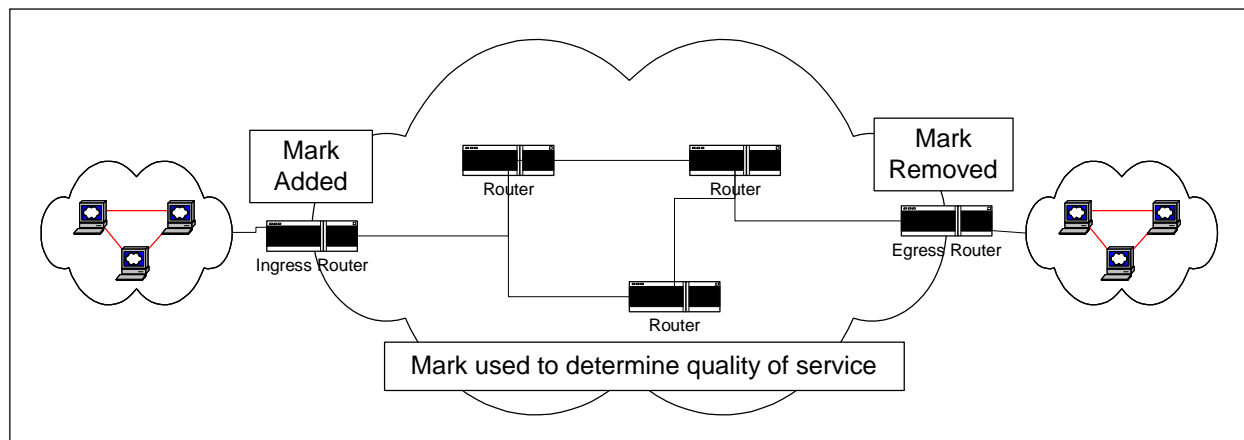


Figure 1: QoS is scalable

The last goal the diffserv team had was to keep the protocol lightweight. For the majority of hops, little processing was desired. This fit right in to the scalable situation described above as most routers simply check a few extra bits in the IP header to determine the packets level of service. This means only the ingress router has to do a significant amount of processing.

There are three main components of the differentiated services architecture. First, packets are filtered into streams of similar traffic. Diffserv works to provide a certain kind of guaranteed service specific to each stream. A stream of traffic includes data with similar quality of service requirements. These requirements can include bandwidth, delay, jitter, and/or loss. Guaranteed bandwidth or throughput requirements are used for applications where the majority of data needs to get through quickly. Video conferencing is an excellent example of a stream of traffic requiring a guaranteed throughput. Delay is the amount of time it takes a packet to travel from its source to its destination. In this case, the amount of the data is not of concern, however the data needs to arrive quickly and in order to be processed correctly and provide uninterrupted service. Internet telephony is an example where only voice, which is highly compressed, is transmitted and is dependent on small delay. Jitter is also important in the area of voice over IP. Jitter is the difference in the delay over time. If the time a packet takes to travel the same distance varies over time, it can be extremely difficult for the application to provide good service. To provide a low level of jitter means that the packets will take the same amount of time to traverse the network the majority of the time. How the data is grouped into streams can be based on any information in the IP header. The most popular methods of filtering are based on the source or destination IP address, the transport protocol being used, or the size of the packets. Once the packet has been associated to a certain stream, marking is performed.

Marking a packet is the method used to tell the downstream routers if the stream is behaving according to the QoS parameters set up. Marking can be configured in a number of different ways. When a contract is set up between a client and a service provider, certain things are agreed upon on each side. The provider assures a certain level of service as long as the client does not exceed a given rate or usage percentage. When packets arrive at an ingress router, the packet is filtered and passed to the marking scheme. The marker compares the packet to the allotted throughput and determines whether it is within the allowed range. The packet is then marked according to whether or not it is abiding by the contract provided. Once the packet is marked, it is ready to be sent out the correct outgoing interface.

The last component of diffserv is queuing strategy. Queuing strategy involves taking the mark on the packet and using it to adjust how the packet is sent. A simple, but extreme example of this is

just dropping any packets that are not completely within a streams allocated throughput. A queuing strategy can be simple, as shown above, or it can be extremely complex, involve many different algorithms. The ultimate goal of the queuing strategy is to use the marking to provide as close of service as possible that specified in the contract.

2. Related Work

The need for QoS on the Internet has been proven many times. Fortunately, no standard implementation has been picked which leaves the area open to continued research. The areas of research can be broken down into the research being performed on each component of diffserv. I will first cover research in the filtering area, followed by research in the marker area, and conclude with some of the queuing strategy research currently being performed.

Filtering Research

The filtering process is used to associate the data with different streams. The QoS parameters are mapped to different streams and it is imperative that each packet is associated with the correct stream. If a packet is placed into the wrong stream, not only does it mess up the packet's service, but it also disrupts the statistics of the traffic stream it was placed in. Most likely these wouldn't be isolated incidents either, as once rules are set up on the routers, they don't change often. An interesting area of research is determining the best method of detecting overlapping rules, as well has how to solve the problem. The ruleset shown in figure 2 is an example of conflicting filters, which could exist in a router.

```
1. src = *; dst = 10.*.* => 10 Mbps  
2. src = 10.*.*; dst * => 100 Mbps
```

Figure 2: Conflicting Filters

This is a clear example of conflicting filters. If a packet comes in from the 10.*.* subnet and has a destination on the 10.*.* subnet, which rule is supposed to be applied? Different possibilities exist, for example, the most common idea is to take the first matching filter and apply that one. However, this can become very confusing as now the order in which the rules are

added affects the outcome. Another possible solution is to assign weights to each of the rules and follow the rule with the highest value. The most beneficial way, of solving this problem is by using algorithms. Research has been performed and algorithms have been developed that will detect conflicting filters and generate patch filters to cover the overlapping areas [3].

Marking Research

When a packet arrives at the marking step in a router supporting diffserv, the average rate the stream has been transmitting at is calculated and compared to the allowed rate. The packet is marked indicating whether it is below, at, or above the allowed rate. Because data rates are increasing rapidly, the method used to calculate a flow's data rate must be incredibly efficient. With bandwidth reservations approaching Giga-bits per second, the routers need perform calculations on the order of billions per second. Currently, there are two main methods for performing this calculation in practice today: token bucket and time sliding window.

First is the token bucket method, which is described, in complete detail in rfc 2212 [4]. The token bucket contains two values, t and r . t is defined as the total number of tokens in a bucket, and r is the rate that tokens are added. r is the rate that the connection is allowed per second. For example, if a stream is allowed 10 bytes per second, then the token bucket will have 1 token added to it 10 times per second. The other variable, t , is more difficult to calculate. If t is low, then the token bucket does not handle bursts well, as everything over 10 bytes in one second is marked non-conformant. However, if t is a large value, the bursty traffic can be handled, but streams can take up more than the fair share of bandwidth. Two examples of token bucket marking schemes include the Single Rate Three Color Marker [5] and the scheme I implemented, the Two Rate Three Color Marker [6].

The other method of metering traffic is based on a time sliding window (TSW) mechanism. The time sliding windows consists of two independent components [7]. The first component is the rate estimator. This is an algorithm running which takes into account the bursty and silent nature of data traffic and provides an estimate of the data rate to the TSW tagger. The tagger marks the packet as IN if the calculated rate is below the target rate, and OUT if it is above. The problem with this is that the algorithm tends to be generous in the number of packets it marks as IN [8].

Research has been performed in developing new algorithms that provide more accurate calculation of the rate of bursty streams of data. The Enhanced TSW profiler was created by using dynamic numbers in the algorithms and eliminates the extra friendliness of common TSW implementations [8]. TSW has also been modified to add a third class of traffic called OUT-IN meaning that the stream is over its target rate, but hasn't been in the past [9].

Queuing Strategy Research

The last research area in diffserv involves the queuing strategies used to drop, delay, or send packets. This is by far the most complicated area of diffserv. Delaying or dropping packets messes with the order the data will arrive at the destination, which wreaks havoc on the end applications. Also, if TCP is being used, it will detect a loss and cut its transmission rate in half, when all it needed to do was drop it a tiny bit. Research has been done at Texas A & M in which the sender's TCP stack is changed [9]. They modified the marker to send a notification back to the sender if it had a packet marked with an OUT. The sender would then reduce its window size by one packet to avoid sending too much data again. There has also been research into not dropping important TCP packets no matter how the packet is marked. For example, TCP SYN packets should not be dropped in order to let the client establish new connections [10]. Ideally the first chunk of packet would not want to be dropped to let the TCP connection establish itself and let the congestion window drop. However, this would require keeping state on the routers with is not allowed. Many of the problems associated with queuing strategies would be easily resolved if state were allowed to be kept on all routers. However, with core routers servicing hundreds of thousands of connection concurrently, this is simply not feasible.

RED is a popular queuing strategy among routers in the core of the Internet. There has been research on how RED can be modified to help the spread of diffserv. RIO [11] introduces multiple drop thresholds for different streams of traffic. This provides dynamic allocation of resources to streams that have a reserved bandwidth higher than what is available currently on the router. The router is able to shift extra resources to ensure that the connection receives the amount quality of service required.

3. Implementation Discussion

For this project, I chose to implement the Two Rate Three Color Marker [6] marking scheme for diffserv. This marking scheme is based on two rates, a committed information rate (CIR) and a peak information rate (PIR). The CIR is the rate that a client is guaranteed. If a packet comes in from the filtering step and is below the CIR, the packet is marked green. If a packet arrives at a rate between the PIR and CIR, the packet is marked yellow. In all other cases, the packet is marked red. This marking scheme uses a token bucket to keep track of the individual streams incoming data rates.

I chose to implement this marking scheme in the C programming language. My implementation uses four different threads to implement the marking functionality. The threads communicate with each other through the use of queues implemented as linked lists. Figure 3 shows the flow diagram of my implementation.

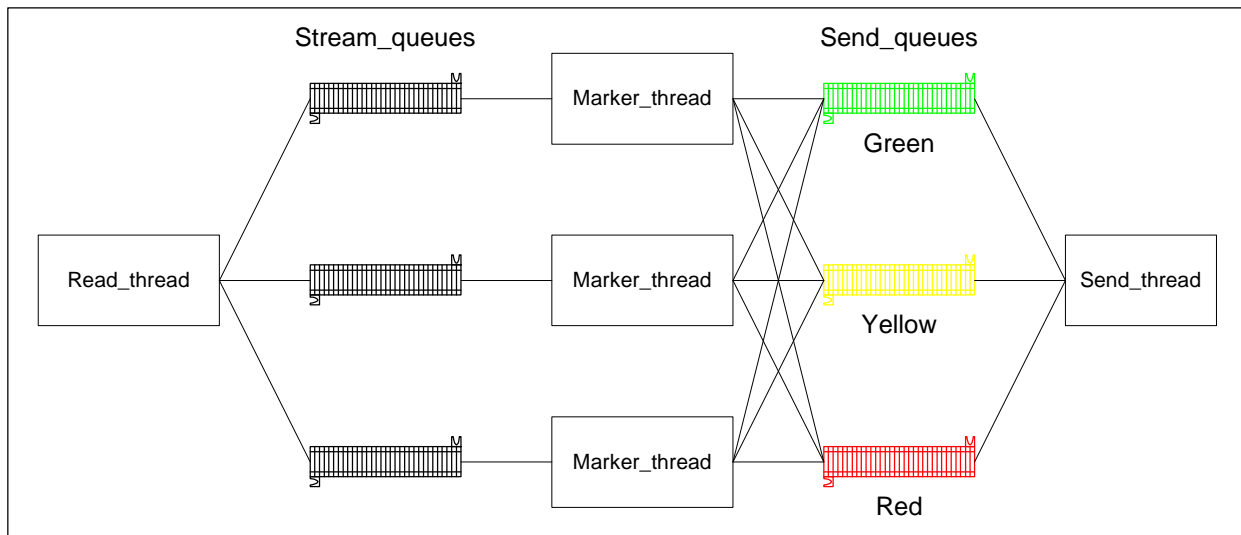


Figure 3: Flow Diagram

The main thread (not shown above) is responsible for all the setup. The read thread takes in all the incoming packets on the first network interface. It determines which stream the packet belongs to and places it in the correct queue. The marker thread dequeues the packet and calculates the mark that should be placed on the packet. Once it has marked the packet, the

marker thread places it on the appropriate send queue. The send thread cycles through all three send queues and sends data out the second network interface.

Main Thread

The main thread is responsible for initializing all the queues and spawning the thread processes. In this implementation, all queues are static and must be declared globally in the main.c source file before compilation. The main thread runs an init function on all queues to initialize the mutex lock on each queue. This is due to the fact that two threads will be accessing each queue. Next, the main thread spawns the send thread. The send thread has access to the three queues as they have been declared global, and starts checking to see if packets are ready to be sent. Then the main thread spawns the marker threads. The number of marker threads to be spawned must equal the number of stream queues. When a marker thread is spawned, it is passed a pointer to a structure. This structure (called a stream structure) contains the CIR and PIR for that stream, as well as a pointer to the queue that it is to service. This structure also contains a unique id whose purpose will be described later. Lastly, the main thread spawns the read thread which starts reading packets on the incoming interface and queuing them into the appropriate stream queue.

Read Thread

The read thread is passed a string, which it will use to filter the incoming packets. This string is superset of all the individual filters. The read thread then uses pcap to open a raw socket connection and start receiving packets. Upon receiving a packet, the thread checks which flow the packet belongs to and starts processing the packet. The read thread must be hard coded with the properties of a packet that determine which queue it should be placed in. For example if all packets from 10.0.0.1 belong to stream1 and all packet from 10.0.0.2 belong to a second stream, two queues must be statically declared, and the thread must be programmed to check the source IP and place the packet in the appropriate queue. Because the pcap capturing software is driven by the arrival of packets, it places each packet into the same location in memory. Therefore, the read thread must copy this packet to a dynamically allocated place in memory before placing a pointer to the packet in the queue. Once the thread has performed this copy, it wraps the packet in a pkt_node structure. This structure adds a pointer to the next node for use in the queues. It then enqueues the packet onto the correct queue according to the source IP address. In order for

the thread to enqueue a packet, it must get a lock on the queue guaranteeing that it is the only one accessing the queue at that time. Once the pointers have been manipulated, the lock is freed and others are free to access the queue.

Marker Thread

Each marker thread is passed a pointer to a unique stream structure. This stream structure contains the CIR and PIR for the stream, as well as a pointer where the queue the packets for its stream will be placed. Ideally, the CIR and PIR token buckets are to be decremented by the number of bytes in a packet, and incremented by one CIR or PIR times per second. Unfortunately, this is not possible in my implementation. In order to get around this I chose to update the number of tokens once per second. I start a timer at the arrival of the first packet that fires every second, resetting the CIR and PIR buckets to their initial values. When the timer fires, all slots in a global array are set to 1. Each time the marker thread checks to see if a new packet has arrived for marking, it checks this value using the unique id number that is included in the stream structure. If the value is 1, the thread resets its local token counts to the CIR and PIR passed in. The algorithm shown in figure 4 is performed on all packets process by the marker thread.

```
Let Tp be the number of tokens in the PIR token bucket
Let Tc be the number of tokens in the CIR token bucket
Let B be the number of bytes the packet is

if (B > Tp)
    Color(RED)
    Enqueue(Red)
else if (B >= Tc)
    Decrement(Tp)
    Color(YELLOW)
    Enqueue(Yellow)
else
    Decrement(Tc)
    Decrement(Tp)
    Color(GREEN)
    Enqueue(Green)
```

Figure 4: Marking Algorithm

If the packet is larger than the amount of tokens left in the peak bucket, then the packet is marked red and enqueued onto the red sending queue. If the packet is in between the CIR and PIR, then the packet is marked yellow, the PIR token bucket is decremented by the number of bytes in the packet, and the packet is enqueued onto the yellow send queue. If the packet contains fewer bytes than are currently in the CIR token bucket, both token buckets are decremented by the size of the packet. The packet is then marked green and placed in the green sending queue. Setting the TOS bits in the IP header in different ways signifies the color of the packet. The number 1 in the 8-bit TOS field represents a green packet. The number 2 represents a yellow packet, and 3 a red packet. When the mark function is called on a packet, the TOS bits are set appropriately.

Send Thread

The send thread has the easiest job of all the threads. It is responsible for emptying all three send queues as fast as it can. There are many different queuing strategies that could have been implemented here to use the different colors, however since my project was focusing on the marking, I implemented a simple queuing discipline. Each queue has a length associated with it. This length is the number of packets currently in the queue. First, the send thread checks the green queue to see if any packets are waiting to be sent. If there is a packet it sends it, otherwise to moves on to the next queue. It checks to see if there are any packets in the yellow queue. If there are packets in the yellow queue AND the green queue still has no packets, it can send a packet from the yellow queue. If packets had arrived in the green queue, the thread will move back and empty the green queue and try the yellow again. If the green queue is empty and the yellow queue is empty, the thread will check to see if any packets are in the red queue. If it finds one and the green and yellow queues are still empty, it can send the red packet. Upon deciding to send a packet, the send thread needs to load the socket structure with the mac address of the destination. The send thread also needs to compute the IP checksum as changing the TOS bits in the IP header invalidates the old one.

Results

In order to test my implementation, I needed a device that would be able to create much data to send at my application. I chose the SmartWindows application as I could customize the speed

and contents of the data being dispersed. The setup I used to test the application is shown in figure 5.

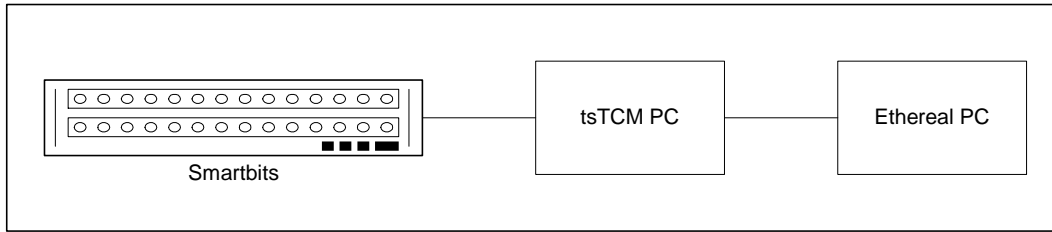


Figure 5: Testing Setup

The smartbits box was wired through the patch panel directly into the eth0 interface of the Two Stream Three Color Marker (tsTCM) PC. A crossover cable was used to directly connect the tsTCM PC and the PC running Ethereal. The smartbits box and the eth0 interface of the tsTCM PC were members of the 192.168.160 subnet, while eth1 of the tsTCM PC and the Ethereal PC were on the 192.168.140 subnet. The smartbits box generated packets with a destination of the ethereal computer. The mac address in the packets was the hardware address of the eth0 interface on the tsTCM PC. The following chart shows the results produced using one stream of traffic.

CIR (KB/s)	PIR (KB/s)	Send Rate (KB/s)	Total Data (KB)	Green Data (KB)	Yellow Data (KB)	Red Data (KB)	Total Processed
50	75	50	1000	952	48	0	1000
50	75	75	1500	1000	441	59	1500
50	75	100	2000	1000	500	500	2000
500	750	500	10000	9940	38	0	9978
500	750	750	15000	9935	4801	20	14756
500	750	1000	20000	9938	5000	5000	19938

Chart 1: Data results processing 1 stream of traffic

This chart shows that the two rate three color marker marking scheme is accurate to within 5%. At the lower speeds, the marker handles all packets, processing them appropriately and is able to get them all back on the line. However, at higher speeds, the marker falls behind and packets begin to get dropped. However, the packets that are handled emerge from the marker in the correct color. The next chart shows what happens when two clients send data at once.

Stream	CIR (KB/s)	PIR (KB/s)	Send Rate (KB/s)	Total Data (KB/s)	Green Data (KB)	Yellow Data (KB)	Red Data (KB)	Total Processed
1	10	20	20	400	248	110	34	392
2	20	40	20	400	348	43	0	391
1	50	75	75	1500	1003	153	24	1180
2	100	150	75	1500	1168	0	0	1168

Table 2: Data results on multiple data streams

I had to send the data at much slower rates, as many collisions were resulting and the software could not keep up. In the first two rows, stream one is sending at his peak rate, and stream 2 is sending at his committed rate. Both streams had an even amount of bytes processed, but notice stream two came out with more green packets. The second two rows show the same as the first; Even though packets are getting dropped, the number of outgoing packets still obeys the marking scheme.

Future Work

The user interface of the program could use much work. Currently everything needs to be hard coded into the program and recompiled. It would be nice to be able to invoke the program and change the number of marker threads, and what they are filtering traffic by. Also, currently the mac address that the send thread puts in the packet is hard coded into the program. It would be nice to have the send thread do an arp on the destination IP address and insert the correct mac address of the next hop into the packet.

This implementation only supports udp traffic, as the software will only process packets in one direction. In order for TCP to be able to run across the implementation, the acknowledgment packets must be able to make it back in the other direction. Enabling the software to process TCP would open many research areas as TCP reacts to loss as congestion and drops its rate in half. It would be interesting to see if the rate achieved is close to the committed rate.

An interesting area of research would be to get a faster computer, which can process the packets at line speed. Currently, the packets are queued, waiting to be marked. The packets should be marked immediately as they enter the machine in order to ensure 100% accuracy. The other area

that needs to be changed is the token bucket scheme. Finding a clock that was accurate to the millionth of a second so that one token could be added CIR or PIR times per second, instead of adding CIR or PIR tokens once per second may produce different results.

The last improvement that would be interesting to implement would be performing the IP checksum by some more efficient means. One possibility would be to have a hardware chip that is dedicated to performing IP checksums. This would increase the speed, as the hardware can process the checksum faster than using software. Another possibility would be to somehow manipulate the IP checksum to reflect the changes made to the TOS bits. The marking performed only affects one to two bits in the header and it would be awesome if the new checksum could be determined without recalculation.

4. Conclusion

This paper provides a background on Internet quality of service. It quickly covers the Integrated Services architecture and gives a detailed explanation of the Differentiated Services standard. The paper discusses each component of diffserv including filtering, marking, and queuing strategy in detail. It also explores the current research areas in each of the three components. Once the background has been explained, the paper describes my implementation of a Two Rate Three Color Marker marking scheme. The implementation description covers the flow of a packet from the time it is taken off the line until it is placed back via the second interface. Each thread is described in detail including some of the implementation compromises I made. The results show that the implementation is accurate over a wide range of data streams, even if the computer was overloaded.

References

- [1] Nikolaos Vasiliou, *Overview of Internet QoS and Web Server QoS*, Department of Computer Science, The University of Western Ontario, London, Ontario, Canada. <http://citeseer.nj.nec.com/vasiliou00overview.html>
- [2] S. Blake, D. Black, M. Carlson, E. Davies, Z. Wang, W. Weiss. *An Architecture for Differentiated Service*. RFC 2475. December 1998.
- [3] H. Adishesu, S. Suri, and G. Parulkar. *Detecting and resolving packet filter conflicts*. In Proc. INFOCOM, volume 3, pages 1203--1212. IEEE, March 2000. <http://citeseer.nj.nec.com/hari00detecting.html>
- [4] S. Shenker and J. Wroclawski. *Specification of guaranteed quality of service*. Internet-draft, Internet Engineering Task Force, July 1995. <http://citeseer.nj.nec.com/shenker97specification.html>
- [5] J. Heinanen, R. Guerin, *A Single Rate Three Color Marker*. Informational RFC2697. September 1999.
- [6] J. Heinanen, R. Guerin, *A Two Rate Three Color Marker*. Informational RFC2698. September 1999.
- [7] Wenjia Fang. *Differentiated Services: Architecture, Mechanisms and an Evaluation*. PhD Dissertation - Princeton University. 2000.
- [8] W. Lin, R. Zheng, and J. Hou, *How to make assured services more assured*, in Proceedings of the 7 th International Conference on Network Protocols (ICNP'99), Toronto, Canada, Oct. 1999.
- [9] Ikjun Yeom and A. L. Narasimha Reddy, *Realizing Throughput Guarantees in a differentiated services network*, to be appeared at Proceedings of ICMCS, June, 1999. <http://citeseer.nj.nec.com/yeom99realizing.html>
- [10] A. Habib, S. Fahmy, and B. Bhargava, *Design and evaluation of an adaptive traffic conditioner in differentiated services networks*, IEEE ICCCN, Scottsdale, Arizona, USA, pp. 90--95, Oct 2001.
- [11] D.D. Clark and W. Fang, "Explicit allocation of best effort packet delivery service," IEEE/ACM Transactions on Networking, vol. 6, 4, pp. 362--374, 1998.