Preccs:インターネットのための通信プロトコル記述言語

服部 健太
hattori@isp.co.jp
(株) システム計画研究所

概要

我々は通信プロトコルを記述するための仕様記述言語 Preccs を提案し，Preccs によって書かれた仕様から C のプログラムを自動的に生成する処理系を開発している．Preccs による通信プロトコルの記述は正規表現とプロセス計算に基づいており，プロトコルのメッセージ形式や送受信手順を簡潔に記述できるという特徴を持つ．本稿では，Preccs を用いた通信プロトコルの記述方法について説明するとともに，インターネットで利用されている通信プロトコルのいくつかを，実際に Preccs を用いて実装を行い，評価した結果を報告する．

1 はじめに

今日，インターネット上では様々なサービスが提供されており，それに応じて通信プロトコルの種類も多様化している．近年では IKE（インターネット鍵交換）[1] に代表されるように複雑な通信プロトコルも多く，これらのプロトコルを正しく短期間で実装するのは困難である．このような背景から，我々は通信プロトコルを簡潔に記述することが可能な通信プロトコル記述言語 Preccs を提案し，さらに Preccs による記述から C 言語のコードを生成する通信プロトコルコンパイラの開発を行っている [2, 3]. Preccs による通信プロトコルの記述は正規表現とプロセス計算に基づいており，プロトコルのメッセージ形式や送受信手順を宣言的に簡潔に記述できるという特徴を持つ．また，Preccs コンパイラによって，実際にプロトコルの処理を行う C 言語のコードを自動的に生成することが可能である．

通信プロトコルの分野では，LOTOS や Estelle といった仕様記述言語が知られており [4]，これらによる記述から実際にプロトコルの処理コードを生成するツールもいくつか報告されている．しかしながら，これらは本来，プロトコルの正しさを検証することが目的で設計されたものであり，プロトコルの記述にあたっては専門的な知識が要求されたりと難しく，実装には適していない．実際，TCP/IP をはじめとしてインターネット上で利用されている通信プロトコルは，ほとんどが C 言語によって実装されており，これらの仕様記述言語が一般に利用されているとは言い難い．一方，我々の提案する Preccs は，一般的なプログラマでも通信プロトコルを簡単に実装できることを目的として設計されており，実際の通信プロトコル開発において広く利用されることを目指している．インターネットで利用されているプロトコルは，まず実装によって動作が確かめられ，その後でプロトコルの仕様がまとめられるという手順を踏むものが多い．このようなある種の文化的な事情を考えると，LOTOS のようにまず仕様の検証をきっちりと行ってから，実装の段階へ進むことを前提としたものよりも，Preccs のように簡単にプロトコルを記述でき，すぐに実際の動作を確認できるような処理系の方が適していると考えられる．本稿では，インターネット上で実際に利用されているいくつかのプロトコルを Preccs によって実装し評価した結果を報告し，Preccs の有効性を示す．

2 通信プロトコル記述言語 Preccs

一般に通信プロトコルは (1) メッセージ形式と (2) メッセージの送受信手順の 2 点によって規定されるが，Preccs による通信プロトコルの仕様記述は，この事実と素直に対応した構成になっている．本節では，Preccs を用いた通信プロトコルの仕様記述方式について，簡単な例を交えながら説明する．

2.1 メッセージ形式の記述

Preccs では，メッセージ形式は正規表現のパタンを記述することによって定義する．ただし，純粋な正規表現ではなく，通信プロトコルで用いられるメッセージ形式を定義しやすいようにいくつかの工夫がなされている．

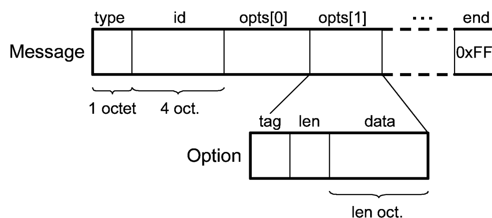

図 1: メッセージ形式の例

```
// メッセージ全体の構造定義
Message ::=
  type : octet,      // 1 オクテット長
  id   : octet[4],   // 4 オクテット長
  opts : Option*,    // オプションの並び
  end  : "FF"h;      // 0xFF で終端
// オプション構造の定義
Option ::=
  tag  : octet,
  len  : octet,
  data : octet[len]; // len オクテット長
```

図 2: Preccs によるメッセージの記述例

2.1.1 メッセージ形式の記述例

例として，通信プロトコルによく見られるメッセージ形式を図 1 に示す．図 1 に示したメッセージは，1 オクテットと 4 オクテットの二つの固定長フィールドの後に 0 個以上のオプションが続き，最後は 0xFF で終わるという形式である．また，オプションも構造を持ち，オプション種別を表すフィールド tag とオプションのデータ長を示すフィールド len，さらにオプションデータのフィールド data からなる．これを Preccs で記述すると図 2 のようになる．図 2 において，`Message ::=` …; の部分でメッセージ形式 Message を定義している．Message の各フィールドにはラベル名を付与することができる．たとえば，`type`, `id` はラベル名であり，それぞれ `octet`, `octet[4]` というフィールドが対応している．フィールドの間を区切るコンマ (`,`) は，正規表現の連接に相当し，各フィールドが連続していることを意味している．`opts` ラベルで指定されるフィールド `Option*`はメッセージ形式 Option が 0 個以上連続することを意味しており，アスタリスク (`*`) は正規表現における閉包である．Preccs ではこの他にも選択 (`|`)，1 回以上の繰り返し (`+`)，0 回か1 回のパタン (`?`) といった，正規表現の記法を使用することができる．

```
// 要求メッセージの定義
RequestMsg ::= Message{type\"01"h};
// 応答メッセージの定義
ReplyMsg   ::= Message{type\"02"h};
```

図 3: 派生パタンを用いたメッセージの記述例

2.1.2 ラベル参照による繰り返し

Preccs では正規表現を拡張して，ラベルの示すフィールドの値を参照することによって可変長のデータが表現できるようになっている．図 2 のメッセージ形式 Option の定義において，data フィールドの長さは直前の `len` ラベルで指定されたフィールドの値によって規定される．たとえば，len フィールドの値が 0x04 ならば，data フィールドの長さは 4 オクテットということになる．このようなメッセージの構造は通信プロトコルでは一般的であり，Preccs ではそれを直接的に表現することが可能である．

2.1.3 派生パタン

通信プロトコルで用いられるメッセージは，あるフィールドの値によってメッセージの種別が規定されることが多い．たとえば，図 1 の例で示した Message 中の type フィールドはメッセージの種別を表しており，このフィールドの値が 0x01 ならば要求メッセージ，0x02 の場合は応答メッセージを意味しているものとする．これを Preccs では図 3 のように記述することができる．

このように元々のメッセージ形式の一部のフィールドを置き換えることによって，新たなメッセージを定義することが可能である．これを派生パタンと呼ぶ．派生パタンによるフィールドの置き換えは，元々のメッセージ形式の範囲内でのみ許される．上記の例で言えば，type フィールドの元々のパタンは `octet` なので，置き換えができるのは `octet` に含まれるパタンのみである．たとえば，

```
Message{type\("01"h|"02"h)}
```

のような置き換えは可能だが，

```
Message{type\octet[4]}
Message{type\"0123"h}
```

のような置き換えは許されない．

派生パタンを用いることによって，通信プロトコルのメッセージが持つ自然な階層関係を簡潔に記述することができる．

2.2 送受信手順の定義

Preccs におけるメッセージ送受信手順の記述は，プロセス計算と呼ばれる体系の一つである Milner の CCS[5] がモデルとなっている．Preccs では，チャネルと呼ばれる仮想的な通信路に対して，同期的にデータの送受信を行う複数のプロセスを定義することによって，通信プロトコルの送受信手順を記述していく．

2.2.1 送受信手順の記述例

ここでは，送受信手順の記述例として，標準入力から入力された文字列を echo サーバに送信し，受信した応答メッセージを標準出力に表示するという処理を，回数を数えながら繰り返すという，簡単な echo クライアントを取り上げる．図 4 は，echo クライアントの Preccs による記述例である．2 行目の `EchoClientProc(count:int) ::= …;` の部分が，クライアントの送受信手順の処理をプロセスとして定義しているところである．プロセスはパラメータを持つことができ，EchpClientProc プロセスは，現在の繰り返し回数を保持するための int 型のパラメータ `count` が与えられている．3 行目は標準入力から文字列を読み込み，その値を文字列型の変数 `key` に束縛するという動作を表している．この動作に続いて，4 行目で変数 `key` の内容をソケットに出力している．コンマ (`,`) 記号は逐次動作を意味しており，前の動作が完了した後に次の動作を実行する．

5 行目から 8 行目までは選択動作を表している．ソケットからメッセージを受信した場合には，標準出力に`"reply: "`という文字列とともに受信メッセージを出力する．また，5 秒間待ってもメッセージを受信できなかった場合にはタイムアウトし，標準出力に`"timeout\n"`を出力する．縦棒記号 (`|`) はこのような選択的な動作を表すために用いる．このように，あるメッセージの入力を待ちつつ，一定時間を過ぎた場合にはタイムアウトするという処理は，通信プロトコルを実装する際によく用いられる．このような処理を C で記述する場合には，一般に Unix の select システムコールや Win32 API の WaitForMultipleObjects などの比較的低レベルな API を用いた煩雑なコーディングが必要となる．Preccs では選択動作 (`|`) を用いることで，このような処理でも簡潔に記述することができる．

EchoClientProc プロセスは，これらの選択動作のいずれかを実行した後，9 行目で，`count` に 1 を足した値をパラメータとして新しいプロセスを再帰的に生成し，自身のプロセスは終了する．

2.2.2 チャネル入出力

プロセス同士は，チャネルを通じて同期的にデータを送受信することが可能である．先ほどの例に出てきた `stdin` や `sockout` は組み込みのチャネルであり，これらのチャネルは Preccs のプロセスが外部とのメッセージをやり取りするために用いられる．

プロセスの中で新しくチャネルを生成することもできる．たとえば，整数型のデータをやり取りするためのチャネル `ich` を生成するには，プロセス定義の中で `new ich:<int>`のように記述する．

すべてのチャネルは型を持っており，そのチャネルを通して送受信できるデータの型は制限される．標準入出力チャネルやソケット入出力チャネルは，文字列型のデータ[1]のみ送受信できる．

2.2.3 タイムアウト処理

`timer` はタイムアウト処理を行うための特別な組み込みチャネルで，プロセスは`timer` チャネルに渡された秒数だけブロックされる．たとえば，`timer!10` とすると，プロセスは 10 秒間ブロックされる．これによってタイムアウトやスリープといった処理を実現することができる．

2.2.4 パタンマッチ

あるデータに対して，そのデータの持つパタン種別によって動作を振り分けたい場合にはパタンマッチ構文 (`match`〜`with`) を使用する．たとえば，先ほどの echo クライアントに機能を追加して，標準入力から`"quit\n"`と入力されたらプログラムを終了させることを考える．この場合，図 4 の 3, 4 行目を以下のように変更すればよい．

```
stdin?key:string,
( match key with
    "quit\n" -> stop
  | string   -> skip
```

[1]Preccs における文字列型は `octet*`パタンと同義である．

```
1:  // echo クライアントの送受信手順の定義
2:  EchoClientProc(count:int) ::=
3:      stdin?key:string,
4:      sockout!key,
5:      (
6:          sockin?msg:string -> stdout!("reply: " + msg)
7:        | timer!5           -> stdout!("timeout\n")
8:      ),
9:      EchoClientProc(count+1);
```

図 4: echo クライアントの記述例

```
    ),
    sockout!key,
```

ここでは，変数 key が保持しているデータのパタンに応じて動作の振り分けを行っている．パタンマッチは先頭から順に試みられ，最初にマッチしたパタンに対応する動作が実行される．したがって，上記の例では，変数 key は，まず"quit\n"とのマッチングが試みられる．このマッチングに失敗した場合には，続いて string パタンとマッチング[2]が行われる．ここで stop はプロセスの終了動作を，skip は後続の処理の実行を意味する予約語である．正規表現のパタンとして記述したメッセージ形式の定義をパタンマッチングの機構で用いることによって，受信メッセージの種類による動作の振り分けを簡単に記述することが可能となる．

2.2.5 メッセージ生成

通信プロトコルでは，ネットワーク上に送信するメッセージを組み立てるという処理が必要である．Preccs のプロセス中では新しくメッセージを生成することが可能で，そのメッセージはデフォルトのデータで初期化される．たとえば，図 3 で定義した RequestMsg 形式のメッセージを生成したい場合には，プロセス定義の本体に

```
    ... , new msg:RequestMsg, ...
```

と記述すればよい．新しく RequestMsg 形式のメッセージが生成され，msg 変数に束縛される．ここで生成されるメッセージはデフォルトのデータで初期化されている．octet のデフォルト値は 0x00 であり，閉包パタン (*) のデフォルト値は ϵ というように，各基本パタンに対応するデフォルト値が決められており，全体のデフォルトデータはそれらから導かれる．たとえば，RequestMsg のデフォルトデータは 0x0100000000FF となる．

[2] 変数 key は文字列型なので，このマッチングは必ず成功する．

2.2.6 インライン C

実際の通信プロトコルでは，OS のシステムコール呼び出しを始めとして，低レベルの処理が必要となることが多い．これらの必要な処理に対して，Preccs の記述の中に C 言語のコードを直接埋め込むことができれば便利である．Preccs では，C 言語のコードを C{〜C}で囲むことによって，インラインで C 言語の処理を呼び出すことができる．

例えば，int 型チャネルから入力された値を 16 進で表示するプロセスは，C 言語の printf 関数を用いて，次にように定義することができる．

```
PrintHex(ch:<int>) ::=
    ch?n,
    C{ printf("%x\n", $n$); C},
    PrintHex(ch);
```

インライン C の中のnは，Preccs の変数 n を参照するための記法である．

3 実験

実際の通信プロトコルの適用例として，Preccs を用いて DHCP クライアントと TCP/IP の一部をそれぞれ実装した．本節ではその結果を報告する．

3.1 DHCP プロトコル

DHCP[6] はインターネットに接続されたホストに対して IP アドレス等の設定パラメータを与えるためのプロトコルである．DHCP はクライアント・

0	8	16	24 (31)
op (1)	htype (1)	hlen (1)	hops (1)
xid (4)			
secs (2)		flags (2)	
ciaddr (4)			
yiaddr (4)			
siaddr (4)			
giaddr (4)			
chaddr (16)			
sname (64)			
file (128)			
options (可変長)			

図 5: DHCP メッセージの形式

サーバ方式で，クライアントは同一の LAN セグメント上に存在する DHCP サーバに対して問い合わせを行い，必要な設定情報を受け取る．

3.1.1 DHCP クライアントの実装

DHCP メッセージの形式は，図 5 に示すとおりであり，固定長のフィールドの並びの後に，複数個のオプション列が続くというものである．基本的に DHCP のオプションは，オプションの種別を示す code，後続のデータ長を示す len，そして，len オクテットの長さを持つオプションデータ value の 3 つのフィールドからなる形式であり，これは，図 1 で示したオプションの構造と同様のものである．

DHCP のメッセージ形式を Preccs で記述すると，おおよそ図 6 のようになる[3]．これを見てもわかるように，Preccs では通信プロトコルのメッセージ形式を，直接的に定義することが可能である．

DHCP クライアントの動作は，図 7 に示す[4]状態遷移にしたがう．クライアントの状態は「初期化」状態から始まる．最初に DHCPDISCOVER メッセージを生成し，それをブロードキャストする．次に，クライアントは一定時間 DHCPOFFER を収集し[5]，それを参考に（例えば、最初のメッセージであるとか以前アクセスしたサーバからのメッセージ）サーバを選択する．そして，選択したサーバを指定した

[3]実際の DHCP オプションにはマジックコードやパディングなどが含まれるが，ここでは省略している．

[4]再初期化に関する遷移は省略した．

[5]LAN セグメント上に複数の DHCP サーバが存在する場合，それぞれのサーバが DHCPOFFER を返す可能性がある．

```
// DHCP メッセージの定義
DhcpPacket ::=
  op     : (BOOT_REQUEST|BOOT_REPLY),
  htype : octet[1],  //ハードウェア番号
  hlen   : octet[1],  //HW アドレス長
  hops   : octet[1],  //ホップ数
  xid    : octet[4],  //トランザクション ID
  secs   : octet[2],  //経過時間
  flags : octet[2],  //フラグ
  ciaddr: octet[4],  //クライアント IP
  yiaddr: octet[4],  //割当て IP アドレス
  siaddr: octet[4],  //サーバ IP アドレス
  giaddr: octet[4],  //リレーエージェント
  chaddr: octet[16], //HW アドレス
  sname : octet[64], //サーバ名
  file   : octet[128],//ブートファイル名
  opts   : DhcpOption*;
// DHCP オプションの定義
DhcpOption ::=
  code   : octet,
  len    : octet,
  value : octet[len];
```

図 6: Preccs による DHCP メッセージ形式の記述

DHCPREQUEST をブロードキャストする．最後に，指定されたサーバからの DHCPACK を受信するとクライアントは初期化を終了して「リース」状態へと移行する．

Preccs では，このようなクライアントの各状態を一つのプロセスとして定義し，ある状態を表すプロセスが，別の状態を表すプロセスを起動することによって，プロトコルの状態遷移を表現することができる．Preccs による DHCP クライアントの状態遷移の記述を図 8 に示す．2〜7 行目で，「初期化」状態を示すプロセス DhcpInit を定義している．DhcpInit プロセスは，DHCPDISCOVER メッセージを生成するプロセス CreateDhcpDiscoverPkt を起動し，チャネル ch からその結果を受け取る．そして，受け取ったパケットは sockout チャネル[6]を通じてブロードキャストし，「選択中」状態を表す DhcpSelecting プロセスを起動した後，自分自身は終了する．

DHCPDISCOVER メッセージの形式は，以下の

[6]sockout チャネルは，ここではブロードキャスト送信用の UDP ソケットにバインドされている．

```
 1: // 初期化
 2: DhcpInit() ::=
 3:     new ch:<DhcpDiscoverPkt>,
 4:     CreateDhcpDiscoverPkt(ch),
 5:     ch?sPkt:DhcpDiscoverPkt,
 6:     sockout!sPkt,
 7:     DhcpSelecting();
 8: // 選択中
 9: DhcpSelecting() ::=
10:     sockin?msg:octet* ->
11:       ( match msg with
12:             rPkt:DhcpOfferPkt -> SetOptions(rPkt.opts),
13:                                  new ch:<DhcpRequestPkt>,
14:                                  CreateDhcpRequestPkt(ch,rPkt),
15:                                  ch?sPkt:DhcpRequestPkt,
16:                                  sockout!sPkt,
17:                                  DhcpWaitReply(sPkt)
18:           | _ -> stdout!"Illgal packet recvd.\n", DhcpInit())
19:   | timer!10 -> stdout!"Timeout.\n", DhcpInit();
20: // 応答待ち
21: DhcpWaitReply(sPkt:DhcpRequestPkt) ::=
22:     sockin?msg:octet* ->
23:       ( match msg with
24:             DhcpOfferPkt -> DhcpWaitReply(sPkt)
25:           | DhcpNakPkt   -> stdout!"NAK recvd.\n", DhcpInit()
26:           | DhcpAckPkt   -> DhcpLease(sPkt)
27:           | _            -> stdout!"Ilegal packet recvd.\n", DhcpInit())
28:   | timer!10 -> DhcpInit();
29: // リース
30: DhcpLease(sPkt:DhcpRequestPkt) ::=
31:     C{ extern int set_ifconf(void); set_ifconf(); C},
32:     ( sockin?msg:octet* ->
33:         ( match msg with
34:             (DhcpOfferPkt|DhcpNakPkt|DhcpAckPkt) -> DhcpLease(sPkt)
35:           | _ -> stdout!"Illegal packet recvd.\n", DhcpInit()
36:         )
37:     | timer!3600 -> sockout!sPkt, DhcpExtraLease(sPkt));
38: ...
```

図 8: DHCP クライアント状態遷移の記述

ように DhcpPacket からの派生パタンとして定義されている．

```
DhcpDiscoverPkt ::=
        DhcpPacket{op\BOOT_REQUEST}
                  {flags\BOOT_BROADCAST}
                  {opts\DhcpDiscoverOpt};
```

3 行目で生成されるチャネル ch は，このような型を持つメッセージを送受信するためのチャネルである．

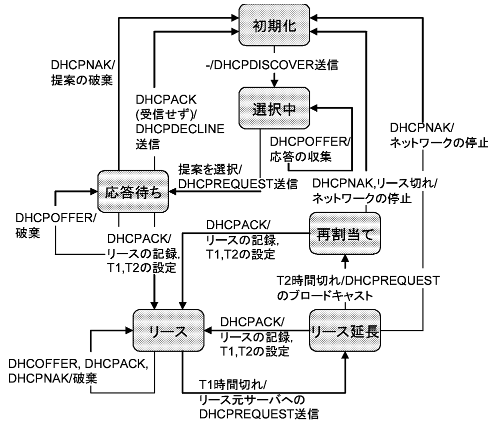

図 7: DHCP クライアントの状態遷移 (一部)

次の 9〜19 行目では,「選択中」状態を表す `DhcpSelecting` プロセスを定義している.ここでは,`sockin` チャネルからメッセージを受信する (11 行目) か,10 秒間待ってタイムアウトする (19 行目) という選択的動作を行っている.メッセージを受信した場合には,パタンマッチによってメッセージを解析する.解析した結果が DHCPOFFER メッセージ (`DhcpOfferPkt`) ならば,オプションを設定する `SetOptions` プロセスを呼び出し,DHCPREQUEST メッセージを生成して,再びブロードキャストした後に,「応答待ち」状態の `DhcpWaitReply` プロセスを起動する[7].その後,`DhcpWaitReply` プロセスでは,DHCPACK メッセージを待って,「リース」状態である `DhcpLease` プロセスを起動する.

これまで説明してきたように,Preccs ではプロトコルの状態遷移を各プロセスの起動によって表現する.また,タイムアウト処理やパタンマッチの機構を用いて,簡潔にプロトコルの処理を記述することが可能である.

3.1.2 C による実装との比較

以上のように Preccs で実装を行ったものと同等の機能を有する DHCP クライアントを C 言語でスクラッチで実装し,記述量と実行時間の比較を行った.実装範囲としては,IP アドレスの取得,設定,リース期限切れ後の再取得,これに必要な DHCP オプションの一部などが含まれる.

表 1 に,DHCP クライアントの記述量の比較結果

[7]DhcpSelecting プロセスは一番最初に受信した DHCPOFFER メッセージを選択する.

表 1: DHCP クライアントのコード記述量の比較

内訳	Preccs	C	比率
メッセージ形式定義	79 行	54	146.3%
メッセージ解析	42	354	11.8
メッセージ組立て	17	100	17.0
送受信処理	27	313	8.6
その他	239	501	47.7
合計	404	1322	30.6

表 2: DHCP クライアントの実行時間の比較

内訳	Preccs	C	比率
real	104.7 ms	82.1	1.28
user	6.6	1.4	4.86
sys	7.2	6.4	1.13

を示す.これを見ると,Preccs による記述は C に比較して,全体で 3 割程度に収まっていることがわかる.特に送受信処理とメッセージ組立て,解析処理の記述量が大幅に削減されている.このことから,Preccs におけるメッセージ形式と送受信手順の記述方式は,非常に有効であることがわかる.

次に Preccs による実行時のオーバヘッドを見るために,実行時間の比較を行った.ここではプログラムの起動から IP アドレスを取得,設定するまでの時間を計測した.評価のために,同一の LAN セグメント内に接続されたクライアントとサーバの計 2 台のマシンを用意した.表 2 に DHCP クライアントの実行時間の比較結果を示す.実行時間は time コマンドを用いて測定し,10 回の平均値を算出した.

結果を見ると,DHCP クライアントに関する限り,Preccs による実行時間 (real) のオーバヘッドは高々30 %程度に過ぎないことがわかる.DHCP のようにシビアな処理性能を求めないプロトコルでは,この程度のオーバヘッドは十分に許容範囲内である.また,待ち時間を除く,CPU 使用時間 (sys+user) のオーバヘッドも 2 倍程度に抑えられている.

3.2 TCP/IP

TCP/IP はインターネットの基盤となるプロトコル群であり,一般に TCP や UDP,IP,ICMP などの様々な階層にまたがった複数のプロトコルの総称であることが多い.Preccs で TCP/IP のような

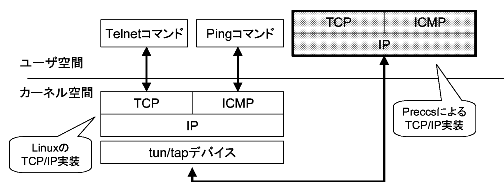

図 9: Preccs による TCP/IP 実装の概要

複数のプロトコル群（プロトコルスタック）を記述することで，ある程度の規模の大きなプロトコルにおいて，どのような問題点，あるいは利点があるのか，また，生成コードの性能はどの程度かといったデータを得ることができる．このように一つのベンチマークとして TCP/IP の実装を行った．

今回，実装した機能は TCP/IP のごく一部であるが，ping の応答，telnet コマンドを用いた簡単な TCP データ送受信が可能である．以下に主な実装項目を挙げる．

- ICMP エコー要求と応答
- IP，TCP ヘッダの処理
- TCP セッションの確立と終了
- TCP データ送受信と ACK 応答

TCP の輻輳制御や再送処理，スロースタートなどの重要な機能は実装していない．また，複数の TCP セッションを確立することはできない．この制限に関しては後で議論する．

3.2.1 TCP/IP の実装の概要

Preccs コンパイラが生成する C のコードは Linux のユーザプログラムとして動作するものであり，現時点では，これをそのままカーネルに組み込むことは難しい．このため，今回は TCP/IP スタックを Linux のユーザプロセスとして実装した．図 9 に，Preccs による TCP/IP 実装の概要を示す．

ユーザプロセスから直接，IP 層にアクセスするために tun/tap 仮想デバイスを利用している．また，実装を簡単にするため，実際のネットワーク I/F は使用せず，Linux の TCP/IP スタックと Preccs の TCP/IP スタックを tun デバイスを経由して同一の Linux マシン上で接続する構成となっている．それぞれの TCP/IP スタックはちょうどピアツーピアで接続された状態になっており，Linux の telnet や ping コマンドを用いて，Preccs 側の TCP/IP スタックとデータの送受信を行うこともできる．

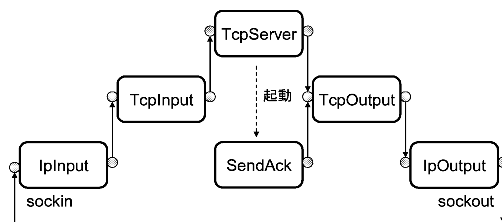

図 10: Preccs の TCP/IP プロセス構成

プロトコルスタックを Preccs で実現する場合には，各プロトコルの処理を Preccs のプロセスとして実装し，プロトコル間のデータ送受信は，プロセス間を結ぶチャネルに対するデータの入出力として実装するのが，自然な方法である．この場合，パケットの受信処理を行う各プロセスは，例えば，以下のように記述できる．

```
ProtoIn(lower:<Pkt>,upper:<Pkt>) ::=
    lower?p:Pkt,    // 下位レイヤからの入力
    ...             // パケットの処理
    upper!p,        // 上位レイヤへの出力
    ProtoIn(lower,upper);
```

このようなプロセスをチャネルを用いてパイプラインのように接続していくことによって，プロトコルスタックの一連の入力処理を実現することができる．例えば，以下のように記述できる．

```
Proto1In(ch1,ch2),Proto2In(ch2,ch3), ...
```

出力処理も同様である．

今回実装を行った TCP/IP のプロセス構成を模式的に示すと，図 10 のようになる[8]．図中の IpInput や TcpOutput などはそれぞれ IP プロトコルの入力処理，TCP プロトコルの出力処理を行うプロセスを示している．また，小さな丸はチャネルを意味している．IpInput プロセスは tun デバイスにバインドされた sockin チャネルから IP パケットを受信すると IP ヘッダの解析を行い，TCP パケットならば TcpInput プロセスに，ICMP の場合には（図中には描かれていないが）IcmpInput プロセスにそれぞれパケットを渡す．TcpInput プロセスは TCP ヘッダの解析を行い，TcpServer プロセスにデータを渡

[8] この図はかなり簡略化してあり，実際には ICMP の処理を行うプロセスや，画面の入出力を行うプロセスなどが存在する．

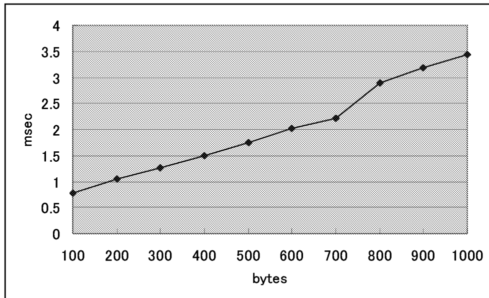

図 11: Ping コマンドの応答時間

す．TcpServer プロセスは，一つの TCP セッションを管理するプロセスで，セッション確立や終了，ACK 送信などの処理を行う．

以上のような方針で実装を行った結果，コードの記述量は Preccs による記述が約 600 行，その他，チェックサムの計算や tun/tap デバイスの処理など，C 言語による記述が約 200 行となり，合計 800 行程度となった．

3.2.2 TCP/IP の性能

ここでは，Preccs で実装した TCP/IP スタックの性能を測定した結果を示す．測定に使用したマシンは，DELL Dimension XPS R400(Pentium II 400MHz，主メモリ 384MB) で，OS は Fedora-Core3(Linux Kernel 2.6.12) である．

まず，Preccs の TCP/IP スタックに対して ping コマンドを用いて ICMP パケットを投げ，様々なデータサイズに対する応答時間の測定を行った．結果は，図 11 のとおりである．グラフの横軸は ICMP パケットのデータ長を示しており，縦軸が応答時間である．これをみると，データ長に比例して応答時間が伸びていることがわかる．これは，Preccs では正規表現のパタンマッチ処理が重く，この処理にはパタンマッチを行うデータの長さ比例した時間がかかるためと考えられる．参考までに，同じ環境でループバックアドレス (127.0.0.1) に対して行った ping の応答時間は，データ長に対して緩やかに伸びてはいるものの，その範囲はおおよそ 0.08〜0.10msec に留まった．Preccs の TCP/IP スタックでは，カーネルとユーザ空間で余分なデータコピーが含まれているため，一概に比較することはできないが，ping 応答では 10 倍〜40 倍程度のオーバヘッドが生じていることがわかる．

次に，TCP の処理性能をみるために，Preccs 側スタックに対して，連続的にデータ送信を行った時間を測定してみた．結果，おおよそ 280〜290KBytes/sec のスループットが得られた．参考のために，ループバックデバイスを用いて Linux のプロトコルスタック自体のスループットも測定してみたが，こちらは 23〜25MBytes/sec 程度だった．したがって，Preccs では 100 倍近い処理オーバヘッドがかかっていることになる．

4 議論

DHCP と TCP/IP に関する実験の結果から，Preccs は DHCP のように上位層に近いプロトコルの実装で，より効果を発揮すると考えられる．これらの層に含まれるプロトコルの種類は多種多様であり，実装する機会も多い．また，メッセージ形式や処理手順が複雑なものが多いため，Preccs で実装することによって，開発工数の大幅な削減が期待できる．特にプロトタイピングツールとしての利用が考えられるだろう．生成コードの処理性能も，これらのプロトコルではあまり問題にならないことが多い．

一方，TCP のようなトランスポート層のプロトコルも実装することは可能であるが，一般的にこれらの層に含まれるプロトコルは，高い処理性能の要求を満たす必要があり，現在の Preccs 処理系では不十分である．ただし，今後，実行時ライブラリの実装を見直したり，生成コードの最適化を行ったりすることによって，Preccs でも十分な性能を発揮できるようになる可能性はおおいにある．

その他，現在の Preccs の言語仕様では，TCP の複数セッションの扱いが難しいことが，TCP プロトコルの記述過程で明らかになっている．複数セッションに対応するためには，図 10 の TcpServer プロセスをセッションの数だけ用意してやるのが自然な構成だが，そのためにはセッションが確立されるたびに，TcpServer プロセスと新たなチャネルを生成し，TcpInput プロセスと接続する必要がある．このような動的に接続関係が変化するプロセス構成に対して，Preccs がベースとしているプロセス計算の枠組みである CCS では，うまく対応することができない．一方，CCS の拡張として π 計算 [7] が知られている．π 計算では，チャネル自身をチャネルを通して送受信することが可能であり，これによって，プロセスやチャネルの接続関係が動的に変化するような構成にも柔軟に対応できる．Preccs も π 計

算に基づく拡張を行うことにより，TCP の複数セッションの扱いを自然に表現できるようになると期待できる．

5 関連研究

Preccs と同じように，通信プロトコルを実装するための言語としては Prolac[8] がある．これは静的に型付けされたオブジェクト指向言語に位置づけられる．Prolac コンパイラはメソッド呼び出しのインライン展開やクラス階層の解析による動的ディスパッチの除去などの最適化を行うことによって，ソースコードの可読性を保ったまま，性能の高いコードを生成することが可能である．実際に，Prolac によって TCP を実装した結果，Linux 2.0 の TCP と同等の性能が得られたと報告している．ただし，Prolac では Preccs のような正規表現に基づいたメッセージ形式を定義することはできないので，IKE のような複雑なメッセージ形式を持ったプロトコルを記述する場合には，Preccs の方がより適していると考えられる．

河野らはクライアント・サーバ型のアプリケーション層プロトコルの実現を支援する April[9, 10] フレームワークを提案し，April によって実際に POP，HTTP，SMTP などのプロトコルを記述した結果を報告している．April では正規表現を用いてメッセージ形式の記述を行い，そこから受信メッセージを解析するコードを自動生成するなど，我々の提案する手法と共通する部分も見られる．ただし，April はアプリケーション層プロトコルを対象としており，送受信されるメッセージは文字列を前提としている．一方，Preccs ではより広い範囲のプロトコルが対象であり，特に，メッセージがバイナリ形式でも問題なく扱うことができる．

6 おわりに

本稿では，通信プロトコル記述言語 Preccs を用いたプロトコルの記述方法について説明し，インターネットで利用されているいくつかのプロトコルを Preccs で実装した結果について報告した．今後の課題として，生成コードの最適化が挙げられる．現時点では最適化処理をほとんど行っていないため，今後，さまざまな最適化の技術を Preccs に適用していくことによって，より高い性能を持つコードを生成することが可能であると考えられる．そのほか，効果的な syntax sugar を導入することにより，さらに記述量を削減するといったことも可能である．また，π 計算に基づく言語仕様の見直しも検討している．現在，これらの改良点を整理している最中であり，これに基づいて Preccs の次期バージョンの開発を行う予定である．

謝辞

Preccs は，IPA（情報処理推進機構）の公募事業 2004 年度第 2 回未踏ソフトウェア創造事業（伊知地 PM），および，2005 年度下期未踏ソフトウェア創造事業（並木 PM）に採択され，支援を受けている．

参考文献

[1] Harkins, D. and Carrel, D.: The Internet Key Exchange (IKE) (1998). (RFC2409).

[2] 服部健太, 数馬洋一: 正規表現とプロセス代数に基く通信プロトコルのための仕様記述言語の提案 (2005). 情報処理学会プログラミング研究会発表資料.

[3] 服部健太, 数馬洋一: 正規表現とプロセス代数に基づく通信プロトコルコンパイラ, 日本ソフトウェア科学会第 22 回大会論文集 (2005).

[4] 水野忠則 (監修): プロトコル言語, カットシステム (1994).

[5] Milner, R.: *Communication and Concurrency*, Prentice Hall (1989).

[6] Droms, R.: Dynamic Host Configuration Protocol (1997). (RFC2131).

[7] Milner, R.: *Communication and Mobile Systems:the π-Calculus*, Cambridge University Press (1999).

[8] Kohler, E., Kaashoek, M. F. and Montgomery, D. R.: A Readable TCP in the Prolac Protocol Language, *ACM SIGCOMM'99*, Vol. 29, pp. 3–13 (1999).

[9] 河野健二: アプリケーション層プロトコルの実現を容易にするフレームワーク, 情報処理学会論文誌 : プログラミング, Vol. 44, No. SIG 2(PRO 16), pp. 25–35 (2003).

[10] 阿部勝幸, 岩崎英哉, 河野健二: アプリケーション層プロトコルの記述に基づく拡張性に優れたプロトコル処理コード生成系, 日本ソフトウェア科学会第 22 回大会論文集 (2005).