# 適応的アプリケーションのための資源管理機構および資源量通知機構を持つフレームワーク

安田 泰勲, 稲村 浩
NTT 情報通信研究所
〒 239-0847 神奈川県 横須賀市 光の丘 1-1
Internet: yasunori, inamura@isl.ntt.co.jp

**概要**

移動ネットワーク環境では, マルチメディアアプリケーションは柔軟な QoS (Quality of Services) を達成するために, ネットワークインターフェースの切替えに対して適応的に振舞わなければならない. 本研究では, アプリケーションの適応制御のために, ネットワークインターフェースの切替をトリガとしてその変化を通知し, 資源管理を行なう機構を持つフレームワークについて述べる. 我々はこのフレームワークを RT-Mach マイクロカーネル上に実装し評価を行なった. 本研究の特徴はインターフェースの切替を適応動作のトリガとしたこと, '資源管理機構' および '資源量通知機構' が協調動作することである.

## 1. はじめに

ラップトップコンピュータでは様々なネットワークデバイスが利用可能になっている. これらのネットワークデバイスは PCMCIA カードで提供されている. ユーザはコンピュータを利用中にデバイスを交換すること (hot-swapping) が可能であり, ユーザのモビリティは格段に向上した. 例えば, オフィスにいるときはラップトップコンピュータを Ethernet につなぎ, 会議室に移動する時は無線 LAN に切替えて使うことによって, ネットワークを利用するアプリケーションを使い続けることができる.

我々はこのような環境を移動ネットワーク環境と呼ぶ. 移動ネットワーク環境では, マルチメディアアプリケーションは適応的に振舞わなければならない, なぜならネットワークデバイスの切替えによって利用可能なネットワーク資源の量が大きく変化するため, アプリケーションの QoS (Quality of Service) に大きく影響するからである. デバイスの切替が起こった場合, 通常の静的な資源予約ではうまくいかない. 例えば, Ethernet に接続されたラップトップコンピュータで, インターネット電話アプリケーションとファイル転送アプリケーションを同時に利用する場合を考える. 最初は, インターネット電話は 2Mbps の帯域を予約し, ファイル転送アプリケーションは残りの帯域を利用している. ここで通信中にユーザがネットワークデバイスを Ethernet(10Mbps) から WaveLAN(2Mbps) に切替える. この場合, インターネット電話は帯域を再予約するだけでなく, 自分自身の QoS のレベルを落し, 利用する帯域を減らす必要がある. このような適応動作を実現するためには, システムがネットワークデバイスの変化をアプリケーションに通知しなければならない.

本研究の特徴は, 移動ネットワーク環境において柔軟な QoS 制御を達成するために, デバイスの切替をアプリケーションの適応動作のトリガーとしたこと, このフレームワークでは切替えによって変化した利用可能なネットワーク資源量を通知する機構およびネットワーク資源の管理機構の二つの機構が協調動作することである.

我々は, 利用可能な資源量を通知する機構, および帯域幅を制御する機構をもつフレームワークを設計, 実装した. このフレームワークの評価として, 簡単な適応的アプリケーションを用いたデバイス切替えのハンドリングの所要時間, 帯域幅制御の正確さ, パケット送信のオーバヘッドの評価を行なった.

まず, 2節ではこのフレームワークの概要について述べ, 3節でプロトタイプの設計と実装について述べる. 次に 4節で本プロトタイプの評価について述べる. 最後に 5節で関連研究, 6節で今後の予定について述べ, 7節でまとめる.

## 2. 適応的アプリケーションのためのフレームワークの概要

この節では我々が提案する適応的アプリケーションのためのフレームワークの概要について述べる. このフレームワークは二つのメカニズム, 資源管理機構および資源量通知機構から構成される.

まず, 資源管理機構および資源量通知機構について述べ, 次にフレームワークの全体のモデルについて述べる.

### 2.1. 資源管理機構

資源管理機構では, ラップトップコンピュータで動作する全てのアプリケーションの使用する帯域幅を制御する. アプリケーションは利用したい帯域幅の割り当て要求を資源管理機構に伝える. 資源管理機構は, アプリケーションからの要求が現在利用可能な帯域幅より少ない場合は帯域幅を割り当てを行ない, 不十分な場合には割り当てを行なわない.

資源管理機構を導入する利点は二つある. 一つは他のアプリケーションが帯域幅を使い尽くさないようにアプリケーション毎に保護することが可能になることである. 通常のオペレーティングシステムでは全てのアプリケーションに対して暗黙に公平に帯域幅を割り当てる. 例えば, マルチメディアアプリケーションとそれ以外のアプリケーションを同時に使う場合を考えると, マルチメディアアプリケーションの方に帯域幅を余分に割り当てたいという要求がある. この要求を満たすためには資源管理機構によって帯域幅を分割することが必要になってくる.

二つめの利点として, 適応動作を行なうマルチメディアアプリケーションを実装する場合, 資源管理機構を利用することによってプログラミングが簡単になる. もしこの機構が利用できなければ, アプリケーションプログラマはアプリケーション自身が帯域幅の利用を調節する複雑なコードを書かなければならない.

### 2.2. 資源量通知機構

ラップトップコンピュータで利用可能なネットワークデバイスの例を表1に示す.

| device | bandwidth(bps) | latency(msec) |
|---|---|---|
| Ethernet | 10 M | 1 ··· 10 |
| Wireless LAN | 1 ··· 2 M | 1 ··· 10 |
| serial/modem | 28.8 ··· 64 k | 10 ··· 100 |

表 1: 移動ホストで利用できるネットワークデバイスと特性

この表からわかるように, ネットワークデバイスの帯域幅や遅延はデバイスによって一桁から二桁のオーダで違うため, 固定的な資源予約だけでは許容できる QoS を達成することは難しい. それゆえ, アプリケーション自身が利用可能なネットワークに適応して QoS を変化させなければならない. そこで本システムでは適応制御のトリガとしてネットワークデバイスの切替えを用い, 切り替えを資源管理機構に通知する機構を用意する.

利用可能なネットワーク資源の量は, 資源管理機構は使用可能なデバイス名から検索した最大帯域幅から各アプリケーションが利用している帯域幅を差し引いた値をアプリケーションに通知する. アプリケーションはこの通知された値をもとに, 再割り当て要求を出し, 適応動作を行なう.

### 2.3. フレームワークのモデル

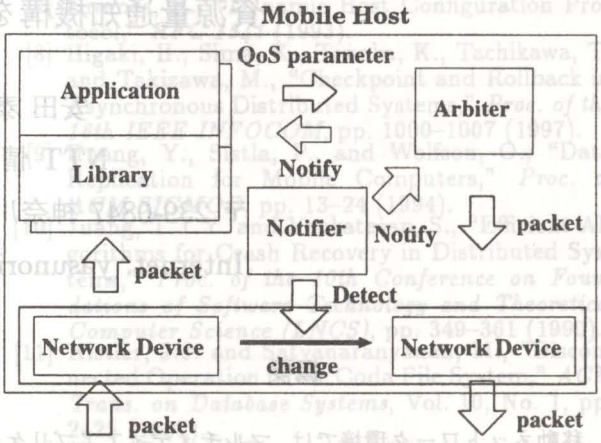


図 1: フレームワークのモデル

ここではフレームワーク全体のモデルについて述べる. 図1に全体のモデルを示す. 我々のフレームワークは三つのコンポーネントから構成される.

一つめは 'Arbiter' であり, これは資源管理機構を実現している. 'Arbiter' はアプリケーションからのパケットを全てうけとり, アプリケーションからの QoS パラメータ (帯域幅, 遅延) をもとにスケジューリングを行なう.

二つめは 'Notifier' であり, これは PCMCIA カードイベントからネットワークデバイスの切替を検知し, ネットワークデバイス名を 'Arbiter' に通知する. 'Arbiter' は 'Notifier' から通知を受けとると, 利用可能な帯域幅と遅延に変換し, アプリケーションに通知する. 資源量通知機構は 'Notifier' と 'Arbiter' が連係することによって実現されている.

三つめは 'Library' であり, これはアプリケーションにリンクされている. これはアプリケーションと 'Arbiter' とのインターフェースであり, アプリケーションはこの 'Library' を利用することによって 'Arbiter' に QoS パラメータを伝えたり, 'Arbiter' からの通知を受けとったりすることが可能になる.

## 3. フレームワークの設計と実装

この節では2節で述べたフレームワークのモデルをもとにした設計と実装について述べる. まずこのフレームワークを設計, 実装したプラットフォームについて述べ, 次に各コンポーネントの設計, 実装について述べる. 最後に各コンポーネントの協調動作の様子について述べる.

我々はプラットフォームとして RT-Mach[7] マイクロカーネル環境を選択した. RT-Mach マイクロカーネルはマルチメディアアプリケーションを実装するためのいくつかの便利な機能 (リアルタイムマルチスレッド, リアルタイムスケジューラ, ネットワーク透過なプロセス間通信など) を提供している.

### 3.1. Arbiter の設計と実装

Arbiter はパケットスケジューリングによって帯域幅を制御する. Arbtier はアプリケーションからの要求をもとにキューを作り, QoS パラメータを設定する. QoS パラメータは帯域幅と遅延の組である. このフレームワークでは UNIX ソケット互換のプログラミングインターフェースを採用しており, キューは一つの UNIX ソケットに対して一つ用意される. Arbiter はアプリケーションからソケット経由で受けとった全てのパケットをキューに入れ, キューのパラメータに基づいてスケジューリングを行ない, ネットワークデバイスに書き込む.

本実装ではパケットスケジューリングアルゴリズムとして Weighted Deficit Round Robin (WDRR)[6] アルゴリズムを採用した. このアルゴリズムは Weighted Round Robin (WRR) を改良し, 可変長パケットに対応したものである. WDRR は帯域幅を制御することが可能であり, Max-min fair share に従う. パケットスケジューリングアルゴリズムとして Weighted Fair Queuing (WFQ) [6] アルゴリズムが知られているが, 計算時間の短さ, 実装の容易さから WFQ ではなく, WDRR を選択した. 本実装では主に帯域幅の制御を行なっているが, 遅延に関しては低遅延, 高遅延の 2 レベルを用意することで, 対話処理を行なうプログラムなどの低遅延が必要なプログラムに対応している. 低遅延のキューに入っているパケットは, 常に高遅延のキューに入っているパケットよりも優先して送出される.

キューはプロトコルスタック処理部とデバイスドライバの間に位置する. デバイスドライバの中にキューを作った場合, 切替える可能性のある全てのデバイスのデバイスドライバを変更しなければならないため, 実装の手間が大きい. しかし, このように設計するによって, 特定のプロトコルやデバイスに依存せずに, 様々なプロトコルのパケットやデバイスを一元的に扱うことができ, 実装の手間も減らすことができる.

### 3.2. Notifier の設計と実装

Notifier はネットワークデバイスの切替を検知し, Arbiter に通知する機能を持つ. ラップトップコンピュータの切替え可能なデバイスは PCMCIA カードで提供されているため,PCMCIA カードイベントをハンドルすることにより, ネットワークデバイスの検知を行なう.

Notifier は, MKng プロジェクト [5] によって RT-Mach 環境に移植された PAO[1] と連係して動作する. PAO は FreeBSD 用のモバイルパッケージであり, PCMCIA カードイベントハンドラ (pccardd) や各種デバイスドライバなどを持つ. Notifier は, PCMCIA カードイベントハンドリングを行なう pccardd からイベントを受けとり, ネットワークデバイスの設定が終った後に, Arbiter にメッセージとして有効になったデバイス名を通知する.

### 3.3. Library の設計と実装

Library はアプリケーションが Arbiter と通信するための Application Programming Interface (API) を提供しており, アプリケーションはこれらの API を使うために Library をリンクする. これらの API では主に Arbiter に QoS パラメータを設定, 変更, 削除する関数, 適応動作するためのハンドラを登録する関数などがある. データの送受信には, Library に用意されている socket(), bind(), send(), recv() などを用い, UNIX ソケットプログラミングと同様に行なう. アプリケーションは本 API を用いて各キューにパラメータを設定する. 適応動作のためのハンドラも各キューに対して設定する. 帯域幅の設定は現在利用可能なデバイスの最大帯域幅に対する割合で指定し, 遅延の設定は低遅延, 高遅延のどちらかを指定する. パラメータの設定がない場合には, 割り当てられていない残りの帯域幅を利用することになり, ハンドラを設定しない場合には適応動作なしでそのまま動作する.

RT-Mach マイクロカーネル環境では, 通常アプリケーションがネットワークを利用する場合には, Lites 4.4 BSD-Lite UNIX server (以下 Lites) でプロトコル処理を行ないデータを送受信する. しかし, これとは別に処理性能を向上させるために, プロトコル処理を行なうライブラリ libsockets[3] が用意されている. libsockets では UNIX ソケットインターフェースを持ち, 通常のネットワークプログラミングスタイルでコーディングを可能にしている. 本実装は libsockets をベースとし, Arbiter との通信用 API の追加, およびキューイングを行なうための UNIX ソケットシステムコールの内部処理の変更を行なった.

### 3.4. 各コンポーネントの協調動作

ここでは Arbiter, Notifier, および Library の協調動作の様子について述べる. 本プロトタイプの実装の全体図を図 2に示す.

アプリケーションはまず Library を用いて UNIX ソケットを作り, 次にソケットにバインドされたキューに QoS パラメータを設定する. QoS パラメータは帯域幅と遅延の組である. アプリケーションはパケットをソケットに送ることによって Arbiter のキューに入れる. パラメータを設定しない場合には共用のデフォルトキューにパケットが送り込まれる. デフォルトキューにあるパケットは未割り当ての帯域幅がある場合にはその分を適宜利用して送出される.

Arbiter は QoS パラメータを受けとると, アドミッションコントロールを行なう. つまり, 現在利用可能な帯域幅をチェックし資源が十分にある時には割当を行ない, 十分でない場合には利用可能な帯域幅をアプリケーションに通知し, 割り当ては行なわない.

PCMCIA カードのネットワークデバイスが利用可能／不可能になった場合, pccardd がカードイベントを処理し, message sender を呼び出し, インターフェースを enable/disable する. message sender は利用可能／不可能になったデバイス名とその状態 (利用可能／不可能) を

Arbiter に通知する．Arbiter は利用不可能のメッセージを受けとった場合，ただちにパケットの送出を止め，アプリケーションの書き込みをブロックする．Arbiter は利用可能のメッセージを受けとった場合，パケットの送出を再開し，アプリケーションの書き込みのブロックを解除する．さらに，新しい利用可能な帯域幅と平均遅延をアプリケーションに通知する．

アプリケーションはメッセージを受けとると，通知された情報を元に資源の再割り当て要求を出すことができる．再割当要求を出さない場合には以前の割当のままで動作を続ける．

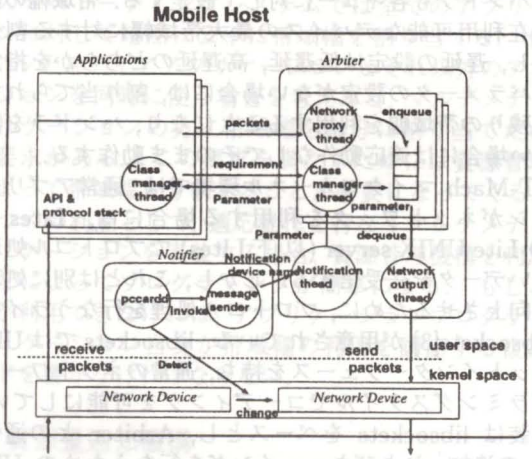


図 2: RT-Mach 上でのフレームワークの実装

## 4. 評価

この節では本実装の評価について述べる．以下の三つの基準で評価を行なった．

**デバイス切替え処理の所要時間** デバイスの変化に対するフレームワークの反応時間を評価する．通知機構はデバイスが変化した場合，可能な限り素早く変化を通知し，資源管理機構と協調動作すべきであり，もっとも重要な評価項目である．

**帯域幅制御の精度** 資源管理機構を評価する．指定した帯域幅と達成した帯域幅の差で評価を行ない，差が小さいほど良い結果が得られているといえる．

**パケット送出のオーバヘッド** この本実装でのスケジューラの配置の妥当性を評価する．この項目では RT-Mach マイクロカーネル環境で通常の設定でネットワークを利用する場合と比較して，一つのパケットを送出する処理に対してどの程度オーバヘッドが生じるかを測定し，システムの実装の妥当性を判断する．

実験環境は以下の通り．

**送り側** DEC HiNote Ultra-II(Pentium 150MHz,56MB memory,3COM 3C589, AT&T WaveLAN), RT-Mach 3.0 + Lites (MKNG-003 snapshot)

**受け側** EPSON ENDEAVOR(Pentium 133MHz,16MB memory,3COM 3C509B), RT-Mach 3.0 + Lites (MKNG-003 snapshot)

### 4.1. デバイス切替え処理の所要時間

この項目では二つの単純なテストアプリケーションを実装し評価を行なった．これらのアプリケーションは 1472 バイトの UDP パケットを連続的に送信するプログラムである．違いは，Arbiter に帯域幅の割り当て要求および変化への適応動作をする適応的アプリケーションか，適応動作をしない非適応的アプリケーションかである．

この実験の手順は，まずラップトップコンピュータに Ethernet(10Mbps) と WaveLAN(2Mbps) の PCMCIA カードを予め挿入しておき，Ethernet の方だけを有効にしておく．次に適応的アプリケーションは Ethernet に対して 800kbps になるように割合を計算して Arbiter に帯域の割り当て要求を出し，パケットを送出する．また、非適応的アプリケーションは特別な処理はせずに単にパケットを送出する．そして，ユーザがネットワークインターフェースを切替えるコマンドを呼びだし，Ethernet から WaveLAN に有効なネットワークデバイスを切替え，ネットワークインターフェースやルーティングテーブルの再設定を行なう．Notifier はデバイスの切替を検知するとメッセージを Arbiter に送る．Arbiter はメッセージを受けとると，適応的アプリケーションに利用可能な帯域幅を通知する．適応的アプリケーションのハンドラは Arbiter からのメッセージを受けとると，その値をもとに再割当要求を出す．この一連の処理の間も非適応的アプリケーションは Arbiter によって余った帯域幅を利用してパケットを送り続けている．

適応的アプリケーションの受け側での 100 パケット毎のスループットおよびデバイスの切替によるハンドリングの所要時間を計測した結果を図 3に示す．

図 3 の上部は適応的アプリケーションのスループットを表している．この図からは Ethernet と WaveLAN の両方でほぼ 800 kbps のスループットが得られている．帯域幅の指定値と達成値の差は Ethernet と WaveLAN の両方あわせて，最悪でも ±8.5% の範囲に収まっている．この測定では，結果をファイルに出力しながらアプリケーションを動作させているため，測定結果の出力に要した時間も含まれているため，ずれが生じている．純粋な帯域幅制御の精度の評価は 4.2節で行なっている．

図 3 の下部はネットワークデバイスの切替えのシーケンスを表している．まず，切替えコマンドが呼び出されると，Notifier は切替を検知し，Arbiter に device down メッセージを送る．Arbiter はそのメッセージを受けとると，ネットワーク用のポートを disable し，パケットの送出を止める．ここまでの所要時間は 105msec である．次に，Ethernet インターフェースを disable してからルー

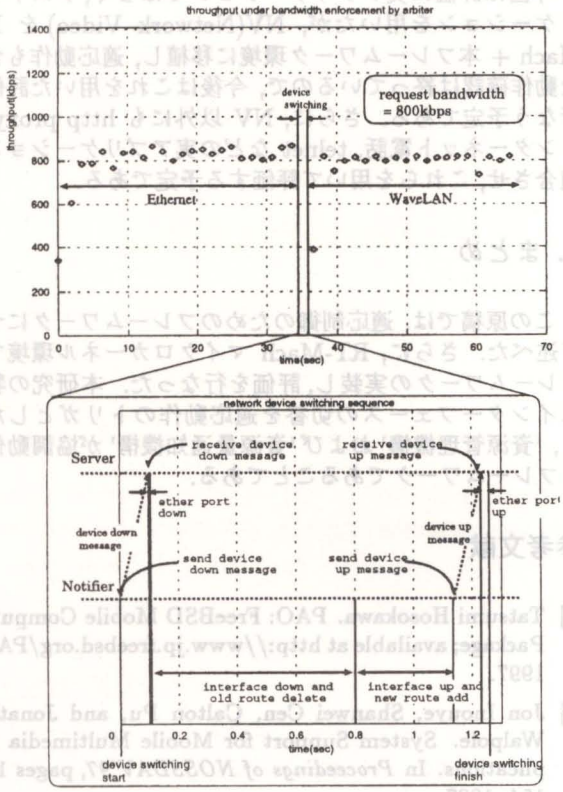


図 3: デバイスの切替えのハンドリングのチャート.

ティングテーブルを flush し, さらに WaveLAN インターフェースを enable してからルーティングテーブルを更新するのにかかった時間は 701msec. 最後に, device up メッセージを Arbiter に送り, Arbiter がネットワーク用のポートを再設定し, パケットを再び送り出すのにかかった時間は 373msec. 合計でデバイス切替のハンドリングの所要時間は 1.337sec であり, これは十分妥当であると考える. なぜなら, すべてのアクティブな TCP のコネクションは, この実験で得られたハンドリングの所要時間ではタイムアウトせずにコネクションを維持できるからである.

## 4.2. 帯域幅制御の精度

この項目では二つの単純なテストアプリケーションを実装し評価を行なった. これらのアプリケーションは 1472 バイトの UDP パケットを連続的に送信するプログラムである. 一つは帯域幅を $x$% 指定し (以下 *Primary*) 1000 個パケットを送出し, もう一つは帯域幅 $(100 - x\%)$ 指定し (以下 *Conpetition*), Primary とネットワークアクセスの競合が起きるように連続的にパケットを送出する.

測定は受け側で Primary の 1000 個のパケットを送出するのにかかった時間を Pentium counter を用いて Ethernet, WaveLAN のそれぞれで計測し, スループットを算出した. 結果を図 4に示す.

図からわかるように Ethernet では指定値と達成値の差は 0% から -0.3% の範囲におさまっており, WaveLAN では ±0.5% の範囲におさまっているというように, ほぼ指定通りに帯域幅の制御が行なわれた.

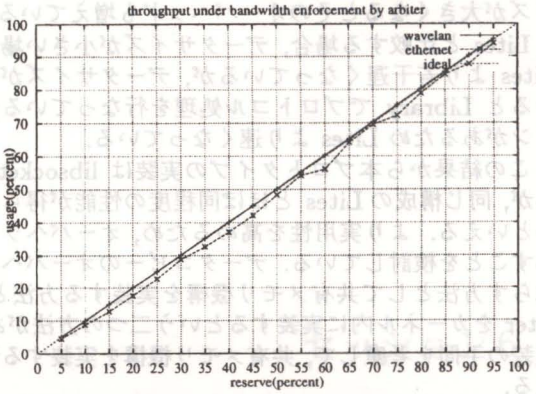


図 4: 資源管理機構によって達成されたスループット
横軸は指定した帯域幅 (%) で縦軸は達成した帯域幅 (%).

## 4.3. パケット送出のオーバヘッド

この項目では UDP パケットを送出するために, アプリケーションが write() を呼んでからデバイスドライバにデータが入るまでの所要時間を計測した. この実験を我々のフレームワークを利用した場合, Lites を利用した場合, libsockets を利用した場合の 3 通りについて 1000 回繰り返し, その平均値を図 5にプロットした. 我々の Library は libsockets をベースにしているため, これと比較を行なう. Lites は RT-Mach 上でネットワークを利用するための通常の方法であり, さらにアプリケーションがネットワークデバイスに直接パケットを書かず, ユーザレベルサーバにパケットを渡すという点で我々のフレームワークの実装と同様の構成のため比較の対象とした.

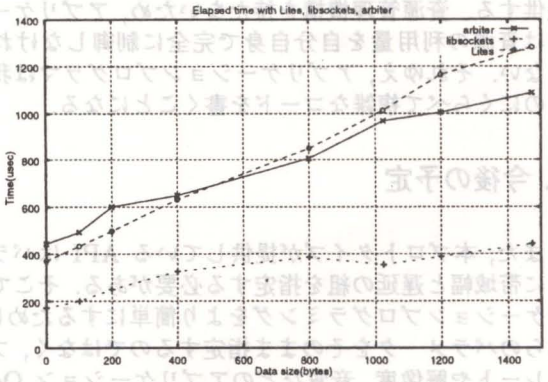


図 5: パケット送出の所要時間.
横軸はパケットの大きさ (bytes) で縦軸は所要時間 (usec).

本プロトタイプはlibsockets と比較すると常に遅くなっている．この原因は以下の二つであった．一つはアプリケーションが Arbiter に固定長の IPC を利用してパケットを渡すパスが libsockets と比べて余分にあるため，時間がかかっている．さらに，パケットを受けとる時にデータコピーを行なうオーバヘッドもあり，パケットサイズが大きくなるとそのオーバヘッドも増えている．

Lites と比較する場合，データサイズが小さい場合には Lites より若干遅くなっているが，データサイズが大きくなると Library でプロトコル処理を行なっている分のゲインがあるため Lites より速くなっている．

この結果から本プロトタイプの実装は libsockets に劣るが，同じ構成の Lites とほぼ同程度の性能が得られているといえる．より実用性を高めるため，オーバヘッドを減らすことを検討している．データコピーのオーバヘッドを減らす方法として共有メモリ機構を実装する方法と，Arbiter をカーネル内に実装するという二つの方法があるが，実装の手間を考慮して，共有メモリ機構を実装する予定である．

## 5. 関連研究

移動ネットワーク環境において適応的アプリケーションのためのフレームワークは幾つか提案されているが，資源管理と通知機構の両方をもつフレームワークは我々の知る限りではまだない．

最も近い関連研究は Jon Inouye' のアプリケーションに特化した適応型システム [2] である．このシステムではビデオプレーヤがネットワークインターフェースの変化に適応的に振舞う。本研究との違いは，このシステムでは帯域幅の実測値をパラメータとしたフィードバックドリブンの適応制御を採用し，さらに資源管理機構をもっていないことである．

他の適応型システムの関連研究として Brian D. Noble と M. Satynarayanan による Odyssey[4] がある．Odyssey は実測ベースの適応制御を採用し，アプリケーションに依存せず，資源のタイプに特化した QoS の適応制御機構を提供している．システムは資源レベルを観測し，関係のあるアプリケーションに予約の参考となる値を提供する．資源管理機構を持たないため，アプリケーションは資源の利用量を自分自身で完全に制御しなければならない．それゆえ，アプリケーションプログラマは我々のものにくらべて複雑なコードを書くことになる．

## 6. 今後の予定

また，本プロトタイプが提供している API はパラメータに帯域幅と遅延の組を指定する必要がある．そこでアプリケーションプログラミングをより簡単にするために，これらのパラメータをそのまま指定するのではなく，フレームレートや解像度，音質などのアプリケーション QoS で指定できるようにし，現在の API を整理／改良する必要がある．

今回は評価に実アプリケーションではなく，テストアプリケーションを用いたが，NV(Network Video) を RT-Mach + 本フレームワーク環境に移植し，適応動作も含めた動作確認は終っているので，今後はこれを用いた評価を行なう予定である．さらに，NV 以外にも http proxy やインターネット電話，telnet などの実アプリケーションを適合させ，これらを用いて評価する予定である．

## 7. まとめ

この原稿では，適応制御のためのフレームワークについて述べた．さらに，RT-Mach マイクロカーネル環境でのフレームワークの実装し，評価を行なった．本研究の特徴はインターフェースの切替を適応動作のトリガとしたこと，'資源管理機構' および '資源量通知機構' が協調動作するフレームワークであることである．

## 参考文献


[1] Tatsumi Hosokawa. PAO: FreeBSD Mobile Computing Package; available at http://www.jp.freebsd.org/PAO/, 1997.

[2] Jon Inouye, Shanwei Cen, Calton Pu, and Jonathan Walpole. System Support for Mobile Multimedia Applications. In *Proceedings of NOSSDAV'97*, pages 143–154, 1997.

[3] Chris Maeda and Brian N. Bershad. Protocol Service Decomposition for High-Performance Networking. In *In Proceedings of SIGOPS'93*, December 1993.

[4] Brian D. Noble, M. Satyanarayanan, Dushyanth Narayanan, James Eric Tilton, Jason Flinn, and Kevin R. Walker. Agile Application-Aware Adaptation for Mobility. In *Proceedings of SOSP-16*, 1997.

[5] Keio-MKng project Keio University SFC. Micro Kernel Next Generation Project; available at http://www.mkg.sfc.keio.ac.jp/, 1997.

[6] S.Keshav. *An Engineering Approach to Computer Networking*. Addison Wesley, 1997.

[7] H. Tokuda, T. Nakajima, and P. Rao. Real-Time Mach: Towards a Predictable Real-Time System. In *Mach Workshop, USENIX Association*, October 1990.