# 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

河野健二[†,††]

インターネットなどの広域分散環境では，アプリケーション層プロトコルを用いて通信を行うアプリケーションが多数利用されている．HTTP や SMTP などのアプリケーション層プロトコルは，文字列を主体としたメッセージのやりとりとして定義されており，その実現には文字列の解析や構成を行う退屈で煩雑なコーディングを要する．また，時系列に沿ったメッセージのやりとりやシステムの状態遷移がプログラム・コード中に埋没し，プロトコルの挙動を把握することが難しい．そのためプロトコル処理部の保守性，安全性が低下し，インターネット・サーバのセキュリティ・ホールの一因となることも多い．本論文では，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を対象に，プロトコル処理部を自動的に生成するコード生成器 April について報告する．April はアプリケーション層プロトコルの記述に特化した記法を提供しており，1) メッセージ・フォーマットの定義，2) 時系列に沿ったメッセージのやりとり，3) システムの状態遷移のみを記述すれば，C 言語のコードを自動的に生成する．メッセージの解析や構成に必要な文字列処理は April の処理系が行い，プログラマが記述する必要はない．また，時系列に沿ったメッセージのやりとりやシステムの状態遷移が明示的に記述されているため，プロトコルの挙動が把握しやすい．本論文では，April によるプロトコルの記述例を示し，April を用いたことによるプロトコル処理のオーバヘッドが十分に小さいことを示す．

# A Framework for Simplifying the Programming of Application-level Protocols

Kenji Kono[†,††]

The growing use of Internet services increases the opportunities for developing application-level protocols such as HTTP, SMTP, and POP. It is a tedious but error-prone task to implement an application-level protocol or to modify the existing implementation. It requires careful manipulation of strings because a trivial bug in the implementation may cause a serious security hole in the Internet service. Making matters worse, an intuitive behavior of the protocol is not clear from the code because message exchanges and state transitions are usually scattered throughout the implementation and buried deeply in the code. In this paper we present the design and implementation of April, a domain-specific language specifically designed for describing application-level protocols of the client-server style. April provides high-level abstractions that are specific to the domain of programming application-level protocols, thus improving safety, productivity, and maintainability of application-level protocols. More specifically, April generates a corresponding C code only if the programmer describes 1) message formats, 2) exchanges of messages between clients and servers in time sequence, and 3) state transitions triggered by messages. In this paper we show a sample of the real protocol in April and demonstrate that the overhead incurred by the use of April is negligible.

## 1. はじめに

情報交換や情報共有の基盤としてインターネットが広く利用されるようになるにつれて，インターネット上でサービスを提供する様々なアプリケーションが次々と開発されている．インターネット上で動作するこれらのアプリケーションは，アプリケーション層プロトコルを用いて通信を行う．アプリケーション層プロトコルとはトランスポート層の上位プロトコルであり，各アプリケーションに固有の通信規約である．アプリケーション層プロトコルには，ウェブページの閲覧に用いる HTTP [1]，メールの送受信に用いる SMTP [2] や POP [3] など多くのプロトコルがある．これらのプロトコルは時系列に沿ったメッセージのやりとりとして定義されており，それぞれのメッセージは文字列を主体として構成されている．たとえば，POP における認証では，クライアントからサーバに “`USER kono\r\n`”

† 電気通信大学情報工学科
Department of Computer Science, University of Electro-Communications
†† 科学技術振興事業団さきがけ研究 21
PRESTO, Japan Science Technology Corp.

という文字列を送信し，それに対し “+OK\r\n” または “-ERR\r\n” という文字列が返信される仕組みになっている．

こうしたメッセージのやりとりを実現するプロトコル処理部は，煩雑で退屈なプログラミングを要する．それぞれのメッセージが文字列を主体として構成されているため，メッセージの解析や構成，文字列とデータ構造の相互変換など，多くの文字列処理を要する．こうした文字列処理は，C 言語などの標準ライブラリで提供されている文字列処理関数を用いて実装されており，些細なプログラミング上の誤りを犯しやすい．プロトコル処理における些細な誤りは，しばしば致命的なセキュリティ・ホールにつながることはよく知られている[4),5)]．また，こうしたプロトコル処理はシステムコール・レベルでの通信プリミティブを用いて実装されていることが多く，オペレーティングシステムに関する詳細な知識を必要とし，プロトコル処理部の可搬性を低下させる一因となっている．

本論文では，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を対象に，プロトコル処理部を自動生成するコード生成器 April について報告する．April はプログラミング支援ツールの一種であり，プロトコルにおけるメッセージのやりとりを定義すると，そのプロトコルを処理する C 言語のコードを自動的に生成する．生成されたコードは RPC や RMI におけるスタブと同様の役割を果たし，アプリケーション・プログラマは C 言語の関数を呼び出すだけで，メッセージの送受信を行うことができる．通信プリミティブの扱いや煩雑な文字列処理は自動生成されたコード中に隠蔽される．そのため，アプリケーション・プログラマはプロトコル処理に特有の煩雑で単調なプログラミングから解放され，より本質的なプログラミングに専念できるようになる．

April ではプロトコルの定義として，1) メッセージ・フォーマット，2) 時系列に沿ったメッセージのやりとり，3) システムの状態遷移のみを定めればよい．これを**プロトコル定義**と呼ぶ．プロトコル定義を記述する April 言語は，アプリケーション層プロトコルに特化した記法を持ち，様々なアプリケーション層プロトコルを宣言的に定義できる．実際，April 言語を用いて POP，HTTP，SMTP などのプロトコル定義を行った．また，POP サーバの 1 つである qpopper のプロトコル処理部を April によって自動生成されたコードに置き換え，元来の qpopper との性能比較を行った．この性能比較では，April によって自動生成されたプロトコル処理部のオーバヘッドは無視できる程度であった．

以下，2 章ではプロトコル処理部の分析を行い，3 章ではプロトコル処理部を自動生成する利点をまとめる．4 章では，プロトコル定義を行う April 言語について述べる．5 章では，April によって生成されたプロトコル処理部の性能について議論する．6 章では関連研究について述べ，7 章で本論文をまとめる．

## 2. プロトコル処理部の分析

April は，プロトコル定義からプロトコル処理部を生成するコード生成器である．プロトコル処理部とは，アプリケーション層プロトコルに従ってメッセージの送受信を行うコードである．April が提供すべき機能を明らかにするため，インターネット上で利用されているいくつかのプロトコル[1)~3),6),7)] とその実装について調査を行い，プロトコル処理部での処理内容について分析を行った．プロトコル処理部では次の処理を行う．

### 2.1 メッセージの解析と送受信

メッセージの送信時には，送信するデータ構造を明示的に文字列に変換し，それらの文字列をプロトコルに従って連結して送信する．たとえば，HTTP における Allow ヘッダでは，サーバが受理できるコマンド名をカンマによって区切った文字列としてクライアントに送信する．Apache ウェブサーバでは，受理できるコマンドをビットマップで管理しており，それを文字列に変換して連結するという処理を行っている．

メッセージの受信時には，受信したメッセージの構文解析を行い，メッセージ中に埋め込まれたパラメータなどの切り出しを行う．たとえば，HTTP における Allow ヘッダでは，カンマによって区切られたコマンド名を切り出し，プログラム中で使用しているデータ構造に変換して記録する．

メッセージの送受信にともなうデータ変換や文字列処理は，RPC や RMI におけるスタブやプロキシでの処理に相当し，単調で退屈なプログラミングを強いられる．特に，煩雑な文字列処理はプログラミング上の誤りを犯しやすい．そのうえ，プロトコル処理における誤りは，しばしば致命的なセキュリティ・ホールとなりうる[4),5)]．したがって，プログラマの負担を軽減しサーバの信頼性や安全性を向上させるためには，RPC や RMI と同様に，こうしたメッセージ処理に関わる処理をプログラマから隠蔽することが望ましい．

### 2.2 状態管理

アプリケーション層プロトコルでは，プロトコルの進行状況に応じてクライアントおよびサーバの状態が

遷移する場合が多い．各状態によって受け付けられるメッセージの種類が異なったり，同じメッセージでも処理内容が異なったりする．たとえば，メールの受信を行う POP プロトコルでは，ユーザ認証を行う認証状態（authorization state）とメールの受信を行う取引状態（transaction state）などがあり，各状態ではやりとりできるメッセージの種類が異なる．したがって，クライアント，サーバともにプロトコルの状態管理を適切に行い，受け付けられないメッセージは拒否するなど，状態に応じた厳密なエラーチェックを行っている．

### 2.3 例外処理

プロトコル処理部では，予期しない通信の切断やタイムアウトなどの通信障害に対処する必要があり，非同期的な例外処理を行う場合が多い．これらの例外処理は，オペレーティングシステムに依存した比較的低レベルなプリミティブを用いて実装されており，インターネット・アプリケーションの可搬性を低下させる要因となっている．また，プロトコル処理を行うコード中に例外処理のコードが混在するため，コードの可読性を低下させる要因ともなっている．通信時の例外処理は，通常時の処理とは分離して記述できることが望ましい．

## 3. April を用いる利点

April では，プロトコルの定義からプロトコル処理部を自動生成するというアプローチをとっている．April を用いる利点は以下のとおりである．

### 3.1 時系列に沿ったプロトコルの記述

April 言語では，プロトコルとして定められたメッセージのやりとりを時系列に沿った形で定義する．メッセージのやりとりが時系列に沿った形で明示的に定義されているため，プロトコルの意味が直感的に把握しやすい．また，ある時点で受け付けられるメッセージの種類が明確になるため，プロトコル違反を検出するコードが自動生成でき，エラーチェックのためのコーディングからプログラマを解放することができる．

### 3.2 文字列処理の軽減

メッセージのフォーマットは正規表現などを用いて宣言的に記述する．受信したメッセージを解析するコードはフォーマットの定義から自動的に生成できるため，メッセージ解析に必要な文字列処理をプログラマが明示的に行う必要がなくなる．

### 3.3 安全性の向上

プロトコル処理にともなう文字列処理からプログラマを解放することによって，文字列処理における誤りに起因したセキュリティ・ホールが生じにくくなる．これによってシステムの安全性向上が期待できる．

### 3.4 データ構造と文字列の半自動相互変換

メッセージの送受信を行うには，プログラム中で使用しているデータ構造とメッセージ中の文字列との相互変換を行う必要がある．April 言語ではプロトコル定義の一部として，メッセージに出現する文字列とプログラム中で使用するデータ構造との対応を宣言することができる．この対応付けによって，メッセージ中の文字列とプログラム中のデータ構造との相互変換がある程度自動化できる．

### 3.5 例外処理の簡潔な記述

予期しない通信の切断やタイムアウトなどの例外処理は，例外処理を行うハンドラを登録するだけで実現できる．通信の切断やタイムアウトを検出するためのコーディングを行う必要はなく，オペレーティングシステムに依存した低レベルなプリミティブを直接使用する必要はない．

帯域外（out-of-band）通信などの特殊なメッセージの送受信もプロトコル定義として定めることができる．帯域外通信を用いるように指定したメッセージは帯域外通信を用いて送信される．帯域外通信を受信した場合，対応するハンドラが自動的に起動される．

## 4. April

April を用いてプロトコル処理部を開発する場合，プロトコルを定義する言語である April 言語を用いてプロトコルを定義する．プロトコル定義から生成されたコードは，メッセージの送受信，それにともなう文字列処理，プロトコルの状態管理，通信障害の検出などを行う．アプリケーション・プログラマは，自動生成された手続き群を呼び出すだけでよい．4.1 節では April によるプロトコル定義について述べ，4.2 節では自動生成されるプロトコル処理部について述べる．

### 4.1 プロトコル定義言語 April

April 言語はアプリケーション層プロトコルにおけるメッセージのやりとりを定義する言語である．April 言語は RPC や RMI におけるインタフェース定義言語（IDL）に相当する．

April 言語によるプロトコルの定義は，1) 状態の定義部，2) 状態遷移の定義部，3) メッセージ・フォーマットの定義部，4) メッセージのやりとりの定義部という 4 個のセクションから成る．

1) の状態の定義部と 2) の状態遷移の定義部では，プロトコルの状態遷移をオートマトンとして定義する．たとえば，POP プロトコルでは認証状態から開

```
1:  /* protocol name */
2:  protocol pop3;
3:  /* definition of protocol states */
4:  states = {
5:        auth : INITIAL;
6:        transaction, retrieve;
7:        terminated : TERMINAL;
8:        error : TIMEOUT, ABORT;
9:  }
10:  /* declaration of valid state transitions */
11:  transitions = {
12:        auth −> transaction, terminated;
13:        transaction −> transaction, retreive, terminated;
14:        retrieve −> retrieve, transaction;
15:        ∗ −> error;
16:  }
```

**図 1** プロトコルの状態と状態遷移の定義
Fig. 1 Definition of protocol states and state transitions.

始し，取引状態，更新状態へと遷移し，最後に終了状態に達するという状態遷移を定義する．3) のフォーマット定義部では，メッセージに出現する部分文字列のフォーマットを定義する．たとえば，POP プロトコルで送受信されるユーザ名のフォーマットなどを定義する．4) のメッセージ定義部では，各状態においてやりとりされるメッセージを時系列に沿って定義する．たとえば，“USER kono\r\n” という文字列を送信し，“+OK\r\n” または “-ERR\r\n” を返信するというメッセージのやりとりを定義する．

ここでは，メール受信プロトコルの 1 つである POP を例にプロトコルの定義を示す．POP は実用的なプロトコルでありながら比較的簡潔であり，April 言語の特徴を示すのに適している．

#### 4.1.1 状態と状態遷移

April 言語によるプロトコルの定義は，プロトコル名の定義から始まり，状態と状態遷移の定義が続く．**図 1** に POP プロトコルにおける状態と状態遷移の定義を示す．この例では，auth，transaction，retrieve など 5 つの状態を定義している．また，コロン記号に続けて各状態の特性を定義している．**INITIAL** は開始状態を表し，この例では auth 状態が開始状態であることを定義している．同様に，**TERMINAL** は終了状態，**TIMEOUT** は通信がタイムアウトしたときに遷移する状態，**ABORT** は通信が切断されたときに遷移する状態であることを表す．

状態遷移の定義では，各状態間で可能な遷移を定義する．auth −> transaction，terminated は，auth 状態から transaction 状態および terminated 状態への遷移が可能であることを示す．∗ −> error はすべての状態（**TERMINAL** 特性などいくつかの特性を持つ状態を除く）から error 状態への遷移が可能であることを示す．April コンパイラは，開始状態は 1 つであること，開始状態からすべての状態に遷移可能であることなどを検査する．

#### 4.1.2 フォーマット定義部

プロトコルの状態と状態遷移の定義に続き，通信メッセージに出現する文字列のフォーマットを定義する．フォーマット定義部では，正規表現を用いてメッセージ中に出現する文字列の並びを定義する．ここで定義したフォーマットは，メッセージ定義部におけるメッセージ定義で使用する．April の対象とするプロトコルの多くは，そのフォーマットが正規表現で記述できる場合が多い．しかし，RFC におけるプロトコル定義は BNF 記法が用いられており，文脈自由文法を用いなければ記述できないものも稀に存在する．現状の April では実装を容易にするために正規表現を用いているが，BNF 記法を用いてフォーマットの定義を行えるようにすることは比較的容易である．

**図 2** にフォーマットの定義例を示す．各フォーマットは，フォーマット変数名，コロン，そのフォーマットの正規表現という形で定義する．図 2 では，line，host など 6 つのフォーマット変数を定義している．フォーマット変数 line は \r\n で終わる文字列を表し，フォーマット変数 host はホストの完全修飾名を表す．フォーマット変数 + と ! は特別な意味を持ち，それぞれメッセージの区切り文字，終端文字を定義するのに用いる．終端文字として定義された文字列は，特に指定のない

```
1: /* definitions of format variables */
2: formats = {
3:       line : *1('\n')  *([^'\n']|[^'\r']'\n') '\r\n';
4:       host : subdomain *('.' subdomain);
5:       greeting : '<' decimal0 '.' decimal1 '@' host '>' { pid:decimal0; time:decimal1; hostname:host; }
6:        + : ' ';        /* delimiter */
7:        ! : '\r\n';     /* terminator */
8:       subdomain : (['a'-'z' 'A'-'Z'])  *1(*(['a'-'z' 'A'-'Z' '0'-'9' '-') (['a'-'z' 'A'-'Z' '0'-'9']));
9: }
```

**図 2** フォーマットの定義部
Fig. 2 Definition of message formats.

限り各メッセージの終端に自動的に付与される．なお，フォーマット変数 greeting の定義中，{ と } で囲まれた部分はフォーマット変数に対応する型を定義する部分であり，その詳細は 4.2 節で説明する．

フォーマット変数は正規表現の中から参照することができる．図 2 では，フォーマット変数 greeting の定義中で，**decimal** および host という 2 つのフォーマット変数を参照している．host はユーザ定義のフォーマット変数であり，**decimal** は April 言語に組み込みのフォーマット変数である．フォーマット変数**decimal** は十進数の並びを表す．ほかに十六進数の並びを表す**hex**，バイトを表す**octet** などの組み込みフォーマット変数がある．

正規表現に用いる記法は，lex などで用いられる標準的な記法を若干拡張したものとなっている．繰返しを表す $*$ は，$n$ 回以上 $m$ 回以下の繰返しを表す $n*m$ という記法に拡張されている．$n$ を省略した場合は 0 を指定したと見なし，$m$ を省略した場合は無限大を指定したと見なす．また，カンマで区切った繰返しを表す $n\#m$ という記法を用意している．たとえば，$2\#3(X)$ という正規表現は，$X, X$ または $X, X, X$ にマッチする．なお，# による繰返しで用いる区切り文字は，必要に応じて別の文字列に定義することができる．

#### 4.1.3 メッセージ定義部

フォーマット定義部に続き，各状態でやりとりするメッセージの定義を行う．**図 3** に各状態でのメッセージ定義を示す．

POP は初期状態である auth 状態からプロトコルのセッションを開始する．図 3 に示したメッセージ定義について説明する前に，POP における APOP 認証の流れを説明する．APOP 認証では，最初にサーバからクライアントに挨拶メッセージとして，“+OK < プロセス識別子．時刻@ホスト名 >\r\n” を送信する．たとえば，“+OK<1896.697170952@cs.uec.ac.jp>” というメッセージを送信する．挨拶メッセージを受信した後，クライアントはサーバに “APOP ユーザ名ハッシュ値 \r\n” を送信する．このハッシュ値は < プロセス識別子．時刻@ホスト名 > とパスワードを連結した文字列から MD5 で求めたハッシュ値である．

図 3 の auth 状態の最初で定義されている start: <='+OK' + greeting は挨拶メッセージの送信を定義している．最初の start は，このメッセージの送受信に用いるメッセージ・ハンドラの名前である．メッセージ・ハンドラの役割については，4.2 節で述べる．メッセージ・ハンドラ名に続く <= は，サーバからクライアントにメッセージを送信することを表し，<= の後に送信するメッセージが続く．この例では，'+OK' + greeting というメッセージを送信する．これは，+OK に続き，区切り文字を挟んでフォーマット変数 greeting にマッチする文字列を送信し，最後に終端文字である \r\n を送信することを意味する．終端文字は自動的に付与されることに注意されたい．

フォーマット変数 greeting は '<' **decimal**$_0$ '.' **decimal**$_1$ '@' host '>' と定義されており（図 2 参照），フォーマット変数である**decimal** や host にマッチする文字列は一意に定まらない．そのため，メッセージの送信時に，それらのフォーマット変数にマッチする文字列を引数として受け渡す必要がある．POP における挨拶メッセージの場合，プロセス識別子，時刻，ホスト名を受け渡す．その詳細は 4.2 節で述べる．

図 3 では，挨拶メッセージの送受信後，クライアントからサーバに “APOP” + user + digest というメッセージを送信している．=> はクライアントからサーバに送信を行うことを示す．ここで送信している user および digest はともにユーザ定義のフォーマット変数である（図 2 では，user，digest の定義は省略している）．

```
1:  /* definition of valid message exchanges */
2:  state auth = {
3:        start: <= '+OK' + greeting;
4:        apop: => "APOP" + user + digest;
5:        ok: <= '+OK' + msg -> transaction;
6:      | err: <= '-ERR' + msg -> terminated;
7:  }
8:  state transaction = {
9:      {   /* STAT command */
10:       stat: => "STAT";
11:       stat_ok: <= '+OK' + decimal + decimal -> transaction; }
12:   | {   /* RETR command */
13:       retr: => "RETR" + decimal;
14:       retr_ok: <= '+OK' + decimal + decimal -> retrieve;
15:     | retr_err: <= '-ERR' + msg -> transaction; }
16:   | {   /* QUIT command */
17:       quit: => "QUIT";
18:       quit_ok: <= '+OK' + msg -> terminated;
19:     | quit_err: <= '-ERR' + msg -> terminated; }
20:   |     /* other commands are omitted */
21:     ...
22: }
23: state retrieve = {
24:       end: <= '.' -> transaction;
25:     | line: <=! line -> retrieve;
26: }
```

**図 3** メッセージの定義部
Fig. 3 Definition of message exchanges.

なお，シングル・クォートで囲まれた文字列は，大文字小文字を区別すること（case sensitive）を表し，ダブル・クォートで囲まれた文字列は大文字小文字を区別しない（case insensitive）ことを表す．

APOP コマンドを受信したサーバは，クライアントに認証の正否を通知する．図 3 に示したように，認証に成功した場合は '+OK' + msg というメッセージを送信する．メッセージの後に続く -> transaction は，メッセージの送信または受信後，transaction 状態に遷移することを表す．認証に失敗した場合は，'-ERR' + msg というメッセージを送信し，terminated 状態に遷移する．この例で示したように，パイプ記号を用いて複数のメッセージを選択的に送受信できる．

図 3 に示した transaction 状態におけるメッセージ定義では，{ と } を用いて，メッセージの送信とその返答を 1 つのグループにまとめ，STAT，RETR，QUIT の 3 種類のメッセージのやりとりを定義している．RETR コマンドに成功すると retrieve 状態に遷移し，メールの受信を行う．なお，retrieve 状態で用いている <=! は，メッセージの末尾に終端文字を付与せずにメッセージを送信することを表す．

### 4.2 プロトコル処理コード

April コンパイラはプロトコル定義から，クライアント，サーバそれぞれで使用するプロトコル処理部を生成する．各プロトコル処理部は，クライアントまたはサーバを実現するコードとともにコンパイルされ，April の実行時ライブラリとリンクされる（**図 4** を参照）．プロトコル処理部は，メッセージを送信する送信ハンドラ，メッセージを受信したときに起動される受信ハンドラ，例外発生時に起動される例外ハンドラから構成されている．

**図 5** に送信ハンドラ，受信ハンドラを用いたメッセージの送受信の様子を示す．この例では，図 3 の 4 行目のメッセージに対応する送信ハンドラ`apop_send()`と受信ハンドラ`apop_receive()`を用いて通信を行っている．送信ハンドラを呼び出すと，現在のプロトコルの状態でそのメッセージが送信可能かどうか検査が行われ，プロトコル定義に違反していなければ，メッセージを構成し送信を行う．

メッセージを受信するサーバ・プログラムでは，April 実行時ライブラリの提供する`april_dispatch()`関数を呼び出し，メッセージの受信待ちになる．プロトコ

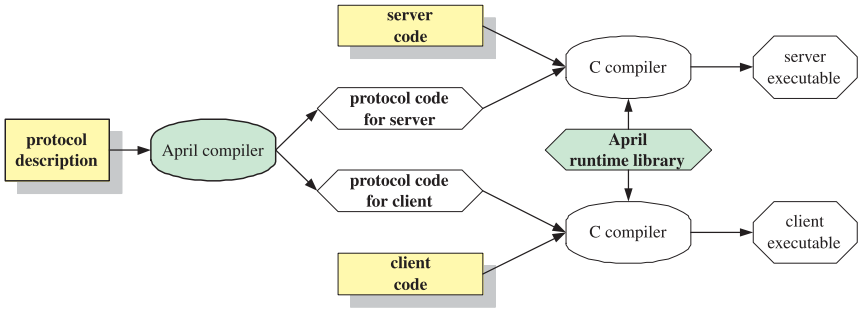


**図 4** April を用いたアプリケーションの作成手順
Fig. 4 Development procedures of application-level protocols using April.

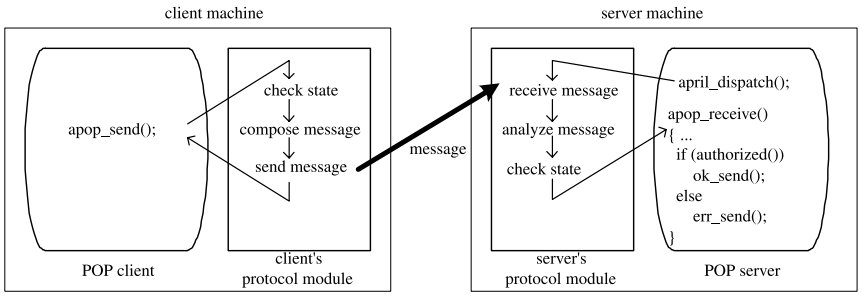


**図 5** April の実行時システムの構成
Fig. 5 Runtime architecture of April.

ル処理部では，メッセージを受信するとメッセージの解析を行い，現在のプロトコルの状態で受信可能なメッセージかどうか検査をする．プロトコル定義に違反していなければ，受信したメッセージに対応する受信ハンドラを起動する．以下，送信ハンドラ，受信ハンドラ，例外ハンドラの詳細について述べる．

#### 4.2.1 送信ハンドラ

送信ハンドラとは，メッセージの送信を行うハンドラである．送信ハンドラは，メッセージ定義から自動的に生成される C 言語の関数であり，送信ハンドラを呼び出すだけで，メッセージの送信を行うことができる．

送信ハンドラは，引数として，1) 通信チャネルと，2) 各メッセージに固有の引数をとる．通信チャネルとは通信路を抽象化したオブジェクトであり，送信ハンドラは引数で指定された通信路に対して送信を行う．メッセージに固有の引数は，メッセージに含まれるフォーマット変数に対応し，各変数に対応する文字列を受け渡すために用いる．図 3 の 4 行目から分かるように，送信ハンドラ`apop_send()` は，通信チャネルに加えフォーマット変数 user，digest に対応する引数をとる．

ユーザ定義のフォーマット変数は，デフォルトでは文字列型に対応する．たとえば，図 2 ではフォーマット変数 host に対応する型を定義していないので，host に対応する型は文字列型となる．April 言語に組み込みのフォーマット変数である**decimal** や**hex** は整数型に対応する．

フォーマット変数に対応する型は，必要に応じてフォーマット定義部で定義することができる．図 2 中のフォーマット変数 greeting では，対応する型として pid，time，hostname をフィールドとして持つ構造体を定義している．pid と time は，フォーマット定義中の $\text{decimal}_0$，$\text{decimal}_1$ にそれぞれ対応する．組み込みのフォーマット変数 decimal は整数型に対応するので，pid と time は整数型のフィールドとなる．hostname はフォーマット変数 host に対応し文字列型となる．なお，フォーマット定義中に正規表現の選択や繰返しがある場合，それぞれタグ付き共用体とリスト型に対応させることができる．

#### 4.2.2 受信ハンドラ

受信ハンドラは，メッセージの受信時に呼び出されるハンドラである．メッセージを受信するには，April の実行時ライブラリの提供する`april_dispatch()` を呼び出し，メッセージの受信待ちに入る．`april_dispatch()` は，複数のチャネルから受信できるよう

に，通信チャネルの配列を引数としてとる．また，アプリケーション・プログラムから受信ハンドラに任意の値を受け渡せるように，*void** 型の引数をとり，それをそのまま受信ハンドラに受け渡すようになっている．

プロトコル処理部では，受信したメッセージを解析し受信ハンドラを呼び出す．受信ハンドラはメッセージの解析結果を引数として受け取る．たとえば，“APOP” + user + digest というメッセージを受信すると，user および digest にあたる部分を切り出し，受信ハンドラ`apop_receive()` を呼び出す．

図 5 に示したように，受信ハンドラ`apop_receive()` 内では受信したパスワードを用いて認証を行う．認証に成功すれば`ok_send()` を呼び出して “`+OK\r\n`” を送信し，認証に失敗すれば`err_send()` を呼び出して “`-ERR\r\n`” を送信する．図 3 の 5，6 行目に示したように，“`+OK\r\n`” や “`-ERR\r\n`” の送受信は状態遷移をともなう．送信ハンドラ`ok_send()` では transaction 状態への遷移が行われ，`err_send()` では terminated 状態への遷移が行われる．メッセージを受信したクライアントでは，受信したメッセージに応じて transaction 状態または terminated 状態への遷移を行ってから，対応する受信ハンドラが呼び出される．

#### 4.2.3 例外ハンドラ

April では，通信の切断，通信のタイムアウト，帯域外通信の受信の 3 つを例外処理として扱う．通信の切断またはタイムアウトが起きた場合，それぞれ**ABORT**，**TIMEOUT** と指定した状態に遷移し，それぞれの状態に対応する例外ハンドラが起動される．これらの例外ハンドラは，状態名と同じ名前を持ち，例外ハンドラには例外の起きた通信チャネル，例外の発生した要因，例外の発生した状態名，送受信しようとしていたメッセージのハンドラ名が通知される．例外ハンドラが起動されると，処理中だった送信ハンドラや`april_dispatch()` の処理は中断される．

帯域外通信は，通常の =>，<= の代わりに ==>，<== を用いて送信する．帯域外通信を受信すると，`april_dispatch()` を呼び出していなくても，対応する受信ハンドラが起動される．受信ハンドラの実行が終了すると，受信ハンドラが起動される前の状態から処理を再開する．

通信セッションの正常終了を意味する**TERMINAL** 状態に遷移すると，後処理を行うために**TERMIANL** 状態に対応するハンドラが起動される．なお，このハンドラは状態名と同じ名前を持つ．

## 5. 評　　価

### 5.1 記 述 性

April によるプロトコルの記述力を検討するため，POP，HTTP，SMTP，FTP の記述を実際に行った．POP は 110 行，HTTP は 223 行，FTP は 156 行，SMTP は 138 行程度で記述できた．参考までに，POP の実装の 1 つである qpopper 3.0.2 では，メッセージの解析，状態管理，エラー処理などを行うコードは約 800 行程度である．

現状の April ではその定義が難しいプロトコルに IMAP がある．IMAP は複数のメッセージを非同期的に処理するため，同時に処理しているすべてのメッセージに対し，その返答が受理できるように仕様を記述しなければならない．そのため，状態数が増加し，現在の April ではプロトコルの自然な定義が難しい．これは，複数のメッセージが同時に処理されることを表すプリミティブを持たないためであり，新たなプリミティブを導入すれば解決できると思われる．複数のメッセージを同時には処理せずに，受信順に処理するようにすれば現在の April でも簡潔に記述できる．

アプリケーション層プロトコルのうち，IMAP のように非同期的な通信を行うプロトコルは稀であり，現在の April でも実用的なプロトコルを記述するのに十分な記述力がある．実際，POP，HTTP，SMTP は April を用いて自然に記述することができ，April の記述力が実用的なプロトコルを記述するのに十分であることを示唆している．

### 5.2 実行時性能

April によって自動生成されたプロトコル処理部の実行時オーバヘッドを調べるため，POP サーバの 1 つである qpopper 3.0.2 のプロトコル処理部を，April によって自動生成されたプロトコル処理部に置き換えて，性能の比較を行った．サーバ計算機には，Linux 2.4.17 の稼働している Athlon1.33 GHz プロセッサ，メモリ 512 MB を搭載した PC/AT 互換機を用い，クライアント計算機には PentiumII 400 MHz，メモリ 128 MB を搭載した PC/AT 互換機を用いた．サーバとクライアントは 100BASE-T スイッチイング・ハブによって接続されている．

**図 6** に実験結果を示す．横軸は受信したメールの個数，縦軸はメールの受信に要する時間（msec）を表す．メールの大きさは 1 通 4 Kbyte とした．このグラフから分かるように，April によって生成されたプロトコル処理部を用いても測定誤差程度のオーバヘッドしか生じていない．これは，プロトコル処理部での

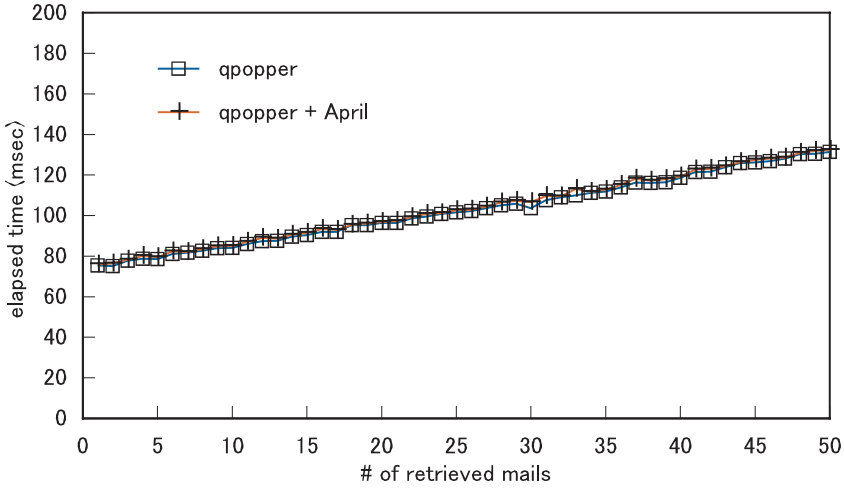


**図 6** メール受信時間の比較
Fig. 6 Comparison of the time needed for mail retrieval.

処理時間に比べ，ネットワークによる通信遅延やディスク・アクセスなどにかかる処理時間の方が相対的に長いためだと考えられる．April が対象とするアプリケーション層プロトコルでは，ネットワークによる通信遅延が生じるため，POP 以外のアプリケーションでも，April によって生じるオーバヘッドは些細なものであると期待できる．

## 6. 関連研究

### 6.1 プロトコル記述言語

April はプロトコルの定義からプロトコル処理コードを自動生成することによって，プログラマの負担を軽減する．プロトコル処理を行うコードを自動生成する言語システムは，これまでにもいくつか提案されている．

Prolac [8] や Promela++ [9] は，プロトコル・スタックの記述に特化したプログラミング言語であり，トランスポート層より低いレイヤのプロトコルの記述を目的としている．PLAN [10] は，プログラマブル・パケットのプログラミングを支援する言語である．また，PacketTypes [11] は通信パケットの仕様を記述する言語であり，パケット操作を必要とする低レイヤでのプログラミングを支援する．これらのシステムは，トランスポート層より低いレイヤを対象としているため，April とは対象とするネットワークのレイヤが異なっている．そのためアプリケーション層プロトコルの記述には適さない．

MSPL [12] は，April と同様にアプリケーション層プロトコルの仕様記述からプロトコル処理部を自動生成するシステムである．MSPL では，メッセージのフォーマット定義しか記述できず，プロトコルの状態や状態遷移は記述できない．そのため，プロトコルの状態管理やプロトコル違反の検査はすべてプログラマが行わなければならい．また，フォーマット定義の記述力も十分とはいえず，MSPL を用いてもメッセージの構成や解析のかなりの部分はプログラマの負担となってしまう．

プロトコルの形式的な仕様検証を目的としたプロトコル仕様記述言語に，プロセス代数に基づいた LOTOS [13] やその拡張である E-LOTOS [14]，有限状態機械に基づいた SDL [15] などがある．これらの仕様記述言語を用いて実用的なプロトコルの検証を行ったという報告[16] や，実用的なコードの生成を試みた研究[17),18] などがある．これらの仕様記述言語を用いるには理論上の制約を理解する必要があり，また自動的に生成されるコードも十分に効率が良いとはいえない．April は形式的な検証を目的としたものではなく，実用的なコードを自動生成することを狙ったプログラミング支援ツールである．

### 6.2 RPC と RMI

April は，ネットワークによる通信の詳細を隠蔽し，ネットワーク・プログラミングの負担を軽減する．CORBA [19] や Java RMI [20] などの遠隔手続き呼び出し（RPC）や遠隔メソッド起動（RMI）でも，通信の詳細をスタブやプロキシに隠蔽し，遠隔の手続きやメソッドでも通常のシンタックスを用いて呼び出せるようにしている．低レベルの通信を行うコードを生成するという点で，April は RPC や RMI に類似

した側面を持つ．しかし，RPC や RMI では通信に用いるメッセージ・フォーマットはプログラマから隠蔽されており，HTTP や SMTP などのアプリケーション層プロトコルの記述を行うことはできない．

## 7. ま と め

本論文では，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 フレームワークの提案を行った．April では，プロトコルの定義から，プロトコルに従ってメッセージの送受信を行うコードを自動的に生成する．April によるプロトコル定義の例として，POP プロトコルの定義例を示した．また，POP サーバの 1 つである qpopper のプロトコル処理部を April によって生成されたコードに置き換え，元来の qpopper との性能比較を行った．この性能比較では，April によって自動生成されたプロトコル処理部のオーバヘッドは無視できる程度であった．

April を用いて多くの実用的なプロトコルを定義することができ，April の適用範囲は十分に広い．実際，POP，HTTP，SMTP などの主要なアプリケーション層プロトコルを定義し，プロトコル処理部のコードを自動生成できる．しかし，IMAP のように非同期的な通信を行うプロトコルの記述には適しておらず，そうしたプロトコルも記述できるように拡張を行う必要がある．

また，April 言語はモジュール化の機構を持たないため，プロトコルに変更が行われるたびにプロトコル定義全体を変更しなければならない．アプリケーション層プロトコルに対する更新や改訂は頻繁に行われるため，そうした更新や改訂を反映しやすくする言語機構が必要であると考えている．

## 参 考 文 献


1) Fielding, R., Gettys, J., Mogul, J., Frystyk, H. and Berners-Lee, T.: Hypertext Transfer Protocol — HTTP/1.1, Internet Request for Comments (RFC) 2068 (1997).
2) Postel, J.B.: Simple Mail Transfer Protocol, Internet Request for Comments (RFC) 821 (1982).
3) Myers, J. and Rose, M.: Post Office Protocol — Version 3, Internet Request for Comments (RFC) 1939 (1996).
4) Aleph One: Smashing the stack for fun and profit (1996). http://www.shmoo.com/phrack/Phrack49/p49-14.
5) Baratloo, A., Singh, N. and Tsai, T.: Transparent Run-Time Defense Against Stack Smashing Attacks, *Proc. 2000 USENIX Annual Technical Conference* (2000).
6) Postel, J. and Reynolds, J.: File Transfer Protocol (FTP), Internet Request for Comments (RFC) 959 (1985).
7) Crispin, M.: Internet Message Access Protocol — Version 4rev1, Internet Request for Comments (RFC) 2060 (1996).
8) Kohler, E., Kaashoek, M.F. and Montgomery, D.R.: A Readable TCP in Prolac Programming Language, *Proc. ACM SIGCOMM*, pp.3–13 (1999).
9) Basu, A., Hayden, M., Morrisett, G. and von Eicken, T.: A Language-Based Approach to Protocol Construction, *Proc. ACM SIGPLAN Workshop on Domain Specific Languages* (1997).
10) Hicks, M., Kakkar, P., Moore, J.T., Gunter, C.A. and Nettles, S.: PLAN: A Packet Language for Active Networks, *Proc. 3rd ACM SIGPLAN International Conference on Functional Programming Languages*, pp.86–93 (1998).
11) McCann, P.J. and Chandra, S.: Packet Types: Abstract Specification of Network Protocol Messages, *Proc. ACM SIGCOMM*, pp.321–333 (2000).
12) Douglas, M.A.L. and Chan, P.K.: A Protocol Language Approach to Generating Client-Server Software, Technical Report, Department of Computer Sciences, Florida Institute of Technology (2000). CS-2000-2.
13) ISO 8807: Information Processing Systems — Open Systems Interconnection — LOTOS — A Formal Description Technique Based on the Temporal Ordering of Observational Behavior (1989).
14) ISO/IEC 15437:2001: Information Technology — E-LOTOS (2001).
15) ITU-T Recommendation Z.100: Specification and description language (SDL) (1999).
16) Pecheur, C.: Using LOTOS for specifying the CHORUS distributed operating system kernel, *Computer Communications*, Vol.15, No.2, pp.93–102 (1992).
17) Leue, S. and Oechslin, P.: On Parallelising and Optimising the Implementation of Communication Protocols, *IEEE/ACM Trans. Networking*, Vol.4, No.1, pp.55–70 (1996).
18) 寺島芳樹，安本慶一，中田明男，東野輝男，谷口健一：並行モバイルエージェント間でのマルチランデブチャネルの動的設定が記述可能な言語とその実装，情報処理学会研究会報告（2001-DPS-102）(2001).

19) Object Management Group: The Common Object Request Broker: Architecture and specification, 2.0ed., Technical Report, Object Management Group (1995).
20) Java Soft: *Java Remote Method Invocation Specification* (1997).
available from http://www.javasoft.com/.

(平成 14 年 7 月 29 日受付)
(平成 14 年 11 月 4 日採録)

**河野 健二**（正会員）

1970 年生．1997 年東京大学大学院理学系研究科情報科学専攻博士課程中退，同専攻助手に就任．理学博士．2000 年 1 月より電気通信大学情報工学科助手，現在に至る．1999 年 10 月より科学技術振興事業団さきがけ研究 21「情報と知」領域研究員を兼務．オペレーティングシステムおよびミドルウェア等のシステムソフトウェア，分散および並列処理に興味を持つ．IEEE/CS，ACM 各会員．平成 11 年度情報処理学会論文賞受賞．