ABSTRACT

KHARE, SHRIKRISHNA G. Testbed Implementation and Performance Evaluation of the Tiered Service Fair Queuing (TSFQ) Packet Scheduling Discipline. (Under the direction of Professor Dr. George Rouskas).

In packet-switched networks, the scheduling algorithm implemented by the routers must possess three important properties: *fairness*, to provide isolation among competing flows and ensure that each flow receives its fair share of the link bandwidth; *bounded delay*, so as to guarantee a bounded end-to-end delay to interactive applications; and *low complexity*, so as to be possible to operate at wire speeds even for large number of flows. Although many fair queuing disciplines have been proposed, the best among them have worst-case time complexity of $O(\log n)$ for a link with n flows.

Tiered Service Fair Queuing (TSFQ), a new queuing discipline, has been proposed to achieve packet sorting and virtual time computation in time that is independent of the number of flows. TSFQ exploits two widely observed characteristics of the Internet, namely, that service providers offer some type of tiered service with a small number of service levels, and that a small number of packet sizes dominate. Consequently, TSFQ maps the competing n flows to p service levels where p is a small constant, and uses a special queuing structure that eliminates the need to sort most packets.

As part of this thesis work, we implement the WF^2Q+ discipline and various TSFQ variants in the Linux kernel as separate loadable modules, and we investigate their relative performance over a small testbed. Our experimental results indicate that TSFQ closely emulates previously proposed fair queuing disciplines.

The main conclusion of our work is that TSFQ is a viable packet scheduler that can be used in networks with heavy traffic loads to achieve fairness in constant time.

Testbed Implementation and Performance Evaluation of the Tiered Service Fair Queuing (TSFQ) Packet Scheduling Discipline

by
Shrikrishna Khare

A thesis submitted to the Graduate Faculty of
North Carolina State University
in partial fulfillment of the
requirements for the Degree of
Master of Science

Computer Science

Raleigh, North Carolina

2008

APPROVED BY:

Dr. Rudra Dutta

Dr. Injong Rhee

Dr. George Rouskas
Chair of Advisory Committee

BIOGRAPHY

Shrikrishna Khare was born on June 26, 1983. He received Bachelor of Engineering in Computer Engineering from University of Pune in July, 2004. Later, he worked as a Member of Technical Staff in Persistent Systems, Pune, India for 2 years. With the defense of this thesis, he will receive Master of Science in Computer Science from North Carolina State University in May, 2008.

ACKNOWLEDGMENTS

I would like to thank my advisor, Dr. Rouskas, without his support, encouragement and timely guidance this would have never been possible. I would also like to thank Dr. Dutta and Dr. Rhee for serving on my thesis committee. I would like to acknowledge National Science Foundation for supporting this research under grant CNS-0434975.

I would also like to thank my parents and my sister, Shruti, for shaping me into the individual that I am today.

TABLE OF CONTENTS

LIST OF TABLES vi

LIST OF FIGURES vii

1 Introduction 1
- 1.1 Packet Scheduling 1
- 1.2 Organization of Thesis 2

2 Queuing disciplines 3
- 2.1 Fundamentals of Queuing disciplines 3
  - 2.1.1 The law of conservation 3
  - 2.1.2 Desired properties of a scheduler 3
  - 2.1.3 Max-min weighted fair share allocation 4
  - 2.1.4 Generalized process sharing (GPS) 4
- 2.2 Round robin emulating GPS 5
- 2.3 Fair Queuing 5
  - 2.3.1 WFQ 5
  - 2.3.2 Concept of Worst-Case Fair Index (WFI) 6
  - 2.3.3 WF^2Q emulation of GPS 7
  - 2.3.4 WF^2Q+ 8
  - 2.3.5 A note on Virtual time 9
- 2.4 Tiered Service Fair Queuing (TSFQ) 9
  - 2.4.1 TSFQ for fixed packet sizes (TSFQ-F) 10
  - 2.4.2 Modified TSFQ for fixed packet sizes (TSF^2Q-F) 12
  - 2.4.3 TSFQ for variable packet sizes (TSFQ-V) 13
  - 2.4.4 Modified TSFQ for variable packet sizes (TSF^2Q-V) 15

3 Linux Implementation 16
- 3.1 Queuing Disciplines in Linux 16
- 3.2 Configuring queuing disciplines 20
  - 3.2.1 Associating weights 20
  - 3.2.2 RTNETLINK Sockets 21
  - 3.2.3 Steps involved in configuring queuing discipline 22
- 3.3 Organization of tc code 24

4 Specific Requirements 26
- 4.1 Configuring WF^2Q+ 26
- 4.2 Configuring TSFQ variants 27

5 Design and Implementation **29**
5.1 User Perspective 29
5.2 System (Kernel) Perspective 30
5.2.1 Per flow queues: Queues from FIFO implementation 30
5.2.2 Support to look up Queue head: Peek functionality 31
5.2.3 WF^2Q+ 32
5.2.4 TSFQ-F 33
5.2.5 TSF^2Q-F 35
5.2.6 TSFQ-V 37
5.2.7 TSF^2Q-V 38

6 Numerical Results **40**

7 Summary and Future Work **52**
7.1 Summary 52
7.2 Future Work 52

Bibliography **53**

Appendices **54**
Appendix A. Experimental Setup 55
Appendix B. Abbreviations 59

LIST OF TABLES

Table 3.1 Table iproute/tc design 25

LIST OF FIGURES

Figure 2.1 TSFQ for fixed packet size (TSFQ-F) 10

Figure 2.2 Modified TSFQ fixed packet size (TSF^{2}Q-F) 12

Figure 2.3 TSFQ variable packet size (TSFQ-V) 14

Figure 3.1 Packet forwarding and sending 16

Figure 3.2 Applications communicating over network 17

Figure 3.3 Kernel Networking Stack 20

Figure 3.4 RTNETLINK sockets 22

Figure 3.5 Configuring Queuing Disciplines 23

Figure 3.6 iproute/tc design 25

Figure 5.1 Qdisc creation using iproute/tc 30

Figure 5.2 Circular queue for ineligible packets 36

Figure 6.1 Testbed setup 40

Figure 6.2 Throughput WF^{2}Q+ - Variable sized packets - 2 Flows 42

Figure 6.3 Throughput WF^{2}Q+ - Variable sized packets - 3 Flows 42

Figure 6.4 Throughput WF^{2}Q+ - Fixed sized packets - 4 Flows - Scenario I 43

Figure 6.5 Throughput WF^{2}Q+ - Variable sized packets - 4 Flows - Scenario I 43

Figure 6.6 Throughput WF^{2}Q+ - Fixed sized packets - 4 Flows - Scenario II 44

Figure 6.7 Throughput WF^{2}Q+ - Variable sized packets - 4 Flows - Scenario II 44

Figure 6.8 Throughput WF^{2}Q+ - Variable sized packets - 22 Flows - Continuous run . 45

Figure 6.9 Throughput WF^{2}Q+ - Variable sized packets - 22 Flows - Flow 1 and 2 terminated after 10 sec 45

Figure 6.10 Throughput WF^{2}Q+ - Variable sized packets - 32 Flows - Continuous run 46

Figure 6.11 Throughput TSFQ-F - Fixed sized packets - 4 Flows - Scenario I 46

Figure 6.12 Throughput TSFQ-F - Fixed sized packets - 4 Flows - Scenario II 47

Figure 6.13 Throughput TSF^{2}Q-F - Fixed sized packets - 4 Flows - Scenario I 47

Figure 6.14 Throughput TSFQ-V - Variable sized packets - 4 Flows - Scenario I 48

Figure 6.15 Throughput TSFQ-V - Variable sized packets - 4 Flows - Scenario II. 48

Figure 6.16 Throughput TSFQ-V - Variable sized packets - 32 Flows - Continuous run 49

Figure 6.17 Throughput TSF^{2}Q-V - Variable sized packets - 4 Flows - Scenario I 49

Figure 6.18 Throughput TSF^{2}Q-V - Variable sized packets - 4 Flows - Scenario II 50

Figure 6.19 Throughput TSF^{2}Q-V - Variable sized packets - 32 Flows - Continuous run 50

Figure 6.20 Throughput TSF^{2}Q-V - Variable sized packets - 32 Flows - Continuous run - Differnt periods 51

Chapter 1

Introduction

1.1 Packet Scheduling

Multiple contenders vying for the same resource warrant some mechanism to resolve the resulting conflict. In the context of networking, we typically have a large number of flows competing with each other to go out over a single network interface card. Every time the output interface is ready to a push new packet over the network, we need to decide which flow gets service next. Thus, every multiplexed resource needs scheduling (provided the statistical fluctuations in traffic result in queuing at multiplexing point).

The choice of service order has direct bearing on the amount of delay incurred by a particular request. Also, if we run out of buffer space, then we need to decide a flow from which to drop a packet. Thus, choice of scheduling disciplines directly affects both fairness (important for best-effort flows) and performance guarantees (important for performance critical applications).

Packet schedulers can be classified into two types:

Work conserving schedulers: In these schedulers, the link is never idle when there are one or more packets waiting for service.

Non-work conserving schedulers: In these schedulers, the link may be idle even if it has packets to serve. One important reason for delaying the service is to reduce delay jitter.

Few packet scheduling schemes like Weighted Fair Queuing (WFQ) and its variants provide good fairness among flows and QoS guarantees, but have relatively high time complexity $O(log\ n)$, where n is number of contending flows in the system. Other packet scheduling techniques like Weighted Round Robin (WRR) have $O(1)$ complexity, but gen-

erally do not have good fairness or bounded delay properties.

As part of this thesis work, we have implemented a new queuing discipline: Tiered Service Fair Queuing [9] in Linux kernel and compared its performance with existing queuing disciplines. TSFQ aims at providing good fairness among flows and QoS guarantees at $O(1)$ time complexity.

1.2 Organization of Thesis

We begin by briefly explaining basic principles in scheduling. Section 2 describes Fair Queuing algorithms. It follows with description of Tiered Service Fair Queuing and its variants. In this section we define the technique and analyze its time complexity. Section 3 describes design of network stack in the Linux kernel with specific emphasis on Queuing disciplines and the role they play in kernel code.

We formally present Tiered Service Fair Queuing algorithms and its variants along with its design constructs as implemented by our work in Section 5. TSFQ internally uses the $\mathrm{WF}^2\mathrm{Q}+$ formula for virtual time computation. This section explains that design in detail as well.

Section 6 analyzes the performance of the implementation. We summarize our work in Section 7 and provide directions for future research. Appendix A provides detailed steps that should be undertaken in order to install and configure a Linux box to test our code and observe performance.

Chapter 2

Queuing disciplines

2.1 Fundamentals of Queuing disciplines

2.1.1 The law of conservation

The sum of the mean queuing delays received by the set of multiplexed connections weighted by their share of the link's load, is independent of the scheduling discipline. In other words, a scheduling discipline can reduce a particular connection's mean delay, compared with FCFS, only at the expense of another connection. The sum of delays with the FCFS scheduling is a tight lower bound whether or not the server is work conserving. More formally [10], we have that:

$$\sum_{i=1}^{N} \rho_i q_i = Constant \tag{2.1}$$

where,

λ_i : mean rate

x_i : service rate

q_i : mean waiting time at scheduler time

$\rho_i = \lambda_i x_i$

2.1.2 Desired properties of a scheduler

A packet scheduler should have following properties [8]:

- Since the schedulers are used in high speed networks, they should have low operational time complexity, preferably $O(1)$.
- The scheduler should maintain delay bounds for guaranteed-service applications.
- The scheduler must provide fairness among flows competing for the shared link (i.e. max-min share allocation as explained in Section 2.1.3).

2.1.3 Max-min weighted fair share allocation

Amongst the competing flows, some might demand larger share than others. We associate weights with flows to reflect their relative resource share. To achieve max-min fairness, the allocation should be done based on following rules:

- Resources are allocated in order of increasing demand, normalized by weight
- No source gets a resource larger than its demand
- Sources with unsatisfied demands get resource shares in proportion to their weights.

2.1.4 Generalized process sharing (GPS)

GPS is an ideal scheduling discipline which is defined as follows:

- Each connection to be multiplexed has a separate logical queue.
- GPS visits each non-empty queue in turn
- It serves an infinitesimally small amount of data every time it visits a queue.

GPS, by definition, creates max-min fair allocation. If a particular flow is idle for a while, then, then the excess bandwidth gets distributed amongst backlogged flows in proportion to their weights.

More formally, a GPS server serving N sessions is characterized by N positive real numbers ϕ_1, ϕ_2 ... ϕ_N. The server operates at fixed rate r and is work-conserving. Let $W_i(t_1, t_2)$ be the amount of session i traffic served in the interval $[t_1, t_2]$. Then, a GPS server is defined as one for which

$$\frac{W_i(t_1, t_2)}{W_j(t_1, t_2)} = \frac{\phi_i}{\phi_j}; \qquad j = 1, 2, ..., N \tag{2.2}$$

holds for any session i that is backlogged throughout the interval $[t_1, t_2]$.

GPS makes an assumption that bit by bit service is possible. In practice, however, information comes in the form of packets and network cards are equipped to deal with packets only.

2.2 Round robin emulating GPS

Round robin (RR) [11] visits each backlogged flow in turn and services the packet at the head of its queue. RR reasonably approximates GPS when all connections have equal weights and when all packets have same size. If flows have different weights, each flow is serviced in turn and in proportion to its weight. This is called as weighted RR (WRR).

However, for connections with different mean packet sizes, to serve the connection based on weights, we need to divide weights by mean packet sizes and normalize before deciding how many packets of a particular connection should be served in one round.

The difficulty in emulating GPS correctly is that WRR should know a source's mean packet size in advance. In practice, however, it may not be possible to predict the packet sizes as they may depend on many factors, e.g., the application, network interface, etc. Another serious flaw with this emulation is that WRR is fair only over 'time scales longer than a round time'. At shorter time scales, some connections may get a service share larger than others.

Thus, if a connection has a small weight or the number of connections is very large, using WRR may lead to long periods of unfairness (thereby eventually defeating the purpose of scheduling discipline).

2.3 Fair Queuing

2.3.1 WFQ

Weighted Fair Queuing schedules the packets in the order of increasing departure times had those been scheduled in GPS. However, this would make the scheduler non work conserving. To avoid this, when WFQ has to pick a packet for scheduling, it picks the first packet that would leave in the corresponding GPS system provided no additional packets were to arrive after that time [6].

As observed in [5], the relative finish order of all packets that are in the system at time t is independent of any packet arrivals to the system after time t. That is, for any two packets p and p' at time t in a GPS system, if p finishes service before p', assuming there are no arrivals after time t, p will finish service before p' for any pattern of arrivals after time t.

Thus, we can do away with actual time computation. We only need to maintain relative GPS finish ordering for packets in WFQ. WFQ defines a virtual time function $V(t)$ to track the progress of GPS [4]. It is defined as:

$$\frac{\partial V(t+\tau)}{\partial \tau} = \frac{1}{\Sigma_{i \in B(t)} \phi_i} \tag{2.3}$$

For each arriving packet, the virtual start and virtual finish time is computed as:

$$S_i^k = max\{F_i^{k-1},\ V(a_i^k)\} \tag{2.4}$$

$$F_i^k = S_i^k + \frac{L_i^k}{\phi_i} \tag{2.5}$$

Where,

ϕ_i : Weight of session i

$B(t)$: Set of sessions that are backlogged at time t

S_i^k : Virtual start time of kth packet of session i

F_i^k : Virtual finish time of kth packet of session i

a_i^k : Arrival time of the kth packet of session i

$V(a_i^k)$: Virtual time at the instant of packet arrival

At any point in time, WFQ picks the packet with the minimum virtual finish time. This policy is referred as 'Smallest virtual finish time first (SFF)'. The virtual finish time of a packet needs to be computed only when the packet arrives and it need not be recomputed even if the set of backlogged sessions changes in the future.

A WFQ implementation using a priority queue data structure would have $O(log\ n)$ time complexity, where n is the number of flows.

2.3.2 Concept of Worst-Case Fair Index (WFI)

In [6], Bennett and Zhang define a measure to assess how *good* a particular queuing disciple is in comparison with another. A service discipline s is called worst-case fair for

session i if for any time τ, the delay of a packet arriving at t is bounded as:

$$d_{i,s}^{k} < a_i^k + \frac{Q_{i,s}(a_i^k)}{r_i} + C_{i,s} \forall i, k \tag{2.6}$$

Where,

r_i : throughput guarantee to session i

$Q_{i,s}(a_i^k)$: queue size of session i at time a_i^k

$C_{i,s}$: constant independent of queues of other competing sessions

A service discipline is called worst-case fair if it is worst-case fair for all sessions.

2.3.3 WF^{2}Q emulation of GPS

In [3], Parekh showed that a packet in the WFQ system would not lag behind the corresponding GPS system by more than the time required for transmission of one maximum sized packet.

$$d_{i,WFQ}^{k} - d_{i,GPS}^{k} \leq \frac{L_{max}}{r} \tag{2.7}$$

Where,

$d_{i,WFQ}^{k}$: Time at which k_{th} packet of session i departs from WFQ system

$d_{i,GPS}^{k}$: Time at which k_{th} packet of session i departs from GPS system

L_{max}: Maximum size of the packet

r : link speed

However, Benett and Zhang showed that while this is true, WFQ might indeed be far ahead of corresponding GPS system in terms of the number of bits served for a session [6].

In the previous section we noted that in order to be work conserving, WFQ picks the next packet to schedule as if no other packet is going to arrive to the system in future. Thus, if the next packet to depart from GPS has not yet arrived, WFQ would pick a packet from some other session (say session i) for processing. Session i would thus go ahead of the corresponding GPS system. The amount by which the service of a session is ahead of the corresponding GPS system can become arbitrarily large. However, it should be noted that in the long run, the service provided by WFQ and GPS for a session would be same. The session that is far ahead currently would receive less service at some future point in time to compensate.

To avoid an individual session running far ahead, unlike WFQ, WF^{2}Q does not pick the next packet to schedule from all the available packets. The WF^{2}Q server considers only those packets whose virtual start time is less than or equal to the current virtual time and picks the packet with the minimum finish time from them. This policy is referred as 'Smallest eligible virtual finish time first (SEFF)' [5]. The virtual time function computation is same as WFQ.

2.3.4 WF^{2}Q+

As a result of SEFF policy, WF^{2}Q provides the smallest WFI amongst fair queuing algorithms. However, since the WF^{2}Q virtual function computation is the same as for WFQ, the worst-case time complexity is still $O(n)$ for n flows. In [5], Bennett and Zhang proposed a new virtual time function:

$$V_{WF^2Q+}(t+\tau) = max\{V_{WF^2Q+}(t) + W(t, t+\tau), \min_{i \in B(t)} \{S_i^{h_i(t)}\}\} \tag{2.8}$$

Where,

$W(t, t+\tau)$: Amount of service provided by server during time $[t, t+\tau]$

$B(t)$: Set of sessions that are backlogged at t

$h_i(t)$: Sequence number of packet at the head of session i's queue

$S_i^{h_i(t)}$: Virtual start time of that packet

With this new definition, the virtual start and finish time for packets need to be computed only for the head-of-line packet of the flow queue and not for every packet. The computation is done as follows:

$$S_i = \begin{cases} F_i & \text{if } Q_i(a_i^k-) \neq 0 \\ max\{F_i, V(a_i^k)\}, & \text{if } Q_i(a_i^k-) = 0 \end{cases} \tag{2.9}$$

$$F_i = S_i + \frac{L_i^k}{r_i} \tag{2.10}$$

Where

$Q_i(a_i^k-)$: queue size of session i just before time a_i^k

S_i : virtual start time of head-of-line packet of queue

F_i : virtual finish time of head-of-line of packet queue

2.3.5 A note on Virtual time

The term 'Virtual time' is actually a misnomer. $V(t)$ represents the normalized fair amount of service that all backlogged sessions should receive by time t in GPS system.

Thus, the virtual start time of packet of certain session i: $S(i,t)$ represents the amount of service the session should have received at the beginning of servicing the packet. Similarly, the virtual finish time of a packet of certain session i: $F(i,t)$ represents amount of service the session should have received on completion of servicing the packet.

When the WFQ scheduler picks the packet of some session i which has minimum virtual start time, it is actually picking the session that has received the least amount of service.

Moreover, WF^2Q considers only the sessions with virtual start time $\leq$ current virtual time. Thus, WF^2Q is picking from sessions which have received lesser service till that time than they would have in the corresponding GPS system.

While fair queuing guarantees minimum bandwidth for each backlogged session, the excess bandwidth of flows that are not backlogged is distributed amongst backlogged flows in proportion of their weights. Thus, when a new session is backlogged, the amount of bandwidth received by previously backlogged sessions decreases thereby increasing their finish time on a real time scale. Similarly, when a session becomes inactive, the finish time of other sessions decreases on a real time scale. However, as noted in Section 2.3.1, we are only interested in relative ordering. The rate of change of virtual time (amount of service received) increases as the number of inactive sessions increases and decreases as more sessions are backlogged.

2.4 Tiered Service Fair Queuing (TSFQ)

In this section, we describe a new fair queuing discipline, TSFQ, which provides good fairness at constant time. TSFQ exploits two widely observed characteristics of the Internet, namely, that service provides offer some type of tiered service with a small number of service levels, and that a small number of packet sizes dominate.

Traffic quantization can be defined as the process of mapping each flow in the network to one of the small set of service levels (tiers), in such a way that the Quality of Service (QoS) is at least as good as that requested by the flow is guaranteed. Traffic

quantization trades off small amount of system resources for simplicity in the core network functions.

For example, in a continuous-rate network, one may request a bandwidth of 99.92 Kbps while another 99.98 Kbps. In this case, network provider faces a difficult task of distinguishing these two rates and enforcing them reliably. A quantized network might assign both the flows to the next higher level of bandwidth, say 100 Kbps. Network operator then only needs to supply policing mechanisms for a small set of rates, *independent* of the number of flows.

We begin by describing TSFQ for fixed sized packets. Later, we remove this assumption and consider more realistic traffic with variable packet sizes.

2.4.1 TSFQ for fixed packet sizes (TSFQ-F)

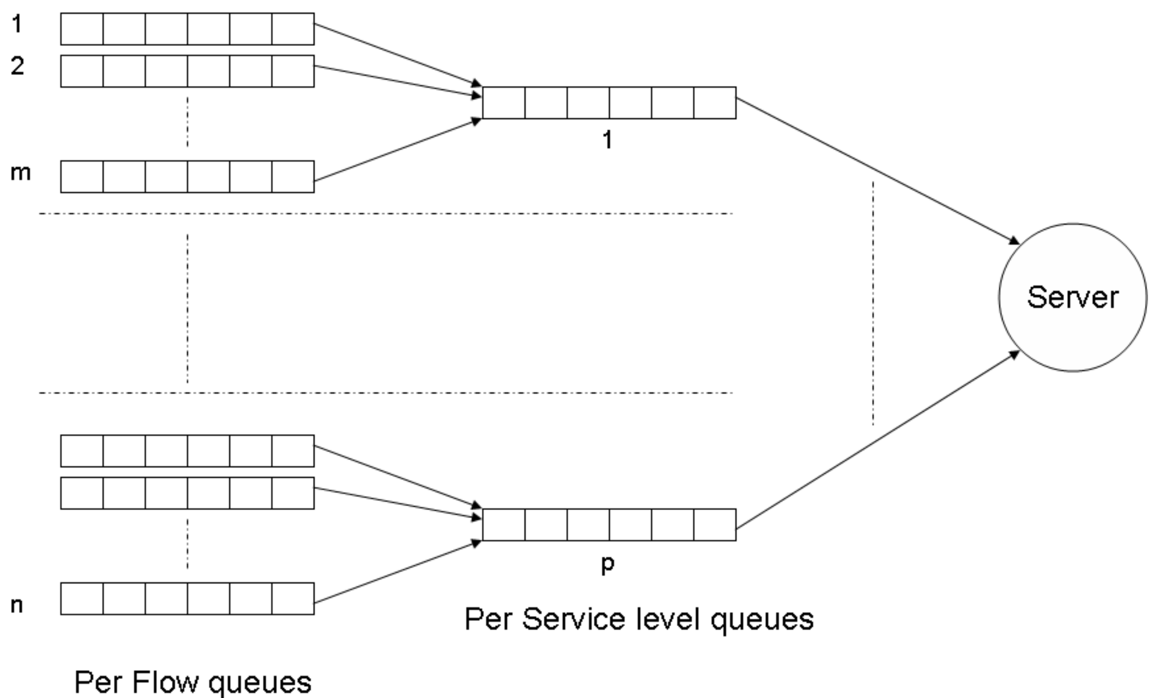

Figure 2.1: TSFQ for fixed packet size (TSFQ-F)

This version of Tiered Service Fair Queuing assumes fixed size packets. Similar to Weighted Fair Queuing, a separate FIFO queue is maintained for each of the flows in the system. These are referred as flow queues. A newly arriving packet for a flow is inserted

at the tail of the flow queue. Thus, packets within a queue are sorted in the order of their arrival times. The scheduler defines a number of service levels (say p) and it maintains one FIFO queue for each of the service levels as well. It is depicted in Figure 2.1.

A certain number of flow queues are mapped to service level queue based on some preconfigured criteria. The FIFO queue at service level l contains the head-of-line packets of all the backlogged flows that are mapped to this service level.

At each dequeue operation, the scheduler looks at the head-of-line packets of each of the service level queues and picks the one with the smallest virtual finish time for processing. Let us say that the scheduler dequeued packet of some flow i from some service level l. The next packet in flow queue i (if present), should be processed only after processing all the packets currently in a service level l FIFO queue. Thus, this new packet is added at the end of the service level queue l.

We need to consider two operations while computing time complexity viz. dequeue operation and virtual time computation. The dequeue operation involves picking a packet from flow with the minimum virtual finish time. Since packets in a service level are sorted by their virtual finish times, we need to compare the finish times of only the packets at the head of service level queues. Comparison of virtual finish times of p packets takes time $O(1)$ given that for a particular system p is small constant.

Virtual time computation requires picking the minimum virtual start time amongst all the backlogged flows. Owing to the fixed packet size, the packets in a service level are implicitly sorted according to their virtual start times as well. Thus, finding the minimum virtual start time also involves comparison of virtual start times of p packets which takes $O(1)$ time for small constant p.

Consider a system where four flows f_1, f_2, f_3, f_4 are mapped to the same service level. Let us say only the first 3 flows are backlogged at a given time and the system picks a packet from flow f_1 first. According to the algorithm described above, the next packet of f_1 would then make it to service level queue right behind flow f_2 and f_3 head-of-line packets. If flow f_4 becomes active while we are processing flow f_2 or flow f_3 packet, its packet would be queued *behind* the f_1 packet. However, a true emulation of GPS should service this packet *before* flow f_1 packet. The next section modifies this algorithm to circumvent this problem.

2.4.2 Modified TSFQ for fixed packet sizes (TSF^{2}Q-F)

The behavior of servicing flow f_1 packet before flow f_4 packet from the example in the previous section, owes to the fact that we admit the flow f_1 packet to the service level FIFO *much sooner* than we should. Thus, the problem can be solved by inserting a packet into service level FIFO queue only when the current virtual time is equal to the virtual start time of the packet.

Thus, the new head-of-line packet of a flow queue can be added to the service level queue only when current virtual time becomes greater than or equal to the virtual finish time of the previous head-of-line packet of the flow queue i.e. F_i^{k-1}.

In addition to the service level queue, this technique maintains another queue, called the GPS queue, for each service level as shown in Figure 2.2. The GPS queue contains a (flowid, virtual finish time) tuple for each active flow in the scheduler. At any point in time when the GPS queue is non-empty, we have an event scheduled to trigger off at the virtual finish time of the packet at the head of the GPS queue.

Consider a scenario when a packet is at the head of a flow queue while that flow

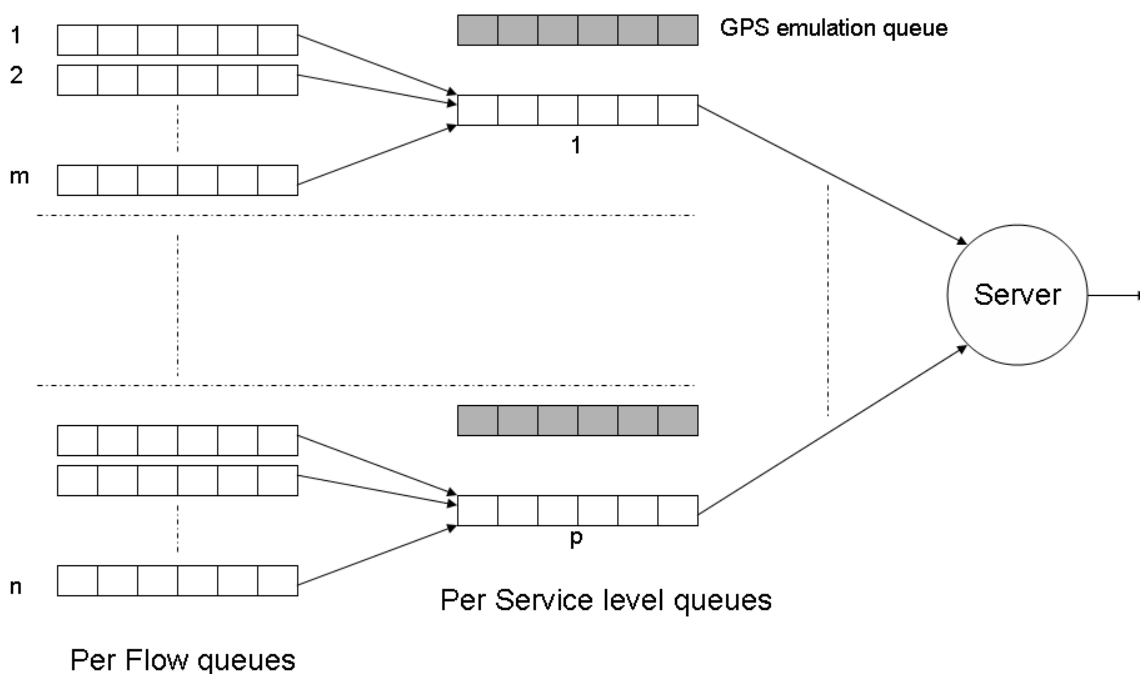

Figure 2.2: Modified TSFQ fixed packet size (TSF^{2}Q-F)

does not have any of its packet in the service level queue. This can happen in two cases, viz.:

- When a packet arrival causes the new flow to become active. In this case, if the virtual start time of this new packet is $\leq$ current virtual time, the packet directly makes it to service level queue. However, if the virtual start time $>$ current virtual time, we simply enqueue this packet in the flow queue. This packet would make it to service level queue when the corresponding event in the GPS queue triggers off.
- If a packet departs from a flow which was already active, we do not add the new flow queue head to the service level queue immediately. Instead, we wait for the trigger to go off. When the trigger goes off, we check the flow queue corresponding to the flow id of the current GPS head-of-line. If the flow queue is non empty, we add the tuple (flow id, virtual finish time of next packet) to the GPS queue. Also, the next packet in the flow queue makes it to the service level queue at this time.

Thus, the packets are inserted only when they are eligible for service i.e. at their virtual start time. The time complexity of this approach is the same as before i.e. $O(1)$.

2.4.3 TSFQ for variable packet sizes (TSFQ-V)

Under real network traffic conditions, packets typically have different sizes. The technique described in the previous section relies on implicit ordering of packets in the service level queue only because the packets are of the same size. However, with variable packet sizes, this need not be the case.

Consider a system with two flows i and j having equal weights mapped to the same service level. Also, consider that a packet with size $size_i$ belonging to flow i is enqueued in the service level queue. Another packet belonging to flow j with size $size_j$ such that $size_j < size_i$ arrives at the service level queue. In the GPS system, this packet belonging to flow j may exit the system before the packet of flow i. Thus, unlike in the fixed packet size case, we cannot simply enqueue the packets in the service queue in the order of their arrival, but we need to insert them in the service queue according to their finish time.

However, we can exploit the fact that in the Internet, certain packet sizes dominate [7]. As shown in Figure 2.3, we maintain a separate FIFO for each of the flows in the system. As before, a certain number of flow queues are mapped to a service level based on some

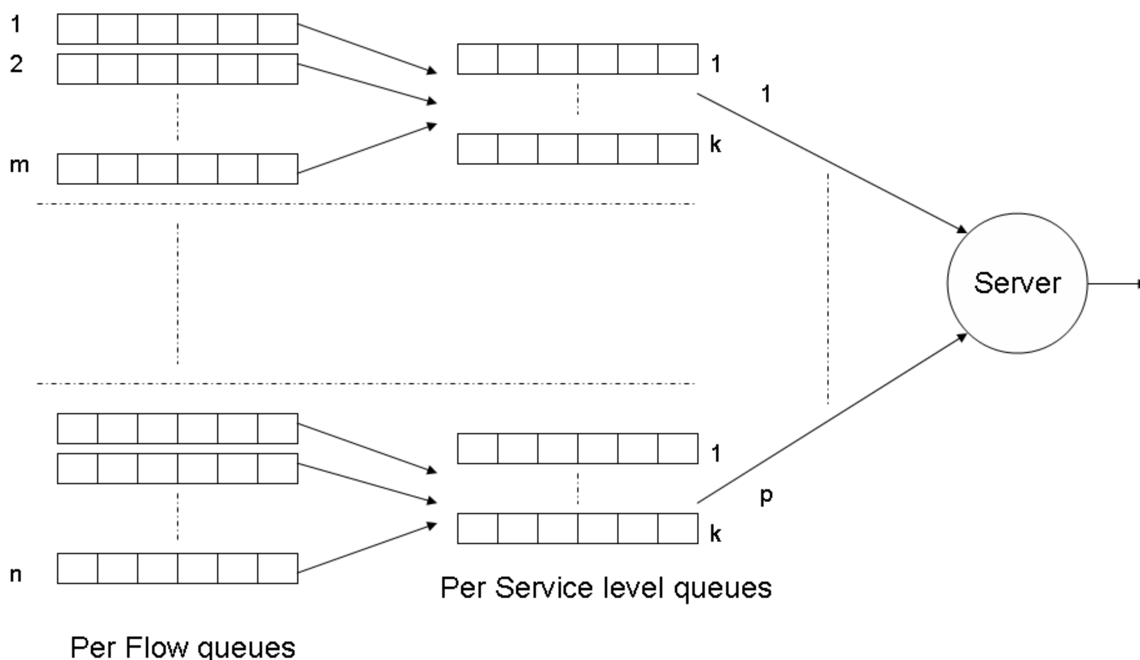

Figure 2.3: TSFQ variable packet size (TSFQ-V)

preconfigured criteria. Instead of one service level queue, we maintain k different service level queues for each service level. Each of the service level queues correspond to one of the common packet sizes like 40, 576, 1500 bytes etc. For the remaining packet sizes we designate queues as for example, one service level queue for packet sizes in the range 41-575 bytes, another for size 577-1499 bytes and so on.

When a packet at the head of the flow queue becomes *eligible* to make it to the service level queue, we check its size and make a decision as to which service level queue it needs to go. If it is one of the fixed packet size service level queues, we directly enqueue since it is implicitly sorted according to its virtual finish time.

However, if the packet belongs to one of the variable size service level queues, then we need to insert the packet at the appropriate location as explained before. Since more than 90% of the Internet traffic consists of packets with a common size, no sorting operation is necessary for the large majority of packets. The sorting operation takes place infrequently (less than 10% of the time) and involves only relatively short queues, since less than 10% of the packets are spread over several queues at l different service levels. The time complexity of sorting operations depends only on the network load and the ratio of packets with non

common size and is independent of the number of n flows.

2.4.4 Modified TSFQ for variable packet sizes (TSF^{2}Q-V)

Similar to TSFQ-F and TSF^{2}Q-F we define two variants of TSFQ for variable size. The variant where we directly admit a flow queue head-of-line packet to service level queue when packet belonging to that flow departs the system is termed TSFQ-V. The variant which delays insertion of flow queue head-of-line packet till it becomes eligible is called TSF^{2}Q-V. Like TSF^{2}Q-F, TSF^{2}Q-V also requires to maintain one GPS queue for each of the service levels.

Chapter 3

Linux Implementation

3.1 Queuing Disciplines in Linux

Linux implements the most commonly used networking protocols in the Internet viz. TCP/IP protocol suite. This section presents an overview of the network stack design in Linux Kernel 2.6.24.6 [4]. However, the design artifacts discussed here do not vary significantly across the 2.6.x.x series of Linux Kernels. At the time of writing, the latest Kernel available was 2.6.26.2 [4].

This subsection presents a birds-eye-view of the design. Succeeding sections take a closer look at the components, specifically focusing on the components that affect the design of new queuing discipline.

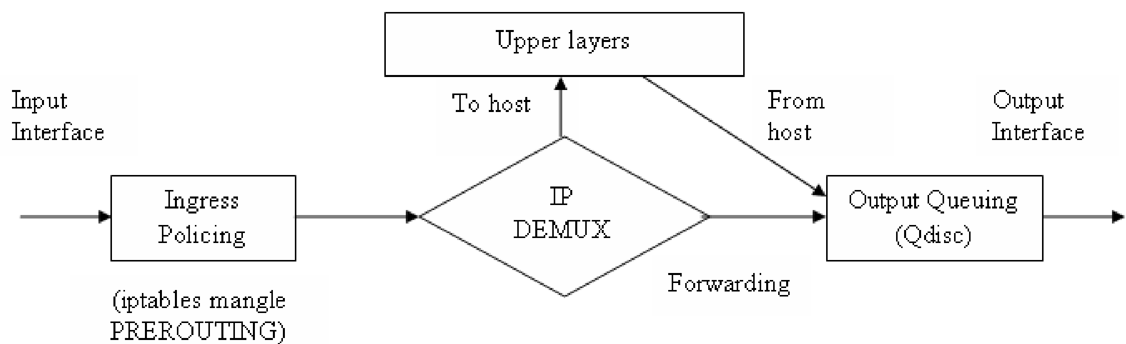

Figure 3.1: Packet forwarding and sending

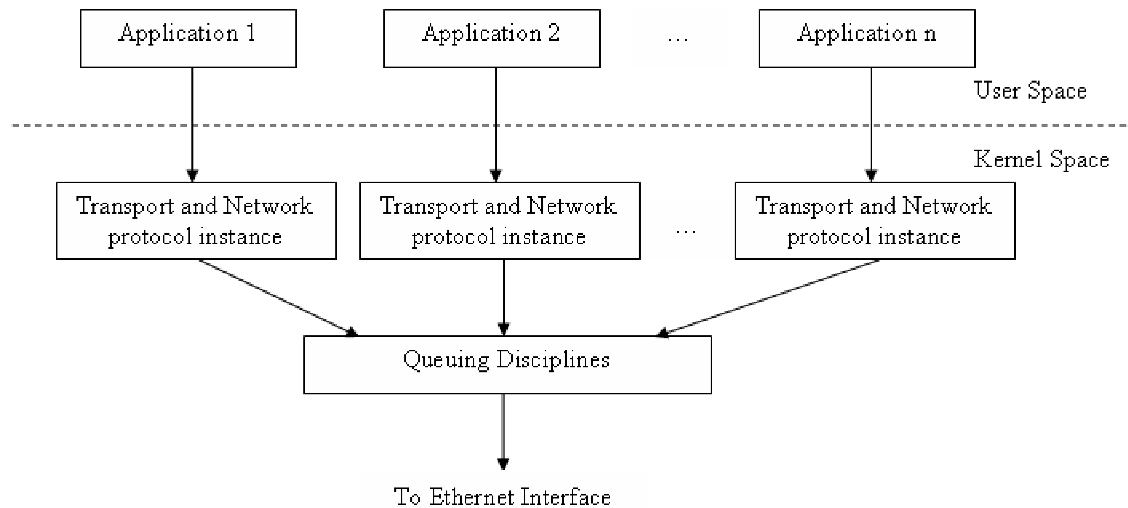

Figure 3.2: Applications communicating over network

Figure 3.1 depicts where Queuing disciplines stand in the context of networking software at an end host.

An incoming packet is subjected to a series of rules before being fed to the IP demultiplexer (IP DEMUX). The packet could be destined to the host itself or it could be delivered for forwarding. Also, an end host itself might generate packets for sending out.

All the packets that are to be sent out are queued at the output interface. When we have competing flows vying for the same output interface card we need some way to resolve the conflict. That is where the scheduling discipline comes into the picture.

Let us refer to Figure 3.2. When an application running in user space wants to transmit data across the network, it uses the underlying transport and network layer mechanisms implemented in kernel. A set of data structures (struct socket, struct sock etc.) is allocated for each connection during connection setup. These data structures also hold function pointers to respective socket operation routines.

Data is transmitted in the form of packets. As a packet travels down the stack, transport and network layer protocols add their own headers. If more than one application is transmitting data simultaneously, then packets from different applications reach the outgoing interface simultaneously. When contention occurs, the criteria used to decide which packet gets service next effectively decides the amount of bandwidth share a particular

application's data flow receives.

An important aspect to note is that the kernel needs to make this decision only when the competing packets are absolutely ready to ship out, i.e., all protocol specific operations are already carried out. Thus, queuing disciplines, by definition, are not part of the transport/network layer. Their implementation can be viewed as an independent sublayer below the network layer. By default, each network interface in Linux is associated with a First-In-First-Out (FIFO) queuing discipline.

The Linux kernel source code can be broadly broken down into two parts viz. code that implements basic operating system functionality and code that is part of the operating system but loaded only on demand. The former constitutes the kernel's network stack. This section looks at each of the components in greater detail.

An application network programmer has a number of address families, transport layer protocols, network layer protocols and options available at his disposal. Linux implements an abstraction, called sockets, to allow easy interface for the network programmer.

Let us consider an example to gain some insight into the Linux Kernel design. Consider some application (say application 1) which wants to use AF_INET address family and communicate using UDP over IP. Consider another application (say application 2) which also wants AF_INET address family but wants to communicate using TCP over IP. Bear in mind that routines specific to address families (like AF_INET), transport protocols (like TCP, UDP) or network protocol (like IP) are separate layers and thus are implemented as separate logical entities in the code.

In terms of socket operations, supporting a particular data flow involves operations such as initializing the socket at the time of creation, data send and receive, and finally releasing the resources.

One way to implement this is as follows: every time a transport layer protocol is to be invoked from an address family specific routine, one can choose which transport layer protocol routine to invoke based on some connection identifier. Such a design would be a set of if-else statements (or switch-cases) to pick the routine corresponding to desired transport protocol. Note that this action needs to be carried out every time such a function call is to be made, i.e., not just at the socket initialization and release but also for each of the data packet that is beings sent out, thereby making it expensive.

Contrast this with another design approach: Let us say an address family specific routine does not actually choose and invoke a transport layer protocol function but it just

invokes a function pointed by a generic transport layer protocol function pointer. Now all one needs to do is to load this function pointer with the corresponding transport layer function at the time of socket initialization. This would avoid the sets of if-elses described in the first approach and also make the code cleaner and easier to understand. This is the approach employed by the kernel.

More specifically, a socket function call of the socket API returns a unique descriptor (usually referred sockfd). This descriptor maps to a unique structure object within kernel code of data type struct socket. One of the members of struct socket is struct proto_ops, which holds function pointers for every socket operation that a user can perform: bind, connect, listen, accept, sendmsg, recvmsg, etc. These function pointers are set to point to desired functions when a socket is created.

Figure 3.3 pictorially represents this concept. For simplicity, we depict the stack from the perspective of a single data flow. Usage of function pointers at each of the levels in Figure 3.3 means that an application programmer can choose any valid combination of (address specific family, transport protocol and network protocol) at the time of socket creation. Since a queuing discipline defines how various such connections interact with each other in terms of bandwidth, this choice of queuing discipline is not available to application programmer requesting connection. A network administrator can configure which queuing discipline to use and what amount of bandwidth to dedicate to a particular connection. The next section looks at this aspect in greater detail.

As a side note, we would like to observe that this operation is inherently polymorphic since the functions should be called on the basis of the run time data type of the invoker. The programming language C achieves this polymorphism with the help of function pointers.

As observed in the previous section, most of the network stack functionality is implemented as loadable kernel modules. These modules are loaded only when they are needed for the first time. When a connection is set up (3-way handshake for TCP), the kernel loads all the relevant modules. Each of these modules has an init function which is invoked at this time. This function sets up ‘function pointers’ to point to the set of functions this particular connection would be needing. This is all done at run time depending on what parameters the user program specified while invoking the Socket API for connection setup.

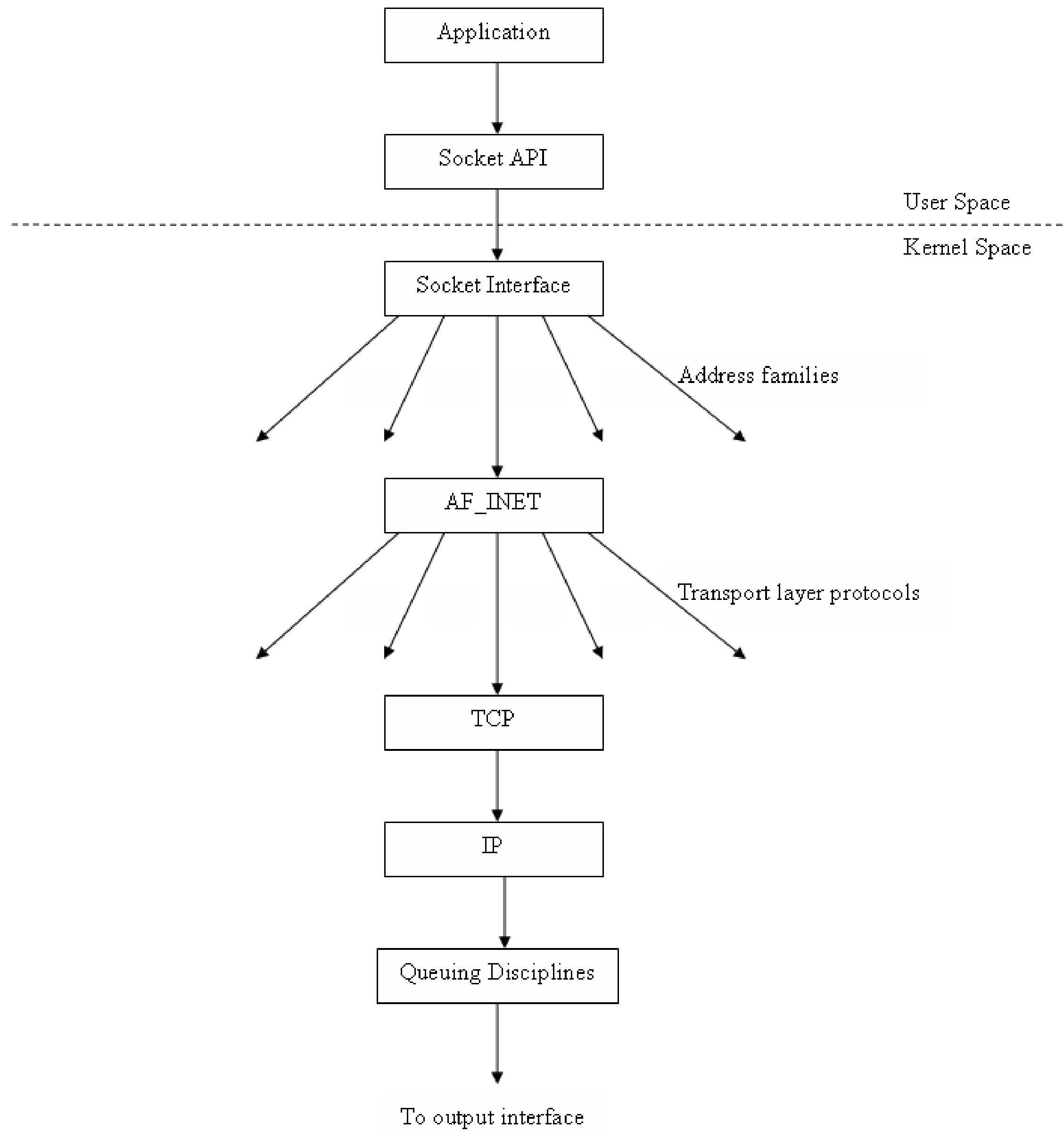

Figure 3.3: Kernel Networking Stack

3.2 Configuring queuing disciplines

3.2.1 Associating weights

Before we delve further into the details, it is imperative to define what constitutes a flow. In our context, simply put, a flow is a stream of packets that we would want to

guarantee certain bandwidth. The queuing discipline receives packets from various applications running on that end host. One way to distinguish packets belonging to one flow from another is on the basis of the port number they are destined to. There could be several other ways to do that, e.g., source port, transport layer protocol being used, or any combination of these, etc.

This is a policy decision that a network administrator would want to take depending on its subscribers, their privileges, the nature of data being carried and its bandwidth requirement. To provide this flexibility to the network administrator, we must have a user level interface that allows us to configure the criteria for what constitutes a flow as well as what bandwidth to assign to trains of packets classified into that flow. Since ensuring that the specified bandwidth requirement is met depends on queuing discipline, we need a mechanism that allows us to directly talk with queuing discipline from user level code.

One way to configure a bandwidth for a particular flow is specifying an actual value while defining the flow, e.g., defining something like 10 Mbps for flow 1, 20 Mbps for flow 2, and so on. Such an approach would require prior knowledge of the capacity of the outgoing interface. Alternatively, we can normalize the associated bandwidth and specify the percentage of available bandwidth that a particular flow should use e.g. 10% for flow 1, 20% for flow 2, and so on. We prefer this latter approach.

3.2.2 RTNETLINK Sockets

Note that this configuration cannot be done at the time of conventional socket creation because, though we want to associate bandwidth to an individual flow, the queuing discipline configuration is applicable to all the flows in the system.

One commonly used way to directly manipulate parameters within the kernel is to use the ioctl (defined in sys/ioctl.h) system call. However, queuing discipline configuration is achieved using a different technique. A special type of socket is created to carry out this configuration. NETLINK [12] provides a facility for user applications to directly communicate with kernel modules, query their status, configure and effectively control them. It is an extension of the standard socket implementation as shown in Figure 3.4.

RTNETLINK is implemented as a different address family (socket system call with address family AF_NETLINK). One still has to go through the usual process of opening a socket, binding the socket to a local address, sending a message to an end point, receiving

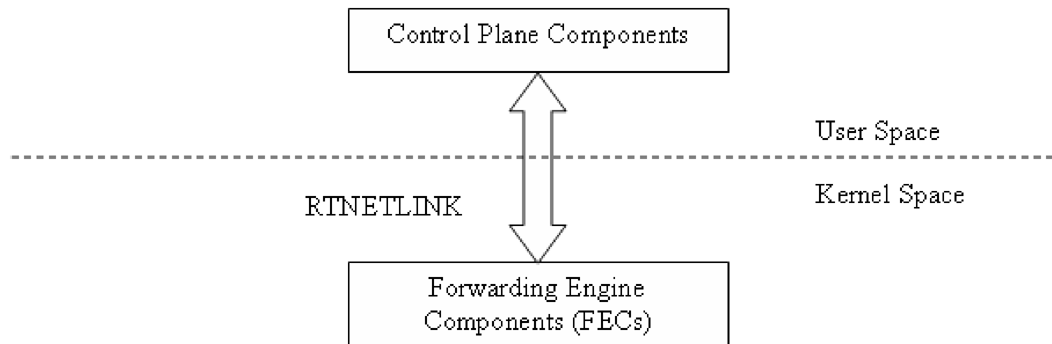

Figure 3.4: RTNETLINK sockets

a message from another end point and closing the socket.

Alexey Kuznetsov et al [2] developed a software named iproute which contains tools for manipulating routing tables, tunnels etc. (called ip) and a traffic control tool (tc). The tc command opens an RTNETLINK socket to communicate and configure queuing disciplines. The tool comes packaged with any standard Linux distribution (typically present in /sbin/ip, /sbin/tc) or else can be built from sources available at [2].

Figure 3.5 combines the ideas we discussed in the last couple of sections. It shows two separate lines of control. Packets traverse down the stack through socket API, address family specific routines, TCP and then IP. While configuring queuing discipline, classifier and related attributes, we can bypass this usual hierarchy with the help of tc command with a socket of type AF_NETLINK.

3.2.3 Steps involved in configuring queuing discipline

With this perspective in mind, let us provide more details about the iproute utility. Iproute contains several networking utility functions like traffic control (tc), policy routing, NAT capabilities, packet scheduling, packet filtering etc. [3].

The tc command operates on an individual network interface. We can associate a specific queuing discipline (usually referred to as qdisc) with an interface using this command. A sample command may look like

tc qdisc add dev eth0 root handle 1:0 wf2q+

Every traffic control element is internally recognized by 'major number: minor

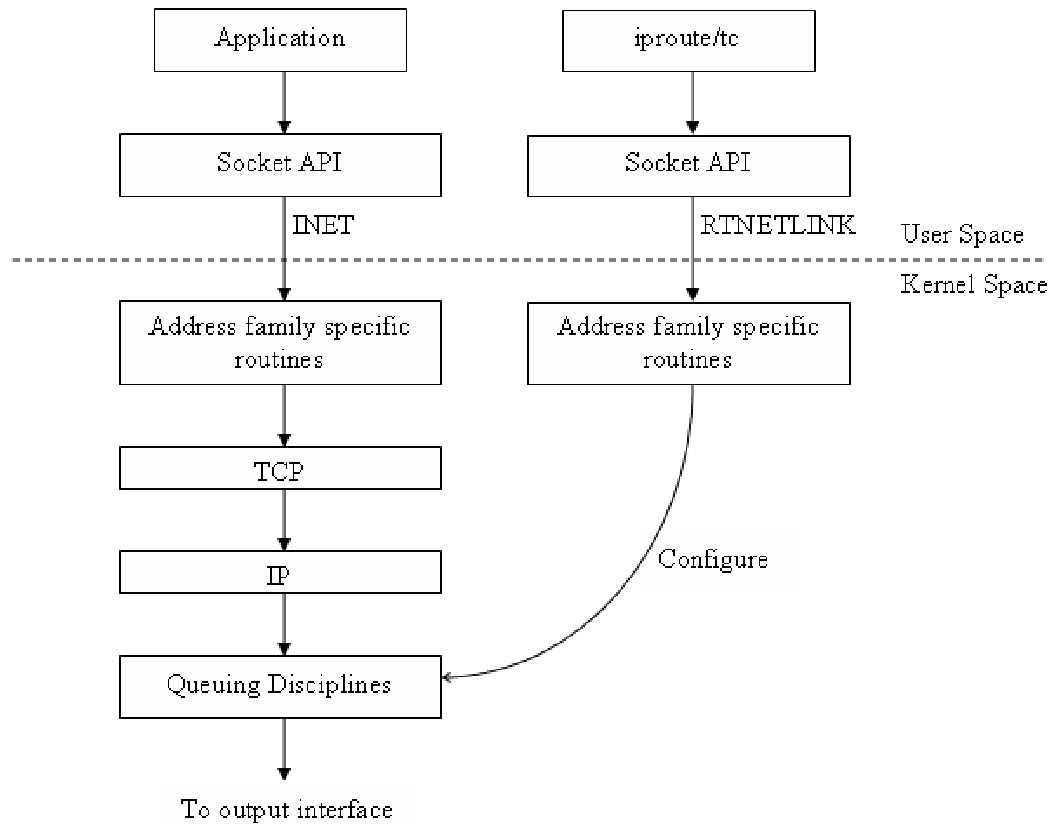

Figure 3.5: Configuring Queuing Disciplines

number'. The tuple is often called within the code. In the aforementioned example, major number is 1 while minor number is 0.

Once we associate a queuing discipline (WF^{2}Q+ in our example below) with an interface say eth0, we would further like to associate bandwidths with different flows going out of eth0. We first define a parent class and then define flows within it. This can be achieved by commands:

tc class add dev eth0 parent 1:0 classid 1:1 wf2q+

tc class add dev eth0 parent 1:1 classid 1:3 wf2q+ weight 90

tc class add dev eth0 parent 1:1 classid 1:4 wf2q+ weight 10

Each flow is said to belong to a different class. Note that while the classes have

minor numbers 3 and 4, the major number remains same viz. 1.

The set of packets that are mapped to a particular flow can be further classified and scheduled. By default they are scheduled according to FIFO.

Let us look at third option of tc now viz. tc filter. It allows a user to define what constitutes a flow.

tc filter add dev eth0 parent 1:0 protocol ip prio 1 u32 match ip dport 7000 0xffff flowid 1:3

This command picks any packet with destination port 7000 and applies identifier flowid 1:3 to it. By previous tc class commands, flow with id 1:3 is supposed to have 90% of bandwidth. Thus, the connection with destination port 7000 has assured bandwidth share of 90%;

To summarize, configuration consists of the following steps

- Associate a queuing discipline with network output interface (tc qdisc).
- Define classes within qdisc. Associates identifiers, weight (tc class).
- Define rule which can classify given packet into defined class (tc filter).

Each of the tc command invocations prompt the tc implementation to open an RTNETLINK dialog with the kernel. It then sends a command in the format (command-type, parameters) using the same.

3.3 Organization of tc code

Figure 3.6 depicts the design of iproute/tc utility. As described above, a user can issue various commands like add or delete Qdisc, add or delete class, add or delete filter etc. RTNETLINK socket communication to kernel functionality is segregated in library iproute/lib. Other iproute tools use this functionality as well.

While communicating this command to kernel using RTNETLINK socket, tc utility sends an identifier specifying the command time. This identifier is mapped to the kernel function performing the corresponding action. The association between identifier and corresponding kernel function is set up using rtnl_register when the basic packet scheduler module (i.e. FIFO) is loaded in kernel (this happens during booting itself).

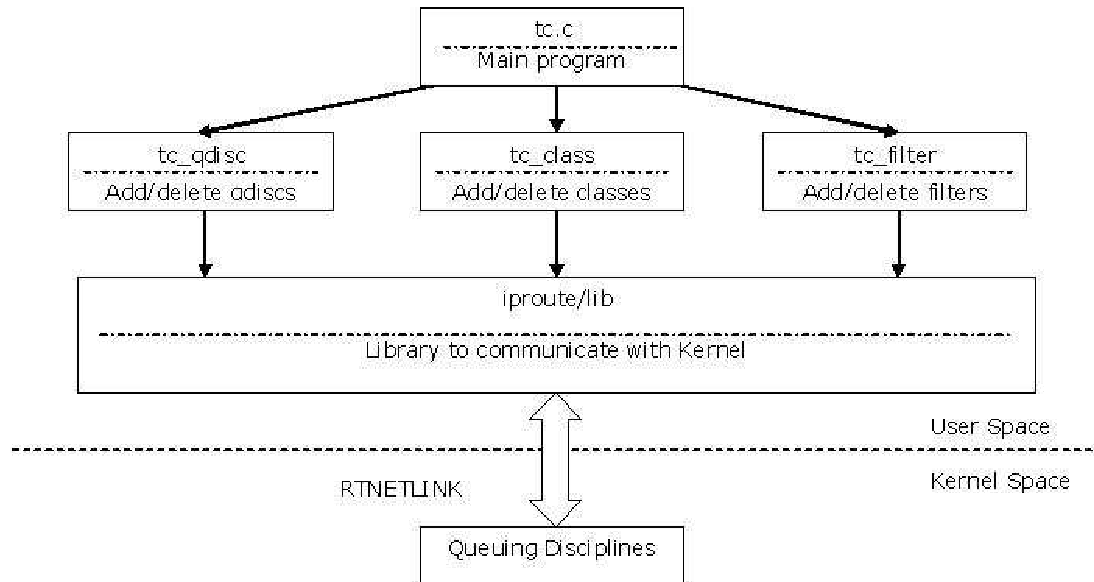

Figure 3.6: iproute/tc design

For example, if the 'tc qdisc add' command is issued, the RTNETLINK message sent by tc's socket contains the command type 'RTM_NEWQDISC '. This command in turn is mapped to the kernel function tc_modify_qdisc. Table 3.1 presents the complete list for iproute version 2-2.6.19-061214 running on Linux kernel 2.6.24.6.

Table 3.1: Table iproute/tc design

Command	**Command type**	**Kernel function**	**Operation**
tc qdisc add	RTM_NEWQDISC	tc_modify_qdisc	Add new qdisc
tc qdisc del	RTM_DELQDISC	tc_get_qdisc	Delete qdisc
tc class add	RTM_NEWTCLASS	tc_ctl_tclass	Add new class
tc class del	RTM_ DELTCLASS	tc_ctl_tclass	Delete class
tc filter add	RTM_NEWTFILTER	tc_ctl_filter	Add new filter
tc filter del	RTM_DELTFILTER	tc_ctl_tfilter	Delete filter

Chapter 4

Specific Requirements

4.1 Configuring WF^{2}Q+

- An interface to associate the queuing discipline with the outgoing network interface in question (eth0 in our example).
  tc qdisc add dev eth0 root handle 1:0 wfq bandwidth 100Mbit avpkt 1000 cell 8

- A parent class, which is parent of every flow.
  tc class add dev eth0 parent 1:0 classid 1:1 wfq bandwidth 100Mbit prio 8 allot 1514 cell 8 maxburst 20 avpkt 1000 bounded

- For WF2Q+, we define 'tc class' such that each class corresponds to a separate competing flow. The following command defines a parent class which is associated with the outgoing interface in question (eth0 in this example).
  tc class add dev eth0 parent 1:1 classid 1:3 wfq bandwidth 100Mbit weight 90 prio 5 allot 1514 cell 8 maxburst 20 avpkt 1000 # flow 1
  tc class add dev eth0 parent 1:1 classid 1:4 wfq bandwidth 100Mbit weight 10 prio 5 allot 1514 cell 8 maxburst 20 avpkt 1000 # flow 2

- We would want to service the packets mapped to particular flow using FIFO.
  tc qdisc add dev eth0 parent 1:3 handle 30: pfifo # flow 1
  tc qdisc add dev eth0 parent 1:4 handle 40: pfifo # flow 2

- Tell the classifier to filter the packet with certain destination port to be mapped to

specific class id. For example, the following configuration would map packet with destination port 6000 to class 1:3 and 6001 to class 1:4.

tc filter add dev eth0 parent 1:0 protocol ip prio 1 u32 match ip dport 6000 0xffff flowid 1:3
tc filter add dev eth0 parent 1:0 protocol ip prio 1 u32 match ip dport 6001 0xffff flowid 1:4

4.2 Configuring TSFQ variants

- An interface to associate the queuing discipline with the outgoing network interface in question (eth0 in our example).
  tc qdisc add dev eth0 root handle 1:0 tsfq bandwidth 100Mbit avpkt 1000 cell 8

- A parent class, which is parent of every flow.
  tc class add dev eth0 parent 1:0 classid 1:1 tsfq bandwidth 100Mbit prio 8 allot 1514 cell 8 maxburst 20 avpkt 1000 bounded

- For TSFQ, we define 'tc class' such that each class corresponds to a separate service level. Following command defines a parent class which is associated with the outgoing interface in question (eth0 in this example).
  tc class add dev eth0 parent 1:1 classid 1:3 tsfq bandwidth 100Mbit weight 90 prio 5 allot 1514 cell 8 maxburst 20 avpkt 1000 # flow 1

  tc class add dev eth0 parent 1:1 classid 1:4 tsfq bandwidth 100Mbit weight 10 prio 5 allot 1514 cell 8 maxburst 20 avpkt 1000 # flow 2

- Quantization step: Tell the classifier to filter the packet with certain destination port to be mapped to specific class id. For example, the following configuration would map packets with destination port 6000 or 6001 to class 1:3 and 6002 and 6003 to class 1:4.

  tc filter add dev eth0 parent 1:0 protocol ip prio 1 u32 match ip dport 6000 0xffff flowid 1:3
  tc filter add dev eth0 parent 1:0 protocol ip prio 1 u32 match ip dport 6001 0xffff flowid 1:3
  tc filter add dev eth0 parent 1:0 protocol ip prio 1 u32 match ip dport 6002 0xffff flowid 1:4

tc filter add dev eth0 parent 1:0 protocol ip prio 1 u32 match ip dport 6003 0xffff flowid 1:4

We implement 4 variants of tsfq viz. tsfqv1, tsfqv3, tsfqvarv1, tsfqvarv3. The desired TSFQ variant can be configured by replacing tsfq in syntax noted above with the name of TSFQ variant. Succeeding sections explain these TSFQ variants.

Chapter 5

Design and Implementation

5.1 User Perspective

Figure 5.1 elaborates on the qdisc creation part from iproute/tc perspective. Class and filter creation is done in a similar manner.

Tc_qdisc_modify looks up for the queuing discipline supplied by the tc qdisc add command and refers to corresponding q_<qdisc name>.c file. This is done dynamically. POSIX provides an interface to dynamic linking loader with functions like dlopen, dlerr, dlsym and dlclose (defined in header <dlfcn.h>)

Once the function pointer is dynamically loaded with the requested queuing disciplines pointer, it invokes the corresponding parsing routine to parse parameters. This routine also carries out the validation. If this phase is successful, now we need to communicate this configuration to the kernel. This part of the code is broken down separately into iproute/lib. [Note that apart from Traffic control (tc), iproute also performs other useful tasks like managing routing tables, tunnels etc (ip). The functionality required to communicate with the kernel is required by both and is thus segregated into a separate directory.]

The two squares with thick bordering, viz. q_wf2q+.c and q_tsfq.c, represent code that is part of our work. Once again, for brevity, we represent the queuing discipline file as q_tsfq.c. While in reality, we have one such file for each of the TSFQ variants viz. q_tsfq-f.c q_tsf2q-f.c q_tsfq-v.c q_tsf2q-v.c. With the iproute/tc framework already in place, all this code needs to do is to parse, validate and send parameters down to the kernel in a manner exactly analogous to other queuing disciplines. Thus, we do not deal with it in greater

depth here.

5.2 System (Kernel) Perspective

Sections 5.2.2 and 5.2.1 deal with design issues that are common to implementation of all the queuing disciplines. Subsection 5.2.3 onwards, we deal with design issues specific to each of the queuing disciplines.

5.2.1 Per flow queues: Queues from FIFO implementation

WF^2Q+ requires us to maintain one separate queue for each of the competing flows, while TSFQ maintains individual queue for each flow as well as a separate queue for each of the predefined service level.

Enqueuing for flow queues is done on packet arrival while dequeuing from flow queues is governed by virtual time computation. However, not withstanding the criteria for

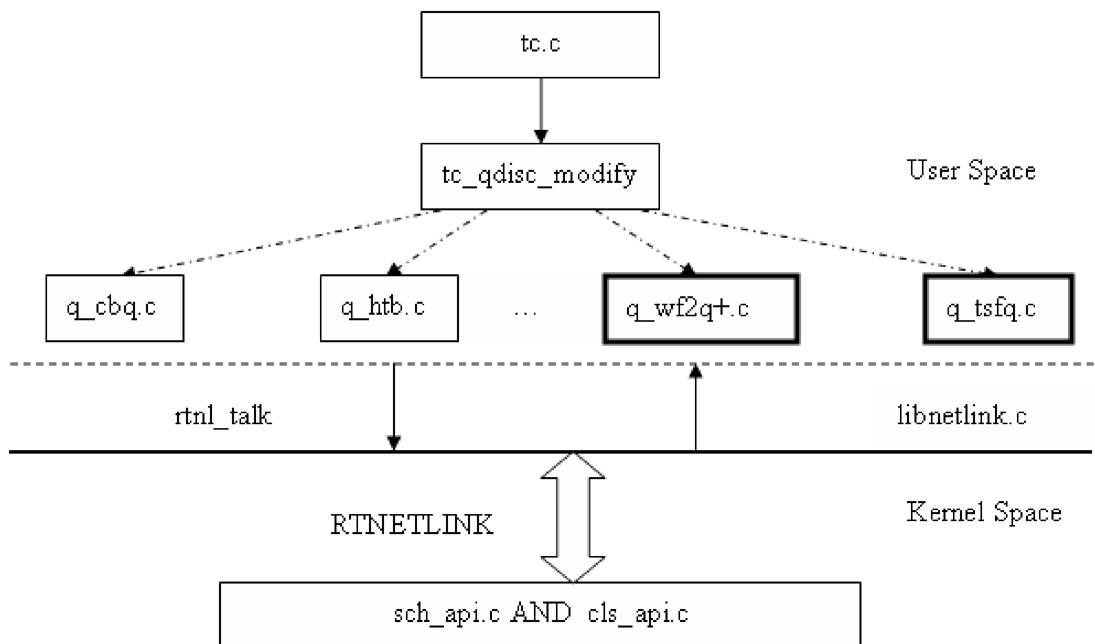

Figure 5.1: Qdisc creation using iproute/tc

committing these operations, we still expect a data structure with basic queue functionality.

The Linux kernel defines a data structure socket buffer as struct sk_buff (defined in source tree at include/linux/sk_buff.h). It holds one pointer each for headers of each of the layers. It also holds a data pointer. In essence, one sk_buff instance has all the data that is necessary for transmitting a packet. An sk_buff instance pointer is in fact the currency used by queuing disciplines to talk with external modules. Thus, a particular flow queue needs to maintain a chain of sk_buff pointers and provide enqueue and dequeue operations on them.

As described before, the Linux kernel includes by default the FIFO queuing discipline. Thus, we can reuse this functionality by making each of the WF^2Q+ flow queue or TSFQ flow/service level queue as the FIFO queue of the existing implementation. While this is a cleaner implementation, it is also consistent with the hierarchical queuing discipline philosophy. The FIFO scheduler code is implemented using struct Qdisc and functions to perform operations on Qdisc (defined in include/net/sch_generic.h).

5.2.2 Support to look up Queue head: Peek functionality

As described in Section 2.3.4, WF^2Q+ computes the virtual time finish using following formula:

$$F_i = S_i + \frac{L_i^k}{r_i} \tag{5.1}$$

where L_i is the length of the packet.

A packet can reach the head-of-line of a queue in two scenarios viz. when a newly arriving packet finds its flow queue empty, or when a packet at head-of-line of a particular flow queue departs and the immediately succeeding queued packet makes it to the head-of-line of the queue. When the newly arriving packet makes it to head-of-line of queue directly, we have a pointer to the packet's sk_buff structure and thus can easily Figure out L_i. However, when a previously queued packet makes it to the head-of-line of a queue, we need a mechanism to find the length of the top of the queue without actually dequeuing the packet. The existing FIFO implementation does not support this functionality. Thus, we added that functionality to the kernel code.

Function prototype:

*struct sk_buff * (*peek)(struct Qdisc *dev);*

The modifications to the FIFO interface have been *italicized*:

```
static struct Qdisc_ops pfifo_fast_ops = {
.id = "pfifo_fast",
.enqueue = pfifo_fast_enqueue,
.dequeue = pfifo_fast_dequeue,
.peek = qdisc_peek,
.
. };
```

Since this requires change in the FIFO implementation which is part of the kernel boot image, this functionality is not available as a loadable module but requires rebuilding Linux kernel.

5.2.3 WF^{2}Q+

We maintain a separate FIFO for each flow. A flow, in our case, is characterized by the destination port as configured using *tc filter*. When we receive a packet enqueue request, we need to match the incoming packet against all the filter rules configured previously to find a potential match. The classifier routine returns a pointer to the corresponding data structure wfq_class.

For each flow in the system there is a *unique wfq_class* instance. This structure holds the class identifier (an integer used to uniquely identify the flow), the weight corresponding to that flow, a pointer to that flow's FIFO, the virtual start and finish time of the packet at the head-of-line of FIFO for that flow etc.

One such instance is generated for each *tc class* command. The weight field within the class is loaded at the same time. First, the *tc qdisc* command loads this kernel module. The module maintains a global variable for the current virtual time.

Algorithm 1 WF^{2}Q+ Virtual time computation

$min_start_time \Leftarrow$ Minimum start time amongst packets at head of queues

$delta \Leftarrow$ Time elapsed since previous virtual time update

$current\ virtual\ time = max\{current\ virtual\ time + delta,\ min_start_time\}$

The virtual time computation algorithm used is shown in Algorithm 1. The routine that updates the virtual time after each enqueue/dequeue maintains a static variable

for the previous timestamp. The time difference between current and previous timestamp represents the amount of service done since the last virtual time update. We obtain the current time using the psched_get_time function defined in the source tree at include/net/pkt_sched.h

Algorithm 2 states the enqueue operation discussed before in algorithmic format.

Algorithm 2 WF^2Q+ Enqueue(skb)

$cl \Leftarrow$ pointer to wfq_class instance that this skb belongs to (return value of classifier)
$flowid \Leftarrow$ flow id of arriving packet skb
if $flowid$ queue is empty **then**
  $S_{flowid} = max\{F_{flowid},\ current\ VT\}$; $F_{flowid} = S_{flowid} + \frac{skb->len}{cl->weight}$
end if
Enqueue skb in $cl \rightarrow fifo_queue$
Update Virtual time: Algorithm 1.

The dequeue operation involves choosing the flow with the minimum virtual finish time. The virtual time is updated after dequeue operation as shown in algorithm 3.

Algorithm 3 WF^2Q+ Dequeue

$cl \Leftarrow$pointer to wfq_class instance of backlogged flow with minimum virtual finish time
$skb \Leftarrow$dequeued packet from $cl \rightarrow fifo_queue$
if $flowid$ queue is not empty **then**
  $new_head_skb \Leftarrow$pointer to new head of flow queue
  $S_{flowid} = F_{flowid}$; $F_{flowid} = S_{flowid} + \frac{new_head_skb->len}{cl->weight}$
end if
Update Virtual time: Algorithm 1.

This functionality is available as a loadable module with name sch_wf2q+.ko.

5.2.4 TSFQ-F

With Tiered Service Fair Queuing, we need to maintain a separate queue for each flow as well as each service level. We define one instance of the data structure *struct tsfqf_class* to correspond to one service level. This structure holds the class identifier, the weight corresponding to that service level, and an array of pointers to each of the flow

FIFOs mapped to this service level. We also need to maintain the virtual start and virtual finish time for the packet at the head of the queue of each flow queue within the service level. This is stored as an array within struct tsfqf_class as well.

We perform the quantization step manually, i.e., while configuring filters using *tc filter* if we want flows destined for port 6000 and port 7000 to be mapped to same service level, then two *tc filter* commands must be issued with the same classid.

When an input packet arrives for enqueuing, the classifier routine runs through the filter list and returns pointer to the corresponding service level instance of struct tsfqf_class. We look at the destination port and use it to index among the flow queue pointer array to pick the flow queue the packet belongs to.

Since we can access the flow queues within a service level by indexing, we need not carry the actual packet in the service level queue. The service level queue is implemented as a sequence of index numbers that refer to flow queues. If a particular service level is chosen for servicing, we dequeue from the service level queue to get an index. This index is then used to perform a dequeue operation on the appropriate flow queue within that service level.

In this TSFQ version, the head of flow queue directly makes it to service level if no other packet of that flow is previously queued. We formally present it in Algorithm 4.

Algorithm 4 TSFQ-F Enqueue(skb)

$cl \Leftarrow$ pointer to tsfqf_class instance that this skb belongs to (return value of classifier)

$flowid \Leftarrow$ flow id of arriving packet skb (used to index in flow queue array)

if $flowid$ queue is empty **then**

  $S_{flowid} = max\{F_{flowid},\ current\ VT\}$; $F_{flowid} = S_{flowid} + \frac{skb->len}{cl->weight}$

  Enqueue in $cl \rightarrow$ service level queue

end if

Enqueue skb in $cl \rightarrow fifo_queue[flowid]$

Update Virtual time: Algorithm 1.

Though algorithm 4 refers to the same update virtual time algorithm as WF^{2}Q+, there is a difference in the way the minimum virtual start time of the backlogged sessions is computed. While WF^{2}Q+ needs to look at each flow, TSFQ-F needs to look at only the head of the service level queues. For fixed sized packets, the service level queues are implicitly sorted by minimum virtual start time as well.

Let us look at the dequeue operation now. We need to pick the service level with the minimum virtual finish time. That gives us the index, which is used to dequeue packet from appropriate flow queue of the chosen service level. If new packet now makes it to head of the flow queue, we must compute virtual start and finish time for it. Also, this new packet would be enqueued in service level queue. We state it formally in Algorithm 5.

Algorithm 5 TSFQ-F Dequeue

$cl \Leftarrow$pointer to tsfqf_class instance service level with minimum virtual finish time
$flowid \Leftarrow$Index into the flow queues, points to flow queue to dequeue from
$skb \Leftarrow$dequeued packet from $cl \rightarrow fifo_queue[flowid]$
if $flowid$ queue is not empty **then**
  $new_head_skb \Leftarrow$pointer to new head of flow queue
  $S_{flowid} = F_{flowid}$; $F_{flowid} = S_{flowid} + \frac{new_head_skb->len}{cl->weight}$
  Enqueue in $cl \rightarrow$ service level queue
end if
Update Virtual time: Algorithm 1.

This functionality is available as a loadable module with name sch_tsfqv-f.ko.

5.2.5 TSF^{2}Q-F

As explained in Section 2.4.2, we circumvent the defect in previous version by delaying admission of a flow queue packet in the service level queue till it is eligible. However, the additional GPS queue technique also requires a mechanism that should trigger off an event when the current virtual time equals the virtual finish time of the packet at the head of the line of GPS queue.

Alternatively, every time we update the virtual time we need to check if any of the flow queue head-of-line packets that was previously refused admission, has now become eligible to make it to service level queue. One method is to scan through list of all flow queues within every service level queue checking for the same. However, this would be computationally inefficient. We solve this problem as follows:

We maintain a circular array (say ineligiblepkts) subscripted by virtual time. Each element ineligible[i] in the array is head to the list of pointers to flow queues whose head-of-line packet's virtual start time equals array index i. When we update current virtual time (in enqueue/dequeue), we look at the ineligiblepkts[current_virtual_time] entry. All

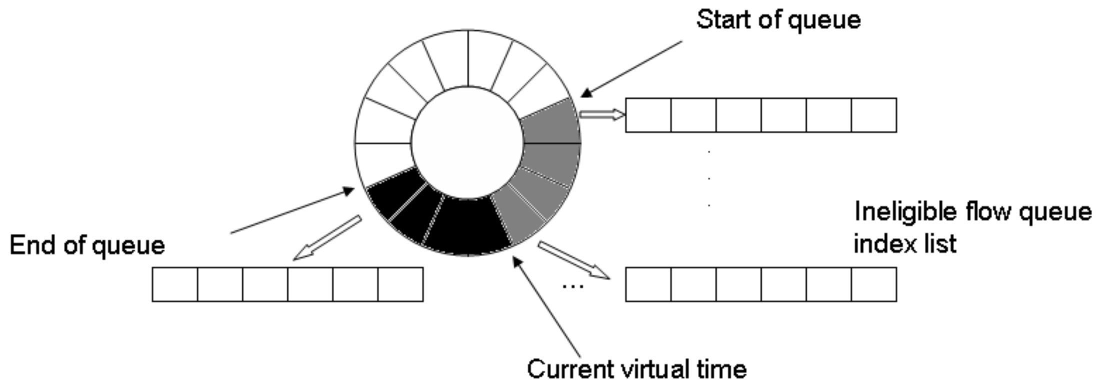

Figure 5.2: Circular queue for ineligible packets

the packets in that list as well as all the packets in all lists from beginning of circular array till index current_virtual_time are now eligible for admission to the service level queue. This is illustrated in Figure 5.2. Packets pointed by all the elements in all *grey* colored lists would make it to the service level queue in this pass. Start of the queue would be reset to current virtual time i.e. at the beginning of *black* colored cells.

Virtual time grows with passage of time, however the number of ineligible packets at any point in time has upper bound limited by number of flows in the system. Thus, we use circular array of the order of number of flows. The algorithm is stated in Algorithm 6. Enqueue and Dequeue operations remain same as TSFQ-F except that we add packet to the service level queue only if current virtual time $\leq$ virtual start time of packet and we invoke algorithm 6 immediately after updating the virtual time.

Algorithm 6 TSF^{2}Q Check for eligibility for service level admission

for all i such that $start_of_array \leq i \leq current_virtual_time$ **do**
  for every $< classid, flowid >$ in list $ineligible[i]$ **do**
    Remove from $ineligible[i]$
    Enqueue $flowid$ in $cl \rightarrow$ service level queue
  end for
end for
$start_of_array \leftarrow current_virtual_time$

This functionality is available as a loadable module with name sch_tsf2q-f.ko.

5.2.6 TSFQ-V

In real network situations, packets often have variable size. As explained in Section 2.4.3, we maintain certain k number of queues at each service level. Some of the queues are dedicated to fixed packet sizes that frequently occur in network while others map packets from certain range of size. In particular, we use three fixed size queues for packet sizes 40, 1200 and 1500 bytes. We also use three variable sized queues for packet sizes in range 1-39 bytes, 41-1199 bytes and 1201-1499 bytes.

Data structure *struct tsfqv_class* instance corresponding to a service level is similar to *struct tsfqf_class* defined before. Again, service level queues are sequence of index numbers that refer to flow queues. Unlike *struct tsfqvf_class* though, we now maintain an array of service level queues in *struct tsfqv_class*.

We need to determine following things for an arriving packet:

- Service level it belongs to: Done by classifier routine by running through the filter list.
- Flow queue it belongs to: Done by looking at the destination port in the packet.
- If the packet is at head of flow queue then, the service level queue it belongs to: Done by looking at the packet size.

Service level queues dedicated to fixed packet sizes are implicitly sorted and thus simple enqueue works for it. However, the service level queues for range of packet sizes require insertion of the packet at appropriate position in order to maintain the service level queue sorted according to virtual finish time. However, certain packet sizes dominate Internet traffic. A study in [7] found that 3 common packet sizes constitute 90% of all Internet traffic. Thus, sorting operations needs to be done only on the remaining 10%. Moreover, these 10% packets are distributed over service level queues of various service levels.

Fixed size packet service level queues are sorted according to both virtual start and finish times. However, variable size packet service level queues are sorted according to their virtual finish times only. Thus, we maintain one variable per service level which holds minimum start time of packet in that service level. We update this variable when a packet is inserted in service level or when a packet is dequeued from a service level.

When a packet is dequeued from a service level, the dequeued packet might have been the one with minimum virtual start time. Thus, we need to look at every head of the queue of fixed packet size service level queue and every packet in variable packet size service level queue to determine the new minimum start time. The enqueue algorithm is described in Algorithm 7.

Algorithm 7 TSFQ-V Enqueue(skb)

$cl \Leftarrow$ pointer to tsfqv_class instance that this skb belongs to (return value of classifier)
$flowid \Leftarrow$ flow id of arriving packet skb (used to index in flow queue array)
if $flowid$ queue is empty **then**
  $S_{flowid} = max\{F_{flowid},\ current\ VT\}$; $F_{flowid} = S_{flowid} + \frac{skb->len}{cl->weight}$
  $fifoid \Leftarrow$ fifoid of arriving packet skb based on its size
  Enqueue in $cl \rightarrow$ service level queue[$fifoid$]
  Update $cl \rightarrow min_virtual_time$
end if
Enqueue skb in $cl \rightarrow fifo_queue[flowid]$
Update Virtual time: Algorithm 1.

The dequeue operation needs to pick the flow with minimum virtual finish time from all the service level queues across all the service levels. It then recalculates minimum virtual start time for that service level. This version of TSFQ directly admits next packet, if any, from the flow queue we just dequeued from into the service level queue. Since the next packet might be of different size, we need to figure out which of the service level queue it belongs to. The dequeue algorithm is described in Algorithm 8.

This functionality is available as a loadable module with name sch_tsfq-v.ko.

5.2.7 TSF^{2}Q-V

Section 5.2.5 delays admission of the head-of-line packet of flow queue into the service level queue till it becomes eligible. Section 5.2.6 solves the problem of variable packet sizes by maintaining different service level queues. We note that these problems (and their solutions) relate to different pieces of logic in the scheduler. Thus, combining these codes is straightforward. Enqueue and dequeue operations remain same as described in Section 5.2.6. However, the update virtual time function is followed by algorithm 6 to

Algorithm 8 TSFQ-V Dequeue

cl ⇐pointer to tsfqv_class instance service level with minimum virtual finish time

$fifiod$ ⇐index into the service level queue within the service level cl

$flowid$ ⇐top element of $cl \rightarrow$ service level queue[$fifoid$]

skb ⇐dequeued packet from $cl \rightarrow fifo_queue[flowid]$

Recompute $cl \rightarrow min_virtual_time$

if $flowid$ queue is not empty **then**

  new_head_skb ⇐pointer to new head of flow queue

  $S_{flowid} = F_{flowid}$; $F_{flowid} = S_{flowid} + \frac{new_head_skb->len}{cl->weight}$

  $newfifoid$ ⇐ fifoid of new_head_skb based on its size

  Enqueue in $cl \rightarrow$ service level queue[$newfifoid$]

end if

Update Virtual time: Algorithm 1.

check eligibility of packet to make it to the service level queue. For every eligible packet, based on its size, we figure out which of the service level queues within a service level it would map to and then enqueue it.

This functionality is available as a loadable module with name sch_tsf2q-v.ko.

Chapter 6

Numerical Results

Figure 6.1 pictorially represents the testbed setup we used to carry out our experiments. Machine 2 has two Ethernet ports. One is connected to each to Machine 1 and another to Machine 3. The link between Machine 1 and Machine 2 is configured to run at 1 Gbps while the link between Machine 2 and Machine 3 is configured to run at 10Mbps. Machine 1 sends UDP packets to Machine 3. Output interface of machine 3 (i.e. eth1) is configured with our queuing discipline.

We use iperf [1] server on Machine 3 and send data from iperf clients running at Machine 1. Iperf allows up to pick any data rate within the maximum supported speed by outgoing interface. Thus, we can pump data into Machine 2 at rate much larger than 10Mbps. This results in queuing at Machine 2 outgoing interface eth1. We run multiple iperf servers at Machine 3 and multiple iperf clients at Machine 2. This enables us to test

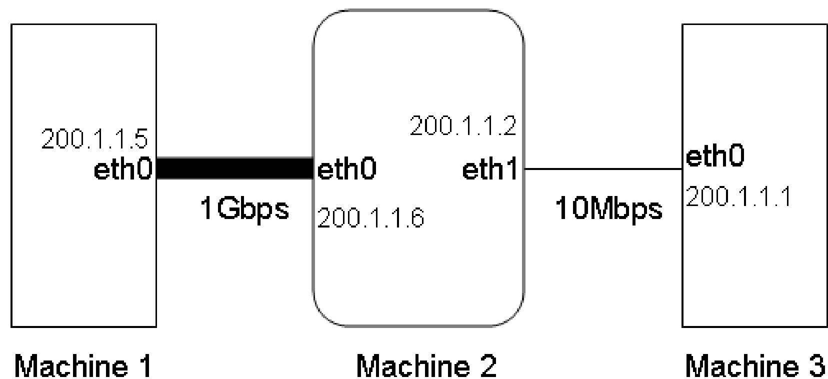

Figure 6.1: Testbed setup

how our implementation fairs.

However, while iperf allows us to choose UDP packet size for a particular client-server session, it does not support variable packet sizes within a client-server session. Thus, we wrote a simple UDP client-server program to achieve the same. Our UDP sender randomly generates packet sizes such that packet sizes 40, 1200 and 1500 bytes have 30% probability each. There is 10% chance that a packet size between 1-39 bytes, 41-1199 bytes or 1201-1499 bytes would be picked . This aims at mimicking the usual network traffic pattern of a small number of packet sizes dominating the traffic. Our UDP receiver keeps count of number of bytes received in a particular period to compute the throughput. User can choose a particular period at the time of starting receiver.

A detailed description of how to configure these machines to test our implementation is included in Appendix A.

We carried out following experiments to test our implementation:

- Start multiple flows with different weights at the same time. After a while, we terminate flows one by one to see its effect on throughput of then backlogged flows (say Scenario I).
- Start one (or very few flows) and allow it run for a while. Then we introduce new flows with different weights. We terminate these newly introduced flows after a while to see its effect on the remaining flows (say Scenario II).
- Run very large number of flows spanning across very few service levels (for TSFQ variants).

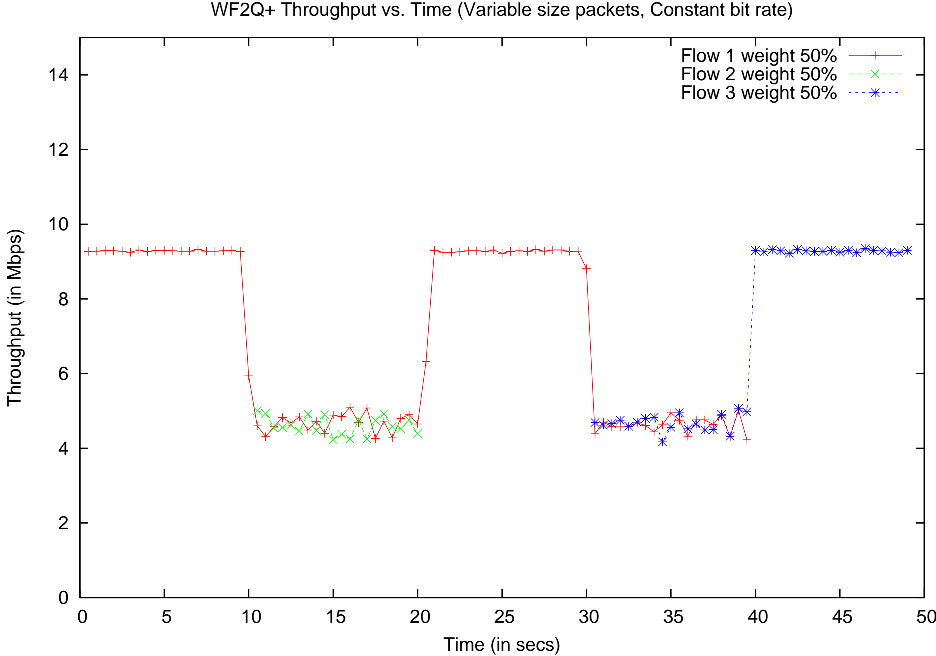

Figure 6.2: Throughput WF^{2}Q+ - Variable sized packets - 2 Flows

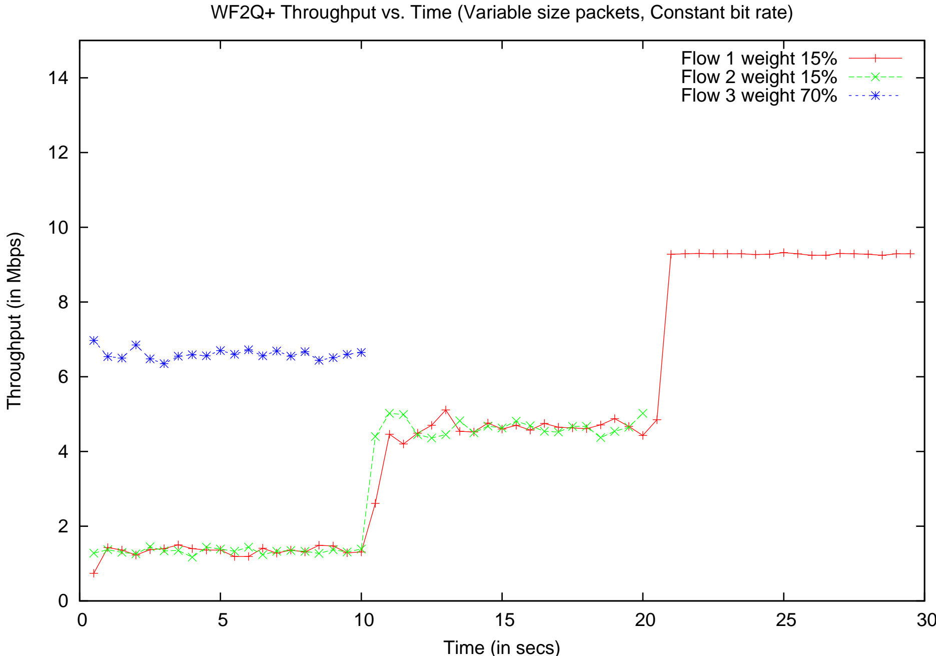

Figure 6.3: Throughput WF^{2}Q+ - Variable sized packets - 3 Flows

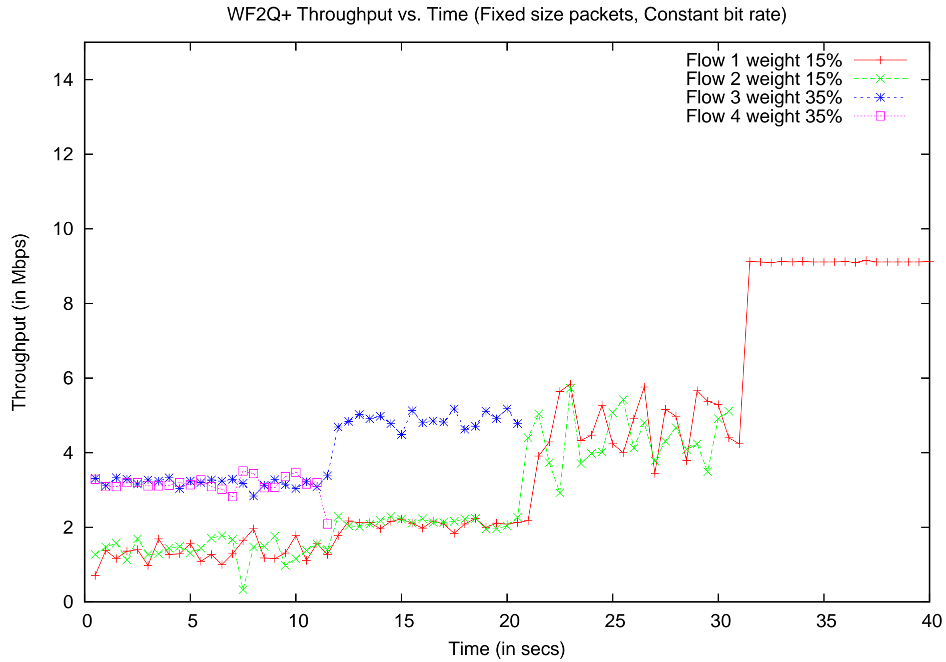

Figure 6.4: Throughput WF^{2}Q+ - Fixed sized packets - 4 Flows - Scenario I

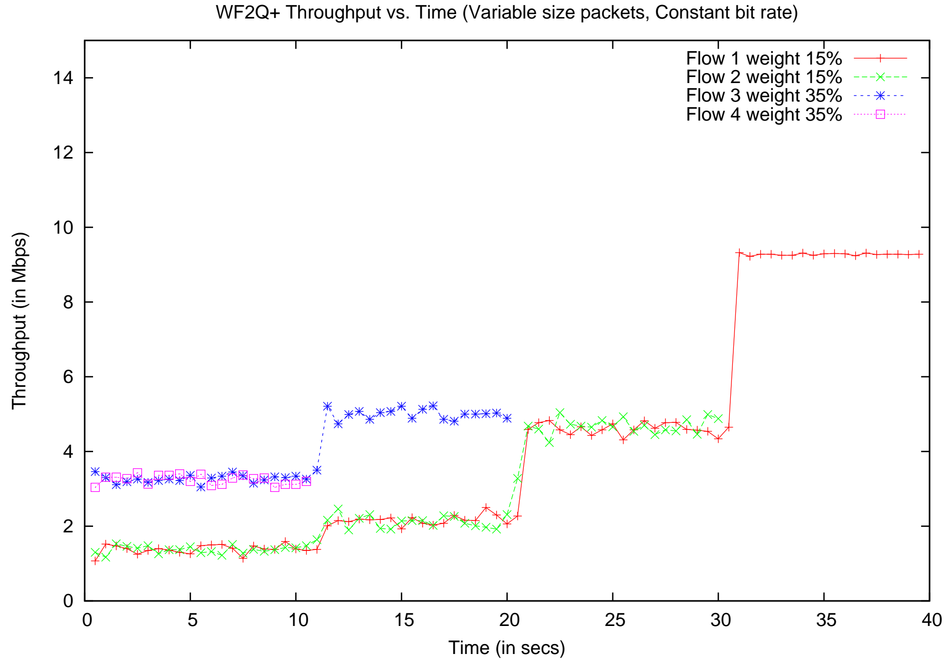

Figure 6.5: Throughput WF^{2}Q+ - Variable sized packets - 4 Flows - Scenario I

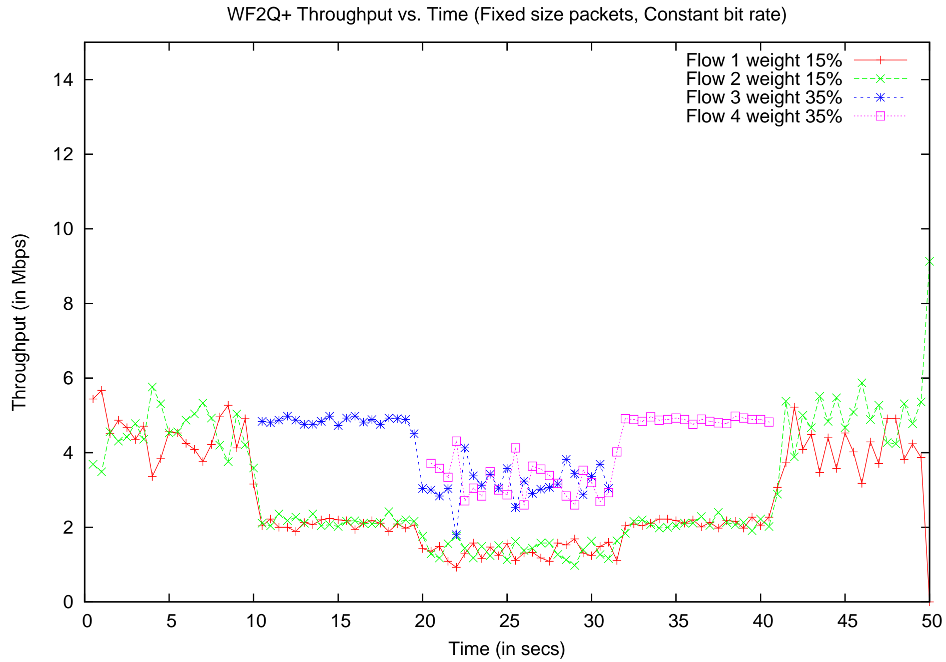

Figure 6.6: Throughput WF^{2}Q+ - Fixed sized packets - 4 Flows - Scenario II

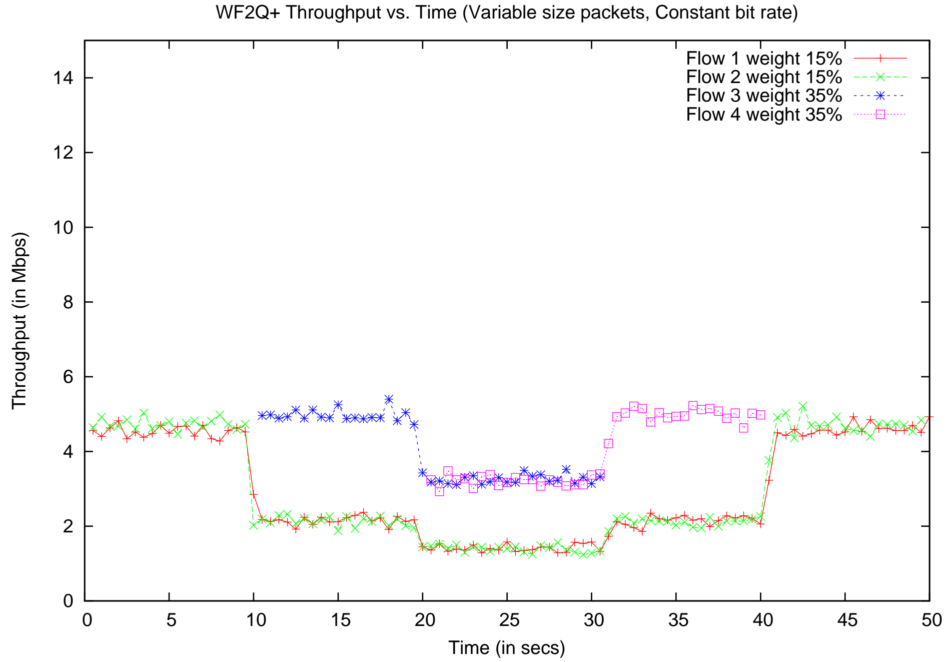

Figure 6.7: Throughput WF^{2}Q+ - Variable sized packets - 4 Flows - Scenario II

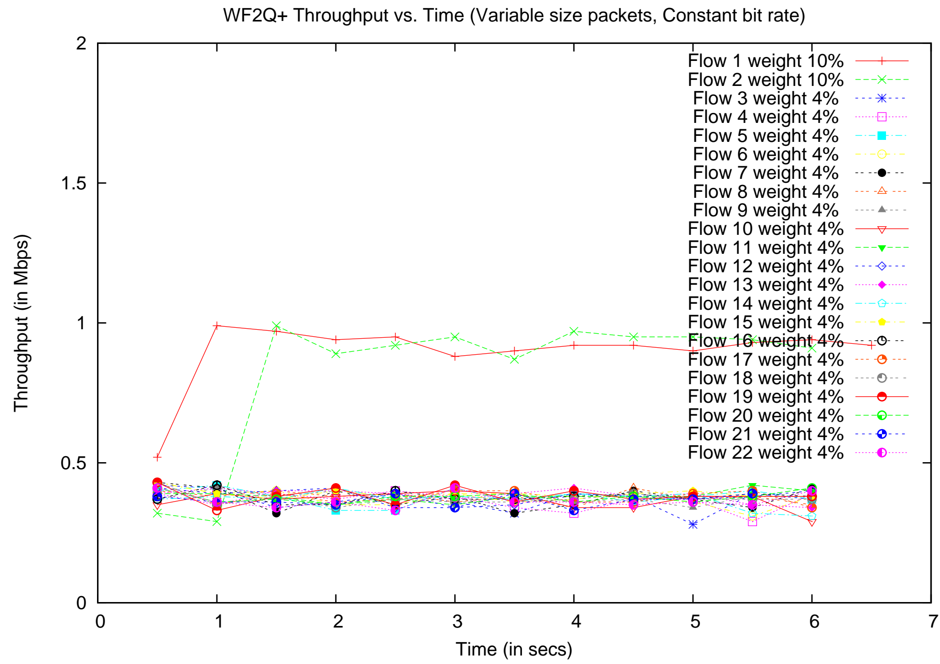

Figure 6.8: Throughput WF^{2}Q+ - Variable sized packets - 22 Flows - Continuous run

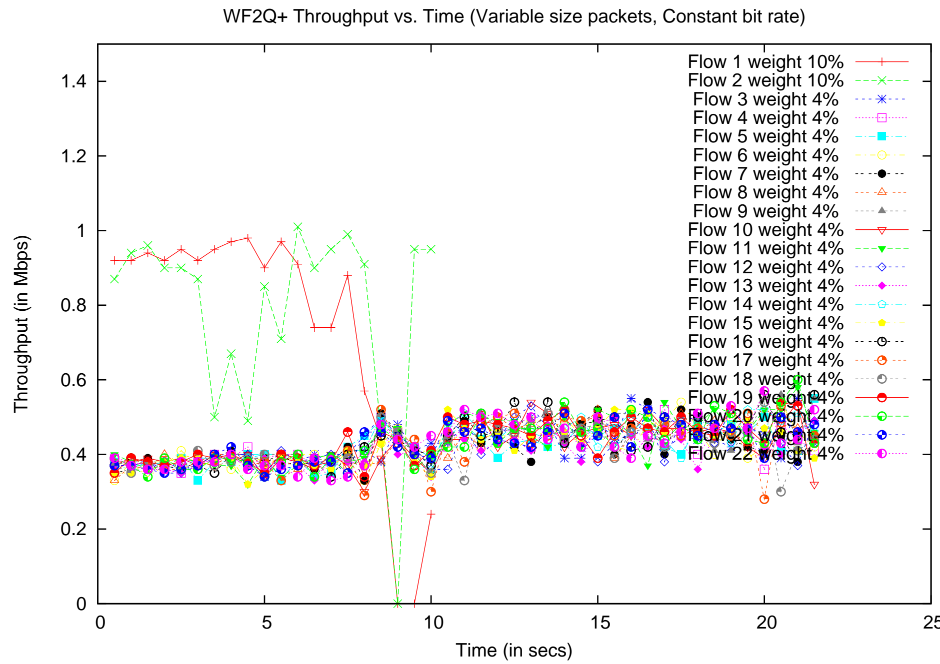

Figure 6.9: Throughput WF^{2}Q+ - Variable sized packets - 22 Flows - Flow 1 and 2 terminated after 10 sec

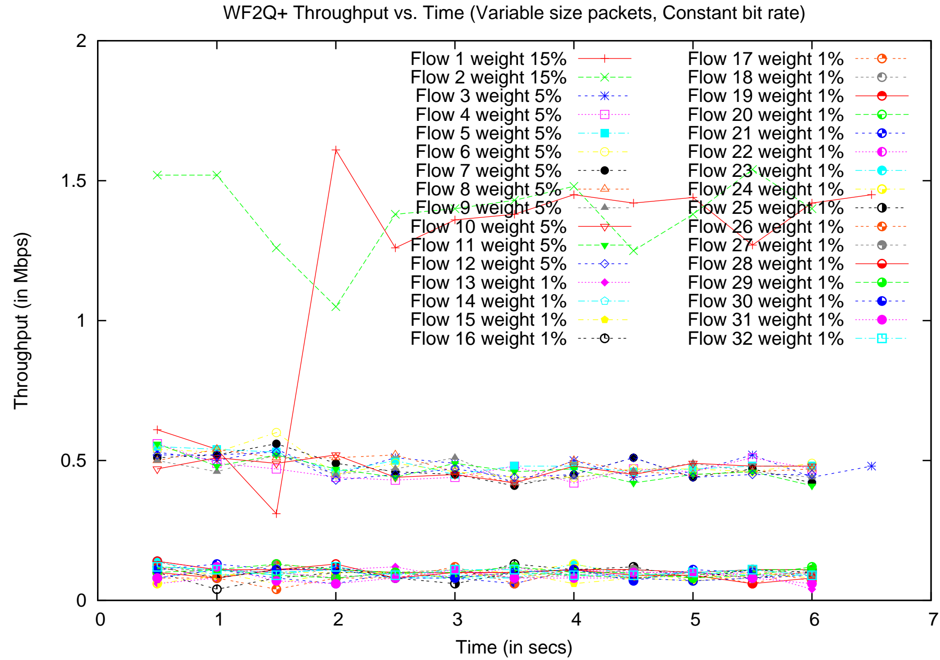

Figure 6.10: Throughput WF^{2}Q+ - Variable sized packets - 32 Flows - Continuous run

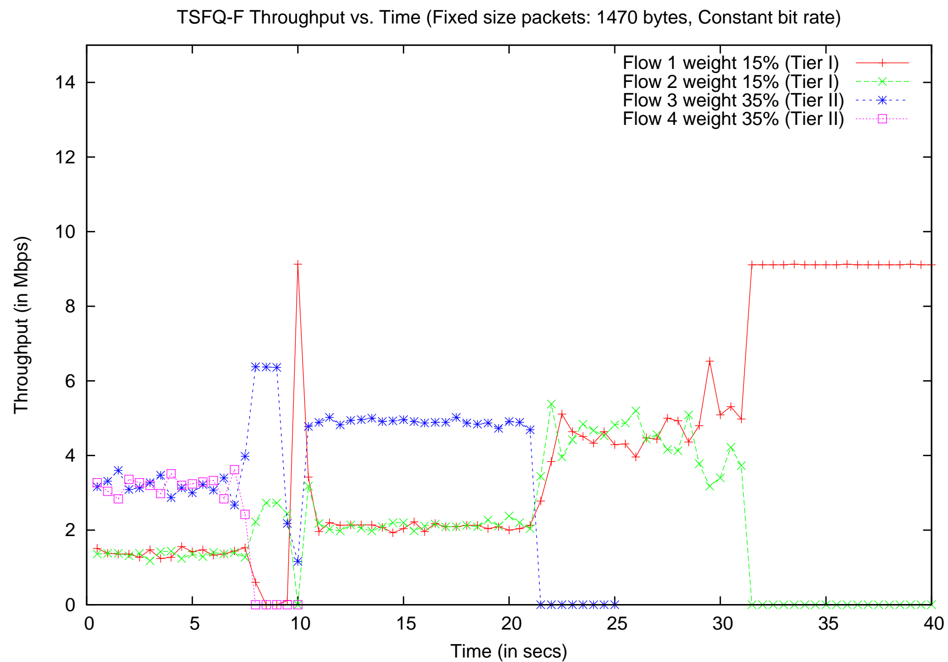

Figure 6.11: Throughput TSFQ-F - Fixed sized packets - 4 Flows - Scenario I

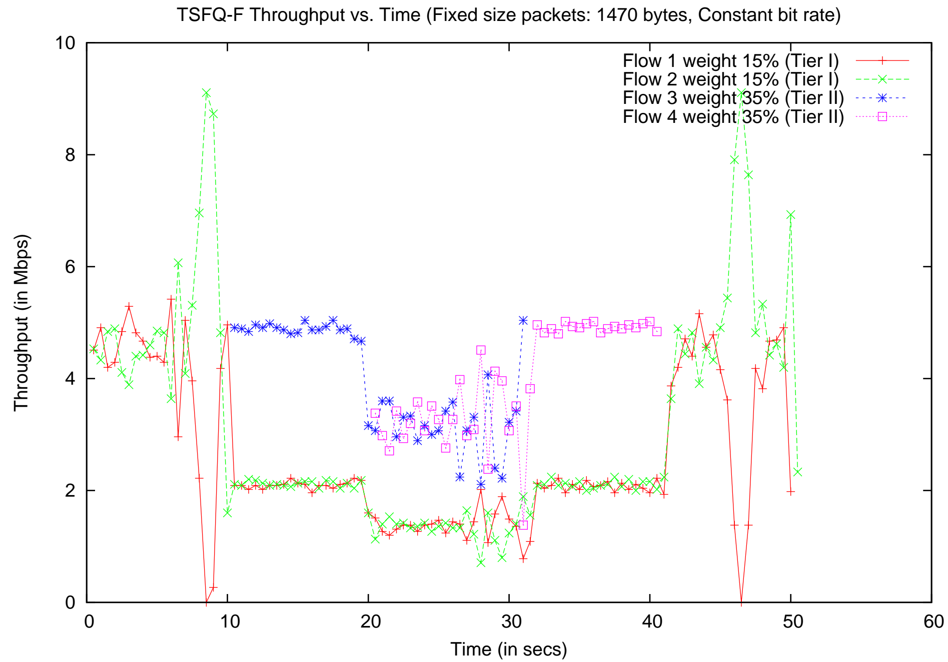

Figure 6.12: Throughput TSFQ-F - Fixed sized packets - 4 Flows - Scenario II

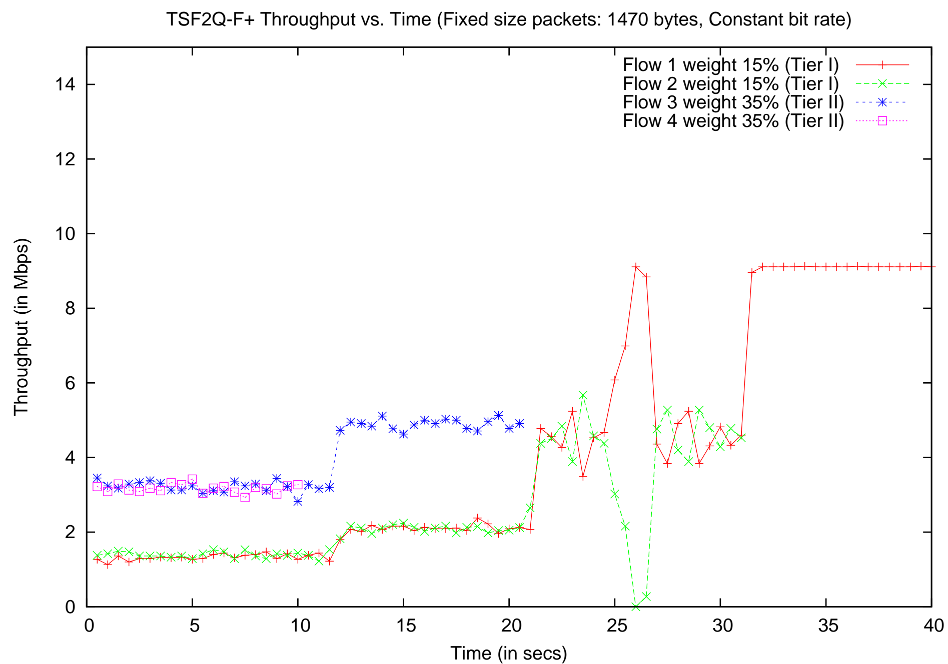

Figure 6.13: Throughput TSF^{2}Q-F - Fixed sized packets - 4 Flows - Scenario I

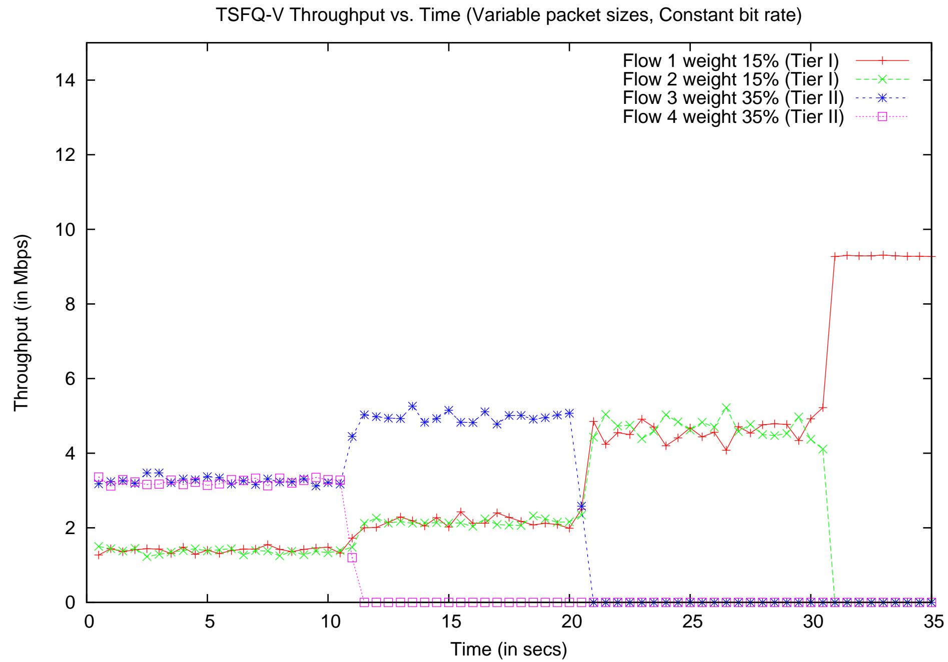

Figure 6.14: Throughput TSFQ-V - Variable sized packets - 4 Flows - Scenario I

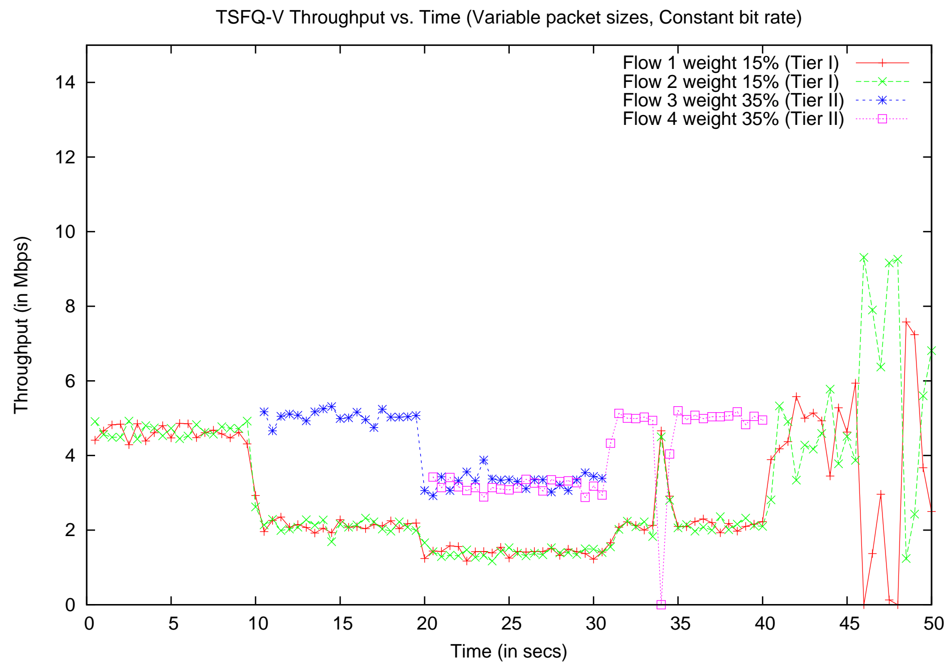

Figure 6.15: Throughput TSFQ-V - Variable sized packets - 4 Flows - Scenario II

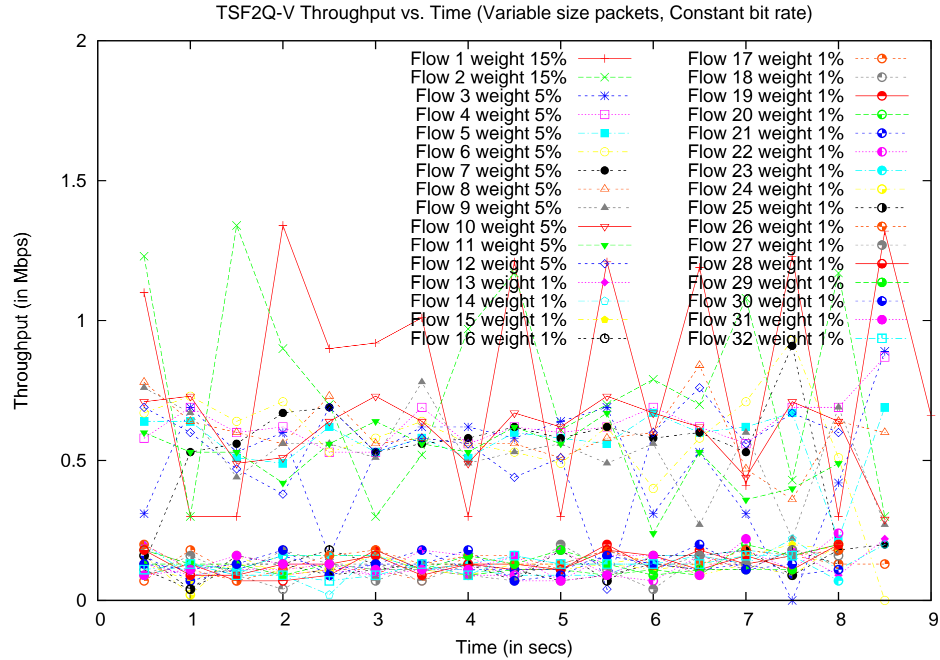

Figure 6.16: Throughput TSFQ-V - Variable sized packets - 32 Flows - Continuous run

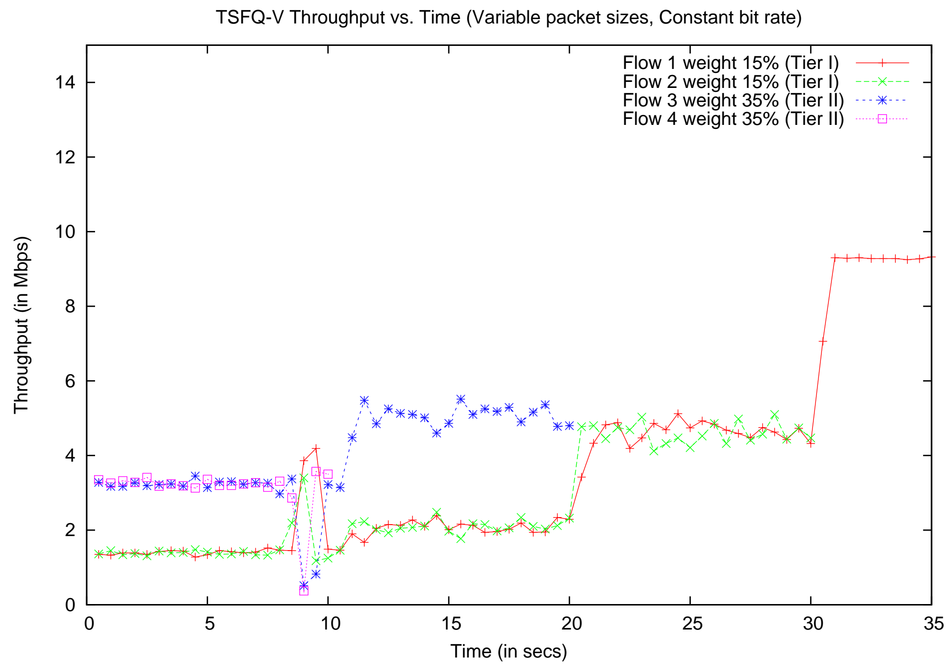

Figure 6.17: Throughput TSF^{2}Q-V - Variable sized packets - 4 Flows - Scenario I

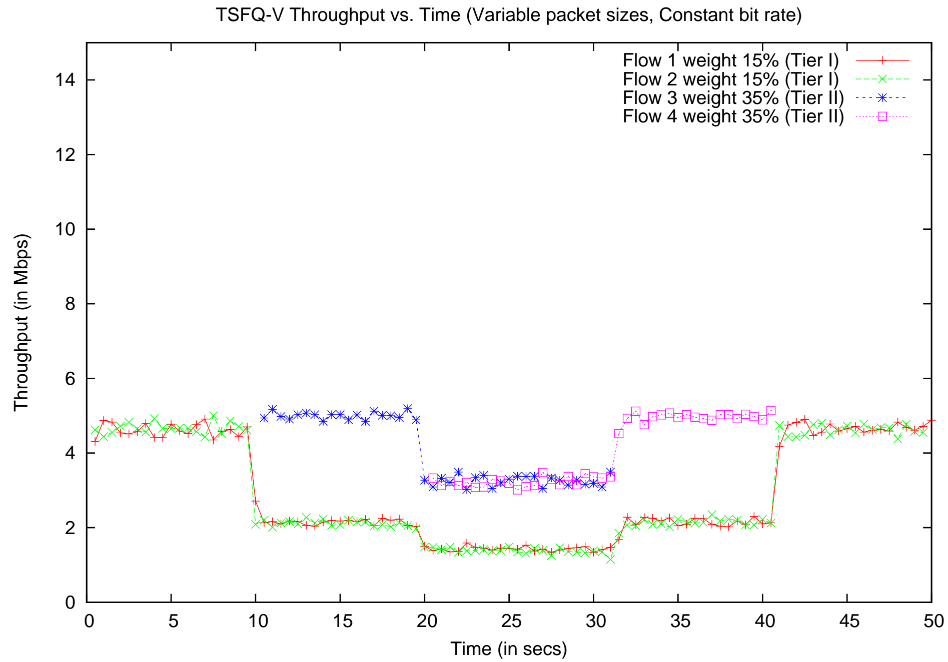

Figure 6.18: Throughput TSF^{2}Q-V - Variable sized packets - 4 Flows - Scenario II

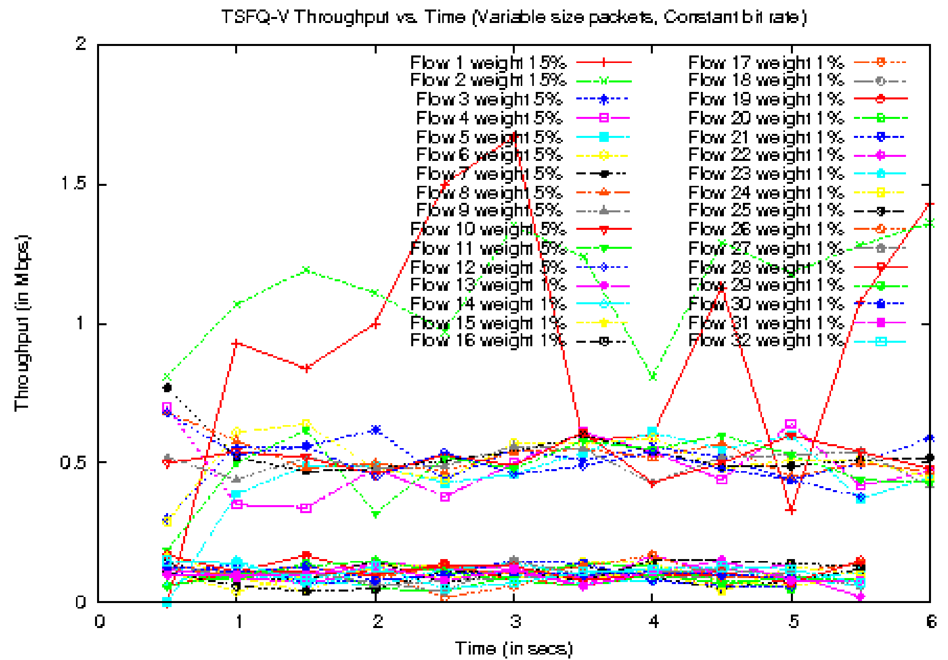

Figure 6.19: Throughput TSF^{2}Q-V - Variable sized packets - 32 Flows - Continuous run

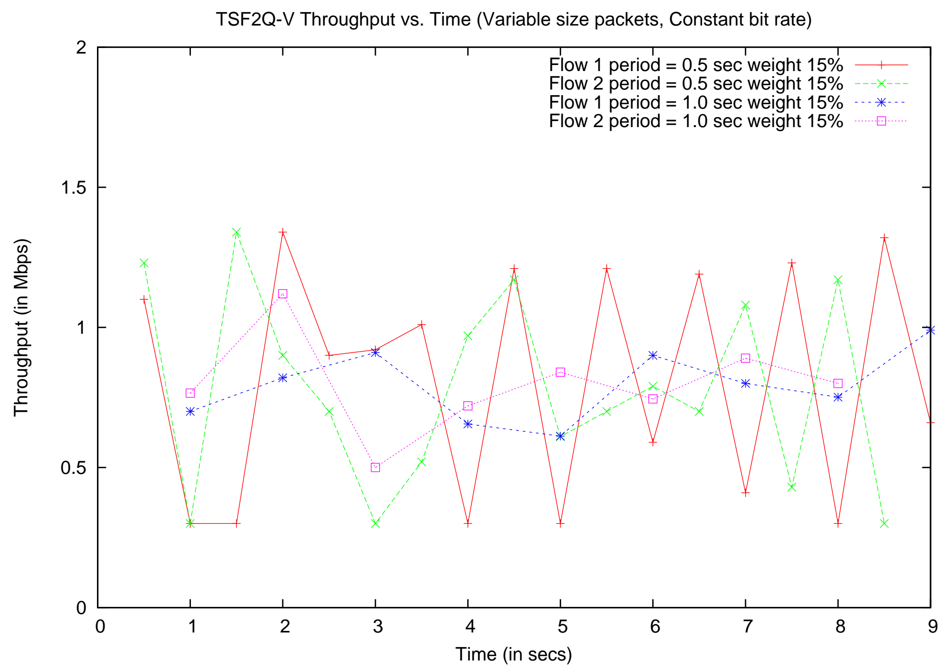

Figure 6.20: Throughput TSF^{2}Q-V - Variable sized packets - 32 Flows - Continuous run - Differnt periods

Chapter 7

Summary and Future Work

7.1 Summary

We implemented WFQ and four TSFQ variants in Linux Kernel as loadable modules. A Linux box can be easily configured to use our queuing discipline implementations to test their performance.

We carried out experiments on real testbed. Our experimental results indicate that TSFQ closely emulates previously proposed fair queuing disciplines. Even when we run the experiment with comparatively large number of flows (32 in our case), we observe that TSFQ does almost as good as WF^2Q+ in constant time.

7.2 Future Work

- We showed that TSFQ behavior resembles WF^2Q+ while managing to do the computations in constant time. Quantitative study needs to be done with regards to resource overhead (caused by quantization) versus time saved by TSFQ.
- We compared TSFQ variants with WF^2Q+ for our performance evaluation. TSFQ variants should be compared with existing Linux queuing discipline implementations like CBQ, HTB etc. for performance evaluation.
- In order to test our implementation, we generated UDP traffic based on observation about common packet sizes in real life networks. It would be interesting to run our implementation in real life network and since performance results.

Bibliography

[1] http://dast.nlanr.net/projects/iperf.

[2] http://devresources.linux-foundation.org/dev/iproute2/download.

[3] http://svana.org/kleptog/packet-shaping-howto.txt.

[4] www.kernel.org.

[5] J. C. R. Bennett and H. Zhang. Hierarchical packet fair queuing algorithms. In *Proc. ACM SIGCOMM'96*, pages 143–156, Aug. 1996.

[6] J. C. R. Bennett and H. Zhang. WF^2Q: Worst-case fair weighted fair queuing. In *Proc. IEEE INFOCOM'96*, pages 120–128, San Francisco, CA, Mar. 1996. IEEE.

[7] K. Thompson G. J. Miller and R. Wilder. Wide-area internet traffic patterns and characteristics. pages 10–23. IEEE Network, Nov/Dec 1997.

[8] A. Bhaskar G. N. Rouskas. Tiered service fair queuing (TSFQ): A Practical and Efficient Fair Queuing Algorithm. 2006.

[9] Z. Dwekat G. N. Rouskas. A practical and efficient implementation of WF^2Q+.

[10] S. Keshav. *An Engineering Approach to Computer Networking: ATM Networks, the Internet, and the Telephone Network*. Pearson Education, 2004.

[11] J. Nagle. On packet switches with infinite storage. In *IEEE Trans. on Comm.* IEEE, April 1987.

[12] J. Salim, H. Khosravi, A. Kleen, and A. Kuznetsov. Linux netlink as an ip services protocol. Technical report, 2003.

Appendices

Appendix A

Experiment setup

A.1 Testbed configuration

The experimental setup depicted in Figure 6.1 can be done by executing following commands:

Machine 1:

- ifconfig eth0 200.1.1.1/30 up
- route add default gw 200.1.1.2 # Set Machine 2 as default gateway
- /etc/rc.d/init.d/iptables stop

Machine 2:

- ifconfig eth0 200.1.1.2/30 up
- ifconfig eth1 200.1.1.6/30 up
- /etc/rc.d/init.d/iptables stop
- sysctl net/ipv4/ip_forward=1 # Enable ip forwarding
- ethtool -s eth1 autoneg off

- ethtool -s eth1 speed 10 duplex full # Set Ethernet to run at 10Mbps

Machine 3:

- ifconfig eth0 200.1.1.5/30
- route add default gw 20
- /etc/rc.d/init.d/iptabl
- ethtool -s eth1 auto
- ethtool -s eth1 speed 10 duplex full

A.2 Configuring Queuing disciplines

A.2.1 Configuring WF^{3}Q+

- An interface to associate the queuing discipline with the outgoing network interface in question (eth0 in our example).
  tc qdisc add dev eth0 root handle 1:0 wfq bandwidth 100Mbit avpkt 1000 cell 8

- A parent class, which is parent of every flow.
  tc class add dev eth0 parent 1:0 classid 1:1 wfq bandwidth 100Mbit prio 8 allot 1514 cell 8 maxburst 20 avpkt 1000 bounded

- For WF2Q+, we define 'tc class' such that each class corresponds to a separate competing flow. The following command defines a parent class which is associated with the outgoing interface in question (eth0 in this example).
  tc class add dev eth0 parent 1:1 classid 1:3 wfq bandwidth 100Mbit weight 90 prio 5 allot 1514 cell 8 maxburst 20 avpkt 1000 # flow 1
  tc class add dev eth0 parent 1:1 classid 1:4 wfq bandwidth 100Mbit weight 10 prio 5 allot 1514 cell 8 maxburst 20 avpkt 1000 # flow 2

- We would want to service the packets mapped to particular flow using FIFO.
  tc qdisc add dev eth0 parent 1:3 handle 30: pfifo # flow 1
  tc qdisc add dev eth0 parent 1:4 handle 40: pfifo # flow 2

- Tell the classifier to filter the packet with certain destination port to be mapped to specific class id. For example, the following configuration would map packet with destination port 6000 to class 1:3 and 6001 to class 1:4.

  tc filter add dev eth0 parent 1:0 protocol ip prio 1 u32 match ip dport 6000 0xffff flowid 1:3

  tc filter add dev eth0 parent 1:0 protocol ip prio 1 u32 match ip dport 6001 0xffff flowid 1:4

A.2.2 Configuring TSFQ variants

- An interface to associate the queuing discipline with the outgoing network interface in question (eth0 in our example).
  tc qdisc add dev eth0 root handle 1:0 tsfq bandwidth 100Mbit avpkt 1000 cell 8

- A parent class, which is parent of every flow.
  tc class add dev eth0 parent 1:0 classid 1:1 tsfq bandwidth 100Mbit prio 8 allot 1514 cell 8 maxburst 20 avpkt 1000 bounded

- For TSFQ, we define 'tc class' such that each class corresponds to a separate service level. Following command defines a parent class which is associated with the outgoing interface in question (eth0 in this example).
  tc class add dev eth0 parent 1:1 classid 1:3 tsfq bandwidth 100Mbit weight 90 prio 5 allot 1514 cell 8 maxburst 20 avpkt 1000 # flow 1

  tc class add dev eth0 parent 1:1 classid 1:4 tsfq bandwidth 100Mbit weight 10 prio 5 allot 1514 cell 8 maxburst 20 avpkt 1000 # flow 2

- Quantization step: Tell the classifier to filter the packet with certain destination port to be mapped to specific class id. For example, the following configuration would map packets with destination port 6000 or 6001 to class 1:3 and 6002 and 6003 to class 1:4.

  tc filter add dev eth0 parent 1:0 protocol ip prio 1 u32 match ip dport 6000 0xffff flowid 1:3

  tc filter add dev eth0 parent 1:0 protocol ip prio 1 u32 match ip dport 6001 0xffff flowid 1:3

  tc filter add dev eth0 parent 1:0 protocol ip prio 1 u32 match ip dport 6002 0xffff flowid

1:4
tc filter add dev eth0 parent 1:0 protocol ip prio 1 u32 match ip dport 6003 0xffff flowid 1:4

We have implemented 4 variants of TSFQ viz. tsfqf, tsf2qf, tsfqv, tsf2qv. The desired TSFQ variant can be configured by replacing tsfq in syntax noted above with the name of TSFQ variant. Succeeding section*s explain these TSFQ variants.

A.3 Running UDP flows

A.3.1 Running iperf flows for fixed packet sizes

Machine 3: iperf server

iperf -s -p <portno> -u

Machine 1: iperf client

iperf -c 200.1.1.5 -u -p <portno> -u -b <speed>

A.3.2 Running our UDP application for variable packet sizes

Machine 3: UDP receiver

udp_receiver <portno> <period>

Machine 1: UDP sender

udp_sender 200.1.1.5 <portno>

A.4 Observing results

Both iperf and our UDP client-server program periodically prints the throughput. Also, one can observe queue size on Machine 2 using dmesg command. We have put printk statements within our kernel module implementation to observe that.

Appendix B

Abbreviations

CBQ	Class-Based Queuing
FIFO	First In First Out
GPS	Generalized Process Sharing
HTB	Hierarchical Token Bucket
IP	Internet Protocol
QoS	Quality of Service
RR	Round Robin
SEFF	Smallest Eligible Virtual Finish Time First
SFF	Smallest Virtual Finish Time First
TCP	Transmission Control Protocol
TSFQ-F	Tiered Service Fair Queuing - Fixed packet size
TSFQ-V	Tiered Service Fair Queuing - Variable packet size
TSF^{2}Q-F	Worst-case Fair Tiered Service Fair Queuing - Fixed packet size
TSF^{2}Q-V	Worst-case Fair Tiered Service Fair Queuing - Variable packet size
UDP	User Datagram Protocol
WFI	Worst-case Fair Index
WFQ	Weighted Fair Queuing
WF^{2}Q	Worst-case Fair Weighted Fair Queuing
WF^{2}Q+	Modified Worst-case Fair Weighted Fair Queuing
WRR	Weighted Round Robin