Vol.2019-IOT-44 No.22
2019/3/7

# 動的な機能変更を可能にするエッジコンピューティング基盤の実装と評価

小林 海[1]　木村 隼人[1]　大東 俊博[1]　渡邉 英伸[2]　相原 玲二[2]
近堂 徹[2]

**概要**：筆者らは、多様化するデータトラフィックに対する処理をモジュール化して広域ネットワーク上に能動的に配置することで、デバイス・エッジノード・クラウドの 3 層で連動した処理が可能な、モジュラー型エッジコンピューティング基盤の開発を進めている。本稿では、複数のセンサー情報をトリガーにプラットフォームが動的な機能変更を実現する手法について示し、カメラデバイスを用いた実アプリケーションによる実装と評価について述べる。

# Implementation and Evaluation on the Edge Computing Platform with Dynamic Modular Configuration

Kai Kobayashi[1]　Hayato Kimura[1]　Toshihiro Ohigashi[1]　Hidenobu Watanabe[2]　Reiji Aibara[2]
Tohru Kondo[2]

***Abstract:*** We have been developing an edge computing platform that can coordinate with each computing resources of device, edge node and cloud for diversifying data can be modularized and actively deployed in cooperation within the wide area network. This platform brings transparent processing environment for IoT devices according to a data characteristic or use purpose. This paper describes dynamic control method for the platform based on multiple sensor information, and show an implementation and evaluation by actual application using camera devices.

## 1. はじめに

インターネットに接続される機器が急速に拡大し、IoT デバイスの普及も確実に広がってきている。2020 年には 400 億台のデバイスがインターネットに接続され、1 ヶ月に流通するデータ量も 250 エクサバイトを超えるとの予測もある [1]。

IoT における多種多様なデータの流通への期待に伴い、ネットワークに対してもソフトウェアによる柔軟かつ効率的な制御が望まれている。そのひとつしてエッジコンピューティングがある。IoT プラットフォームがより広い領域で活用されるためには、従来クラウド内に閉じているデータ蓄積や解析をエッジノードまで広げて自立分散かつ相互連携により処理することで、通信トラフィックの削減、リアルタイム性・レスポンスの向上、処理負荷の分散を実現していく必要がある。

筆者らはこれまで、データ特性に応じて組み替え可能なモジュラー型エッジコンピューティング基盤の開発を行ってきた [2]。これは、エッジノード内にコンテナ仮想化によるアプリケーション実行環境を提供し、多様化するデータトラフィックに対する処理をモジュールインスタンス化し、本実行環境に動的に配置、コントローラによる処理フローを定義することで、エンドデバイス・エッジノード・クラウドの 3 層で相互連動した処理を実現するものである。これにより必要な機能を必要な箇所に適切かつ迅速に展開できることが期待できる。

しかしながらこれまでの研究開発では定義する処理フ

[1] 東海大学情報通信学部, School of Information and Telecommunication Engineering, Tokai University
[2] 広島大学情報メディア教育研究センター, Information Media Center, Hiroshima University

ローは固定的なものにとどまっており、データ特性や状態の変化に対するフローの動的な制御には十分に対応することができていなかった。そこで本研究では、ノードの資源情報やデータ特性をコントローラが把握し、処理フローの構成を動的更新できるようにする。本稿では、複数のセンサー情報をトリガーにプラットフォームが動的な機能変更を実現する手法について示し、カメラデバイスおよびセンサーを用いた実アプリケーションによる実装と評価について述べる。

本稿の構成は以下の通りである。まず、2 章で要素技術と関連研究について述べ、3 章で提案プラットフォームと広域分散環境への展開について述べる。4 章で本プラットフォームを利用した評価実験について示し、最後に 5 章でまとめを述べる。

## 2. 要素技術と関連研究

IoT プラットフォームでは、流通するデータ量の増減や処理能力の要求に対して柔軟性と迅速性を担保することがあり、またプラットフォーム上にサービスを容易に展開可能であることが望まれる。このことから、プラットフォームのバックエンドとして仮想化技術を利用することが一般的となっている。特に、コンテナ型仮想化を用いたプラットフォームが存在する [3][4]。

Kubernetes[*1]は Docker[*2]コンテナによるアプリケーションデプロイ、自動スケーリング、状態監視等の制御が可能なコンテナオーケストレーションツールである。アプリケーションのポータビリティを向上させることができるため、Google やマイクロソフト、AWS といった大手クラウドベンダでもマネージドサービスが提供されるようになってきている。プラットフォームとしての汎用性が高いが、副作用として複雑性が課題となる。また、導入の多くがコンピューティング資源が集中したクラウド環境を想定しており、資源が広範囲に分散するエッジ環境への導入は、Kuberenetes コミュニティ（IoT Edge Working Group）で課題の議論が行われているところである。

Bruno らが提案する FITOR[5] は、IoT アプリケーション用のオーケストレーションシステムのひとつである。Docker によるコンテナ仮想化環境の上に Calvin とよばれるオープンソースソフトウェアによる制御層を設け、エッジコンピューティングに対応するアプリケーションのプロビジョニングを最適化する手法が提案されている。FogFlow[6] は NEC が提案する、FITOR と同様にデータ処理フローを定義可能なプラットフォームである。クラウドとエッジ上にてシームレスでスケーラブルに管理し、IoT サービスのオーケストレーションを実現する分散型実行フレームワークを具備する。IoT デバイスや上流にある処理フローからのデータを入力に、Docker コンテナでインスタンス化されたタスク処理により新たなデータが生成される。

以上の通り、IoT 向けプラットフォームもこれまで数多く提案されている。いずれも、既存のオーケストレーションツールを活用し、その上位に抽象化レイヤを定義することでデータフローの考え方を導入した、エッジコンピューティング用途に特化した処理プラットフォームを実現しているものである。これにより、デバイスやデータ発生源の分散性やコンピューティング資源に対する要求のばらつき等を考慮するものである。一方で、上述したプラットフォームの多くがデバイスやデータ発生源に依存したノード配置と固定的な制御にとどまっており、デバイスの増減や発生源の移動に伴うトラフィックの変動を考慮したエッジノードの資源管理についても課題が残る。

## 3. 提案システム

### 3.1 プラットフォームの概要

図 1 に提案プラットフォームの概要を示す。本プラットフォームはコンテナ仮想化技術の管理フレームワークと連携するプラットフォームコントローラ（以下、コントローラ）と、エッジノード上でコンテナとして動作するプロセスモジュール（以下、モジュール）の 2 つの要素から構成される。エッジコンピューティング基盤に求められる要件は、必要な機能を必要な場所に適切かつ迅速に展開できることである。本プラットフォーム制御機構は、コンテナによる広域分散環境を管理するオーケストレーションツールの上位層に位置し、よりデータ特性や処理特性を考慮してコンテナをより適切に制御することで、エッジコンピューティングを実現する。なお、オーケストレーションツールとして前章でも示した Kubernetes を利用し、関連するコンテナのグルーピングや IP アドレスの管理、複数コンテナを利用した負荷分散、コンテナ監視等の機能を効果的に利用する。

開発するコントローラは RESTful API を提供し、プラットフォーム利用者（以下、ユーザ）のエッジコンピューティング環境の展開を支援する。コントローラはユーザからのリクエストを解釈して、Kubernetes Master（マスターノード）と連携して各ノードの制御を行う。また、マスターノードは Kubernetes のフレームワークを利用することで分散型設定共有サービス etcd や Docker リポジトリ機能を有し、各ノードとコントローラでスムーズな連携を可能としている。ユーザが指定するデータ処理フロー（以下、フロー）をコントローラがトポロジーとして展開し、任意の場所にプロセスモジュールを配置する。これにより、デバイスからの情報を自動的に収集、さらにデータ発生源に応じてモジュールの配置場所を動的に変更することも可能になる。これらはユーザが YAML ファイル形式の記述言語

[*1] https://kubernetes.io/ (2019-1-22 参照)
[*2] https://www.docker.com/ (2019-1-22 参照)

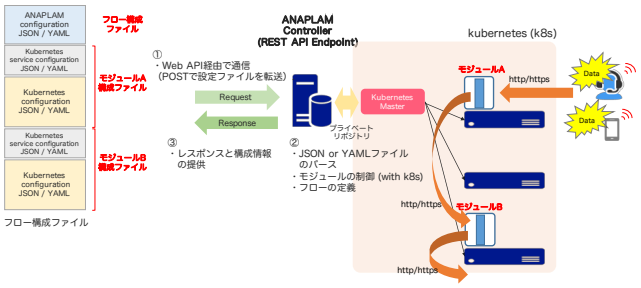


**図 1** プラットフォーム概要

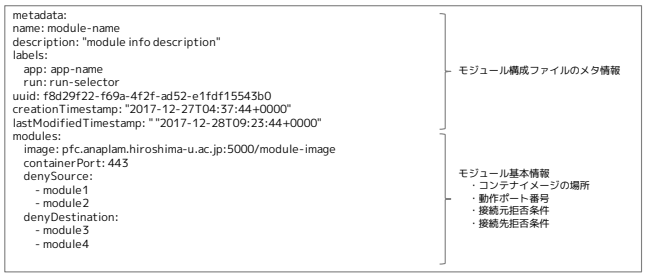


**図 2** モジュール構成ファイル

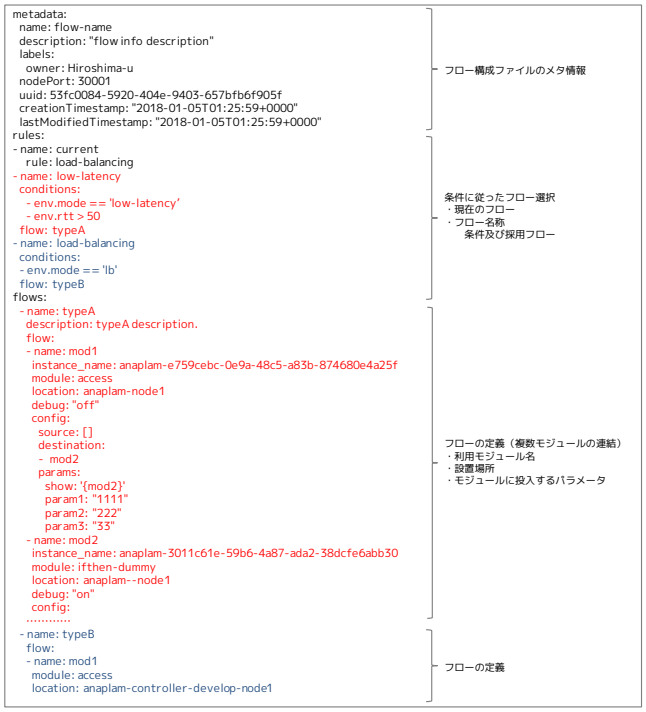


**図 3** フロー構成ファイル

で定義する。YAML 形式を採用することで、フローを処理するプロセスモジュールを構造化されたデータとして表現することができ、ユーザが取り扱いやすい形で管理できる。

### 3.2 プラットフォームコントローラ

表 1 にコントローラが具備する API の一覧を示す。API は、ユーザからのリクエストに応じてノードの管理、モジュールの制御、フローの制御を行う Northbound API と、展開したモジュール（コンテナ）に対する制御を行う Southbound API を定義した。開発するプラットフォームやモジュールはこれらの API に準拠した設計にすることで、本研究の目的であるモジュール配置や組み替え等の柔軟な制御が可能となる。通信には GET、POST、DELETE 等の HTTP 標準のメソッドを利用することで、シンプルかつ一貫性のあるリクエストの標準化が実現している。なお、コントローラではあらかじめセッションキーを発行し、API リクエストにこのキーを付与しなければ制御することはできない。

図 2 にコントローラに投入するモジュール構成ファイル、図 3 にフロー構成ファイルのサンプルを示す。モジュール構成ファイルでは、プロセスモジュールをプラットフォームに登録するためのものであり、モジュールイメージの保存場所やモジュール処理の前後関係の制約条件を記述することができる。一方、フロー構成ファイルはモジュールの連結によりフローを定義するものである。

コントローラは Kubernetes の機能を利用して各ノードで動作するプロセスモジュール（Docker コンテナ）の稼働状況やノード自身のステータスを把握することができる。また、Kubenetes とあわせて監視ソフトウェアの zabbix を利用してより詳細な資源利用状況も収集している。これにより、ノードの資源情報やデータ特性をコントローラが把握し、状況に応じてフロー構成を動的に更新することを可能にしている。処理の流れについては、3.4 節で示す。

### 3.3 プロセスモジュール

プロセスモジュールは Docker のコンテナイメージで提供する機能ブロックである。表 2 に現在実装しているプロセスモジュールの一覧を示す。

各モジュールは表 1 中の Southbound API で定めた共通 API を有し、コントローラからの制御が可能である。各モジュール特有の動作は図 3 で示したフロー構成ファイル内の params キー配下で指定された内容に従う。一例として if-then モジュールの動作を示す。本モジュールでは、アップロードされた JSON フォーマットのデータに対して任意で設定した条件式に従った処理を実現するものである。条件式としては一般的な比較演算子・論理演算子をサポートし、これらを YAML で定義できる構造とした。これらの条件式はモジュール起動中であっても Southbound API 経由でアップデートすることで動的に変更することができるようになっている。さらに、条件式に該当した処理に対して変数を組み込めるように設計し、動的な処理を実現している。このように処理の内容をコード化し、プラットフォーム内にモジュールとして展開することで任意の機能を任意の場所に展開することが容易に可能な設計となっている。

### 3.4 処理の流れと動的変更な機能変更

図 4 に本プラットフォームにおける処理の流れを示す。

**表 1** コントローラが具備する API リスト

ノードの管理

| メソッド | HTTP リクエスト | 引数 | 戻り値 (YAML) | 備考 |
|---|---|---|---|---|
| list | GET /nodes | none | ノード構成情報 | Kubernetes ノードの構成情報を取得 |

モジュール制御用

| | | | | |
|---|---|---|---|---|
| get | GET /modules/moduleId | モジュール ID | モジュール構成情報 | 指定した名前/ID のモジュール情報を取得 |
| define | POST /modules | モジュール構成情報 | モジュール構成情報 | モジュール定義の作成 |
| undef | DELETE /modules/moduleId | なし | 成功 / 失敗 | モジュール定義の削除 |
| list | GET /modules | なし | モジュール一覧 | 登録されたモジュール定義一覧 |
| update | POST /modules/moduleId | モジュール構成情報 | モジュール構成情報 | モジュール定義の更新 |

フロー制御用

| | | | | |
|---|---|---|---|---|
| get | GET /flows/flowId | フロー ID | フロー構成情報 | 指定した名前/ID のモジュール情報を取得 |
| create | POST /flows | フロー構成情報 | フロー構成情報 | フローの生成 |
| delete | DELETE /flows/flowId | なし | 成功 / 失敗 | フローの削除 |
| list | GET /flows | なし | フロー一覧情報 | 処理フローの一覧 |
| update | POST /flows/flowId | フロー構成情報 | フロー構成情報 | フロー情報の更新 |
| rule | GET /flows/flowId/rules/ | なし | フロー構成情報 | フローのルールのみを取得 |
| modify | POST /flows/flowId/rules/ | 条件式 | フロー構成情報 | ルールに従ったフロー情報の更新 |

図 3 で示したフロー構成ファイルに従い、任意のノード上に希望するモジュールを展開し、エンドデバイスからのデータ入力に対してフローに従った処理を行う。エンドデバイスからのデータ処理対象として HTTP/HTTPS の POST による JSON 形式のデータアップロードを想定している。

次に動的な機能変更を実現するフロー更新処理について述べる。フロー更新処理の際には、フローの root から leaf（処理するモジュールの最後尾から先頭）に対して再構築を行いながらデータパスの再定義を行い、モジュールの再配置とデータパスの再定義が完了するとフロー更新が完了する動きとなる。フロー更新が完了すると、旧フローを削除しノードの資源開放を行う。これにより、動的にフロー切り替えが可能となる。なお、フローが切り替わってもエンドデバイスが最初にデータをアップロードするエンドポイント自体は変更されることはなく、本プラットフォーム内で更新されたフローの先頭に位置するモジュールインスタンスへ自動的に転送されるため、エンドデバイスの送信先等の設定変更は必要ない。

## 4. 基礎評価実験

本章では、コンテナモジュールを利用したプラットフォームの基礎評価として、複数モジュールを利用した動的なフロー変更に関する評価について述べる。

### 4.1 実験構成

実験構成図を図 5 に示す。エッジサーバには、表 6 に示すスペックのサーバを利用し、本プラットフォームコントローラと連携するコンテナ仮想化環境が動いている。なお、

**表 2** プロセスモジュール一覧

| 用途 | モジュール名 | 機能概要 |
|---|---|---|
| データ加工 | 閾値判定 | 特定ルール (if-then) に基づき、データの閾値判定として処理 (フィルタリングとデータ転送) |
| | データ圧縮 | 指定した方式でデータ圧縮転送 |
| | データ永続化 | 外部ストレージに対してデータを保存 |
| セキュリティ | ID ベース暗号 | ID ベース暗号によるデータ暗号化と復号、鍵発行・配布機能 |
| | プロキシ暗号 | プロキシ暗号による暗号化と復号 |
| フィードバック | 外部通知 | Slack API を使った外部通知 |
| 画像処理 | 画像品質変換 | ffmpeg を利用した画像品質変換 |

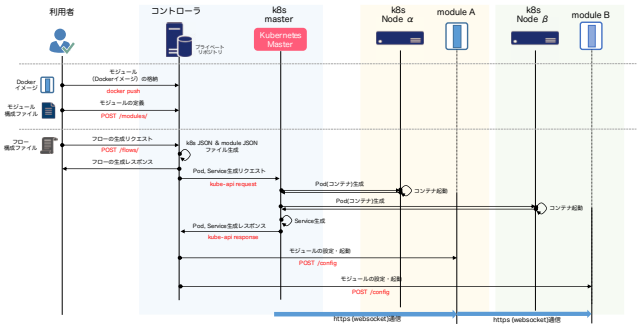


**図 4** 処理の流れ（フロー設定時）

エンドデバイスと制御用サーバには表 3 に示す Raspberry Pi を使用し、クラウドリソースとしては図 5 に示したさくらのクラウド*3の石狩リージョンを利用している。

*3 https://cloud.sakura.ad.jp/

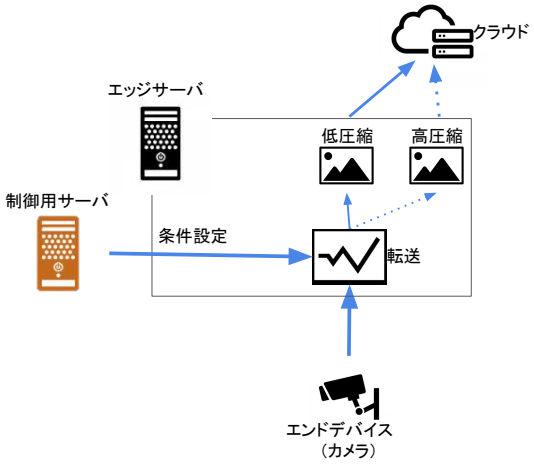


**図 5** 実験構成図

**表 3** 実験環境 (エンドデバイス, 制御用サーバ)

| ホスト情報 | |
|---|---|
| CPU | ARMv8 Cortex-A53 |
| OS | Linux version 4.14.34-v7+ |
| RAM | 1GB |
| Python | 3.5.3 |
| cURL | 7.52.1 |

**表 4** 実験環境 (エッジサーバ)

| ホスト情報 | |
|---|---|
| CPU | i7-6700K CPU @ 4.00GHz |
| OS | Linux version 3.10.0-957 |
| RAM | 32GB |
| Docker | Docker version 18.09.1 |
| 各モジュール情報 | |
| OS | Linux version 3.10.0-957 |
| 割り当て CPU | 2CPU |
| 割り当て RAM | 512MB |
| Python | 3.5.2 |
| Library | Flask-Classy 0.6.10 |
| cURL | 7.47.0 |
| ffmpeg | 2.8.15 |

**表 5** 実験環境 (クラウド)

| ホスト情報 | |
|---|---|
| CPU | Xeon CPU E5-2640 @ 2.50GHz |
| OS | Linux version 3.10.0-862.14.4 |
| RAM | 8GB |
| Docker | Docker version 18.06.1 |
| 各モジュール情報 | |
| OS | Linux version 3.10.0-957 |
| 割り当て CPU | 2CPU |
| 割り当て RAM | 512MB |
| Python | 3.5.2 |
| Library | Flask-Classy 0.6.10 |
| cURL | 7.47.0 |

### 4.2 実験方法

本研究で実装したカメラデバイスを用いた実アプリケーションは、エンドデバイスとしてのカメラから得られた画像をエッジサーバで加工をしてクラウドへ転送・保存する動作を行う。その際、周辺デバイスの情報からコントローラ（制御用サーバ）が判定をし、クラウドに送信する前の圧縮や解像度を動的に変更可能にする。動的な変更は制御用サーバから送信された Config ファイルによって転送用モジュールがカメラ画像を低圧縮モジュールか高圧縮モジュールのどちらに転送するかを切り替えることで実現している。

本評価実験は、計測するためにエンドデバイスから取得するカメラ画像をあらかじめ用意して、転送モジュールに POST することで行う。次の送信は、前の POST に対するレスポンスが返ってきてから 1 秒後に行うものとする。転送モジュールに POST する画像データは汎用性のために Base64 でエンコードしたものを用いる。実際に用意した画像を図 6 に示す。また、制御用サーバはトリガーとなるセンサー情報を利用せず、指定した機能変更の回数や間隔で転送モジュールに Config ファイルを POST する Python のプログラムを使用した。各圧縮モジュールは、POST された Base64 を画像へデコードし、ffmpeg を利用し指定したオプションで画像を加工した後にクラウドへ画像をアップロードを行う。評価実験は、(1) 制御用サーバからの動作の変更要求（config の送信）によってフローが切り替わるまでの時間の計測、(2) フローの切り替えによるカメラ画像の加工時間やアップロード時間の変化に関する計測の 2 種類を行う。

フローが切り替わるまでの時間の計測では、制御用サーバの POST プログラムを用いて 10 秒間隔で転送先を交互に低圧縮モジュールと高圧縮モジュールに切り替える動作を計 30 回行い、その平均を算出する。カメラ画像の加工時間やアップロード時間の変化に関する計測では、圧縮モジュールを固定してエンドデバイスから計 30 回ずつデータを POST し、その処理や通信に掛かった時間の平均を算出する。各圧縮モジュール内で実行される ffmpeg のコマンドを表 6 に示す。低圧縮の場合は圧縮率のオプションである-q を最小の 1 に指定し、高圧縮の場合は-q を最大の 31 に指定する他に-s オプションにより画像サイズを 6000x4000 から 1800x1200 に縮小している。

### 4.3 実験結果と考察

実験結果を表 7 と表 8 に示す。表 7 より、200msec 未満でフローを切り替えることがわかった。表 8 より、50KB と 5MB の画像をクラウドへ POST する際の通信時間は高圧縮のほうが早かった。これは HTTP/HTTPS 通信のための Web サーバの応答時間が影響していると考えており、更に大きなデータを少ない回数通信する場合に通信環境に

**図 6** 評価実験用画像

**表 6** ffmpeg の実行文

| 圧縮率 | 実行文 |
|---|---|
| 低 | ffmpeg -y -i /var/www/module/image.jpg -loglevel quiet -qmin 1 -q 1 /var/www/module/imageLow.jpg |
| 高 | ffmpeg -y -i /var/www/module/image.jpg -loglevel quiet -q 31 -s 1800x1200 /var/www/module/imageHigh.jpg |

**表 7** フロー切り替えの実験結果

| 処理時間 [ms] |
|---|
| 188.2 |

**表 8** 圧縮情報, 通信時間測定の実験結果

| 圧縮率 | 圧縮前 (KB) | 圧縮後 (KB) | 圧縮時間 (ms) | 通信時間 (ms) |
|---|---|---|---|---|
| 低 | 12538.6 | 5266.80 | 678.6 | 362.2 |
| 高 | 12538.6 | 55.3 | 480.6 | 330.4 |

よる差異が生じると考えられる。圧縮時間も高圧縮時の方が早かった。圧縮率を高くすることで処理時間が長くなると予想していたが、本実験ではデータサイズを小さくするために圧縮率の変更以外に解像度も変更していたため、画像サイズを縮小してから圧縮処理をした関係で処理時間が小さくなったと考えられる。

表 8 の圧縮後のファイルサイズより、必要に応じて低品質（高圧縮率）に切り替えることで通信トラフィックの削減が可能である。それにより、ある通信帯域で同時に複数のモジュールが通信を行う際により多くのモジュールが展開できる。さらに、クラウドで保存するデータサイズの節約が可能であるという利点もある。

## 5. まとめ

本稿では、ノード内にコンテナ仮想化によるアプリケーション実行環境を提供し、データトラフィックに対する処理をモジュールインスタンス化し、本実行環境に動的に配置可能なプラットフォームについて示した。特徴として、プラットフォーム制御を行うコントローラに対して処理フローを定義することで、エンドデバイス・エッジノード・クラウドの 3 層で相互連動した処理を実現することができる。

基礎評価実験では、カメラデバイスを用いた実アプリケーションの評価を行った。コントローラからの命令で動的に機能変更をする際の切り替え時間の計測および切り替えによるアプリケーションの影響について評価した。その結果、提案方式による機能の切り替えは本アプリケーションに関しては許容できる時間で終了することを確認できた。プラットフォームを実現することができた。

本稿では、基礎的な評価として同一エッジサーバ上で複数モジュールを切り替えるシンプルな構成で実験を行なっている。また、フロー切替のトリガーもコントローラを経由しない形で行なっている。今後の課題としては、コントローラ経由で複数のエッジサーバ間でフローの切替を行い、フロー切替に要する時間や通信量削減、レイテンシの改善の観点からの評価を行う予定である。

**謝辞** 本研究は総務省 SCOPE（受付番号 162108102）の委託、および日本学術振興会科学研究費助成金 18K11266, 16H02808 の助成を受けて実施した。

## 参考文献

[1] 総務省, 平成 30 年版情報通信白書, online available at http://www.soumu.go.jp/johotsusintokei/whitepaper/ja/h30/pdf/index.html

[2] Tohru Kondo, Hidenobu Watanabe, Toshihiro Ohigashi: ”Development of the Edge Computing Platform based on Functional Modulation Architecture”, Proc. the 2017 IEEE Annual International Computers, Software and Applications Conference, pp. 284-285, 2017.

[3] Claus Pahl, Brian Lee, ”Containers and Clusters for Edge Cloud Architectures – A Technology Review”, Proc the 3rd International Conference on Future Internet of Things and Cloud, pp.379-386 , 2015.

[4] Corentin Dupont, Raffaele Giaffreda, Luca Capra, ”Edge computing in IoT context: Horizontal and vertical Linux container migration”, 2017 Global Internet of Things Summit (GIoTS), 2017.

[5] Bruno Donassolo, Ilhem Fajjari, Arnaud Legrand, Panayotis Mertikopoulos , ”Fog Based Framework for IoT Service Provisioning”, Proc. the IEEE Consumer Communications & Networking Conference, 2018.

[6] Bin Cheng, Gurkan Solmaz, Flavio Cirillo, Erno Kovacs, Kazuyuki Terasawa, Atsushi Kitazawa, ”FogFlow: Easy Programming of IoT Services Over Cloud and Edges for Smart Cities”, IEEE Internet of Things Journal, Vol.5, Issue:2, pp.696-707, 2018.