組込みシステム向け TCP/IP プロトコルスタックの開発と評価

大塚　雄三，並木　美太郎
東京農工大学大学院　工学研究科

近年，組込みシステムの分野においても TCP/IP プロトコル通信を提供する必要が高まっている．TCP/IP のプロトコルスタックとその API として BSD ソケットが有名ではあるが，ハードウェア資源については考慮されていない点で組込みシステムには向かない．しかし，組込みシステムでは使用できるメモリ量や CPU パワーなどハードウェア資源に対する制約が厳しいという点を考慮に入れた設計が必要である．本研究では，プロトコルスタックでのデータコピーをなくし，またバッファ領域を減らすことで，メモリ使用量を減らし CPU 利用効率向上させるという目標を達成させる．また，本プロトコルスタックでは BSD ソケットとは異なるコピーレスに適した独自の API を設計し，送受信の際のバッファへのアクセスや手順などを定めた．以上の設計に基づき組込みボードに実装しその評価を行った．その結果，データコピーを行わないことにより最大 6 割ほどの速度向上が確認できた．

A development and evaluation of TCP/IP protocol stack for embedded systems

OTSUKA Yuzo, NAMIKI Mitaro
Graduate School of Engineering, Tokyo University of Agriculture and Technology.

Recently, a requirement to support TCP/IP protocols is increasing in the feild of embedded systems. Although BSD socket is famous, as the protocol stack and the API of TCP/IP, it's unsuitable for embedded systems from view of hardware resources. However, a design of protocol stacks for hardware resources such as memory using and CPU power is needed. A target of this research is reducing memory using and CPU usage by zero copy in a protocol stack. Based on this design, the protocol stack was implemented on a board, and the evaluation was performed. As the result, the performance was improved 60%.

1．はじめに

近年，プリンタやコピー機など LAN 接続されて PC 等と通信する機器や，情報家電をネットワークで結び，ホームネットワークを構築するようになってきた．そこで，組込みシステムの分野においても TCP/IP プロトコルによる通信を提供する必要が高まっている．

本論文では，ハードウェア資源に対する制約が厳しいという組込みシステムの特性を考慮した TCP/IP プロトコルスタックの設計について述べる．

現在，TCP/IP のプロトコルスタックとその API として広く利用されている通信プロトコルスタックとして BSD ソケットがある．この BSD ソケットはハードウェア資源について考慮されていないという点において組込みシステムには向かない．具体的には，受信用バッファとして BSD ではソケットあたり 32KB(default 値)のメモリを必要とし，送信時にはアプリケーションとプロトコルスタック間で最低 1 回，受信時にはプロトコルスタックとソケットバッファ，ソケットバッファとアプリケーション間の 2 回のデータコピーが行われている．その他にも，TCP/IP 以外のプロトコルにも柔軟に対応することができるよう設計となっており，その実装もスループットやネットワーク輻輳を回避するような通信機能を重視したものとなっている[6].

しかし，組込みシステムでは使用できるメモリ量や CPU パワーなどハードウェア資源に対する制約が厳しいという点を考慮に入れた設計が必要である．

このような背景から，本論文ではプロトコルスタックでのデータコピーをなくし，またバッファ領域を減らすことで，メモリ使用量を減らし，CPU 利用効率向上させることを目標とする．

本プロトコルスタックは基本的にゼロコピーで動作する．ただし，ハードウェアが DMA をサポートしているという条件付である．DMA をサポートしておらず，デバイスからの転送時にコピーが起こる場合には，ハードウェアの仕様上回避できないため，1 回コピーが起き，1 コピーとして動作する．

2．概要

本プロトコルスタックの大きな特徴は，データコピーを行わないことである．従来のプロトコルスタックの処理手順は，まず，受信したパケットをデバイスからプロトコルスタックへと渡す．プロトコルスタックでは，そのパケットのヘッダを解析し，行う処理を決定し，多くの場合はさらに上位のプロトコル，またはアプリケーションにデータをコピーする．送信の際は，まずアプリケーションなどで作成したデータをプロトコルスタックのバッファへコピーする．そして，そのコピーされたデータにヘッダなどを付与し，パケットとして完成させ，デバイスへ渡して送信する．この結果，従来のプロトコルスタックではアプリケーションとプロトコルスタック間で少なくとも 1 回のコピーが必ず起こる．ここで，デバイスがイーサネットの場合には，1 つのパケットを送信するのに，最大で 1460Byte のデータコピーが起こる．本プロトコルスタックでは，このコピーを回避し，そのメモリコピーによる CPU 処理や，バッファのためのメモリを節約できるという利点を持つ．

また，本プロトコルスタックは通信端点としての利用を前提としており，ルーティング機能は実装しない．通常，通信端点として動作するアプリケーション，たとえばメーラや Web ブラウザ，ファイル転送などにルーティング機能は必要ない．とりわけ，組込みシステム向けということで，限定された用途で利用されることを考慮に入れると必要ない．もし，ルータとして利用したい場合には，別途ルータに適した設計を行うべきである．このルーティングは TCP の処理の中でも大きな割合を占める部分であり，これを避けることでプロトコルスタック内での処理のオーバヘッドを軽減できる．

本プロトコルスタックでは，独自の API を採用している．これは，従来のソケット API では，ユーザが確保した任意のバッファとデータをコピーするように設計されているため，ゼロコピーで動作させることが困難であると判断したためである．

3．本プロトコルスタックの全体構成

本プロトコルスタックは，下図 1 に示したようにユーザインタフェース部，バッファ管理部，プロトコル部，送受信部からなっている．

次章で説明するが，送信用バッファはそれぞれのアプリケーションが持っている．受信用バッファはプロトコルスタックが持っており，それをアプリケーションから直接参照する形になっている．

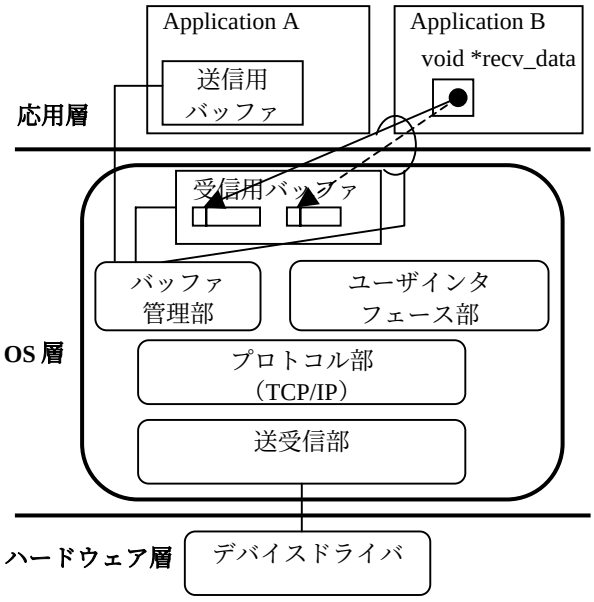

図 1　全体構成図

4．コピー回避の手法

本プロトコルスタックでは，以下の方法によりこのメモリコピーを回避する．

(1)送信用バッファはそれぞれのアプリケーション領域に持たせ，受信用バッファはプロトコルスタックに持たせる．

(2)アプリケーションから上記送受信用バッファへ直接アクセスさせる．

(3)上記バッファは，データ読み書きの際にパケットヘッダのための領域分空けておき，オフセットすることにより，バッファにパケットのデータ構造を持たせ，再配置を必要とせず直接デバイスとのやりとりを可能にする．

4.1. 送受信バッファの位置

従来のプロトコルスタックでは，プロトコルスタック内部に送受信のためのバッファを持ち，アプリケーションでもバッファを用意していた．

しかし，本プロトコルスタックではコピーレスな動作を基本としているため，送受信用バッファバッファをそれぞれアプリケーションとプロトコルスタックと 2 カ所に持つ必要はない．送受信用バッファはコネクションに対し，それぞれ 1 つだけ存在し，そのバッファへ直接アプリケーションからアクセスが行われ，また直接デバイスを通して送受信が行われる．

本プロトコルスタックでは，送信用バッファはそれぞれのアプリケーションに，受信用バッファはプロトコルスタックに持たせる．

送信用バッファをアプリケーションに持たせる理由は，それぞれのアプリケーションが必要と思われるバッファサイズを，その設計時に見積もって確保することで適切なサイズで確保でき，バッファ領域を減らすことができると考えたためである．

また，受信用バッファをプロトコルスタックに持たせる理由は，受信したパケットはヘッダの解析を行わないと，どのアプリケーションに宛てたものだかを判別することが出来ない．そのため，受信したパケットは，一度プロトコルスタックに格納し，それを解析して宛先のアプリケーションを判断する．ここで，コピーを行わないことが前提であるので，このプロトコルスタック内に格納した受信データを宛先のアプリケーションにアドレスを限定し直接参照させる．

4.2. アプリケーションからのアクセス

アプリケーションから受信用バッファに任意にアクセスすることはできない．また，送信用バッファに関しても，アプリケーション領域にあるが，管理はプロトコルスタックが行っておりアプリケーションからはブラックボックス化されており，無作為にアクセスしてはならない．これら送受信用バッファにアクセスする際には，後述の API を用い限定されたアドレスに対してのみ，直接アクセスすることでコピーを回避する．

4.3. パケットのデータ構造を持つバッファ

この送受信用バッファは，アプリケーションから直接アクセスされる．また，コピーなしにこのままここからデバイスを通して送信，また受信が行われる．

そのため，この送受信用バッファはすでにパケットとしての構造を持ち，ヘッダ，データなどが連続してパケットとして格納されている．アプリケーションからの読み書きの際は，このヘッダのための領域を避ける必要がある．これは，ヘッダ分をオフセットした場所を後述の API が指し示すため，アプリケーションプログラマにとってデータの読み書きの際のパケット構造やヘッダ領域のオフセットなどバッファの中はブラックボックスとなっている．

こうして，送受信用バッファがパケットとしてのデータ構造を持っていることで，直接デバイスに渡して送受信が可能となっている．

5．本プロトコルスタックの API

本プロトコルスタックでは，下表 1 に示した API を提供している．

本プロトコルスタックでは，独自の API を採用している．これは，従来のソケット API では，ユーザが確保した任意のバッファとデータをコピーするように設計されているため，ゼロコピーで動作させることが困難であると判断したためである．

また，その際に従来のソケット API の冗長な

表 1　提供している API

API(C 言語仕様)	説明
TypeSOCK *sk = **cre_sock**(ipaddr *saddr, int sport, ipaddr *daddr, int *dport, char *protocol, int opt);	ディスクリプタを生成する
void **del_sock**(TypeSock *sock);	ディスクリプタを削除する
int len = **ask_rbuf**(void **p);	受信したパケットのデータのアドレスとサイズを得る．p にアドレスを入れて返される．
void **read_done**(void *p, int len);	データの読み出し終了をプロトコルスタックに通知する．
int err_code = **set_sbuf**(void *buf_addr, int buf_len);	送信用バッファを登録する．
int len = **ask_sbuf**(void **p);	送信したいデータの書き込みアドレスとサイズを得る．p にアドレスを入れて返される．
int err_code = **send**(void *p, int len, int opt);	送信する．

手順を省いた．ソケット API では前準備として socket(), bind(), listen(), accept()や，その引数に AF_INET など，また確保した構造体 sockaddr_in の初期化などのおきまりの手順が必要である．これらは，TCP での通信を行う場合には，ほとんどがおまじないとなっており冗長なものとなっている．本システムの cre_sock()は，従来のソケット API の socket(), bind,listen(), accept(), connect()を含んでいる．引数として，送信元，送信先のアドレス，プロトコルを指定する．前準備は，cre_sock()と，送信の場合にはバッファの確保とその登録 set_sbuf()だけである．

また，従来のソケットにはない read_done()という API が用意されている．従来のプロトコルスタックでは，パケットが到着すると，受信用領域に格納され，解析が行われる．その結果，宛先のコネクションが決定すると，そのソケットバッファにデータがコピーされ，受信用領域からは解放される．しかし，本システムでは解析が行われ，宛先のコネクションが決定した後でも，パケットを受信用領域に保持しておく．そして，アプリケーションからこの領域に保持してあるパケットのデータを直接参照させる．こうすることでコピーを回避しているものの，このパケットデータ領域をいつ解放してよいのかを，プロトコルスタックからは判断することができない．そこで，アプリケーションが不要となったことを明示的に通知する必要がある．そのための API が read_done()である．

本システムでは，データの読み書きの前には，どこを読み書きするべきかを API ask_rbuf(), ask_sbuf()により得る必要がある．受信データを読み出す場合は当然ながら，送信したいデータを書き込む場合にもそのアドレスが限定される．これはコピー回避とバッファのデータ構造に関連しており，理由は次の章で説明する．

6．内部設計

内部設計として，送受信用バッファにどうパケットが格納されているかについて述べる．また，本プロトコルスタックの特徴であるコピーレスをどう実現させているかを，送受信の際のバッファへのアクセスを中心に述べる．

6.1 カプセル化に伴うバッファ中のデータ構造

本プロトコルスタックの送信用バッファは，アプリケーションで確保する．データコピーを行わないので，このバッファから直接デバイスへ転送し送信する．デバイスへの転送方法はハードウェアに依存するが，転送元のパケットは連続した領域に格納されていることが望ましい．ここで，送信したいデータには前後に色々なヘッダやフッタがつく，つまりカプセル化される．具体的な例として，送信するパケットが TCP のプロトコルでデバイスがイーサネットの場合には，送信したいデータの前には 20Byte の TCP ヘッダが付くことによって TCP パケットとなり，その TCP パケットは 20Byte の IP のヘッダが付き，さらにその IP パケットが 14Byte のイーサのヘッダによってカプセル化されイーサのパケット(フレーム)となる．つまり，もともと送信したかったデータの前には，54Byte (20+20+14)ものヘッダがつけられる．(図 2)

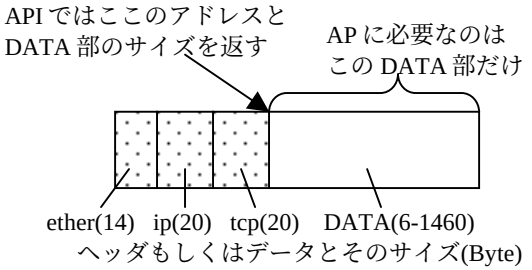

図 2　イーサフレームのカプセル化

最終的にデバイスへ転送するイーサのパケットを連続した領域に書き込むことを考えると，空き領域の先頭からヘッダのために 54Byte 進めた場所からデータを書き込む必要がある．(図 3) この 54Byte というのはプロトコルやデバイスによって変わる．

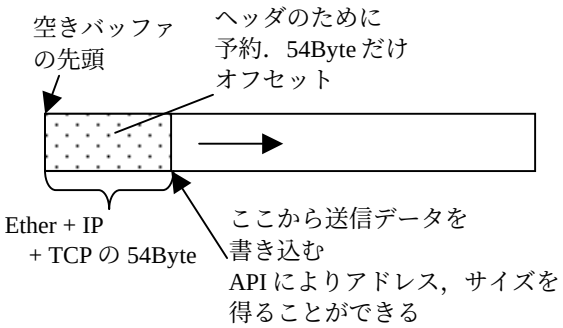

図 3　送信用バッファへの書込み

アプリケーションプログラマがこのヘッダなどを考慮しつつデータを読み書きすると負担が大きい．そこで，アプリケーションから送受信用バッファへアクセスする際は API を用いることで，プロトコルスタックがバッファ中のヘッダやフッタなどの領域を取り除いたデータ部分へのアドレスを示す．アプリケーションプログ

ラマは，その API からの返り値をだけを利用すればよく，バッファの中がどうなっているかを知る必要はない．むしろ，送信用のバッファはアプリケーション領域にあるからといって，無作為にアプリケーションが自ら場所を指定して書き込むなどしてはならない．

6.2. 送受信用バッファ

受信用バッファの確保は，プロトコルスタックの初期化，またはカーネルの初期化で行われる．これを固定長のブロックごとに分割して利用する．このブロックは，受信したパケット 1 つを保持するバッファ本体，そのバッファのサイズ，リストとして連結するためのポインタから構成されている．バッファ本体のサイズは基本的に MTU の値で設定される．したがって，受信用バッファはパケットのための領域を固定長としリストで管理されている．その理由は，受信したパケットデータの削除はアプリケーションの動作に影響し，複数のコネクションがある場合には特に，そうでなくても TCP では到着順所が保証されていない．そのため，到着したパケットデータから順に削除されず，どういった順番で削除されるかが未知となるからである．そこで，これら到着パケットの入れ替えなどが頻繁に起こることを想定したためである．

送信用バッファの確保は，アプリケーションが行う．それを API でプロトコルスタックに登録し管理を委ねる．送信用バッファはリングバッファとして利用する．パケットそれぞれの領域は可変長とする．受信用のものと異なり，リングバッファとし可変長とすることで，バッファを効率的に利用できるようにした．送信の際には，送信用バッファはそれぞれのアプリケーションで確保するため，このバッファを利用するコネクションは 1 つである．そして，アプリケーションからの send()により送信パケットがリングバッファに追加されていく．そのデータは送信後も，すぐには削除されず再送のために残されるが，TCP の ACK が返ってきたら，返ってきたところまでを削除する(図 4)．そのため，先入れ先出しの概念が成り立つためリングバッファとの相性がよい．そこで，バッファの利用効率を考え，リングバッファとし，パケットの個々の領域を可変長として扱うよう設計した．

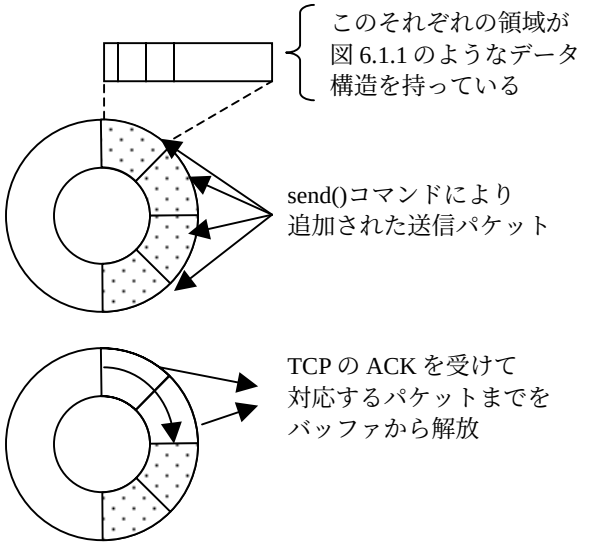

図 4　送信用リングバッファへの
パケットデータの追加と削除

6.3. 受信の流れ

次に本プロトコルスタックでの受信の際の内部処理について説明する．

受信は下表 2 の手順で処理される．

表 2　受信の流れ

1.パケット受信
2.ヘッダの解析
3.受信データの位置情報を要求(ask_rbuf())
if 受信データがある
4.データへのアドレス，サイズを示す
5.データの読出し
6.読出し終了を通知(read_done())
7.領域解放

パケットが受信されるとデバイスを通して，プロトコルスタック中の受信バッファへパケットが格納される(表中 1)．プロトコルスタックではヘッダを解析し(表中 2)，それに従って処理を進める．多くの場合は，さらに上のプロトコル，アプリケーションにデータに渡すことになる．

ここで，このデータを渡す方法として，本プロトコルスタックではデータをコピーするのではなく，受信したパケットをそのままプロトコルスタック内の受信バッファに留めておき，アプリケーションにはそのプロトコルスタック中のバッファにあるデータの先頭アドレスを渡す(表中 4)．そして，アプリケーションはプロトコルスタック中のデータを直接参照する(表中 5)．その後，アプリケーションから読み終わったことを示す read_done()を受けて(表中 6)，パケットデータ領域を解放する(表中 7)．

この挙動を次ページ図 5 に示す．

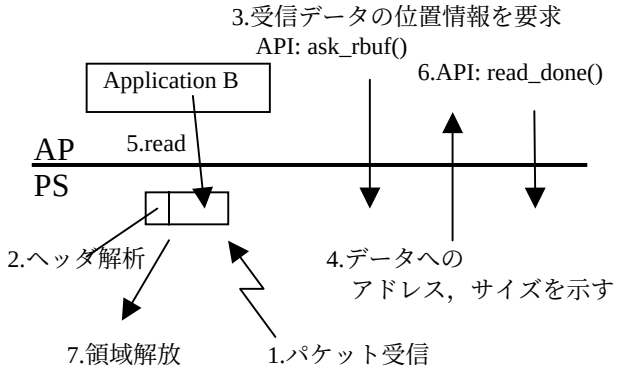

図 5　受信の流れ

プロトコルスタック内のバッファ，つまりカーネル空間のメモリ領域に対して，通常のユーザプログラムから直接アクセスするということは，保護の面で問題となる．しかし，組込みシステム向けということで，保護よりも性能を重視することにした．

この方法により，従来では受信の際にイーサの場合で最大 1460Byte のメモリコピーが 2 回起きていたが，本方式ではポインタのコピー，つまり 4Byte 程のコピーで済む．

6.4. 送信の流れ

送信は下表 3 の手順で行われる．

表 3　送信の流れ

0.バッファを確保し，登録する(set_sbuf())
1.送信データの書込み位置情報を要求(ask_sbuf())
if 空き領域がある
2.空き領域からヘッダの分を取り分けて， アドレス，サイズを示す
3.データを書込む
4.送信要求(send())
5.ヘッダの付与などの処理を行い，送信する

送信のためのバッファ領域は，アプリケーションで確保し，それをプロトコルスタックに登録する(表中 0)．以降，この送信用のバッファはプロトコルスタックで管理するので，その中がどうなっているかはアプリケーションプログラマにとってはブラックボックスとなっている．

次に，アプリケーションからデータを送信する際には，まず送信用のバッファのどこへ送信データを書き込むべきかを，ask_sbuf()を用いプロトコルスタックに尋ねる(表中 1)．プロトコルスタックでは，バッファの空き領域からヘッダのための領域を取り分けておき，パケットのデータ部分の先頭にあたるアドレスを示す(表中 2)．プロトコルスタックから示された書き込む先頭アドレス，空き領域のサイズに従って，データを書き込む(表中 3)．データの書き込みが終わったら，先頭アドレスと書き込んだサイズを send()を用いプロトコルスタックに通知し，送信してもらう(表中 4)．プロトコルスタックでは，そのデータにヘッダを付与し，パケットとして完成させ，デバイスへ渡して送信する(表中 5)．

この挙動を下図 6 に示す．

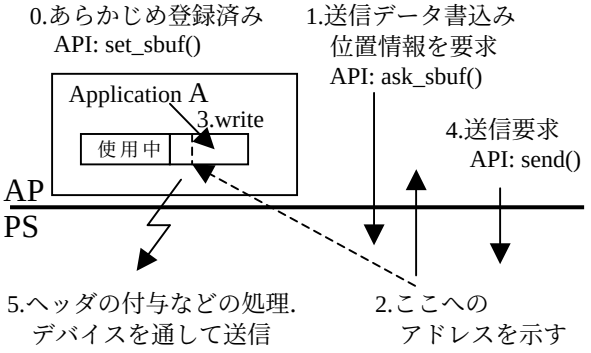

図 6　送信の流れ

この方法により，送信の場合も受信の場合同様，データコピーが起こらないので，従来ではイーサの場合で最大 1460Byte のメモリコピーが，本方式では 4Byte 程のメモリコピーで済む．

7．実装

本論文で述べた設計に基づいたプロトコルスタックを実装した．

ターゲットマシンは，CQ 出版の「PPC403GA 評価ボード」である．スペックは以下の表 4 に示す．

表 4　「PPC403GA 評価ボード」スペック

機能ブロック	仕様
CPU	PowerPC403GA 33.3MHz
メモリ(ROM)	フラッシュ ROM 512KB
メモリ(RAM)	DRAM 4MB
Ethernet (10baseT)	chip: DP80932 (NE2000 互換)
シリアルポート	16550 互換

本プロトコルスタックは，C 言語で作成した．各モジュールのおおよその行数は以下の通りである．

表 5　モジュールサイズ

モジュール	行数
API	300
buffer	500
protocol	1200
transmit	400
(EthernetDeviceDriver)	(700)

8．評価

今までに説明した設計に基づき実装を行い，その評価を行った．ここではその結果を示す．

本プロトコルスタックのコードサイズは，約 80KB となった．これは，簡易 OS 機能を持ったカーネルを含めたものである．また，実行時のデータサイズはコンフィグにより異なるが最低で 10KB 弱であり，今回の評価実験の際のデータサイズは約 20KB であった．

8.1. コピーレス

ここでは，データコピーが行われないことが効いてくる部分の測定を行う．

まず，実装環境上でのメモリコピーにかかる時間を測定してみた．

・DRAM メモリ間のデータコピー
　1518 Byte で 0.9 msec

次に，プロトコルスタックでデータのコピーが起きていた個所について考える．従来のソケットで送信時にコピーが生じていた個所は write()などの送信コマンドを発行した際である．その処理の中で，アプリケーションから示されたアドレスのバッファからデータをソケットのバッファへコピーしていた．本プロトコルスタックでの送信コマンドの発行は API: send()にあたる．そこで，本プロトコルスタックでパケットの送信にかかる時間を以下の方法で計測する．以下の方法では，結果値を妥当なものとするため 1000 回送信した時間を計測している．

本システムでは，図 7 の A の手順で送信を行う．本システムでは従来のものと異なり，送信の前に書き込むべきアドレスを得る必要がある．そこで，送信時の時間計測部分を send()だけでなく，ask_sbuf()も含め計測した．

```
#define SEND_SIZE 1460
#define LOOP_CONT   1000

{
  char    *buf, *wk;
  int   len;
  int   i, j;

TIMER_START;
  for(i=0; i<LOOP_CONT; i++)
  { // SEND_SIZE(byte)適当なデータを送信
    //ask_sbuf(),書込むアドレス，サイズを得る
    len = ask_sbuf(&buf);
    if(len < SEND_SIZE) error("not enough buf size");

    /* //時間計測から外したいのでコメントアウト
    wk = buf;
    for(i=0; i<SEND_SIZE; i++){
      *wk++ = i;  // 適当にデータ詰め
    }
    */

    // API: send(),送信コマンド
    if(send(buf, SEND_SIZE) <0) error("cannot send");
  }
  TIMER_STOP;
}
```

A

図 7　送信の時間測定法

その結果を次ページに示す．

この結果よりコピーを行っていないが，処理時間と送信するサイズには若干の相関があることがわかる．本システムでは，コピーを行わないため，大きなサイズを送信する場合には，非常に有効であることが確認できた．特に，今回実装したボードに載っているイーサネットでの送れる最大サイズ 1460Byte を 1 回送信する場合の処理時間は 614 μsec に対して，もしコピーを行うとその処理に 880 μsec 余分にかかることになり，全体で 1490 μsec かかることになる．コピーを回避したことで，約 6 割の速度向上している．

また，時間だけでなく，メモリコピーにかかる CPU パワーやメモリ使用量といったハードウェア資源も節約されている．

表 6　送信時の時間計測結果

SEND_SIZE (Byte)	LOOP_CONT (回数)	時間 (m sec)	メモリコピーにかかる時間（計算値）(m sec)	コピーレスによる速度向上率（計算値）
100	1000	453	60	0.11 = 60 / (453+ 60)
500	1000	521	300	0.37 = 300 / (521+300)
1460	1000	614	880	0.59 = 880 / (614+880)

9．おわりに

本論文では，ハードウェア資源に対する制約が厳しいという組込みシステムの特性を考慮に入れ，コピーレスで動作するプロトコルスタックの設計を示した．

従来のプロトコルスタックの問題点であったハードウェア資源の消費を，バッファの位置と管理方法，アプリケーションからそのバッファへ独自 API を用いたアクセスの方法，バッファにパケットしての構造を持たせデバイスと直接やりとり可能とする，といった手法により解決するような設計を行った．

また，その設計に基づき実装を行い，その評価を行った．

参考文献

[1] 大塚雄三，並木美太郎，組込みシステム向け TCP/IP プロトコルスタックの設計，第一回 FIT2002　平成 14 年 9 月．

[2] C.A. Thekkath, T.D. Nguyen, E. Moy, E.D. Lazowska, *Implementing Network Protocol at User Level*, Technical Report 93-03-01, Department of Computer Science and Engineering, University of Washington, Mar 1993.

[3] 高田広章，組み込みシステムに適した TCP/IP API，コンピュータシステム・シンポジウム平成 10 年 11 月，情報処理学会

[4] 井戸上彰，加藤聰彦，鈴木健二，パーソナルコンピュータおよびワークステーションのための OSI7 層ボードの実装と評価，情報処理学会論文誌，Vol.36，No.3，pp.763-pp.774，Mar.1995.

[5] 中本幸一，高田広章，田丸喜一郎，組込みシステム技術の現状と動向，情報処理，38 巻 10 号，pp871-903，1997 年 10 月．

[6] Stephen T. Satchell　H.B.J.Clifford　小嶋隆一訳 Linux IP Stacks Commentary，セレンディップ，2001.