ロードバランサーを用いた追加サーバーのCPU使用量を抑えるソフトウェア

野木 空良[1,a)]　串田 高幸[1]

概要：動的分散方式によるロードバランサーは，最も CPU 使用量の少ないサーバーに対して，動的にリクエストを振り分ける。サーバーは，1 リクエストを処理するのに CPU を消費する為，ロードバランサーのリクエストが増加する．リクエストが増加した際の課題として，新しくサーバーを追加すると，追加したサーバーの CPU 使用量が急激に上昇する．サーバーの CPU 使用量が急激に上昇すると，応答時間の遅れや処理のエラーが発生する．本提案では，新たに追加したサーバへのリクエストを分散するため，仮想サーバの追加と，リクエストの分散を行う．また，本提案は追加したサーバーの CPU 使用量を減少させることを目的とする．その為，ロードバランサーは CPU 使用量の一番低いサーバーにリクエストを割り当てる．実験方法は，Kubernetes の Pod を用いてサーバーを作成し，負荷試験ツールの Locust で HTTP リクエストを送信している時に，二つの Pod を追加して，CPU 使用量の測定を行う．結果として，ロードバランサーが追加したサーバーに送信する負荷を制限していた為，本研究の課題は発生しなかった．しかし，本研究の課題を発生させる過程で Kubernetes の type:NodePort を用いた際，Pod の CPU 使用量に偏りが発生した．原因は，type:NodePort で Port 番号を指定しない場合，Service が Pod に対してランダムにリクエストを送信していることであった．今後は，Pod の CPU 使用量の最も低い Pod にリクエストを割り振るロードバランサーを開発し，type:NodePort と比較した CPU 使用量の計測を行う．

1. はじめに

1.1 背景

近年，クラウドサービスの需要が増加している [1]．クラウドとはユーザーがソフトウェアを持たなくても，インターネットを通じて，必要なサービスを必要な分だけ利用できる仕組みである [2]．クラウドサービスの需要増加に伴い，サーバーへのアクセス数が増加し，サーバー 1 台では管理できない場合がある [3]．その際，サーバー稼働台数の増加や, 仮想サーバーを導入し，負荷分散装置のロードバランサーを使用する．ロードバランサーは静的分散方式と動的分散方式の 2 つに分類される．

静的分散方式

静的分散方式は，例として DNS ラウンドロビンがある [4]．DNS ラウンドロビンは，サーバー運営者があらかじめ決めた順番で，サーバーにリクエストを振り分ける方法である．静的分散方式は，レプリケーション等の，分散先のサーバの処理能力が同じ場合に使われており，分散先はサーバー運営者が決めている．あらかじめ設定した割り当てを行い，負荷を分散させる為，サーバ側にエラーや故障が発生した場合，対処できない問題が発生する．静的分散方式で問題を対処できない場合は，動的分散方式を考える必要がある．

動的分散方式

ロードバランサーの分散方式である動的分散方式の概要を図 1 に示す [5]．図 1 では，ユーザーが，クライアントソフトウェアに，サーバーのアクセスを要求する．その後, クライアントソフトウェアは，サーバーに対してリクエストを送信する．その際，ロードバランサーを用いることで，リクエストの分散が可能となる．動的分散方式には，クライアントソフトウェアから送信されたリクエストを，サーバーの状態を取得したうえで、負荷の少ないサーバーに分散する仕組みがある．つまり動的分散方式で，負荷の少ないサーバーにリクエストを割り当てるには，サーバーの状態を把握する必要がある [6]．例えばファイルサーバーやメールサーバー，ストリーミングサーバーの負荷がデータ転送量に依存する場合は，TCP/UDP コネクション数を測定し，コネクション数が最も少ないサーバーにリクエストを送信する [7]．また，サーバーの CPU 使用量をロードバランサーが収集し，各サーバーの負荷状況によってリクエストを分散する方式もある [8]．CPU 使用量を用いた

[1] 東京工科大学コンピュータサイエンス学部
〒 192–0982 東京都八王子市片倉町 1404-1
[a)] C0118220

負荷分散方式では各サーバーの CPU 負荷を測定して分散する．そのため各サーバの処理能力や運用形態が異なった場合でも，各サーバーの CPU 負荷の状況に応じてリクエストの分散が可能である．オープンソースソフトウェアの Kubernetes は，サーバーのコネクション数や，CPU 使用量を把握し，負荷の動的分散を可能にする [9]．さらに，複数のコンテナを組み合わせてクラスタとして管理することで，サーバーの負荷分散が可能になる [10]．そこでクラスタのスケーリングによる負荷分散，エラーの自動修復，クラスタ間での通信の管理を可能とする Kubernetes を活用することで動的分散方式のロードバランサーを運用することが可能となる．

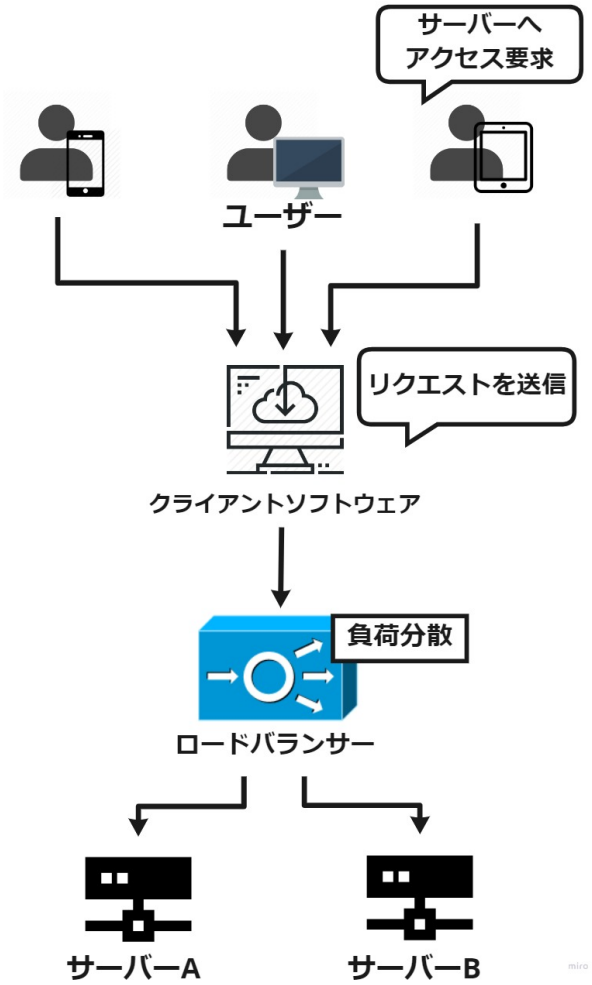

図 1　ロードバランサーの概要

比べて負荷の少ない，追加された「サーバー 5」に，リクエストが送信される．しかし急激なリクエスト数の上昇により，追加された「サーバー 5」の CPU やメモリ，リソースが許容範囲を超えた場合，「サーバー 5」の負荷が上昇する．ユニクロの EC サイトでは，2017 年 11 月 23 日に，サーバーへの大規模なアクセスが集中し，サーバーダウンが発生した．つまり動的分散方式を用いたサービスの運用をする場合は，両方のサーバーの負荷上昇も考える必要がある．株式会社 IIM の調査では，サーバーの CPU 負荷と HTTP ログのアクセス件数の相関係数が 0.7 以上を示しており，二つの相関が高いことを確認している．つまりアクセス数が増加すれば CPU 使用量も上昇する．本研究では追加したサーバーの CPU 使用量上昇を課題とし，追加したサーバーへのリクエスト数を分散させる事を目的とする．

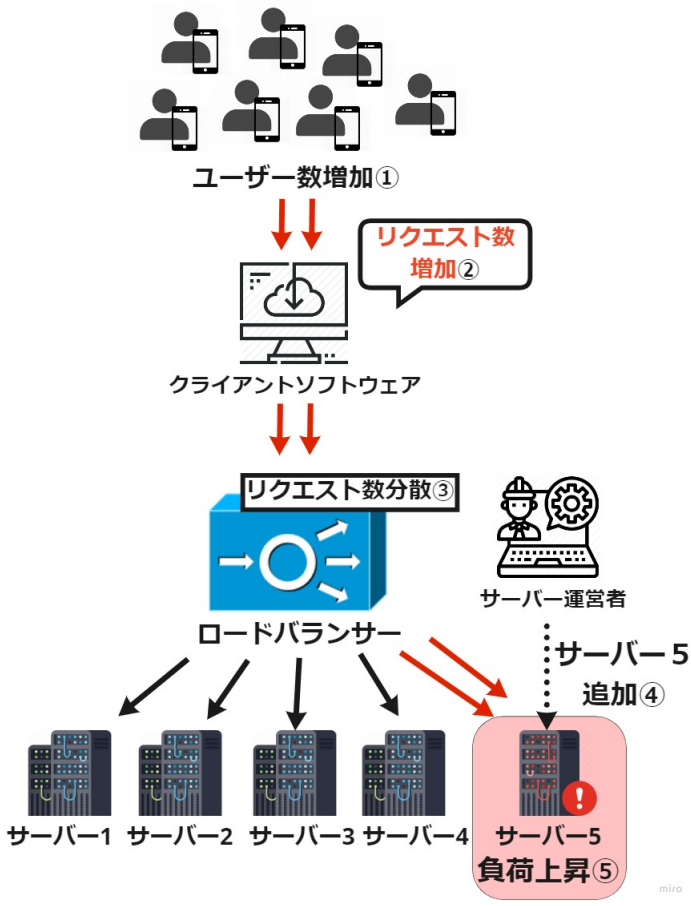

図 2　サーバー 5 の CPU 使用量上昇

課題

ロードバランサーの課題を図 2 に示す．図 2 では，ユーザー数が増加し，サーバー 5 を追加した際に，サーバー 5 の負荷が上昇することを示している．例えば，期間限定イベントでは，Web サイトの人気が上がると，ユーザーのアクセス数が増加し，サーバーを追加する必要がある [11]．動的分散方式の場合、「サーバー 1」から「サーバー 4」に

各章の概要

2 章の関連研究では，今回の研究に関連した他の研究を提示し比較を述べる．次に，3 章の提案では，今回の研究における提案内容を解説する．さらに，4 章の実装と実験環境では，実装方法と実験環境について示していく．5 章の評価と分析では，検証方法とその結果を述べる．6 章の議論では，現状の課題とその解決方法を議論する．最後

に，今回の研究の提案とそれらを伴い得られた成果を改めて示す．

2. 関連研究

サーバーの負荷に関する研究として，ネットワーク内の負荷を検知するロードバランサーが提案されている [12]．この研究では，ネットワーク速度が加速したときに NIDS センサー (ネットワーク侵入検知センサー) の容量によって、アクセスを検知するシステム機能が制限される課題について述べている．解決案として，ロードバランシングによって，センサーノード全体にネットワークトラフィックの負荷を分散することで制限がかからないシステムを提案している．センサーノードとは，コモディティハードウェアと呼ばれる並列コンピュータの上に，NIDS センサーが搭載されたものである．研究の結果，ネットワークトラフィックの変化に適応し，検知システムの機能が，NIDS センサーの容量に関わらず，運用できるシステムが可能となった．しかし，ネットワークトラフィックの負荷が想定以上に多かった時に，負荷分散が追い付かずシステム機能が制限されてしまう可能性がある．想定外の負荷がかかるのを解決するにはあらかじめ，かかる負荷自体を制限することで解決できる．

また，Kubernetes を用いたロードバランサーの研究では，Kubernetes クラスター用のポータブルロードバランサーの提案をしている [13]．Kubernetes クラスター用のポータブルロードバランサーとは Kubernetes 専用の負荷分散装置である．Kubernetes はクラウドプロバイダーが提供する外部のロードバランサーに依存しているため，サポートされているロードバランサーがない環境で使用することが困難である．そこで提案では，あらゆる環境で使用できるポータブルロードバランサーを提案している．Linux カーネルのインターネットプロトコル仮想サーバーである IPVS を使用して，Kubernetes によって実行されるコンテナ化されたソフトウェアロードバランサーを実装している．ロードバランサーのコンテナ化により IPVS ロードバランサーが機能を犠牲にすることなく Web サービスの移植性の改善を実現可能にした．しかしポータブルロードバランサーで使用できる環境にも限りがある為，実際に使用するまでには至らない．ポータブルロードバランサーを実用化するには，クラウドプロバイダーが提供するロードバランサーのような機能性を持ち，ソフトウェアの用途に合わせた環境を用意する必要がある．

更に，新しいアルゴリズムで分散するロードバランサーを提案している研究がある [14]．提案では負荷の最も少ないサーバーにリクエストを割り振りながら同じサーバーに連続で割り振りを行わないようにする方式を開発している．まず，データセンターからロードバランサーに送信されたリクエストの割り振り先を決める．割り振りの基準には，全てサーバーの中からロードバランサーへのレスポンスの応答速度が，最も早いサーバーに振り分ける手法を使用している．一番最初の割り振り先は全サーバーの応答速度が 0 であることから DNS ラウンドロビンの方式に従っている．次のリクエストが来ると, どのサーバーに振るかをチェックする．サーバーが使用可能な状態であり、かつ前の割り当てで使用されていない場合は，リクエストが割り当てられサーバーの ID がデータセンターに返される．それ以外の場合は次に負荷が少ないものをチェックする．従来のロードバランサーでは最も負荷の少ないサーバーを選択するが，前回の割り当てまではチェックされない．つまり提案方式では前回の割り当てまで考慮するため過大な割り当てを防ぐことが出来る．しかしあらかじめ決められた割り当てでは，レスポンス数が急激に増加し，サーバーを増設したい場合に割り振りをし直さなければならないという点がある．新しいサーバーは応答速度が 0 である為，サーバーをチェックする際に選択に含まれない可能性がある．つまり一番最初の割り振りで従った方式である、DNS ラウンドロビンで新しいサーバーのレスポンスも受け取る必要がある．

3. 提案

図 3 は，提案における仮想サーバーの追加と削除の手法をあらわす．本研究の提案システムは，リクエストを分散させるロードバランサー，課題である CPU 使用量が上昇したサーバー，CPU 使用量の上昇率を減少させる目的である追加サーバーと仮想サーバーの 4 つによって構成されている．本研究の目的は，サーバーの CPU 使用量の減少である為，ロードバランサーの構成は，CPU 使用量の低いサーバーに割り振るアルゴリズムとする．クライアントソフトウェアのリクエスト数が増加すると，ロードバランサーはサーバー 1 からサーバー 4 の，4 つのサーバーにリクエストを割り振る．サーバー 1 からサーバー 4 の負荷が上昇した際，サーバー 5 を手動で追加する．次に，サーバー 5 の追加と同時に，仮想サーバーの自動追加を行う．サーバー 5 のリクエストを分散する事で，サーバー 5 のリクエストを分散させ，CPU 使用量が減少し，課題の解決が可能となる．更に，サーバー 5 のリクエストを分散後，仮想サーバーを稼働する必要が無い為，自動で削除をする．自動削除の条件は，サーバー 1 からサーバー 4 の CPU 使用量の平均値を取り，サーバー 5 が平均値に達した時とする．以下の 5 つの手順は，課題を解決するまでの流れである．

(1) クライアントソフトウェアからのリクエストが上昇
(2) CPU 使用率の最も低いにサーバーにリクエストを振り分ける
(3) サーバー 5 と仮想サーバーを同時に自動追加する
(4) サーバー 5 のリクエストを分散させて CPU の上昇率を減少させる

(5) 仮想サーバーの自動削除を行う

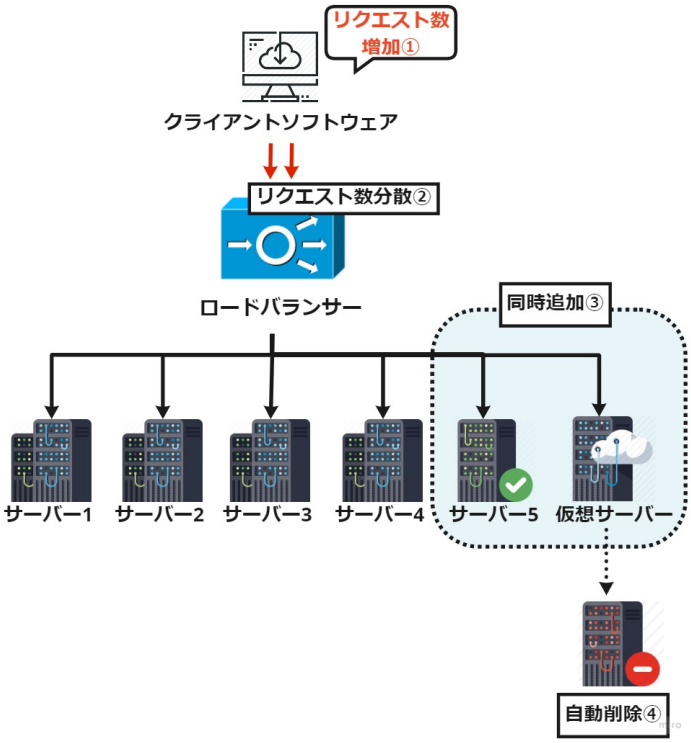

図 3 仮想サーバーの同時追加と削除

4. 実装と実験環境

4.1 実装

実装の概要として全体図を図 4 に示す．サーバーは Kubernetes の Pod を用いて実験を行った．Pod は，nginx の cgi ファイルを使って作成した．Pod の中では，CPU の使用量を上げるために，2 の 6 乗を計算するプログラムが動作している．リクエスト数 1 個につき 1 回，2 の 6 乗を計算する仕組みで，CPU が命令を実行し，CPU 使用量を上昇させた．ロードバランサーの機能を付けたソフトウェア (以下 Dispadd と名称) を Pod に入れ，分散対象の Pod と，Pod 間通信を行うことでサーバーの分散を可能にした．さらに，Pod に HTTP リクエストを振り分ける為，Pod の IP アドレスと CPU 使用量を取得した．取得方法は，Kubernetes API を使用した．初めに，負荷試験ツールの Locust で，ロードバランサーの Dispadd に HTTP リクエストを送信する．Dispadd は Kubernetes API にリクエストを送る．次に，Kubernetes API は Pod に対して IP アドレスと CPU 使用量を取得するリクエストを送信する．Pod のレスポンスを受け取った Kubernetes API は，Dispadd にリクエストを渡す．取得した IP アドレス，CPU 使用量を基に Dispadd は，CPU 使用量の一番低い Pod に，HTTP リクエストを送信する．また，送信先の Pod と，Pod 間通信を行うために Dispaad は Pod の中で起動させる．以下は実装の手順である．

(1) Locust で HTTP リクエストを Dispadd に送信
(2) Dispadd は Kubernetes API を使用，各 Pod からデータを取得
(3) CPU 使用率の一番低い Pod にリクエストを送信
(4) ReplicaSet を+2 にして New Pod と Virtual Pod 追加

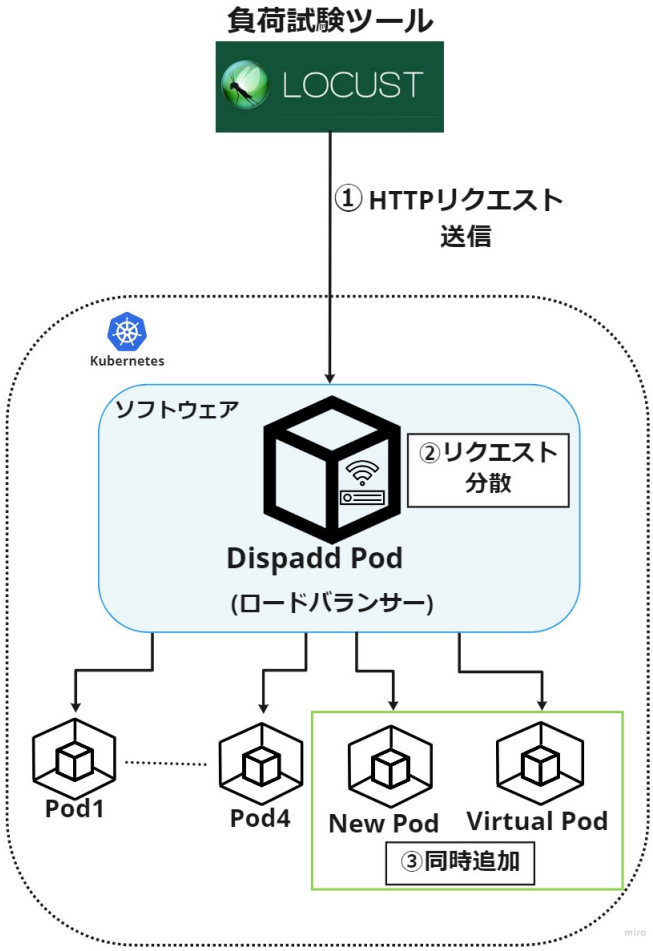

図 4 実装の概要図

実験のソフトウェア構成図を図 5 に示す. 負荷分散機能の作成に，Flask を使用した．Locust で送信する HTTP リクエストは，Python の Flask で Web サーバーを立ち上げて受け取る．次に，Kubernetes API で取得した CPU 使用量と IP アドレスを取得し, 取得した CPU 使用量を基に，一番 CPU 使用量の低い Pod に，リクエストを送信した．

Pod1 から Pod4 のリクエストを分散させる為, New Pod, Virtual Pod の自動追加をする．Pod の追加方法として，Kubernetes の Deployment を使用した．初期のレプリカ数を 4 に設定し，Dispadd で HTTP リクエストを送信する．次に，ReplocaSet を 6 に設定する設定ファイルを実行し，Pod を二つ追加する. その後，Locust で HTTP リクエストを送信し，監視ツールの Datadog で，CPU 使用量を取得する．

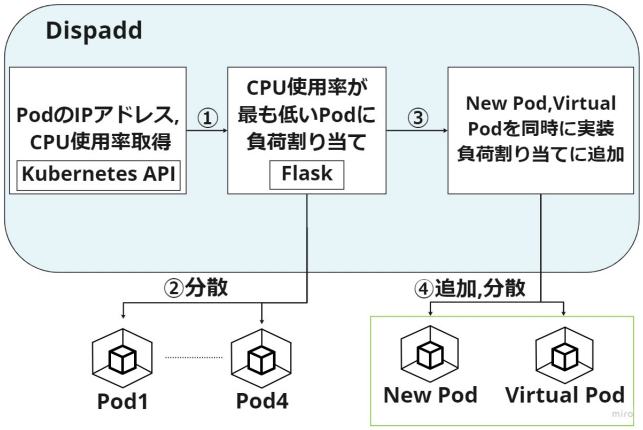

図 5　実験のソフトウェア構成図

4.2　実験環境

実験環境の図を図 6 に示す．本研究の環境は，Ubuntu 18.04.2 のマシン上に Kubernetes をインストールして，実験を行った．Pod は，Deployment を使用し，spec の replicas 初期値を 4 に設定した yaml ファイルを実行して生成している．また，Pod 内では，nginx の cgi ファイルを使用し，1 リクエストにつき，2 の 6 乗を行う計算を行うプログラムが構成されている．Pod にリクエストを送信した際に，CPU を消費するプログラムを実行させる必要がある．Pod の CPU 使用量が上昇した際に，New Pod を追加し，New Pod にリクエストを送信させる為である．その為，Pod 内で 2 の 6 乗の計算を行うプログラムを実行させた．Pod の IP アドレスと CPU 使用量を取得する手法は Kubernetes API を採用した．また，Pod 内のプログラムを実行させる為，HTTP リクエストを送信する分散負荷実験ツールとして，Locust を使用した．Locust で送信した HTTP リクエスト，Pod から取得したデータは，Web アプリケーションフレームワークの Flask を使用し，ソフトウェアである Dispadd に構築した．

- OS:Ubuntu 18.04.2
- Kubernetes(コンテナ管理, 複製)
- Kubernetes API(Kubernetes クラスタとの対話)
- Flask(負荷を受けとる Web アプリケーション)

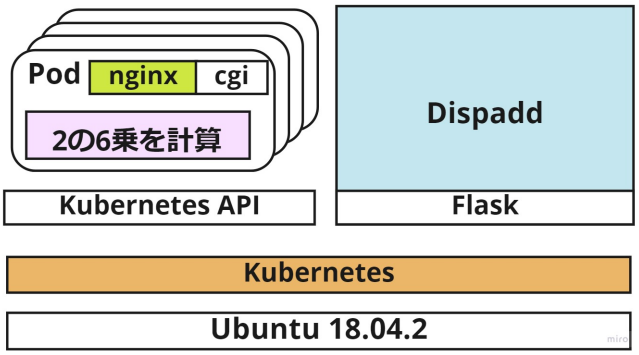

図 6　実験環境の図

5.　評価と分析

本研究の課題を発生させるために，Kubernetes の Service に構成されている type:NodePort を使用した．全 Pod の CPU 使用量のグラフを，横軸を New Pod が追加されてからの経過時間，縦軸を CPU 使用量 [mcores] として図 7 に示す．図 7 は，2 分 26 秒で New Pod を追加すると，CPU 使用量が他の Pod よりも，上昇するかを実験したグラフである．Pod は，1 リクエストにつき，2 の 6 乗を 1 回計算するプログラムで動いている．監視ツールは Datadog を使用した．また,Locust を使用し，Pod1 から Pod4 に 1 秒間に 500 個の HTTP リクエストを送信した．図 7 では，New Pod の CPU 使用量は 2 分 26 秒から 3 分 11 秒までに約 600 [mcores] 上昇している．研究の結果，New Pod の CPU 使用量が, 他の Pod よりも高いことを課題としていたが，New Pod の CPU 使用量は他の Pod に比べ上昇しないことが分かった．原因は，ロードバランサーが New Pod へ送信するリクエスト数を制限するためである [15]．また，Pod にリクエストを送信しているにも関わらず，全ての Pod の CPU 使用量が 0 [mcores] まで減少している．Pod の CPU 使用量が 0 [mcores] の場合，Pod 自体が停止している可能性がある為，原因を調査する必要がある．

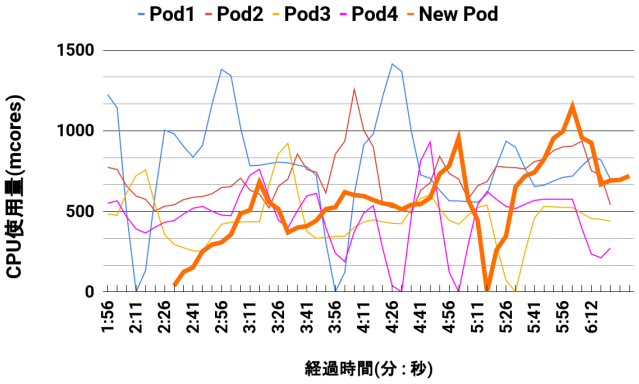

図 7　Pod の CPU 使用量

図 8 は，Locust で New Pod にリクエストを送信した際に，200 番を返すレスポンス数を示している．横軸は，New Pod がレスポンスを取得してからの経過時間を表しており，縦軸は New Pod のレスポンス数を示している．New Pod は，Service のロードバランサーに，プログラムの実行結果を返している．5 分 11 秒の New Pod の CPU 使用量と，レスポンス数を比較すると，CPU 使用量が 0 [mcores] に対して，New Pod はレスポンスの値は，約 75 回を示している．結果，New Pod は停止しておらず，プログラムを実行している．Pod に異常が無い場合，Pod の CPU 使用量の取得方法に原因がある．Pod の CPU 使用量は，Datadog が Linux OS のカーネルにリクエストをして取得している．

リクエストを受けた Linux OS のカーネルは，Pod の CPU 使用量を取得する．しかし，カーネルが値を取得している間は，システム側にレスポンスが割り当てられている．つまり，Linux OS のカーネルが Pod の CPU 使用量を取得している際に、システムレベルのプロセスに CPU が割り当てられる．Datadog よってその間の CPU 使用量を 0 [mcores] と出力している．つまり，Datadog によって，0 [mcores] と出力している部分は，Pod の真の CPU 使用量ではない．その為，図 7 の Pod の CPU 使用量は，0 [mcores] 以外の部分を有効な値とする．

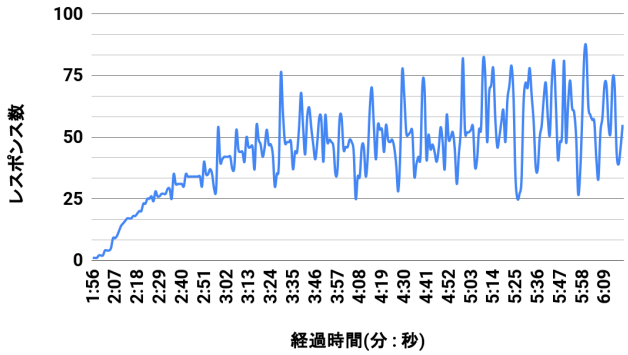

図 8 New Pod のレスポンス数

図 7 のグラフは，Pod1 から Pod4 と，追加した New Pod の CPU 使用量の値に偏りがある．例えば，Pod1 の 4 分 26 秒の部分では，約 1400 [mcores] 上昇しているのに対して，Pod4 は約 500 [mcores] と，二つの Pod で約 900 [mcores] の差が発生している．これは Kubernetes の type:NodePort が均等にバランシングしていないということである．Kubernetes の type:NodePort は Port 番号を指定しない場合，ランダムにリクエストを分散する [16]．ランダムにリクエストを送信した場合、Pod の状況に関係なくリクエストを送信する為，CPU 使用量の偏りが発生すると考える．Kubernetes には Type:LoadBalancer という，GCE や AWS などのクラウドプロバイダー上でアルゴリズムを設定できるロードバランサーがある [17]．しかし，type:LoadBalancer は，クラウドプロバイダーから，ロードバランサーを購入する必要があり，コストの増加につながる [18]．よって，サービスの運営が一時的な場合や，コストを重視している場合は Type:NodePort を使用する．しかし，リクエストに偏りのある負荷分散は，Pod の性能に依存してしまう．Pod の性能が均一で無い場合，負荷の偏りによって負荷上昇に繋がりサービス運営者にとっては問題になる [19].

6. 議論

本研究では，ロードバランサーの送信先にサーバーを追加した時に起こる CPU 負荷上昇の解決案を提案した．しかし，実験を行った際に別の課題が発生した．Kubernetes の type:NodePort で，リクエストの負荷分散を行うと，Pod の CPU 使用量に偏りが発生する課題である．本研究で提案した，CPU 使用量が最も少ない Pod にリクエストを振り分ける方式により，Pod の CPU 使用量に偏りが発生しない負荷分散を実現できる．Kubernetes API により，Pod の CPU 使用量を取得し，値の一番低い Pod に対して，Flask で割り当てる．これにより，図 7 の CPU 使用量のばらつきを抑えることが可能となる．今後は，提案手法のロードバランサーを開発し，Pod の CPU 使用量が type:NodePort を使用した時と比較してどのように変化するかを計測する．

7. おわりに

本研究では，ロードバランサーの動的分散方式に着目し，追加したサーバーにリクエストを送信した時の，CPU 使用量の上昇を課題とした．解決方法として，追加サーバーと同時に仮想のサーバーを追加し，リクエストを分散させるという提案をした．実験方法は，Kubernetes の Pod を用いてサーバーを作成し，負荷試験ツールの Locust で HTTP リクエストを送信している時に，二つの Pod を追加して実験を行う．また，本研究の課題を発生させる為に，Kubernetes の Type:NodePort を使用した．その結果，ロードバランサーが追加したサーバーに送信する負荷を制限していた為，本研究の課題は発生しなかった．しかし，Type:NodePort により負荷分散を行うと，Pod の CPU 使用量に偏りがある課題が発生した．解決方法として，本研究の CPU 使用量の最も低い Pod にリクエストを送信するアルゴリズムを提案した．今後は，Pod の CPU 使用量の最も低い Pod にリクエストを割り振るロードバランサーを開発し，CPU 使用量の計測を行うことで，本研究で生じた課題の解決を行う．

参考文献

[1] Hussain, A. A., Bouachir, O., Al-Turjman, F. and Aloqaily, M.: AI techniques for COVID-19, *IEEE Access*, Vol. 8, pp. 128776–128795 (2020).
[2] Grossman, R. L.: The case for cloud computing, *IT professional*, Vol. 11, No. 2, pp. 23–27 (2009).
[3] Sorenson III, J. C. and Laurence, D. S.: Multipath routing in a distributed load balancer (2018).
[4] Yokota, H., Kimura, S. and Ebihara, Y.: A proposal of DNS-based adaptive load balancing method for mirror server systems and its implementation, *18th International Conference on Advanced Information Networking and Applications, 2004. AINA 2004.*, Vol. 2, IEEE, pp. 208–213 (2004).
[5] Hong, Y. S., No, J. and Kim, S.: DNS-based load balancing in distributed web-server systems, *The Fourth IEEE Workshop on Software Technologies for Future Embedded and Ubiquitous Systems, and the Second International Workshop on Collaborative Computing, Integration, and Assurance (SEUS-WCCIA'06)*, IEEE, pp. 4–pp (2006).

[6] Wei, W., Fan, X., Song, H., Fan, X. and Yang, J.: Imperfect information dynamic stackelberg game based resource allocation using hidden Markov for cloud computing, *IEEE Transactions on Services Computing*, Vol. 11, No. 1, pp. 78–89 (2016).

[7] Mohammed, M. A., Hasan, R. A., Ahmed, M. A., Tapus, N., Shanan, M. A., Khaleel, M. K. and Ali, A. H.: A Focal load balancer based algorithm for task assignment in cloud environment, *2018 10th International Conference on Electronics, Computers and Artificial Intelligence (ECAI)*, IEEE, pp. 1–4 (2018).

[8] Soni, G. and Kalra, M.: A novel approach for load balancing in cloud data center, *2014 IEEE international advance computing conference (IACC)*, IEEE, pp. 807–812 (2014).

[9] Bernstein, D.: Containers and cloud From lxc to docker to kubernetes, *IEEE Cloud Computing*, No. 3, pp. 81–84 (2014).

[10] Nathan, S., Ghosh, R., Mukherjee, T. and Narayanan, K.: Comicon: A co-operative management system for docker container images, *2017 IEEE International Conference on Cloud Engineering (IC2E)*, IEEE, pp. 116–126 (2017).

[11] Mickulicz, N. D., Narasimhan, P. and Gandhi, R.: To auto scale or not to auto scale, *10th International Conference on Autonomic Computing ({ICAC} 13)*, pp. 145–151 (2013).

[12] Schaelicke, L., Wheeler, K. and Freeland, C.: SPANIDS: a scalable network intrusion detection loadbalancer, *Proceedings of the 2nd Conference on Computing Frontiers*, pp. 315–322 (2005).

[13] Takahashi, K., Aida, K., Tanjo, T. and Sun, J.: A portable load balancer for kubernetes cluster, *Proceedings of the International Conference on High Performance Computing in Asia-Pacific Region*, pp. 222–231 (2018).

[14] Domanal, S. G. and Reddy, G. R. M.: Optimal load balancing in cloud computing by efficient utilization of virtual machines, *2014 Sixth International Conference on Communication Systems and Networks (COMSNETS)*, IEEE, pp. 1–4 (2014).

[15] Jiang, H., Iyengar, A., Nahum, E., Segmuller, W., Tantawi, A. N. and Wright, C. P.: Design, implementation, and performance of a load balancer for SIP server clusters, *IEEE/ACM transactions on networking*, Vol. 20, No. 4, pp. 1190–1202 (2012).

[16] Dinesh, S.: Kubernetes NodePort vs LoadBalancer vs Ingress? When should I use what?, *Marzo de* (2018).

[17] Vohra, D.: Kubernetes on Google Cloud Platform, *Kubernetes Management Design Patterns*, Springer, pp. 49–87 (2017).

[18] Dinesh, S.: Kubernetes NodePort vs LoadBalancer vs Ingress? When should I use what?, *Marzo de* (2018).

[19] Syta, E., Jovanovic, P., Kogias, E. K., Gailly, N., Gasser, L., Khoffi, I., Fischer, M. J. and Ford, B.: Scalable bias-resistant distributed randomness, *2017 IEEE Symposium on Security and Privacy (SP)*, Ieee, pp. 444–460 (2017).

