修士論文

題目

ネットワーク中間ノードに於けるフロー制御に関する研究

指導教員

所 真理雄　客員教授

天野 英晴　助 教 授

慶應義塾大学 大学院

理工学研究科 計算機科学専攻

1997年度

89624262

久曽神 宏

修士論文要旨

専　攻	計算機科学	学　籍 番　号	89624262	フリガナ 氏　名	キュウソジン ヒロシ 久曽神 宏

(論文題名)

ネットワーク中間ノードに於けるフロー制御に関する研究

(内容の要旨)

（内容の要旨は約25行程度で記入のこと）

本研究では，Weighted Fair Queueing (WFQ) を用いることにより，ネットワーク中間ノードに於けるフロー間の公平性を実現する．フローを孤立化させることにより，あるフローの振舞いが他のフローへ影響を及ぼすというようなことは起こらない．既存のスケジューリングアルゴリズムは，現時点から未来に向けての公平性のみを考慮していたのに対し，本研究で提供する枠組は，各フローが発生して以来の履歴情報を用いて公平性を提供する．また，パケットドロップアルゴリズムとして Random Early Detection (RED) を用いることにより，輻輳制御を行なうと共に，一時的なバーストトラフィックへの対応，対話型通信の保護，及び遅延の減少を実現した．

一方，データリンク層に Ethernet のような共有メディアを用いている場合にも，各ノードの出力キューに於いてトークンバケットによるトラフィックシェーピングを行なうことにより，ノード間及びフロー間での公平性を実現した．周期毎に生成されるトークンの量を動的に変更することにより，ネットワークの状態に動的に適応可能なトラフィックシェーピングを実現した．

ネットワーク中間ノードに於けるフロー制御に関して過去に行なわれてきた研究は，単にアルゴリズムとして提案されているものや，シミュレーションにより実証しているものがほとんどであるが，本研究では PC ルータ上に実装すると共にその評価を行なった．

本研究で設計，実装，評価を行なったアルゴリズムを用いることにより，ネットワーク中間ノードに於いて，コネクション，トラフィック及びネットワークの状態に動的に適応し，各フローに対して公平かつ効率の良い転送を実現することが可能となる．本研究で提供する枠組は，送信者，受信者，及びネットワークに対する前提を持たない為，柔軟なフロー制御をネットワーク中間ノードに於いて行なうことができる．

慶應義塾大学大学院理工学研究科前期博士課程

Summary of M. S. Thesis

Major	Student No.	SURNAME, Firstname
Department of Computer Science	89624262	KYUSOJIN Hiroshi

Title

Flow Control Mechanism at the Gateway

Weighted Fair Queueing (WFQ) provides fairness between flows at the gateway. To isolate flows, per-flow protection is realized. Compared with previous proposed scheduling algorithms which do not keep the state, our algorithm keeps per-queue historical information since the creation of the flow. Then, it is possible to provide long term fairness that consider past states. Further, using Random Early Detection (RED) mechanism, congestion control of burst traffic, protection of interactive communication, and decrease of delay is achieved.

On the other hand, our new traffic shaping mechanism based on the token bucket algorithm can provide fairness between flows or nodes, when shared media like Ethernet is used as a data-link layer. The quantum of the token created at each period is dynamically changed according to the state of the network.

Although most proposed algorithms for flow control at the gateway are only based on analysis and simulation, we have actually implemented our algorithm on the PC-based gateway and evaluated them.

By applying our approach, it is possible to adapt to the connection, network state, and traffic dynamically and enable fair and efficient forwarding of packets. Further, our approach has no assumption for senders, receivers, gateways as the network middle node, and the network as a data-link layer, so the flexibility of flow control at the gateway can be realized.

Graduate School of Science and Technology, KEIO UNIVERSITY

(The summary of thesis should be written within 35 lines)

目　次

付録

図 目 次

表　目　次

第1章

序論

1.1 はじめに

コネクションレスのパケットネットワークには，ネットワークへの接続端末やネットワーク自身に対する前提がほとんど無い為，柔軟性やロバスト性と言った利点がある．しかしその一方で，ネットワークの状態に従ってパケットの送信量を動的に制御する為の “フロー制御” が必要となる．いろいろな処理速度の端末がいろいろな帯域幅を持つネットワークやルータを通して通信する非一様なネットワーク環境に於いて，フロー制御は不可欠な機能である．フロー制御には2通りあり，1つは始点と終点の間という end-to-end で行なうもので，もう1つはルータやゲートウェイ等のネットワークの中間に位置するノードで行なうものである．

end-to-end のフロー制御は，送信側の端末と受信側の端末の処理速度が異なる場合や，送信側の出力リンクに比べて中間リンクの帯域幅が狭い場合等に必要となる．インターネットの標準的なトランスポートプロトコルである TCP は，この end-to-end のフロー制御を実現するものであり，パケットロスを素早く発見するように拡張した TCP-Reno [Jac88] や，輻輳回避に関する改良を行なった TCP-Vega [Bra95] 等，数多くの提案が行なわれているだけでなく，実際に運用されている．

一方，ネットワーク中間ノードに於けるフロー制御に関して過去に行なわれてきた研究は，単なるアルゴリズムとして提案されたものであったり，シミュレーションを基に提案されているものがほとんどである．これは，ルータやゲートウェイ等のネットワーク中間ノードの多くが専用のハードウェアや専用のオペレーティングシステムを用いていることから，実装して評価を取ることが困難な為と考えられる．しかし，近年のパーソナルコンピュータ (Personal Computer，PC) の性能の向上及びネットワークカードの高速化に伴い，PC をルータやゲートウェイとして用いることが現実的となってきた．本研究では，過去に提案されてきた幾つかのアルゴリズムを PC ルータ上に実装すると共に評価を行ない，その問題点を解決する．

ネットワーク中間ノードに於いて，enqueue されるパケットは送信者に依存する．しかし，enqueue されたパケットをどのようなスケジューリングアルゴリズムに従って dequeue するかは各ノードの裁量によって決定できる．また，そのノードの処理能力以上のパケットが流れ込んできた場合や，入力リンクに比べて出力リンクの帯域幅が狭い場合等に行なうパケットドロップアルゴリズムに関しても，各ノードに任されている．

本研究では，ノードに於けるスケジューリングアルゴリズムとして Weighted Fair Queueing

(WFQ) を用いることにより，フロー間での公平性を実現する．通常，WFQ では Tail-Drop によるパケットをドロップを行なうが，本研究ではパケットドロップアルゴリズムに大きく 2 つの変更を加えた．1 つは，パケットドロップを行なうキューを選択する際に，キュー毎の履歴情報を利用する点である．Fair Queueing に於いてパケットドロップの必要が生じた際，Tail-Drop はその時点で enqueue されているパケット数が最大のキューからパケットをドロップする．この為，バーストトラフィックを不当に扱うという問題がある．本実装では，各キューがスケジューリングされて以来どの程度のパケットを送信したかという履歴情報を基にパケットドロップを行なうことにより，このような問題を回避する．もう 1 つは，Random Early Detection (RED) を用いる点である．Tail-Drop ではバッファが溢れるまでは全く制御を行なわない為，バッファが一杯またはそれに近い状態が発生しやすい．このような状態では，小量のバーストトラフィックに対してもパケットドロップが発生しやすくなる．また，同時に複数のコネクションに対してパケットドロップを行なった結果，フロー間での同期現象 (global synchronization) を誘発する可能性もある．輻輳の初期段階に於いてランダムにパケットドロップを行なう RED を用いることにより，このような問題は回避される．一方，RED では適応型のフローと非適応型のフローを完全に分離することができないという問題が存在するが，本研究では WFQ によってフロー間での公平性を提供する為，この問題は発生しない．

データリンク層に於いて Ethernet のような共有メディアが使われている場合，ネットワーク中間ノード間で帯域幅の公平な共有を行なわねばならない．Ethernet は LAN に於ける基本技術として現在広く利用されているが，100Mbps といった広帯域規格の登場と共に，ネットワーク中間ノードの出力リンクが Ethernet のような共有メディアである場合は多々あると考えられる．しかし，一方のノードから TCP のような適応型のフローが，他方のノードから UDP のような非適応型のフローがそれぞれ同時に Ethernet 上で通信を行ない，帯域に公平な共有を行なおうとしても，適応型フローである TCP は帯域幅をほとんど確保することができない．これは，適応型フローと非適応型フローが共有メディア上で競合することにより発生する問題である為，ノードの出力リンクが共有メディアの場合には，ノード間だけの問題ではなく，1 つのノードを通過するフロー間でも発生する問題となる．

本研究では，各ノードの出力キューに於いてトークンバケットアルゴリズムを用いたトラフィックシェーピングを行なうことにより，出力リンクが共有メディアの場合にも，ノード間及びフロー間での公平性を実現する．本実装では，周期毎に与えるトークンの量を動的に変更することにより，コネクション，トラフィック，及びネットワークの状態に応じたトラフィックシェーピングを実現する．

1.2 目的

本研究の目的は，ネットワーク中間ノードに於いて，コネクション，トラフィック及びネットワークの状態に動的に適応することにより，各フローに対して公平かつ効率の良い転送を実現することである．

コネクションレスのパケットネットワークでは，送信者，受信者，及び中間ノードとしてのネッ

トワーク接続端末や，データリンク層としてのネットワーク自身に対する前提をほとんど必要としない．この柔軟性を失わぬよう，本研究で提供する枠組も end-to-end に於ける前提条件を持たない．これにより，ネットワーク中間ノードに於ける柔軟なフロー制御が可能となる．

1.3 本論文の構成

本論文は全 8 章から構成される．2 章では研究の背景として，既存のネットワーク中間ノードに於けるフロー制御の問題点を述べると共に，本研究のアプローチ，実装の概要等を説明する．3 章ではノードに於ける制御に関して，4 章では共有メディアに於ける制御に関して，それぞれ設計，実装，評価を述べる．5 章では評価結果を踏まえた上での考察を行なう．6 章では関連研究を紹介し，7 章で今後の課題についてふれ，8 章で結論を述べる．

第2章

背景

2.1　ノードに於ける問題

2.1.1　Fair Queueing

ネットワーク中間ノードでは，パケットの dequeue やドロップを行なう為のキュー管理機構が必要となる．最も単純なキュー管理機構は，1 つのネットワークインタフェースに対して 1 つの出力キューのみを持ち，最初に受け取ったパケットを最初に enqueue する “First In First Out (FIFO)” 方式である．通常の市販ルータや，オペレーティングシステムとして UNIX を利用している端末はこの FIFO 方式によりネットワークサービスを提供している．

FIFO 方式は複数のコネクションを 1 つのキューに enqueue する為，実装が容易であり，また管理する状態がほとんど無いと言った利点がある．しかし FIFO 方式ではコネクション毎のサービスの質 (Quality of Service，QoS) を保証するこは難しく，送信元毎に送信パケット量を調整したとしても，あるコネクションが大量のパケットを送り始めると，キューの長さが伸びてバッファが溢れてしまう．このような事態を避ける為には，図 2.1 に示すように送信元毎にキューを分ければ良い．

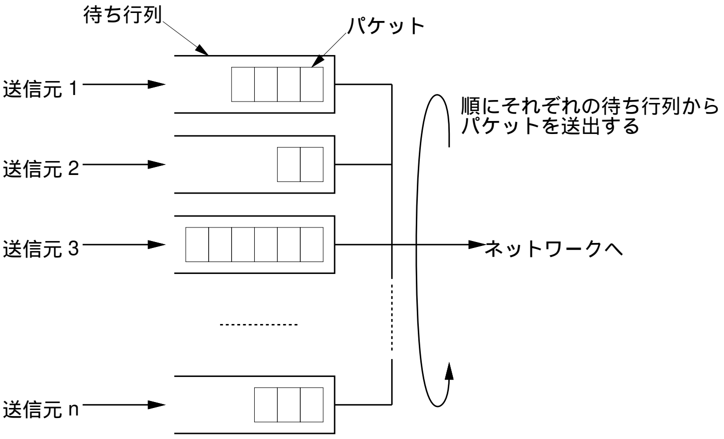


図 2.1: Fair Queueing

これをコネクションの孤立化といい，このような手法を “Fair Queueing” [Nag87] という．パケットは送信元のパケット毎に公平になるよう，それぞれの待ち行列から順に Round Robin [Hah86] で送出される．Fair Queueing を用いることにより，リンクの帯域幅を仮想的にではあるが公平に分割することができる．

2.1.2　バーストトラフィック

Tail-Drop によるパケットドロップが必ずしも公平ではないことを確認する為，図 2.2(左) のような環境に於いて，端末 1 から TCP のフローを，端末 2 から UDP のフローを，それぞれ端末 4 に向けて同時に送信し，ルータ R1 の出力リンクを共有するという簡単な実験を行なった．Fair Queueing を行なうルータ R1 のバッファサイズは 64K バイトであり，図 2.2 (右) は TCP と UDP 各フローのバッファ使用量を示している．縦軸はキューサイズ (Kbytes)，横軸は時間 (秒) である．なお，TCP はパケットドロップに応じて転送レートを変更してしまう為，このルータ R1 では TCP のパケットがドロップされないように特殊な設定を行なった．

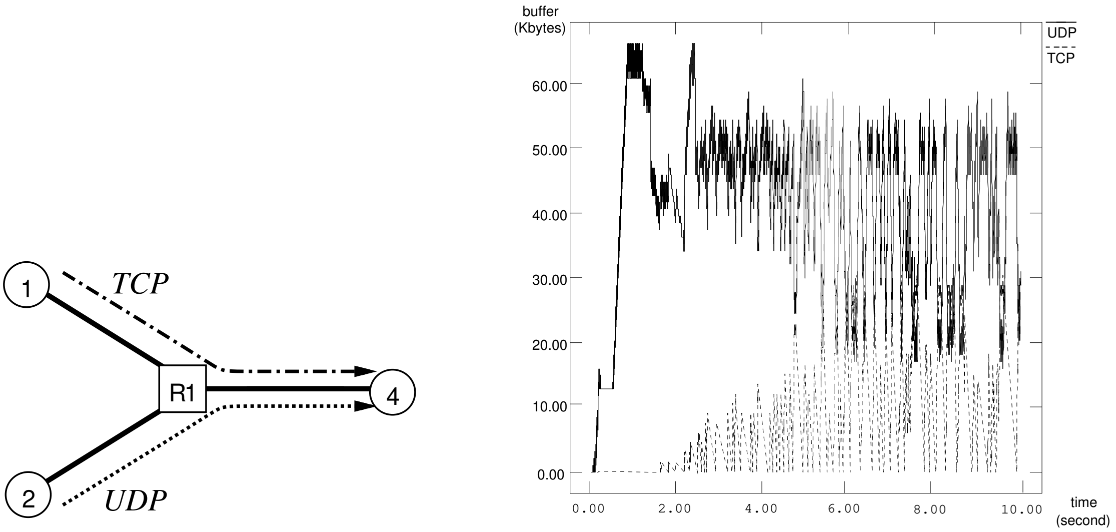


図 2.2: バッファ使用量 (左:環境，右:結果)

UDP のバッファ使用量が圧倒的に高いにも関わらず，TCP のバッファ使用量が上回る瞬間が存在することに注目して欲しい．Tail-Drop を行なうルータやゲートウェイ等のネットワーク中間ノードでは，最大キューサイズを設定しており，キューサイズがその閾値に達すると，キューサイズが閾値以下になるまでは全ての到着パケットをドロップする．Fair Queueing に於いてパケットドロップの必要が生じた場合には finish number が最大のパケット，つまりは最も長いキューからパケットがドロップされる．通常の Fair Queueing 及び Tail-Drop を用いている限り，この例に於いて TCP パケットのドロップは簡単に発生する．TCP は 1 つのパケットドロップで輻輳回避状態に入り，送信量を減らして適応しようとするのに対し，UDP は幾らパケットがドロップされても関知しない為，ドロップするパケットの選択は非常に重要な問題である．

別の例として，適応型だが Round Trip Time (RTT) が非常に大きな為にバーストトラフィックとなってしまうフローと，複数の非適応型のフローとが共存するような環境を考えてみる．これらのフローが統合されるルータで Fair Queueing のパケットドロップアルゴリズムが用いられると，ほとんど帯域を使っていない適応型フローのパケットがドロップされる可能性が高くなってしまう．

このように Fair Queueing では，現在のキュー情報を元にパケットドロップを行なう為，バーストトラフィックを不当に扱うという問題がある．

2.1.3 バッファの空き領域

Tail-Drop では，キューサイズがある閾値に達するとパケットをドロップする．つまり裏を返すと，バッファが一杯又はそれに近い状態にならない限り何もしないことになる．この為，バッファが一杯に近いという状態に成りやすく，そのような状況では小量のバーストトラフィックが発生した場合でもすぐに閾値を越えて，パケットがドロップすることになる．このような状態が，コネクション間での公平性の提供を困難にしているとも考えられる [Flo91]．

また，バッファが溢れるまでパケットをドロップしないと，バッファが溢れた瞬間に，ほぼ同時に複数のコネクションに於いてパケットがドロップされることになる．通常，適応型のフローはパケットドロップによって輻輳を検知する為，トラフィック量が一気に減少する．ネットワーク全体でのトラフィック量が減少すると，それを関知したコネクションはトラフィック量を増加し，再び輻輳状態へと戻ってしまう．このように，トラフィックがフロー間で同期する現象を “global synchronization” という．

定常的にバッファが一杯に近いという状態から発生する問題はこれだけではない．Fair Queueing に於いても，各キューでは FIFO スケジューリングが行なわれている．そのキューに enqueue されているパケットの数が増えれば増えるほど，dequeue までにかかる時間は長くなる．つまり，バッファの使用率は “遅延” として，経由するルータの個数分だけそのフローに影響を与える．当然，遅延は少なければ少ないほど望ましいが，特に対話型通信のアプリケーション等にとっては重要な問題である．

2.2 共有メディアに於ける問題

出力先のリンクが Ethernet のような共有メディアの場合，TCP がトランスポート層に於いて全二重コネクションを仮想的に提供したとしても，データリンク層に於いては CSMA/CD (Carrier Sense Multiple Access with Collision Detection) [ISO93] によって up link 及び down link によるメディアの共有が行なわれている．根本的に，Ethernet は公平性の提供や保証といったものができないメディアである [Bog88]．

Ethernet のような共有メディアを，UDP のような確認応答による適応を行なわないフローと，逆に適応を行なう TCP が混在する状況を考えてみる．データリンク層が許す限りのパケットを Ethernet 上に流すと，RTT 時間経過後，確認応答パケットと送信パケットが衝突を起こしてし

まう．確認応答の得られない TCP は輻輳が発生したと判断して送信量を減少させる [Cla82]．しかし，その減少分の帯域は UDP によって使われてしまい，いつまでたっても TCP は確認応答を得ることができない．結果として，TCP が自ら帯域を放棄するような状態が起こる．

つまり，複数のノードが共有メディア上で同時に通信を行なう場合，パケットの送信機会が各ノードに均等に与えられたとしても，非適応型のフローを発生するノードが存在すると，帯域幅の公平な分割は不可能である．また，これはノード間に於ける問題だけではなく，あるノードの出力リンクを適応型のフローと非適応型のフローが共有するといったような単一のノードに於いても発生する問題である．この問題を回避するためには，共有メディア上に確認応答を受け取る為の帯域幅を残さねばならない．

2.3 本研究の方針

これまで説明してきたように，ネットワーク中間ノードに於けるフロー制御を既存の機構を組合せて行なう際，以下に挙げるような問題がある．

1. ノードに於ける問題点
    - Fair Queueing のパケットドロップアルゴリズム
        - バーストトラフィックを不当に扱う
    - バッファが一杯になるまでパケットをドロップしない
        - 小量のバーストトラフィックでパケットがドロップする
        - フロー間での同期現象 (global synchronization) の発生
        - 予測不可能な遅延の増加
2. 共有メディアに於ける問題点
    - 適応型のフローと非適応型のフローを発生するノード間で，公平な帯域の共有ができない
    - 1 つのノードを適応型のフローと非適応型のフローが通過する場合，フロー間で公平な帯域の共有ができない

以下の項では，それぞれを解決する為の枠組と，実装の概要を述べる．

2.3.1 ノードに於ける制御

バーストトラフィック

本研究では，各キューに重み付けを行なうことができる Fair Queueing である “Weighted Fair Queueing (WFQ)” [Dem89] を実装する．どのキューに重み付けを置くかはポリシーの問題とな

るが，例えば回線使用に支払っている金額であったり，通信量やその頻度であったりする．WFQに於いて，各フローが発生して以来の履歴情報を基にパケットドロップを行なうキューを決定することにより，バーストトラフィックを不当に扱う問題を解決する．

また，Fair Queueing ではコネクション毎の状態を保持することになるが，コネクションの数が増大するに従って管理コストが膨大になってしまう．加えて，フローからキューへの対応付けを行なうためのメモリ参照が多いため，高速ネットワークに適しているとは言えない．本研究では，“Stochastic Fairness Queueing” [McK91] を用いることにより，ハッシュ関数を用いてコネクションを固定個のキューに対応付け，この管理コストの低減を実現する．また，各キューに対するスケジューリングを，単純な Round Robin ではなく “Deficit Round Robin” [Shr95] で行なうことにより，コネクション毎にパケットサイズが異なる場合や，1つのコネクション内でもパケットサイズが固定長でない場合等にも高い公平性を提供を実現する．

バッファの空き領域

バッファが溢れていない場合にもパケットをドロップする Early-Drop スケジューラとして，“Early Random Drop (ERD)” [Has89] や “Random Early Detection (RED)” [Flo93] が提案されている．キューサイズが drop level と呼ばれる閾値に到達すると，定数値の確率によってパケットをドロップする ERD に対して，RED は瞬間のキューサイズでは無く平均キューサイズがある閾値に達すると，平均キューサイズを変数とする線形関数で表されたパケットドロップ率に従ってパケットドロップを行なう．ERD や RED では非適応型のフローの特定はできるが，パケットをランダムにドロップする為，適応型のフローが非適応型のフローの影響を全く受けないようにすることはできない [Zha89, Flo93]．本研究では RED を WFQ と組み合わせて用いることにより，この問題の解決をする．

RED ではランダムにパケットをドロップすることによって，完全な輻輳状態に陥る以前に送信者に対する輻輳通知を行なうことができる．結果として，各コネクションに対して輻輳の通知を異なるタイミングで行なうことができる為，フロー間での同期現象 (global synchronization) を避けることができる．

また，enqueue されているパケットの数を少なく保つことにより，バーストトラフィックに対応できることに加え，ネットワーク中間ノードで損失するパケット数の減少，対話型通信に於ける遅延の減少が可能となる [Bra97]．

2.3.2 共有メディアに於ける制御

共有メディアに於ける公平性の問題は，トークンバケットアルゴリズム [Tur86] を用いて，出力キューに於けるトラフィックシェーピングを行なうことにより解決する．トークンバケットアルゴリズムでは，バケットには一定の間隔で生成されるトークンが溜められる．トークンバケットの例を図2.3に示す．

いまバケットには3つ分のトークンが溜められている．端末から5つのパケットが送信されたとする．トークン1つに付き1つのパケットが送出可能だとすると，まず最初の3つのパケットが同時に送出される．そして残りの2つは新たにトークンが生成されるまで待たされる．つまり

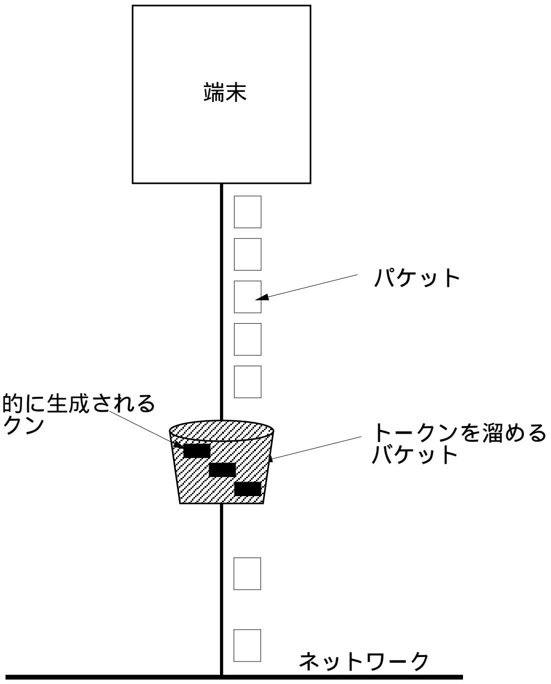


図 2.3: トークンバケット

トークンバケットアルゴリズムに於いて，バケットの深さは 1 度に送ることのできるバーストの大きさを制限していることになる．また，トークンの生成レートは長期的なパケットの送出レートを決定することになる．本研究では，生成するトークンの量をネットワークの輻輳状態に応じて変更することにより，常に確認応答パケットを受け取るだけの余裕を残す．

2.3.3 実装

本実装では，BSD 系 UNIX である FreeBSD [Fre92] に於いて新たなキューイング機構を提供する枠組である altq [Cho97] を用いた．通常，ルータやゲートウェイ等のネットワーク中間ノードに置かれる機材は，専用のハードウェア及び専用のオペレーティングシステムを用いているものが多い．しかし専用のハードウェアを用いると，実装が困難であるだけでなくデバッグのコストも増大する．一方，近年のパーソナルコンピュータ (Personal Computer，PC) の性能の向上及びネットワークカードの高速化に伴い，PC をルータやゲートウェイとして用いることが現実的に可能となってきた．そこで本研究では，PC に複数のネットワークインタフェースカードを挿し，ルータとして使用することを想定して実装を行なった．

入力キューはトラフィックコントロールに対しては全く影響力を持たない為，altq は出力キューに於いて実現される．BSD UNIX に於いて出力キューのインタフェースを抽象化した構造体である `ifnet` に幾つかメンバを追加し，キュー及びその制御を行なう `IF_ENQUEUE()`，`IF_DEQUEUE()` を置き換えている．BSD UNIX では，出力キューは構造体 `if_snd` で管理されており，キューの

制御は`IF_ENQUEUE()`及び`IF_DEQUEUE()`というマクロにより行なわれる(図2.4参照)．これらのマクロは，各デバイスに応じたenqueueを行なう関数`if_output()`，及び各デバイスに応じたdequeueを行なう関数`if_start()`にて用いられる[McK96]．なお，altqに追加した本実装のソースコードを付録Cに添付する．

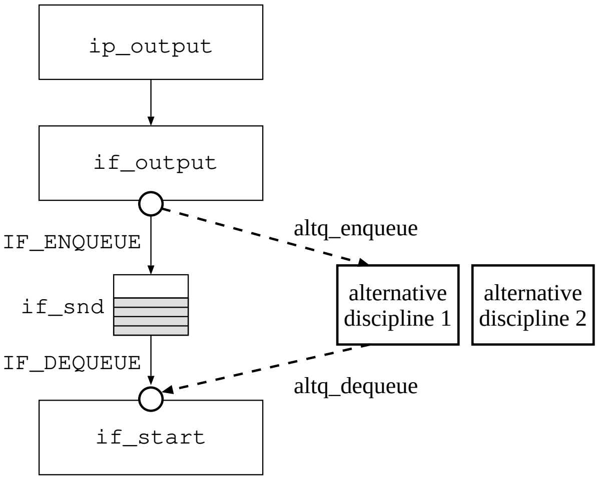


図2.4: Altenative Queueingの構造

ノードに於ける公平性に関しては3章で，共有メディアに於ける公平性に関しては4章でそれぞれ設計，実装，評価を説明する．両者ともaltqを用いている．最後に，本研究で用いるアルゴリズムを表2.1にまとめる．

表2.1: 本研究で用いるアルゴリズム

項目	アルゴリズム
スケジューリング	Weighted Fair Queueing (WFQ)
(enqueue)	Stochastic Fairness Queueing
(dequeue)	Deficit Round Robin
パケットドロップ	Random Early Detection (RED)
トラフィックシェーピング	トークンバケット

第3章

ノードに於ける制御

3.1 設計

Weighted Fair Queueing (WFQ) をノードに於ける制御の基本的な枠組として用いた．これは，WFQ が最もよく研究されているスケジューリングアルゴリズムの1つであり，その有用性が十分に示されているだけでなく，実装が比較的容易であることも理由として挙げられる．加えて，過去に行なわれた研究資産を活用することが可能であり，本研究では enqueue のアルゴリズムとして Stochastic Fairness Queueing を，dequeue のアルゴリズムとして Deficit Round Robin を用いる．また，パケットドロップに関しては，Active Queue Management [Bra97] を行なう RED を用いた．

3.1.1 Enqueue

キューの数

WFQ のキュー管理に関するコストの増大を避ける為，フローを固定個のキューにハッシュする Stochastic Fairness Queueing を用いる．1988 年に MIT にある ARPANET のゲートウェイ上に同時に存在したフローの数は最大でも 180 であった [Fel89]．つまり，例え大規模ネットワークだとしても，180 程度のキューを用意してハッシュ関数を慎重に選べば，高速な Fair Queueing として用いることができる．

キューの数はスーパーユーザが指定可能な可変値とし，指定が無い場合は180 を上回る最小の2の冪乗である 256 とする．なお最小値は1とし，この場合は単なる First In First Out (FIFO) 方式の出力キューとして機能する．逆に，最大値は標準値の4倍である 2,048 とした．

キューの重み付け

各キューの重み付けは初期状態では全て等しいが，スーパーユーザにより重み付けの変更を行うことができる．

ハッシュ関数

Stochastic Fairness Queueing を用いるため，パケットを enqueue するキューを決定するにはハッシュ関数が必要となる．そこで，以下に挙げる 3 つのハッシュ関数を用意する．デフォルトでは受信者アドレスによるハッシュを行なうが，ハッシュポリシはスーパーユーザにより変更可能とした．

1. 受信者アドレスによるハッシュ
   大規模ネットワークのバックボーン等では，ネットワークやリンク毎でキューを分けることにより，各ネットワークやリンクの混雑度に応じたサービスを提供できる．また，これに加えて WFQ に於いて接続リンクの帯域に応じた重み付けを行なうことにより，ネットワークやリンクの処理速度に応じたサービスを提供できる．

2. 送信者ポート番号によるハッシュ
   送信者側で複数のアプリケーションが実行されている場合などに，アプリケーション毎に異なるキューを持たせることにより，アプリケーション間での公平性を実現できる．また well-known ポートに関しては，例えば telnet のような対話的アプリケーションの重み付けを上げるといったようなこともできる．

3. 送信者/受信者 × アドレス/ポート番号によるハッシュ
   送信者及び受信者のアドレス及びポート番号用いることにより，基本的には各フロー毎に異なるキューを持つことができる．

3.1.2 Dequeue

dequeue するキューの選択は Deficit Round Robin により行なう．スケジューリングが行なわれたキューに与える quota (転送許可容量) のサイズは 512 バイトとする．これは標準的な実装が行なわれた ftp の転送パケットサイズである．

当然，この quota は重み付けに応じて変更される．例えば，重み付けが通常の 2 倍であるキューに対しては 1,024 バイトの quota が与えられ，重み付けが半分のキューに対しては 256 バイトの quota が与えられる．

3.1.3 パケットドロップ

本研究では，ドロップの決定に各フローが発生して以来の履歴情報を用いると共に，Head-Drop 及び Random Early Detection (RED) を採用した．

履歴情報の利用

バッファ全体からドロップするパケットを選び出す際，バッファ使用率の最も高いキューからパケットをドロップすると，前章で示したように，一時的にバーストが発生したフローを不利に

扱う可能性がある．そこで本研究では各キューの履歴情報を基に，そのキューが使われて以来どれだけのデータが dequeue されたかという，長期的なスループットを用いる．

Head-Drop

Tail-Drop は最も標準的な手法であり，実装も非常に容易である．スケジューラはキューの先頭や末尾のポインタに触れることなく，単に enqueue しようとしたパケットをドロップするだけで良い．しかし，いずれにせよパケットをドロップしなければならないような状況では，送信された時間が早いパケットをドロップした方が，ドロップしたという情報が早く送信者に伝わることになる [Lak96]．特に，TCP のような Fast Retransmit [Jac88] を用いている送信者に対して，Head-Drop は非常に効果的な手法となる．

そこで本研究では，パケットをドロップするキューが決定した場合，そのキューの先頭からパケットをドロップする．

Random Early Detection (RED)

RED はバイト単位でもパケット単位でも実現可能だが，本研究では dequeue アルゴリズムとして Deficit Round Robin を用いていることから，バイト単位で平均キューサイズなどの計算を行なう．

1. 平均キューサイズの計算

   RED では，パケットが到着する毎に平均キューサイズを計算する．そこで，ある重み付け w_q を用いて，以下のように平均キューサイズ avg を求める．ここで，q は現在のキューサイズである．なお，w_q としてどのような値を選択するかに関しては後述する．

   $$\begin{aligned} avg &= (1-w_q)\,avg + w_q \cdot q \\ &= avg + w_q\,(q - avg) \qquad (3.1) \end{aligned}$$

   平均キューサイズの計算は定期的では無くパケット到着毎に行なう．暫くの間キューが空だったとしても，パケットが到着しない限り平均キューサイズの計算は行なわれない．よって，キューが空のときに式 (3.1) に当てはめて計算を行なっても，平均キューサイズはなかなか減少しない．

   $$\begin{aligned} avg &= (1-w_q)\,avg + w_q \cdot 0 \\ &= (1-w_q)\,avg \end{aligned}$$

   そこでこのような場合，まずこの期間に送り出すことができたであろうパケットの数 m を求める．ここで q_time はキューが空の状態が開始した時刻，逆に $time$ は終了した時刻，s はパケット転送に関する平均処理時間とする．

$$m = (time - q\_time)/s$$

キューが空の間に m 個のパケットを送信できたとすると，式 (3.1) で示した平均キューサイズの計算が m 回行なわれることになる．そこで，キューが空のときにパケットが到着した場合の平均キューサイズは以下のようにして求める．

$$avg = (1 - w_q)^m \cdot avg \qquad (3.2)$$

2. ドロップの判断

RED は平均キューサイズに基づきパケットドロップの判断を行なう．この際，2 つの閾値を用意し，最初の閾値 (min_{th}, minimum threshold) よりも平均キューサイズが小さな場合は何もしないが，min_{th} と 2 つ目の閾値 (max_{th}, maximum threshold) の間に平均キューサイズがある場合はある確率 (probability) に従ってパケットをドロップし，max_{th} を越えた場合は全てのパケットをドロップする (図 3.1 参照)．

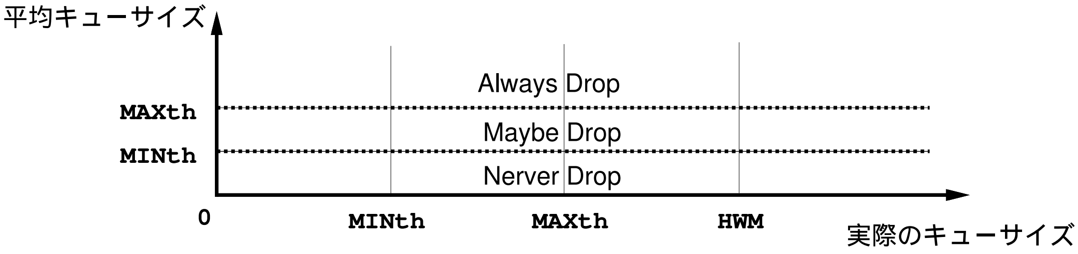


図 3.1: キューサイズとパケットドロップ (通常の RED)

max_{th}，min_{th} の値は，どの程度のバーストトラフィックを許容するか，またリンク使用率をどの程度と想定するかによって大きく異なってくる．本研究ではバッファサイズを 64K バイトとし，max_{th}，min_{th} はそれぞれ 48K バイト，16K バイトとした．

ところで，ftp のようなバルクデータ転送の場合は，コネクション設立直後に TCP の Slow-Start によって大量にデータが送信が開始される．当然，実際のキューサイズの増加を追うような形で平均キューサイズは増加する．RED ではパケットドロップの判断に平均キューサイズを用いているため，実際のキューサイズが大きいにも関わらずパケットをドロップしないような状況が起こり，素早い反応を行なうことができない．

また，送信元に於いて転送が終了した場合，実際のキューサイズは急激に減少し始める．しかしこの場合も，平均キューサイズはすぐには減少しない．よって，バッファが空に近い状態にも関わらずパケットがドロップされる状況が考えられる．

本研究では平均キューサイズだけでなく，実際のキューサイズを用いてパケットドロップの判断を行なうことにより，このような問題を解決する [Riz97] (図 3.2 参照)．

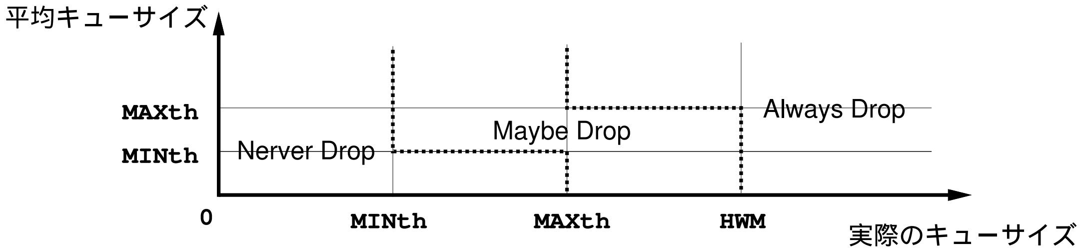


図 3.2: キューサイズとパケットドロップ (本研究)

3. パケットドロップ確率の計算

RED がある確率に基づいてドロップすると判断したとき (図 3.2 における “Maybe Drop” の範囲)，その確率を計算しなければならない．確率の最大値を max_p，2 つの閾値をそれぞれ min_{th}，max_{th} とすると，パケットドロップ確率 p_b は以下のように求められる．なお，max_p の値としては 1/50 程度の値を選択する．

$$\begin{aligned} p_b &= \frac{max_p\,(avg - min_{th})}{max_{th} - min_{th}} \\ &= \frac{max_p}{max_{th} - min_{th}} \cdot avg - \frac{max_p \cdot min_{th}}{max_{th} - min_{th}} \end{aligned} \quad (3.3)$$

max_p，min_{th}，max_{th} は定数のため，パケットドロップ確率は平均キューサイズ avg の一次関数となる．パケットが連続してドロップされたりすることなくドロップの間隔が分散するよう，前回パケットをドロップしてから今迄に送ったパケットの総数 $count$ を用いて，以下の式が成り立つときにはパケットをドロップする．ただし，R は 0 以上 1 以下の乱数である．

$$R < \frac{p_b}{1 - count \cdot p_b}$$

ここで，$1 - count \cdot p_b$ が負になるのは，$count$ が十分に大きいときである．よって，この場合は R の値に依らずパケットをドロップする．$1 - count \cdot p_b$ が正だと仮定すると，上式は以下のように変更することができる．

$$R \cdot (1 - count \cdot p_b) < p_b \quad (3.4)$$

4. 平均キューサイズ計算の重み付け

w_q の値を小さくし過ぎると，実際のキューサイズの細かい振動の影響を平均キューサイズが受けることになる．逆に w_q の値が大き過ぎると，実際のキューサイズの状態に素早く反応することができなくなる．キューサイズ及び，平均のキューサイズが 0 を初期状態とし，L 番目のパケットが到着した際の平均キューサイズ avg_L を求めると以下のようになる．

$$\begin{aligned} avg_L &= \sum_{i=1}^{L} i \cdot w_q\,(1-w_q)^{L-i} \\ &= w_q\,(1-w_q)^L \sum_{i=1}^{L} i\,(\frac{1}{1-w_q})^i \\ &= L+1+\frac{(1-w_q)^{L+1}-1}{w_q} \end{aligned} \tag{3.5}$$

ただし，ここでは以下の式を用いている [Han75]．

$$\sum_{i=1}^{L} i \cdot x^i \quad = \quad \frac{x+(L\cdot x-L-1)\,x^{L+1}}{(1-x)^2}$$

式 (3.5) の関係を図 3.3 に示す．例えば，$w_q = 0.001$ のときキューサイズ L が 0 から 100 に増加すると，平均キューサイズ $avg_{100} = 4.88$ となる．

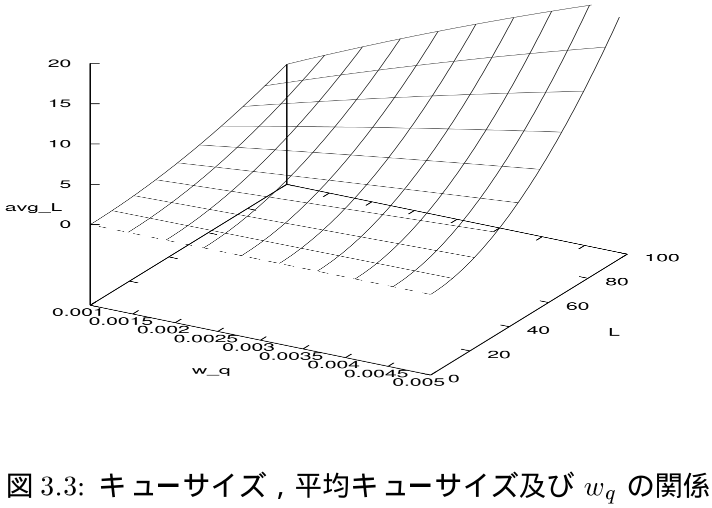


図 3.3: キューサイズ，平均キューサイズ及び w_q の関係

定常状態で平均キューサイズ avg_L が min_{th} よりも小さくなければ，不用意にパケットがドロップされることになる．よって，以下の式が成り立たねばならない．

$$L+1+\frac{(1-w_q)^{L+1}-1}{w_q} \quad < \quad min_{th}$$

例えば，min_{th} を 5 パケット，L を 50 パケットとすると，$w_q \leq 0.0042$ でなければならない．

3.2 実装

Stochastic Fairness Queueing によりフローとキューとの対応付けに関するオーバーヘッドが減ったとしても，各キュー毎の状態管理は避けることができない．WFQ では，どのような構造で，どのような情報を管理するかが問題となる．本実装では，WFQ の状態を保持する構造体 `wfq_state` 及び各キューの状態を保持する構造体 `weighted_fair_queue` を用意した．

```
typedef struct wfq_state{
    struct wfq_state *next;             /* for wfqstate list */
    struct ifnet *ifp;
    int nqueues;                        /* number of queues */
    int bytes;                          /* total bytes in all the queues */
    wfq *rrp;                           /* round robin pointer */
    wfq *queue;                         /* pointer to queue list */
    int hash_policy;                    /* policy for hash function */
    int (*hash_fnc)(struct flowinfo *, int);
    int hwm;                            /* high water mark */
#ifdef TOKEN_BUCKET
    int token;                          /* token for dequeue */
    int bucket;                         /* token bucket size */
#endif /* TOKEN_BUCKET */
#ifdef RED
    int count;                          /* packet count since last dropped */
    int avg;                            /* average enqueued bytes */
#endif /* RED */
}wfq_state_t;
```

```
typedef struct weighted_fair_queue{
    struct weighted_fair_queue *next, *prev;
    struct mbuf *head, *tail;
    int quota;                          /* bytes sent in this round */
    int weight;                         /* weight in percent */
    int bytes;                          /* bytes in this queue */
    long idle;                          /* start time of idling */
    int sent_bytes;                     /* bytes sent in this queue */
    int sent_packets;                   /* packets sent in this queue */
    int drop_bytes;                     /* bytes dropped in this queue */
    int drop_packets;                   /* dropped packets */
}wfq;
```

`wfq_state` は WFQ に対して 1 つ必要となり，`weighted_fair_queue` はキューの数と同数だけ必要となる．共にインタフェースを attach する際にメモリ確保が行なわれる．この構造体間の関係の概略を図 3.4 に示す．

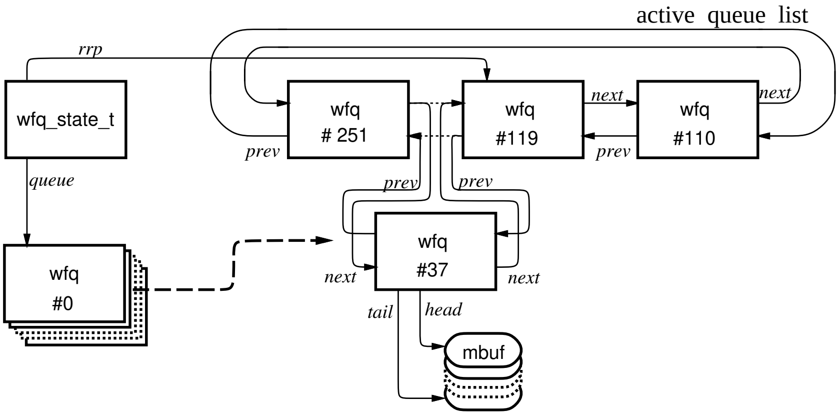


図 3.4: WFQ の構造

パケットの送出は，active queue list という linked list に繋がれているキューを `rrp` (Round Robin Pointer) で辿ることによって行なう．パケットが到着すると，ハッシュ関数によって enqueue されるキューが決定される．そのキューが active queue list に繋がっていない場合は `prev`，`next` 等のポインタの付け替えを行なう (図中 #37 のキュー)．逆に，送出すべきパケットの無くなったキューは list から取り除かれる．以下の項では，この流れをより詳しく追っていく．

3.2.1 Enqueue

受け取ったパケットはハッシュ関数により enqueue されるキューが決定する．このハッシュ関数は，`wfq_state` 構造体内の関数へのポインタ `hash_func` により変更することができる．本実装では

以下の 3 つの関数を用意したが，任意の関数を容易に追加することができる．なお，`hash_policy` という値は，ハッシュ関数の変更を行なう場合等に参照する変数である．

1. 受信者アドレスによるハッシュ

   図 3.5 に示すように，受信者アドレス 32 ビットを 8 ビットづつ右シフトしながら，各ビットの排他的論理和を取る．こうして得られる Hashed Value をキューの数で割った剰余が enqueue するキューの番号となる．

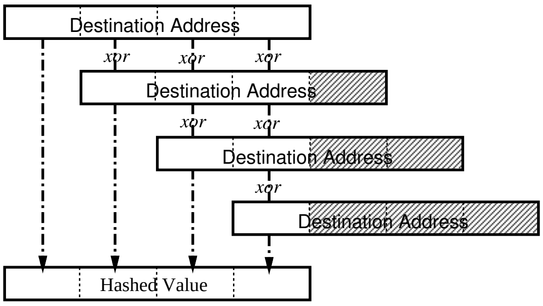


図 3.5: 受信者アドレスによるハッシュ

```
val = fp->fi_dst.s_addr;
val = val ^ (val >> 8) ^ (val >> 16) ^ (val >> 24);
val = val % n;
```

2. 送信者ポート番号によるハッシュ

   図 3.6 に示すように，送信者ポート番号 16 ビットを，8 ビット右シフトした値との間で各ビットの排他的論理和を取る．この場合も，前述のように剰余を用いる．

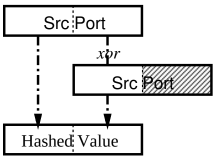


図 3.6: 送信者ポート番号によるハッシュ

```
val = fp->fi_sport;
val = val ^ (val >> 8);
val = val % n;
```

3. 送信者/受信者 × アドレス/ポート番号によるハッシュ

   基本的には前述 2 つのアルゴリズムの組み合わせである．まず，送信者と受信者のアドレス，ポート番号に関してそれぞれ排他的論理和を求める．次に，アドレスに関しては受信者アドレスによるハッシュのビット演算，ポート番号に関しては送信者ポート番号によるハッシュのビット演算を行なう．最後に得られた 2 つの値の排他的論理和を求めてから剰余を得る．

```
val = fp->fi_dst.s_addr ^ fp->fi_src.s_addr;
val = val ^ (val >> 8) ^ (val >> 16) ^ (val >> 24);
val2 = fp->fi_dport ^ fp->fi_sport ^ fp->fi_proto;
val2 = val2 ^ (val2 >> 8);
val = val ^ val2;
val = val % n;
```

キューが決定された際，mbuf [McK96] に対する `head` 及び `tail` などのポインタの付け替えを行なうことによりパケットの enqueue を行なう．同時に，enqueue されているバイト数を示す `wfq_state`，`weighted_fair_queue` 両構造体のメンバ `bytes` の値を増加する．

パケットを enqueue した結果，バッファが `wfq_state` 構造体で指定される使用制限 (`hwm`) に達していた場合や，RED アルゴリズムによってパケットをドロップすると判断した場合は，`weighted_fair_queue` 構造体のメンバから得られる情報を基にどのキューからパケットをドロップするかを決定する．なお，パケットドロップの判断に関する詳細は次節で述べる．

そのキューが active queue list に繋がれていない場合には，`rrp` によって指されているキューの直前に挿入する．これは `next` 及び `prev` のポインタを幾つか付け替えることによって実現される．また，`rrp` が NULL だった場合，つまり active queue list に 1 つもキューが繋がれていなかった場合は， active queue list にキューを繋ぐと同時に，dequeue の為の関数を起動する．

3.2.2 Dequeue

パケットの dequeue は，active queue list を `wfq_state` の `rrp` で辿ることにより，Deficit Round Robin でスケジューリングを行なう (図 3.4 参照)．

通常の Deficit Round Robin では，パケットサイズが quota 以下かどうかを確認してから dequeue を行なうが，mbuf へのポインタ (`queue->head`) を辿り，パケットサイズの確認を行なうのはオーバーヘッドが大きい．そこで，quota に負の値を用い，quota が正であればパケットを dequeue するようにした．パケットを dequeue した後，そのパケットサイズ分の quota が溜るまでは次のパケットを dequeue できない．

```
struct mbuf *mp;
int byte;

if(queue->quota > 0){
  mp = queue->head;
  byte = mp->m_pkthdr.len;
  queue->quota -= byte;
  ⋮
}
```

スケジューリングされたキューでは `weighted_fair_queue` 構造体の各メンバに対して，`WFQ_QUOTA` を `quota` に加え，送信したバイト数を `sent_bytes` に加え，逆に `bytes` から減じ，最後に，送

信したパケット数を `sent_packet` に加える．また，送信したバイト数は `wfq_state` 構造体のメンバ `bytes` からも減じる．ただし，3.1.2 項で説明したように，quota に与える値はキューの重み付けに応じて変化する．

送出すべきパケットを持たないキューが active queue list から除かれる際，`microtime()` によって得られた値 (1970 年 1 月 1 日 0:00:00 からの秒数) を `weighted_fair_queue` 構造体のメンバ `idle` に代入する．この値は，再びこのキューが active queue list に繋がれる際に参照され，4 秒以上経過していた場合は `weighted_fair_queue` 構造体の以下の 4 つのメンバの値をゼロクリアし，過去の履歴を消去する．

- 送信バイト数 (`sent_bytes`)
- 送信パケット数 (`sent_packets`)
- ドロップバイト数 (`drop_bytes`)
- ドロップパケット数 (`drop_packets`)

キューが空となっていた期間を元に情報のリセットを行なうことにより，異なるフローが同じキューにハッシュされてしまった場合などにも対応可能となる．また，ある瞬間の混雑の影響が，そのキューに於いて永遠に履歴として残るといったようなことも避けることができる．

3.2.3　パケットドロップ

履歴情報の利用

`weighted_fair_queue` 構造体のメンバ `idle` と `sent_bytes` から，キューが active queue list に繋がれた時間と，送信バイト数が求まる．そこで，この 2 つの値を基にスループットの計算を行なう．ただし，現時点で全くパケットが enqueue されていないキューは，当然ドロップするパケットが存在しない為に候補から除かれる．

Head-Drop

図 3.4 に示したように，`wfq_state` 構造体のメンバ `head` がそのキューに enqueue されているパケットの先頭を指している．この為，メモリの解放と共に `head` 及び mbuf 構造体内のポインタを幾つか付け替えるだけで Head-Drop は比較的容易に実装可能である．

Random Early Detection (RED)

RED では多くの計算がパケット到着毎に行なわれる為，パラメータを注意深く選ばなければ計算量のオーバーヘッドが膨大となってしまう．

1. 平均キューサイズの計算
   式 (3.1) に於いて平均キューサイズの計算法を示したが，この式では減算 1 回，加算 1 回，乗算 1 回の演算が必要となる．平均キューサイズの計算はパケット到着毎に実行されるため，

計算量は少なければ少ないほど良い．ここで，w_q として 2 の冪乗の値を選択すると，乗算を 1 回のシフト演算として実行できる．本実装では，$0.001953125 = 1^{-9}$ を用いることにより，9 ビット右シフト演算に置き換えた．

```
#define RED_IWAIT       9               /* negative power of two */

wfqp->avg += (wfqp->bytes - wfqp->avg) >> RED_IWAIT;
```

一方，キューサイズが 0 のときに届いたパケットに関しては式 (3.2) を用いている．本実装では，10Mbps Ethernet に於いて 1Kbyte のパケットの転送処理時間が 0.8 ミリ秒となることから，パケット転送に関する平均処理時間 s を約 1 ミリ秒であると仮定した．

$$10Mbps = 10Mbyte/8s = 1Kbyte/0.8ms$$

また，q_time 及び $time$ は `microtime()` によって得られる値 (1970 年 1 月 1 日 0:00:00 からの秒数) を用いた．microtime() は構造体 `timeval` に値を返すため，秒からミリ秒及びマイクロ秒からミリ秒への変換が必要となる．ここでも 1,000 ではなく 1,024 で乗算，除算を行なうことにより，10 ビット左及び右シフトへの置き換えを行なった．もともと s の値を決めるに当たっての不確定要素が多いため，この近似は問題とはならない．

```
#ifndef _SYS_TIME_H_
struct timeval{
  long tv_sec;          /* seconds */
  long tv_usec;         /* and microseconds */
};
#endif /* _SYS_TIME_H_ */

struct timeval now;
static struct timeval last = {0, 0};    /* last packet sent time */

microtime(&now);
msec = ((now.tv_sec - last.tv_sec) << 10)
       + ((now.tv_usec - last.tv_usec) >> 10);
last = now;
```

$(1 - w_q)$ の m 乗は，m を 2 の冪乗の積へと分解し，$(1 - w_q)^{2^n}$ の積の形で計算を行なう．例えば $m = 100$ のとき，$(1 - w_q)^{100} = (1 - w_q)^{64} \times (1 - w_q)^{32} \times (1 - w_q)^{4}$ となる．m を 2 の冪乗へ展開する際に，2^n 乗の項が含まれるかどうかはビット積を求めるだけで得られる．

なお，カーネル空間に於いて浮動小数点演算を行なうと，ユーザ空間の Floating Point Unit (FPU) を破壊してしまう為，10 ビット左シフトした状態で演算を行なった．よって，定数 $(1 - w_q)^{2^n}$ は以下のようになる．

$$
\begin{array}{lcrcr}
(1-0.001953125)^{1} & \times & 1,024 & = & 1,022 \\
(1-0.001953125)^{2} & \times & 1,024 & = & 1,020 \\
(1-0.001953125)^{4} & \times & 1,024 & = & 1,016 \\
(1-0.001953125)^{8} & \times & 1,024 & = & 1,008 \\
(1-0.001953125)^{16} & \times & 1,024 & = & 992 \\
(1-0.001953125)^{32} & \times & 1,024 & = & 962 \\
(1-0.001953125)^{64} & \times & 1,024 & = & 904 \\
(1-0.001953125)^{128} & \times & 1,024 & = & 797 \\
(1-0.001953125)^{256} & \times & 1,024 & = & 621 \\
(1-0.001953125)^{512} & \times & 1,024 & = & 376 \\
(1-0.001953125)^{1,024} & \times & 1,024 & = & 138 \\
(1-0.001953125)^{2,048} & \times & 1,024 & = & 19 \\
(1-0.001953125)^{4,096} & \times & 1,024 & = & 0
\end{array}
$$

4,096 ミリ秒 $=$ 4 秒以上キューにパケットが全く無い状態が続くと，平均キューサイズは必ず 0 となる．

```
#define RED_AVG_SHIFT   10

int i, pow2 = 2048;
const int w[12] =
        {19, 138, 376, 621, 797, 904, 962, 992, 1008, 1016, 1020, 1022};

if((msec == 0)
  ;
else if(msec < 4096)
  for(i = 0; i < 12; i++){
    if(msec & pow2){
      if((wfqp->avg = (wfqp->avg * w[i]) >> RED_AVG_SHIFT) == 0)
        break;
      if((msec ^= pow2) == 0)
        break;
    }
    pow2 >>= 1;
  }
else
  wfqp->avg = 0;
```

[Flo93] に於いて提案された RED の性能は，シミュレータ ns [McC96] を用いて実証されおり，計算量に対する考慮はあまり行なわれていない．そこで，論文内で提案されている平均キューサイズ計算アルゴリズムと本実装で用いるアルゴリズムを ns 上で比較した．図 3.7 のようなネットワークトポロジに於いて，$min_{th} = 4$，$max_{th} = 12$，パケットサイズ $=$ 1,024 byte，$w_q = 0.001953125 = 1^{-9}$ という条件で，s1 から s3 に向けて ftp のフローを流し，その 3 秒後に s2 から s4 に向けて別の ftp のフローが流れ始めるというシミュレーショ

ンを行なった．なお，図中の “10Mb,2ms” などの表示は “帯域，Round Trip Time (RTT)” を表している．

```
            s1                 s3
             \                 /
   10Mb,2ms \  1.5Mb,20ms   / 10Mb,4ms
               r1 --------- r2
   10Mb,3ms /               \ 10Mb,5ms
             /                 \
            s2                 s4
```

図 3.7: ns に於けるネットワークトポロジ

このシミュレーション結果を図 3.8 に示す．なお太い実線は，本研究で用いたアルゴリズムをシミュレーション上で用いた場合の平均キューサイズの値であり，太い破線は論文内で提案されているアルゴリズムを忠実に実現した場合の平均キューサイズの値である．

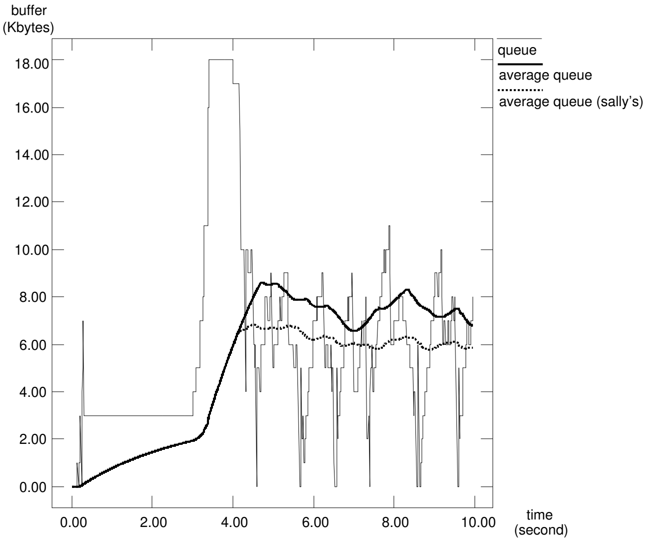


図 3.8: 平均及び実際のキューサイズ (シミュレーション)

計算量を抑える為の省略を行なった影響から，多少ではあるが，平均キューサイズが実際のキューサイズに連られていることが分かる．

2. ドロップ確率の計算

式 (3.3) で示した通り，ドロップ確率の計算は乗算 1 回，減算 1 回から成り立つ．ここでも，avg の係数を 2 の冪乗となるように定数 max_p，min_{th}，max_{th} を選ぶことにより，乗算を 1 回のシフト演算として実行できる．本実装では，$max_{th} = 48 \cdot 2^{10}$，$min_{th} = 16 \cdot 2^{10} = 2^{14}$，$max_p = 0.3125 = 2^{-5}$ とした．

$$
\begin{aligned}
\frac{max_p}{max_{th} - min_{th}} &= \frac{2^{-5}}{32 \cdot 2^{10}} = \frac{2^{-5}}{2^{15}} = 2^{-20} \\
\frac{max_p \cdot min_{th}}{max_{th} - min_{th}} &= \frac{2^{-5} \cdot 2^{14}}{32 \cdot 2^{10}} = \frac{2^{9}}{2^{15}} = 2^{-6}
\end{aligned}
$$

ここでも，浮動小数点演算を避けるため，15 ビット左シフトした状態で演算を行なった．

```
#define RED_PB_SHIFT    15
#define RED_IMAXp       5               /* negative power of two */
#define RED_SMAXp       (1 << (RED_PB_SHIFT - RED_IMAXp))
#define RED_IC1         20              /* negative power of two */
#define RED_SIC1        (RED_IC1 - RED_PB_SHIFT)
#define RED_IC2         6               /* negative power of two */
#define RED_SC2         (1 << (RED_PB_SHIFT - RED_IC2))

if((spb = (wfqp->avg >> RED_SIC1) - RED_SC2) > RED_SMAXp)
  spb = RED_SMAXp;
```

式 (3.4) に加えて，$1 - count \cdot p_b$ が負の場合もパケットをドロップするため，条件式は以下のようになる．

```
#define RED_MAXr        1
#define RED_SMAXr       (RED_MAXr << RED_PB_SHIFT)

if((tmp = RED_SMAXr - wfqp->count * spb) < 0 ||
   (((random() % RED_SMAXr) * tmp) >> RED_PB_SHIFT) < spb)
  drop a packet;
```

3. フロー毎の制御

3.1.3 項で示した通り，本実装では平均キューサイズなどの値に依らず，到着したパケットは必ず enqueue される．enqueue した後に必要に応じてパケットをドロップする．

```
wfq_ifenqueue()
{
  enqueue the packet;

  if(random_drop(wfqp, queue))
    drop_queue = wfq_maxthqueue(wfqp);

  while(drop_queue != NULL){
    drop a packet;
    drop_queue = (wfqp->hwm < wfqp->bytes) ? wfq_maxthqueue() : NULL;
  }
}
```

パケットをドロップするかどうかの判断は，`random_drop()`に於いて以下のように行なわれる．

```
random_drop()
{
  calculate the average queue bytes;

  if(wfqp->hwm < wfqp->bytes ||
     RED_MAXth < wfqp->avg && RED_MAXth < wfqp->bytes)
    return 1;
  else if(RED_MAXth < wfqp->bytes ||
          RED_MINth < wfqp->bytes && RED_MINth < wfqp->avg)
    return drop by probability;
  else
    return 0;
}
```

まず，パケットドロップの判断に必要となる平均キューサイズ`wfqp->avg`の計算が行なわれる．ここで，図3.2を参照して欲しい．“Always Drop”のとき及び，“Maybe Drop”のときにREDのアルゴリズムに従ってドロップすると判断された場合，長期的に最も帯域幅を使っているキューからパケットをドロップする．“Never Drop”のときはパケットをドロップしない．

3.3　評価

本節では，ノードに於ける制御に関して行なった実験の評価結果を示す．まずオーバーヘッドの測定結果を示し，その後WFQによって実現した各種フロー間での公平性に関する評価結果，REDによって実現したバッファの空き容量の提供に関する評価結果を示す．

なお，評価に用いたネットワークカードに関しては表A.2を，1，2，3，4，5，R1，R2等の端末を表す番号に関しては表A.3を参照して欲しい．

3.3.1　オーバーヘッド

WFQ及びREDのオーバーヘッド測定は，5種類のカーネルを用いて行なった．GENERICは，FreeBSDで特殊な設定を全く行なっていないカーネルである．altqにwfqのコード(付録C参照)を加え，`#define RED`したものがaltq (wfq + red)，`#undef RED`したものがaltq (wfq)である．また，本実装ではREDはWFQと一体となっている為，単純なREDのみのオーバーヘッドは，altqでGENERICカーネル同様のFIFOキューイング機構を提供した上で測定した．それぞれがaltq (fifo)，altq (fifo + red)に対応する．

図 3.9 のような環境に於いて，`netperf` [Jon93] を用いて TCP 及び UDP のスループットの測定を行なった．TCP，UDP のフローはそれぞれ単独に発生し，測定の際，他のフローは一切存在しないとする．なお，`netperf` のパラメータに関しては付録 A.a を参照して欲しい．

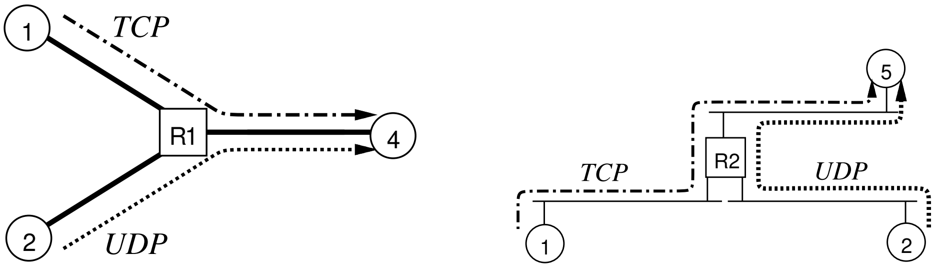


図 3.9: 実験環境 (左:155Mbps ATM，右:10Mbps Ethernet)

表 3.1 に示すように，ネットワークインタフェースの帯域幅が 155Mbps 程度の場合には，全く問題とならないことが分かる．

表 3.1: オーバーヘッド (WFQ，RED)

カーネル	スループット (Mbps)			
	155Mbps ATM		10Mbps Ethernet	
	UDP	TCP	UDP	TCP
GENERIC	134.63	106.70	9.43	6.51
altq (fifo)	134.63	106.68	9.43	6.49
altq (fifo + red)	134.63	106.35	9.43	6.48
altq (wfq)	134.63	106.49	9.43	6.48
altq (wfq + red)	134.63	106.01	9.43	6.47

3.3.2 キューの重み付け

図 3.10 のようなネットワークで，R1，R2 の WFQ に於いてキューの重み付けを変更して出力リンクの共有を行なった．他のフローは存在しない為，Deficit Round Robin によりスケジューリングが行なわれるのはこの 2 つのキューのみである．

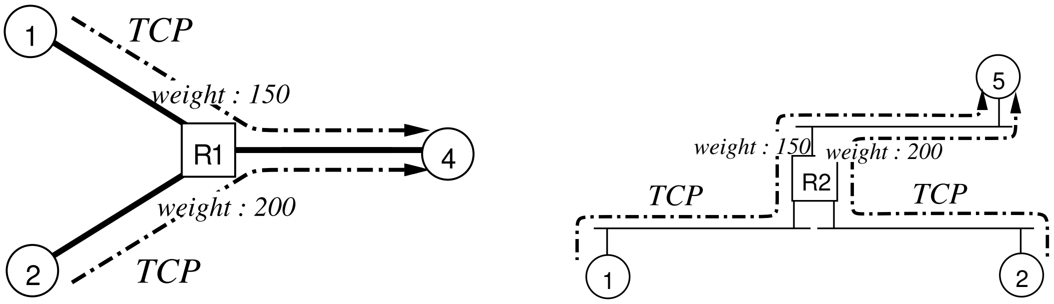


図 3.10: WFQ 重み付け実験環境 (左:155Mbps ATM，右:10Mbps Ethernet)

表 3.2 に示す通り，重み付けに応じた帯域の共有が行なわれていることが分かる．

表 3.2: WFQ 重み付け実験 (左:155Mbps ATM，右:10Mbps Ethernet)

重み付け	フロー	スループット (Mbps)
150	TCP (1 → 4)	48.32
200	TCP (2 → 4)	58.90

重み付け	フロー	スループット (Mbps)
150	TCP (1 → 5)	3.00
200	TCP (2 → 5)	4.05

3.3.3　フローの保護

図 3.9 に於いて，TCP と UDP のフローを同時に流すことにより，適応型のフローと非適応型のフローとでルータ R1，R2 の出力リンクの共有を行なった．

測定は，GENERIC カーネルと WFQ 及び RED を用いたカーネルとの比較に加えて，Tail-Drop を行なう WFQ との比較により行なった．なお，RED に於ける非適応型のフローを特定する機構を，本実装では WFQ により提供している為，RED 単独での評価は行なっていない．

155Mbps ATM

表 3.3 に示す測定結果に於いて，各カーネルの比較から以下のことが分かる．

- GENERIC ↔ altq (wfq) – Tail-Drop –/altq (wfq)/altq (wfq + red)

  WFQ を用いることにより，TCP のような適応型のフローと UDP のような非適応型のフローとの間で公平性を実現できる

- altq (wfq) – Tail-Drop – ↔ altq (wfq)/altq (wfq + red)

  履歴情報を用いたパケットドロップを用いることにより，TCP の起こす瞬間的なバーストトラフィックに適応できる

- altq (wfq) ↔ altq (wfq + red)

  RED を用いることにより，中間ノードでの混雑を早い段階で送信者に伝えることができる為，適応型のフローはスループットが向上する

表 3.3: TCP と UDP の競合実験 (155Mbps ATM)

カーネル	フロー	スループット (Mbps)
GENERIC	TCP (1 → 4)	1.25
	UDP (2 → 4)	133.59
altq (wfq) – Tail-Drop –	TCP (1 → 4)	12.17
	UDP (2 → 4)	118.96
altq (wfq)	TCP (1 → 4)	47.54
	UDP (2 → 4)	84.85
altq (wfq + red)	TCP (1 → 4)	51.41
	UDP (2 → 4)	85.41

10Mbps Ethernet

表 3.4 に示す測定結果を表 3.3 と比較すると，違いは明確である．例え WFQ を用いてフロー毎のキューを用意したとしても，共有メディアである Ethernet 上では帯域の公平な共有ができない．これは，2.2 節で示した問題が発生する為であり，この解決に関しては 4.4 節でふれる．

表 3.4: TCP と UDP の競合実験 (10Mbps Ethernet)

カーネル	フロー	スループット (Mbps)
GENERIC	TCP (1 → 5)	0.03
	UDP (2 → 5)	9.39
altq (wfq) – Tail-Drop –	TCP (1 → 5)	0.04
	UDP (2 → 5)	9.39
altq (wfq)	TCP (1 → 5)	0.04
	UDP (2 → 5)	9.39
altq (wfq + red)	TCP (1 → 5)	0.04
	UDP (2 → 5)	9.39

3.3.4　キューサイズ

ここまでは WFQ によるフローの保護に関する評価結果を示してきたが，本項では RED を用いることによる，バッファの空き容量に関する評価結果を示す．

まず，3.2.3 項で示したシミュレーションのネットワークトポロジ (図 3.7 参照) を意識し，図 3.11 のようなネットワークトポロジを用意した．実験のシナリオも同様とし，送信者 1 から受信者 2 に向けて ftp1 のフローが流れ始めた 3 秒後に，送信者 5 から受信者 3 に向けて ftp2 のフローが流れ始めるという設定である．

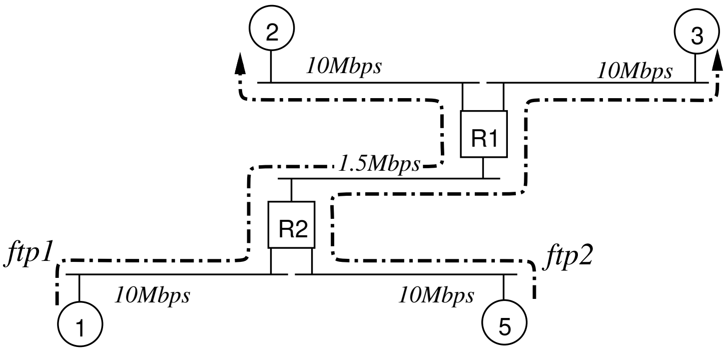


図 3.11: ネットワークトポロジ

R2 に於いて，GENERIC カーネルを用いて通常の FIFO 方式によるサービスを行なった際の出力キューの様子を図 3.12 に示す．

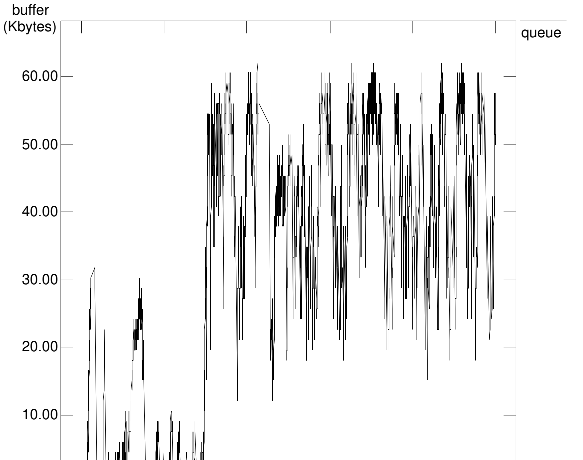


図 3.12: RED を用いなかった場合のキューサイズ

0~3秒の間で，パケットがドロップするまでウィンドウサイズを広げるTCPのSlow-Startの様子が伺える．3秒後にフローftp2が発生したことにより，急激にキューサイズが上昇し，その後はキューサイズが高い位置に留まり続けている．この結果，[Bra97]に於いて指摘された通り，バッファにほとんど空きが無い状態となっている．これに対し，ルータR2に於いて本実装のカーネルによりサービスを行なった際の出力キューの様子を図3.13に示す．なお太い実線は，平均キューサイズの値である．

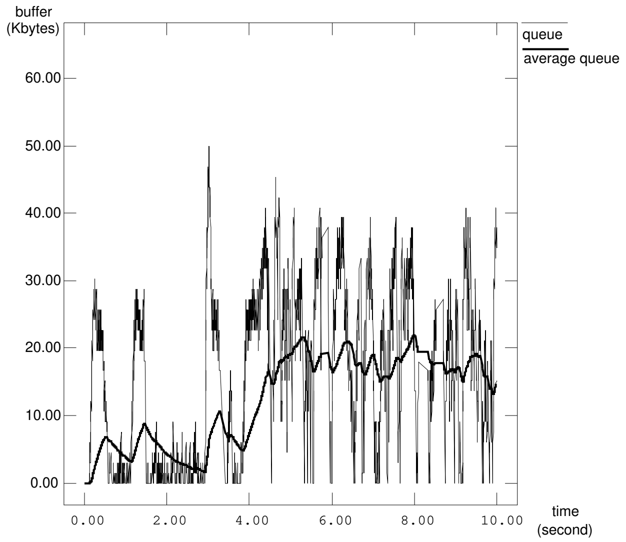


図3.13: 平均及び実際のキューサイズ(本実装)

バッファが溢れる前に，確率的にパケットをドロップすることにより，フローftp2が発生した後も，常にバッファに空き領域としての余裕を持っていることが分かる．

第4章

共有メディアに於ける制御

4.1　設計

本研究では，トークンバケットアルゴリズムを用いることにより，インタフェースからの出力をシェーピングする．トークンバケットに於いて，周期毎に生成されるトークンの量は動的に変更される．これにより，コネクション，トラフィック及びネットワークの状態に応じたトラフィックシェーピングを実現する．

4.1.1　トークンの生成

周期

トークンの生成周期は kernel clock (10 μ 秒) 毎とし，トークン 1 つに付き 1 つのパケットを送出可能とする．

初期化

生成するトークンの量は，通信開始時に初期化する．図 4.1 に初期化の流れを示す．なお，図の左から右に向かって時間は進んでいる．

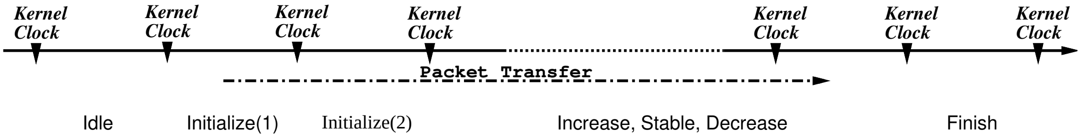


図 4.1: トークン生成量の初期化

当然ではあるが，通信の開始が kernel clock と同期するとは限らない．そこで，実際の通信が開始された `Initialize(1)` に於いて，10 μ 秒に満たない半端な時間を利用しない．その次の kernel clock に当たる `Initialize(2)` に於いて，シェーピングされていない状態でどれだけのパケットを送信したかを元に初期値を設定する．

状態遷移

初期化後のトークンの生成量は，前回与えたトークンを使い切っていた場合には増加し，少量だけ残っていた場合には変化せず，衝突が発生した等の理由で大量に残った場合には減少する．また，与えたトークンが全く使われなかった場合には，全ての状態をクリアする．この状態遷移を図 4.2 にまとめる．

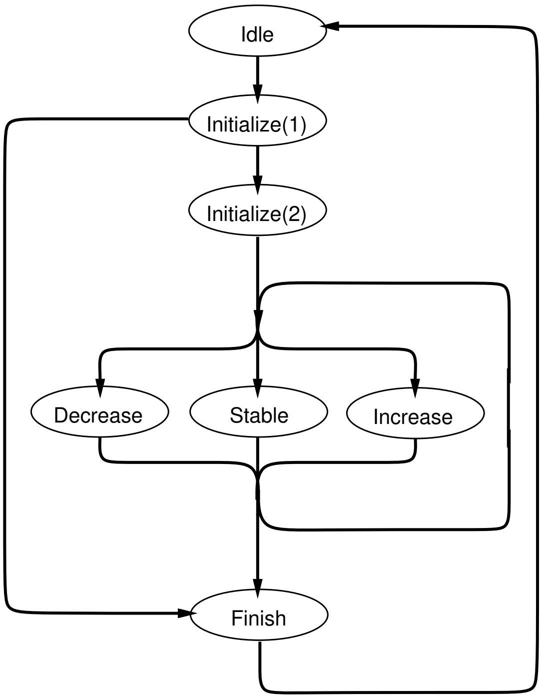


図 4.2: トークンの増減に関する状態遷移図

`Idle` のときはパケットが完全に満たされている．通信が開始されてトークンが使用されると `Initialize(1)` へ移る．前述のように，次の kernel clock で周期的に生成するトークンの量が決定されるが，このときパケットの転送が全く行なわれないと `Finish` を経て `Idle` に戻る．通信が引き続き行なわれた場合には，`Initialize(2)` に於いて初期化が行なわれる．その後は使用したトークンの量に応じて `Increase`，`Stable`，`Decrease` を繰り返す．与えたトークンが全く使われなかった場合には `Finish` に於いて変数のクリアを行なって `Idle` に戻る．

再計算の粒度

生成されるトークンの量は，パケットの中に残っているトークンの量を基に増減する．しかし，多くの通信は kernel clock よりも大きな粒度でパケットの増減が発生する．つまり，キューにパケットがあまり溜っていない状態とは，今後もキューにはその程度しかパケットが溜らない状態が続く可能性と同時に，近い将来に大量のパケットが到着する可能性を含んでいる．

与えるトークンの量を kernel clock 毎に再計算すると，図 4.3 に示すように，不安定な増減を示す．なお，以降本項で示す図は，あるルータを経由する複数の TCP 及び UDP のフローが，10Mbps Ethernet の出力リンクの帯域を使い切るような通信を行なおうとした際のトークンの増減を示している．横軸は時間，縦軸はトークンの量である．また，破線は与えたトークンの量，実線はバケットの中に残ったトークンの量である．

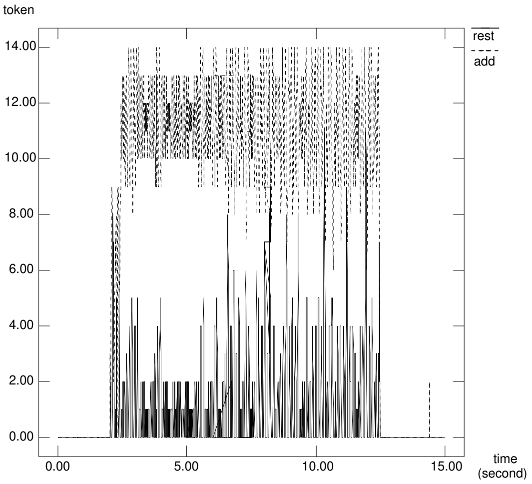


図 4.3: トークンの増減 (kernel clock 毎再計算)

本研究では，トークン量の再計算の粒度を上げ，16 回の kernel clock (160 μ 秒) 毎とする．もともと，トークンバケットの目的は，シェーピングによりトラフィック量を一定に保つと共に，ある程度のバーストトラフィックを許容することにある．再計算の粒度を大きくすると，使われなかったトークンはバケットに溜る為，このようなトークンバケットの機能を充分に発揮することができるようになる．

増減に対する **bias**

再計算の粒度を上げることにより，トークン量は図 4.4 に示すようにある程度安定した状態となる．しかし，与えたトークンが足りなければ増加し余るようなら減少するというように，その時点でのトークンの残量のみに従って生成するトークン量の増減を行なうと，図に示した通り振動することになる．

ここで例として，トークンの残量が 0 となっている場合を考えてみる．トークンが足りないと判断して与えるトークンの量を増加すると，実際は現在のトークンの量で丁度均衡していた為に大量にトークンが余るかも知れない．トークンが大量に余ると，当然，次の段階ではトークンの量が減少する．また丁度均衡しているときに，偶然トークンが残ってしまった場合などは，トークンの量が減少してしまうと，確実に足りなくなる．それ以前に，トークンバケットによるシェーピングの目的は，確認応答を受ける為に帯域に余裕を残すことである．つまり，トークンを使い切るよりも，少し余裕のある状態でトークンの量が安定することが望ましい．

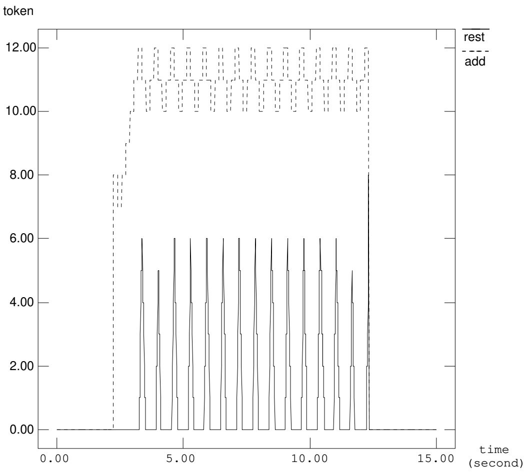


図 4.4: トークンの増減 (bias 無し)

本研究では，帯域を使い切らずに多少の余裕を残すよう，トークンの増加に関して bias を掛ける．ただし，トークンの量を増加すること自体に bias を掛けるとトークン生成量の初期化に失敗するだけでなく，トラフィックの変化に素早く対応することが困難となってしまう．そこで，1 度トークン量が減少した後に増加しようとした場合にのみ指数的な bias を掛けた (図 4.5 参照)．

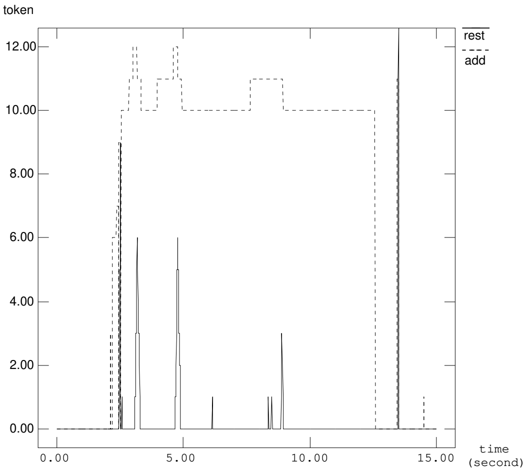


図 4.5: トークンの増減 (本研究)

4.1.2　パケットサイズ

例えば，10Mbps Ethernet に於いて 1K バイトのパケットを送信して帯域を使い切るには kernel clock 毎に 13 個のトークンが必要となる．同様に，155Mbps ATM に於いて 8K バイトのパケットを送信して帯域を使い切るには 25 個のトークンが必要となる．

$$\begin{aligned} 10Mbit/s &= 12.5Kbyte/10ms &= 12.5 \cdot 1Kbyte/10ms \\ 155Mbit/s &= 193.75Kbyte/10ms &= 24.21875 \cdot 8Kbyte/10ms \end{aligned}$$

本研究では，トークンを溜めることができるパケットのサイズを 128 とした．もちろんパケットサイズはアプリケーションに依存する問題で一概には何とも言えない．この為，パケットサイズはスーパーユーザによる変更を可能とした．

4.2　実装

インタフェースを enable する際に `timeout()` を用いることにより，トークンを与える関数 `wfq_addtoken()` が kernel clock (10 μ 秒) 後に呼び出される．引数として，WFQ の状態を保持する構造体へのポインタである `wfqp` を取ることにより，現在のパケットサイズ及び，現在パケットに入っているトークンの量を参照可能となる．

```
timeout((timeout_func_t)wfq_addtoken, (void *)wfqp, hz/100);
```

関数 `wfq_addtoken()` 内では，図 4.2 に示した状態遷移を行ない，トークンを生成してパケットに追加する．関数の実行終了時には `timeout()` により自分自身を再帰的に呼び出し，再び kernel clock 後にトークンの生成を行なう．

なお，インタフェースが disable された際には `untimeout()` により終了する．

```
untimeout((timeout_func_t)wfq_addtoken, (void *)wfqp);
```

4.3　評価

本節では，共有メディアに於ける制御に関して行なった実験の評価結果を示す．評価環境に関しては，前章同様に付録 A を参照して欲しい．

4.3.1　オーバーヘッド

3.3.1 項で行なった WFQ のオーバヘッド測定と同じ条件で，トークンバケットのオーバーヘッド測定を行なった結果を表 4.1 に示す．なお，RED 単独の場合と同様に，トークンバケットは altq (fifo) に追加する形で実装を行なった．

表 4.1: オーバーヘッド (トークンバケット)

カーネル	スループット (Mbps)			
	155Mbps ATM		10Mbps Ethernet	
	UDP	TCP	UDP	TCP
GENERIC	134.63	106.70	9.43	6.47
altq (fifo)	134.63	106.68	9.43	6.49
altq (fifo + token bucket)	131.39	100.91	9.25	5.95

トークンバケットを有効にすると，数%のスループット低下が見られる．これはオーバーヘッドと言うよりも，シェーピング機能により帯域を絞られている為であると考えられる．

4.3.2　フロー間での共有

TCP と UDP の競合実験

共有メディアである Ethernet の同一セグメント上で，適応型フローである TCP と非適応型フローである UDP とで帯域の共有を行なった (図 4.6 参照)．測定は，トークンバケットによるシェーピングをルータ R1，R2 の両者が使っていない場合，TCP 及び UDP のフローを発生するそれぞれ一方が行なっている場合，両者が行なっている場合の計 4 通りで行なった．

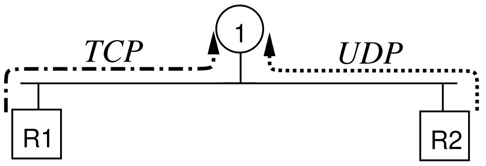


図 4.6: TCP と UDP の競合実験環境 (10Mbps Ethernet，同一セグメント)

測定結果を表 4.2 に示す．2.2 項で示した通り，GENERIC カーネルでは TCP が帯域を確保できないことが分かる．一方，本実装で用いるトークンバケットを用いることにより，UDP のような確認応答による適応を行なわないフローと TCP が混在する状況での帯域の共有が可能となる．また，TCP はネットワークの状態に適応してフローを調整する為，シェーピングは非適応型フローである UDP を発生するノードでのみ行なえば良いことが分かる．

表 4.2: TCP と UDP の競合実験 (10Mbps Ethernet，同一セグメント)

カーネル	フロー	スループット (Mbps)
GENERIC	TCP (R1→1)	0.14
	UDP (R2→1)	9.38
R1 のみ altq (fifo + token bucket)	TCP (R1→1)	0.04
	UDP (R2→1)	9.42
R2 のみ altq (fifo + token bucket)	TCP (R1→1)	2.07
	UDP (R2→1)	6.01
altq (fifo + token bucket)	TCP (R1→1)	1.98
	UDP (R2→1)	6.42

TCP 間での競合実験

ネットワーク中間ノードに於いてシェーピングを行なった際に，適応型フロー間で競合が発生した場合を想定し，図 4.6 と同じ環境で，ルータ R1，R2 共に端末 1 に向けて TCP のフローを発生した場合の結果を表 4.3 に示す．

表 4.3: TCP 間での競合実験 (10Mbps Ethernet，同一セグメント)

カーネル	フロー	スループット (Mbps)
GENERIC	TCP (R1→1)	3.43
	TCP (R2→1)	3.43
R1 のみ altq (fifo + token bucket)	TCP (R1→1)	3.49
	TCP (R2→1)	3.51
altq (fifo + token bucket)	TCP (R1→1)	3.40
	TCP (R2→1)	3.40

4.4　ノードに於ける制御との組合せ

3.3.3 項に於ける測定結果からも明らかなように，共有メディアに於ける適応型フロー及び非適応型フローの競合は，単一ノードを通過するフロー間でも発生する．これは，WFQ 及びトークンバケットを統合したカーネルを用いることによって解決される．なお，評価環境に関しては前節同様に付録 A を参照して欲しい．

4.4.1　オーバーヘッド

3.3.1 項と同じ条件で，WFQ 及びトークンバケットを統合した場合のオーバーヘッドの測定を行なった．この測定結果を表 4.4 に示す．

表 4.4: オーバーヘッド (WFQ，RED + トークンバケット)

カーネル	スループット (Mbps)			
	155Mbps ATM		10Mbps Ethernet	
	UDP	TCP	UDP	TCP
GENERIC	134.63	106.70	9.43	6.47
altq (wfq + red + token bucket)	131.39	100.91	9.25	5.95

表 4.1 で示したトークンバケットのオーバーヘッドと値が全く変わらないことが分かる．このことからも，4.3.1 項で述べたように，実装によってオーバーヘッドが増加したわけではなく，シェーピングの機能により帯域幅が絞られているということが分かる．

4.4.2　スループット

TCP と UDP の競合実験

3.3.3 項の測定に於いて，公平な出力リンクの共有が行なわれなかった図 3.9(右) と同じ条件で，WFQ 及び FIFO，それぞれにトークンバケットを加えたカーネルを用いて評価を行なった．この結果を表 4.5 に示す．

表 4.5: TCP と UDP の競合実験 (10Mbps Ethernet，シェーピング有)

カーネル	フロー	スループット (Mbps)
GENERIC	TCP (1→5)	0.03
	UDP (2→5)	9.39
altq (fifo + token bucket)	TCP (1→5)	0.02
	UDP (2→5)	9.26
altq (wfq + red + token bucket)	TCP (1→5)	2.42
	UDP (2→5)	6.76

当然ではあるが，WFQ によるフローの保護が行なわれなければ，トークンバケットによるトラフィックシェーピングを行なっても帯域の公平な共有は不可能である．WFQ によるフローの保護を行なった場合，表 3.4 と比較してみると違いは明らかである．トークンバケットによるトラフィックシェーピングにより，共有メディアの公平な共有が実現されている．

TCP 間での競合実験

ネットワーク中間ノードに於いて複数の適応型フローが存在した場合のオーバヘッドを測定する為，図 3.9(右) と同じ条件に於いて，端末 2 から端末 5 に向けても TCP のフローを流した場合の測定結果を表 4.6 に示す．

表 4.6: TCP 間での競合実験 (10Mbps Ethernet，シェーピング有)

カーネル	フロー	スループット (Mbps)
GENERIC	TCP (1→5)	3.60
	TCP (2→5)	3.60
altq (fifo + token bucket)	TCP (1→5)	3.58
	TCP (2→5)	3.58
altq (wfq + red + token bucket)	TCP (1→5)	3.58
	TCP (2→5)	3.58

トラフィックシェーピングが，適応型フローに対して影響を及ぼさないことが分かる．

第5章

考察

ノードの出力キューに於いて，WFQによるパケットスケジューリングによりコネクションの孤立化を行ない，適応型フローと非適応型フローの分離を実現した．各フローは，重み付けに応じた公平なスケジューリングが行なわれる(表3.2参照)．

加えて，単なるTail-Dropではなく，各キューの状態を元にドロップするパケットを選択することにより，一時的なバーストトラフィックを不当に扱うというFair Queueingの問題は回避され，より公平な出力キューの共有がフロー間で行なわれるようになった(表3.3参照)．ただし，Ethernetのような共有メディア上では，出力キューに於いて公平なスケジューリングを行なっただけでは，リンクの公平な共有を行なうことができないということが分かった(表3.4参照)．

また，バッファが一杯で無い場合にもパケットドロップを行なうREDによって素早い輻輳通知を行なうことにより，バッファに定常的に空き領域が保たれる(図3.13参照)．enqueueされているパケットが減った為，そのノードを通過する全フローの遅延が減少され，同時に，バーストトラフィックが発生した場合にもパケットドロップが発生しにくくなった．

しかし，ネットワーク中間ノードに於けるキューサイズを見ると，安定状態となっている適応型フローに於いても細かい揺らぎが発生していることが分かった(図3.13参照)．例えパケットの送信周期が一定となっていたとしても，その瞬間の端末やネットワークの負荷等によって，微妙な違いが発生することが伺える．特に，このような揺らぎは，ネットワーク中間ノードに於いてフローが統合される度に増幅すると考えられ，このことからも，各キューに関する情報を保持すると共に，ノードに於いてバッファの空き領域を保持する必要がある．

トークンバケットによるトラフィックシェーピングを行なうことにより，共有メディア上で，帯域の動的かつ公平な割り当てを実現した(表4.2参照)．本研究で提供する枠組は，ノード間に於ける公平性を提供するにも関わらず，他のノードに対する前提を持たないという特徴がある．もちろん，非適応型フローを発生するノードに於いてトラフィックシェーピングが行なわれない場合には，公平性の実現は不可能だが，逆に非適応型フローを発生するノードでのみ行なわれるような環境では，トラフィックシェーピングが適応型フローと協調することにより，帯域の公平な共有を行なうことができる．

一方，トラフィックシェーピングの弊害として，使用可能帯域が絞られることが分かった(表4.1参照)．しかし，共有メディアということは，その帯域を占有できるような状態は稀であり，適応型フローであるTCPによりノード間で帯域幅を共有するような場合には影響がほとんど見られないことが分かる(表4.3参照)．また，ネットワーク中間ノードでは各インタフェース，つ

まりは出力キューに対応するメディアが予め分かっているのが普通であり，共有メディアで無い場合はトラフィックシェーピングを行なわなければ良い．データリンク層の特性を考慮した実装を行なうことにより，トークンバケットのオーバーヘッドの影響は最小限にとどめることができると考えられる．

また，スケジューリングの公平性のみでは実現不可能であった，接続リンクがEthernet等の共有メディアの場合のフロー間の公平性は，このトークンバケットに前述のWFQを組み合わせることにより実現可能となる(表4.5参照)．当然，トークンバケットのオーバーヘッドは含まれることになるが，単独に用いる場合と同様に，適応型フロー間でメディアを共有している限りはほとんどその影響は無く(表4.6参照)，データリンク層に応じた実装を行なえばほとんど影響は無いと考えられる．

本研究ではトークンバケットによるシェーピングをWFQと組み合わせて利用したが，トラフィックシェーピングは出力キューに於いて行なっている為，あらゆるスケジューリングアルゴリズムと組み合わせて実現することができるという柔軟性を持つ．

第6章

関連研究

[Kes91] では，WFQ のスケジューリングに関する計算コストを述べると共に，バッファリング機構の違いによる性能評価を行なっている．しかし，本実装の結果からも分かるように，近年のコンピュータの高性能化に伴い，WFQ の計算コストはほとんど無視できる範囲となっている．また，通常の BSD 系 UNIX ではメモリ機構として mbuf を用いており，バッファリング機構は mbuf の性能に依存することになる．

WFQ の実装コストに占める比率が大きなものとして，パケット到着毎に更新する round number の計算がある．このコストを抑えようとして考えられたのが Self-Clocked Fair Queueing (SCFQ) [Dav90, Gol94] である．しかし，SCFQ は実装が容易である反面，短期間に於ける公平性を保証することはできず，end-to-end の遅延が通常の WFQ よりも大幅に悪くなる場合があった．この遅延に関する問題を解決し，単期間での公平性を向上したものが Start-time Fair Queueing (SFQ) [Goy96] である．また，Worst-case Fair weighted Fair Queueing (WF^{2}Q) [Ben96] では，スケジューリング対象とするパケットの選択を工夫することで，コネクション間での公平性を増している．

このように，WFQ に関する拡張の多くはスケジューリングの工夫に関するものであった．しかし Ethernet のような共有メディア上では，dequeue されたパケットが全て受信者に届くわけではない．よって，スケジューリングの工夫を行なっただけでは，公平な帯域の共有はできない．また，RED などを用いて enqueue されているパケットの数自体を低く抑え，スケジューリング対象を減らす方が，より簡単な形で公平性を増すことができると考えられる．RED は，enqueue されているパケットの数を減らすだけでなく，輻輳制御機構も持つ．このようなネットワーク中間ノードに於ける輻輳制御機構は数多く提案されており [Man91]，代表的なものとして DECbit [Ram88, Ram90] や，ATM Forum で提案されている EERC (End-to-End Rate-based flow-Control) 等がある．どちらの手法に於いても，輻輳を検知した中間ノードがパケットまたはセルに含まれる輻輳通知の為のビットをセットし，受信者が何らかの形でその情報を送信者に返すというものである．送信者は，ネットワーク中間ノードでビットがセットされることと，受信者からその情報が戻されることを前提としている．これに対し，RED は end-to-end に対する仮定を必要としないと言う点で大きく異なる．

[Vil94] では IBM RS/6000 を用いたルータに於いて，RED を用いた実験を行なっている．しかし，実験の目的は高速ネットワークに於ける TCP のスループット測定であり，RED 自体に関する評価はほとんど行なわれていない．また，[Gay96] では NetBSD に RED を実装し，Tail-Drop

と RED との比較を行なっている．しかし，適応型フローである TCP のみを用いた実験は，シミュレーション結果の再試に留まっている観がある．また RED に改良を加え，計算量及びメモリ使用量が少ない Derivative Random Drop (DRD) を提案しているが，RED に於ける問題点は計算量ではなく，フロー間に於ける保護を提供する仕組みであると考えられる．

しかし，RED の提案を行なった [Flo93] では，非適応型のフローを特定する方法は述べられているが，その実現方法に関する言及は行なっていない．これに対し，RED を拡張した Flow Random Early Drop (FRED) [Lin97] では，各フローが enqueue できるパケットの数に動的に上限を設けることにより，これを実現した．また FRED では，各フローに対して，最低でも数個のパケットはドロップされずに enqueue できるという保証も行なっている．ftp のような大量のデータ転送を行なうフローと，telnet のような対話的な通信を行なっているフローとでは，パケットが 1 つドロップされた場合の影響が大きく異なるが，この機構により対話型通信の保護も可能となる．しかし，バッファの使用量に制限を設けることは，バッファの使用効率を低下させるだけでなく，バーストトラフィックにも柔軟に対応できる RED の特性を失う危険性がある．

Ethernet に於ける衝突 (collision) の増大を回避する方法としては，Class Based Queueing (CBQ) [Flo95] 等を用いて各フローを完全に別のものとして扱い，バッファや使用帯域などの全ての資源を完全に分離するというものも考えられる．しかし CBQ では，帯域の使用などに関してネットワーク全体で予め合意しておく必要があり，実用に向けてのコストが大きいと考えられる．また，IEEE 802.4 トークンバス [IEE85] を元に変更を加えた RETHER [Ver95] のような手法もあるが，セグメント中の全てのマシンがトークンバス方式に従うと言う前提が必要となる．

共有メディアに於ける公平性に関連して，Ethernet 上で帯域を保証した通信を行なう為のアドミッションコントロールに関する研究等も行なわれている [Yav97]．本研究で提案した枠組は，このような Ethernet に於ける資源予約を行なう際の前提として考慮すべき問題だと考えられる．

第7章

今後の課題

今回の評価測定は，Round Trip Time (RTT) が非常に小さく，ホップ数も最大で 3 というローカルネットワーク上で行なった．他のトラフィックが全く存在しない環境で測定を行なうことにより，各ノードの動きを詳細に追うことができた．
その一方で，複数のフローが働きあって発生する輻輳や，長い RTT が原因で起こる TCP のバースト性等の実験は行なっていない．WFQ や RED は，ネットワーク中間ノードに於いて，特にこのようなフロー間での公平性を実現に於いて効力を発揮する．これらは，大規模ネットワーク上で実際に運用することにより実証できると考えられるが，大規模ネットワークに於ける実験を行なうのは現実問題として容易ではない．しかし，本研究で示したアルゴリズムや評価結果を用いることにより，より正確なシミュレーションを行なうことができると考えられる．

本研究では，ネットワーク中間ノードに於ける公平性の実現を目標としたが，この公平性の定義に関する問題がある．直観的には，帯域幅を公平に分け合うことが公平な共有ということになる．しかし，帯域幅は単位時間当たりどの程度のデータ量を送信したかということであり，つまりは時間という概念が含まれる．通常，公平性はスケジューリングアルゴリズムを工夫することにより実現されるが，スケジューリングは現在の情報のみに依るのが普通である．本研究では履歴情報を元にパケットドロップを行なったが，スケジューリングにもある程度このような情報を付加することにより，より公平な共有を実現できるのかもしれない．

第8章

結論

ネットワーク中間ノードに於けるフロー制御は，ノード内及びノード間という異なる2つの状況に於いて，各フローをどのように扱うかと言う問題である．

本研究では，ノードに於いてWeighted Fair Queueing (WFQ)を用いてフロー毎の保護を行なうことにより，フロー間での公平性を実現した．既存のスケジューリングアルゴリズムは，現時点から未来に向けての公平性のみを考慮していたのに対し，本研究で提供する枠組は，そのフローが発生して以来の履歴情報を用いて公平性を提供する．また，Random Early Detection (RED)により，輻輳制御を行なうと共に，一時的なバーストトラフィックへの対応，対話型通信の保護及び遅延の減少を実現した．

一方，トークンバケットアルゴリズムによるトラフィックシェーピングを用いて，データリンク層の状態に応じてパケット転送を行なうことにより，共有メディアの帯域幅をノード間で公平に共有する機構を実現した．

両者を組み合わせることにより，ネットワーク中間ノードに於いて，コネクション，トラフィック及びネットワークの状態に動的に適応し，各フローに対して公平かつ効率の良い転送を実現することが可能となる．

本研究で提供する枠組は全て，end-to-endに於ける前提条件を全く必要としない為，いかなる量，数，種類のパケット，フロー，データリンク層に於いてもフロー間での公平性を提供できる．

謝辞

本研究を行なう機会を与えて頂き，本研究全般に於いて常に的確な指導をして下さいました慶應義塾大学理工学部客員教授 所 真理雄 博士，慶應義塾大学理工学部助教授教授 天野 英晴 博士に心から感謝致します．

(株) ソニーコンピュータサイエンス研究所の長 健二郎 氏には，研究の端緒を与えて頂いただけでなく，お忙しい中で多くの時間を割いて頂き，暖かい励ましと貴重な御意見を頂き大変感謝しております．

所研究室 修士2年の三輪泰孝 氏，矢崎啓之 氏は，共に研究し，本研究の多くの段階に於いて多大な御協力を頂き，心強かったと同時に深く感謝しております．また本研究に関わらず度々相談に乗って頂き，いろいろとお付き合い頂いた所研究室ネットワークグループの福田健介 氏，舌間一宏 氏を始め，研究室の全ての方々に感謝致します．

最後に，多くの励ましを与えてくれた両親と姉，弟に心から感謝します．

参考文献

[Ben96] Jon C. R. Bennet and Hui Xhang: WF^{2}Q : Worst-Case Fair Weighted Fair Queueing, in *IEEE INFOCOM '96*, 1996, (`ftp://ftp.cs.cmu.edu/user/hzhang/INFOCOM96.ps.Z`).

[Bog88] D. R. Boggs, et al.: Measured Capacity of an Ethernet : Myths and Reality, in *ACM SIGCOMM '88*, Aug, 1988.

[Bra95] Lawrence. S. Brakmo and Larry. L. Peterson: TCP Vegas : End to End Congestion Avoidance on a Global Internet, *IEEE Journal on Selected Areas in Communication*, Vol. 13, No. 8, pp. 1465–1480, Oct, 1995, (`http://www.cs.arizona.edu/Xkernel/Papers/jsac.ps.Z`).

[Bra97] Internet Engineering Task Force — Bob Braden, et al.: Recommendations on Queue Management and Congestion Avoidance in the Internet, Internet-Draft, Mar, 1997, (`ftp://ds.internic.net/internet-drafts/draft-irtf-e2e-queue-mgt-00.txt`).

[Cho97] Kenjiro Cho: Alternate Queueing for BSD Unix, in *1998 USENIX Annual Technical Conference (Submission)*, 1997, (`http://www.csl.sony.co.jp/person/kjc/programs.html`).

[Cla82] Internet Engineering Task Force — David D. Clark: Window and Acknowledgement Strategy in TCP, Internet Request For Comment RFC813, Jul, 1982, (`http://www.etext.org/Internet/RFC/rfc813.txt`).

[Dav90] J. Davin and A. Heybey: A Simulation Study of Fair Queuing and Policy Enforcement, *Computer Communication Review*, Vol. 20, No. 5, pp. 23–29, Oct, 1990.

[Dem89] Alan Demers, Srinivasan Keshav, and Scott Shenker: Analysis and Simulation of a Fair Queueing Algorithm, in *ACM SIGCOMM '89*, Sep, 1989.

[Fel89] David C. Feldmeier: Estimated performance of a gateway routing-table cache, Technical report, MIT Laboratory for Computer Science, Mar, 1989.

[Flo91] Sally Floyd and Van Jacobson: On Traffic Phase Effects in Packet-Switched Gateways, *ACM Computer Communication Review*, Vol. 21, No. 2, Apr, 1991, (`ftp://ftp.ee.lbl.gov/papers/phase.ps.Z`).

[Flo93] Sally Floyd and Van Jacobson: Random Early Detection Gateways for Congestion Avoidance, in *IEEE/ACM Transactions on Networking*, Aug, 1993, (`ftp://ftp.ee.lbl.gov/papers/early.ps.gz`).

[Flo95] Sally Floyd and Van Jacobson: Link-sharing and Resource Management Models for Packet Networks, *IEEE/ACM Transactions on Networking*, Vol. 3, No. 4, Aug, 1995, (`ftp://ftp.ee.lbl.gov/talks/linktalk.ps.Z`).

[Fre92] The FreeBSD Project, 1992, (`http://www.freebsd.org/`).

[Gay96] Mark Gaynor: Proactive Packet Dropping Methods for TCP Gateways, Technical report, Ecole Nationale Supérieure des Télécommunications de Bretagne, Networks and Multimedia Department, Oct, 1996, (`http://www.eecs.harvard.edu/~gaynor/`).

[Gol94] S. Jamaoddin Golestani: A Self-Clocked Fair Queueing Scheme for Broadband Application, in *IEEE INFOCOM '94*, pp. 636–646, Jun, 1994.

[Goy96] Pawan Goyal, Harrick M. Vin, and Haichen Cheng: Start-Time Fair Queueing : A Scheduling Algorithm for Integrated Services Packet Switching Networks, in *ACM SIGCOMM '96*, Aug, 1996, (`http://www.acm.org/sigcomm/sigcomm96/papers/goyal.ps`).

[Hah86] E. Hahne and R. Gallager: Round Robin Scheduling for Fair Flow Control in Data Communications Networks, in *IEEE International Conference on Communications*, Jun, 1986.

[Han75] Hansen: *A Table of Series and Product*, Prentice Hall Publishing Company, Inc., 1975.

[Has89] E. Hashem: Analysis of Random Drop for Gateway Congestion Control, Technical Report LCS TR-465, Laboratory for Computer Science Massachusetts Institute of Technology, 1989.

[IEE85] IEEE Inc: *IEEE/ANSI Standard 802.4 – Token Passing Bus Access Method and Physical Layer Specifications*, 1985.

[ISO93] ISO/IEC 8802-3: *Carrier Sense Multiple Acccess with Collision Detection (CSMA/CD) Access Method and Physical Layer Specifications*, 4th edition, Jul, 1993.

[Jac88] Van Jacobson: Congestion Avoidance and Control, in *ACM SIGCOMM '88*, Vol. 18, pp. 314–329, Aug, 1988, (`ftp://ftp.ee.lbl.gov/papers/congavoid.ps.Z`).

[Jon93] Rick Jones, Hewlett-Packard: Netperf : A Benchmark for Measuring Network Perfromance, 1993,
(`http://www.cup.hp.com/netperf/`).

[Kes91] Srinivasan Keshav: On the Efficient Implementation of Fair Queueing, *Internetworking: Research and Experience*, Vol. 2, No. 3, pp. 157–173, Sep, 1991,
(`ftp://tenet.berkeley.edu/pub/tenet/Papers/Keshav91b.ps`).

[Lak96] T. V. Lakshman, Arnie Neidhards, and Teunis J. Ott: The Drop from Front Strategy in TCP over ATM, in *IEEE INFOCOM '96*, 1996.

[Lin97] Dong Lin and Robert Morris: Dynamics of Random Early Detection, in *ACM SIGCOMM '97*, Sep, 1997,
(`http://www.acm.org/sigcomm/sigcomm97/advance-program.html#ab078`).

[Man91] Internet Engineering Task Force — A. Mankin, Kadangode K. Ramakrishnan, and editors for the Performance and Congestion Control Working Group: Gateway Congestion Control Survey, Internet Request For Comment RFC1254, Aug, 1991,
(`http://www.etext.org/Internet/RFC/rfc1254.txt`).

[McC96] Steven McCanne, Sally Froyd, and Kevin Fall, Lawrence Berkeley National Laboratory: UCB/LBNL/VINT Network Simulator NS, Nov, 1996,
(`http://www-mash.cs.berkeley.edu/ns/`).

[McK91] Paul E. McKenney: Stochastic Fairness Queing, *Internetworking: Research and Experience*, Vol. 2, pp. 113–131, Jan, 1991.

[McK96] Marshall Kirk McKusick, Keith Bostic, Michael J. Karels, and John S. Quarterman: *The Design and Implementation of the 4.4 BSD Operating System*, Addison-Wesley Publishing Company, Inc., 1996.

[Nag87] Joins Nagle: On Packet Switched with Infinite Storage, *IEEE Transaction on Communications*, Vol. 35, No. 4, pp. 435–438, Apr, 1987.

[Ram88] Kadangode K. Ramakrishnan and Raj Jain: A Binary Feedback Scheme for Congestion Avoidance in Computer Networks with a Connectionless Network Layer, in *ACM SIGCOMM '88*, pp. 303–313, Aug, 1988.

[Ram90] Kadangode K. Ramakrishnan and Raj Jain: A Binary Feedback Scheme for Congestion Avoidance in Computer Networks, *ACM Transactions on Computer Systems*, Vol. 8, No. 2, pp. 158–181, 1990.

[Riz97] Luigi Rizzo: RED and Non-Responsive Flows, end-to-end mailing list, Jun, 1997,
(`ftp://ftp.isi.edu/end2end/end2end-interest-1997.mail`).

[Shr95] M. Shreedhar and George Barghese: Efficient Fair Queuing using Deficit Round Robin, in *ACM SIGCOMM '95*, Sep, 1995,
(`http://dworkin.wustl.edu/~varghese/PAPERS/fq.ps`).

[Tur86] J. S. Turner: New Directions in Communications (or Which Way to the Information Age), *IEEE Communication Magazine*, Vol. 24, pp. 8–15, Oct, 1986.

[Ver95] Chitra Verkatramani and Tzi cker Chiueh: Design, Implementation, and Evaluation of a Software-based Real-Time Ethernet Protocol, in *ACM SIGCOMM '95*, Aug, 1995,
(`http://www.acm.org/sigcomm/sigcomm95/papers/venkatramani.ps`).

[Vil94] Curtis Villamizar and Cheng Song: High Performance TCP in ANSNET, *ACM Computer Communication Review*, Vol. 24, No. 5, pp. 45–60, Oct, 1994,
(`ftp://engr.ans.net/pub/papers/tcp-performance.ps`).

[Yav97] Internet Engineering Task Force — Raj Yavatkar, et al.: SBM (Subnet Bandwidth Manager) : A Proposal for Admission Control over IEEE 802-style networks, Internet-Draft, Jul, 1997,
(`ftp://ds.internic.net/internet-drafts/draft-ietf-issll-is802-sbm-04.txt`).

[Zha89] Lixia Zhang: A New Architecture for Packet Switching Network Protocols, Technical Report LCS TR-455, Laboratory for Computer Science Massachusetts Institute of Technology, 1989.

付　録

A 評価環境

A.a netperf パラメータ

`netperf` による測定は以下の条件の下で行なった．

- 100 回の測定の平均値を用いる
- 測定時間は 1 回当たり 10 秒間とする

また，特別な指定が無い場合は各インタフェースに応じて表 A.1 に挙げるパラメータを用いる．

表 A.1: netperf パラメータ

インタフェース	フロー	ソケットサイズ (bytes)		パケット長 (bytes)
		送信	受信	
155Mbps ATM	TCP	57,344	57,344	8,192
	UDP		—	
10Mbps Ethernet	TCP	32,768	32,768	1,024
	UDP		—	

A.b ネットワークカード

3，4 章で行なった評価の際に用いたネットワークカードを表 A.2 に示す．

表 A.2: ネットワークカード

ネットワーク	製造元	カード
10Mbps Ethernet	3Com	3C905 Fast Etherlink XL PCI
155Mbps ATM	Efficient Networks	ENI-155p

A.c 端末

本文中で用いた端末番号と端末仕様との対応付けを，表 A.3 に示す．全ての端末がオペレーティングシステムとして，FreeBSD 2.2.1R に altq のパッチを当てたものを用いている．

表 A.3: 端末

端末番号	CPU	主記憶
1，2，3，R1	Pentium 166.19MHz 586-class	64Mbyte
4，R2	Pentium 199.43MHz 586-class	64Mbyte
5	Pentium 81.18MHz 586-class	64Mbyte

B インタフェース

B.a ioctl

WFQ 及びトークンバケットの制御は，デバイスファイル`/dev/wfq`に対して`ioctl`システムコールを用いて行なう．また，このユーザインタフェースとして WFQ モニタ及び WFQ デーモンを実装した．

```
#include<sys/fcntl.h>
#include<netinet/wfq.h>

int fd;
char *argp;

fd = open("/dev/wfq", O_RDONLY);
ioctl(fd, request, argp);
```

request には表 B.1 に挙げる 11 個の制御要求のいずれかが入る．

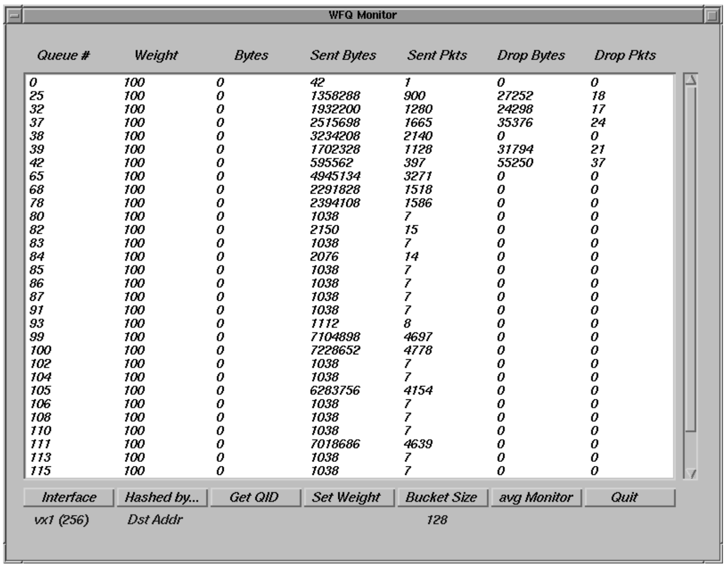
表 B.1: `/dev/wfq` に対する ioctl

request	説明
WFQ_ENABLE	WFQ インタフェースを有効化する
WFQ_DISABLE	WFQ インタフェースを無効化する
WFQ_IF_ATTACH	メモリ確保，変数設定等の初期化作業を行なう
WFQ_IF_DETACH	メモリ解放等の終了作業を行なう
WFQ_CHECK_HASHFNC	利用されているハッシュ関数を調べる
WFQ_CHANGE_HASHFNC	ハッシュ関数の変更を行なう
WFQ_BUCKET_SIZE	トークンバケットサイズの変更を行なう
WFQ_GET_QID	フローがどのキューにハッシュされるかを調べる
WFQ_SET_WEIGHT	あるキューに対する重み付けの変更を行なう
WFQ_QUEUE_STATS	あるキューの状態を得る
WFQ_WFQ_STATE	WFQ の状態を得る

基本的な制御はスーパーユーザ権限を必要とするが，情報を取り出すのみの *request* である WFQ_CHECK_HASHFNC，WFQ_GET_QID，WFQ_QUEUE_STATS，WFQ_WFQ_STATE は一般ユーザでも利用可能である．

B.b　WFQ モニタ

WFQ モニタは，利用されているキューに関する情報を視覚的に表示するユーザインタフェースである．図 B.1 に示したように，左から順にキューの番号 (Queue #)，重み付け (Weight)，現在 enqueue されているバイト数 (Byte)，送信バイト数 (Sent Byte)，送信パケット数 (Sent Pkts)，ドロップバイト数 (Drop Byte)，ドロップパケット数 (Drop Pkts) が表示され，1 秒毎に更新される．



Queue #	Weight	Bytes	Sent Bytes	Sent Pkts	Drop Bytes	Drop Pkts
0	100	0	42	1	0	0
25	100	0	1358288	900	27252	18
32	100	0	1932200	1280	24298	17
37	100	0	2515698	1665	35376	24
38	100	0	3234208	2140	0	0
39	100	0	1702328	1128	31794	21
42	100	0	595562	397	55250	37
65	100	0	4945134	3271	0	0
68	100	0	2291828	1518	0	0
78	100	0	2394108	1586	0	0
80	100	0	1038	7	0	0
82	100	0	2150	15	0	0
83	100	0	1038	7	0	0
84	100	0	2076	14	0	0
85	100	0	1038	7	0	0
86	100	0	1038	7	0	0
87	100	0	1038	7	0	0
91	100	0	1038	7	0	0
93	100	0	1112	8	0	0
99	100	0	7104898	4697	0	0
100	100	0	7228652	4778	0	0
102	100	0	1038	7	0	0
104	100	0	1038	7	0	0
105	100	0	6283756	4154	0	0
106	100	0	1038	7	0	0
108	100	0	1038	7	0	0
110	100	0	1038	7	0	0
111	100	0	7018686	4639	0	0
113	100	0	1038	7	0	0
115	100	0	1038	7	0	0

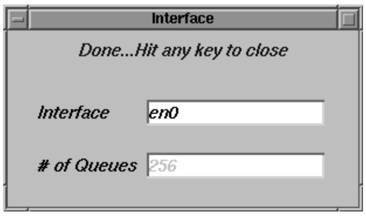
vx1 (256)　Dst Addr　128

図 B.1: WFQ モニタ

ウィンドウ下部に 8 つのボタンがあり，ボタンの凹凸はインタフェースの状態によって変わる．凸型となっているものが実行可能であり，Quit をクリックすると終了する．

1. インタフェース

   Interface ボタンをクリックすることにより，インタフェースの有効化/非有効化を行なうことができる．インタフェースが有効化されていない場合は図 B.2 に示すウィンドウが表示され，インタフェース名，キューの数を指定する．有効化が行なわれると，ボタンの下にインタフェース名が表示される．逆にインタフェースが有効化されている際にボタンをクリックすると，インタフェースを無効化してよいかを確認してくる (図 B.3 参照)．

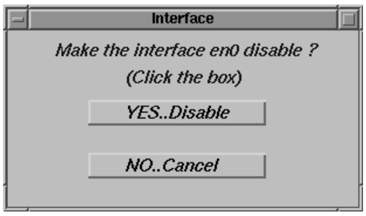


Done...Hit any key to close

Interface: en0

# of Queues: 256

図 B.2: 有効化ウィンドウ



Make the interface en0 disable ?
(Click the box)

図 B.3: 非有効化ウィンドウ

キューの数は Queues ボタンの下に表示される．キューの数はインタフェースを有効化する際にのみ指定可能で，その後の変更はできない．なお，何も指定しない場合には 256 となる．

2. ハッシュ関数

   Hashedy by... ボタンの下に，現在利用されているハッシュアルゴリズムが示される．このボタンをクリックすると図 B.4 に示すウィンドウが表示され，ハッシュ関数の変更を行なうことができる．

Click the box !
(Selected box is colored yellow)

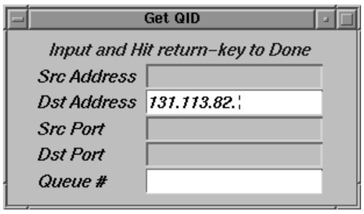
☐ Dst Addr
☐ Src Port
☐ Full (SrcAddr,DstAddr,SrcPort,DstPort)

Hit any key to close

図 B.4: ハッシュ関数指定ウィンドウ

3. キュー番号

   各フローがどのキューにハッシュされるかが分からなければ，フローに関する情報を得たり，また重み付けの変更を行なうこともできない．Get QID ボタンをクリックすると図 B.5 に示すウィンドウが表示される．白抜きの部分にアドレス及びポート番号を入力することにより，フローがハッシュされるキューの番号が得られる．なお，白抜き部分は利用しているハッシュ関数によって異なる．



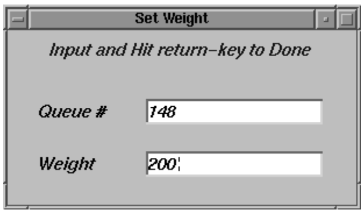
Input and Hit return-key to Done

Src Address:
Dst Address: 131.113.82.
Src Port:
Dst Port:
Queue #:

図 B.5: キュー番号取得ウィンドウ

4. 重み付け

   Set Weight ボタンによりキューの重み付けを変更できる (図 B.6 参照)．なお，Queue # は前述の Get QID ボタンにより求めることができる．



Input and Hit return-key to Done

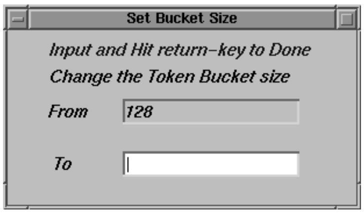
Queue #: 148

Weight: 200

図 B.6: 重み付け指定ウィンドウ

5. トークンバケットサイズ

   トークンバケットアルゴリズムに於いて，バーストの大きさを制限するバケットサイズを変更できる．Bucketsize ボタンをクリックし，バケットを入力する (図 B.7 参照)．



Input and Hit return-key to Done
Change the Token Bucket size
From 128
To

図 B.7: バケットサイズ指定ウィンドウ

6. 平均キューサイズ

   avg Monitor ボタンをクリックすることにより，平均キューサイズのモニタを行なうことができる (図 B.8 参照)．

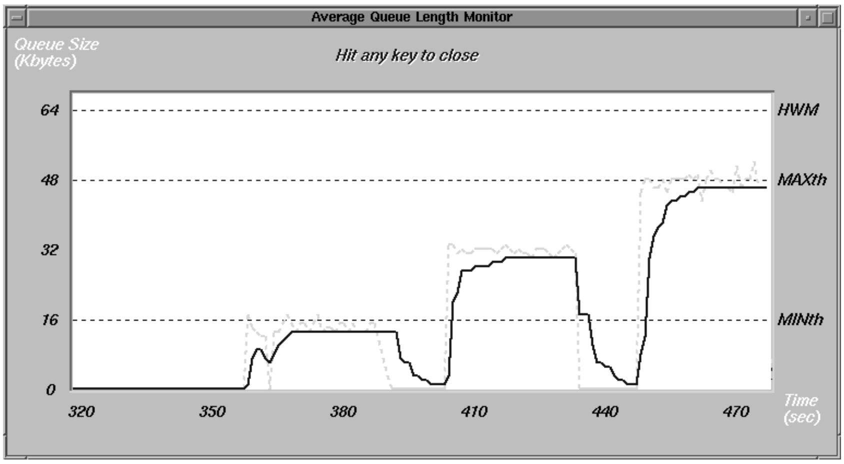


図 B.8: 平均キューサイズモニタ

破線で表示されるのが実際のキューサイズであり，実線で表示されるのが平均のキューサイズである．値は 1 秒間隔でプロットされ，80 秒毎にスクロールされる．

B.c　WFQ デーモン

WFQ デーモンは，コマンドラインから WFQ の制御を行なうためのユーザインタフェースである．

1. インタフェース

   インタフェースの有効化は `enable` コマンド，無効化は `disable` コマンドによって行なう．インタフェースを有効化する際にキューの数を指定することができるが，何も指定しない場合には 256 となる．また，`quit` コマンドにより WFQ デーモンを終了することができる．

```
> if_name enable [ nqueues ] ↵
  ⋮
> if_name disable ↵
> quit ↵
```

なお，`quit` コマンドが入力されたときには，有効化されている全てのインタフェースを非有効化するため，`disable` を実行せずに終了することもできる．

2. ハッシュ関数

現在利用されているハッシュ関数を調べるには，`chkhash` コマンドを用いる．また，ハッシュアルゴリズムを変更する際には，`cnghash` コマンドを DSTADDR，SRCPORT，FULL のいずれかを引数として入力する．

```
> if_name chkhash ↵
> if_name cnghash DSTADDR|SRCPORT|FULL ↵
```

3. キュー番号

`getqid` コマンドにより，各フローがハッシュされるキューの番号を得ることができる．`getaid` コマンドは，利用されているハッシュアルゴリズムに従って以下の 3 つの書式をとる．なお，ハッシュアルゴリズムの確認は，前述の `chkhash` コマンドを用いて行なうことができる．

```
> if_name getqid DstAddr ↵
> if_name getqid SrcPort ↵
> if_name getqid SrcAddr DstAddr SrcPort DstPort ↵
```

4. 重み付け

キューの重み付けを変更には `setweight` コマンドを用いる．qid を引数として取るが，これは前述 `getqid` コマンドにより得ることができる．なお，weight の初期値は 100 である．

```
> if_name setweight qid weight ↵
```

5. トークンバケットサイズ

`bucketsize` コマンドにより，トークンバケットアルゴリズムに於いて，バーストの大きさを制限するバケットサイズを変更できる．なお，size の初期値は 128 である．

```
> if_name bucketsize size ↵
```

C ソースコード

C.a wfq.h

```
#ifndef __NETINET_WFQ_H
#define __NETINET_WFQ_H

#include<netinet/in.h>

#ifndef RED
#define RED                     /* Enable Random Early Detection Algorithm */
#endif /* RED */
#ifndef TOKEN_BUCKET
#define TOKEN_BUCKET            /* Enable Token Bucket Algorithm */
#endif /* TOKEN_BUCKET */

#define DEFAULT_QSIZE           256
#define MAX_QSIZE               2048
#ifdef TOKEN_BUCKET
#define DEFAULT_BUCKET          128
#endif /* TOKEN_BUCKET */

#ifdef KERNEL

#define HWM                     (64 << 10)
#define WFQ_QUOTA               512     /* quota bytes to send at a time */
#define WFQ_ADDQUOTA(qp)        ((qp)->quota += WFQ_QUOTA * (qp)->weight / 100)

#define ENABLE                  0       /* enable interface */
#define DISABLE                 1       /* disable interface */
#define CHECK                   0       /* check the hashing policy */
#define CHANGE                  1       /* change the hashing policy */

typedef struct weighted_fair_queue{
```

```
    struct weighted_fair_queue *next, *prev;
    struct mbuf *head, *tail;
    int quota;                          /* bytes sent in this round */
    int weight;                         /* weight in percent */
    int bytes;                          /* bytes in this queue */
    long idle;                          /* start time of idling */
    int sent_bytes;                     /* bytes sent in this queue */
    int sent_packets;                   /* packets sent in this queue */
    int drop_bytes;                     /* bytes dropped in this queue */
    int drop_packets;                   /* dropped packets */
}wfq;


typedef struct wfq_state{
    struct wfq_state *next;             /* for wfqstate list */
    struct ifnet *ifp;
    int nqueues;                        /* number of queues */
    int bytes;                          /* total bytes in all the queues */
    wfq *rrp;                           /* round robin pointer */
    wfq *queue;                         /* pointer to queue list */
    int hash_policy;                    /* policy for hash function */
    int (*hash_fnc)(struct flowinfo *, int);
    int hwm;                            /* high water mark */
#ifdef TOKEN_BUCKET
    int token;                          /* token for dequeue */
    int bucket;                         /* token bucket size */
#endif /* TOKEN_BUCKET */
#ifdef RED
    int count;                          /* packet count since last dropped */
    int avg;                            /* average enqueued bytes */
#endif /* RED */
}wfq_state_t;


#ifdef RED

/*
 * feedback weight : 0.001953125 = 2^(-9)
 */
#define RED_WAIT        0.001953125     /* the value for Wq : not in use */
#define RED_IWAIT       9               /* negative power of two */
```

```
#define RED_AVG_SHIFT   10

/*
 * maximum probability : 0.03125 = 2^(-5)
 */
#define RED_PB_SHIFT    15
#define RED_MAXp        0.3125          /* the value for MAXp : not in use */
#define RED_IMAXp       5               /* negative power of two */
#define RED_SMAXp       (1 << (RED_PB_SHIFT - RED_IMAXp))

/*
 * scalled minimum and maximum queue threshold for dropping packets
 */
#define RED_MINth       (16 << 10)
#define RED_MAXth       (48 << 10)

/*
 * as defined ...
 *  Floyd Sally, Jacobson Van,
 *  Random Early Detection Gateways for Congestion Avoidance
 *  IEEE/ACM Transactions on Networking, August 1993
 *
 * C1 = (RED_MAXp) / ((RED_MAXth) - (RED_MINth)) = 2^(-20)
 * C2 = ((RED_MAXp) * (RED_MINth)) / ((RED_MAXth) - (RED_MINth)) = 2^(-6)
 */
#define RED_C1          0.00000095367431640625  /* not in use */
#define RED_IC1         20                      /* negative power of two */
#define RED_SIC1        (RED_IC1 - RED_PB_SHIFT)
#define RED_C2          0.015625                /* not in use */
#define RED_IC2         6                       /* negative power of two */
#define RED_SC2         (1 << (RED_PB_SHIFT - RED_IC2))

#define RED_MAXr        1
#define RED_SMAXr       (RED_MAXr << RED_PB_SHIFT)

#endif /* RED */

#endif /* KERNEL */

#ifndef IFNAMSIZ
#define IFNAMSIZ        16
```

```
#endif /* IFNAMSIZ */

typedef struct wfq_interface{
    char wfq_ifacename[IFNAMSIZ];
    u_int wfq_ifacelen;
    int nqueues;
}wfq_iface_t;

#define WFQ_ENABLE              _IOW(’Q’, 1, struct wfq_interface)
#define WFQ_DISABLE             _IOW(’Q’, 2, struct wfq_interface)

#define WFQ_IF_ATTACH           _IOW(’Q’, 3, struct wfq_interface)
#define WFQ_IF_DETACH           _IOW(’Q’, 4, struct wfq_interface)

#include <net/if_altq.h>

struct wfq_getqid{
    wfq_iface_t         iface;
    struct flowinfo     flow;
    int                 qid;
};

#define WFQ_GET_QID             _IOWR(’Q’, 5, struct wfq_getqid)

struct wfq_setweight{
    wfq_iface_t iface;
    int qid;
    int weight;
};

#define WFQ_SET_WEIGHT          _IOWR(’Q’, 6, struct wfq_setweight)

struct wfq_queuestate{
    wfq_iface_t iface;
    int qid;
    int weight;                 /* weight in percent */
    int bytes;                  /* bytes in this queue */
    long idle;                  /* start time of idling */
    int sent_bytes;             /* bytes sent in this queue */
    int sent_packets;           /* number of packets sent in this queue */
    int drop_bytes;             /* bytes dropped in this queue */
```

```
    int drop_packets;           /* number of packets dropped in this queue */
};

#define WFQ_QUEUE_STATS         _IOWR('Q', 7, struct wfq_queuestate)

struct wfq_wfqstate{
    wfq_iface_t iface;
    int nqueues;                /* number of queues */
    int bytes;                  /* total bytes in all the queues */
    int hash_policy;            /* policy for hash function */
    int hwm;                    /* high water mark */
#ifdef TOKEN_BUCKET
    int token;                  /* token for dequeue */
    int bucket;                 /* token bucket size */
#endif /* TOKEN_BUCKET */
#ifdef RED
    int count;                  /* packet count since last dropped */
    int avg;                    /* average enqueued bytes */
#endif /* RED */
};

#define WFQ_WFQ_STATE           _IOWR('Q', 8, struct wfq_wfqstate)

struct wfq_changehashfnc{
    wfq_iface_t iface;
    int hash_policy;
};

#define WFQ_HASH_DSTADDR        0       /* hash by dst address */
#define WFQ_HASH_SRCPORT        1       /* hash by src port */
#define WFQ_HASH_FULL           2       /* hash by all fields */

#define WFQ_CHANGE_HASHFNC      _IOWR('Q', 9, struct wfq_changehashfnc)
#define WFQ_CHECK_HASHFNC       _IOWR('Q', 10, struct wfq_changehashfnc)


#ifdef TOKEN_BUCKET
struct wfq_bucketsize{
    wfq_iface_t iface;
    int bucket;
};
```

```
#define WFQ_BUCKET_SIZE         _IOWR('Q', 11, struct wfq_bucketsize)
#endif /* TOKEN_BUCKET */

#endif /* __NETINET_WFQ_H */
```

C.b wfq.c

```
#include "opt_altq.h"
#ifdef WFQ

#include <sys/types.h>
#include <sys/param.h>
#include <sys/mbuf.h>
#include <sys/uio.h>
#include <sys/socket.h>
#include <sys/systm.h>
#include <sys/proc.h>
#include <sys/errno.h>
#include <sys/time.h>
#include <sys/kernel.h>

#include <net/if.h>
#include <net/if_types.h>
#include <netinet/wfq.h>

/* #define WFQ_DEBUG */

static int              wfq_setenable(struct wfq_interface *, int);
static int              wfq_ifattach(struct wfq_interface *);
static int              wfq_ifdetach(struct wfq_interface *);
static int              wfq_ifenqueue(struct ifnet *, struct mbuf *,
                                        struct flowinfo *, int);
static wfq              *wfq_maxqueue(wfq_state_t *);
static wfq              *wfq_maxthqueue(wfq_state_t *);
static struct mbuf      *wfq_ifdequeue(struct ifnet *, int);
static int              wfq_flush(struct ifnet *);
static int              wfq_getqid(struct wfq_getqid *);
static int              wfq_setweight(struct wfq_setweight *);
static int              wfq_queuestate(struct wfq_queuestate *);
static int              wfq_wfqstate(struct wfq_wfqstate *);
static int              wfq_changehashfnc(struct wfq_changehashfnc *, int);
int                     wfq_hashbyfull(struct flowinfo *, int);
int                     wfq_hashbydstaddr(struct flowinfo *, int);
int                     wfq_hashbysrcport(struct flowinfo *, int);
#ifdef RED
static int              random_drop(wfq_state_t *, wfq *);
```

```
#endif /* RED */
#ifdef TOKEN_BUCKET
static void             wfq_addtoken(wfq_state_t *);
static int              wfq_bucketsize(struct wfq_bucketsize *);
#endif /* TOKEN_BUCKET */


/* global value */
static wfq_state_t *wfq_list = NULL;    /* pointer to wfq queue list */

static int
wfq_setenable(ifacep, flag)
     struct wfq_interface *ifacep;
     int flag;
{
  struct ifnet *ifp;
  wfq_state_t *wfqp;
  int error;

  if((ifp = ifunit(ifacep->wfq_ifacename)) == NULL)
    return ENXIO;

  wfqp = (wfq_state_t *)ifp->if_altqp;

  switch(flag){
  case ENABLE:
    error = if_altqenable(wfqp->ifp, 1);
    break;
  case DISABLE:
    error = if_altqdisable(wfqp->ifp);
    break;
  }
  return error;
}


static int
wfq_ifattach(ifacep)
     struct wfq_interface *ifacep;
{
  int error = 0, i;
```

```
  struct ifnet *ifp;
  wfq_state_t *new_wfqp;
  wfq *queue;
  struct timeval now;

  if((ifp = ifunit(ifacep->wfq_ifacename)) == NULL){
#ifdef WFQ_DEBUG
    printf("wfq_ifattach()...no ifp found\n");
#endif
    return ENXIO;
  }

  if(! ALTQ_IS_READY(ifp)){
#ifdef WFQ_DEBUG
    printf("wfq_ifattach()...altq is not ready\n");
#endif
    return ENXIO;
  }

  if(ifacep->nqueues <= 0 ||  MAX_QSIZE < ifacep->nqueues)
    ifacep->nqueues = DEFAULT_QSIZE;

  /* allocate and initialize wfq_state_t */
  MALLOC(new_wfqp, wfq_state_t *, sizeof(wfq_state_t), M_DEVBUF, M_WAITOK);
  bzero(new_wfqp, sizeof(wfq_state_t));
  MALLOC(queue, wfq *, sizeof(wfq) * ifacep->nqueues, M_DEVBUF, M_WAITOK);
  bzero(queue, sizeof(wfq) * ifacep->nqueues);

  new_wfqp->next = wfq_list;
  wfq_list = new_wfqp;

  /* keep the ifp */
  new_wfqp->ifp         = ifp;
  new_wfqp->nqueues     = ifacep->nqueues;
  new_wfqp->bytes       = 0;
  new_wfqp->rrp         = NULL;
  new_wfqp->queue       = queue;
  new_wfqp->hash_policy = WFQ_HASH_DSTADDR;
  new_wfqp->hash_fnc    = wfq_hashbydstaddr;
  new_wfqp->hwm         = HWM;
#ifdef TOKEN_BUCKET
```

```
  new_wfqp->bucket      = DEFAULT_BUCKET;
  new_wfqp->token       = DEFAULT_BUCKET;
#endif /* TOKEN_BUCKET */
#ifdef RED
  new_wfqp->count       = 0;
  new_wfqp->avg         = 0;
#endif /* RED */

  microtime(&now);
  for(i = 0; i < new_wfqp->nqueues; i++, queue++){
    queue->next         = queue->prev           = NULL;
    queue->head         = queue->tail           = NULL;
    queue->quota        = 0;
    queue->weight       = 100;
    queue->bytes        = 0;
    queue->idle         = now.tv_sec;
    queue->sent_bytes   = queue->sent_packets   = 0;
    queue->drop_bytes   = queue->drop_packets   = 0;
  }

  /*
   * set WFQ to this ifnet structure.
   */
  if(if_altqattach(ifp, new_wfqp, wfq_ifenqueue, wfq_ifdequeue, ALTQT_WFQ))
    (void)wfq_ifdetach(ifacep);

#ifdef TOKEN_BUCKET
  timeout((timeout_func_t)wfq_addtoken, (void *)new_wfqp, hz/100);
#endif /* TOKEN_BUCKET */

  return error;
}


static int
wfq_ifdetach(ifacep)
     struct wfq_interface *ifacep;
{
  int error = 0;
  wfq_state_t *wfqp;
  struct ifnet  *ifp;
```

```
  if((ifp = ifunit(ifacep->wfq_ifacename)) == NULL)
    return ENXIO;

  wfqp = (wfq_state_t *)ifp->if_altqp;

  /* free queued mbuf */
  wfq_flush(ifp);

  /* remove WFQ from the ifnet structure. */
  (void)if_altqdisable(ifp);
  (void)if_altqdetach(ifp);

  /* remove from the wfqstate list */
  if(wfq_list == wfqp)
    wfq_list = wfqp->next;
  else{
    wfq_state_t *wp = wfq_list;
    do {
      if(wp->next == wfqp){
        wp->next = wfqp->next;
        break;
      }
    }while((wp = wp->next) != NULL);
  }

#ifdef TOKEN_BUCKET
  untimeout((timeout_func_t)wfq_addtoken, (void *)wfqp);
#endif /* TOKEN_BUCKET */

  /* deallocate wfq_state_t */
  FREE(wfqp->queue, M_DEVBUF);
  FREE(wfqp, M_DEVBUF);
  return error;
}


static int
wfq_ifenqueue(ifp, mp, flow, mode)
     struct ifnet *ifp;
     struct mbuf *mp;
```

```
     struct flowinfo *flow;
     int mode;
{
  wfq_state_t *wfqp;
  wfq *queue, *drop_queue = NULL;
  int q_id, byte, error = 0;

  if(mode != ALTEQ_NORMAL)
    return 0;

  wfqp = (wfq_state_t *)ifp->if_altqp;
  q_id = (*wfqp->hash_fnc)(flow, wfqp->nqueues);
  queue = &wfqp->queue[q_id];
  mp->m_nextpkt = NULL;
  byte = mp->m_pkthdr.len;

  /* enqueue the packet */
  if(queue->tail == NULL)
    queue->head = mp;
  else
    queue->tail->m_nextpkt = mp;
  queue->tail = mp;

  if(queue->next == NULL){
    /* this queue gets active. add the queue to the active list */
    if(wfqp->rrp == NULL){
      /* no queue in the active list */
      queue->next = queue->prev = queue;
      wfqp->rrp = queue;
      WFQ_ADDQUOTA(queue);
    }
    else{
      /* insert the queue at the tail of the active list */
      queue->prev = wfqp->rrp->prev;
      wfqp->rrp->prev->next = queue;
      wfqp->rrp->prev = queue;
      queue->next = wfqp->rrp;
      queue->quota = 0;
    }
  }
```

```
  /* if need, drop the packet at the head of the queue */
#ifdef RED
  if(random_drop(wfqp, queue))
#else /* RED */
  if(wfqp->hwm < wfqp->bytes + byte)
#endif /* RED */
    drop_queue = wfq_maxthqueue(wfqp);

  wfqp->bytes += byte;
#ifdef RED
  wfqp->count++;
#endif /* RED */
  queue->bytes += byte;

  while(drop_queue != NULL){
    if(drop_queue == queue)
      error = ENOBUFS;

    if((mp = drop_queue->head) == NULL){
      /* never happens */
      printf("NULL mp\n");
      break;
    }

    if((drop_queue->head = mp->m_nextpkt) == NULL)
      drop_queue->tail = NULL;
    mp->m_nextpkt = NULL;

    byte = mp->m_pkthdr.len;
    m_freem(mp);
    wfqp->bytes -= byte;
#ifdef RED
    wfqp->count = 0;
#endif /* RED */
    drop_queue->bytes -= byte;
    drop_queue->drop_packets++;
    drop_queue->drop_bytes += byte;

    drop_queue = (wfqp->hwm < wfqp->bytes) ? wfq_maxthqueue(wfqp) : NULL;
  }
```

```
  /*
   * call the driver's start routine.
   */
  ifp = wfqp->ifp;
  if(ifp->if_start && (ifp->if_flags & IFF_OACTIVE) == 0)
    (*ifp->if_start)(ifp);

  return error;
}


static wfq *
wfq_maxthqueue(wfqp)
     wfq_state_t *wfqp;
{
  int throughput, max_throughput = 0, delta;
  wfq *queue, *max_queue = NULL;
  struct timeval now;

  if((queue = wfqp->rrp) == NULL)
    /* never happens */
    return NULL;

  microtime(&now);
  do{
    if(queue->head != NULL && 1 < (delta = now.tv_sec - queue->idle) &&
       max_throughput <
       (throughput = queue->sent_bytes / delta * (100 / queue->weight))){
      max_queue = queue;
      max_throughput = throughput;
    }
  }while((queue = queue->next) != wfqp->rrp);

  if(max_queue == NULL)
    max_queue = wfq_maxqueue(wfqp);

  return max_queue;
}


static wfq *
```

```
wfq_maxqueue(wfqp)
     wfq_state_t *wfqp;
{
  int bytes, max_bytes = 0;
  wfq *queue, *max_queue = NULL;

  if((queue = wfqp->rrp) == NULL)
    /* never happens */
    return NULL;

  do{
    if(max_bytes < (bytes = queue->bytes * (100 / queue->weight))){
      /* "+1" prevents zero divide */
      max_queue = queue;
      max_bytes = bytes;
    }
  }while((queue = queue->next) != wfqp->rrp);


  return max_queue;
}


static struct mbuf *
wfq_ifdequeue(ifp, mode)
     struct ifnet *ifp;
     int mode;
{
  wfq_state_t *wfqp;
  wfq *queue;
  struct mbuf *mp;
  int byte;

  wfqp = (wfq_state_t *)ifp->if_altqp;

  if(mode == ALTDQ_FLUSH){
    wfq_flush(ifp);
    return NULL;
  }

  if(wfqp->bytes == 0 || (queue = wfqp->rrp) == NULL
```

```
     /* no packt in the queues */
#ifdef TOKEN_BUCKET
     || wfqp->token <= 0
     /* or no tokens for dequeue */
#endif /* TOKEN_BUCKET */
     )
    return NULL;


  while(1){
    if(queue->quota > 0){
      if(queue->bytes == 0){
        /* this queue no longer has packets.
           remove the queue from the active list. */
        struct timeval now;

        microtime(&now);
        queue->idle = now.tv_sec;

        if(queue->next == queue){
          /* no other active queue
             -- this case never happens in this algorithm. */
          queue->next = queue->prev = NULL;
          wfqp->rrp = NULL;
          return NULL;
        }
        else{
          queue->prev->next = queue->next;
          queue->next->prev = queue->prev;
          /* the round-robin pointer points to this queue, advance the rrp */
          wfqp->rrp = queue->next;
          queue->next = queue->prev = NULL;
          queue = wfqp->rrp;
          WFQ_ADDQUOTA(queue);
          continue;
        }
      }
      else{
        /* dequeue a packet from this queue */
        mp = queue->head;
```

```
        if(mode == ALTDQ_DEQUEUE){
          byte = mp->m_pkthdr.len;
          if((queue->head = mp->m_nextpkt) == NULL)
            queue->tail = NULL;
          mp->m_nextpkt = NULL;
          queue->quota -= byte;
          queue->bytes -= byte;
          queue->sent_packets++;
          queue->sent_bytes += byte;
          wfqp->bytes -= byte;
#ifdef TOKEN_BUCKET
          wfqp->token--;
#endif /* TOKEN_BUCKET */
        }
        return mp;

        /* if the queue gets empty by this dequeueing, the queue
           will be removed from the active list at the next round */
      }
    }
    else{
      /* advance the round-robin pointer */
      queue = wfqp->rrp = queue->next;
      WFQ_ADDQUOTA(queue);
    }
  }
}


#ifdef TOKEN_BUCKET
/*
 * this routine is called at every kernel clock (10ms),
 * and one token makes one paket dequeue
 *  cf.  10Mbit/s =  1.25Mbyte/s   =  12.5Kbyte/10ms
 *      155Mbit/s = 19.375Mbyte/s = 193.75Kbyte/10ms
 */
static void wfq_addtoken(wfqp)
     wfq_state_t *wfqp;
{
  static int count = -1, token = 1, rest_token, rest_tokens, bias, challenge;
```

```
if(count == -1){
  if(wfqp->token == wfqp->bucket){
    /* idling face */
    ;
  }
  else{
    /* initializing face (1)...ignore the fragment time */
    wfqp->token = wfqp->bucket;
    count     = 0;
  }
}
else if(count == 0){
  /* initializing face (2)...use the non-fragment 10 usec */
  wfqp->token = wfqp->bucket - wfqp->token;
  token       = wfqp->token;
  rest_token  = 0;
  rest_tokens = 0;
  count       = 1;
  bias        = 1;
  challenge   = 0;
}
else if(token == wfqp->token - rest_token){
  /* finishing face (return to the idling face) */
  wfqp->token = wfqp->bucket;
  count       = -1;
}
else{
  if((rest_token = wfqp->token) == 0 && bias <= ++challenge){
    /* increase face */
    token++;
    challenge = 0;
  }
  else{
    rest_tokens += wfqp->token;
    if(count++ % 16 == 0){
      if(rest_tokens / token != 0){
        /* decrease face */
        if(token != 1)
          token--;
        bias <<= 1;
        challenge = 0;
```

```
        }
        rest_tokens = 0;
      }
    }
    if(wfqp->bucket < (wfqp->token += token))
      wfqp->token = wfqp->bucket;
  }

  timeout((timeout_func_t)wfq_addtoken, (void *)wfqp, hz/100);
}
#endif /* TOKEN_BUCKET */


static int
wfq_flush(ifp)
     struct ifnet *ifp;
{
  struct mbuf *mp;

  while((mp = wfq_ifdequeue(ifp, ALTDQ_DEQUEUE)) != NULL)
    m_freem(mp);
  return 0;
}


static int
wfq_getqid(gqidp)
     struct wfq_getqid *gqidp;
{
  struct ifnet *ifp;
  wfq_state_t *wfqp;

  if((ifp = ifunit(gqidp->iface.wfq_ifacename)) == NULL)
    return ENXIO;

  wfqp = (wfq_state_t *)ifp->if_altqp;
  gqidp->qid = (*wfqp->hash_fnc)(&gqidp->flow, wfqp->nqueues);
  return 0;
}
```

```
static int
wfq_setweight(swp)
     struct wfq_setweight *swp;
{
  struct ifnet *ifp;
  wfq *queue;
  int old;

  if(swp->weight < 0){
    printf("set weight in natural number\n");
    return ENXIO;
  }

  if((ifp = ifunit(swp->iface.wfq_ifacename)) == NULL)
    return ENXIO;

  queue = &((wfq_state_t *)ifp->if_altqp)->queue[swp->qid];
  old = queue->weight;
  queue->weight = swp->weight;
  swp->weight = old;
  return 0;
}


static int
wfq_queuestate(qsp)
     struct wfq_queuestate *qsp;
{
  struct ifnet *ifp;
  wfq *queue;

  if((ifp = ifunit(qsp->iface.wfq_ifacename)) == NULL)
    return ENXIO;

  queue = &((wfq_state_t *)ifp->if_altqp)->queue[qsp->qid];

  qsp->weight           = queue->weight;
  qsp->bytes            = queue->bytes;
  qsp->sent_bytes       = queue->sent_bytes;
  qsp->sent_packets     = queue->sent_packets;
  qsp->drop_bytes       = queue->drop_bytes;
```

```
  qsp->drop_packets     = queue->drop_packets;

  return 0;
}




static int
wfq_wfqstate(wsp)
     struct wfq_wfqstate *wsp;
{
  struct ifnet *ifp;
  wfq_state_t *wfqp;

  if((ifp = ifunit(wsp->iface.wfq_ifacename)) == NULL)
    return ENXIO;

  wfqp = (wfq_state_t *)ifp->if_altqp;

  wsp->nqueues          = wfqp->nqueues;
  wsp->bytes            = wfqp->bytes;
  wsp->hash_policy      = wfqp->hash_policy;
  wsp->hwm              = wfqp->hwm;
#ifdef TOKEN_BUCKET
  wsp->bucket           = wfqp->bucket;
  wsp->token            = wfqp->token;
#endif /* TOKEN_BUCKET */
#ifdef RED
  wsp->count            = wfqp->count;
  wsp->avg              = wfqp->avg;
#endif /* RED */

  return 0;
}


static int
wfq_changehashfnc(chf, flag)
     struct wfq_changehashfnc *chf;
     int flag;
{
```

```
  struct ifnet *ifp;
  wfq_state_t *wfqp;
  int error = 0, old;

  if((ifp = ifunit(chf->iface.wfq_ifacename)) == NULL)
    return ENXIO;

  wfqp = (wfq_state_t *)ifp->if_altqp;
  old = wfqp->hash_policy;

  if(flag == CHANGE){
    switch(chf->hash_policy){
    case WFQ_HASH_DSTADDR:
      wfqp->hash_policy = WFQ_HASH_DSTADDR;
      wfqp->hash_fnc    = wfq_hashbydstaddr;
      break;
    case WFQ_HASH_SRCPORT:
      wfqp->hash_policy = WFQ_HASH_SRCPORT;
      wfqp->hash_fnc    = wfq_hashbysrcport;
      break;
    case WFQ_HASH_FULL:
      wfqp->hash_policy = WFQ_HASH_FULL;
      wfqp->hash_fnc    = wfq_hashbyfull;
      break;
    default:
      error = EINVAL;
    }
  }

  chf->hash_policy = old;

  return error;
}


int wfq_hashbyfull(flow, n)
     struct flowinfo *flow;
     int n;
{
  struct flowinfo_in *fp;
  u_long val, val2;
```

```
  if(flow == NULL || flow->fi_family != AF_INET)
    /* currently only AF_INET is supported, use qid 0. */
    return 0;

  fp = (struct flowinfo_in *)flow;
  val = fp->fi_dst.s_addr ^ fp->fi_src.s_addr;
  val = val ^ (val >> 8) ^ (val >> 16) ^ (val >> 24);
  val2 = fp->fi_dport ^ fp->fi_sport ^ fp->fi_proto;
  val2 = val2 ^ (val2 >> 8);
  val = val ^ val2;
  return (val % n);
}


int wfq_hashbydstaddr(flow, n)
     struct flowinfo *flow;
     int n;
{
  struct flowinfo_in *fp;
  u_long val;

  if(flow == NULL || flow->fi_family != AF_INET)
    /* currently only AF_INET is supported, use qid 0. */
    return 0;

  fp = (struct flowinfo_in *)flow;
  val = fp->fi_dst.s_addr;
  val = val ^ (val >> 8) ^ (val >> 16) ^ (val >> 24);

  return (val % n);

}

int wfq_hashbysrcport(flow, n)
     struct flowinfo *flow;
     int n;
{
  struct flowinfo_in *fp;
  u_long val;
```

```
  if(flow == NULL || flow->fi_family != AF_INET)
    /* currently only AF_INET is supported, use qid 0. */
    return 0;

  fp = (struct flowinfo_in *)flow;
  val = fp->fi_sport;
  val = val ^ (val >> 8);

  return (val % n);

}


#ifdef TOKEN_BUCKET
static int
wfq_bucketsize(bs)
     struct wfq_bucketsize *bs;
{
  struct ifnet *ifp;
  wfq_state_t *wfqp;

  if((ifp = ifunit(bs->iface.wfq_ifacename)) == NULL)
    return ENXIO;

  wfqp = (wfq_state_t *)ifp->if_altqp;

  if(bs->bucket > 0)
    wfqp->bucket = bs->bucket;

  return 0;
}
#endif /* TOKEN_BUCKET */


/*
 * wfq device interface
 */
#include <sys/ioctl.h>
#include <sys/conf.h>

d_open_t        wfqopen;
```

```
d_close_t       wfqclose;
d_ioctl_t       wfqioctl;

int
wfqopen(dev, flag, fmt, p)
     dev_t dev;
     int flag, fmt;
     struct proc *p;
{
  return 0;
}


int
wfqclose(dev, flag, fmt, p)
     dev_t dev;
     int flag, fmt;
     struct proc *p;
{
  struct ifnet *ifp;
  struct wfq_interface iface;
  wfq_state_t *wfqp;
  int s;

  s = splimp();
  while((wfqp = wfq_list) != NULL){
    ifp = wfqp->ifp;
    sprintf(iface.wfq_ifacename, "%s%d", ifp->if_name, ifp->if_unit);
    iface.wfq_ifacelen = strlen(iface.wfq_ifacename);
    wfq_ifdetach(&iface);
  }
  splx(s);
  return 0;
}


int
wfqioctl(dev, cmd, addr, flag, p)
     dev_t dev;
     int cmd;
     caddr_t addr;
```

```
     int flag;
     struct proc *p;
{
  int   error = 0;
  int s;

  /* check cmd for superuser only */
  switch(cmd){
  case WFQ_GET_QID:
  case WFQ_QUEUE_STATS:
  case WFQ_WFQ_STATE:
  case WFQ_CHECK_HASHFNC:
    break;
  default:
    if(error = suser(p->p_ucred, &p->p_acflag)){
#ifdef WFQ_DEBUG
      printf("wfqioctl()...not super user\n");
#endif
      return error;
    }
    break;
  }

  s = splimp();
  switch(cmd){
  case WFQ_ENABLE:
    error = wfq_setenable((struct wfq_interface *)addr, ENABLE);
    break;

  case WFQ_DISABLE:
    error = wfq_setenable((struct wfq_interface *)addr, DISABLE);
    break;

  case WFQ_IF_ATTACH:
    error = wfq_ifattach((struct wfq_interface *)addr);
    break;

  case WFQ_IF_DETACH:
    error = wfq_ifdetach((struct wfq_interface *)addr);
    break;
```

```
  case WFQ_GET_QID:
    error = wfq_getqid((struct wfq_getqid *)addr);
    break;

  case WFQ_SET_WEIGHT:
    error = wfq_setweight((struct wfq_setweight *)addr);
    break;

  case WFQ_QUEUE_STATS:
    error = wfq_queuestate((struct wfq_queuestate *)addr);
    break;

  case WFQ_WFQ_STATE:
    error = wfq_wfqstate((struct wfq_wfqstate *)addr);
    break;

  case WFQ_CHANGE_HASHFNC:
    error = wfq_changehashfnc((struct wfq_changehashfnc *)addr, CHANGE);
    break;

  case WFQ_CHECK_HASHFNC:
    error = wfq_changehashfnc((struct wfq_changehashfnc *)addr, CHECK);
    break;

#ifdef TOKEN_BUCKET
  case WFQ_BUCKET_SIZE:
    error = wfq_bucketsize((struct wfq_bucketsize *)addr);
    break;
#endif /* TOKEN_BUCKET */

  default:
    error = EINVAL;
    break;
  }
  splx(s);
  return error;
}


#ifdef RED
#include <sys/time.h>
```

```
static int
random_drop(wfqp, queue)
     wfq_state_t *wfqp;
     wfq *queue;
{
  int ret_val = 0;                      /* default : Never Drop */
  struct timeval now;
  static struct timeval last = {0, 0};  /* last packet sent time */

  microtime(&now);

  if(queue->head == queue->tail && 4 < now.tv_sec - queue->idle){
    /* over four seconds idle */
    queue->sent_bytes           = 0;
    queue->sent_packets         = 0;
    queue->drop_bytes           = 0;
    queue->drop_packets         = 0;
  }

  /*
   * calculate the average queue bytes
   */
  if(wfqp->bytes != 0)
    /* queue is not empty, so compute weighted average */
    wfqp->avg += (wfqp->bytes - wfqp->avg) >> RED_IWAIT;
  else{
    /* queue is zero, so use funny average */
    int msec;

    /* 10Mbit/s = 10Mbyte/8s = 1Kbyte/0.8ms */
    if((msec =  ((now.tv_sec - last.tv_sec) << 10)
        + ((now.tv_usec - last.tv_usec) >> 10)) == 0)
      ;
    else if(msec < (1 << 12)){
      int i, pow2 = (1 << 11);
      /* w[n]  = (1 - RED_WAIT)^(2^(11 - n)) << RED_AVG_SHIFT
         cf. (1 - RED_WAIT)^(2^12 = 4096) << RED_AVG_SHIFT = 0 */
      const int w[12] = {19, 138, 376, 621, 797, 904, 962, 992,
                         1008, 1016, 1020, 1022};
```

```
    for(i = 0; i < 12; i++){
      if(msec & pow2){
        if((wfqp->avg = (wfqp->avg * w[i]) >> RED_AVG_SHIFT) == 0)
          break;
        if((msec ^= pow2) == 0)
          break;
      }
      pow2 >>= 1;
    }
  }
  /* over four seconds idle */
  else
    wfqp->avg = 0;
}
last = now;

/*
 * decide to drop the packet or not
 */
/*
 *  (wfqp->avg)
 *
 *       | Never Drop | Maybe Drop |    Drop         Drop
 * MAXth +            +            +------------+
 *       | Never Drop | Maybe Drop   Maybe Drop |    Drop
 * MINth +            +------------+            +
 *       | Never Drop   Never Drop | Maybe Drop |    Drop
 *       +------------+------------+------------+------------- (wfqp->bytes)
 *      0            MINth        MAXth        HWM
 */

if(wfqp->hwm < wfqp->bytes ||
   RED_MAXth < wfqp->avg && RED_MAXth < wfqp->bytes){
  /* Drop */
  ret_val = 1;
}

else if(RED_MAXth < wfqp->bytes ||
        RED_MINth < wfqp->bytes && RED_MINth < wfqp->avg){
  /* Maybe Drop */
  int spb, tmp;
```

```
    /* pb = C1 * wfqp->avg - C2
       => Spb = SC1 * wfqp->avg - SC2 */
    if((spb = (wfqp->avg >> RED_SIC1) - RED_SC2) > RED_SMAXp)
      spb = RED_SMAXp;

    /* R < (pb / (1 - count * pb))...(S1 - count * Spb) > 0
       => R * (1 - count * pb) < pb => R * (S1 - count * Spb) < Spb */
    if((tmp = RED_SMAXr - wfqp->count * spb) < 0 ||
       (((random() % RED_SMAXr) * tmp) >> RED_PB_SHIFT) < spb){
      wfqp->count = -1;
      ret_val = 2;
    }
  }

  return ret_val;
}
#endif /* RED */

#endif /* WFQ */
```