エッジコンピューティングにおける IoT アプリケーションの最適配置

2014SE017 林龍之介　2017SE038 幸村颯人
2017SE077 杉田凱也
指導教員：宮澤元

1　はじめに

身の回りのあらゆるものをインターネットに接続する技術 Internet of Things (IoT) が普及している．IoT デバイスからの大量のデータを処理するための技術として，エッジコンピューティングが注目されている．エッジコンピューティングとは，エッジサーバと呼ばれる IoT デバイスからネットワーク的に近い場所の計算リソースを活用する技術である．IoT デバイスからのデータ処理要求をエッジサーバで処理することでクラウドの負荷の軽減につながる．しかしクラウドと異なり，エッジコンピューティングではさまざまな計算ノードがエッジサーバとして利用されるので，行わせる処理の効率は使用するエッジサーバによって異なる可能性がある．

そこで，エッジコンピューティングにおけるアプリケーション実行基盤は，アプリケーションを動作させるエッジサーバを選択する際に計算ノードの多様性を考慮するようにする必要がある．一方，クラウドでの利用を前提として開発された既存のコンテナオーケストレータは，クラウドの計算ノードの均一性を前提として，計算ノードの負荷状況だけを用いてアプリケーションを実行する計算ノードを決定するので，エッジコンピューティングで利用するには不十分である可能性がある．

本研究の目的は，エッジコンピューティングにおいて計算ノードの負荷状況が IoT アプリケーションの配置に与える影響を調査することである．

これを踏まえ以下の 2 点を研究課題とする．

- 既存のコンテナオーケストレータにおけるノード負荷に応じたサービスの配置状況の確認
- エッジコンピューティングにおけるサービスインスタンスの配置手法に関する考察

具体的には，異なる種類の計算ノードを利用するコンテナオーケストレータを用いてサービス起動実験を行う．ノード負荷を変更してサービスを起動する場合，異なる種類の計算ノードを含むクラスタでのサービスインスタンス配置は必ずしも最適な状況にはならないことを確認する．

2　研究の背景

コンテナオーケストレータとは，コンテナの監視と制御を行うためのシステムである．ここで，コンテナとはプロセスの名前空間と使用するリソースを制御することによって作り出された隔離空間である．コンテナを利用すると 1 台の物理ホスト上で複数の独立したサーバ環境を提供することができる．コンテナ型仮想化には Docker[1] と呼ばれるコンテナ型仮想化環境が用いられることが多い．

Kubernetes[2] は代表的なコンテナオーケストレータの一つである．同じようなリソース要求であったり，通信を共有する複数のコンテナをポッドと呼ばれる単位でまとめてコンテナの管理を行う．1 つのポッドには Docker などのコンテナが 1 つ以上含まれている．Kubernetes は，ポッドの CPU とメモリのリソース要求に合わせてスケジューリングを行い，管理する複数のノードからいずれかを選んでポッドを実行する．

ポッドのリソース要求にあったノードを選び出す際に kube-scheduler[3] が動作しており，フィルタリング，スコアリングと呼ばれる作業を行なっている．フィルタリングでは，ポッドの要求が CPU とメモリ以外（ノードの指定など）にないかを確認しており，ポッドのリソース要求を満たすノードを探し出す．要求を満たすノードが複数ある場合にはスコアリングが行われる．スコアリングでは，フィルタリングで選び出されたノードのリソース使用量などをみてランク付けを行う．ランク付けされたノードの中から，最もランクの高いノードが選択される．Kubernetes では，このような工程でポッドを配置するのに適したノードを選び出している．

3　エッジコンピューティングにおけるサービス配置

エッジコンピューティング環境において，Kubernetes 標準のスケジューリングアルゴリズムがサービスを動作させるポッドを配置する計算ノードを選ぶ方法について説明する．一般的にエッジサーバは，クラウドサーバに比べ性能が低い．そのため，Kubernetes のクラスタ内の計算ノードに性能差が生じる．エッジコンピューティング環境でポッドが一つも配置されていない場合，ポッドのリソース要求を満たすサーバの中でも性能の良いサーバが選ばれる．性能の良いサーバというのは，CPU のコア数やメモリ量が他のサーバと比べて多いサーバのことである．次に，クラスタ内でポッドが配置されているサーバがいくつかある場合，まだポッドが配置されていないサーバで新しく作成されたポッドのリソース要求を満たしているサーバがあれば，その中から性能の良いサーバが選ばれる．このとき，すでにポッドが配置されているサーバが新しく作成されたポッドのリソース要求を満たしており，サーバのリソース容量に余裕があったとしても，ポッドが一つも配置されていないサーバが選ばれる．すべてのサーバにすでに

ポッドが配置されている場合，ポッドのリソース要求を満たすサーバの中で残りリソース容量が多いサーバが選ばれる．もし，他のポッドが配置されていることによって，新しく作成されたポッドのリソース要求を満たすサーバが一つもなかった場合，新しく作成されたポッドは待機状態になる．待機状態になったポッドは，すでに配置されているポッドが処理を終了し，削除されるとそのポッドがあったサーバへと自動的に配置される．

4 実験

この節では，エッジコンピューティング環境を模した環境でサービスインスタンスを配置する実験について示す．

4.1 実験の目的

異なる種類の計算ノードを利用するコンテナオーケストレータで起動したサービスが配置される様子を確認するために実験を行った．ノード負荷を変更した場合に異なる種類の計算ノードを含むクラスタでのサービスインスタンス配置は必ずしも最適な状況にはならないことを確認する．

4.2 実験内容

図 1 のようにマスターノードと，計算ノードとしてクラウドサーバ 2 台とエッジサーバ 2 台を持つコンテナオーケストレータを利用してサービス起動実験を行った．新規にサービスを起動する際に，サービスが起動されるサーバを確認する．また，IoT デバイスに見立てた利用者ノードを用意し，起動したサービスへのアクセスにかかった時間を計測した．

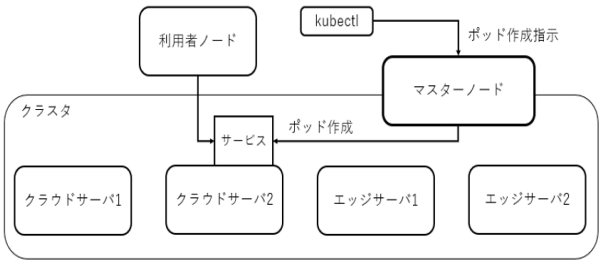

図 1 システム構成

4.2.1 実験の種類

サービスの起動状況を以下のように変更していく 3 種類の実験を行った．

- 同じリソース要求のポッドを作成していく
- 新規ポッドのリソース要求を満たせないようにポッドを配置していく
- サーバの残りリソース容量が少ない状況で高負荷のサービスを起動する

4.3 実験環境

コンテナオーケストレータには Kubernetes 1.19.4 を用いた．利用した計算機の仕様を表 1 に示す．各ノードは 1000Base-T Ethernet で接続した．負荷状況を変更するためのサービスとして nginx 1.19.6 を用いた．高負荷サービスとして http でアクセスすると円周率を 2000 桁計算して返送するサービスを用いた．Web サーバへのアクセスには curl 7.68.0 を用い，端末への表示オーバヘッドを無視するために標準出力を /dev/null にリダイレクトした．アクセス時間の計測には bash 5.0.17 の time コマンドを利用し real time の小数点以下 2 桁を有効として扱った．

4.4 実験結果

4.4.1 同一リソース要求ポッドの逐次作成実験

以下の手順に従い，リソース要求がそれぞれ同一のポッド 1 からポッド 4 を順に作成する実験を行った．各サーバに配置されているポッド数と残りリソース容量の変化を表 2 に示す。

表 2 同一リソース要求ポッドの逐次作成実験における残りリソース量変化

	クラウドサーバ 1	クラウドサーバ 2	エッジサーバ 1	エッジサーバ 2
初期状態				
	8, 32GiB	12, 32GiB	4, 4GiB	4, 4GiB
手順 1		ポッド 1		
	8, 32GiB	11.5, 31GiB	4, 4GiB	4, 4GiB
手順 2	ポッド 2	ポッド 1		
	7.5, 31GiB	11.5, 31GiB	4, 4GiB	4, 4GiB
手順 3	ポッド 2	ポッド 1	ポッド 3	
	7.5, 31GiB	11.5, 31GiB	3.5, 3GiB	4, 4GiB
手順 4	ポッド 2	ポッド 1	ポッド 3	ポッド 4
	7.5, 31GiB	11.5, 31GiB	3.5, 3GiB	3.5, 3GiB

※ 上段: 配置されているポッド
下段: 残りリソース量 (CPU コア数, メモリ量)

手順 1 ポッド 1(リソース要求: CPU コア 0.5, メモリ 1GiB) を作成
ポッド 1 のリソース要求はすべてのサーバの残りリソース容量の範囲内なので，残りリソース容量が最も多いクラウドサーバ 2 へ配置される．

手順 2 ポッド 2(リソース要求: CPU コア 0.5, メモリ 1GiB) を作成
まだポッドが一つも配置されていないサーバの中で残りリソース容量が最も多いクラウドサーバ 1 が配置先になる．

手順 3 ポッド 3(リソース要求: CPU コア 0.5, メモリ 1GiB) を作成
まだポッドが一つも配置されていないエッジサーバ 1

表 1　利用した計算機の仕様

ノード	マスター	サービス利用	クラウド 1	クラウド 2	エッジ 1	エッジ 2
CPU	Intel(R) Core(TM) i7-7700K CPU @ 4.20GHz	Intel(R) Core(TM) i7-8700K CPU @ 3.70GHz	Intel(R) Core(TM) i7-7700K CPU @ 4.20GHz	Intel(R) Core(TM) i7-8700K CPU @ 3.70GHz	NVIDIA Jetson Nano Developer Kit Tegra X1 @ 1.48GHz	
RAM	32GB	32GB	32GB		4GB	
OS	Ubuntu 20.04.1 LTS	Ubuntu 20.04.1 LTS	Ubuntu 20.04.1 LTS		Ubuntu18.04.5	
CPU コア数	8	12	8	12	4	

へ配置される．

手順 4　ポッド 4(リソース要求: CPU コア 0.5, メモリ 1GiB) を作成
まだポッドが配置されていないエッジサーバ 2 へと配置される．

4.4.2　サーバがリソース要求を満たせないポッドの作成実験

以下の手順に従い，すべてのサーバがリソース要求を満たせるポッド 1，ポッド 2 を作成した後に，サーバがリソース要求を満たせないポッド 3 を作成する実験を行った．各サーバに配置されているポッド数と残りリソース容量の変化を表 3 に示す

表 3　サーバがリソース要求を満たせないポッドの作成実験における残りリソース量変化

	クラウドサーバ 1	クラウドサーバ 2	エッジサーバ 1	エッジサーバ 2
初期状態	8, 32GiB	12, 32GiB	4, 4GiB	4, 4GiB
手順 1		ポッド 1		
	8, 32GiB	9.0, 29GiB	4, 4GiB	4, 4GiB
手順 2	ポッド 2	ポッド 1		
	5.0, 29GiB	9.0, 29GiB	4, 4GiB	4, 4GiB
手順 3	ポッド 2	ポッド 1		
	5.0, 29GiB	9.0, 29GiB	4.0, 4GiB	4.0, 4GiB

※ 上段: 配置されているポッド
下段: 残りリソース量 (CPU コア数, メモリ量)

手順 1　ポッド 1(リソース要求: CPU コア 3.0, メモリ 3GiB) を作成
残りリソース容量が最も多いクラウドサーバ 2 へ配置される．

手順 2　ポッド 2(リソース要求: CPU コア 3.0, メモリ 3GiB) を作成
まだポッドが一つも配置されていないサーバの中で残りリソース容量が最も多いクラウドサーバ 1 が配置先になる．

手順 3　ポッド 3(リソース要求: CPU コア 10, メモリ 30GiB) を作成
ポッド 3 は配置されず，待機状態となった．

4.4.3　さまざまなサービスの実行中に高負荷サービスを起動する実験

さまざまなリソース要求をしているポッドを作成していき，各サーバの残りリソース容量に余裕がなくなった状況で，高い負荷のサービスを起動する実験を行った．以下の手順に従って，ポッドを作成した．

手順 1　ポッド 1(リソース要求: CPU コア 2.4, メモリ 9.6GiB) を作成

手順 2　ポッド 2(リソース要求: CPU コア 9.6, メモリ 25.6GiB) を作成

手順 3　ポッド 3(リソース要求: CPU コア 0.5, メモリ 1GiB) を作成

手順 4　ポッド 4(リソース要求: CPU コア 0.5, メモリ 1GiB) を作成

手順 5　ポッド 5(リソース要求: CPU コア 1.6, メモリ 21.4GiB) を作成

手順 6　ポッド 6(リソース要求: CPU コア 1.5, メモリ 5GiB) を作成

手順 7　ポッド 7(リソース要求: CPU コア 1.5, メモリ 1GiB) を作成

手順 8　ポッド 8(リソース要求: CPU コア 3.0, メモリ 3GiB) を作成

以上の手順を終了した時のサーバのポッド配置と残りリソース状況を表 4 に示す．クラウドサーバ 1 とエッジサーバ 2 の残りリソース容量が比較的多く，前者が CPU コア数 4.0 とメモリ量 1GiB，後者が CPU コア数 2.0 とメモリ量 2GiB となっている．

この状況で，高負荷のサービスを組み込んだポッドを起動した．このポッドのリソース要求を CPU コア数 1.0, メモリ量 1.0 GiB としたところ，残りメモリ量が多いエッジサーバ 2 に配置された

この実験で用いた高負荷サービスについて，クラウドサーバで動作する場合とエッジサーバで動作する場合のアクセス時間の差を確認する実験を行った．ほかに動作するポッドが何もない状況で，ポッドをクラウドサーバ 1 と

表 4 高負荷サービス起動実験におけるサーバ状況

サーバ名	クラウド 1	クラウド 2	エッジ 1	エッジ 2
配置されているポッド	1,5	2,6	3,8	4,7
残り CPU コア数	4.0	0.9	0.5	2.0
残りメモリ量	1GiB	1.4GiB	0GiB	2GiB

エッジサーバ 2 に配置し，利用者ノードからのアクセス時間を計測した．各サーバで 10 回ずつ実験を行った結果の平均と標準偏差を表 5 に示す．アクセス時間の平均は，ポッドをクラウドサーバ 1 で実行する方がエッジサーバ 2 より約 3.7 秒短く，およそ 3.8 倍速かった．

表 5 高負荷サービスへの利用者ノードからのアクセス時間 (秒)

	クラウドサーバ 1	エッジサーバ 2
平均	1.31	5.02
標準偏差	0.01	0.01

5 考察

我々は，標準の Kubernetes のスケジューリング方法で発生する問題の原因は，新たに作成したポッドを，サーバのリソース状況を考慮することなく，未使用のサーバに対して優先的に配置していることにあると考えた．

本研究の実験で起きた具体的な問題は以下の 2 点である．

- リソースが他のポッドの処理に割かれ，新規のポッドが配置されずに待機状態になる
- サーバの持つ CPU の性能を考慮せずに，ポッドの配置を行う．

これらの問題を解決するためにスケジューリングの拡張を行う必要があると考える．具体的には，ポッドに優先度を付け，優先度を考慮したスケジューリングを行うことや，候補に選ばれたサーバに順位を付け，順位付けを考慮したスケジューリングを行うことを検討する．

4.4.1 節の実験では，標準の Kubernetes のスケジューリング方法では，サーバのリソース状況を考慮することなく，新たに作成されたポッドを，まだポッドを配置していない未使用のサーバに対して優先的に配置していることが分かった．サーバのリソース状況を考慮してポッド配置を行うためにはスケジューラを拡張する必要があると考える．

4.4.2 節の実験では，ポッドのリソース要求がサーバの空きリソースを上回る場合，どのサーバにも配置されることなく待機状態になることが分かった．これを防ぐためには，既に配置されたポッドとこれから配置を行うポッドの優先度を比較しながらスケジューリングを行う必要があると考える．

4.4.3 節の実験では，本来クラウドサーバに配置したいポッドを配置できずにエッジサーバに配置しなくてはならない問題が発生することが分かった．表 5 からわかるように，相対的な残りリソース容量が少ないとしても，計算性能自体が高いクラウドサーバにポッドを作成する方が良い結果になることがある．しかし，ポッドの配置に際して計算ノードの計算性能を考慮に入れない場合，相対的に残りリソース容量が多いエッジノードが選ばれてしまうことになる．このようなスケジューリング方法ではサーバが持つリソースを効率的に活用することができないので，我々はスケジューリング方法を改善する必要があると考える．

6 おわりに

本研究では，エッジコンピューティング基盤において IoT アプリケーションを適切なサーバへ配置する手法を検討した．我々は標準の Kubernetes のスケジューリングでは作成されたポッドがどのように配置されるのかを示すために，複数の処理性能が違うサーバを用意し実験を行った．結果として標準の Kubernetes のスケジューリングではまだポッドが配置されていないサーバへ優先的にポッドを配置することが分かった．

問題点として，クラウドサーバにもリソースの余裕があるにもかかわらず，処理負荷の大きいサービスを処理性能が低いエッジサーバに配置してしまう場合があることを示した．従って，ポッドの最適配置を行うためには，CPU の処理性能を考慮して，クラウドサーバに処理負荷の大きいポッドを優先的に配置し，エッジサーバでは比較的処理負荷の小さいポッドを配置することが適切であると考える．

今回の実験は，利用者ノードからクラウドサーバとエッジサーバへの通信遅延がほぼ同じ環境で行ったが，エッジサーバは本来であれば利用者の近くに配置されるのが一般的である．従って，実際のポッドの配置にあたっては，利用者ノードとサーバ間の通信遅延を考慮する必要があると考える．

今後の課題は，実際にサーバの CPU の処理性能と利用者ノードとサーバ間の通信遅延を考慮してスケジューリングを行えるようにすることである．また，ポッドの最適配置を行うためには，これら以外にもサーバの計算リソースやサービスの処理内容なども判断基準として検討する必要がある．最終的には検討したスケジューラの拡張を Kubernetes に組み込んで実装の評価を行う．

参考文献

[1] Docker overview，https://docs.docker.com/get-started/overview/

[2] Kubernetes の概要，https://kubernetes.io/ja/docs/concepts/overview/

[3] Google.Inc: Kubernetes のスケジューラー，https://kubernetes.io/ja/docs/concepts/scheduling-eviction/kube-scheduler/.