解説論文

こんなツール，使っています

SDN のプログラマブルデータプレーンによるネットワーク主導型 TCP フロー制御調和手法

Network-driven TCP Harmonized Management Utilizing a Programmable SDN Data Plane

妙中雄三 Yuzo Taenaka† 塚本和也 Kazuya Tsukamoto††

Summary Software Defined Networking（SDN）技術の一つであるプログラマブルデータプレーンによってネットワーク内で任意のパケット処理を実装できるようになった．ネットワーク内では，一つの端末では得られない，競合する通信フローの情報が容易に把握できるため，プログラマブルデータプレーンでその情報を活用して任意の新しいパケット制御を実現できれば，端末のみでは難しかった競合フローを直接的に意識した通信制御の実現が期待できる．本研究では，アプリ要求を意識した TCP フローの競合制御に着目する．一般的な TCP では，競合フローとのスループットの公平性のみを目指すが，実際にはアプリ要求との不一致が生じる．つまり，競合フロー間でスループットが公平であっても，一方のアプリ要求には過剰，若しくは不足する，といった非効率な資源利用状態に陥る．この不一致解消には，アプリ要求と競合関係を理解し，スループットに意図的な傾斜を付ける制御が必要となる．本稿では，プログラマブルデータプレーンとその開発言語の P4 を解説し，上記の制御の実現に向けた初期検討として，競合する 2 フローの TCP スループットを事前に定義したアプリ要求に合わせて調和させる TCP フロー制御調和手法を試作する．

Key Words TCP フロー制御，制御調和，P4，プログラマブルデータプレーン，Software Defined Networking

1 はじめに

ネットワークをソフトウェアで制御可能にする Software Defined Networking（SDN）が広く知られるようになった．一般的なネットワークは，ネットワークスイッチ（以降，「スイッチ」）を用いて，スイッチに事前実装された固定のネットワーク機能のみを用いて構成されている．これに対して，1990 年代からの任意の機能をネットワーク内機器に埋め込み，パケット転送途中で実行するアクティブネットワークの研究を起点に，スイッチの通信制御機能を分割・分離してそれぞれをプログラマブルにする SDN の研究が現在まで活発に行われている [(1)]．

近年用いられる SDN のスイッチの概念図を図 1 に示す．スイッチはコントロールプレーンとデータプレーンによって構成される．コントロールプレーンは IP 層の経路制御や，データリンク層の MAC アドレスの学習と学習結果に基づく送信先の物理ポートの決定などの意思決定の役割を担い，データプレーンはコントロールプレーンからの指示に従って MAC アドレスの書換えや，物理ポートへのパケット送出等のパケット処理実行の役割を担う．従来のスイッチでは，このコントロールプレーンとデータプレーンはハードウェアに統合された上で専用機能として実装される．これに対して，SDN ではコントロールプレーンとデータプレーン間で情報や命令を交換する方法（インタフェース）を規定し，そのインタフェースに従う限り，コントロールプレーンとデータプレーンを自由に選択できるようになった．また SDN は，スイッチごとにコントロールプレーンを導入する分散形と，複数のコントロールプレーンを一つに集約して複数のデータプレーンを一括制御する集中形に分類される．SDN 技術の一つである OpenFlow [(2)] の登場により，集中形のコントロールプレーンが主に使われるようになった（図 1 右側）．データプレーンごとに独立して意思決定していた分散形に対して，集中形では複数のデータプレーン，つまりネットワーク全域で一貫した意思決定が可能となった．

OpenFlow ではコントロールプレーン／データプレーンのインタフェースと，そのインタフェースを介し

† 奈良先端科学技術大学院大学，生駒市
Nara Institute of Science and Technology, Ikoma-shi, 630-0192 Japan
†† 九州工業大学，飯塚市
Kyushu Institute of Science and Technology, Iizuka-shi, 820-8502 Japan

ⓒ 2024 電子情報通信学会

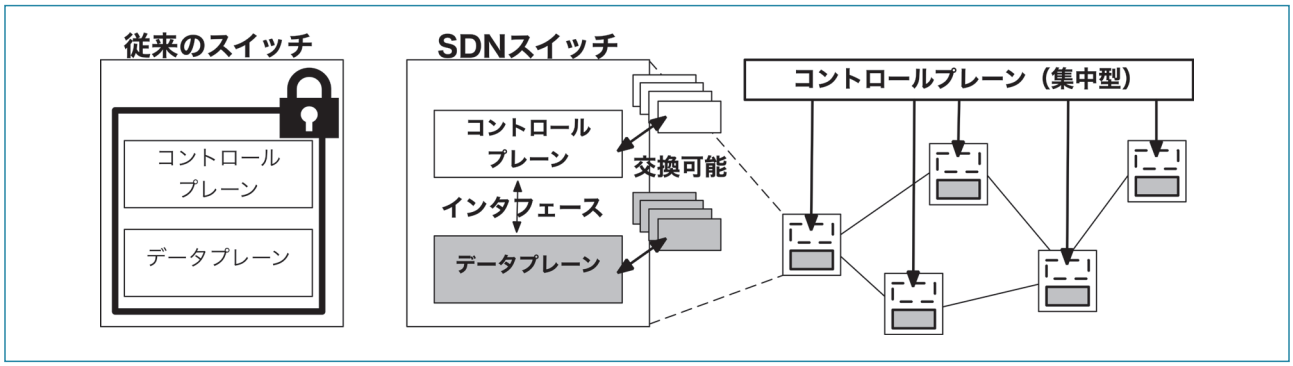

図 1　SDN の概念図

て交換するデータ形式が規定され，特に汎用的なパケット制御指示方法として「フロー単位のマッチ・アクション」が考案された．これはパケットヘッダに含まれる送信元や宛先などのマッチ条件にマッチしたパケットに対して，対応するアクションを実行する仕組みである．マッチ条件とアクションのペアは制御命令として十分な記述性を備えていたため，OpenFlow に限らず SDN の制御指示方法として広く普及した．ただし，OpenFlow は研究の概念実証を元にした技術であり，マッチ条件は一部のプロトコルのヘッダ情報のみを前提とし，アクションは事前定義された方法から選択するように設計されていたため，新しいプロトコルや任意のアクションを扱えない等の問題が残った．そこで，OpenFlow で得られた知見を基に任意のマッチ条件の指定と任意のアクションを実行できる仕組みを備えるスイッチのデータプレーンアーキテクチャと，そのデータプレーンでのパケット処理機能をプログラムする言語 P4（Programming Protocol-Independent Packet Processors）[(3)] が提案された．

筆者らは，上記のプログラマブルなデータプレーン上でのパケット転送途中の処理に着目し，ネットワーク内部だからこそ実現できる「競合する複数の通信フローの TCP フロー制御を調和させる手法」に取り組んでいる．TCP は，一般的に端末が接続する無線 LAN 等の狭い帯域を保持するアクセスネットワーク（ボトルネックリンク）でのほかの競合 TCP フローとの公平性を間接的に意識しながらパケット送信を制御するが，端末は，競合通信フローの存在や制御状況を把握して直接的に意識した制御はできない．つまり，競合フローを直接的に意識して制御するためには，複数フローの競合状況や TCP の制御状況を直接的に観測できるネットワーク内部での制御が欠かせない．加えて，近年では様々な種類のアプリが利用され，アプリによるネットワークへの要求が多様化しているものの，端末のオペレーティングシステム（OS）が提供する TCP 通信機能はアプリによらず共通の通信制御を提供する．つまり，複数の端末が生成する必要なスループット値（アプリ要求）が異なるフローが競合すると，TCP がスループットを公平となるように制御することで，アプリ要求を満たせないフローが現れてしまう．そこで，プログラマブルデータプレーンを活用して，ボトルネック区間の競合フローの存在や通信制御状況を直接的に観測し，観測結果とフローごとのアプリ要求に合わせて意図的に送信量を不均一にする競合フロー間の調和制御，例えば過剰なスループットのフローのパケット送信量を適切に抑制し，その抑制した分でスループットが不足する競合フローのパケット送信量をアプリ要求を満たすまで増やすことができれば，全てのアプリ要求を満たせる可能性がある．

P4 を用いて TCP 制御を扱う研究はこれまで多数提案されている [(4)]．本研究と同様にネットワーク内部から TCP 制御する手法も多く提案されているが，制御対象とする一つのフローの TCP スループットを向上させる一般的な TCP 制御の研究にとどまり，多様性が増すアプリ要求を満足するための調和制御手法は検討課題として残されている．したがって，本研究ではプログラマブルデータプレーンを用いてアプリ要求に応じた TCP フロー制御調和手法の実現可能性を示す．

2　プログラマブルデータプレーン

従来のスイッチはアーキテクチャ，ハードウェアとソフトウェアがスイッチごとに異なり，同一処理を実装する場合でもスイッチ単位の個別開発が必要であった．SDN ではスイッチのアーキテクチャを共通化し，P4 言

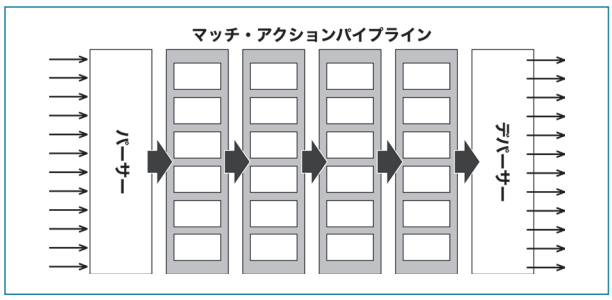

図２　PISA のパイプラインアーキテクチャ

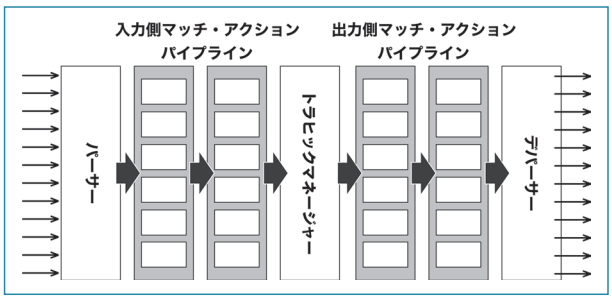

図３　V1Model のアーキテクチャ

語で記述することで，共通のプログラムコードで多様なスイッチで同一処理を実行可能とした．3．で提案する調和制御手法の具体的な実装詳細の理解を助けるために，本章ではプログラマブルデータプレーンのアーキテクチャと，そのデータプレーンを高級言語でプログラミング可能な P4 について解説する．

2.1　パイプラインアーキテクチャ

スイッチのデータプレーンでは複数のパケット処理を並列実行するための多数のパイプラインを持ち，マッチ・アクションを実行できるプログラマブルなパイプラインを備えたリファレンスアーキテクチャが Protocol Independent Switch Architecture（PISA）と呼ばれている．図２に PISA のアーキテクチャを示す．PISA は，パーサー，マッチ・アクションパイプライン，デパーサーの三つの要素で構成され，全てプログラマブルである．パーサーは，スイッチに入力されたパケットを開発者が作成したプログラムコードに従って解析して，以降のパイプラインに利用するための情報をパケットごとのメタデータとして格納する．マッチ・アクションパイプラインは，パケットのメタデータに基づいて，指定された条件にマッチするかパケットごとに検証し，マッチするパケットに対してプログラムしたアクション（パケット処理）を実行する．デパーサーは，メタデータからパケットのヘッダを再構成し，指定された物理ポートから処理済みのパケットを送出する．ただし，このアーキテクチャは P4 言語が想定するパイプライン処理の概念モデルであるため，このモデルにのっとった実際のスイッチが必要である．

そこで，プログラマブルデータプレーンを実際に用いてネットワーク機能を試作・検証するために，PISA で示されたパイプラインを備える V1Model というアーキテクチャを採用した Behavioral Model version 2（BMv2）と呼ばれるソフトウェアスイッチが開発・活用されている [5]．V1Model のアーキテクチャを図３に示す．V1Model では，入力側と出力側でそれぞれマッチ・アクションパイプラインがあり，その間にトラヒックマネージャと呼ばれる固定機能が配置される．スイッチでは，パケットのコピーや送出先ポートへのパケットのキューイングなど，定型的なパケット処理が多数あるため，あらかじめトラヒックマネージャがそれらの処理機能を備えることで，入力側のマッチ・アクションではその実行指示のみをする構成となっている．マッチ・アクションパイプラインが入力側と出力側で分かれているのは，マルチキャストやブロードキャスト等のスイッチが受信した後に複製し，複数のポートへ送出するパケットについて，入力側でパケットを受信した時点で共通の処理を実行し，パケットを複製した後に出力側で送出先ごとに異なる処理を実行できるようにしているためである．

2.2　P4：Programming Protocol-Independent Packet Processors

P4 は 2．1 で説明したパイプラインアーキテクチャ PISA において，パケット処理をプログラムするための高級言語である．通常，スイッチは CPU や FPGA，ASIC などの複数のハードウェアを組み合わせて構成されており，スイッチごとに適合する専用のプログラムを開発する必要があるが，P4 ではプログラムコードにほとんど変更を加えることなく多様なハードウェアに対応可能となっている．例えば，C 言語にも同様の機能が備わっており，CPU には x64 や armv8 等があるものの，gcc 等でコンパイルするときに対象を指定するだけで，プログラムコードの改変なくハードウェアに合わせたバイナリーコードを生成できる．この概念と同様に，スイッチのハードウェアの違いを考慮する必要がなく，共

図４　P4 プログラムのテンプレート

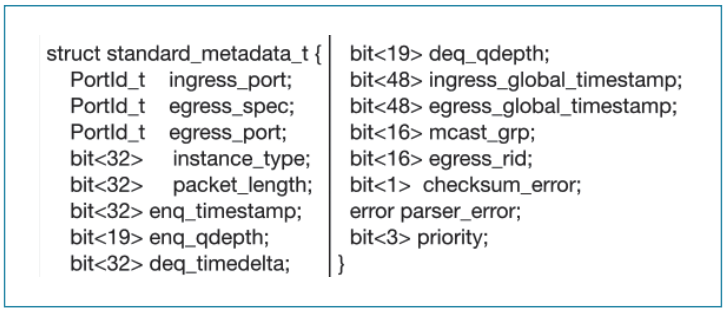

図５　standard_metadata の定義

通の言語でデータプレーンのプログラムコードを作成できるのが P4 の利点である．

図４に BMv2 を対象とした P4 プログラムのテンプレートを示す．詳しい記述方法は P4 言語仕様 [6] を参照されたい．ここでは詳細を省いた概念レベルで概説する．このプログラムコードは BMv2 が採用する V1Model を想定しているため，冒頭で V1Model に対応するヘッダファイル v1model.p4 [7] をインクルードしている．また，P4 ではプロシージャを parser ブロック（{ と } で囲まれた範囲）や control ブロックで定義している．

図４の右の最下部の V1Switch を見ると，六つのプロシージャが並んでいる．V1Switch は C 言語の main 関数に相当し，図３にあるプログラマブルなパイプラインの順番（パーサー，入力側マッチ・アクション，出力側マッチ・アクション，デパーサー）に対応する処理内容を記述したプロシージャを指定している．ただし，図３には明記していないが，V1Model ではチェックサムの検証（MyVerifyChecksum）がパーサーの後，チェックサム更新（MyComputeChecksum）がデパーサーの前にある．これは V1Model 独自のパイプラインで，チェックサム検証・更新を実行するプロシージャを指定する．本稿ではこれらチェックサム処理の説明は省略する．

パケット処理の流れを説明する．パケットを受信する度に図３のパーサーに対応する，パーサーの処理を記述したプロシージャ（図４左側の MyParser）を実行する．パーサーでは受信したパケットを解析し，その結果は V1Model ではパケットごとに付与するメタデータとして，図５に示す standard_metadata_t 構造体に格納する．この構造体のメンバ変数には，解析結果以外にもトラヒックマネージャの制御内容（例えば，送出先ポー

トの番号など）を指定する変数が含まれている．ほかにもメンバ変数が複数あり，詳しい情報はBMv2のドキュメント[8]を参照されたいが，例えば，Ingressが付くメンバはパケット入力に関連する変数で，Egressが付くとパケット出力に関連する変数である．次に，パーサーの次は入力側のマッチ・アクションの処理内容が書かれた（図4左側下部のMyIngress）が実行される．ここではパケットに対して処理を実行する．例えば，standard_metadata_t構造体のメンバ変数ingress_portからパケットが到着したスイッチの物理ポートの番号を把握し，egress_specにパケットを出力する物理ポートを指定できる．これによって，図3の入力側マッチ・アクションの次にある，トラヒックマネージャが指定された物理ポートのキューにパケットを挿入する．同様に出力側のマッチ・アクション（図4右側のMyEgress）を実行し，最終的にはデパーサー（図4のMyDeparser）がstandard_metadata_t構造体のデータに基づいてパケットを再構築して送出する．このようにP4では，パーサーやパケット処理の内容を高級言語で記述し，各パイプラインに対応して実行させる構造となっている．

2.3　P4を用いたマッチ・アクションの実装

P4を用いてマッチ・アクションを実装する方法を解説する．本節では，P4のチュートリアルで紹介されているIPフォワードを行うL3スイッチ（Basic Forwarding[9]）の実装例であるbasic.p4を用いて説明する．

P4では，最初にマッチ条件で使うパケットのメタ情報をパーサーで抽出する．図6にパーサーのコードを示す．parserブロックは，startという状態から始まり，acceptかrejectの2種類の終了状態に到達するまでの状態遷移を記述する．その間に経る状態は，stateステートメントで任意に記述できる．図6ではイーサネットフレームの入力を前提として，start状態からparse_ethernetへ遷移し，イーサネットヘッダの情報を抽出している．その後，EtherTypeフィールドに応じてaccept状態とするか，IPv4ヘッダの情報を抽出するparse_ipv4へ遷移するかをC言語のswitch文に似た構文（select文）で分岐している．ここで抽出するヘッダ情報は図4上部で定義されるstruct headerの構造体

```
parser MyParser(packet_in packet,
                out headers hdr,
                inout metadata meta,
                inout standard_metadata_t standard_metadata) {
    state start {                                    ← 初期状態
        transition parse_ethernet;
    }                          ↙ 状態遷移
    state parse_ethernet {                           ← イーサネットヘッダの情報抽出
        packet.extract(hdr.ethernet);
        transition select(hdr.ethernet.etherType) {
            TYPE_IPV4: parse_ipv4;                   ↖ EtherTypeの値で状態遷移先選択
            default: accept;
        }                          ↙ 状態遷移
    }
    state parse_ipv4 {                               ← IPヘッダの情報抽出
        packet.extract(hdr.ipv4);
        transition accept;
    }
}
```

図6　Basic Forwordingのパーサー

変数hdrに保存され，以降のパイプラインでも利用できる．

マッチ・アクションを実行するcontrolブロックを図7に示す．マッチ・アクションは，tableステートメントでマッチ条件とアクションの対応付けを保存できるフローテーブルの型を定義する．ここでは，フローのマッチ条件（フローテーブルのカラム名key）の型と，マッチしたパケットへのアクション（フローテーブルのカラム名actions）の選択肢を定義しており，テーブルの型定義のみではテーブル内には何の値も挿入されていない．

IPフォワードを実装する図7では，宛先のIPアドレスに基づいてパケットのフォワード先を決定するために，宛先IPアドレスをマッチ条件として定義している．ここでlpmはlongest prefix matchの略で，一般的な経路制御に用いられる最長一致検索を行うことを示している．

tableステートメントのactionsに一覧される処理は，actionステートメントを用いて，パケット処理内容をプログラムコードで記述する．IPフォワードを行うアクションのipv4_forward（図7中「アクションipv4_forwardの定義」参照）は，スイッチを経由するたびにイーサネットフレームの送信元・先のMACアドレスを書き換える処理（IPフォワードを実施するスイッチの一般的な動作）が記述されている．パケットを廃棄するdropはV1Model固有のメタデータstandard_metadataに破棄フラグを追加している．（図7中「アクションdropの定義」参照．）この情報を基にトラヒックマ

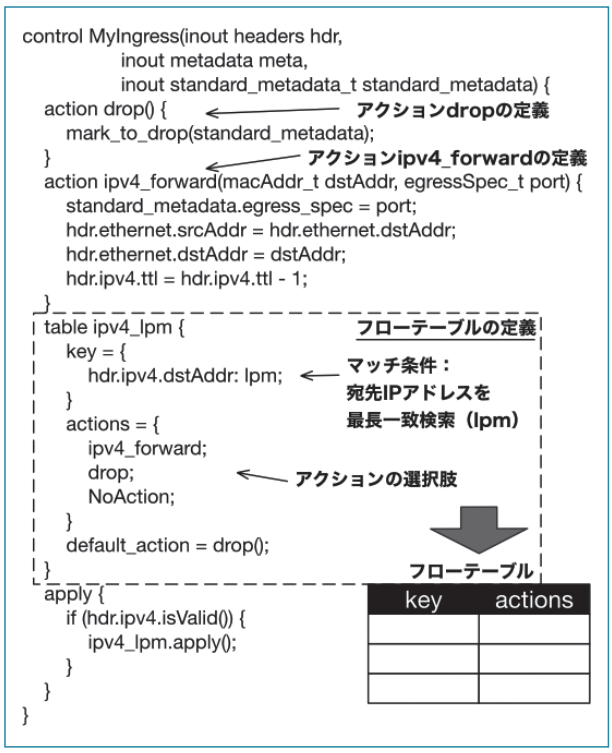

図 7　Basic Forwording のマッチ・アクション

ネージャがパケットを破棄する．NoAction は予約語で，その名のとおり，何も処理しないアクションである．デパーサーはここには掲載しないが，パケット処理の記述はなく，パケット送出のみ行う．

ここで定義したフローテーブルは，パケットが到着するたびに参照され，対象のパケットに対してテーブル内のデータ単位でマッチ条件を比較する．例えば，フローテーブルに「192.168.0.1, ipv4_forward」というデータが挿入されていた場合，宛先 IP アドレスが 192.168.0.1 のパケットに対して，ipv4_forward の処理が実行される．データが複数挿入されている場合には，パケットごとにテーブル内のデータと順番に比較し，マッチした場合には指定されたアクションを実行する．

上記のパーサーとマッチ・アクションで生成するフローテーブルの機能を使うことで最低限の IP フォワードを実現できる．本来は転送するパケット以外の処理，例えば ARP の処理なども必要であるが，ここでは扱わない．

図 7 のコードでは，フローテーブルの型を定義するのみで，具体的な値，つまりマッチ条件とアクション指定の情報は挿入されていない．よって，具体的な宛先 IP アドレスに対して，次ホップの MAC アドレスとパケット送出する物理ポート番号を指定した，マッチ条件とアクションのペア（フローテーブルに登録するデータ）自体はコントロールプレーンが登録する必要がある．BMv2 では RuntimeCLI と呼ぶコマンドラインツールがあり，手動でマッチ条件とアクションのペアを挿入し，データプレーンで実装した機能をテストできる．しかし，実ネットワークで利用する場合には，P4 で記述したデータプレーンと通信して，P4 で定義したフローテーブルにルールの追加・編集・削除するコントロールプレーンの実装のための P4Runtime が開発されている [(10)]．本稿はプログラマブルデータプレーンに着目しているため，コントロールプレーンの技術である P4 Run-time の説明は省略する．

3　ネットワーク主導型 TCP フロー制御調和手法

本章では，2．で説明したプログラマブルデータプレーンを用いてネットワークが主導して複数の競合 TCP フロー間の調和を図る制御手法について説明する．

3.1　ネットワーク主導のTCPフロー制御調和

TCP のふくそう制御アルゴリズムは基本的には送信端末が，自身が送信したパケットに対する確認応答（ACK パケット）の受信に応じて送信量（ACK パケット受信前に連続送信可能なデータ量を表すウィンドウサイズ）を増減することで制御する．送信端末は，転送経路上の帯域が十分に利用可能な場合には ACK パケットを遅延やロスなく受信できるため，ウィンドウサイズを適応的に増やす．一方で，帯域不足によってデータ／ACK パケットの損失・遅延が起きた場合，送信端末はウィンドウサイズを減らす．結果として，ボトルネックリンクで競合する TCP フローは帯域を公平に分けるように送信量を制御する（図 8）．これに対してアプリに着目すると，アプリの用途に応じて必要なスループットが異なる場合がある．例えばビデオ視聴が再生速度に必要なデータ量を転送できれば途切れないように，全データを一度にまとめて転送する必要がない等が挙げられる．このようにアプリ要求に差があっても，TCP は公平にボトルネック帯域を利用するため，差異のあるアプリ要求と TCP 制御の結果得られる公平なスループットに不一致が生じ，アプリが真に必要とするデータ転送に必要なネットワーク帯域を効果的に活用できない．

本研究では上記のような競合するTCPフロー間のアプリ要求とネットワーク帯域間の不一致を調和させるために，アプリ要求に対して過剰なスループットを提供しているTCPフローのデータ送信量の調整を，プログラマブルデータプレーンを用いて実現する．TCPフローのスループットを制御する方法は既存技術で様々提案されている．端末のTCP制御を拡張した場合には，前述のとおり，ネットワーク内部でフロー競合の存在や競合状況に応じて，フローそれぞれの要求を満足する調和が本質的に困難である．また，パケットをスイッチのキュー（記憶領域）に一時的に保存する手法や，キュー内のパケット順序制御や破棄する手法では，遅延や損失が増加する．更に，パケットを破棄した場合，送信端末のTCP制御によって再送されるパケットによる帯域の消費や，ネットワーク内でのパケット順序の入れ替わりによってアプリ品質が低下（例えば，ビデオ視聴では帯域を消費して届けたにもかかわらず映像が乱れる．）してしまう．

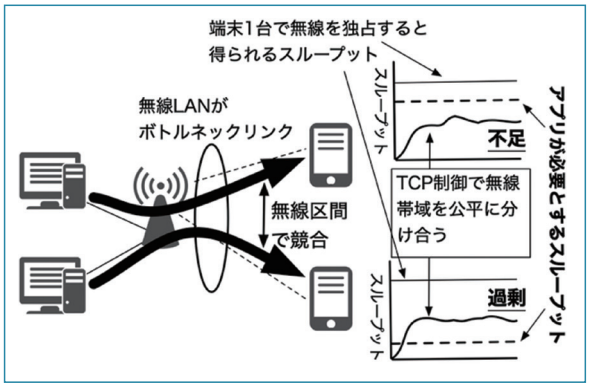

図８　TCP制御とアプリ要求の不一致

そこで本研究では，データプレーンを用いてネットワーク主導で従来のトラヒック制御を変更することなく，TCP制御を誘導することでアプリ要求に合わせて端末のパケット送出を調整し，遅延や損失を抑えながらアプリが真に必要なパケットを効果的に宛先に届ける調和制御を実現する．

この目的を実現するために，ボトルネック区間に実装したプログラマブルデータプレーンを用いて，パケット送信量を抑制したいTCPフローのACKパケットを編集して返送する方法を用いる．TCPでは送信端末が送信量（ウィンドウサイズ）をACKパケットの受信状況の観測結果に応じて増減する値（cwnd）とACKパケットに含まれるadvertised window（awnd）の小さい方の値に設定する．つまり，ボトルネックリンク上の競合フローの中でスループットを抑えたいフロー（アプリ要求＜転送スループット）を対象に，ACKパケット中のawnd値をプログラマブルデータプレーンを活用して動的に変更することで，ネットワーク主導で送信量を制御する．一方で，アプリ要求に対してスループットが不足しているフローは，通常のTCP制御を継続することでスループットの向上を図る．このように，本研究ではネットワークがアプリ要求を考慮して複数フローの制御を調和させる．

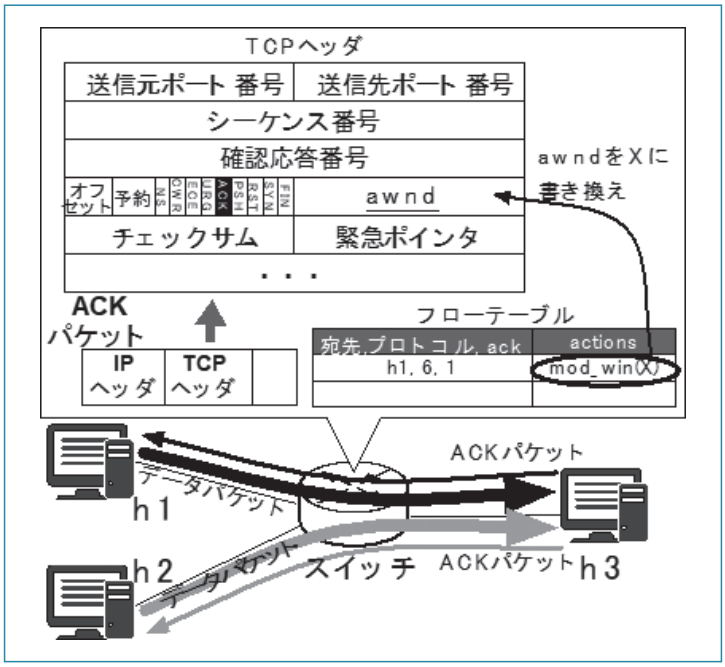

図９　awnd書換えの概念図

3.2　P4を用いたTCPフロー制御調和の実装詳細

3.1で説明したawnd値を書き換えるパケット処理機能を，2.3で説明したBasic Forwardingに追加実装する．本稿では，試作したデータプレーンのTCPフロー制御の調和手法の機能検証のために，図9に示す二つの通信フローのみが転送され，ネットワーク環境に変動がないシンプルな有線ネットワークの構成を想定する．図9では，3台の端末が有線でスイッチに接続し，2台（h1とh2）が残りの1台（h3）へ向けてTCPフローを送信する．この場合，スイッチとh3の区間がボトルネックとなり，通常のTCPふくそう制御によって公平に帯域を分け合うと，h1がアプリ要求に対して過剰，h2がアプリ要求に対して不足する状況を想定する．つまり，h1とh3間のTCPフローのスループットを抑制する．

この仕組みを実現するために，図6と図7のプログラムコードに図10と図11を追加した．本研究では，ACKパケットにマッチさせてawndを書き換えるため，図10のTCPヘッダをマッチ条件に用いるために定義した．また，awndを書き換える際にTCPチェックサムの再計算が必要となる．そこで，チェックサム計算に用いる擬似ヘッダ用のTCPセグメントサイズもメタデータとして定義した．図11は，パーサーとマッチ・アクションの追加部分のみ抜粋した．図6のパーサーはIPヘッダの抽出のみであったが，TCPヘッダを抽出するparse_tcpという状態を定義した．マッチ・アクションでは，対象フローのACKパケットにマッチする条件と，そのawndを変更するアクションであるmod_winを実行するフローテーブル（図9）の型を定義する．マッチ条件には，IPヘッダの宛先IPアドレスとプロトコル番号，TCPヘッダのACKフラグの三つを挿入できる構成で定義し，それらが完全一致した場合にアクションを実行する評価方法を指定した．今後，多数のフロー

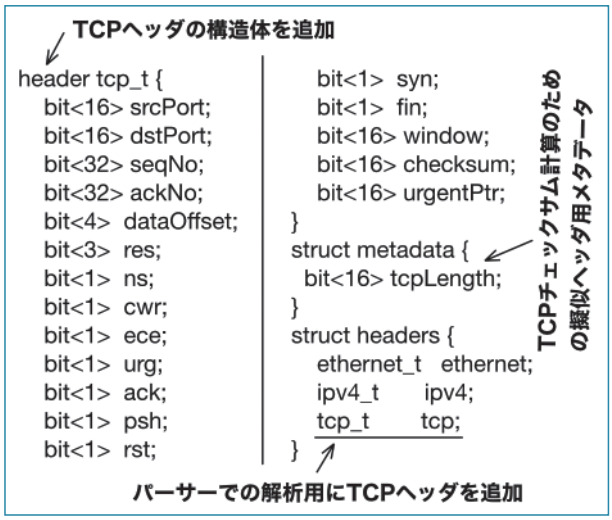

図10　定義の追加部分

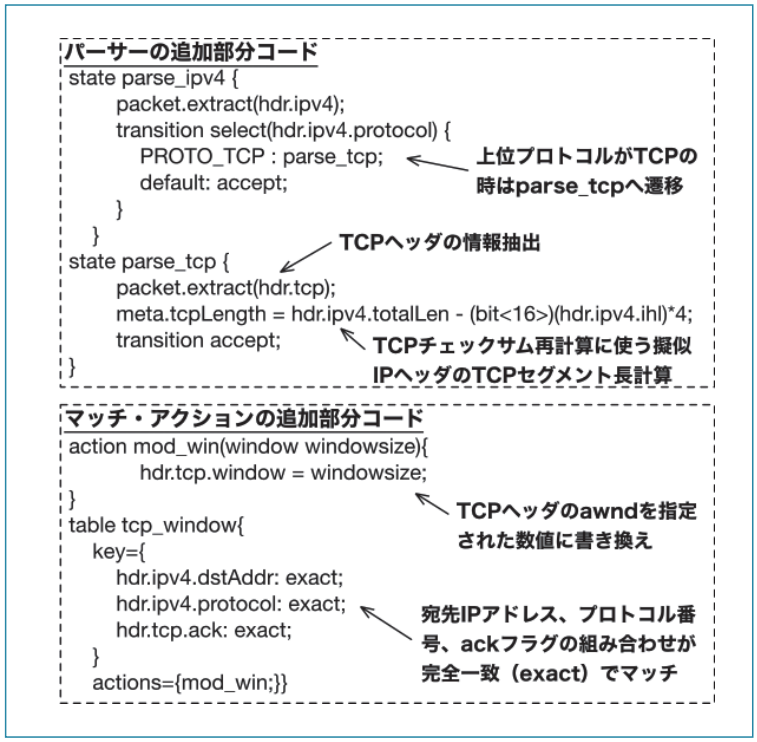

図11　パーサーとマッチ・アクションの追加部分

が流入する状況ではTCPポート番号も条件に含める等の工夫が必要である．アクションの内容を記載したmod_win（図11下段，action mod_win参照）はTCPヘッダ内のawndフィールドを指定された値に置き換える．最後に，デパーサーで変更後のawndを含むTCPヘッダのチェックサムを再計算する．デパーサーの具体的な記述は省略する．

4 基礎実験

本章では，3．で実装したデータプレーンを用いて，実際にアプリ要求に応じたスループット調和が可能であることを実験を通じて確認する．

4.1 実験環境

実験環境は，日本P4ユーザ会が配布する「P4チュートリアル用仮想マシン」を用い，その仮想マシン上に準備されたmininetとBMv2を組み合わせた環境で実験する．仮想マシンの中にあらかじめチュートリアルの実行環境が準備されており，本研究がベースとしたBasic Forwardingのディレクトリでトポロジーを図9の構成に書き換えた．また，BMv2のソフトウェア実装によって高速なパケット処理が困難なため，実験環境をスケールダウンしてTCPフローを競合させるために図9の各リンクの伝送帯域を10 Mbit/s，片方向遅延を5 msに設定した．

実験は，3.2の環境と同じく，h1とh2がh3へTCPフローを送信する．ただし，h1のアプリの要求スループットは2 Mbit/s，h2のアプリ要求は8 Mbit/sとして，競合区間で不均等なスループットを要求する状況を想定する．ここで3．の実装を用いて，宛先がh1のACKパケットのawndをフロー開始当初から747へ書き換える．この数値は，同一実験環境でpingを用いて事前に計測したRTTと上記のアプリ要求スループットから算出した．具体的には，要求スループット（bit/s）＝ウィンドウサイズ（Byte）×8/RTT（s）の関係式からウィンドウサイズを逆算した．実際のウィンドウサイズは，ACKパケットのawndフィールドの値×Window scale（本実験環境では512）で計算されるため，awndは747と算出される．

3.2の実装ではkeyとactionを指定したフローテーブルの型を定義したが，フローの具体的なマッチ条件の値と処理内容のペアは登録していない．本稿の機能検証ではコマンドラインツールのruntimeCLIでマッチ条件とアクションのペアをあらかじめ手動で登録した．具体的には，runtimeCLI実行後，待機状態になってから下記のコマンドを入力する．

RuntimeCmd: table_add MyIngress.tcp_window MyIngress.mod_win [h1のIPアドレス] 6 0b1 => 747

ここで「=>」の左側はルールを追加するテーブル名と指定するアクション名，フローテーブルのkeyを順に指定しており，右側はactionのプロシージャに渡す引数を指定する．今回は，3.2で作成したフローテーブルtcp_windowでアクションとしてmod_winを設定し，マッチ条件として制御対象フローのデータ送信端末（ACKの宛先）のIPアドレス（h1）とプロトコル番号6（TCP），ackフラグが0b1（バイナリで1）の組合せを指定した．また，図11のアクションmod_winに対して，その引数となるwindowsizeとして「=>」の右側に記載した数値747を指定する．これにより，keyにマッチしたACKパケットのawndフィールドは747に置換される．

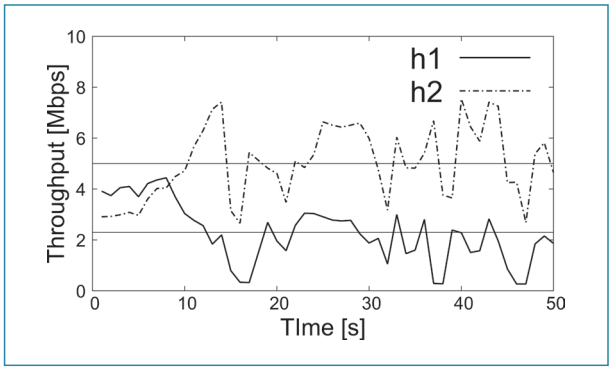

図12　スループットの計測結果

4.2 実験結果

図12にTCPフロー開始から終了までの各フローのスループットの時系列グラフを示す．制御を全く行わないTCPフロー同士であればスループットはほぼ同等になるのに対し，提案手法を導入することで端末の制御を変更することなくh1のスループットのみを想定する2 Mbit/s付近に抑制できた．一方，h3は端末によるTCP制御に任せることで約5 Mbit/sを獲得している．当初想定した8 Mbit/sまで増えていないが，これはBMv2

のソフトウェア処理のボトルネックに起因することが判明しており，ボトルネック区間の上限スループットを獲得できたことを確認している．以上の結果より，本研究で実装したデータプレーンがアプリ要求を意識した競合フローの調和制御を実現可能な機能を備えていることを示した．

5 まとめ

本稿では，プログラマブルデータプレーンのアーキテクチャと，アーキテクチャに合わせてパケット処理をP4言語で記述する方法を解説した．また，P4言語を用いてデータプレーンに対してTCPヘッダ内のawndを書き換える，という従来のSDNには備わっていなかった機能を実装することで，ネットワーク内部でフローの競合状況に合わせて，競合する複数TCPフロー間のふくそう制御を調和させるための基礎技術が実現できることを示した．今後は，図8で示したような無線変動が生じる無線環境も考慮した調和制御や，フロー単位で適切なawndを決定する方法，及びフロー開始・終了に合わせて動的にマッチ条件とアクションのペアを管理するコントロールプレーン制御手法等を検討する必要がある．

謝辞　本研究は総務省SCOPE（JP235007003）とJSPS科研費JP21H03430，国立研究開発法人情報通信研究機構（NICT）の委託研究No.05501の助成を受けた．

■文献

(1) N. Feamster, J. Rexford, and E. Zegura, “The road to SDN: an intellectual history of programmable networks,” ACM SIGCOMM Comput. Commun. Rev., vol. 44, no. 2, pp. 87-98, April 2014.
(2) N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, “OpenFlow: enabling innovation in campus networks,” ACM SIGCOMM Comput. Commun. Rev., vol. 38, no. 2, pp. 69-74, April 2008.
(3) P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, and D. Walker, “P4: programming protocol-independent packet processors,” ACM SIGCOMM Comput. Commun. Rev., vol. 44, no. 3, pp. 87-95, July 2014.
(4) J. Gomez, E. F. Kfoury, J. Crichigno, and G. Srivastava, “A survey on TCP enhancements using P4-programmable devices,” Comput. Networks, vol. 212, July 2022.
(5) BEHAVIORAL MODEL (bmv2). https://github.com/p4lang/behavioral-model
(6) P4_16 Language Specification version 1.2.4 (2023-05-15). https://staging.p4.org/p4-spec/docs/P4-16-v1.2.4.html
(7) GitHub - p4c/p4include/v1model.p4 at main. https://github.com/p4lang/p4c/blob/main/p4include/v1model.p4
(8) The BMv2 Simple Switch target. https://github.com/p4lang/behavioral-model/blob/main/docs/simple_switch.md
(9) p4lang/tutorials: P4 language tutorials. https://github.com/p4lang/tutorials
(10) P4Runtime Specification version 1.3.0 (2020-12-01). https://staging.p4.org/p4-spec/p4runtime/v1.3.0/P4Runtime-Spec.html

(2023年9月30日受付，12月15日再受付)

妙中雄三（正員）

奈良先端大准教授．2010奈良先端大で博士（工学）取得．東大情報基盤センター助教を経て，2018-04から現職．サイバーセキュリティやネットワークを専門とする．

塚本和也（正員）

九工大教授．2006九工大で博士（情報工学）取得．その後，JSPS特別研究員，米国UCI客員研究員を経て，2007から九工大に勤務．2022-01から現職．各種ネットワークにおける通信制御を専門とする．