PC Unix ルータによるトラフィック制御の実現

Traffic Control by PC Unix based Routers

長 健二朗[†]
Kenjiro Cho

† (株) ソニーコンピュータサイエンス研究所
Sony Computer Science Laboratory Inc.

概 要

トラフィック制御機能を持つ PC Unix ベースのルータを実現するため、 ALTQ という BSD 系 Unix への汎用キューイング ・インターフェイスを設計実装し、そのフレームワーク上に CBQ の移植を行なった。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。

1 はじめに

インターネットにとって QoS 保証サービス等を取り込んだインテグレーテッド ・サービスの実現は重要な課題である。

そこで今回、 ALTQ という BSD 系 Unix の汎用キューイング ・インターフェイスを設計実装し、そのフレームワーク上に CBQ [4] の移植を行なった。CBQ は RSVP [11] と連動して、動的な資源予約による帯域割り当てが可能である。

本プロジェクトは、

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. 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 ALTQ: 代替キューイング機構

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 [6] [4] [9] [3] 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 を設計した。 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な機構等を持つ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。

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。

2.1 設計

トラフィック制御の効果があるのは出力キューなので、キュー構造を工夫するのは出力キューのみでよい。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が、一般には、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、非ワークコンサービングなキューでは、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ず、その場合はこの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。キューをフラッシュする操作も必要である。

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、キューの先頭を覗き見る操作は、いくつかのデバイスドライバが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ために使っているが、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。

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た。

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は、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ため、運用上の利便や信頼性が向上し、デバッグもしやすい。

2.2 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ため、結局いずれにしても変更が必要となる if_output 関数内でフロー情報を取り出すこととした。

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。

代替キューをアクティベイトするには、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。

3 CBQ

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高度なキューイング機構をその上に実装して検証を行なう必要がある。そこで RSVP に利用可能でソースコードが公開されている CBQ を選び、移植を行なった。

CBQ (Class-Based Queueing) [4] は、ローレンスバークレイ研究所 (LBNL) の Van Jacobson によって提案され、同研究所の Sally Floyd によって解析、シュミレーションが行なわれてきた。

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。

CBQ の実装は以下のような構成要素を持つ。

1. 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。この際、クラスに設定されたキューの最大長を越える場合にはパケットは廃棄される。
2. 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決定する。各クラスにはプライオリティが与えられるようになっており、プライオリティの高いクラスからパケット送出が行なわれる。単なるプライオリティキューとの違いは、後述の使用帯域制限が効くため低プライオリティクラスが枯渇することが

ないことである。同一プライオリティを持つクラス間ではウエイテッド・ラウンドロビン・スケジューリングとなる。

3. 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遅延機構によって使用帯域が制限される。
4. 遅延機構 (delayer) は、使用帯域を超過したクラスの使用を制限するために、ある期間パケット送出を差し止める機能を提供する。

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。特に、推量機構とパケット・スケジューラは簡単な計算で、かつ、Unix カーネルのタイマの粒度で各クラスの使用帯域を近似し、送出量を制御できるよう工夫がされている。

CBQ は LBNL から NS シュミレータ [8] 用のコードが公開されている。また、UCL が LBNL と共同で Solaris への実装を試み [10]、それを Sun が引き継いで Solaris の RSVP [11] のためのトラフィック制御機構として開発して公開している他、いくつかのベンダが CBQ の実装を試みているようである。今回、CBQ の ALTQ への実装は、Sun のコードをもとに移植を行なった。

3.1 CBQ の移植

CBQ の動作原理、シュミレーション結果等はオリジナルの出典を参照されたい [4] [10]。ここでは、CBQ の移植とパフォーマンスに関して問題となった項目についてのみ触れる。

3.1.1 性能限界

現在の PC はハードウエア的には、適切な選択をすれば、CPU 性能、IO 性能、ネットワークカード性能とも高速ルータとして十分に耐えられる性能を持っている。しいて言えば、メモリアクセスがボトルネックとなる。

CBQ の性能面で一番問題になったのがタイマ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リミット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、タイマ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マに頼ることになる。

ほとんどの Unix システムのデフォルトのタイマの粒度は 10 ミリ秒であるが、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、次のタイマイベントまでリジュームされない状態となる。この状態が続くと転送レートは、$rate = packetsize * maxburst * 8/0.02$ 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 で 9.6Mbps、MTU が ATM の 9180 で 58.8Mbps にしか達しないことになる。この問題は特に高速でかつ 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、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。それでも、TCP 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。

3.1.2 帯域共有アルゴリズム

Sally 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 [4]。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。やはり、このような理論的

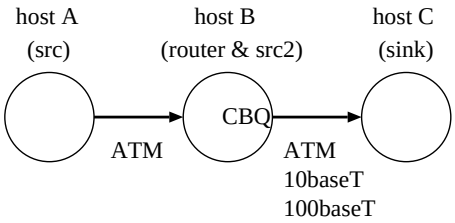

図 1　テストシステム構成

根拠の不明確な経験則を組み込んだアルゴリズムをチューニングする作業は容易ではない。

4　性能評価

測定は 3 台の PentiumPro マシン (200MHz, 440FX) を用いて行なった。OS は FreeBSD-2.2.2 で、altq-0.3.2 による ALTQ/CBQ を使っている。

図 1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 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 を切替えて使用した。CBQ はホスト B 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。

表 1　CBQ のスループット

device	cbq off(Mbps)	cbq on(Mbps)
ATM	133.20	133.31
10baseT	6.39	6.46
100baseT	93.04	92.89
local loop	326.58	302.30

表 2　ATM での CBQ の遅延

CBQ	request size (bytes)	response size (bytes)	trans. per sec	RTT (usec)
off	1	1	2821.89	354
on	1	1	2744.03	364
off	64	64	2301.06	435
on	64	64	2243.14	446
off	1024	64	1476.31	677
on	1024	64	1454.39	688
off	8192	64	394.59	2534
on	8192	64	392.76	2546

表 3　10baseT での CBQ の遅延

CBQ	request size (bytes)	response size (bytes)	trans. per sec	RTT (usec)
off	1	1	2277.37	439
on	1	1	2234.13	448
off	64	64	1800.75	555
on	64	64	1768.02	566
off	1024	64	681.05	1468
on	1024	64	676.45	1478
off	8192	64	116.64	8573
on	8192	64	116.67	8571

4.1　オーバーヘッド

CBQ によるオーバーヘッドの測定をスループットと遅延に対して行なった。

4.1.1　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 CBQ を使わない場合と使った場合で比較している。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。

いずれの場合も 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 7%程度のオーバヘッドとなっている。この値が CPU の処理能力の

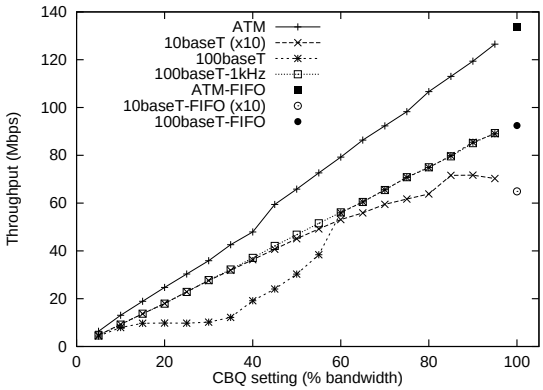

図 2　各メディアにおける CBQ の帯域制限

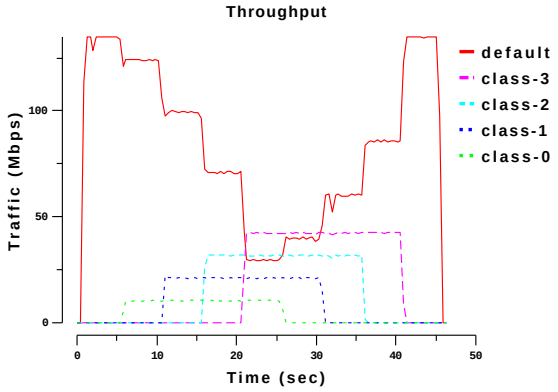

図 3　CBQ の帯域保証性能

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ったとしても十分な処理能力があることを示している。さらに、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。

4.1.2　遅延

表 2、3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の測定結果とパケットあたりのラウンドトリップタイムが示してある。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。測定結果からパケットあたりの CBQ のオーバーヘッドは 10us 程度であることがわかる。

4.2　帯域制限

図 2は CBQ によって使用帯域を 5% ~ 95%まで 5%きざみに変化させた場合の TCP のスループットを測定した結果である。なお、10baseT のスループットは、同一のグラフに入れるために 10 倍してプロットしてある。また、-FIFO で示されているのは、CBQ を使わない状態で観測されたスループットである。

図からわかるように ATM、10baseT の場合はほぼリニアに帯域制限が行なわれている。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、3.1.1 章で述べたタイマ粒度による 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 のタイマでもある程度の精度は期待できるが、100baseT の場合はタイマ精度を上げる効果が明確に表れて、理論値とも一致する。

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。

4.3　帯域保証

図 3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。

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 400ms 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

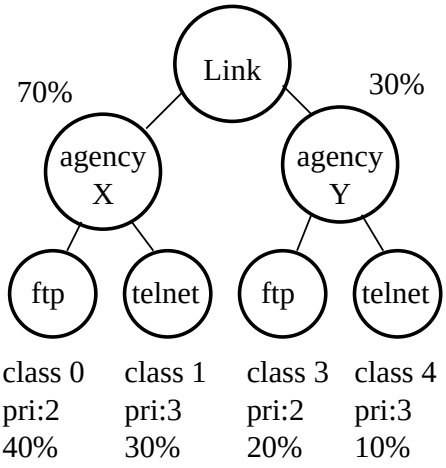

図 4　帯域共有のクラス設定

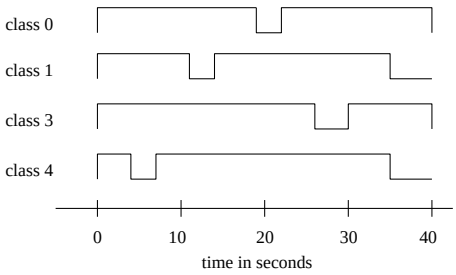

図 5　帯域共有トラフィック発生パターン

まったく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。

4.4　借り入れによる帯域共有

帯域共有の測定は、LBL で行なわれている 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 [4]。帯域共有の設定は図 4 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、借り入れはまずは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 5 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。

図 6 は図 3 の場合と同様の手法でデータをとった

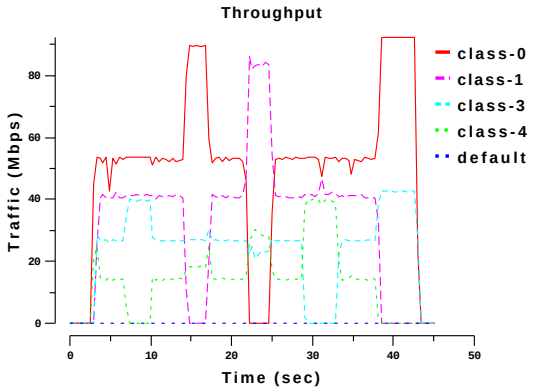

図 6　帯域共有のトレース

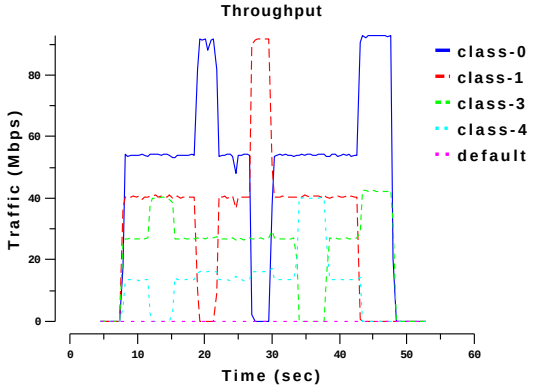

図 7　同一プライオリティによる帯域共有のトレース

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 7 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。

4.5　未評価項目

以下に今回評価できなかった項目を上げておく。

- プライオリティ・スケジューリング: CBQ ではクラスごとに独立したキューを持つため、プライオリティを使わなくとも他のトラフィック

による遅延変動が少ないため、プライオリティ・スケジューリングの有効性の検証が難しい。また、今回のテストシステム構成ではバックグラウンドトラフィックがデスティネーションマシンに負荷を加えることなく遅延測定をすることができなかった。

- クラス数が増大した時のオーバーヘッド: クラス数が増大するとクラス分け機構のフィルタ・マッチング処理に効いてくる。しかしながら、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しているため、数百程度までは性能低下は現れてこないはずである。それよりも各クラス構造が消費するメモリ量が問題となってくると思われる。

5 現状

FreeBSD 上の代替キューイングの実装は、1997 年 3 月からアルファ・リリースという形で一般に公開している [1]。キューイング方式では CBQ と WFQ [2] [7] [5] が含まれ、5 種類のイーサネットドライバ、1 種類の ATM ドライバ、3 種類の同期シリアルドライバが利用可能である。

現在 ALTQ は WIDE プロジェクトで RT-Bone の一部として Cisco ルータと専用線で接続して運用されている [12]。また、いくつかの研究機関で RSVP 関連をはじめとした研究用プラットフォームとして利用されている。

6 まとめ

現在まで数多くのキューイング機構が提案、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として ALTQ という代替キューイング機構を設計実装し、その上で CBQ と WFQ を実装したプロトタイプを作成した。

今回のプロトタイプで、基本的な機能の動作確認、性能確認、RSVP とのインターフェイス等、当初の目標は達成できたと考えている。とくに、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、WIDE プロジェクトの RT-Bone における実験等の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。また、他のキューイング方式の実装や IPv6 への対応も検討中である。

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] Kenjiro Cho. Alternate queueing for bsd unix. , Sony CSL, 1997. http://www.csl.sony.co.jp/person/kjc/programs.html.

[2] Alan Demers, Srinivasan Keshav, and Scott Shenker. Analysis and simulation of a fair queueing algorithm. In *SIGCOMM*, pp. 1–12, Austin, Texas, September 1989. ACM.

[3] Sally Floyd and Van Jacobson. Random early detection gateways for congestion avoidance. *IEEE/ACM Transaction on Networking*, Vol. 1, No. 4, pp. 397–413, August 1993.

[4] Sally Floyd and Van Jacobson. Link-sharing and resource management models for packet networks. *Transactions on Networking*, Vol. 3, No. 4,, August 1995.

[5] P. E. McKenney. Stochastic fairness queueing. In *INFOCOM*, San Francisco, California, June 1990. IEEE.

[6] John Nagle. On packet switches with infinite storage. *IEEE Trans. on Comm.*, Vol. 35, No. 4,, April 1987.

[7] Abhay Parekh. A generalized processor sharing approach to flow control in integrated services networks. LIDS-TH 2089, MIT, February 1992.

[8] McCanne S. and Floyd S. Ns (network simulator). , LBNL, 1995. http://www-nrg.ee.lbl.gov/ns/.

[9] Scott Shenker, David D. Clark, and Lixia Zhang. A scheduling service model and a scheduling architecture for an integrated services packet network. Technical report, Xerox PARC, Palo Alto, California, August 1993.

[10] Ian Wakeman, Atanu Ghosh, Jon Crowcroft, Van Jacobson, and Sally Floyd. Implementing real-time packet forwarding policies using streams. In *Proc. of Usenix*, pp. 71–82, New Orleans, Louisiana, January 1995.

[11] Lixia Zhang, Steve Deering, Deborah Estrin, Scott Shenker, and Daniel Zappala. RSVP: a new resource reservation protocol. In *Proceedings of the International Networking Conference (INET)*, pp. BCB–1, San Francisco, California, August 1993. Internet Society.

[12] 石井ほか. 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.