アクセス経路の比較からサーバースケーリングを行う方法

山口 舜[1,a)]　串田 高幸[1]

概要：アクセス集中によりサーバダウンが起こり, サーバが機能しなくなることがある. それを解決するためクラウドのサーバにはオートスケールという機能が存在し, アクセスの負荷に応じてサーバの台数を自動で増減する機能をもつ. このオートスケール機能の課題としてサーバの追加処理の速度が低速で, 予期せぬアクセス集中に対応できないという課題がある. この課題の解決方法としてアクセス経路を参照しサーバースケーリングを行う方法を提案し検証を行う. 検証を行った結果, サーバースケーリングを行うことにより負荷を減らすことを示した.

1. はじめに

背景

近年世界のインターネットを利用するユーザーは増え続けており, 2019 年には世界のインターネットを利用するユーザーの数が 40 億人を超え, 世界の web サイト数が 17 億を超えていることから web サイトの需要が世界的に高いことがわかる. 企業にとって自社の web サイトへの大量のアクセスはうれしいものだが, サーバはアクセスが集中し, リクエストの量が限界を超えると処理が追い付かずサーバーダウンが起こり, サーバが機能しない状態になる. このようなリクエスト限界量を増やすことによりサーバーダウンを防ぐオートスケールという機能がある. オートスケールとはサーバの負荷に応じて自動的にサーバの台数を増減させる機能のことで, web サイトにアクセスが集中したときはサーバを自動で増やし, アクセスが少ないときは自動でサーバを減らすことで常に必要最小限のサーバ数でシステムを安定的に稼働させることができる. それにより無駄なコストをかけずに大量のアクセスにも耐えることが出来るという利点がある [1][2]. またオートスケールを使用することで, ユーザーはパブリッククラウドで広く使用されている動的な負荷に応じ, クラウドリソースの容量をタイムリーにスケーリングできる [3]. このオートスケールという機能を用い, クラウドコンピューティングの特徴である急速な弾力性を活用するためには, 新しく追加されたサーバに対して負荷に応じて自動で分散できる必要がある. このような場合ロードバランサ [4] が用いられ, オートスケールによりアプリケーションが水平に拡張された場合でも負荷に応じて分散することが出来る.

このようなオートスケールやロードバランサを簡単に運用管理と自動化することを目的に設計された kubernetes[5] というオープンソースのシステムがある. この kubernetes の操作対象は Docker[6] で作成されるコンテナ [7] で, Docker に不足しているシステム全体の管理に効果的に力を発揮し, 複数のコンテナの管理と自動化を容易にすることが出来る. 本論文では Docker を用いて簡易的な VM を作成し, kubernetes を用いて管理することにより実験を行っていく.

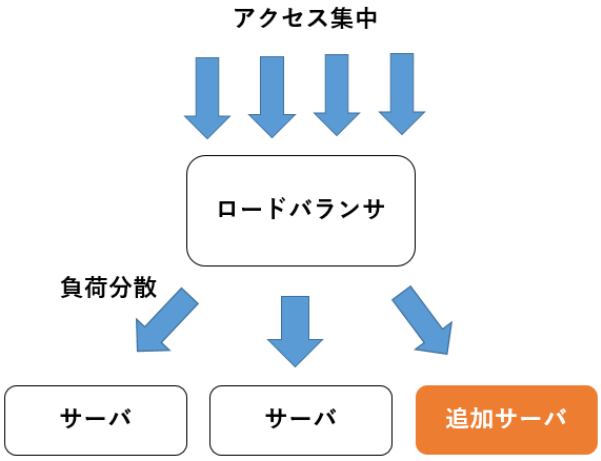

図 1　スケールアウト図

課題

クラウド型サーバは容量がとても大きく, オートスケール機能によりサーバをほぼ無限に作ることができることにより大量のアクセスにも耐えられることが出来るが, オートスケールの課題に予期せぬ急なアクセス集中に対応でき

[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)] C0118272

ないという課題がある [8][9]. この課題の要因としてサーバの追加処理が低速なため, いくらサーバの数を増やせたとしても処理量の限界に到達して処理が低速になってしまう. これらの問題が起こると企業では業務の停滞を招くことにより損失を発生させたり, 対処や再発防止のための説明責任も発生するため, 企業に対して悪影響が大きい.

各章の概要

第 2 章では本論文の関連研究について述べる. 第 3 章では第 1 章で述べた課題を解決するシステムの提案について述べる. 第 4 章では提案するシステムの実装と実験環境について述べる. 第 5 章では提案するシステムの評価と分析について述べる. 第 6 章では提案, 実験, 実験環境, 評価に関する議論について述べる. 第 7 章では本研究の結果から得られた成果について述べる.

2. 関連研究

本論文ではオートスケールの新しい判別方法の提案としているが, もともとの一般的なオートスケールの判別方法はワークロードを用いている. 例として kubernetes で提供されているオートスケールでは kind を HorizontalPodAutoscaler にすることで簡単にオートスケールを行うことができ, target value で設定した CPU 使用率に応じて pod 数が調節される. このオートスケールの判別方法に対する関連研究として, 自動スケーリングの利点は認識されているが, クライアントのワークロードパターンの大幅な変動に直面してリソース使用量を正確に見積もる必要性から生じる複数の課題があるため, 自動スケーリングの可能性を最大限に引き出すことが困難である. この課題を解決する方法として, リソースの自動スケーリングに使用されるワークロード予測のモデル予測アルゴリズムを開発することにより解決し, 運用コストを低く抑えながら, アプリケーションの QoS の両方を満たす方法で, アルゴリズムによってリソースを割り当てて処理できる結果を示す [10]. この研究では課題をワークロードを予測して急なアクセスの対処を行うことにより課題の解決を行っている.

同じようにワークロードのパターンの予測について研究した [9] では, 既存の研究では事前に予測されたワークロードに基づいてリソースを調整することを提案しているが, 実際のワークロードは、履歴に関係なく増大する可能性がある. そのため予測不可能な負荷変動を採用するという課題を解決するため, 予測されたワークロードに基づいてターゲットシステムのスケールを変更する新しい自動スケーリングメカニズムの提案を行うという方法を用いて課題を解決している. この研究では本研究の課題をワークロードの予測から自動スケーリングのメカニズムを提案することからアクセス経路は見ておらず差別化されている.

他に自動スケーリングメカニズムの重要な要素として 2 つ挙げ, 1 つ目は自動スケーリンググループからのリソースを追加または終了するための CPU 閾値の設定で, 2 つ目はワークロードの急増に対処するためにリソースを追加するインスタンスの数の決定を行うスケーリングサイズとした研究 [11] では, CPU 使用率の上限しきい値とスケーリングサイズ係数の設定がクラウドサービスのパフォーマンスに与える影響をシミュレートして調査し, コストと SLO 応答時間の両方を考慮し, 入力負荷に基づいてこれらのパラメーターを調整するための最適化問題を定式化して解決する方法を用いて行った. この研究では本研究の課題を CPU 使用率とスケーリングサイズを定式化して数学的に課題解決を行っており, アクセス経路を見ていないことから差別化されている.

また弾力性を管理するための最も実用的なアプローチとして, アプリケーションへの仮想マシンインスタンスの割り当て/割り当て解除に基づいており, VM レベルの弾力性は、特に需要が急速に変動するアプリケーションの場合, かなりのオーバーヘッドと追加コストの両方を伴うという課題を挙げた研究 [12] では, クラウドアプリケーションの費用効果の高い弾力性を実現するための軽量アプローチを提案することにより課題を解決する. 内容としては VM レベルのスケーリングに加えて, リソースレベル自体（CPU, メモリ, I/O）で, きめ細かいスケーリングを実行を行う.

これら 4 つの関連研究では本論文では見ていない CPU 使用率, メモリ使用率, I/O といった負荷情報とその負荷情報のパターンからの予測と閾値設定による判別方法で解決方法を提案しているため, 負荷のパターン化では急なアクセス集中には対応できないことから差別化されている.

3. 提案

急なアクセス集中が起こる原因としてネット広告, テレビ CM, SNS の宣伝によるアクセスの増加といった様々な要因が考えられる. これらの広告媒体が興味を持たれることにより, それぞれ同じようなアクセス方法でアクセスが増えるので, サーバへアクセスされた UserAgent[13] のブラウザと OS の統計データを取り, 取得したデータの統計比率が変化するタイミングでスケールアウトを行うことによりサーバ追加処理を行うことにより, CPU 使用率やメモリ使用率といったワークロードを計測する必要がなくアクセスが集中する前にアクセス集中を予想しサーバ追加処理を行うことが出来る. また前もってスケールアウトによりサーバを追加することでサーバ一つあたりの負荷を分散することができる.

具体的な提案方法として, 本研究では図 2 のようにユーザーが web ブラウザにアクセスし, リクエストがロードバランサに渡され, 渡されたリクエストをロードバランサがサーバに均等に分散していくことにより負荷分散を行う. その後サーバにリクエストを渡し, アクセスしてき

たユーザーの情報を UserAgent として取得し, 取得した UserAgent をデータベースに送りデータベース内で統計情報を記録し, オートスケール管理サーバがデータベースから UserAgent を受け取り, ブラウザと OS のデータの比率を取り, それぞれのブラウザか OS が大きく変化したタイミングでサーバの追加を行うスケールアウトの処理を行い, リクエストを受けるサーバを増やすことにより負荷を分散する.

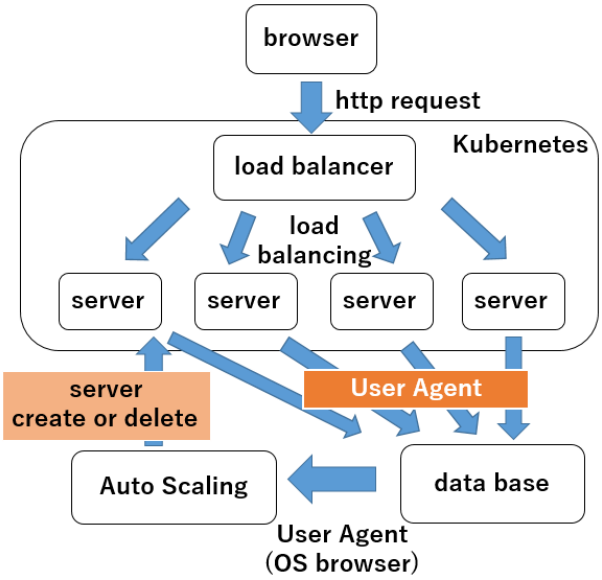

図 2　提案内容の概念図

4. 実装と実験環境

実装の概要図であるシステム構成図を図 3 に示す.

4.1 実装

動的に増加する web サーバを kubernetes を用いて pod を作成した. 各 pod の中に sendlog という名前のコンテナを一つ作成しコンテナのイメージとして, flask[14] という Python の web アプリケーションフレームワークを持ちいて web アプリケーションを作成し, 作成した web アプリケーションを Docker を用いてイメージにし, Docker Hub に上げて用いることにより kubernetes で pod を作成した. sendlog をはじめとする実装に用いられたソフトウェア間の処理の流れを図 4 に示す.

この sendlog コンテナの処理として, ロードバランサで分散され渡されたリクエスト情報を処理し, 結果を返しブラウザに処理結果を出力する. その後コンテナにアクセスしてきたアクセス情報を UserAgent として MariaDB[15] が置かれている VM の dbflask.py というソフトウェアに送る処理を行う. この dbflask.py は MariaDB とデータをやり取りするためのソフトウェアで, MariaDB への書き込みと読み込みの際にこのソフトウェアを介して通信を

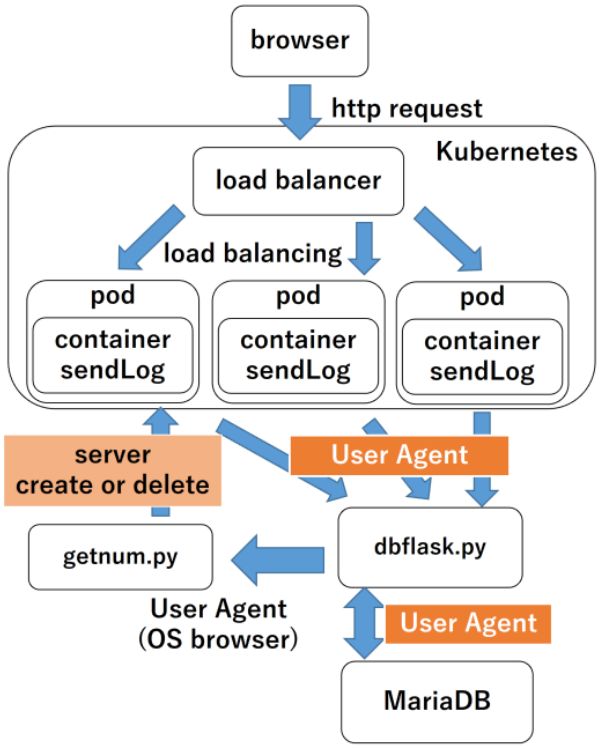

図 3　システム設計図

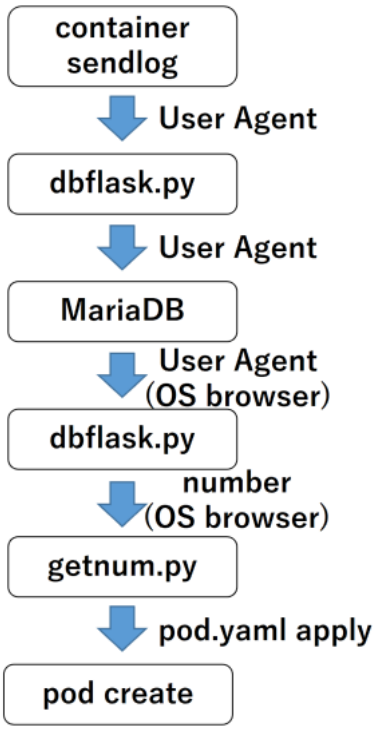

図 4　ソフトウェア間の処理の流れ

行う. MariaDB への書き込みが行われた後 MariaDB から記録した UserAgent のブラウザと OS 情報を読み取る. 読み取る情報として, 通常時のデータベースに保存されている Internet Explorer, Microsoft Edge, Chrome, Firefox, Safari, Windows, macOS, Linux, iPhone, Android の数を

取り出し, 最新 100 件の中に含まれる同じデータの数を getnum.py というソフトウェアに送る.

次にオートスケールを管理するソフトウェアである getnum.py について説明する. getnum.py も sendlog コンテナと同じように flask という Python の web アプリケーションフレームワークを持ちいて作成した. getnum.py の処理として, dbflask.py から送られてきたデータを通常時の件数と最新の件数で比較を取り, 10 件以上離れていなければ何も処理をしないが, 10 件以上離れている場合に新しく pod.yaml ファイルを作成し, 作成した pod.yaml を create することにより pod を新しく作成する. 作成された pod は kubernetes のロードバランサを用いて分散されている web サーバの一つとなり処理を行う.

4.2 実験環境

実験環境として, Ubuntu18.04.2LTS の OS を使用した VM を 2 つ用いる. 一つの VM には Docker と kubernetes を使用する環境を持ち, もう一つの VM にはリレーショナルデータベースである MariaDB を用いる. それぞれのソフトウェアは Python3 で記述し, Flask を用いてソフトウェアを動作させている. pod に負荷をかける方法として locust を用いて負荷をかけ, Zabbix を用いて pod にかかる負荷の変化を計測する.

実験はオートスケールにより増える sendlog コンテナを含む pod の数を 1 個から始め, locust で負荷をかけながら様々な OS とブラウザからランダムにアクセスをかけ, pod にかかる負荷がどのように変化したのかを計測する.

評価実験の構成図を図 5 に示す.

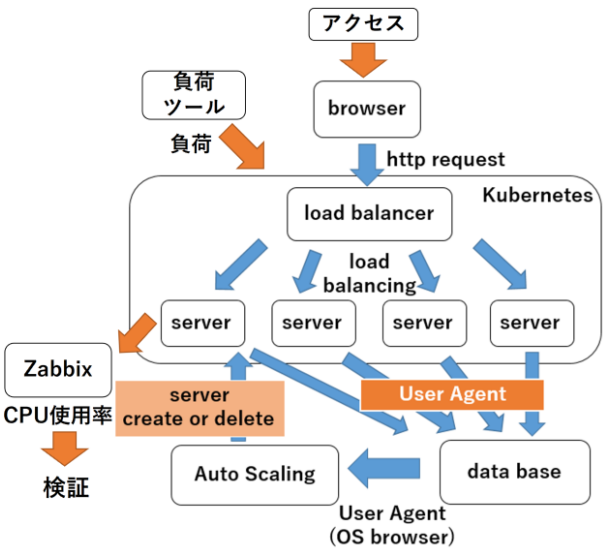

図 5 評価実験の構成図

5. 評価と分析

実験の内容は locust を用いて 180 秒間ロードバランサに負荷をかけることによる pod の CPU 使用量の変化の様子を計測する. locust では 100 の同時接続ユーザ数を上限とするユーザアクセスを初期値 10 から 1 秒ごとに 10 ずつ上がるように設定する. 設定した負荷をかけるユーザアクセスの変化の様子を図 6 に示す.

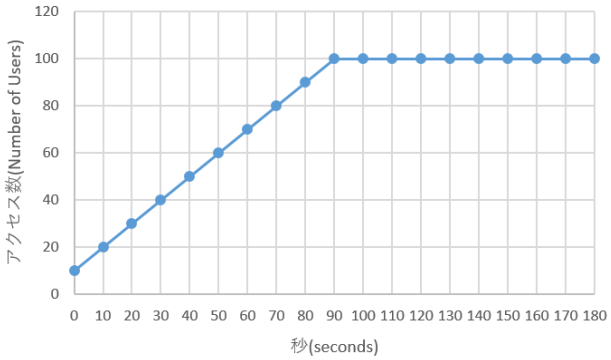

図 6 ユーザアクセスの変化

ユーザアクセスすることによりをロードバランサに負荷をかけ, ロードバランサの割り振り先である sendlog コンテナを持つ pod に負荷をかけ pod の数の変化と負荷の変化を計測する. pod の数を 1 個から始め, locust のアクセスとは別にランダムな UserAgent を持つアクセスをすることにより pod が 20 秒の時と 90 秒の時にオートスケール機能により pod が追加された. この実験で 10 秒ごとに CPU 使用量を計測し, CPU 使用量の変化の結果を図 7 に示す.

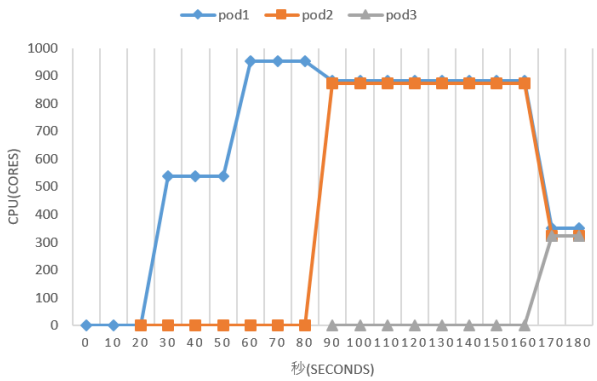

図 7 CPU にかかる負荷の変化

この結果から pod が活動可能な状態になると pod がもともと処理していた仕事の量が分散され, pod 一つあたりの負荷を減らせていることがわかる. また pod が作成されてから活動可能な状態になるまで 60 秒から 70 秒かかることから本論文の課題であるサーバの追加処理が低速という課題を確認することができ, CPU の負荷を参考にしてオートスケールしていないことから, CPU に対する負荷が高くなってからサーバを追加し始めるということが起こっていない. よってアクセスが集中し負荷が高くなる前から pod

を追加することが出来ることにより, サーバの追加処理が低速という問題を事前に行うことにより解決している.

また同じ環境で行った 1 秒間に処理されたリクエスト数を 10 秒ごとに記録したグラフを図 8 に示す. 図 8 から pod が追加された 20 秒と 90 秒の時 1 秒当たりのリクエスト処理数が約 40 上がったことから, pod が追加されたことによるリクエスト処理効率が上がったことがわかる.

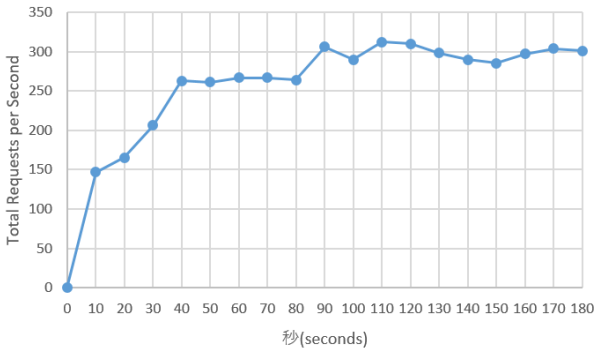

図 8 1 秒間に処理されたリクエスト数の変化

またレスポンス時間を計測したところ pod が追加されても約 180ms から変化がなくレスポンスタイムには変化が見られなかった. これには pod のメモリ上限を設定していないことが考えられ, この pod を立てている VM のメモリに処理能力が依存しているからだと考えられる.

6. 議論

本研究ではオートスケールは通常 CPU やメモリの使用率を確認し, その負荷の量によりサーバを増やしたり減らしたりするスケールアウトやスケールインといった処理を行うが, ほかの方法の提案としてまだ誰も行っていない UserAgent を参考にし, UserAgent の傾向からも分析することにより負荷だけでは CPU 使用率が高くなってから始めるスケールアウトの処理を前もってできると考え取り組んできた. しかし, 当初やろうとしていたことは負荷と UserAgent の両方からスケーリングの判断しようとしており, 負荷の面からはサーバの負荷情報を計測し, CPU 使用率が 70%を上回ったときスケールアウトを行い, CPU 使用率が 40%を下回り, 尚且つ減少傾向にある場合スケールインを行うという方法を考え, UserAgent の面からはサーバへアクセスされた UserAgent の OS とブラウザの統計データを取り, OS とブラウザの統計比率が変化するタイミングでアクセスが集中, または減少すると考えたため, 変化のタイミングで負荷が上昇傾向にあればスケールアウトを行い, 負荷が減少傾向にあればスケールインを行うことによりサーバ追加処理をアクセス集中前に行い, サーバの数を少なくしても処理に影響がなければサーバを削除するという方法を考えていた. だがこの方法では負荷の面では既存の kubernetes でも簡単に実装できるような内容であり, オートスケールを考えるうえで大切な閾値の設定を考えるための実際に計測できる現場の環境がなく, 自分で負荷をかけたところで実際に実装してどうさせた場合と違い, 自分の考えている範囲内のことしか起きないことから閾値の設定がマジックナンバーになってしまうという欠点があり, 関連研究で紹介した研究のように, 実際に企業に使ってもらいデータを取らなければより効果的で信頼性のある閾値設定を行うことが出来なかった. また, 参照データがなかったため閾値設定を数学的に考えることが出来ず実装することが出来なかった. また UserAgent の面ではそもそも提案内容は「急なアクセス集中が起こる原因としてネット広告, テレビ CM, SNS の宣伝によるアクセスの増加といった様々な要因が考えられる. これらの広告媒体が興味を持たれることにより, それぞれ同じようなアクセス方法でアクセスが増えるので, サーバへアクセスされた UserAgent のブラウザと OS の統計データを取り, 取得したデータの統計比率が変化するタイミングでスケールアウトを行うことによりサーバ追加処理を行い, サーバ一つあたりの負荷を分散することができる. 」としているが, この提案内容はアクセス集中が起こるとユーザの偏りが出来ることが前提としているが, このことを裏付けるデータはなく, 同じような研究を行っている資料も見つからなかったことから提案内容の根拠が不十分である. これらの方法の改善案として, 企業で実際に統計を取り, 実際にアクセスが増える時どのようなことが起こり, ユーザに偏りが出来ているかというデータを取らなければ信頼性がないことがわかる. また UserAgent だけで判断するだけでは関連研究でも触れられている通り, 負荷を減らし, コストを減らすことが最適なスケーリングとなるため, 負荷を減らすためには負荷情報を参照しなければ無駄にスケールアウトを行ってしまい, コストを余計にかけてしまう恐れがあるため, 負荷と UserAgent の両面からスケーリングの判断が行えるようになればより効率のいいスケーリングが行えることがわかる.

本研究の他の提案内容として SNS やチケット, 新商品の発売日や通常時の時間と曜日ごとの情報をもとにスケーリングするスケジューリングを組むという方法が考えられる. 本研究の課題である急なアクセス集中に対応するためには, アクセスが集中する前にアクセスが集中する時間がわかればサーバを事前に増やして対応することが出来る. この解決策として今回は UserAgent を取得しアクセスログを集計することにより解決しようと考えたが, ほかの案として通常時の負荷のの上下を記録しその情報をもとにスケーリングを行う, それに加えアクセスが集中する要因として新商品の販売やチケットの予約が考えられるの. このアクセス集中する前兆にアクセスするユーザは SNS でつぶやくことが考えられるため, SNS の関連情報がどれだけつぶやかれているかを参考にし, このアクセス集中する規模を特定

することが出来ればどの程度のサーバを前もって立てて置けばよいのかがわかり, アクセス集中に対応することが出来る.

7. おわりに

この研究の課題としてサーバ追加処理が低速なため急なアクセス集中に対応できないという課題に対し, アクセス集中が起こる前にアクセス集中を検知することができるようにするため UserAgent を用いてアクセスログを記録し, UserAgent を参考にしてスケーリングを行う方法の提案を行った. 提案方式は通常時と最新の UserAgent を比べ, 大きな変化を記録したときアクセス集中が起こる前兆としてスケールアウトを行うという方法を取り, サーバ 1 つあたりにかかる負荷を減少させた. 本研究では新たなスケーリング方法の提案として UserAgent を用いた方法の提案を行った. このシステムは一つのブラウザや OS からのアクセスに対応しているクラウド型サーバ以外のすべてのオートスケール機能で用いることが出来る. また今後の課題として本論文で提案した方法はアクセス集中を予測してスケールアウトを行うシステムでありコスト面で効率的ではないため, 効率的なコストを実現するために SNS の情報からも参照し, より確信的なタイミングでスケーリングを行う機能が必要になる.

参考文献

[1] Lorido-Botran, T., Miguel-Alonso, J. and Lozano, J. A.: A review of auto-scaling techniques for elastic applications in cloud environments, *Journal of grid computing*, Vol. 12, No. 4, pp. 559–592 (2014).

[2] Mao, M. and Humphrey, M.: Auto-scaling to minimize cost and meet application deadlines in cloud workflows, *SC '11: Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 1–12 (2011).

[3] Bao, J., Lu, Z., Wu, J., Zhang, S. and Zhong, Y.: Implementing a novel load-aware auto scale scheme for private cloud resource management platform, *2014 IEEE Network Operations and Management Symposium (NOMS)*, pp. 1–4 (online), DOI: 10.1109/NOMS.2014.6838340 (2014).

[4] Heinzl, S. and Metz, C.: Toward a Cloud-Ready Dynamic Load Balancer Based on the Apache Web Server, *2013 Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises*, pp. 342–345 (online), DOI: 10.1109/WETICE.2013.63 (2013).

[5] Medel, V., Rana, O., Bañares, J. Á. and Arronategui, U.: Modelling performance & resource management in kubernetes, *Proceedings of the 9th International Conference on Utility and Cloud Computing*, pp. 257–262 (2016).

[6] Li, Y. and Xia, Y.: Auto-scaling web applications in hybrid cloud based on docker, *2016 5th International Conference on Computer Science and Network Technology (ICCSNT)*, pp. 75–79 (online), DOI: 10.1109/ICCSNT.2016.8070122 (2016).

[7] Abdullah, M., Iqbal, W. and Bukhari, F.: Containers vs virtual machines for auto-scaling multi-tier applications under dynamically increasing workloads, *International Conference on Intelligent Technologies and Applications*, Springer, pp. 153–167 (2018).

[8] Singh, P., Gupta, P., Jyoti, K. and Nayyar, A.: Research on auto-scaling of web applications in cloud: survey, trends and future directions, *Scalable Computing: Practice and Experience*, Vol. 20, No. 2, pp. 399–432 (2019).

[9] Hirashima, Y., Yamasaki, K. and Nagura, M.: Proactive-Reactive Auto-Scaling Mechanism for Unpredictable Load Change, *2016 5th IIAI International Congress on Advanced Applied Informatics (IIAI-AAI)*, pp. 861–866 (online), DOI: 10.1109/IIAI-AAI.2016.180 (2016).

[10] Roy, N., Dubey, A. and Gokhale, A.: Efficient Autoscaling in the Cloud Using Predictive Models for Workload Forecasting, *2011 IEEE 4th International Conference on Cloud Computing*, pp. 500–507 (online), DOI: 10.1109/CLOUD.2011.42 (2011).

[11] Al-Haidari, F., Sqalli, M. and Salah, K.: Impact of CPU Utilization Thresholds and Scaling Size on Autoscaling Cloud Resources, *2013 IEEE 5th International Conference on Cloud Computing Technology and Science*, Vol. 2, pp. 256–261 (online), DOI: 10.1109/CloudCom.2013.142 (2013).

[12] Han, R., Guo, L., Ghanem, M. M. and Guo, Y.: Lightweight Resource Scaling for Cloud Applications, *2012 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (ccgrid 2012)*, pp. 644–651 (online), DOI: 10.1109/CCGrid.2012.52 (2012).

[13] Pham, K., Santos, A. and Freire, J.: Understanding website behavior based on user agent, *Proceedings of the 39th International ACM SIGIR conference on Research and Development in Information Retrieval*, pp. 1053–1056 (2016).

[14] Vogel, P., Klooster, T., Andrikopoulos, V. and Lungu, M.: A low-effort analytics platform for visualizing evolving Flask-based Python web services, *2017 IEEE Working Conference on Software Visualization (VISSOFT)*, IEEE, pp. 109–113 (2017).

[15] Tongkaw, S. and Tongkaw, A.: A comparison of database performance of MariaDB and MySQL with OLTP workload, *2016 IEEE conference on open systems (ICOS)*, IEEE, pp. 117–119 (2016).