社団法人　情報処理学会　研究報告
IPSJ SIG Technical Reports

2006－ＱＡＩ－18（７）
2006／1／20

IPノード動的構成方式の提案

狩野　秀一†　地引　昌弘†

† NECシステムプラットフォーム研究所
神奈川県川崎市中原区下沼部1753
E-mail: {karino@da, jibiki@bx}.jp.nec.com

あらまし　パケット処理コンテキストを動的に構成可能なプロトコルスタックを提案する．ネットワーク技術革新の加速と利用分野の拡大により，ネットワークプロトコルやその実装の変更頻度が高まっていることから，従来のモノリシックな構成に代わり，動的再構成を容易にするモジュール構成方式の必要性が高まっている．本稿では，各プロトコル実装をコンポーネント化することで動的に構成可能にするとともに，パケットごとにその処理コンテキストを動的に設定可能とする方式を提案する．本方式により，各モジュールの実装コストを低く抑えたまま，プロトコルスタックの拡張や構成変更を行うことができる．さらに，試作実装により従来の構成との性能比較を行い，有効性を検証する．
キーワード　プロトコルスタック，モジュール構成，制約解消

A Dynamic Construction Method for IP nodes

KARINO SHUICHI† and JIBIKI MASAHIRO†

† System Platforms Research Laboratories, NEC Corporation
1753 Shimonumabe, Nakahara-ku, Kawasaki-shi, 211-8666, Japan
E-mail: {karino@da, jibiki@bx}.jp.nec.com

Abstract　A protocol stack which can dynamically constructs packet processing contexts is proposed. There is a serious requirement for modular architecture of netowrk nodes which are usulaly implemented with monolithic style because there are many changes of network protocols and implementations in recent years. In this paper, we show a method that IP nodes can be dynamically constructed with protocols be implemented as components and packet processing context can be dynamically constructed per packets. With the method, you can be extend and modify protocol stacks keeping module implementation cost low. We compare performance of protocol stacks with each methods and evaluate efficiency of proposed method.
Key words　Protocl Stack, Modular Architecture, Constraint Resolution

1. まえがき

近年，ネットワーク技術革新の加速と利用分野の拡大により，ネットワークプロトコルやその実装の変更頻度が高まっている．このため，ネットワークノードには，変更／拡張を用意にするモジュール化された構成が採用されてきている．TCP/IPスタックについても，モジュール構成を実現する技術が開発されてきている[1], [2]．これらの技術ではモジュールの組み合わせ方をユーザーが指示する必要があり，複雑なノードを構成するには高度な判断と管理の手間がかかる．

本稿では，各プロトコルモジュールをコンポーネント化することで動的に構成可能にするとともに，パケットごとにその処理コンテキストを動的に設定可能とすることで，つぎの特長を有する方式を提案する．

- パケットに応じた処理モジュールの自動連結
- 無停止でのモジュールの追加／更新

本方式により，各モジュールの実装コストを低く抑えたまま，プロトコルスタックの拡張や構成変更を行うことができる．

さらに本稿では，上記方式にコンテキストキャッシュ機構を追加し，処理を高速化した試作ソフトウエアを用いて，同方式と従来の構成との性能比較を行い，有効性を検証する．

2. IPノードのモジュール化

遮断すべきトラフィックの増加や，不正なアクセスの増大により，企業網や家庭内ネットワークの入り口に位置するルータには，様々なセキュリティ機能が搭載されるようになっている．また，オーバレイ技術や，上位層に依存したルーティング等により，高機能／高品質な通信サービスを提供する技術の開発が

進んでいる [3]．一方，様々な攻撃手法が登場していることにより，プロトコルの実装は近年頻繁に更新されるようになってきている．

これらにより，ネットワークノードの実装は頻繁に拡張／更新が行われるようになってきている．従来，ルータやスイッチ等のファームウエアは，単一のコンポーネントとして実現されるモノリシックな構造で実装されてきたが，近年，拡張性や柔軟性を高めるためにそれらはモジュール化されてきている [4], [5]．これらに適用するモジュール化技術には，UNIX 等のマルチタスク OS のプロセスを利用するのが簡単だが，プロトコルスタックは一般に単一プロセスで実装されるため，プロセスを単位とするモジュール化は困難である．

2.1 モジュール化プロトコルスタック

プロトコルスタックのモジュール化機構として STREAMS [1], click modular router [2] が知られている．これらの機構によれば，プロトコルやその一部を実装したモジュールを連結することにより，プロトコルスタックの構成を容易に拡張，変更することができる．STREAMS では，図 1 に示すように，プロトコルをモジュールとして実装し，それを組み合わせてプロトコルスタックを構成できる．ストリームヘッドにつながるユーザープロセスから，新しいプロトコルモジュールを ioctl により push することでストリームへ繋げることができる．図 1 の例では，SCTP プロトコルをプロトコルスタックに追加している．

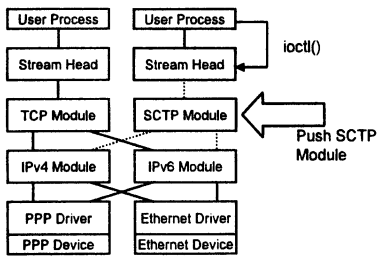

図 1　STREAMS におけるモジュールの構成

これらの機構によりプロトコルスタックを拡張するには，モジュールの連結順を，ノードの管理者またはユーザーがシステムコール等により明示する必要がある．この構成作業は，連結すべきモジュール数の増加や，モジュールの結合のしかたが複雑になると困難になり，また，設定間違い等により不具合を起こす危険が大きくなる．

たとえばトンネルプロトコルやパケットフィルタの追加／変更を行うには，同種類の他の機能との間でのパケット受け渡し順等の設定が複雑になり得る．IPv6 over IPv4 トンネルを例にとれば，configured tunnel [6], 6to4 [7], ISATAP [8] の 3 種類のトンネルが広く知られている．これらを単一ノード内に実装する場合，各々ヘッダ内の異なった位置の情報をもとに振り分けを行う必要があり（図 2），振り分け処理の実装や設定が複雑になる．たとえば STREAMS では，図 2 にあるような各種類のヘッダの組合わせをすべて認識してパケット振り分けを行うようなマルチプレクサを実装する必要がある．

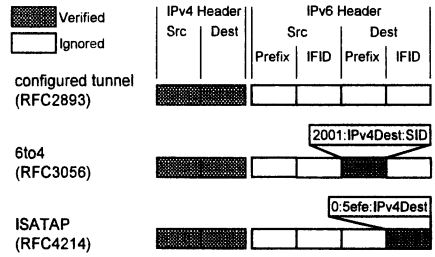

図 2　IPv6 over IPv4 トンネル終端（受信）の判定に必要なヘッダ情報

また，プロトコルスタックに既に組み込まれているプロトコルモジュールの機能を更新する場合には，従来の機構ではいったんモジュールを取り外して新しいモジュールを取りつける必要がある．このような処理を行うと，通信中のセッションが切断されるなど，サービスへ大きな影響を与えることになる．

今後，新規プロトコルの登場やモジュール構成の変更が頻繁になることが予想されるため，モジュールを自動的に連結してプロトコルスタックを構成することにより，プロトコルスタックの構成／変更コストを低減する技術が必要である．また，上記のような機構はサービス無停止でモジュールを更新できるべきである．

3. モジュール動的連結方式

モジュールを自動的に連結するには下記 2 点をあらかじめ定義しておく必要がある．

- モジュールが接続して送受信データや制御情報を受け渡せるようにモジュール間のインタフェースを規定すること
- モジュールがどのように連結すべきかを示す情報が与えられること

このうち前者の要件については，[1], [2] ですでに実現されている．後者の要件については，つぎの二つの技術の研究が進んでいる．

- 設計情報を詳細に記述してプログラムを自動的に生成する方式 [10]
- オントロジを用いて web サービスを自動合成する方式 [9]

前者の方式は，ソフトウエアの構成をすべて書き下す必要があり，モジュール組み合わせの複雑さによる開発のコストを減らせるとは言えない．後者の方式は，厳密に仕様に準拠した一貫したソフトウエアを合成する用途には不向きである．これらより，システム全体のモジュール構成を把握する必要がなく，かつ厳密にプロトコル処理を記述できるような，モジュールの自動連結方式が必要である．さらに，プロトコル処理は実時間性および処理効率の制約が大きいため，自動連結によるオーバーヘッドを低く押さえる必要がある．

これらを解決するため，本研究では，次のような特長をもつ効率的なモジュール自動連結方式を提案する．

- 各モジュールにメタ情報を用意し，該当情報をもとに，パケットごとにモジュールの連結順をきめる
- 各モジュールで処理可能なプロトコルをメタ情報として

定義する．

- 各モジュールのメタ情報と送受信データを制約として利用し，制約を解消すると処理可能な順にモジュールが並ぶようにする
- 計算したモジュール連結順（処理コンテキスト）をキャッシュすることにより，高速にパケット処理を行えるようにするとともに，無停止によるモジュール更新を実現する．

以降で，これらの各機能について説明する．

3.1 制約解消による連結順計算

モジュールを組み合わせてプロトコルスタックを自動構成するには，各モジュールからプロトコルスタックを構成するための情報を収集し，送受信データを処理可能なように組み合わせる必要がある．そのような情報を与えるには，モジュール間の依存関係を各モジュールのメタ情報に記載することが考えられる．しかし，モジュールの依存関係などを細かく指定する必要があると，その連結情報に記述ミスが発生しやすい．

このため本研究では，各モジュールどんなパケット／プロトコルを処理可能かをモジュールのメタ情報に記述し，各パケットに対して，それを処理可能なモジュールの連結順を計算することにより，自動連結を行う方式を採用する．この方式により，各モジュールのメタ情報から，他のモジュールとの関係に依存する記述を減らせるため，モジュール提供者への負担を減らすことができる．

また，メタ情報をプロトコルの処理可否により定義することで，完全に自由なメタ情報記述を許す場合と比して，一貫したパケット処理を行うのに必要なメタ情報を，比較的容易に記述できると考えられる．

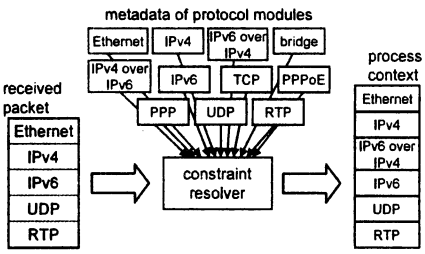

図 3　制約解消によるモジュール連結順計算

具体的には，次節にて表記を説明するメタ情報を用いて，処理対象パケット／データに対して処理可能なモジュールを検索して順に並べることにより，モジュールを自動的に連結する方式を採用する（図 3）．

このようにすれば，各モジュールが処理できるプロトコルの種類やパケットの部位をメタ情報に記載することにより，適切な位置にモジュールを追加できるため，プロトコルスタックの拡張や更新が容易になる．

3.2 メタ情報によるモジュール情報定義

メタ情報は，モジュールの連結順をユーザーの補助なく決定でき，さらに，該当モジュールがプロトコルスタック内のどの位置にあるべきかを厳密に示せることが必要である．処理対象のパケットがある場合には，該当パケットのどの部位を処理できるかにより，プロトコルモジュールの位置が決まる．このことから，メタ情報は，パケットのどの部分を処理できるかを表現することにする．

3.2.1 受信パケットがある場合

パケットの部位の表現に，プロトコルに依存した記法を導入するとメタ情報記述の拡張性が損なわれるため，プロトコルに依存しない方式として，本研究ではパケットのビット列パタンによりメタ情報を表現することとした．

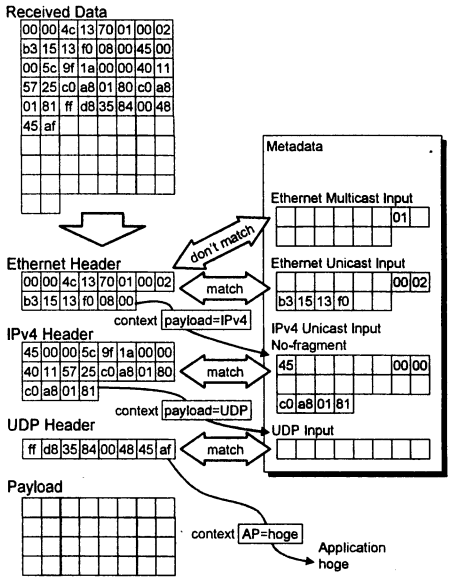

図 4　ビットパタンによるメタ情報定義

図 4 にビットパタンによるメタデータ表現の例と，具体的なパケットとのパタンマッチの例を示す．Ethernet フレームの受信処理判定として，ユニキャストフレーム用とマルチキャストフレームのメタデータがあり，受信パケットには，宛先アドレスのビット列より，ユニキャスト用がマッチする．また，Ethernet フレームのプロトコルタイプフィールドの値を以降の制約解消処理に利用するコンテキストとして保存しておき，IPv4 パケットの受信処理判定のときに利用する．このような処理を繰り返すことにより，受信パケットごとに正しい順で処理モジュールを連結することができる．

本研究では，メタ情報を XML ベースの記述言語で表現できるようにした．IPv4 受信モジュールのメタ情報の定義例を付録 1. に示す．また図 2 の例では，各トンネルを実装したモジュールごとに，各々のトンネルヘッダにマッチするビット列パタンをメタ情報に定義し，ビット列の最長マッチを行うようにすれば，STREAAM で必要とされるようなマルチプレクサを書かずにモジュールを追加できる．

3.2.2 受信パケットがない場合

一方，アプリケーションから送信を行う場合など，処理すべきパケットが存在しない場合は，送信側アプリケーション等が用意するコンテキストと，メタ情報を照合して処理可能性を判

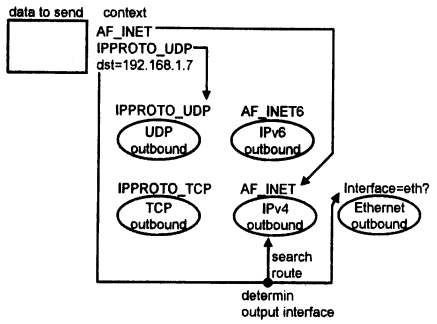

図 5　アプリケーションが与えるコンテキストを利用したメタ情報定義

定することとした．

すなわち，たとえばバークレイソケットを用いるアプリケーションが AF_INET, SOCK_DGRAM を引数に指定して socket システムコールを発行すれば，UDP, IPv4 等のモジュールにはマッチするが，TCP, IPv6 等のモジュールはマッチしない（図 5）．

3.3　連結順キャッシュ

3.3.1　高　速　化

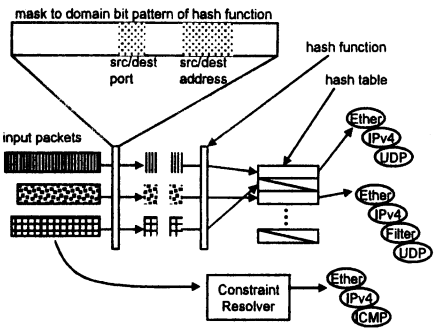

図 6　連結順キャッシュによる処理高速化

前節の方式では，パケットごとにモジュールの連結順を計算することから，パケットごとにモジュール連結処理が行われ，パケット処理におけるプロトコルスタック自動構成のオーバーヘッドが非常に大きくなると予想される．

これを解決するため，同一のモジュール列により処理できると見られる一連のパケットについては，一度計算したモジュール連結順をキャッシュしておき，毎回その順にモジュールを連結してパケットを処理する高速化手順を採用する．

図 6 に処理の流れを示す．入力されたパケットの所定の部分にマスクをかけ，ビット列を取り出す．取り出したビット列をハッシュ関数にかけてハッシュ値を得る．このハッシュ値ごとに連結順を記録しておけば，同一ビットパタンのパケット列に対しては高速に処理コンテキストを構成できる．

本方式により，たとえば TCP コネクションごとに処理コンテキストをキャッシュすることができる．特定の TCP コネクションのパケット処理をキャッシュするには，IP ヘッダのアドレスとプロトコル番号，L4 ヘッダのポート番号フィールドをマスクすればよい．いったん TCP コネクションが確立してしまえば同一モジュール列でパケットを処理できるため，毎回連結順を計算するよりもオーバーヘッドを大幅に削減できる．

3.3.2　無停止更新

また，プロトコルモジュールがセッション状態などを保持している場合，モジュール更新によりその状態が失われてセッションが切断されるような不具合も，本機構で解決できる．

すなわち，キャッシュしているモジュール連結順にモジュールの実装への参照を保持しておく．新しいモジュール実装が追加されるときは，古いモジュールはそのまま残しておき，メタ情報から参照できるモジュールのみ新しい方に更新する．

既設のセッションに属するパケットついては，キャッシュがヒットすることにより古いモジュールが使われるため，既存セッション状態を使って処理できる．新しいセッションについては新規にモジュール連結順が計算されるため，新しいほうのモジュールが使われる．このようにすれば，モジュール更新を無停止で実現できる．

4.　実　　装

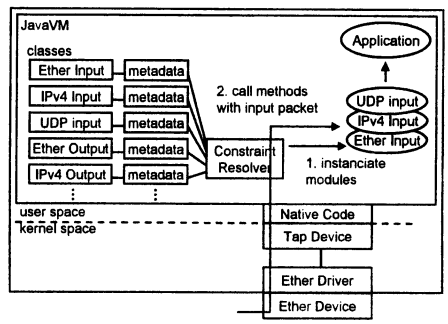

図 7　試作実装の構成

モジュールを動的に連結可能な IP ノードを Java により実装した（図 7）．各プロトコルモジュールを動的にロード／連結可能なクラスとして表現し，該当クラスにメタ情報を付与することにより，プロトコルモジュールとして扱えるようにした．

- プロトコル単位を目安に，モジュールをクラスとして実装した．各モジュールは，パケット処理用のメソッドを持ち，コンテキストにしたがってモジュール間でパケットを受け渡せるよう，インタフェースを定義した．
- 各モジュールに対応したメタ情報を XML ベースのメタ情報記述言語で記載できるようにし，JavaVM 内からそれを読み込んで制約解消エンジンに保持できるようにした．
- 受信したデータと，各モジュールのメタ情報から，モジュールの連結順を計算する制約解消エンジンを実装した．
- 同エンジンにより，モジュール連結順が決定されると，各モジュールのインスタンスを生成し，処理対象パケットを各インタフェースのパケット処理メソッドに順に受け渡すことにより，処理を実行するようにした．
- JavaVM にてプロトコルスタックを実装するため，パケットの読み書きを行うインタフェースをネイティブコードにより実装した．

5. 評　　価

前記の実装を利用して提案方式の評価を行った．提案方式では，モジュールの動的連結を送受信の都度行うため，静的に結合されたプロトコルスタックよりも送受信処理のオーバーヘッドが大きい．本説では，このオーバーヘッドを送受信性能の実測により評価し，提案方式の性能上の有効性を検証する．

5.1 評価シナリオ

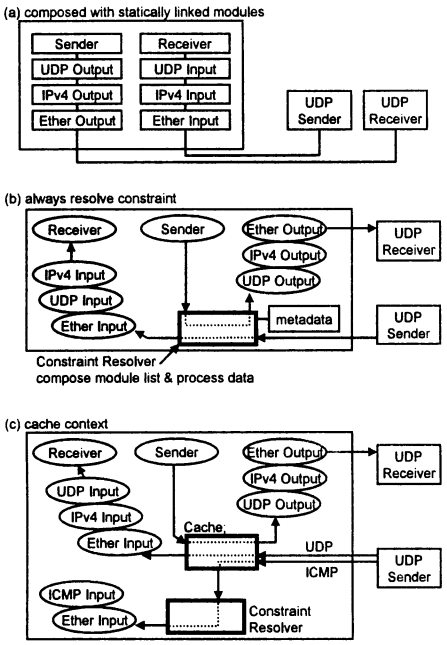

図 8　評価シナリオ毎のシステム構成

次の 3 シナリオについて性能評価を行った．

(a)　モジュールを静的に連結する

(b)　パケットごとに毎回モジュールの連結順を計算する

(c)　モジュール連結キャッシュを利用する

各々のシナリオで評価に用いたシステムの構成を図 8 に示す．前節で説明した実装ベースに上記各シナリオ用のターゲットを実装した．プロトコルモジュールとして，TinyIP [11] 程度の機能を有する UDP, IP, Ethernet 各々の送信および受信モジュールを用意した．また，ターゲットとの間で UDP データグラムの送受信を行う対向ノードを用意した．ターゲットノードおよび対向ノードの諸元は各々表 1 の通りである．上記環境にて，

表 1　測定ノードの諸元

項目	ターゲット	対向ノード
CPU	Pentium3 800MHz	Celeron 2.4GHz
メモリ	512Mbytes	512Mbytes
OS	Fedora Core 3	Fedora Core 3
JavaVM	J2SE 1.4.2_08	なし

ターゲットと対向ノード間で UDP データグラムの送受信を行い，ターゲットの送信および受信性能を次の手順で測定した．

- パケットを連続して送受信し，その送受信にかかった累計時間を測定する．1 回の測定における送受パケット数は 10 万〜100 万パケットとした．
- UDP データグラム長 64, 256, 1024, 1472 バイト各々の場合について上記測定を行う．
- 上記各測定を，条件ごとに 10 回行い，その平均と標準偏差を求めた．

5.2 結　　果

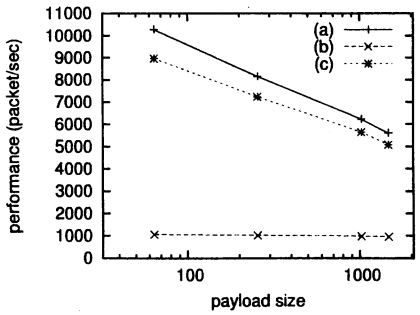

図 9　性能測定結果（送信）

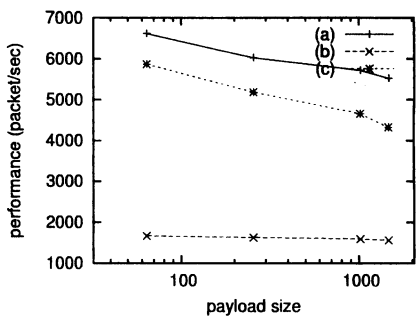

図 10　性能測定結果（受信）

本評価の測定結果を図 9（送信），図 10（受信）に各々示す．図から読み取れるように，キャッシュ機構を導入した提案方式のプロトコルスタック (c) は，静的にモジュールを連結したプロトコルスタック (a) と比較して，性能の低下が 15%程度にとどまっていることがわかる．このことから，Java 言語によるプロトコルスタックの実装では，提案方式は十分実用的な水準で利用可能であることが期待できる．

6. む　す　び

モジュールを自動的に連結して動的に構成可能なプロトコルスタックを提案した．従来，モジュールの連結は手作業により行う必要があり，複雑な構成や頻繁な拡張を行ううえで，ユーザーに負担がかかる課題があった．本稿では，モジュールが処理可能なプロトコル情報をメタ情報として用意し，これを用いて送受信データごとに自動的にモジュールを連結する方式を提案した．この方式により，モジュールの追加／更新時にはモ

ジュールに特化した情報をメタ情報として記載するだけで，自動的なプロトコルスタック再構成が可能となり，プロトコルスタックの拡張／更新のコストを低減できる．さらに，連結順をキャッシュする方式を導入し，モジュールを連結するための制約解消計算のオーバーヘッドを低減するとともにモジュールの無停止更新を実現できるようにした．同方式を利用した実装と，静的にモジュールを連結した実装との性能比較を行い，性能差が実用的に見て許容できる程度であることがわかった．

今後は，プロトコルモジュール内の制御用メッセージ流通機構等の検討を進める予定である．

謝辞 本研究は総務省の委託研究「次世代バックボーンに関する研究開発」プロジェクトの成果である．

文　献

[1] Dennis M. Ritchie, "A Stream Input-Output System," AT&T Bell Laboratories Technical Journal, 1984.

[2] Eddie Kohler, Robert Morris, Benjie Chen, John Jannotti, and M. Frans Kaashoek. "The Click modular router," ACM Transactions on Computer Systems 18(3), August 2000, pages 263-297.

[3] Cisco Systems, "Cisco Application-Oriented Network," White Paper, http://www.cisco.com/application/pdf/en/us/guest/products/ps6438/c1244/cdccont_0900aecd8033e9a4.pdf

[4] エクストリームネットワークス, "革新をもたらす ExtremeWareXOS," White Paper, http://www.extremenetworks.co.jp/download/Whitepaper/XoS_WP.pdf

[5] Cisco Systems, "Requirements for Next-Generation Core Routing Systems," White Paper, http://www.cisco.com/en/US/products/ps5763/products_white_paper09186a008022da42.shtml

[6] R. Gilligan and E. Nordmark, "Transition Mechanisms for IPv6 Hosts and Routerso," RFC2893, August 2000.

[7] B. Carpenter and K. Moore, "Connection of IPv6 Domains via IPv4 Clouds," RFC3056, February 2001.

[8] F. Templin, T. Gleeson, M. Talwar and D. Thaler, "Intra-Site Automatic Tunnel Addressing Protocol (ISATAP)," RFC4214; October 2005.

[9] S. McIlraith, T. Son, and H. Zeng. Semantic Web services. In IEEE Intelligent Systems (Special Issue on the Semantic Web), March/April 2001.

[10] Object Management Group, Model Driven Architecture, http://www.omg.org/mda/

[11] 楯岡孝道，阿部公輝，"教育用簡易 UDP/IP スタック TinyIP の設計と実装,", 信学論, Vol J86-B No.8 pp.1553-1560, Aug, 2003.

付　録

1. メタ情報の例

IPv4 パケット受信処理モジュールのメタ情報

```
<メタデータ モジュール名="IPv4 受信" バージョン="1">
  <!-- このメタデータの制約解消計算に必要な情報 -->
  <必要情報>
    オフセット
    ペイロード情報
  </必要情報>
  <!-- このメタデータが示すモジュールが
       この受信データを処理できるかどうかを
       判定する条件 -->
  <処理対象判定>
    <and>
      <bit-data オフセット="0" 長さ="4"> 0010 </bit-data>
      <添付情報 key="ペイロード情報">
        <プロトコル> IP </プロトコル></添付情報>
    </and>
  </処理対象判定>
  <添付情報>
  <!-- このメタデータが示すモジュールが
       この受信データを処理した結果添付される情報
  -->
    <switch>
      <!-- 宛先アドレス -->
      <byte-data オフセット="16" 長さ="4"/>
      <case value="192.168.1.129">
        <switch>
          <!-- フラグメントオフセット -->
          <bit-data オフセット=51 長さ="13">
          <case value="0">
            <ペイロード情報>
              <プロトコル><switch>
                <!-- ペイロードプロトコル -->
                <byte-data オフセット="9" 長さ="1"/>
                <case value="0x01"> ICMP </case>
                <case value="0x04"> IP-in-IP </case>
                <case value="0x06"> TCP </case>
                <case value="0x11"> UDP </case>
                <case value="0x41"> IPv6-over-IP </case>
              </switch></プロトコル>
            </ペイロード情報>
            <プロトコルヘッダ
                オフセット="0" 長さ="20"/>
            <オフセット 演算="加算">
              <bit-data
                オフセット="4" 長さ="4" シフト="4"/>
              </オフセット>
            </case>
            <case value="otherwise">
                <リアセンブルフラグ> </case>
          </switch>
        </case>
        <case value="otherwise"> <転送フラグ> </case>
      </switch>
    </添付情報>
</メタデータ>
```