Vol.2017-MBL-85 No.26
Vol.2017-ITS-71 No.26
2017/11/16

自律分散型タスクオフロードを用いたエッジコンピューティング

志野 嘉紀[1]　笹井 一人[2]　北形 元[2]　木下 哲男[2]

概要：Internet of Things (IoT) の発展に伴う低遅延制御の要求や通信トラヒック増加の対策としてエッジコンピューティングが注目されている．これは，従来はクラウドサーバ上で行っていた処理をデータの発生源であるデバイス付近のノードに取り付けた計算資源に分散することで，低遅延制御や通信トラヒック削減を可能にする技術である．分散先のノードは移動体基地局やゲートウェイ，センサデバイスなど様々な例が考えられる．本研究では，アプリケーションが自律的に最適な実行ノードを決定し処理をオフロードする手法を提案する．これにより，稼働アプリケーションの増加やノード間のトポロジ変化に強い分散処理を行うことが可能となる．本稿では，シミュレーションにより本手法におけるエッジ環境変化への対応能力を検証した．

キーワード：エッジコンピューティング クラウドコンピューティング タスクオフロード 自律分散システム

Decentralized Autonomous Task Offloading for Edge Computing

Yoshiki Shino[1]　Kazuto Sasai[2]　Gen Kitagata[2]　Tetsuo Kinoshita[2]

Abstract: In recent years, since the number of devices connected to the internet has increased drastically, processing the requests and analyzing data from the devices have been heavy loads even in a cloud environment. Nonetheless, the applications on the devices requiring real-time control such as autonomous cars and other robots need the quick response from the processors or the remote processing servers. Therefore, the concept of edge computing that places the computing processor or the servers, called edge servers, closer to the devices than the cloud servers has started attracting as the solution of above difficulties. The node that executes the application can offload the task to the cloud, the base station, the gateway, and so on. In this paper, we propose decentralized autonomous task offloading for the system that can adapt to changes in network topology and increasing applications. The results of the evaluation experiment show the ability of the proposed method to adapt to changes in edge environment.

Keywords: Edge Computing, Cloud Computing, Task Offloading, Autonomous decentralized system

1. はじめに

IoT(Internet of Things) における情報の解析・機器の制御におけるプラットフォームとして，ネットワークを通してコンピュータ資源を使用するクラウドコンピューティングが利用されている [1,2]．クラウドサーバは世界各地に分散配置され，豊富な計算資源やストレージを持ち，今後普及すると考えられる IoT においては，センサから受け取ったデータの処理やそれに基づいた情報の提示，アクチュエータの制御を受け持つ．クラウドサーバでの制御における今後の課題は，急速な IoT デバイス増加への適応と低遅延を要求するアプリケーションへの対応である．Cisco 社の予

[1] 東北大学大学院 情報科学研究科
Graduate School of Information Sciences, Tohoku University
[2] 東北大学 電気通信研究所
Research Institute of Electrical Communication, Tohoku University

測では，2019 年までに人，機械，モノから発生するデータは 500ZB にもなり，そのうち約 45%は IoT によって生み出されるデータであると考えられている [3,4]．2020 年までにインターネットに接続するデバイスは 500 億を超え，その結果，クラウドサーバの資源不足やデータ送信経路における通信帯域の圧迫が重大な問題となるとも言われている．また，遠隔地のクラウドサーバに接続する場合，通信遅延は往復数 100 ミリ秒になるため，自動運転における車の制御などの低遅延の要求に応える事が困難な場合がある．

これらの問題を解決する新しい IoT プラットフォームとして，エッジコンピューティング [5] が注目され始めた．このプッラトフォームは，データの発生場所や消費場所に近い計算資源をエッジサーバとして使用し，クラウドサーバで行われていた処理の一部を分散することで資源不足や通信帯域圧迫を回避する．また，エッジサーバと IoT デバイスの間の通信遅延は条件が良いときには数ミリ秒に抑えられるため，アプリケーションの低遅延化が可能である．

エッジコンピューティングにおいて，情報の処理ノードはクラウドサーバ，移動体基地局に取り付けられた計算資源，IoT ゲートウェイに取り付けられた計算資源，センサやアクチュエータの持つ計算資源など様々である．これらの処理ノードは処理能力とネットワーク状態が異なり，IoT アプリケーションの処理形態も様々である．そのため，エッジサーバを効率的に利用するには，どのアプリケーションをどのノード上に分配するべきか決定する手法が重要である．

タスク分配の手法は静的な手法と動的な手法に分類可能である．静的なタスク分配手法として，計算量が小さく計算頻度の高い処理をエッジサーバで行い，ビッグデータ解析のような膨大な計算を必要とする処理をクラウドサーバで行う手法がある．この手法には 2 つの欠点がある．1 つは，どの程度までの処理をエッジサーバで行うべきかはタスクをデプロイする環境によって異なる点である．つまり，アプリケーションを開発する度にどのような環境にデプロイされるか調査する必要がある．また，複数の環境にデプロイする場合，その環境ごとにアプリケーションの設定を見直す必要がある．もうひとつは，IoT デバイスの増加によってエッジサーバへの負荷が大きくなることである．エッジサーバの処理能力不足がアプリケーションのボトルネックになった際，エッジサーバの増強かアプリケーションの再設計が必要となってしまう．このような欠点のため，エッジコンピューティングにおけるタスク分配は動的な手法が望ましい．

動的なタスク分配手法として，エッジサーバの計算資源の情報や通信リンクの情報を監視し，その情報からタスク分配をグラフ分割問題や多次元ナップサック問題として解く手法がある [6,7]．これらの手法では，環境の変化に応じて効率的なタスク分配先の決定が可能である．しかし，これらの手法は情報の収集や計算をあるノードに集約して行うため，デバイス増加時に情報収集や計算のコストが増加し，大きな負荷が掛かる．また，管理ノードの障害時に弱いという欠点もある．

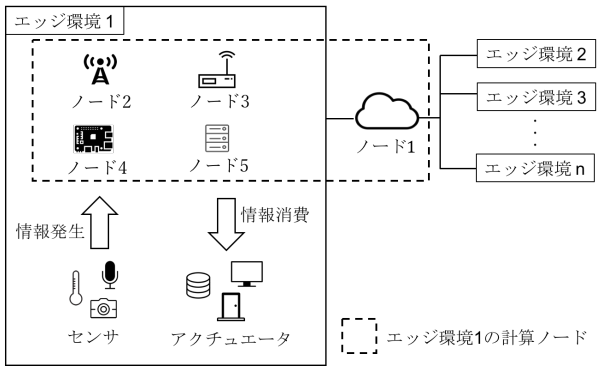


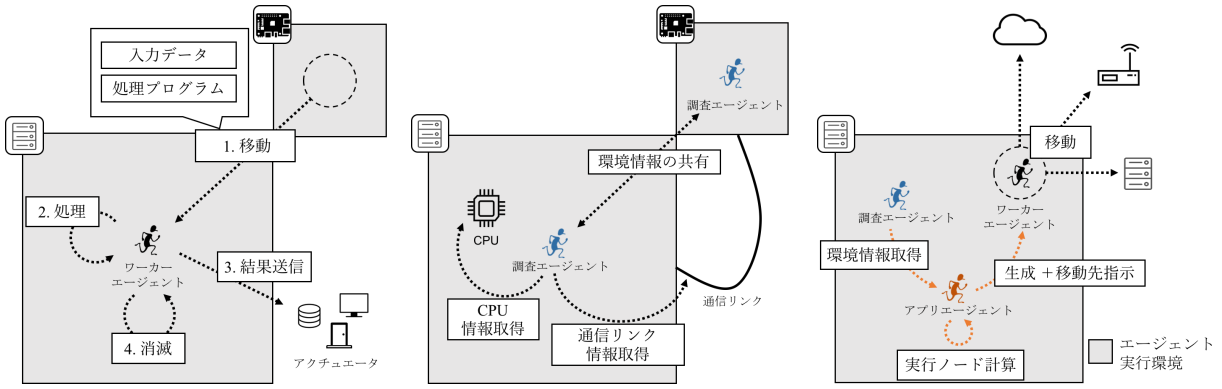
図 1 想定するエッジコンピューティングの構成

そこで，本研究では IoT におけるアプリケーションが自律的に最適な実行ノードを決定し処理をオフロードする手法を提案する．この手法は開発者が使用されるサーバの環境を考慮せずに IoT アプリケーションを作成することを可能にする．また，エッジコンピューティングを使用されるアプリケーションの特性や数，ネットワーク状態の変化に強いシステムの構築を可能にする．

以降の構成は以下のようになる．２章では本研究の想定ネットワークについて述べる．３章では自律分散型タスクオフロード手法について述べ，4 章でその実装と実験を行う．最後に，5 章では本稿の結論と今後の課題を述べる．

2. 想定ネットワーク

図 1 に本研究で用いる IoT におけるエッジコンピューティングのモデルを示す．各地にエッジコンピューティングを使用する環境 (以下，「エッジ環境」という) が存在し，各エッジ環境は複数のセンサ，複数の計算ノード，複数のアクチュエータの三種類のノードから構成される．情報はセンサから発生し，計算ノードで処理され，アクチュエータで消費される．センサは，温度や圧力，画像などの実世界の情報を収集する機器である．計算ノードは移動体基地局に取り付けられた計算資源，IoT ゲートウェイに取り付けられた計算資源，センサやアクチュエータを搭載したデバイス (Raspberry Pi などを想定) の計算資源などがある．また，クラウドサーバは複数のエッジ環境で共有して使用する計算ノードとして考慮する．本研究では，電気信号を運動に変換するロボットアームのようなものだけでなく，情報の可視化先や情報の保存先などもアクチュエータと定義する．このようなモデルで処理されるアプリケーションの例として，カメラとオートロック自動ドアとクラウドサーバの連携が挙げられる．カメラと自動ドアはネットワークに接続されているとする．この例では，カメラが



(a) ワーカーエージェント　(b) 調査エージェント　(c) アプリエージェント

図 2　提案システムの概要

センサ，クラウドサーバが計算ノード，自動ドアがアクチュエータとして振る舞う．カメラが動画データをクラウドサーバに送信し，クラウドサーバで画像解析，認証などを行う．最後に，クラウドサーバが自動ドアにドアを開けるか開けないかの制御情報を送信することで，このアプリケーションのタスクが完了する．各計算ノードにはそれぞれ CPU クロック数を定義する．CPU は 1 ノードに対して 1 つであり，シングルコアであるとする．計算ノードは CPU クロック数が異なり，クラウドサーバはクロック数の高い CPU を持つが，センサやアクチュエータの持つ CPU のクロック数は低い．これらの計算ノードを集中的に管理する計算ノードは存在しないとする．計算ノード間の通信リンクは最大スループットとラウンドトリップタイムを定義する．クラウドサーバと別の計算ノードとの通信はインターネットを通すためラウンドトリップタイムが大きい．また，ZigBee や Bluetooth などの無線通信を利用する通信リンクでは最大スループットは小さくなる傾向にある．

IoT アプリケーションはいずれかの計算ノードに配置され，定期的にタスクを実行する．タスクはどの計算ノードでも達成が可能なものと想定する．また，この処理の並列化は想定しない．アプリケーションはタスクの完了に必要な CPU サイクル数とタスクの入力データのサイズを定義する．タスクを円滑に処理するには，これら 2 つのパラメータを考慮した計算ノードの選択が必要である．例えば，必要 CPU サイクル数が大きいアプリケーションでは処理能力の高い CPU を持つ計算ノードを割り当て，入力データサイズが大きい場合は極力通信せずに処理が可能な計算ノードを割り当てるとタスクの円滑な処理が可能である．

3. 自律分散型タスクオフロード手法

本研究では，エッジコンピューティングにおける自律分散型タスクオフロード手法を提案する．このシステムでは，知能ソフトウェアの 1 つであるソフトウェアエージェント (以下，「エージェント」と呼ぶ) [8] を用いる．エージェントは永続性，自律性，社会性，反応性を持ったプログラムである．本手法では，エージェントが動作する環境 (以下，「エージェント実行環境」と呼ぶ) を各計算ノードに配置し，3 種類のエージェントを連携させることによって，自律的なタスクオフロードを実現する．次節以降でそれらのエージェントの動作について述べる．

3.1 ワーカーエージェント

図 2(a) にワーカーエージェントの動作を示す．このエージェントには移動エージェントの特性を持たせる．移動エージェントは処理を行うためのプログラムと処理対象のデータを持って別のエージェント動作環境に移動し，元の環境とは切り離された状態で処理を実行することが可能である [9]．本研究におけるワーカーエージェントは，IoT アプリケーションと処理対象のデータを持って，自らが別のエージェントから与えられた知識を元に最適な実行サーバへ移動し，そこで処理を実行する．処理終了後は処理結果をデータベースや可視化先のサーバ等に送信し，自ら消滅することでタスクを達成する．この手法により，エージェント実行環境を整備すれば，1 箇所にアプリケーションをデプロイするだけですべての実行ノードでそれを使用することが可能となる．今後は多種多様な IoT アプリケーションの展開が期待されるため，この手法を用いてプログラムデータによるサーバの容量の圧迫を避け，スケーラビリティの高いシステムを構築する．

3.2 調査エージェント

図 2(b) に調査エージェントの動作を示す．このエージェントはアプリエージェントがオフロード先を決定する際に必要とする情報を調査し，各ノードの調査エージェントで共有することを目的とする．このエージェントはアプリケーション実行に使用する各ノードに配置する．本研究では CPU 性能と使用率，通信リンクの性能と使用率を取得する．CPU 性能は静的な値のため，ノードが環境に追加された際にそのノードが自ノードについてのみ調査する．通信リンクの性能はノードの追加によって変化する可能性

があるため，ノードが環境に追加された際，すべてのノードが自ノードと接続可能な別ノードとの間のリンクを調査する．CPU 使用率，通信リンクの使用率に関してはリアルタイムに変化する値のため，定期的に調査する．これらの調査を OS コマンドを利用して行い，取得した情報は別ノードの調査エージェントと共有する．

3.3 アプリエージェント

図 2(c) アプリエージェントの動作を示す．. この目的は，実際にタスクを行うワーカーエージェントを生成し，その実行場所を指示することである．ワーカーエージェントの実行場所を決定するために調査エージェントからの環境情報を利用する．アプリケーション制作者は IoT におけるアプリケーションを実現するためにこのエージェントを設計し，ある 1 か所のノードにデプロイする．このエージェントは実行するプログラムの情報とアプリケーションが優先すべきポリシーを持たせる必要がある．ポリシーはサービス時間を最小にする，通信量を最小にする，使用するノード数を制限する等である．本研究では，IoT アプリケーションのタスク実行受付からタスク実行終了までをサービス時間と定義しこの時間を最小化するポリシーを持つものとする．この時間は，実行ノードまでのデータ転送時間とタスク実行時間の合計時間である．

3.4 タスク実行のフロー

アプリエージェントがタスク実行受付を行ってからタスク実行終了までのフローチャートを図 3 に示す．まず，アプリエージェントの持つ環境情報が自ノードの調査エージェントの最新のものと同様か確認する．違う場合は調査エージェントから新しい情報を得る．次にアプリエージェントはタスク実行ノードの計算を行う．その計算で決定した最適ノードを実行先に指定したワーカーエージェントを生成する．ワーカーエージェントは実行場所が自ノードならそのままタスクを実行し，自ノードでなければ指定ノードに移動した後にそこでタスクを実行する．

3.5 計算アルゴリズム

ここでは，低遅延ポリシー選択時の最適サーバ選択のための計算アルゴリズムを示す．各ノードで実行した際のデータの送信開始から処理終了までの時間 (以下，「サービス時間」という) を推定し，それが最小なノードを選択する手法を採る．アプリケーション i をノード j で実行するとき，エージェントの移動に必要な時間を t_{ij}^{trans}，タスクの実行に必要な時間を $t_{ij}^{\mathrm{process}}$ とする．この時，サービス時間 t_{ij} は次式より求めることができる．

$$t_{ij} = t_{ij}^{\mathrm{trans}} + t_{ij}^{\mathrm{process}} \tag{1}$$

t_{ij}^{trans} は，ノード j へデータを送信する際のスループット r_j，アプリケーション i のタスクがオフロード先ノードへ送信するデータサイズ s_i，ワーカーエージェントのデータサイズ S' を用いて，以下の式で求められる．

$$t^{\mathrm{trans}} = \frac{s_i + S'}{r_j} \tag{2}$$

スループット r_j はウィンドウサイズ W とノード j までのラウンドトリップタイム d_j，最大スループット l_j，通信リンク使用率 u_j^{link} をを用いて以下の式で求められる．

$$\begin{cases} r_j^1 = \frac{W}{d_j} \\ r_j^2 = l_j \cdot u_j^{\mathrm{link}} \\ r_j = \min\{r_j^1, r_j^2\} \end{cases} \tag{3}$$

r^1 は帯域幅が十分広いときのスループット，r^2 は帯域幅が足りず，通信速度が制限される場合のスループットである．スループット r はこのうち，小さいほうのスループットを採用する．また，ノード j が自ノードの場合は $t^{\mathrm{trans}} = 0$ とする．

t^{process} はタスク i の完了に必要な CPU サイクル数 f_i とノード j における CPU クロック数 c_j，CPU 使用率 u_j^{cpu} を用いて以下の式で求められる．

$$t_{ij}^{\mathrm{process}} = \frac{f_j}{c_j \cdot u_j^{\mathrm{cpu}}} \tag{4}$$

以上の式よりノード数を n とし，以下の式を満たすノード j を発見することで最適サーバを決定する．

$$t_{ij} = \min\{t_{i1}, t_{i2}, \cdots, t_{in}\} \tag{5}$$

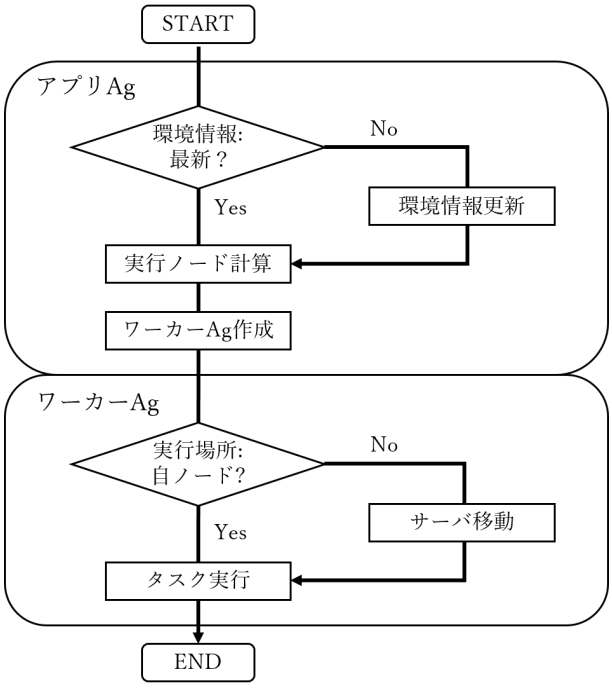


図 3 タスク実行のフロー

4. 実験と考察

本章では本研究の有効性を確認するために提案手法を用いたシミュレーション実験とその考察を行う．

表 1　実験 1: シミュレーション環境

OS	Windows 10
CPU	Intel Core i5-2320@3.0GHz
メモリ	8GB
開発言語環境	Java Development Kit 1.8
エージェント動作環境	DASH/IDEA 1.3.1

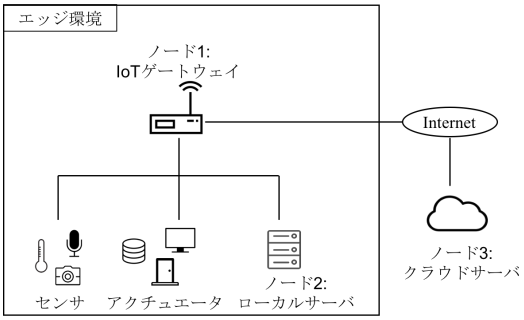


図 4　実験 1: シミュレーションでの想定ノード

表 2　実験 1: 実験パラメータ

ノード 1	
CPU	500 MHz
CPU 使用率	10 %
ノード 2	
CPU	2000 MHz
CPU 使用率	1 秒に 5%単位で上下
ノード 3	
CPU	3000 MHz
CPU 使用率	10 %
ノード 1-ノード 2 間のリンク	
RTT	20 ms
リンク転送レート	10 Mbps
リンク使用率	10 %
ノード 1-ノード 3 間のリンク	
RTT	100 ms
リンク転送レート	10 Mbps
リンク使用率	10 %
アプリケーション	
タスク発生間隔	1 回/秒
必要 CPU サイクル数	100 Mhz
入力データサイズ	100 KB

4.1　実験 1: 自律的なオフロード先決定に関する検証

管理ノードを持たない提案システムにおいて，エージェントがタスクを実行する際に自律的にオフロード先を決定可能であることを検証する．

4.1.1　提案システムの実装

はじめに，提案システムの実装を行った．1 台のサーバ上に複数のエージェント実行環境を展開し，それら 1 つ 1 つにノードとしての役割を持たせた．表 1 に使用したサーバの構成を示す．エージェントの設計および実行環境としてエージェント開発環境 IDEA/DASH [10] を用いた．エージェントは IDEA/DASH 内のインタプリタを使用して実行される．また，エージェントは外部プロセスとして

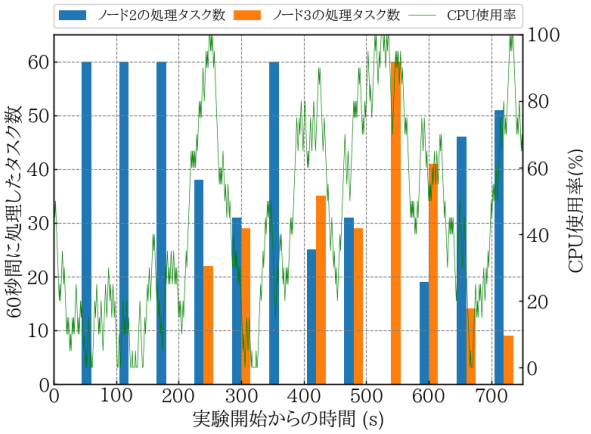


図 5　実験 1: 自律的なオフロード先決定に関する検証

Java プログラムの実行が可能である．ノードの性能は実行環境毎に設定し，通信リンクの性能は各実行環境間で設定した．ノードや通信リンクの使用率は外部プログラムを用いて制御した．

4.1.2　手順

図 4 に想定するノードの種類を，表 2 にそれらのノードの性能を示した．ノード 1，ノード 2，ノード 3 はそれぞれ IoT ゲートウェイ，ローカルに設置されたサーバ，クラウドサーバを想定している．そのため，ノード 3 はノード 1 との通信リンクの性能は低いが，CPU 性能は高く設定した．本実験では，ノード 1 上にアプリケーションエージェントが存在し，そのアプリケーションが実行するタスクをノード 2 やノード 3 にオフロード可能であるとする．オフロードの動きを検証するため，ノード 1 の CPU 性能を低く設定し，常にノード 2 かノード 3 へのオフロードが発生するようにした．ノード 2 の CPU 使用率は外部プログラムによってリアルタイムに変化させ，他のノードの使用率と通信リンク使用率は固定値とした．また，発生させるタスクによる CPU 使用率の変化は考慮しない．以上の環境において，ノード 1 上のアプリケーションにタスクを 1 秒に 1 回ずつ発生させ，ノード 1 のアプリケーションエージェントにそれをオフロードさせた．そのときのノード 2 とノード 3 における 60 秒間のタスク処理数を測定し，オフロード先の変化を検証した．

4.1.3　結果

図 5 にノード 2 の CPU 使用率を変化させたときの自律的なオフロード先決定の様子を示す．横軸は時間軸，縦軸の左はあるノードにおける 60 秒間に処理したタスクの数，右はノード 2 の CPU 使用率である．ノード 2 の CPU 使用率が低い時は通信コストの小さいノード 2 がオフロード対象となり，CPU 使用率が増加するとノード 2 での処理コストが大きくなるためノード 3 へのオフロードが発生している．これらの結果により，アプリケーションエージェントがノードや通信リンクに関する環境情報を用いて各ノー

表 3　実験 2: 測定実験で用いた環境

提案手法	
Product name	Raspberry Pi 3 Model B
OS	Raspbian Jessie
CPU	ARM Cortex-A53 @1.2GHz
Memory	1GB
比較手法	
Product name	Virtual Machine (Amazon EC2)
OS	Ubuntu16.04
CPU	Intel Xeon E5-2670v2@2.50GHz
Memory	1GB

ドでの実行コストを計算することで，自律的かつ動的にタスクのオフロード先を決定していることを確認した．

4.2　実験 2: 実行先ノード決定時間

提案手法とクラウドで複数のエッジ環境を管理する手法について，実行ノード決定時間の解析と比較を行う．

4.2.1　比較手法

比較手法として，複数のエッジ環境に存在する複数のエッジノードのアプリケーションについて，1 つのクラウドサーバで実行ノード計算を集中的に行う手法を用いる．本実験では，1 つのエッジノードにつき，1 アプリケーションがデプロイされていると想定する．本手法では，各エッジ環境が計算を行うクラウドサーバに環境情報をまとめて送信し，その環境内のアプリケーションについて，クラウドサーバがまとめて計算を行うとする．また，各エッジ環境からの計算リクエストの到着はポアソン分布に従うとする．

4.2.2　手順

まず，1 つのエッジ環境における実行ノード計算時間の測定実験を行った．表 3 に測定に使用した環境を示す．提案手法の計算を行うエッジノードはシングルボードコンピュータの Rapberry Pi を，比較手法の計算を行うクラウドサーバには Amazon EC2 のサーバを使用した．

次に，測定実験におけるエッジ環境あたりのノード数 20 の場合の計算時間を使用し，複数のエッジ環境が存在する環境で，再計算間隔を変化させた際の計算の実行ノード決定時間を解析した．実行ノード決定時間は提案手法では比較手法ではエッジノードでの計算時間，クラウドサーバを利用する待ち時間と計算時間の合計である．本実験では，エッジ環境の数を 100 と想定した．

4.2.3　結果

図 6 に測定実験の結果を示す．横軸がエッジ環境あたりのノード数，縦軸がそれぞれの手法の計算時間である．また，エラーバーは計算時間の標準偏差を表す．両手法とも，ノードの増加により，実行先ノードの候補が増えるため，計算時間は増加した．また，提案手法では各ノードの

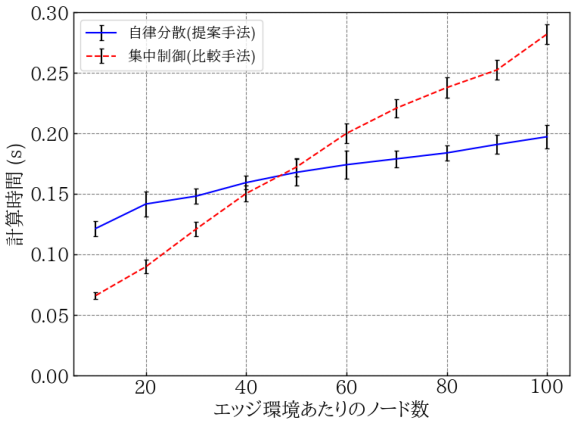


図 6　実験 2: 1 つのエッジ環境における実行ノード計算時間の測定実験

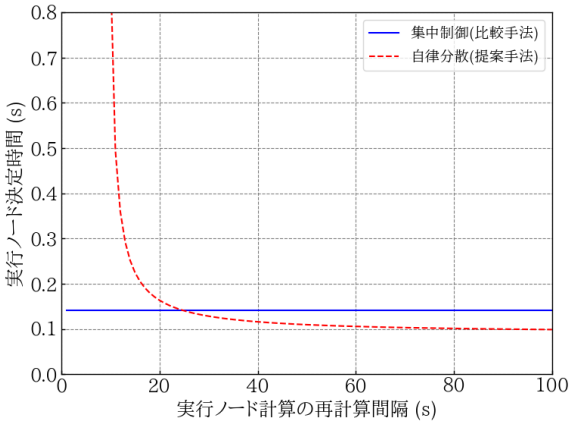


図 7　実験 2: エッジ環境数 100，エッジ環境あたりのノード数が 20 の場合の実行ノード決定時間

アプリケーションが実行先を計算する一方，比較手法では環境ではノード数に応じて計算回数が増加するため，合計の計算時間の増加率は大きくなった．測定実験から得られたエッジ環境あたりのノード数が 20 の場合の計算時間は，提案手法が 0.141 秒，で比較手法が 0.090 秒である．この測定値を使用して行った数値解析の結果を図 7 に示す．横軸が実行先ノードの再計算間隔，縦軸が平均実行ノード決定時間である．今回の条件では，再計算間隔が約 24 秒以下の場合，提案手法のほうが平均実行ノード決定時間が早くなった．また，クラウドサーバで集中的に計算を行う比較手法では，計算時間が大きくなりすぎるため，再計算間隔を 10 秒以下にすることは困難であることが分かる．

4.3　考察

実験 1 では，IoT アプリケーションのタスクをアプリケーションエージェントが自律的にオフロード可能であることを示した．管理ノードが存在しないエッジコンピューティングシステムを構築することにより，ノードの追加が柔軟に行え，障害に強いシステムの構築が可能になる．実

験 2 では，自律分散型のタスクオフロード手法と集中制御型のタスクオフロード手法における実行ノード決定時間を比較した．集中制御型では，再計算間隔を長く取らなければ実行ノード決定時間を低減できないことがわかった．再計算間隔を長くした場合，環境の変化への対応が遅れ，非効率なタスクの配分が行われる可能性がある．よって，自律分散型のタスクオフロード手法は複数のエッジ環境が存在する場合，システムの負荷を低減する手法として有効である．

5. おわりに

この研究ではエッジコンピューティングにおける自律分散型タスクオフロードに焦点を当てた．これにより，処理サーバの事前設定や処理ノードを管理するノードを設置なしに IoT アプリケーションのタスクオフロードが可能となる．実験では，提案手法が環境の変化に応じてオフロード先を動的に変更可能であることを示した．本手法で用いている自律分散型制御には各ノードにおける情報収集やエージェント間のやり取りによるオーバーヘッドが発生する欠点がある．そのため，今後はそれらのオーバーヘッドがシステムにもたらす影響について調査する必要がある．エッジコンピューティングにおける自律分散型タスクオフロードは開発者の負担を減らしつつ，今後急激に増加する IoT アプリケーションへの対応も可能なため，IoT の今後の発展をサポートすることが期待される．

参考文献

[1] J. Jin, J. Gubbi, S. Marusic and M. Palaniswami, ”An Information Framework for Creating a Smart City Through Internet of Things,” in IEEE Internet of Things Journal, vol. 1, no. 2, pp. 112-121, April 2014.
[2] Y. Duan and X. Yu, ”Multi-robot system based on cloud platform,” 2016 IEEE Chinese Guidance, Navigation and Control Conference (CGNCC), Nanjing, 2016, pp. 614-617.
[3] ”Cisco global cloud index: Forecast and methodology, 2014-2019 white paper,” 2014.
[4] D. Evans, ”The Internet of Things: How the next evolution of the Internet is changing everything,” CISCO White Paper, vol. 1, pp. 1-11, 2011.
[5] Garcia Lopez, Pedro, et al. ”Edge-centric computing: Vision and challenges.” ACM SIGCOMM Comput. Commun. Rev., vol 45, no 5, pp. 37-42 2015.
[6] S. Ningning, G. Chao, A. Xingshuo and Z. Qiang, ”Fog computing dynamic load balancing mechanism based on graph repartitioning,” in China Communications, vol. 13, no. 3, pp. 156-164, March 2016.
[7] V. B. Souza, X. Masip-Bruin, E. Marin-Tordera, W. Ramirez and S. Sanchez, ”Towards Distributed Service Allocation in Fog-to-Cloud (F2C) Scenarios,” 2016 IEEE Global Communications Conference (GLOBECOM), Washington, DC, 2016, pp. 1-6.
[8] Wooldridge, M.; Jennings, N. R. (1995). Intelligent agents: theory and practice. 10(2). Knowledge Engineering Review. pp. 115152.
[9] Satoh I. (2013) A Framework for Data Processing at the Edges of Networks. In: Decker H., Lhotsk L., Link S., Basl J., Tjoa A.M. (eds) Database and Expert Systems Applications. DEXA 2013. Lecture Notes in Computer Science, vol 8056. Springer, Berlin, Heidelberg
[10] 木下哲男. “idea”. http://www.k.riec.tohoku.ac.jp/idea/.