CHAPTER 1

Introduction

1.1 Background

In the recent years, there has been a great deal of research focused on achieving and sustaining certain guarantees (QoS)[1] to the traffic that flows through the Internet. The support of these guarantees ultimately translates into the preferential treatment of certain flows of traffic, and the corresponding processing in order to achieve this behavior. The architecture known as "Integrated Services" (*intserv*)[2] was first introduced as a mechanism for establishing QoS. Despite its mathematical proven success on guaranteeing queuing delays on routers along the path, its complexity for large-scale deployment (in terms of buffer resources and RSVP signaling) has given rise to another architecture known as "Differentiated *diffserv*)[3]. This new architecture features a more scalable approach, as complex operations are pushed towards the edges of the network and the core remains at a relatively simple complexity level. In addition, it does not require signaling among routers and provides flexibility on the implementation of new and distinct classes of traffic across many different domains within the Internet.

The *diffserv* architecture is based on the use of traffic conditioning and classification at the edges of the network, by appropriately marking the TOS[4] field of the packet's IP header. Subsequently, scheduling and buffer management algorithms, at the core of the network, forward packets according to their TOS value and the correspondent service level agreements that they are intended to support. The results

[1] Quality of Service
[2] [BCS94]
[3] [Bla98]
[4] Type of Service

translate into “better than best-effort” service that scales along with the growth of the Internet.

Throughout the study presented in this document, we focus on a specific type of traffic conditioner, the TCP Friendly marker[5], and its effects on the overall performance of the traffic flows, especially during episodes of congestion. We couple the experimental implementation of the marker with the use of two buffer management algorithms. The first of them, RIO[6], is implemented from an existing design that had been tested on simulation. The other one, FRIO, object of our own design, comes as a hybrid evolution of the existing FRED[7] and RIO. The overall scheme consists in segregating the total flow of traffic into two classes and subsequently dropping the lower class with a higher preference in the event of gateway congestion.

We demonstrate the benefits of the marker, under several different scenarios and using two different versions of the underlying transport protocol. We show **major improvements in terms of network utilization, timeouts, predictability of service, probability of packet loss, and net data throughput**. We also prove the **independence of the benefits from the version of TCP** implementation **and from complex flow classification at the core of the network**. In addition we show that our **approach is relatively insensitive to parameter misconfiguration and that scales gracefully** as the flows into the system increases in number.

[5] [Fer2000]
[6] [CF98]
[7] [LM97]

1.2 Overview

This report is composed of six chapters. In chapter 2 we discuss the issues regarding performance of TCP Reno and SACK. We also argue the design of the TCP Friendly marker and its implementations issues. In addition, we talk about the buffer management schemes used in our experimentation and the specifics of their implementations.

In chapter 3 we present the basis of our performance analysis and the details of the experimentation setup. In particular, we discuss the metrics used in the evaluation and the configurations under which all of the experiments were carried out. Later, in chapter 4, we present both tabular and graphical results that support our expectations from the analysis in earlier chapters. We show the results obtained under all of the configurations described in chapter 3.

In chapter 5 we argue the direction of future developments of our approach. Finally, on chapter 6, we make a summary of the conclusions of our study and the contributions to the overall area of TCP traffic management research with an emphasis on open source code solutions.

CHAPTER 2

TCP-Friendly Traffic Conditioning and Buffer Management

2.1 TCP Implementations

The original implementations of TCP established a stream-based reliable transfer protocol, which used a sliding window, and acknowledgement packets from the receiver to enable the transmission of new data by the source. They followed a *go-back-n* model that required the source to retransmit data (on the assumption of it being lost during transport) at the expiration of a retransmit timer. These first versions[8] of TCP worked reliably but were unable to maintain acceptable performance in a large shared, and possibly congested, network of hosts.

A later implementation, TCP Tahoe, introduced significant improvements for working over a shared network. Two algorithms[9], *slow start* for congestion control and multiplicative decrease for *congestion avoidance*, were introduce to manage the transmission at the onset of any detected congestion. In addition, *fast retransmit* allowed the retransmission of new data on the presence of three duplicate acknowledgements (dupacks), without having to wait for timer expiration (timeout). The variable *ssthresh* was introduced to keep the threshold value for the size of the sending window (*cwnd*). This variable was initialized to the receiver's advertised window size and, after a timeout, was then set to half of *cwnd* while this last being set to one segment, signaling the beginning of the slow start phase. Subsequently, *cwnd* was increased by one segment for every ack received until it reached the value of *ssthresh*. At this point, the congestion avoidance phase begins and the window size is

[8] 4.2 BSD, in 1983
[9] [Ste97] and [Jac88]

only increased by a fraction of the segment size, equivalent to an increment of one segment per round trip delay.

The subsequent TCP Reno introduced *fast recovery*[10] to improve the performance after retransmission. As the third dupack is received, the Reno source sets *ssthresh* to one half of *cwnd* and retransmits the missing segment. This last is then set to *ssthresh* plus three segments, and later increased by one segment on reception of each dupack that continues to arrive after fast-retransmission. This new scheme allows the transmitter to send new data as *cwnd* is increased beyond its value before fast-retransmission. When an ack arrives which acknowledges all outstanding data sent before the dupacks were received, *cwnd* is set to *ssthresh* so that the sender slows down the transmission rate and enters the linear increase phase.

Since its adoption, Reno has become the de facto standard implementation of TCP and it represents the largest installed based of TCP implementations for hosts on the Internet. Nonetheless, Reno exhibits a fundamental weakness determined by the lost of multiple packets within the same window of data. Dependent upon the size of *cwnd* and the round trip transmission time, as little as three packet drops in a single window trigger multiple timeouts and unnecessary reductions to *ssthresh*. Furthermore, anecdotal evidence has also established[11] that, at the Internet's major exchange points, multiple packet drops within one round trip time are of fairly common occurrence. This happens regardless of the use of RED-based gateways since at the onset of congestion, the dynamics of the dropping mechanism is not correlated with the interleaving of packets belonging to a single flow, but instead with the overall rate of traffic arriving at the gateway. Thus, packets from a single flow within the same window have a high likelihood of being dropped at the start of any instance of congestion, as interpreted by the RED buffer management algorithm. We will consider a solution to this fundamental problem that goes beyond the dynamics

[10] [Ste97]

[11] [FF96]

of Reno and exploits the capabilities of differentiated services architectures, supporting traffic conditioning at the edges of the network and its correspondent buffer management complement at the start of the dropping bottlenecks.

The analysis of such Reno weaknesses ultimately established that multiple packet drops in a single roundtrip time unavoidably required a timeout before the appropriate recovery mechanisms could be executed by the source. Simultaneously, a new evolution of TCP, SACK[12], eliminated this requirement by implementing a new header option that is triggered as the receiver holds data segments that follow lost of previous segments. The receiver then sends dupacks that inform the source of the successfully received segments that arrived after the (supposedly[13]) lost ones. As the third dupack arrives, the source exclusively retransmits the missing segments, as well as any subsequent unacknowledged ones. The information carried by SACK dupacks allows the source to accurately estimate the number of sent data segments that have left the network (i.e. have arrived at the destination), which in turn enables injection of new data during the retransmission process.

The use of the SACK option allows for a sustainable high throughput over a network exhibiting high probability of loss. This behavior is better exploited in long delay links, as TCP SACK enters slow start much less often than its Reno counterpart. More importantly, **the reluctance of SACK to wait for timeouts to act upon packet loss is certainly a critical factor influencing its greater performance**. However, this performance improvement is optimal when the retransmissions have no probability of loss, and maximal when such probability is the lowest possible compared to the probability of the original transmission. We will consider this issue in the design of our diffserv solution to the pathology of this version of TCP.

[12] [MMFR96]

[13] Before 3 dupacks, out-of-order segments are not yet assumed lost by the source.

2.2 TCP-Friendly Packet Marker

From the previous analysis describing the issues of Reno and SACK, we consider the use of a previously proposed "TCP-Friendly" marker that acts as a traffic conditioner element to be used in the edge routers of a particular network domain. According to the reference[14], the original marker is designed with the following purposes on mind:

- **Protect small-window flows from packet loss.**
- **Maintain optimum spacing between "IN" and "OUT" tokens.**

We have now introduced certain modifications to the original algorithm aimed to address the following issues:

- **Protect retransmissions**.

This refers to the idea of marking retransmitted packets (given enough tokens) as "IN" packets. We encounter that retransmissions, since occurring mostly during periods of large congestion, suffer from a significantly higher probability of loss than the overall data traffic. This, of course, worsens as the number of flows into the system increases. Therefore, this modification improves the performance of the data transport, as reducing loss of retransmissions will inevitably lead to less timeouts, under any TCP implementation. We mark retransmissions as "IN" to avoid them being dropped at the bottleneck gateway.

- **Establish a scheme to deal with new connections in an ongoing marking process.**

This refers to the fact that new connections will not be able to get a fair allocation of tokens before the new updating process gets executed. We solve this practical issue by initiating the updating process as soon as a packet from a new connection is

[14] [Fer1999]

identified, thereby allocating tokens for its future packets. However, this new scheme implies that the time interval between updates would not always be constant (as set by the user during configuration), but will instead sometimes be shorten by the arrival of SYN packets from a new connection. This in turn results into an inaccurate estimate of packets from existing flows, which ultimately translates into a lesser number of tokens being allocated for such flows. We solve this problem by exploiting the following two observations:

1. The shorter[15] the inter-updating time (time between successive executions of Algorithm 1, in Figure 2.2.1) the less accurate the byte traffic estimate is, for existing flows. This occurs because existing flows would not be able to introduce as many packets to the marker, on the shorter interval, as they would in the event of a longer and constant inter-updating time.
2. The shorter the inter-updating time, the larger the number of remaining tokens after the allocation is completed. As the byte estimates get shorter (because of inter-updating time getting shorter) the number of tokens needed to fulfill these estimates is proportionally smaller. Therefore, meeting the needs of existing flows (on updating) after a shorter inter-updating period will leave proportionally (to the shortening of the period) more tokens in the common pool.

We then establish the policy that when a flow runs out of its allocated tokens (line 73 in Algorithm 2, Figure 2.2.2), it attempts to use any left over in the remaining pool of tokens (lines 74 and 75). Therefore, in the event of a flow running out of tokens because of a misprediction on its number of packets, there will likely be tokens left over in the common pool and the flow will still be able to get its fair allocation.

[15] Below its configuration setting

We further assume that the average lifetime of a flow is significantly shorter than the average interarrival time for new connections. In other words, the vast majority of the times, the token allocation for a flow will not be affected by the arrival of new flows. This valid assumption makes our practical solution a very efficient one.

Even though our approach to dealing with new connections is efficient and appropriate given the characteristics of the flows, other alternative approaches could be used to allocate tokens for incoming connections. One such approach could consist on "borrowing" tokens from the future common pool in order to protect the new connections until the next updating process. Nonetheless, we have not evaluated this alternative, and do not make any remarks about its feasibility or efficiency.

The complete algorithm for the TCP Friendly marker token allocation process is presented in Figure 2.2.1. Figure 2.2.2 shows the correspondent packet marking behavior.

Algorithm 1: TCP Friendly Marker Token Allocation (i.e. Updating Process)

```
1  T = The Marking Interval
2  N = Number of flows
3  M = Number of Tokens (in bytes) available for the N flows in T seconds
4  W_i = Window Estimate for Flow i
5  E_i = Smoothed estimate (α=0.9) of the number of bytes flow i sends in the marking interval T
6  S_i = Number of consecutive OUT bytes between every two IN bytes for flow i
7  P_i = Number of bytes flow i sent on the previous interval
8  in_count_i = Allocation of IN tokens for flow i in time T seconds
9  left_over_toks = Number of remaining tokens at the end of the complete updating process
10 available = Number of available tokens per interval after small window flows get their share

   Step I
11 for each flow i {
12    E_i = α P_i + (1-α) E_i ;                  /* Get byte estimates for all flows */
13    in_count_i =  0 ;                          /* Reset allocated tokens */
14    if (W_i  < k){                             /* If tiny flow (window less than k) (k defaults to 4) */
15       available - = M - E_i ;                 /* Calculate toks left if all tiny flows get their max  */
16       tiny_list = append(i); }                /* Append tiny flow into the list of tiny flows */`
17    else                                       /* If not-tiny flow */
18       not_tiny_list = append(i); }            /* Append not-tiny flow into list of not-tiny flows */

   Step II
19 if(available <=0){                            /* If no tokens left for not-tiny flows */
20    for each flow i in not_tiny_list           /* If not tiny flow */
21       in_count_i =  0 ;                       /* Do not allocate tokens */
22    fair_allocation (tiny_list , M); }         /* Perform fair allocation on all tiny flows */
23 else{                                         /* If tokens left for not-tiny flows */
24    for each flow i in tiny_list               /* If tiny flow */
25       in_count_i = E_i ;                      /* Allocate max number of tokens */
26    fair_allocation (not_tiny_list , available);}  /* Perform fair allocation on all not-tiny flows */

   Function definition                           /* Function performs fairest allocation */
27 fair_allocation (list, toks){                 /* Allocate toks among all flows in list */
28    num_flows = sizeof (list);                 /* Get how many flows to distribute among */
29    num_full = 0;                              /* Reset the number of flows full to zero */
30    while ((toks!=0) || (num_full!=num_flows)){  /* While there are toks left or flows aren't full */
31       fair_share = (toks/(num_flows- num_full));  /* Calculate fair share on this round */
32       for each flow i in list {               /* Consider each flow in the list */
33          if( in_count_i < E_i){               /* If the flow is not already full */
34             in_count_i += fair_share;         /* Give fair share to the flow */
35             toks -= fair_share;               /* Discount toks just given from the total */
36             if (in_count_i > E_i ){           /* If flow is has now more than it needed */
37                toks+= in_count_i - E_i;       /* Take away any extra from the flow */
38                in_count_i = E_i ;             /* Set the flow to be full */
39                num_full++; } }                /* Acknowledge there is one more flow full */
40          if (in_count_i != 0)                 /* If the flow has at least 1 token */
41             S_i  = (E_i / in_count_i) – 1 ; }}  /* Calculate appropriate spacing */
42    left_over_toks = toks ; }                  /* Record the number of remaining tokens */
```

Figure 2.2.1

Algorithm 2: TCP Friendly Packet Marking

```
43  out_count_i = Per-flow variable used to count of min number of bytes that should be marked as OUT
44  temp_out_count_i = Per-flow variable used count of number of OUT bytes so far (reset to 0 on update)
45  last_time = Time of last update
46  interval_time = Inter-updating time

47  if(packet not TCP){                              /* If packet does not contain a TCP segment */
48     mark packet as OUT ;                          /* Mark packet as OUT */
49     exit ;}                                       /* Do no process any further */
50  now=gettime( ) ;                                 /* Get the time of arrival */
51  if(now – last_time >= interval_time){            /* If it is time to reallocate tokens */
52     updating process ;                            /* Execute the updating process */
53     last_time = gettime( ) ;}                     /* Record time of updating for later comparison */
54  if(packet is SYN){                               /* If packet indicates new flow*/
55     insert flow in tree ;                         /* Insert new flow information into tree of flows */
56     updating process ;                            /* Reallocate tokens to account for new flow */
57     last_time = gettime ( ) ;                     /* Record time of updating for later comparison */
58     mark packet as IN ;                           /* IMPORTANT: mark SYN as IN */
59     exit ;}                                       /* Do not process any further*/
60  if(packet is FIN){                               /* If packet indicates end of flow */
61     remove flow from tree ;                       /* Remove all kept information about that flow */
62     mark packet as IN ;                           /* IMPORTANT: mark FYN as IN */
63     exit ;}                                       /* Do not process any further */
64  flow.cwnd = packet.cwnd ;                        /* Update the congestion window for the flow */
65  flow.interval_bytes += packet.bytes ;            /* Add pack size to this flow bytes in this interval */
66  if(flow.max_seq >= packet.seq_num){              /* If this is a retransmission */
67     if(flow.in_count > packet.bytes){             /* If there are sufficient tokens */
68        mark packet as IN ;                        /* Mark packet as IN */
69        flow.in_count -= packet.bytes ;            /* Discount the tokens used for this packet */
70        exit ;}}                                   /* Do not process any further */
71  else                                             /* If not a retransmission */
72     flow.max_seq = packet.seq_num ;               /* Update the largest sequence sent */
73  if(flow.in_count < packet.bytes){                /* If there are no available tokens */
74     if(left_over_toks >= packet.bytes){           /* If there are any left from the last update (line 42) */
75        mark packet as IN ;                        /* Mark packet as IN */
76        left_over_toks -= packet.bytes ;}          /* Discount amount used from the remaining pool */
77     else                                          /* If none left from the last update */
78        mark packet as OUT ;                       /* Mark packets as OUT */
79     exit ;}                                       /* Do not process any further */
80  if(flow.S == 0){                                 /* If spacing for the flow is 0 */
81     mark packet as IN ;                           /* Mark packet as IN */
82     flow.in_count -= packet.bytes ;               /* Discount the tokens used for this packet */
83     exit ;}                                       /* Do not process any further */
84  if(first flow packet since update){              /* If packet is the first for the flow since last update */
85     mark packet as IN ;                           /* Mark packet as IN */
86     flow.in_count -= packet.bytes ;               /* Discount the tokens used for this packet */
87     flow.out_count = (flow.S * packet.bytes) ;    /* Calculate number of OUT bytes before IN again */
88     exit ;}                                       /* Do not process any further */
89  if(flow.temp_out_count >= flow.out_count){       /* If sufficient bytes have been marked as OUT */
90     mark packet as IN ;                           /* Mark packet as IN */
91     flow.in_count -= packet.bytes ;               /* Discount the tokens used for this packet */
92     flow.temp_out_count = 0 ;                     /* Reset count of OUT bytes since last IN packet */
```

```
93      flow.out_count = (flow.S * packet.bytes) ;   /* Calculate number of OUT bytes before IN again */
94      exit ; }                                    /* Do not process any further */
95    else{                                         /* If not enough bytes have been marked as OUT */
96      mark packet as OUT ;                        /* Mark packets as OUT */
97      flow.temp_out_count += packet.bytes ;       /* Update count of OUT bytes since last IN packet */
98      exit ; }                                    /* Do not process any further */
```

Figure 2.2.2

The token allocation mechanism (Algorithm 1) consists on two major steps. We highlight the following major operations from each one of them.

Step I (lines 11 through 18)

- **Calculate the per-flow expected amount of bytes (line 12)**: The number of bytes that every flow is expected to send during the next time interval is calculated as a weighted average of the previous estimate (10% weight) and the number of bytes sent during the immediate interval preceding the updating process (90% weight).

- **Reset the tokens allocated in the past (line 13)**: We take away any remaining tokens that the flows might still have from the interval that just ended.

- **Identify flows as tiny or not-tiny (lines 14 through 18)**: We append the flows to two different lists depending on whether or not they are tiny flows. A tiny flow is one that has a value of less than k for its sending congestion window, at the time of update. The comments on line 13 mention that the default for k is 4. We approximate each flow's window size by considering the last acknowledgement seen from that flow. We subtract the value of the most recently acknowledged byte from the value of the last byte sent, and divide the resulting number by the segment size established at the start of the connection.

- **Calculate the tokens needed by all tiny flows (line 15)**: We subtract, from the total amount of tokens, the amount needed to fulfill the estimates of all the tiny flows. This is to find out whether we would have any tokens left

after protecting all the tiny flows, by giving them as many tokens as they need.

Step II (lines 19 through 26)

- **Determine whether we can fully serve the tiny flows (line 19)**: If there are not enough tokens for all the tiny flows, we do not even consider giving any tokens to the not-tiny flows.

- **Serve not-tiny flows according to the availability of tokens (lines 21 and 26)**: In the case of not (or just) enough tokens for the tiny flows, we do not give tokens to their not-tiny counterparts (line 21). If all tiny can be fully served, then we fairly divide the rest among the not-tiny ones (line 26).

- **Serve tiny flows according to the availability of tokens (lines 22 and 25)**: If all tiny flows can be fully served, then we simply set their allocation of tokens to the amount they are expected to use (line 25). Otherwise, we fairly divide all the tokens exclusively among the tiny flows (line 22).

The fair allocation process is exclusively performed on either the tiny flows or the not-tiny ones. Its purpose consists on giving all the flows, on which it is called upon, the same number of tokens. It should be done without using any more tokens that the ones given to it as argument, and without allowing any flows to have any more tokens that it is expected to use. We now highlight the major operations of this allocation.

Fair Allocation Function (lines 27 through 42)

- **Determine the number of flows to consider in the sharing (lines 28 and 29)**: We find out how many flows we need to allocate tokens to. We initially assume that all of the flows are empty and wait to be given tokens.

- **Do an iterative equal sharing among flows not full (line 30 through 35)**: We set up an iterative loop (of rounds) that will keep on giving tokens for as long as there any left, and there are flows that need them. At every

round, we calculate the equal amount to give to each flow (not full). A flow is considered full if it has as many tokens as it is expected to use.

- **Subtract extra tokens from the flows and make them available for sharing (lines 36 through 39)**: We check if the flows have be given more than they need, and subsequently discount the extra amount from their allocations and add such amount to the total tokens available. We also mark these flows as full as to not consider them in future rounds of the allocation.

- **Update the byte spacing to be enforced in the marking (lines 40 and 41)**: As we give more tokens to the flows, we recalculate the marking byte spacing as a function of the estimate of their traffic and the amount of tokens they currently have.

- **Record the number of tokens left after the complete allocation (line 42)**: This is done in order to use these tokens (if any) in the event of a flow running out of the number allocated to it.

The marking process (Algorithm 2) is done according to the allocation we have already detailed. We now draw attention to the following major operations of the marking procedure.

- **Mark not-TCP packets as OUT (line 47 through 49)**: When a packet comes into the marker, we first check if it contains a TCP segment. If it does not, we simply mark it as OUT and put it in the forwarding queue.

- **Update token reallocation on the time of packet arrival specifying interval expiration (lines 50 through 53)**: On packet arrival, we record the time in order to determine whether it is the moment for reallocation of tokens. If it is time, we call up the updating process and then record the

ending time of the update, for further comparison, to determine when to update next.

- **Mark SYN packets as IN and process the start of a new connection (lines 54 through 59)**: If the packet contains a SYN segment, we insert the new flow information into the flow-state data structure and call up the update routine. We then mark the packet as IN and put it inside the queue for forwarding.

- **Mark FIN packets as IN and process the end of an existing connection (lines 60 through 63)**: If the packet contains a FIN segment, we remove the flow information from the data structure, mark the packet as IN, and put the FYN segment in the forwarding queue.

- **Update flow's congestion window and byte traffic for the interval (lines 64 and 65)**: We update the congestion window for the flow and the number of bytes seen from that flow during this interval. This information will be use during the next reallocation of tokens for all the flows.

- **Mark retransmissions as IN (lines 66 through 72)**: If the packet contains a retransmission, we check for the availability of tokens for that flow, and mark the packet as IN. If the packet does not contain a retransmission, we update the highest sequence number for the flow, in order to identify future retransmissions.

- **Mark packet as IN or OUT according to the availability of tokens and enforcing the correspondent byte spacing (lines 73 through 98)**: If the flow does not have enough tokens and neither does the common pool of possible remaining tokens (from the last updating process), we simply mark the packet as OUT (lines 73 through 79). If there are tokens, and the spacing of bytes to enforce is zero, we mark the packet as IN (lines 80 through 83). If there is some spacing to enforce, we mark the first packet from the flow

(during this interval) as IN and calculate the number of bytes (from that flow) that have to be marked as OUT before we can mark another of its packet as IN (lines 84 through 88). This last number is kept in *flow.out_count* (line 87). We then continue to mark packets as OUT until the number of bytes marked as OUT is at least equal to *flow.out_count* (lines 95 through 98), at which time we mark the next packet as IN, and recalculate *flow.out_count* (lines 89 through 94).

The implementation of the marker was included in the kernel code of an edge router, running Linux 2.2.10. The marker was configured as a module, which allowed us to make changes to the algorithm and debug the implementation without having to recompile the whole kernel but instead inserting the new module into the operations of the kernel. The communication between the kernel code, that included setting of parameters and obtaining statistics, was performed using the netlink interface provided by the operating system. We incorporated new functionality to the "tc"[16] program in order to be able to handle the new traffic-conditioning element. More specifically, the parameters to be handled were the *limit* (in bytes) of the queue holding the packets to be marked, the number of *tokens* (in bytes) to make available for a single time interval and the correspondent time *interval* (specifiable in seconds, milliseconds or microseconds). Similarly we are able to obtain (at any instance of time) the number of bytes marked as "IN" and the number of bytes marked as "OUT", as well as some other statistics used for debugging of the implementation.

In order to keep the flow state information we built our own data structure aimed to minimize insertion and retrieving time. Specifically, we implemented a binary tree structure where every node in the tree has an ID field containing the source port number of one or more flows in the marking system at any time. In addition every tree node contains a linked list where the other three unique identifiers of a flow

[16] [WHK99]

(source address, destination address, and destination port) are included into every node of the linked list.

A graphical representation of this data structure is included below, in Figure 2.2.3.

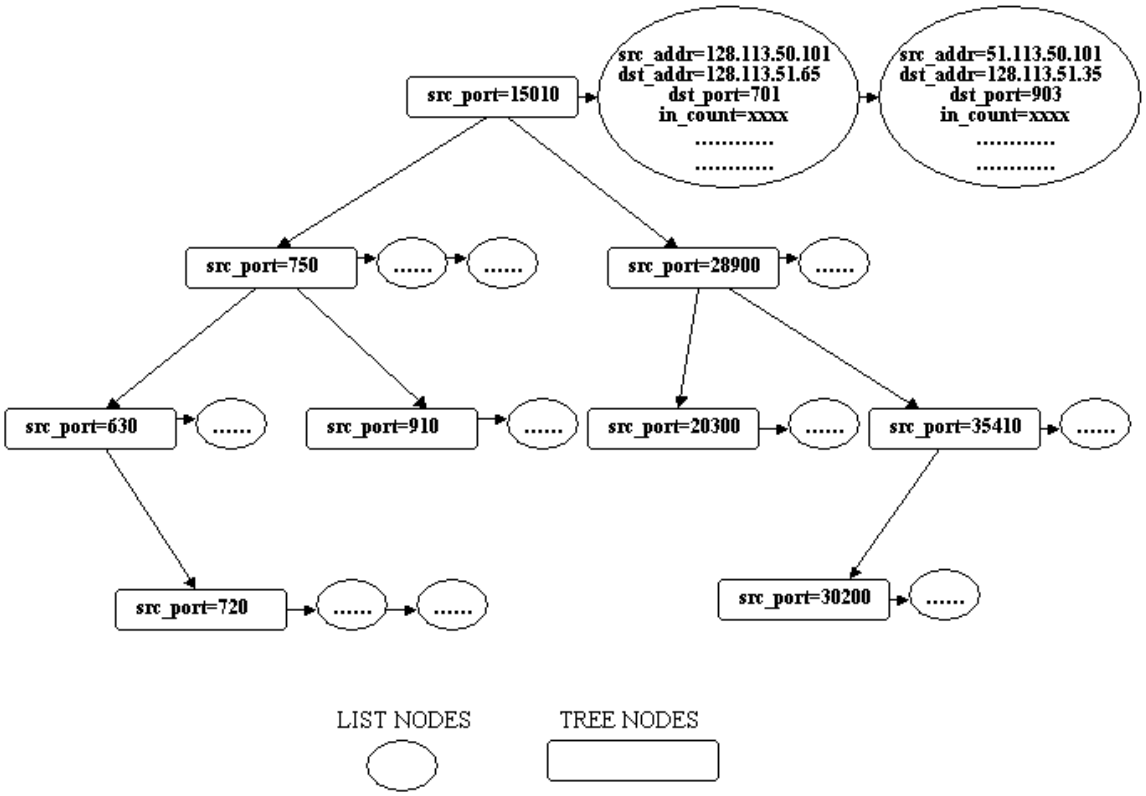

Figure 2.2.3: Graphical representation of flow data structure

We make the valid assumption that at any state of the marking system, the amount of unique source port numbers is always less than the amount of any other of the three identifiers. Therefore as more flows are introduced into the marking system, the number of tree nodes grows much faster than the number of list nodes in any single list. This, of course, keeps the retrieval of flow information, as well as the insertion and removal of flows, to be an O(log n) operation, which does not degrade the efficiency of the gateway when the marking process is implemented.

Finally, we value the superiority of the TCP Friendly marker with respect to a simple token bucket marker, as considered by previous research in the area[17]. The issues addressed by the TCP Friendly marker could not be solved with the use of the token bucket marker, and our experimental results imply that the targeting of these issues are in fact a key element on achieving better than best effort for the Internet.

2.3 FRIO: A New Buffer Management Scheme

In earlier discussions on differentiated services, we established the need for buffer management algorithms at selected points in the core of the network, where the multiplexing of traffic gives rise to congestion. At these bottleneck gateways, the utilization of buffers approaches a constant 100%, and mechanisms like RED[18] are implemented to support random probabilistic drop in order to avoid the exhaustion of buffer resources and lessen the performance degradation that policies like tail drop have on the network.

Nonetheless, it has been shown[19] that basic RED presents a series of weaknesses in its operation. One such weakness corresponds to the idea that the unbiased proportional dropping contributes to unfair link sharing for fragile connections. The basic fact that all flows experienced the same loss probability at the congested gateway inherently means that even flows using less than their fair share will still have their packets dropped. To overcome this issue, a variant of RED, known as FRED, has been proposed[20] in the past. This new buffer management discipline introduces preferential drop for packets belonging to flows that have a large number of them buffered at the congested gateway. FRED uses a variable called min_q that corresponds to the minimum number of packets that any flow is allowed in the queue before the probabilistic dropping behavior of RED starts even considers dropping

[17] [Sah99]
[18] [FJ93]
[19] [May99]

[20] [LM97]

packets from a flow. It also introduces max_q as being the maximum number of packets a flow is allowed in the buffer before further of its packets are deterministically dropped. Finally, it maintains an estimate of the per-flow packet average in the queue (*avgcq*) as well as the number of times (*strike*) that a flow has had it packets dropped because of exceeding max_q. Simulation analysis of FRED has shown that it is often fairer than RED and results into a lower goodput coefficient of variation for distinct types of flows that traverse a congested gateway.

Also, as suggested by the ongoing discussion, the need for a core buffer management that supports preferential drop of "OUT" packets over "IN" packets, is essential for the overall differentiated services dynamics of the approach here proposed. The original work on the TCP Friendly marker established RIO ("RED with In and Out") as the buffer management algorithm used in the core for their simulations. We instead have combined the benefits of FRED with the need for RIO into a new buffer management discipline we call FRIO (short for "Flow RIO").

We have made some modifications to the original version of FRED that we considered[21] resulted into better performance and fairer queuing for low rate TCP flows. These modifications are summarized as follows:

- **Use of instant queue size to approximate the per-flow number of packets (*avgcq*) in the buffer (line 126 in Algorithm 3, Figure 2.3.1)**: The original FRED uses average queue size in order to approximate this value. We encountered that the calculation of *avgcq*, required the use of the instantaneous queue length in order to achieve a fairer performance for slow TCP connections.

- **Estimation of average queue length exclusively on arrival of packets (lines 110 and 114 in Algorithm 3)**: The original FRED claimed that the

[21] Through experimentation with our own Linux implementation

average queue length must be calculated both on arrival and departure of packets into and from the buffer. They argue that the omission of recalculating average queue length on departure resulted into low link utilization due to unnecessary packet drops. We encountered that the standard implementation of RED for Linux[22], avoided this problem by "hand" modeling the arrival of packets into the queue (for periods of no arrivals). They use this model for the calculation of average queue length, done exclusively on arrival. We exploited this approach and do not recalculate average queue length on departure, differing from the original FRED.

We have subsequently merged the dynamics of this new FRED with the dynamics of the preferential dropping achieved by RIO. The resulting algorithm, FRIO, exhibits the following characteristics

- **Support of probabilistic preferential dropping, allowing the transformation of a single buffer into two virtual queues that materialize the differentiation of the servicing of two distinct classes of packets.**

- **Support of deterministic preferential dropping, allowing the performance of fragile and slow TCP connections not to degrade on the presence of non-adaptive connections that are unresponsive to gateway congestion.**

In order to better understand the relationship between the buffer management disciplines here discussed, we make the simple analogy that *FRED is to RED as FRIO is to RIO*.

[22] By Alexey Kuznetsov. See http://lxr.linux.no/source/net/sched/sch_red.c

The complete algorithm for FRIO is presented in Figure 2.3.1 shown below.

Algorithm 3: FRIO Buffer Management

```
99  max_q = Maximum allowed per-flow queue packet size
100 min_q = Minimum per-flow queue packet size before probabilistic drop is considered (default is 4)
101 avgcq = Average per-flow queue packet size
102 Nactive = Number of flows with packets in the queue
103 minthresh_IN = RED min threshold for IN packets (in bytes) (average packet size is 1024 and tunable)
104 minthresh_OUT = RED minimum threshold for OUT packets (in bytes)
105 maxthresh_IN = RED maximum threshold for IN packets (in bytes)
106 maxthresh_OUT = RED maximum threshold for OUT packets (in bytes)
107 temp_qlen = Temp variable to hold the number of packets in the queue for a particular flow
108 temp_strike = Temp variable, holds number of times a flow is penalized for too many packets in buffer

    On enqueuing:
109 If (packet marked as IN){                          /* If this is an IN packet */
110    avg_qlen = RED(avg_qlen_IN) ;                    /* Calculate usual RED avgqlen for IN packs */
111    minthresh = minthresh_IN ;                       /* Set the minthresh to use to be the IN one */
112    maxthresh = maxthresh_IN ;}                      /* Set the maxthresh to use to be the IN one */
113 else{                                               /* If this is an OUT packet */
114   avg_qlen=RED(avg_qlen_IN)+RED(avg_qlen_OUT);      /* RED avgqlen for both IN and OUT packs */
115    minthresh = minthresh_OUT ;                      /* Set the minthresh to be the OUT one */
116    maxthresh = maxthresh_OUT ;}                     /* Set the maxthresh to be the OUT one */
117 if(packet is TCP){                                  /* If this contains a TCP segment */
118    find flow in the tree                            /* Find state of flow to which packet belongs */
119    if(flow not found){                              /* If this is a flow not active in the queue */
120       temp_qlen = 0 ;                               /* Set its number of packs in queue to zero */
121       temp_strike = 0 ;}                            /* Set its number of strikes to zero */
122    else{                                            /* If this is an active flow in the queue */
123       temp_qlen = flow.packets  ;                   /* Get its number of packs in the queue */
124       temp_strike = flow.strikes; }                 /* Get its number of strikes */
125    max_q = (minthresh / average packet size) ;      /* Calculate max_q as minthresh for packets */
126    avgcq = (instant qlen) / (Nactive | 1) ;         /* Set avgcq to packs in queue per active flow */
127    if(((temp_qlen >= max_q) ||                      /* If qlen will be larger than max allowed or */
128      ((temp_qlen > avgcq)&&(temp_strike > 1)))      /* qlen will be larger than per-flow avg and */
129      &&(flow exists in the tree)){                  /* this flow has been penalized and is active */
130       flow.strike ++ ;                              /* Penalize the flow */
131       goto DROP routine ;}}                         /* Drop the packet */
132 if(avg_qlen < minthresh){                           /* If avg_qlen is below minthresh */
133    goto ENQUEUE routine ;                           /* Enqueue the packet */
134 if(avg_qlen >= maxthresh){                          /* If avg_qlen is on or above maxthresh */
135    if((packet is TCP)&&((temp_qlen < 2) ||          /* If flow qlen is only 1 packet or less than */
136        ((temp_qlen < 4)&&(temp_strike <1))))        /* 3 packets but has not been penalized */
137       goto ENQUEUE routine ;                        /* Enqueue the packet */
138    else                                             /* Otherwise */
139       goto DROP routine ;}                          /* Drop the packet */
140 else{                                               /* If avg_qlen between min and max thresh */
141    if((packet is TCP)&&(temp_qlen <= min_q)         /* If flow qlen is less than the per-flow min */
142     &&((temp_qlen < avgcq) || (temp_qlen <= 2)))    /* and less than per flow avg or 2 */
143       goto ENQUEUE routine ;                        /* Enqueue the packet */
```

```
144     else{                                   /* Otherwise */
145        calculate RED probability of drop;   /* Do normal probabilistic drop as RED */
146        if (probability says we should drop) /* If it came out drop */
147           goto DROP routine ;               /* Drop the packet */
148        else                                 /* If it came out enqueue */
149           goto ENQUEUE routine ;}}          /* Enqueue the packet */

     ENQUEUE routine:                          /* This routine process as we enqueue packets */
150  enqueue the packet ;                      /* Simply enqueue the packet in the buffer */
151  if(flow does not exist in the tree){      /* If packet is from flow not active in queue */
152     insert flow in the tree ;              /* Insert flow info into the tree */
153     Nactive ++ ;                           /* Update the number of active flows */
154     flow.packets = 1 ;}                    /* Account for the packet we just enqueued */
155  else                                      /* If packet is from active flow */
156     flow.packets ++ ;                      /* Account for the packet we just enqueued */
157  exit ;}                                   /* Do not process any further */

     DROP routine:                             /* This routine process as we drop packets */
158  drop the packet ;                         /* Simply drop the packet */
159  exit ;                                    /* Do not process any further */

     On dequeuing:
160  if(packet is TCP){                        /* If this contains a TCP segment */
161     find flow in the tree                  /* Find state of flow to which packet belongs */
162     flow.packets -- ;                      /* Account for packet we are now dequeueing */
163     if (flow.packets == 0){                /* If flow is becoming not-active in queue */
164        Nactive -- ;                        /* Update the number of active flows */
165        remove flow from the tree ;}}       /* Remove flow info from the tree */
166  dequeue the packet ;                      /* Simply dequeue the packet */
167  exit ;                                    /* Do not process any further */
```

Figure 2.3.1

The FRIO buffer management scheme performs operations on the enqueuing and dequeuing of packets into and from the buffer. We now highlight the major operations of the scheme.

Enqueuing (lines 109 through 149)

- **Determine priority of the packet and consider thresholds and average queue length accordingly (lines 109 through 116)**: We find out if the packet is either IN or OUT, and set the RED thresholds to be the respective ones for IN or OUT packets. We also calculate the average queue length as the usual RED but in the following manner:

- If the packet is marked as IN, we set the average queue length to the average length of the virtual queue of IN packets.
- If the packet is marked as OUT, we set the average queue length to the sum of the average length of the virtual queue of IN packets and the average length of the virtual queue of OUT packets. This is in practice the same as considering the average queue length of the entire buffer.

• **Perform core of "Flow operations" exclusively on TCP packets (lines 117 through 131)**: We intuitively understand that the inherited FRED functionality can only be applied to packets containing TCP segments. We give all other packets the usual RED treatment, with the distinction of using either IN or OUT parameters (depending on the nature of packet) in determining whether to drop or enqueue.

• **Penalize unresponsive connections (lines 125 through 131)**: We calculate the maximum number of packets that any flow is simultaneously allowed in the buffer as a function of the RED *minthresh* for the type of packet (line 125). We drop any packets that attempt to violate this upper bound and record the fact that an active[23] connection has tried to break the established bound. We also approximate the per-flow average number of packets in the buffer, and drop any packets that would make a penalized flow go beyond this average (lines 126 through 131).

• **Protect fragile and slow TCP connections (lines 135 through 137 and 141 through 143)**: In the event of deterministic drop by RED (i.e. average queue size larger than *maxthresh*) we introduce an exception to the rule that favors flows with a small number of packets in the queue. Specifically, if the packet belongs to a flow that has less than 2 packets in the queue, we do not drop the packet. Similarly, if it belongs to a flow with less than 4 packets,

[23] An active connection, in this context, is one that currently has at least one packet in the queue.

that has not been penalized (since last becoming active), we also enqueue the packet (lines 135 through 137).

In the same fashion, but a little more aggressively, we also protect flows when RED is probabilistically dropping packets (i.e. average queue size is between *minthresh* and *maxthresh*). In our particular case, we never drop the packet if it belongs to a flow with less than 5 packets in the queue. To that effect we do not even compute the probability of drop for a packet in such circumstances (lines 141 through 143).

- **Maintain status of flow activity (lines 150 through 157)**: Every time we choose to enqueue a TCP packet, we record information about the flow it belongs to. Specifically, we increment by 1 the number of packets in the buffer for that flow. In the event of the packet belonging to a flow with no packets in the queue, we insert the flow into the flow data structure and update the number of active flows accordingly.

Dequeuing (lines 160 through 167)

- **Update flow status information (lines 160 through 167)**: On dequeuing of packets, we simply decrement by 1 the count of the packets that the flow has inside the queue. Incidentally, if the flow has no more packets in the queue, we remove its information from the tree flow data structure and say that the flow has become inactive.

The implementation of FRIO was done using the Linux 2.2.10 kernel in a similar fashion as with the TCP Friendly marker. We based our work on an existing implementation of GRED[24] (Generic RED), which is included in the distribution of the kernel. GRED is able to support up to 16 preferential drop priorities and exhibits a switch (referred to as "grio") that enables the calculation of the average queue length

[24] By Jamal Hadi [WHK99]

of a single priority to include the average queue lengths of all the priorities above it. This goes in concordance with RIO that establishes that the calculation of the average queue length for OUT packets should include the average queue length of IN packets, while the opposite not being the case. In order to maintain flow information we used the same data structure we had developed for the TCP Friendly marker, which has been already discussed.

One important final consideration is worth mentioning in our discussion of buffer management algorithms. All of the schemes discussed here are fundamentally just variants of RED. They evolve with a focus on addressing the issues of preferential dropping of classes (IN and OUT) and fairness among types of TCP connections (fragile and unresponsive). Nonetheless there are still some other well-identified weaknesses of RED that are carried on by these evolutions One such difficulty, corresponds to the lack of proven guidelines in the setting of RED-like parameters[25] such as the minimum threshold, maximum threshold, buffer size, and maximum probability of drop. Furthermore, it has been shown that the performance of RED is highly sensitive to its parameters and the characteristics of the traffic flowing through it, and in many cases it performs drastically worse than the simple and stateless tail dropping mechanism. In our experimentation we will consider the fundamental cases where “appropriate” and “inappropriate” parameters are used for our RED-based buffer management disciplines, in order to discuss the resulting differences in performance of the two approaches to parameter setting. We will also provide a better insight as to what we consider “appropriate” and “inappropriate”.

[25] [May99]