代替キューイング機構によるトラフィック管理の実現

長 健二朗

キューイングはトラフィック管理のための重要な要素技術であるが，従来のUNIXシステムで利用できるキューイングは単純なFIFOキューイングのみであった．本稿では，現在のUNIXシステムに種々のキューイング方式を実装する場合の問題点を指摘し，代替キューイング機構ALTQ(Alternate Queueing)を提案する．また，実際にALTQ上にCBQ(Class-Based Queueing)，WFQ(Weighted-Fair Queueing)，RED(Random Early Detection)等複数のキューイング方式を実装して設計の検証を行ない，PCルータのトラフィック管理能力の評価を行なった．

1 はじめに

インターネットにとってトラフィック管理技術の確立は重要な課題である．トラフィック管理は様々なメカニズムとそれらを運用するポリシーによって構成され，各メカニズムは実装される階層に応じた粒度でトラフィックを制御する．キューイングはパケット粒度のトラフィック制御を行なうメカニズムであり，パケットスケジューリングとバッファ管理という2つの機能を持つ．キューイングが特に有効なのは，バースト的なトラフィックによってパケットが溜るボトルネックリンクの入口である．キューイングを工夫することによってフロー間の公平性の実現や干渉の防止が可能であり，上位のフロー制御メカニズムと連動して輻輳回避にも使われる．また，キューイングによって使用帯域，遅延，ジッタ，パケット廃棄を制御することが可能なため，リアルタイム性を保証するサービスを提供することもできる．

現在までに研究レベルではさまざまキューイング方式が提案されている[14][6][5]が，ほとんどが理論的な研究やシュミレーションをもとにしたものである．実際に運用されているシステムのほとんどでは単なるテイル・ドロップのFIFOキューしか実装されていない．また，それに代わるキューイング機構を実装しようにも，そのためのフレームワークがないため実装が困難であり，一般に利用できる実装が存在しないのが現状である．

一方，プロセッサの高速化，IO性能の向上，高速ネットワークカードの登場と，それらの価格低下に伴い，高度なキューイング処理をPCベースのルータに組み込むことは魅力的な選択となってきた．

そこで，BSD系のUNIXに代替キューイングを実装するための汎用フレームワークとしてALTQ(Alternate Queueing)という汎用キューイング・インターフェイスを設計実装した．ALTQが対象として考えるキューイング機構は，なんらかのパケット・スケジューリングを行なって帯域割り当てや遅延の制御を実現するものである．パケット廃棄機構，複数のキューを持つ機構，非ワークコンサービング型など様々なキューイング機構をサポートできるよう考慮している．

また実際にそのフレームワーク上にCBQ[6]，RED

Traffic Management by Alternate Queueing.
Kenjiro Cho, (株)ソニーコンピュータサイエンス研究所, Sony Computer Science Laboratories, Inc.
コンピュータソフトウェア, Vol.16, No.4(1999), pp.10–22.
[論文]1998年6月30日受付.

[5]，WFQ[3][15][12]の実装を行なって設計を検証をした．CBQはRSVP[22]と連動して，動的な資源予約による帯域割り当てにも利用できる．

ALTQによって異なるキューイング方式がカーネル内部のキューイング関連コード，特にドライバ部分の変更を共有できるようになる．また，種々のドライバで動作するための条件も明確となるため，カーネル実装の詳細を知らなくても容易に新しいキューイング方式を実装できるようになる．

本プロジェクトの目的は，

1. 種々のキューイング機構実装のためのフレームワークを提供する．
2. 能動的キュー管理や，ポリシにしたがった帯域共有を行なうネットワーク運用のテストベッドを提供する．
3. RSVPのためのトラフィックコントロール・カーネルを提供する．

という3つの側面を持つ．

本稿では代替キューイング機構ALTQの設計と，FreeBSD上のプロトタイプが動作するPCルータのトラフィック制御性能評価を報告する．

2　従来のキューイング抽象化の問題点

BSD系UNIXでは送出パケットのキューイングは抽象化されたインターフェイス構造体ifnetを介して行なわれる．キューイングにはifqueueというFIFOキュー構造体が使われ，IF_ENQUEUEとIF_DEQUEUEというマクロで操作される．キュー操作を行なうのは，インターフェイス構造体内のif_outputとif_startに登録された関数であり，リンクタイプ毎に用意されるif_output関数がキューへの格納を，デバイス毎に用意されるif_start関数がキューからの取り出しを行なう．

代替キューイングを導入する際に問題となるのが，キューに対する操作が単にパケットの格納，取り出しだけでないことである．具体的には，

- キューが飽和するとパケット廃棄する
- キューの先頭にあるパケットを覗き見る
- パケット送出のためデバイスドライバを呼び出す

等の操作があり，これらの操作は現在のFIFOキュー構造を想定して実装されている．

以下に，一般的なif_output関数がキューへの格納を行なう部分のコードを示す．

```
s = splimp();
if (IF_QFULL(&ifp->if_snd)) {
      IF_DROP(&ifp->if_snd);
      splx(s);
      m_freem(m);
      return(ENOBUFS);
}
IF_ENQUEUE(&ifp->if_snd , m);
if ((ifp->if_flags & IFF_OACTIVE) == 0)
      (*ifp->if_start)(ifp);
splx(s);
```

この一連のコードはキューに関連した3つの操作を行なっている．

1. IF_QFULLマクロによってキューの飽和をチェックし，飽和している場合はパケットを廃棄する．
2. IF_ENQUEUEマクロによってパケットをキューに格納する．
3. デバイスドライバがビジーでなければ，パケット送出のためにドライバを呼び出す．

現状のパケット廃棄はキューが飽和した場合には，最後に格納しようとしたパケットを廃棄することを前提としている．しかしながら，例えばランダムな廃棄をするには，先にパケットをキューに格納した後，キュー内のすべてのパケットから廃棄するパケットを選択することが必要である．つまり，パケット廃棄の判断と廃棄するパケットの選択はキューイング方式に依存する．

また，キュー内にパケットが存在しても送出が行なわれるとは限らない非ワークコンサービング型のキューでは，パケット格納時点でドライバを呼び出すかわりに送出予定タイミングで呼び出す必要がある．また，非ワークコンサービング型のキューでは，パケットの取りだし操作を繰り返してもキューを空にすることができないため，キューを空にする操作も必要となる．

つまり，キューへのパケット格納操作は，パケット廃棄，デバイスドライバの呼びだしを伴うが，いずれの動作もキューイング方式に依存するため，これらを統合して規定するキューイング・インターフェイスを作る必要がある．

また，いくつかのデバイスドライバはキューの先頭を覗き見る操作を行なって，キュー内にパケットが存在するかどうか，あるいは，パケットを送出するためのバッファスペースがあるかどうかを確認する．しかし，その手続きが規定されていないため実装によって手順が異なっている．キューイング方式によっては，キューが複数存在する場合もあるし，また，キューの先頭にあるパケットがかならずしも次に取り出すべきパケットであるとは限らない．したがって，次に送出されるパケットをチェックするための操作も規定する必要がある．覗き見操作の是非は議論の余地があるが，既存のドライバの変更作業を考慮するとサポートするのが賢明であると判断した．

また，BSD 系 UNIX では IF_PREPEND というキューの先頭にパケットを格納するマクロがあるが，この操作もキューが複数存在すると意味をなさない．

そこで我々はキューイングの抽象化にために，ENQUEUE，DEQUEUE，PEEK，FLUSH の 4 つの操作を規定した．ENQUEUE 操作はパケット格納以外にもパケット廃棄とドライバの呼びだしを行なう．DEQUEUE 操作はパケットの取りだしを行なう．PEEK 操作は次に取り出される予定のパケットを覗き見る．FLUSH 操作はキュー内部のすべてのパケットを廃棄してキューを空にする．なお，PREPEND に相当する操作は存在しない．

3 ALTQ: 代替キューイング機構

3.1 設 計

代替キューイング機構の実装は，既存のコードの変更を最小限にする方針をとって検討したが，前記のように現状のキュー操作は十分な抽象化がなされていないため，if_output 関数と if_start 関数の変更が避けられないことが明らかとなった．

なかでも，if_start 関数の変更はデバイスドライバの変更を意味するが，既存のすべてのデバイスドライバを変更するのは容易でない．そこで，代替キューイングをサポートするデバイスドライバとサポートしないデバイスドライバが共存できるようにすることによって，必要なドライバへの変更だけですむようにし，段階的なドライバサポートを可能にした．

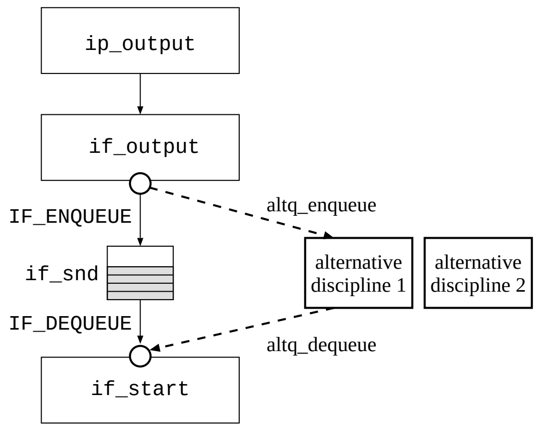

図 1 ALTQ の構成

図 1に ALTQ の構成を示す．代替キューイングはインターフェイスごとの出力キューに対して設定され，代替キューイングが使われていない状態では従来の FIFO キューイングが行なわれる．代替キューイングが使われる場合，代替キューイングは複数のキューイング方式のうちのひとつを選ぶスイッチとして機能する．

このように，従来の FIFO キューイングと代替キューイングを共存させて，動的に切替可能とすることによって，設定の誤り等で代替キューイングに問題が発生しても従来のキューイングに切替えて使うことができる．さらに，運用上の利便や信頼性が向上し，デバッグもしやすい．

3.2 実 装

以下に，ALTQ の実装による変更点の概要を示す．ifnet 構造体に代替キューイングをサポートするためのフィールドを追加し，代替キューイング関係のコードはこれらのフィールドを参照して動作するようにした．

if_output 関数の変更は 2 箇所．ひとつめは，パケットからクラス分けに必要となるフロー情報を抽出する部分である．この情報はキューへの格納操作に渡される．フロー情報の抽出は，ネットワーク層やキューイング関数の中に実装することも可能であるが，ネットワーク層で行なうとその情報を if_output 関数に渡すためにインターフェイスの変更が必要となりカーネル内の他の部分への影響が大きい．また，キューイング関数で行なうと

多様なリンクヘッダを扱わなければならなくなる．そこで，if_output 関数からネットワーク層のヘッダにアクセスするための情報をキューへの格納操作に渡すことにした．

ふたつめは，実際のキューイングを行なう部分である． ALTQ_IS_ON マクロを使って ifnet 構造体内のフラグをチェックし，代替キューイングが使用中であるかどうか判断する．代替キューイングが使用されていない場合は従来のコードが実行されるが，使用中であれば ifnet 構造体に登録された ALTQ の格納関数を呼び出す．格納関数はパケット廃棄とデバイスドライバの起動も行なう．

if_start 関数も同様に， ALTQ_IS_ON マクロを使って代替キューイングが使用中であるかどうか判断し，使用中であれば ifnet 構造体に登録された ALTQ の取り出し関数を呼び出す．

覗き見操作は，取り出し関数を PEEK モードで呼ぶことによって行なう．覗き見操作の直後に取り出し操作を実行すれば同じパケットが返されることが保証される．

if_start 関数に上記変更を行なったデバイスドライバは，ifnet 構造体内のフラグフィールドに ALTQF_READY を立てることによって代替キューイングをサポートすることが識別される．

代替キューを使用するには，キューデバイス (例えば， /dev/cbq) をオープンした後， ioctl を使って代替キューイングをインターフェイスにアタッチしイネーブルする手順をとる．代替キューイングをディセーブルすると従来の FIFO キューイングに戻る．このように比較的簡単な変更でキューイング方式の動的な切替えを実現している．

4 キューイング方式

ALTQ 自体はあくまでフレームワークであり，その実用性を示すためには実際にキューイング方式をその上に実装して検証を行なう必要がある．

4.1 キューイング方式の概要

まず， ALTQ 上に実装されたキューイング方式について概要を示す．なお，各方式の詳細，シュミレーション結果とその解析等はそれぞれの文献を参照されたい．

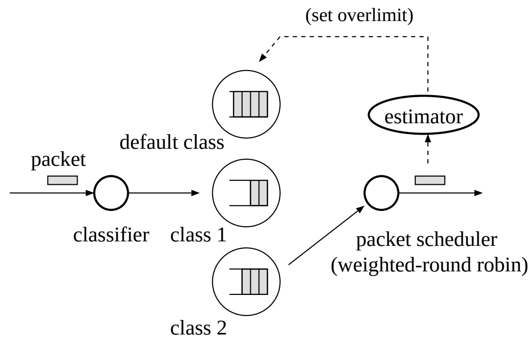

図 2 CBQ の構成

4.1.1 CBQ (Class-Based Queueing)

CBQ [6]は， Jacobson と Floyd によって提案され，解析，シュミレーションが行なわれてきた． CBQ のデザインは実装面を強く考慮してあり，比較的小さいオーバヘッドで機能が実現できる． Sun と LBL が共同で Solaris への実装を行ない，コードを公開している [19] [21]．今回， ALTQ への実装はこのコードをもとに移植を行なった．

CBQ はその名のとおりクラス構造を基本としている．独立したキューをもつクラスを階層的に構成し，各クラスは上位のクラスから帯域を借りることが可能で，それによってリンク共有を実現できる．つまり，上位のクラスの使用帯域に余裕がある間は自分のクラスの割り当てを越えた帯域使用ができるが，全体の使用帯域が足りなくなってくると自分のクラスの割り当て分まで使用量が制限される．割り当てられた帯域は保証されるので， QoS 保証を必要とするサービスに利用できる．

図 2は CBQ の構成要素を示す．クラス分け機構 (classifier) は，入力パケットをパケットフィルタを使ってクラス分けし，対応するクラスのキューに割り当てる．パケットスケジューラは，次にどのクラスからパケットを送出するかを決定する．スケジューリング方式はプライオリティと重み付きラウンドロビンを組み合わせた方式である．推量機構 (estimator) は各クラスのパケットの送出間隔を測定し，その重み付き平均値を管理することによって各クラスの使用帯域を計測する．クラスの使用帯域が設定された値を越えた場合は使用量超過が検出される．使用量が超過しても借り入れが可能な間

は継続してパケット送出ができるが，借り入れができなくなると送出が制限される．

4.1.2 RED (Random Early Detection)

RED[5]も Floyd と Jacobson によって提案された方式で，キュー長が閾値を越えると，キュー長に応じた確率でパケット廃棄または輻輳情報のマーキングを行なう輻輳回避のメカニズムである．RED はバッファ管理のメカニズムとして捉えることもでき，他のパケットスケジューラに組み込むことができる．

我々の実装は NS シュミレータ[11]に含まれる RED モジュールをもとにしている．輻輳情報伝達手段である ECN(Explicit Congenstion Notification)[4]を使う拡張も取り込んでいる．また，RED と ECN は CBQ に組み込むこともでき，クラスごとに RED と ECN の使用を指定できる．

4.1.3 WFQ (Weighted-Fair Queueing)

WFQ[14][3][10]はもっともよく知られ，かつ，研究されてきたキューイング方式である．WFQ は広義には，フローごとにキューを割り当て，公平な帯域使用を実現する方式である．また，フローごとに異なる帯域使用を供給できるようにキューに重みを付けることもできる．狭義には，WFQ は Demers らが提案した特定のアルゴリズムを指す[3]．Parekh はこの方式を使ってエンド・エンドの最大遅延が保証できることを示した[15]．我々の実装はこの意味では厳密に WFQ でなく，むしろ SFQ(Stochastic Fairness Queueing)[12]というアルゴリズムに近い．SFQ では固定数のキューにハッシュ関数を使ってフローをマップする．WFQ と違って，異なるフローが同じキューにマップされ得るので遅延の保証を行なうことはできない．

4.1.4 FIFOQ (First-In First-Out Queueing)

FIFOQ は単なるテイル・ドロップの FIFO キューで，新たにキューイング方式を実装する場合にテンプレートとして使えるようになっている．

4.2 キューイング方式の実装

以下に ALTQ にキューイング方式を実装する際に注意が必要な点に関して述べる．ここでは，特定のキューイング方式の実装に依存したものは含まない．

4.2.1 キューイング方式の追加

キューイング方式は ALTQ 上に実装される前に，シュミレーション等で方式自体の評価が済んでいることを想定している．例えば，NS シュミレータ[11]は異なるキューイング方式が利用できる数少ないシュミレータで，RED や CBQ のモジュールも含まれており，ネットワーク研究者の間で広く使われている．

ALTQ に新たにキューイング方式を追加する場合，FIFOQ の実装がテンプレートとして供給されているので，実装者はキュー操作の部分の実装に専念できる．他の作業は，ALTQ のデバイステーブルにエントリを追加して，open，close，ioctl 関数を作るだけである．必要な ioctl 関数には，attach，detach，enable，disable がある．これらの作成が終ると，追加したキューイング方式は ALTQ がサポートするすべてのデバイスで利用可能となる．

4.2.2 経験則アルゴリズムの使用

Floyd は帯域共有の借り入れアルゴリズムとしていくつかの方式を提案している[6]．フォーマル帯域共有方式は理想的な共有を定義するが，この方式は常に全てのクラスの状態をチェックするためオーバーヘッドが大きい．トップレベル帯域共有方式はフォーマル帯域共有方式を近似する経験則を実装し，公開されているコードはこの方式を使っている．

今回の実装でもこのトップレベル帯域共有方式を用いたが，オリジナルのアルゴリズムはシュミレーション用に想定された特定のクラス構成とトラフィックパターンに合わせて設計されているため，実際のトラフィックに適用するには細かい修正が数多く必要となった．

経験則アルゴリズムは効率のよい実装のためには欠かせないものであるが，理論的なアルゴリズムに比べて有効範囲が不明確になりがちで，十分に注意して利用する必要がある．また，利用する際にはその経験則の特性を十分検討しておくことが，実装時の問題回避に継る．

4.2.3 割り込みの制御

取り出し操作はデバイス割り込みから呼び出されるため，取り出し操作と共有されるデータ構造を操作する際は，割り込みを禁止して競合問題を回避しなければならない．また，if_start は割り込み禁止状態で実行されるので，格納操作はすでに割り込み禁止状態を仮定してよ

い．

4.2.4 整数演算の精度

リンクの帯域は9600bpsのモデムから155MbpsのATMまで広い範囲を持つため，32ビット整数はちょっとした演算で容易にオーバーフローやアンダーフローを起こすことになる．

シュミレータを使っている時は64ビットの倍精度浮動小数点演算が利用でき，また精度誤差を避けるためにも浮動小数を使うことが望ましい．ところが，一般にカーネル内部では浮動小数は利用できないので，アルゴリズムをカーネルに実装する場合，整数演算や固定小数点演算に変換する必要がある．

我々のREDの実装では，NSシュミレータの浮動小数点演算を使ったアルゴリズムを固定小数点演算に変換した．また，できるだけ演算をユーザ空間で浮動小数で行ない，結果をカーネルに渡す手法が薦められる．CBQではこの手法を取っている．将来64ビット整数が一般的になり演算が効率よくなればこの問題は低減されるであろう．

4.2.5 パケット送出の完了

キューイング方式によっては，パケットの送出完了を知ることが必要になる．CBQもそのようなキューイング方式のひとつである．BSD系UNIXにはif_doneというコールバック・エントリがキューが空になった事を通知する目的で定義されている[13]が，この仕組みをサポートするドライバは存在しない．さらに，キューイング方式はキューが空になった時点ではなく，ひとつのパケットが送出完了した時点を知る必要がある．

また，別のアプローチとしてメモリ・バッファの解放時のコールバックを利用する方法もあり，SolarisのCBQ実装はこの方法を使っている．しかし，この場合もコールバックが発生するのはDMAが終了してメモリが解放された時点であり，送出が終了した時点ではない．通常，パケットを送出するには，デバイスにDMAするのに比べて長い時間を必要とする．

我々のCBQの実装ではコールバックは使わずに，パケットをキューに入れる際にパケットサイズから転送終了時間を推測する方法を取っている．これは必ずしも好ましい解決方法ではないが，ドライバに非依存であり，CBQの動作には十分な精度が得られている．

4.2.6 時間測定

キューイング方式によってはパケット・スケジューリングのために時間を測定する必要がでるが，得られる時間の精度とオーバーヘッドに注意を払う必要がある．BSD系UNIXではmicrotime関数を使って，1970年からの時間のオフセットをマイクロ秒の単位で得ることができる．また，インテル・ペンティアム系のプロセッサは，CPUクロックで駆動される64ビットのタイムスタンプ・カウンタを持ち，このカウンタ値は1命令で読み出すことができる．CPUクロックが200MHzの場合，その精度は5ナノ秒である．ほとんどのPC UNIXでは，このカウンタが利用できる場合はこのカウンタを使ってmicrotime関数を実現している．

いっぽう，microtime関数を使う代わりに直接カウンタ値を読み出すことも可能で，より高速により高精度の値が得られる．microtime関数で得られる値は時間調整を受ける可能性があるのに対して，このカウンタ値は修正されることがないので時間計測には都合がよい．しかし，問題はプロセッサごとにクロックが異なるので，カウンタ値を正規化しないと使えないことである．正規化するには乗算と除算を使った演算が必要であるうえ，演算の結果下位ビットに丸め誤差が生じることが避けられない．したがって，このカウンタを利用する場合は精度に注意が必要である．microtime関数の場合はマイクロ秒の精度が得られればよいため，1回の乗算で必要な精度で正規化するように工夫されている．なお，microtime関数の実行には200MHzのPentiumProで約450ナノ秒かかっている．

4.2.7 タイムアウト処理

シミュレータではタイムアウト処理のタイマーはほぼ無限の精度があるが，カーネル内部ではタイムアウト処理はインターバル・タイマーによって実現されていて，限られた粒度しか持たない．

CBQの性能面で一番問題になったのがタイマーの粒度に起因する帯域制限性能である．CBQは各クラスのパケット送出間隔を測定して制限を越えているクラスをサスペンドすることによって帯域制限をかける．サスペンドされたクラスをリジュームするトリガには，タイムアウト処理とパケットの入出力のイベントが併用される

が，パケットイベントがない場合はタイムアウト処理に頼ることになる．

ほとんどの UNIX システムのデフォルトのタイマーの粒度は 10 ミリ秒である．CBQ では 1 ティックの最短期間が保証されないので最小で 2 ティックのタイマを使うため，粒度は 20 ミリ秒になる．

各クラスは一度に最大 maxburst 変数で指定されるパケット数を送ることが可能である．20 ミリ秒のサイクルの始めに maxburst 個のパケットを送ったクラスはサスペンドされ，次のタイマーイベントまでリジュームされない．この状態が続くと転送レートは，

$$rate = packetsize \times maxburst \times 8 \div 0.02$$

であり，maxburst がデフォルト値の 16，パケットサイズが MTU と仮定すると，MTU が Ethernet の 1500 バイトで 9.6Mbps，MTU が ATM の 9180 バイトで 58.8Mbps にしか達しないことになる．特にこの問題は，高速でかつ MTU が小さい 100baseT の場合に顕著となり，帯域の約 1/10 しか転送できない計算となる．

ただし，タイマに頼って帯域制限を行なうのは最悪ケースであり，実際には他のパケット入出力等のトリガがあるため，より細かい粒度で帯域制御が行なわれる．しかしながら，ベンチマークを取る場合などは他のトラフィックが無い状態で行なうことになるため注意が必要である．また，TCP では定常状態で 2 パケットに 1 度 ACK が返ってイベントトリガになるので，UDP に比較して良好な精度が得られる．

4.2.8 デバイスバッファ

ネットワーク・デバイスによっては大きなバッファを持つものがあるが，大きい送信バッファはキューイングの効果を低減してしまう．大きい受信バッファはオーバーフローを避けるのに役立つが，大きい送信バッファがあると，特に遅いリンクの場合，折角スケジューリングされたパケットがバッファに溜って出ていかなくなる．例えば，128Kbps のリンクに 16K バイトのバッファがあると，このバッファには 1 秒分のパケットを貯めることが可能である．この問題は FIFO キューを使っている限り見えてこないが，優先制御を行なうキューイング方式を使い始めた途端に顕著になる．したがって，送信バッファサイズは帯域を使い切るために必要な最小値に設定することが望ましい．

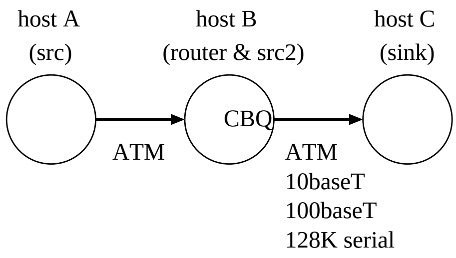

図 3 テストシステム構成

5 性能評価

本章では実装したキューイング方式の性能評価を報告する．性能評価の目的は，現状の PC-UNIX が実用レベルでどの程度の性能を達成できるかを示すこと，また，4.2節で述べているようなシステム実装上の制限がどのように実際の性能に現れてくるかを示すことである．したがって，各種キューイング方式の比較をすることや，特定のキューイング方式のメカニズム自体を評価するものではない．なお，キューイングによるトラフィックの管理はボトルネック・リンクにおいてより有効であるため，必ずしも高速なネットワークに対応した性能が要求されるわけではないことに留意されたい．

測定は主に CBQ を用いて行なっているが，これは CBQ がクラス分け機構を持ち，プライオリティと重み付きラウンドロビンの組み合わせスケジューリングを使う非ワークコンサービングなキューであるため，実装した方式の中で最も複雑で興味深い特性を持つからである．

測定は 3 台の PentiumPro マシン (200MHz，440FX) を用いて行なった．OS は FreeBSD-2.2.5 で，altq-1.0.1 を使っている．図 3に示すように，ホスト A をソース，ホスト B をルータ，ホスト C をデスティネーションとし，netperf[8]ベンチマークを用いた測定を行なった．

ホスト A とホスト B 間は ATM リンクで接続，ホスト B とホスト C 間は，ATM，10baseT，100baseT，128Kbps シリアルを切替えて使用した．CBQ はホスト B 上のホスト C 側のインターフェイスでのみ動作させている．なお，10baseT の場合はハブ

表1 CBQ のスループット

Link Type	orig . FIFO (Mbps)	CBQ (Mbps)	overhead (%)
ATM	132.98	132.77	0.16
10baseT	6.52	6.45	1.07
100baseT	93.11	92.74	0.40
loopback			
MTU 16384	366.20	334.77	8.58
MTU 9180	337.96	314.04	7.08
MTU 1500	239.21	185.07	22.63

を介した接続，100baseT の場合はクロスケーブルによる接続で，100baseT のみフルデュープレクスモードを設定している．

全体を通じてスループットの測定には TCP による測定を用いている．これは，UDP による測定ではパケット廃棄で CPU サイクルが消費されるのに対し，TCP は有効帯域に迅速に適応するので高負荷時のパケットフォワーディング性能の測定がしやすいからである．

しかしながら，TCP を使う場合はエンド・エンドの遅延やキューに溜るパケット数を考慮してウインドウサイズを設定しないと正しい測定ができないので注意が必要である．

5.1 オーバーヘッド

CBQ によるオーバーヘッドは 4 つの要因からなる．

1. パケットからフロー情報を抜き出す
2. フロー情報をもとに対応するクラスを見つける
3. パケットのスケジューリング
4. パケット送出したクラスの帯域使用率を計算し，クラスの状態を更新する

オーバーヘッドに影響を与える要因には，クラス階層の構成，全体のクラス数，パケットを送出するクラス数，クラスのプライオリティ，送出パケット数や送出分布など多様なものがあり，以下の評価がすべての項目を網羅しているわけではない．

5.1.1 スループット

表 1は各リンクタイプにおいてホスト A からホスト C に対して TCP でデータを送った場合のスループットを FIFO と CBQ で比較している．CBQ の最小オーバーヘッドを示すため，3 つのクラスを持つ小さいクラス構成を用いて，他のトラフィックがない状態で測定している．

表2 CBQ の遅延

Link Type	queue type	request/ response (bytes)	trans. per sec	calc'd RTT (usec)	diff (usec)
ATM	FIFO	1，1	2875.20	347.8	
	CBQ		2792.87	358.1	10.3
	FIFO	64,64	2367.90	422.3	
	CBQ		2306.93	433.5	11.2
	FIFO	1024,64	1581.16	632.4	
	CBQ		1552.03	644.3	11.9
	FIFO	8192,64	434.61	2300.9	
	CBQ		432.64	2311.1	10.2
10baseT	FIFO	1,1	2322.40	430.6	
	CBQ		2268.17	440.9	10.3
	FIFO	64，64	1813.52	551.4	
	CBQ		1784.32	560.4	9.0
	FIFO	1024,64	697.97	1432.7	
	CBQ		692.76	1443.5	10.8

いずれのリンクタイプの場合も CBQ によるオーバーヘッドはほとんど観測されない．これは CBQ によるオーバーヘッドが前のパケットの送出時間にオーバーラップするためスループットへの影響がでないからである．

なお，参考までにローカルループをもちいた測定結果も載せてあるが，この場合 MTU サイズに応じて 7% から 23% 程度のオーバーヘッドとなっている．この値が CPU の処理能力の限界と，CPU サイクルに対する CBQ のオーバーヘッドを示している．送受信双方向の負荷のかかるローカルループで 300Mbps を越える処理能力を示すことは，PC ベースのルータでも 100Mbps クラスのインターフェイスを複数持つに十分な処理能力があることを示している．しかも，実際のネットワークインターフェイスでは DMA が利用できるので CPU 負荷はずっと少なくなる．

5.1.2 遅 延

表 2は ATM と 10baseT において，UDP パケットを使ったリクエスト・リプライによるトランザクションが秒あたり何回実行できるかを，パケットサイズを変えて測定している．また，測定結果から算出したパケットあたりのラウンドトリップタイム，および FIFO と CBQ の差が示してある．ここでも，CBQ には 3 つのクラスが設定されており，他のトラフィックがない状態で測定している．測定結果から，リンクタイプやパケッ

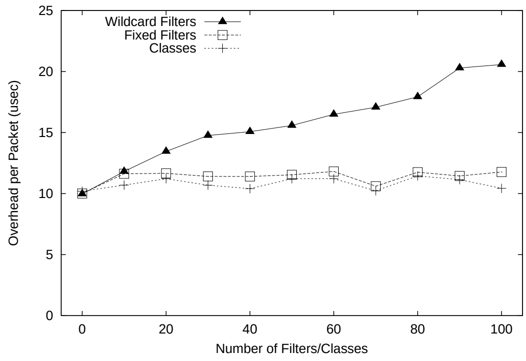

図 4　フィルタ数とクラス数の影響

トサイズにかかわらず，パケットあたりの CBQ のオーバーヘッドは 10 マイクロ秒程度であることがわかる．

5.1.3　遅延とスケーラビリティ

CBQ は全体のクラス数が比較的少ないことを想定して設計されている．通常の構成ではクラス数は２０程度以内に収まるが，それでも，クラス数が増大した場合の性能劣化を調査することは重要である．スケールが上がっていくと，例えばメモリ消費などが絡んで様々な問題が考えられるが，ここではいくつかの簡単な場合に限ってデータを示す．

図 4はフィルタ数，クラス数を 100 まで増加させた場合の遅延変化を示す．値は ATM 上で 64 バイトのリクエスト・リプライを使った場合のラウンドトリップタイムの FIFO との差分である．

図中の “Wildcard Filters” と “Fixed Filters” は 2 種類の異なるタイプのフィルタを示す．クラス分け機構はパケットをクラス分けするのに，パケットヘッダの情報（例えば，IP アドレスとポート番号）とフィルタの値を比較して，マッチするかどうかを調べる．我々の実装ではこの操作を低減するため，フィルタはデスティネーション・アドレスでハッシュ管理されている．しかしながら，フィルタがデスティネーションを指定していない場合はハッシュリストとは別のワイルドカードリストに置かれる．クラス分け機構は，まずハッシュリストを試みマッチするフィルタが見つからなければワイルドカードリストをチェックする．したがって，本実装ではパケットあたりのオーバーヘッドはワイルドカード・フィルタの数に比例して増大することになる．なお，クラス分け機構をより効率的に実装することは可能で，例えば DAG(directed acyclic graph) を使う手法が知られている [1]．

表 3　各キューイング方式の遅延

	FIFO	FIFOQ	RED	WFQ	CBQ	CBQ +RED
(usec)	0.0	0.14	1.62	1.95	10.72	11.97

いっぽう，クラス数は直接スケジューラの性能に影響しない．これは，スケジューラがあるクラスを選ぼうとしたときに，このクラスの使用帯域が設定値以下である限り，他のクラスの状態をチェックせずにスケジュールすることができるからである．しかしながら，もしこのクラスが設定値を越えて帯域を使用している場合は，他に先にスケジュールしなければいけないクラスがあるかどうか調べる必要がでてくる．ただし，設定値を越えるクラス数の最大値はリンクの帯域と最小パケットサイズによって上限があるので，オーバーヘッドの増大もそこで押えられることになる．

帯域使用設定値を越えるクラスが存在する場合，他のクラスのトラフィックの影響を低減する CBQ が FIFO より有利なのは明らかである．今回のテスト環境では CBQ のオーバヘッドと，ソースおよびデスティネーションホストの負荷等の CBQ 以外の要因を区別するのが困難であるため比較データが取れなかった．もし，測定ができたとしても，CBQ を使った場合の RTT の主要因は伝送遅延とデバイス内のバッファによる遅延であり，さらに，ヘッドオブライン・ブロッキングによって測定値が揺れることが予想できる．これらの要因に比べると CBQ によるオーバーヘッド自体はずっと小さいはずである．

5.1.4　キューイング方式による遅延の比較

遅延の測定は他のキューイング方式のオーバーヘッドの測定にも利用できる．表 3は実装したキューイング方式のパケットあたりの遅延を FIFO からの差分で表したものである．測定は ATM 上で 64 バイトのリクエスト・リプライを使っている．オリジナルの FIFO と ALTQ の FIFOQ との違いは，FIFO ではパケットの格納，取り出しがマクロであるのに対して FIFOQ では関数呼びだしになっていることである．

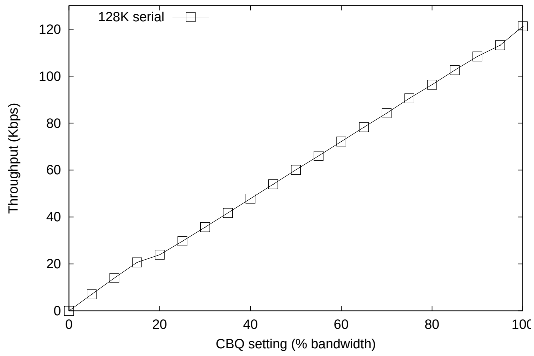

図 5 シリアルリンクにおける **CBQ** の帯域制限

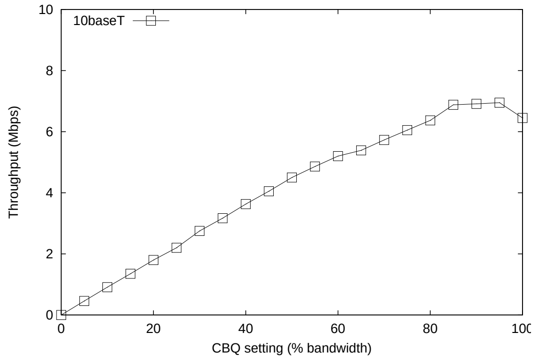

図 6 **10baseT** における **CBQ** の帯域制限

5.1.5 遅延の影響

ネットワーク技術者の多くは，パケットのフォワーディング・パスに処理を追加し遅延が増加することに抵抗がある．しかしながら，遅延の増加を議論する際にはキュー待ち遅延も考慮する必要がある．例えば，1K バイトのパケットを 10Mbps のリンクに送り出すには 800 マイクロ秒を要する．もし，キュー内部にすでに 2 個のパケットがあると，新たに到着したパケットは 1 ミリ秒以上待たされることになる．このキュー待ち遅延はキューイングのオーバーヘッドよりはるかに大きい．もし，エンド・エンドの遅延の主要因がキュー待ち遅延であるなら，必要なパケットが早くスケジューリングされるようキューイングを工夫することは十分に意味がある．

5.2 帯域制限

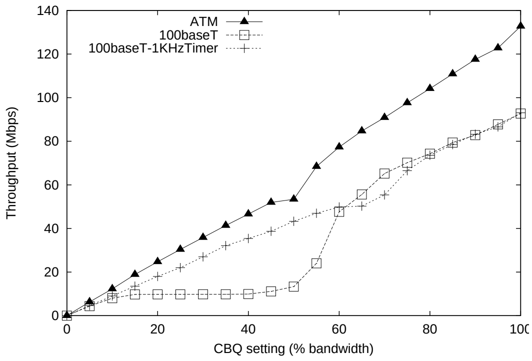

図 7 **ATM** と **100baseT** における **CBQ** の帯域制限

図 5，6，7は CBQ によってクラスの使用帯域を 5% ~ 95% まで 5% きざみに制限した場合の TCP のスループットを測定した結果である．100% の位置には，クラスがルートクラスから帯域の借り入れを行なった場合のスループットが示してある．

図からわかるようにシリアルライン，10baseT，ATM の場合はほぼリニアに帯域制限が行なわれている．また，10baseT の場合，帯域利用率が高くなるとイーサネットの共有メディアの性質上，スループットが飽和することも観測される．

ところが，100baseT の場合，使用帯域が 15% から 55% の範囲で目標を大きく下回っている．これは，4.2.7章で述べたタイマー粒度による 9.6Mbps の限界が現れたものである．参考までに，タイマ精度を 1kHz にしたカーネルによる測定結果を “100baseT-1kHz” として載せてある．この場合は計算上 96Mbps まで動作可能で，期待どおりのスループットが得られている．実際の運用においては，他のトラフィックによるイベントトリガがあるため 100Hz のタイマーでもある程度の精度は期待できるが，100baseT の場合はタイマー精度を上げる効果が明確に表れて，これは理論値とも一致する．

5.3 帯域保証

図 8は CBQ の帯域保証性能を示す．10Mbps，20Mbps，30Mbps，40Mbps の帯域を持つ 4 つのクラスを用意して，デフォルトクラスにマッチするバックグラウンド TCP が流れるなかで，各クラスにマッチ

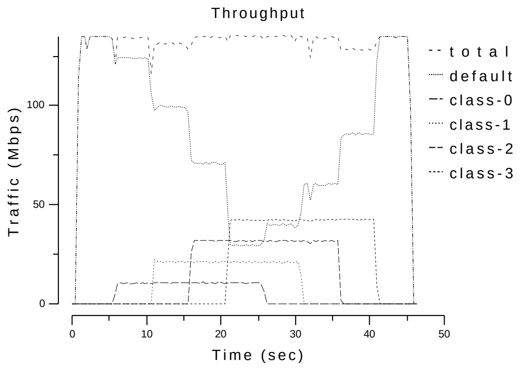

図 8　CBQ の帯域保証性能

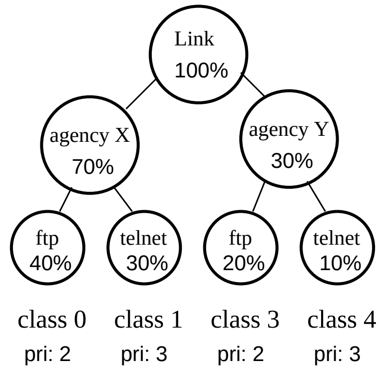

図 9　帯域共有のクラス設定

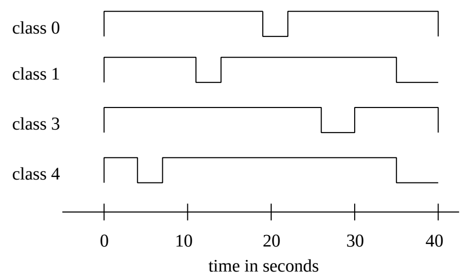

図 10　帯域共有トラフィック発生パターン

する TCP を 5 秒づつずらせて送信開始したものである．各 TCP は利用可能な最大レートで転送している．また，TCP がプロセススケジューリングの影響により振動するのを押えるため，class-0 と class-2 のフローはルータであるホスト B から，他の 3 つのフローはホスト A からホスト C に対して送っている．

測定は ALTQ リリースに含まれている cbqprobe ツールで 400 ミリ秒毎に CBQ 内部の統計データを読みだし，同じくリリースに含まれる cbqmonitor ツールでグラフ化したものである．

図から明らかなように，各クラスには目標どおりの帯域が割り当てられ，まわりのトラフィックにはほとんど影響を受けていないことと，バックグラウンドのフローに余った帯域が割り当てられていることが観測できる．

5.4　借り入れによる帯域共有

借り入れによる帯域共有は，クラスの階層構造に従って正しく帯域を分配する機能である．帯域共有によって複数の組織間で，また，複数のプロトコル間でなんらかのポリシーにしたがった帯域の分配が可能になる．帯域共有には多くの応用がある．例えば，組織間でコスト負担に応じた帯域分配を行なう，また，バルクデータ転送やコンティニュアス・メディアなどのトラフィックタイプ別の帯域使用を制限することもできる．

帯域共有の測定は，LBL で CBQ のシュミレーションに用いられているものとほぼ同様の設定で行なっている [6]．帯域共有の設定は図 9に示すように 2 つの組織がそれぞれ 70%，30% の割合で帯域を共有し，それぞれの組織内で非対話型トラフィックと対話型トラフィックにそれぞれ 40% と 30%，20% と 10% という配分を行なっている．また，対話型トラフィックには高プライオリティが設定されている．余った帯域は借り入れ可能であり，借り入れは最初に同一組織内で配分され，組織で余った帯域は他の組織が借り入れることができる．

組織 X に対応するトラフィックはホスト B から，組織 Y に対応するトラフィックはホスト A から発生し，各クラスに対応するフローは図 10に示すように休止期間を設けて TCP で送る設定となっている．各組織はたとえひとつのクラスが休止していても，組織に割り当てられた帯域を使う権利がある．すなわち，class-0 と class-1，また class-3 と class-4 の合計は常に一定とならなければいけない．

図 11は図 8の場合と同様の手法でデータをとったものである．図からわかるように，両組織が自分のシェアを確保しており，ほぼ目標どおりの帯域共有が実現できている．しかし，部分的に class-4 の高プライオリティ

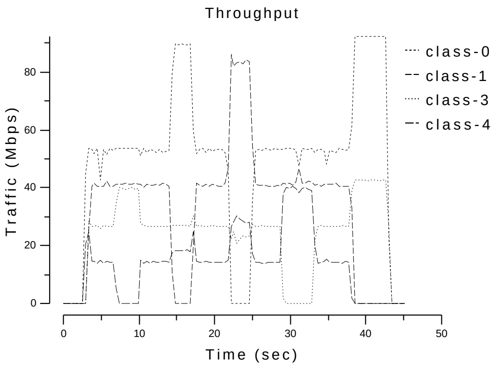

図 11　帯域共有のトレース

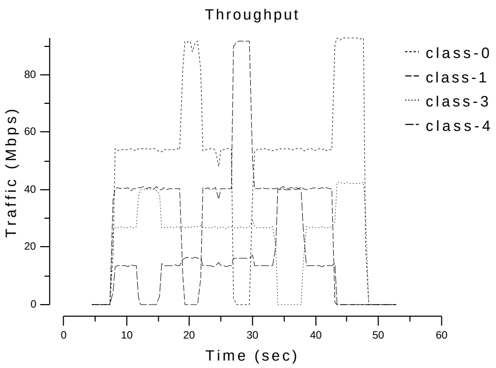

図 12　同一プライオリティによる帯域共有のトレース

クラスが余剰帯域の借り入れ時に目標値を上回る帯域を得ている部分も観測される．

図 12は同様の設定で対話型と非対話型トラフィックに同一のプライオリティを与えた場合の結果である．この場合はプライオリティのある場合の問題点が改善されているのが観測される．したがって，パケット・スケジューリングにおける余剰帯域の分配とプライオリティの連動部分には改善の余地があることがわかる．なお，すべてのクラスに同一のプライオリティを設定しても，インタラクティブなトラフィックはキューにパケットが溜ることが少ないので，遅延は十分小さくなるはずである．

6　関連研究

キューイング実装の汎用フレームワークという発想自体は新しいものではないが，いままで汎用キューイング・フレームワークと呼べるものは筆者の知る限り存在していない．これまでのキューイング研究の実装はその方式のために専用に改造したシステムを使っており[7][18]，フレームワークとして一般化されるにはいたっていない．

ALTQ は複数のキューイング方式に対するスイッチとして実装されていて，この仕組みは BSD UNIX におけるプロトコル・スイッチ構造と似ている．別のアプローチとして，STREAMS [17]や x-kernel [16]のようなモジュール化されたプロトコル・インターフェイスを使ってキューイング方式を実装することも可能で，現に Solaris の CBQ は STREAMS モジュールとして実装されている．しかし，キューイングをプロトコルモジュールとして実装することは可能ではあるが，キューイング機構はプロトコルではないし，実装する際の要求もおおきく異なる．本研究の貢献のひとつは，汎用キューイング・フレームワークの要求事項を明確にしたことである．

7　現　　状

FreeBSD 上の ALTQ の実装は，1997 年 3 月から一般に公開している[2]．現バージョンは FreeBSD-2.x/3.x に対応し，CBQ，RED/ECN，WFQ，FIFOQ のキューイング方式と，ISI の RSVP リリースと CBQ を連動させるためのコードが含まれている．ドライバに関してはほとんどのネットワークカードが利用可能である．

ALTQ は研究用のプラットフォームとして，また，実用ネットワークの運用に世界中で広く使われるようになってきている．最初は RSVP の実験のために利用されるケースが多かったが，最近は実用ネットワークの輻輳の解決に利用するケースが増えてきている．また，バックボーンのトラフィック制御に試験的に使っているプロバイダもある．

WIDE プロジェクトでは RT-Bone の一部として ALTQ を使ったシステムを運用している[9]ほか，WIDE プロジェクトの合宿では合宿内ネットワークを WIDE バックボーンに接続する部分のトラフィック制御に使って効果をあげている[20]．

8 まとめ

現在まで数多くのキューイング方式が提案，研究されてきたが，利用できる実装が少なく，実際にはあまり利用されていないのが現状である．

そこで，さまざまなキューイング機構を実装するためのフレームワークに必要な要件を調査し，ALTQ という代替キューイング機構を設計実装，その上で CBQ，RED，WFQ を実装した．ALTQ を使った PC UNIX ベースのルータのトラフィック制御能力は，当初の予想を上回る性能と精度を示している．

今後は，実際の運用を通じての経験をフィードバックして，より実用的に改良を加えていくつもりである．

ALTQ 開発の努力の少なからぬ部分は多くの人に使ってもらうためのエンジニアリングに費やされている．ALTQ によってネットワークの研究者や管理者が PC と UNIX を使って手軽にキューイングの実験ができるようになった．本研究がキューイング，リンク共有，RSVP といった分野での研究の刺激となり，今後より多くの取り組みが行なわれていくことを期待したい．

参考文献

[1] Baily, M. L., Gopal, B., Pagels, M. A., Peterson, L. L., and Sarkan, P.: PATHFINDER: A Pattern-Based Packet Classifier, *Proceedings of Operating Systems Design and Implementation*, Monterey, CA, November 1994, pp. 115–123.

[2] Cho, K.: Alternate Queueing for BSD Unix, http://www.csl.sony.co.jp/person/kjc/software.html, 1997.

[3] Demers, A., Keshav, S., and Shenker, S.: Analysis and simulation of a fair queueing algorithm, *Proceedings of SIGCOMM '89 Symposium*, Austin, Texas, September 1989, pp. 1–12.

[4] Floyd, S.: TCP and Explicit Congestion Notification, *ACM Computer Communication Review*, Vol. 24,No. 5(1994), pp. 10–23.

[5] Floyd, S. and Jacobson, V.: Random Early Detection Gateways for Congestion Avoidance, *IEEE/ACM Transaction on Networking*, Vol. 1,No. 4(1993), pp. 397–413.

[6] Floyd, S. and Jacobson, V.: Link-Sharing and Resource Management Models for Packet Networks, *IEEE/ACM Transactions on Networking*, Vol. 3,No. 4(1995), pp. 365–386.

[7] Gupta, A. and Ferrari, D.: Resource Partitioning for Real-Time Communication, *IEEE/ACM Transactions on Networking*, Vol. 3,No. 5(1995), pp. 501–508.

[8] Hewlett-Packard Company: *Netperf: A Benchmark for Measuring Network Performance*, 1993. http://www.cup.hp.com/netperf/NetperfPage.html.

[9] 石井公夫, 塩野崎敦, 木幡康弘, 小林克志, 石田慶樹, 長健二朗, 寺岡文男: WIDE プロジェクトにおける実時間通信バックボーンの構築, 日本ソフトウエア科学会 第１４回全国大会, October 1997.

[10] Keshav, S.: On the Efficient Implementation of Fair Queueing, *Internetworking: Research and Experience*, Vol. 2(1991), pp. 157–173.

[11] McCanne, S. and Floyd, S.: NS (Network Simulator), http://www-nrg.ee.lbl.gov/ns/, 1995.

[12] McKenney, P. E.: Stochastic Fairness Queueing, *Proceedings of INFOCOM*, San Francisco, California, June 1990.

[13] McKusick, M. K., Bostic, K., Karels, M. J., and Quarterman, J. S.: *The Design and Implementation of the 4.4 BSD Operating System*, Addison-Wesley Publishing Co., 1996.

[14] Nagle, J.: On packet switches with infinite storage, *IEEE Trans. on Comm.*, Vol. 35,No. 4(1987), pp. 435–438.

[15] Parekh, A.: A Generalized Processor Sharing Approach to Flow Control in Integrated Services Networks, LIDS-TH 2089, MIT, February 1992.

[16] Peterson, L. L., Hutchinson, N. C., O'Malley, S. W., and Rao, H. C.: The x-kernel: A Platform for Accessing Internet Resources, *Computer*, Vol. 23,No. 5(1990), pp. 23–34.

[17] Ritchie, D. M.: A Stream Input-Output System, *AT&T Bell Laboratories Technical Journal*, Vol. 63,No. 8(1984), pp. 1897–1910.

[18] Stoica, I. and Zhang, H.: A Hierarchical Fair Service Curve Algorithm For Link-Sharing, Real-Time and Priority Services, *Proceedings of SIGCOMM '97 Symposium*, Cannes, France, September 1997, pp. 249–262.

[19] Sun Microsystems, Inc.: *Solaris RSVP/CBQ*. ftp://playground.sun.com/pub/rsvp/.

[20] 宇夫陽次朗, 石山政浩, 新善文, 村井純: 大規模な仮説ネットワークテストベッドの設計・構築とその運用, インターネットコンファレンス'97, December 1997, pp. 169–180.

[21] Wakeman, I., Ghosh, A., Crowcroft, J., Jacobson, V., and Floyd, S.: Implementing real-time Packet Forwarding Policies using Streams, *Proceedings of USENIX '95*, New Orleans, Louisiana, January 1995, pp. 71–82.

[22] Zhang, L., Deering, S., Estrin, D., Shenker, S., and Zappala, D.: RSVP: A New Resource ReSerVation Protocol, *IEEE Network*, Vol. 7(1993), pp. 8–18.