2004－ＳＬＤＭ－114　(15)
2004／3／19

リアルタイムチャネルを用いたEnd-to-Endの実時間通信機構

佐々木貴宏†　　山崎　信行†

† 慶應義塾大学理工学部情報工学科
〒223-8522 横浜市港北区日吉 3-14-1
E-mail: †{sasaki,yamasaki}@ny.ics.keio.ac.jp

あらまし　実時間通信を実現するためには，ネットワークにおける実時間性及び，ノード内のプロトコル処理における実時間性を実現する必要がある．さらに通信を行う際，受信側ではまだプロトコル処理されていないパケットが上書きされないような十分なメモリ量が必要となる．そこで，本研究では，ネットワーク及びプロトコル処理の実時間性を実現し，End-to-Endの実時間通信を実現する．それに際し，必要なプロセッサ資源量と必要なメモリ量を少なくする．リアルタイムチャネルを用いネットワーク資源の予約を行うことで，ネットワークにおける実時間性を実現し，プロトコル処理に必要なプロセッサ資源の予約を行うことで，プロトコル処理における実時間性を実現する．

キーワード　実時間通信，リアルタイムチャネル，プロトコル処理，資源予約

End-to-End Real-Time Communication Mechanism Using Real-Time Channel

Takahiro SASAKI† and Nobuyuki YAMASAKI†

† Keio University
3-14-1 Hiyoshi, Kouhoku-ku, Yokohama, Kanagawa 223-8522 Japan
E-mail: †{sasaki,yamasaki}@ny.ics.keio.ac.jp

Abstract　In order to realize real-time communication, it is necessary to meet the time constraint of network and protocol processing. Furthermore, sufficient memory is needed not to overwrite the packet which is not processed yet. This paper describes a mechanism for meeting the time constraint of network and protocol processing to realize End-to-End real-time communication, and we reduce needed processor resource and memory resource. This mechanism reserves network resource using Real-Time Channel to meet the time constraint of network and reserves processor resource to meet the time constraint of protocol processing.

Key words　Real-Time Communication, Real-Time Channel, Protocol Processing, Resource Reservation

1. はじめに

実時間処理は近年，ネットワークを内包する非常に大きなものにまで要求されるようになり，分散環境において実時間システムを実現する要求が高まってきている．分散実時間システムでは，単一ノード内処理だけでなく，ノード間の処理に必要となる通信においても実時間性を保証することが必要となる．多人数参加型のビデオ会議システムやビデオ監視システム等では，周期的な通信を行い，その通信には実時間性が求められる．

実時間通信を実現するためには，ネットワークにおける実時間性及び，ノード内のプロトコル処理における実時間性を実現する必要がある．ネットワークにおける実時間性を満たせない場合，パケットの遅延，ジッタを保証できなく，通信の実時間性を実現できない．また，ノード内のプロトコル処理における実時間性を満たせない場合，たとえネットワークの実時間性を満たしていたとしても，通信の実時間性を実現することはできない．さらに通信を行う際，受信側ではまだプロトコル処理されていないパケットが上書きされないような十分なメモリ量が必要となる．

本研究では，ネットワーク及びプロトコル処理の実時間性を実現し，End-to-Endの実時間通信を実現する．リアルタイムチャネル[1]を用いネットワーク資源の予約を行うことで，ネットワークにおける実時間性を実現し，プロトコル処理に必要なプロセッサ資源の予約を行うことで，プロトコル処理における実時間性を実現する．それに際し，必要なプロセッサ資源量と必要なメモリ量を少なくする．

既存のアプローチを用いてプロトコル処理を行った場合，受信側で必要なプロセッサ資源量と必要なメモリ量にトレードオフがあり，改善が必要となる．In-Kernel Protocol Processing [2] と呼ばれるアプローチを用いた場合，必要なメモリ量は少なくなるが，必要なプロセッサ資源量は大きくなってしまう．また，User-Level Protocol Processing [3] と呼ばれるアプローチを用いた場合，必要なプロセッサ資源量は理想的となるが，必要なメモリ量は大きくなってしまう．近年の研究では必要なメモリ量に着目しているものはなかったため，User-Level Protocol Processing が優れているとされていた．しかしながら必要なメモリ量を考慮した場合には受信側で User-Level Protocol Processing を採用することに問題がある．一方，送信側では，メモリ量における問題は無いために，User-Level Protocol Processing が優れている．本研究では，受信側での必要な資源量のトレードオフを改善する．受信側で必要なメモリ量を少なくするため In-kernel Protocol Processing を採用する．この場合，必要なプロセッサ資源量が大きくなってしまうが，これを改善するため，送信側でメッセージを分割して送信することにより，必要なプロセッサ資源量を少なくする．

2. 実装対象

この章では，本機構の実装対象である *Responsive Processor*，Responsive Processor の持つ実時間通信規格 *Responsive Link*，*Responsive Processor* 上で稼働するオペレーティングシステム *RT-Frontier* について説明する．

2.1 *Responsive Processor*

並列分散実時間処理用プロセッサである *Responsive Processor* [4] は，ロボットシステムやホームオートメーション，オフィスオートメーション等，分散管理型の実時間制御に必要な様々な機能を内蔵したシステムオンチップである．

Responsive Processor の最大の特徴は，レスポンシブ性を持った通信機構である *Responsive Link* [5] を有していることである．レスポンシブ性とは，外界からの入力に対して反射的な動作を行う性質であるリアクティブ性と，実時間性を兼ね備えた性質のことである．またトポロジフリーであることから，接続形態を選ばない柔軟な実時間分散制御システムの構築を可能にしている．

2.2 *Responsive Link*

Responsive Link では，画像や音声などのデータラインとイベントラインを分離し，各リンクの結合形態は point-to-point である．データラインではパケットサイズを固定長で大きめにしてバンド幅を保証するソフトリアルタイムを実現し，イベントラインではパケットサイズを固定長で小さめにしてイベントのレイテンシを保証しハードリアルタイムを実現する．

Responsive Link を用いた通信では，スイッチのプロセッシングコアに対するインターフェースとして Dual Port Memory(DPM) が用意されている．

2.3 *RT-Frontier Operating System*

RT-Frontier [6] は，過度の資源予約をすることなく，実時間性を保証することを目的とした実時間オペレーティングシステムである．*RT-Frontier* はマイクロカーネルを基本としたサーバベースドアーキテクチャであり，オペレーティングシステム内部のコンポーネントとしては，マイクロカーネル，デバイスドライバ群，アプリケーションプログラムインタフェース (API) 及びシステムサービスを提供するためのサーバ群が存在する．

表 1　パラメータ

M_i $(1 \leqq i \leqq N)$	メッセージ
P_i $(P_1 \leqq P_2 \leqq ... \leqq P_N)$	M_i の周期
S_i	M_i のメッセージサイズ
C_i	M_i のプロトコル処理に要する計算時間

RT-Frontier では，プロトコル処理をノード内処理用インターフェースである datalink_write() 及び datalink_read() を用いて行う．datalink_write() は送信するデータの DPM への書き込みを行う．datalink_read() は，DPM または退避バッファから受信先クライアントのバッファへの書き込みを行っている．ここで退避バッファとは受信したパケットの上書きを防止するためのメインメモリ上の領域である．一定数のパケットを受信するごとに DPM から退避バッファへデータをコピーすることで DPM にある未処理のパケットの上書きを防止する．退避バッファの容量が十分に無いと，パケットの上書きが起こってしまうため，通信を行うにあたり十分なメモリ領域を確保しておく必要がある．なお，退避バッファは，オフラインで静的に割り当てられる．

3. リアルタイムチャネル

本研究では，リアルタイムチャネルを用いる事によってネットワークにおける実時間性を実現する．実時間通信を実現する際に資源予約を伴って確立されたソース，ディスティネーションノード間の通信経路は Ferrari らによりリアルタイムチャネルと定義された．リアルタイムチャネルは以下のパラメータを基に資源予約を行う．

- 遅延
- ジッタ
- メッセージサイズ
- バースト長
- メッセージ送信周期

これらのパラメータを用い通信経路の資源予約を行い，チャネルの確立をすることによって，QoS の保証をする．リアルタイムチャネルを確立することにより，ネットワークにおける遅延，ジッタを保証でき時間予測性が高まる．

4. プロトコル処理

この章では，In-Kernel Protocol Processing 及び User-Level Protocol Processing の概要について述べ，受信側におけるトレードオフを定量化する．定量化するために用いるパラメータを表 1 に示す．

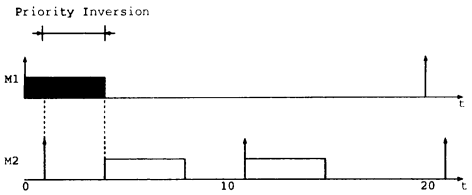

図 1　In-Kernel Protocol Processing を用いたスケジューリング例

4.1 In-Kernel Protocol Processing

4.3BSD のような従来のオペレーティングシステムはこのアプローチを採用している．

このアプローチでは，すべてのメッセージのプロトコル処理がカーネル内でおこなわれ，送受信に FIFO キューを使用する．従って，高い優先度のパケットのプロトコル処理がより低い優先度のパケットのプロトコル処理にブロックされる可能性が生じ，優先度逆転が起こる．例えば，低い優先度のパケットのプロトコル処理をしている最中に，より高い優先度のパケットを受信したとしても，このアプローチを用いた場合には受信した順番にプロトコル処理を行うために，高い優先度の処理はより低い優先度の処理にブロックされてしまう．

周期 20，実行時間 4 の M_1 と周期 10，実行時間 4 の M_2 の 2 つの周期メッセージを受信したときに，このアプローチを用いた場合の受信処理のスケジューリング例を図 1 に示す．

図 1 では，時間 0 で M_1 を受信し，M_1 のプロトコル処理を開始する．時間 1 で M_1 より優先度の高い M_2 を受信するのだが，In-Kernel Protocol Processing では，受信した順番にプロトコル処理を行うため，優先度の逆転が起こり，M_2 の処理は M_1 の処理にブロックされる．

必要な資源量

このアプローチを用いた場合に必要なプロセッサ資源量 (PR_{IKPP}) は，表 1 のメッセージ 1 の周期 P_1 および計算時間 C_i を用いて，式 (1) と表せる．

$$PR_{IKPP} = \sum_{i=1}^{N} \frac{C_i}{P_1} \tag{1}$$

計算時間を C，周期を P とすると，一般にプロセッサ資源量は $\frac{C}{P}$ と表す事ができる．最悪のケースを考えた場合，一番優先度の高い M_1 のプロトコル処理は他の全てのメッセージのプロトコル処理にブロックされてしまう．従って，M_1 のプロトコル処理の実時間性を保証するには，M_1 の周期 P_1 に全てのメッセージのプロトコル処理を行えるプロセッサ資源量が必要となる．しかしながら，このような資源割り当てを行った場合，割り当てられた資源を使い切らないサイクルが生じるためにかなりの無駄となる．

このアプローチを用いた場合，P_1 と他のメッセージの周期の差が大きいほど無駄な資源予約を行う．

一方，必要なメモリ量 (MR_{IKPP}) は表 1 のメッセージサイズ S_i を用いて，式 (2) と表せる．

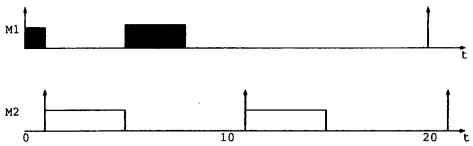

図 2　User-Level Protocol Processing を用いたスケジューリング例

$$MR_{IKPP} = \sum_{i=1}^{N} S_i \tag{2}$$

ほぼ同時に全てのメッセージを受信するような最悪のケースを考えた場合，全てのメッセージのメッセージサイズの総和のメモリが必要となる．

4.2 User-Level Protocol Processing

このアプローチでは，すべてのプロトコル処理がユーザレベルライブラリで行われる．ユーザレベルライブラリが直接ネットワークデバイスにアクセスするために，プロトコル処理の優先度はネットワークアプリケーションの優先度で実行される．

周期 20，実行時間 4 の M_1 と周期 10，実行時間 4 の M_2 の 2 つの周期メッセージを受信したときにこのアプローチを用いた場合の受信処理のスケジューリング例を図 2 に示す．

図 2 では，時間 0 で M_1 を受信し，受信処理を開始する．時間 1 で M_2 を受信し，M_2 のデッドラインのほうが M_1 のデッドラインより近いために，M_2 の受信処理が行われる．時間 5 で M_2 の受信処理が完了し，M_1 の受信処理が再開され，時間 8 で受信処理が完了する．時間 11 で 2 サイクル目の M_2 を受信し，その受信処理は時間 15 で完了する．

必要な資源量

このアプローチを用いた場合に必要なプロセッサ資源量 (PR_{ULPP}) は表 1 式の周期 P_i および計算時間 C_i を用い (3) と表せる．

$$PR_{ULPP} = \sum_{i=1}^{N} \frac{C_i}{P_i} \tag{3}$$

それぞれのメッセージのプロトコル処理はそれぞれのメッセージの優先度に基づき行われるため，それぞれのメッセージに必要なプロセッサ資源量の総和が必要となる．

一方，必要なメモリ量 (MR_{ULPP}) は表 1 の周期 P_i およびメッセージサイズ S_i を用い式 (4) と表せる．

$$MR_{ULPP} = \sum_{i=1}^{N} \left\lceil \frac{P_N}{P_i} \right\rceil \times S_i \tag{4}$$

優先度通りにプロトコル処理を行った場合，プロトコル処理が行われるのが一番遅れるのは，周期が一番大きい M_N であり，最悪のケースを考えた場合，M_N のプロトコル処理が行われるまでに M_i は，$\left\lceil \frac{P_i}{P_1} \right\rceil$ 回受信する．従って，この回数とメッセージサイズの積の総和のメモリが必要となる．

このアプローチを用いた場合，P_N と他のメッセージの周期との差が大きいほど必要なメモリ量が大きくなる．

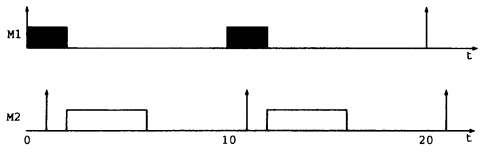

図 3　本研究のアプローチを用いたスケジューリング例

5. 設　　計

この章では，既存のアプローチを用いた場合の受信側でのトレードオフを改善するための本研究の設計方針および設計について述べる．

5.1 設計方針

User-Level Protocol Processing を用いた場合，プロセッサ資源量は理想的だが，受信側で必要なメモリ量が大きくなってしまうため，本研究では必要なメモリ量を少なくするために In-Kernel Protocol Processing を採用する．ただし，In-Kernel Protocol Processing を用いた場合，必要なプロセッサ資源量が大きくなってしまう．この理由は，割り当てられたプロセッサ資源を使い切らないサイクルが生じるためである．この問題を改善するため，本研究では，送信側でメッセージを分割して送信する．これにより，受信側で全てのサイクルで均一にプロトコル処理が行われ，プロセッサ資源の無駄をなくす．分割方法は，全てのサイクルで均一に処理を行うために通信の周期の最小単位 (P_{min}) で分割する．従って，メッセージのパケット数を $Npkt$ とすると，P_{min} に $\left\lceil \frac{Npkt}{P_{min}} \right\rceil$ のパケットを送信する．

本研究のアプローチを用いた場合のプロトコル処理のスケジューリング例を図 3 に示す．この例では，P_{min} は 10 である．

この例においても，周期 20，実行時間 4 の M_1 と周期 10，実行時間 4 の M_2 の 2 つの周期メッセージを受信している．メッセージを送信側で P_{min} で分割するため，M_1 は 2 回に分割して送信し，M_2 は分割しないで送信する．時間 0 で M_1 のパケットを受信し受信処理を開始する．時間 1 で M_2 を受信し，時間 2 から M_2 の受信処理を開始する．時間 10 で M_1 の残りのパケットを受信し受信処理を開始する．時間 11 で 2 サイクル目の M_2 を受信し，時間 12 から M_2 の受信処理を開始する．本研究のアプローチを用いた場合，分割して受信処理が行われるためにすべてのサイクルで均一に受信処理を行うことになる．これにより，In-Kernel Protocol Processing 用いた場合に比べ必要なプロセッサ資源量は少なくなる．

必要な資源量

本研究のアプローチを用いた場合に必要なプロセッサ資源量 ($PR_{Our-Approach}$) は，通信の周期の最小単位 P_{min} および C'_i を用いて式 (5) と表せる．

$$PR_{Our-Approach} = \sum_{i=1}^{N} \frac{C'_i}{P_{min}} \qquad (5)$$

C' は，P_{min} に受信する $\left\lceil \frac{Npkt}{P_{min}} \right\rceil$ 個のパケットのプロトコル処理に要する計算時間である．本研究のアプローチを用いた場合，

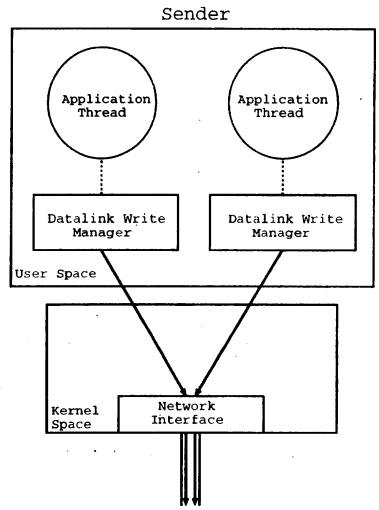

図 4　本機構のアーキテクチャ:送信側

P_{min} に受信するパケット数は全てのサイクルで均一となり，そのパケットを P_{min} 内にプロトコル処理するプロセッサ資源が必要となる．

一方，必要なメモリ量 ($MR_{Our-Approach}$) は表 1 の周期 P_i，S_i および通信の周期の最小単位 P_{min}を用い 式 (6) と表せる．

$$MR_{Our-Approach} = \sum_{i=1}^{N} \frac{S_i}{P_i} \times P_{min} \qquad (6)$$

本研究のアプローチを用いた場合，受信するパケット数は全てのサイクルで一定であり，受信したパケットは P_{min} 内に処理されるため，P_{min} に受信するパケット分だけのメモリ量が必要となる．

5.2 設　　計

本研究のアプローチを用いた場合，送信側でメッセージを分割してプロトコル処理する機構が必要となる．これを実現する機構を図 4 に示す．

アプリケーションスレッドは，プロトコル処理以外にも行う処理はあるために，P_{min} 毎に分割してプロトコル処理を行う事はできない．そこで本研究では，プロトコル処理のみを行うスレッドをメッセージ毎に作成し，そのスレッドが P_{min} に $\left\lceil \frac{Npkt}{P_{min}} \right\rceil$ 個のパケットのプロトコル処理を行う．このスレッドのことを本研究では Datalink Write Manager(DLWM) と呼ぶ．

プロトコル処理をアプリケーションスレッドとは分離して行うために，アプリケーションスレッドと DLWM で通信データの受け渡しをする必要が生じる．この通信データの受け渡しを本研究では，ダブルバッファリングを用いて行う．バッファ 1 にアプリケーションスレッドがデータを書き込んでいる時はバッファ 2 のデータの送信処理を DLWM が行う．一方，バッファ 2 にアプリケーションスレッドがデータを書き込んでいる時は，バッファ 1 のデータの送信処理を DLWM が行う．

一方，図 5 に受信側における本機構のアーキテクチャを示す．

本機構では，受信側で In-Kernel Protocol Processing を採

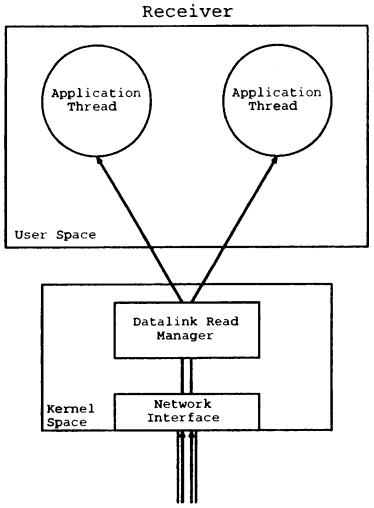

図 5　本機構のアーキテクチャ:受信側

用するため全てのメッセージのプロトコル処理を行うスレッドがカーネルレベルにある．本研究では，このスレッドの事を Datalink Read Manager(DLRM) と呼ぶ．DLRM はパケットを受信した順番にプロトコル処理を行う．

ここで本機構を用いた通信の流れについて説明する．まず受信側のアプリケーションスレッドがリアルタイムチャネル確立要求を出す．リアルタイムチャネル確立要求が出されたら，その通信に必要なプロセッサ資源を DLRM に割り当てる．この時に，このメッセージのプロトコルを行う余裕が無い場合，この通信要求はリジェクトする．

リアルタイムチャネルが確立したら，送信側のアプリケーションスレッドは DLWM を作成する．ここで，DLWM に割り当てるプロセッサ資源の余裕が無い場合，この通信要求はリジェクトされる．DLWM に十分なプロセッサ資源を割り当てる事ができたなら，送信側はメッセージの分割送信を開始する．

リアルタイムチャネルを確立しネットワーク資源の予約を行う事で，ネットワークにおける実時間性を実現し，DLRM 及び DLWM にプロセッサ資源を割り当てる事で，プロトコル処理における実時間性を実現する．

6. 評　　価

この章では，既存のアプローチを用いた場合と本研究のアプローチを用いた場合で必要なプロセッサ資源量及び必要なメモリ量の比較を行う．

6.1 評価環境

Responsive Processor はプロセッシングコアである SPARClite の動作速度及びバスクロックの速度を動的に変更することが可能である．そこで，評価に際しては，それぞれを表 2 の様に設定した．

6.2 予備評価

ここまで，必要なプロセッサ資源量を定量化するために用いてきたプロトコル処理に要する計算時間 C は，アプリケーションスレッド側では分からないために，オペレーティングシステムが割り出す必要がある．プロトコル処理に要する計算時間はパケット数で表すことができ，本研究では，実装対象である *RT-Frontier* の基本性能を測ることにより，計算時間を算出した．

表 2　*Responsive Processor* に関する設定

SPARClite の動作速度	125 MHz
バスクロックの速度	25 MHz

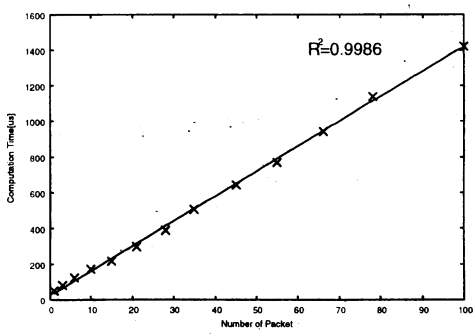

図 6　プロトコル処理に要する計算時間

表 3　メッセージのパラメータ

	周期	メッセージサイズ
メッセージ 1	30ms	15KB
メッセージ 2	[10, 100]ms	15KB

図 6 に，*RT-Frontier* におけるパケット数とプロトコル処理に要する計算時間の関係を示す．

図 6 より，プロトコル処理に要する計算時間は，パケット数におおよそ比例していることがわかる．ここで，パケット数 $Npkt$ は，メッセージサイズ S と *Responsive Link* のデータリンクのペイロードサイズである $RL_{d_{payload}}$ を用いて式 (7) のように表すことができ，プロトコル処理に要する計算時間 C は，式 (7) で導いた $Npkt$ を用いて式 (8) と表せる．

$$Npkt = \left\lceil \frac{S}{RL_{d_{payload}}} \right\rceil \tag{7}$$

$$C = \alpha \times Npkt + \beta \ [\mu s] \qquad \begin{cases} \alpha = 14 \\ \beta = 21 \end{cases} \tag{8}$$

本研究では，この関係を用いて必要なプロセッサ資源量を算出し，割り当てを行った．

6.3 必要な資源量の比較

本研究では，MPEG などの動画の通信を行った場合にそれぞれのアプローチでどれぐらいの資源が必要となるかの比較を行った．2 種類の画像データを周期的に受信するとし，これらのメッセージのパラメータを表 3 とした．

In-Kernel Protocol Processing，User-Level Protocol Processing を用いた場合，それぞれのメッセージの周期の差が変化するにつれて，必要な資源量が大きくなっていくため，メッ

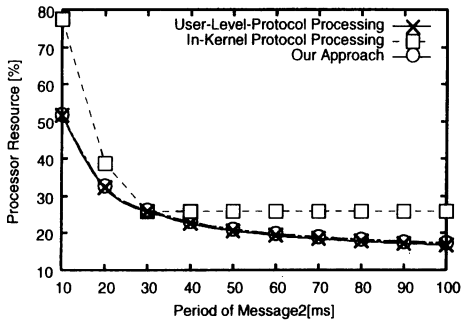

図 7　必要なプロセッサ資源量の比較

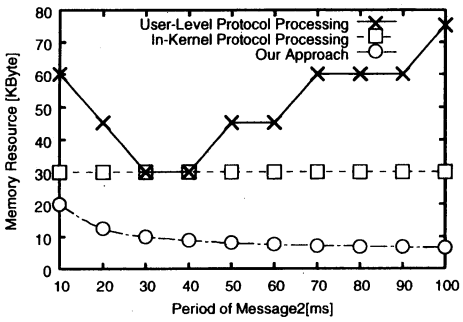

図 8　必要なメモリ量の比較

セージ 1 の周期，メッセージサイズ及びメッセージ 2 のメッセージサイズは固定とし，メッセージ 2 の周期を 10ms から 100ms に変化した時の比較を行った．また，P_{min} は 10ms とした．

それぞれのアプローチで必要なプロセッサ資源量を示したものを図 7 に示す．

図 7 より，User-Level Protocol Processing はメッセージ 2 の周期が大きくなるにつれて下がっていく．In-Kernel Protocol Processing を用いた場合，平均して，User-Level Protocol Processing を用いた場合の 1.3 倍のプロセッサ資源量が必要となる．さらに，メッセージ 2 の周期が 10ms のときには User-Level Protocol Processing の 1.5 倍ものプロセッサ資源が必要となる．これは，In-Kernel Protocol Processing は最小周期にすべてのメッセージの受信処理を行う最悪のケースを想定して資源予約を行うからである．メッセージの周期が 30ms の時はメッセージ 1 の周期と等しいために User-Level Protocol Processing と同じ値となる．メッセージ 2 の周期が 30ms より大きい場合，最小周期がメッセージ 1 の周期の 30ms となるために一定となる．

本研究のアプローチを用いた場合，User-Level Protocol Processing と比べてみると，平均して，1.02 倍のプロセッサ資源が必要となり，ほぼ理想的と言える．

それぞれのアプローチで必要なメモリ量を示したものを図 8 に示す．

図 8 より，User-Level Protocol Processing を用いた場合，In-Kernel Protocol Processing を用いた場合に比べ，平均して，1.7 倍のメモリ量を必要とする．User-Level Protocol Processing を用いた場合，メッセージ 1 とメッセージ 2 の周期の差が大きくなるにつれて，必要なメモリ量が大きくなっていく．この事より，周期が似通ったメッセージを受信する場合には問題はないが，周期が様々なメッセージを受信する場合には必要なメモリ量に問題がある事がわかる．

本研究のアプローチを用いた場合，必要なメモリ量は平均して，User-Level Protocol Processing の 0.2 倍，In-Kernel Protocol Processing の 0.3 倍とかなりのメモリを削減できることがわかる．

7. ま と め

本研究では End-to-End の実時間通信を実現し，必要なプロセッサ資源量及びメモリ量を少なくする機構について述べた．

リアルタイムチャネルを用いネットワーク資源の予約を行いネットワークにおける実時間性を実現し，プロトコル処理に要するプロセッサ資源を予約することで，End-to-End な実時間通信を実現した．そして，プロトコル処理に既存のアプローチを用いた場合の資源量におけるトレードオフを改善するため，受信側では In-Kernel Protocol Processing を採用することにより必要なメモリ量を少なくし，送信側でメッセージを分割して送信する機構を用いる事により必要なプロセッサ資源を少なくした．評価の結果，本研究のアプローチを用いた場合，既存のアプローチを用いた場合に比べ必要なメモリ量を少なくすることが出来ることが確認できた．

本研究では，メモリ領域をオフラインで静的に割り当てていたのだが，今後の課題として，リアルタイムチャネルを確立する際に動的にメモリ領域を確保する機構を実装する必要がある．

謝辞　本論文の研究は，文部科学省の科学技術振興調整費の支援による．また本研究の一部は，科学技術振興機構 CREST の支援による．

文　献

[1] D. Ferrari and D. Verma: "A Scheme for Real-Time Channel Establishment in Wide-Area Networks", IEEE Transactions on Selected Areas in Communications (1990).

[2] S. Leffler, M. McKusic, M. Karels and J. Quarieran: "The Design and Implementation of the 4.3BSD UNIX Operating System", Addison Wesley (1989).

[3] C. Maeda and B. Bershad: "Protocol Service Decomposition for High-Performance Networking", Proceedings of the 14th Symposium on Operating Systems Principles (1993).

[4] N. Yamasaki: "Design and Implementation of Responsive Processor for Parallel/Distributed Control and Its Development Environments", Journal of Robotics and Mechatronics, **13**, 2, pp. 125–133 (2001).

[5] "Responsive Link(IPSJ-TS 0006:2003)", http://www.itscj.ipsj.or.jp/ipsj-ts/02-06/toc.htm.

[6] H. Kobayashi and N. Yamasaki: "An Integrated Approach Implementing Imprecise Computations", IEICE Transactions on Information and Systems, **E86-D**, pp. 2040–2048 (2003).