Kubernetes Podの度数分布を用いた動的再配置によるレスポンスタイムの向上とノードCPU使用率の効率化

伊藤 佳城[1]　串田 高幸[1]

概要：Kubernetes は Pod を新規に作成した際にスケジューリングを行うことで割り当てるノードを決定する．しかし，K8s のノードの CPU 使用率はデプロイされている Pod へのアクセス数それに伴う Pod の処理の増加により常に変化する．そのため，Pod をデプロイする際の 1 度だけスケジューリングの決定の状況とは変化するためその状況が考慮されていない．そこで本研究の提案として Pod ごとの CPU 使用率の度数分布から相対度数を算出し，最も高い割合から最も低い割合の CPU 使用率の変動の幅を求める．その差分の大きい順に各 Pod を重み付けし，この重みを影響度と定義する．影響度の合計をノード数で割ることで平均値を求めその平均値をもとに Pod の再配置をすることでとノードリソース効率化を実現する．

1. はじめに

背景

Kubernetes(K8s) とは Google が開発した OSS のコンテナオーケストレーションシステムである [1]．K8s の管理単位として Pod, ノード，クラスタがある．Pod は Kubernetes がコンテナを実行・管理するための最小単位，1 つ以上のアプリケーションコンテナを内包しているものである．

Pod は新規に作成されたときにノードに割り当てられる．そのことをスケジューリングと呼びその際に kube-scheduler という Kubernetes(K8s) のコンポーネントの 1 つであるスケジューラが使われる [2]．K8s の kube-scheduler はフィルタリングとスコアリングを使いノードのスコアを出すことで Pod のスケジューリングを行っている [1]．しかし，K8s クラスタは常に動的に変更するため Pod に割り当てられたノードの状況は変化する．そのためスケジューリングされた Pod を別ノードに移動することが望ましい状況がある．例として下記のような例が挙げられる．

- 一部のノードが CPU をクラスタ内の他ノードと比べ明らかに CPU を過小または過大に使用している場合
- ノードに障害が発生し Pod が他ノードに移動した後にノードが復旧しても Pod は移動先のノードに常駐する場合
- 新しいノードが K8s クラスタに追加された場合

上記の例はいずれもノード上の Pod が偏っている状況にもかかわらず，一度ノードに割り当てられた Pod は割り当て後の移動をしない．

デフォルトの kube-scheduler ではコンテナをノード全体に均等に分散することで管理するクラスタ内のすべてのノード（物理サーバー）の負荷分散をする．これによりノードが過度にリソースに制約される可能性を減らすことが可能となる．しかし，クラスタ内の基盤となる物理ハードウェアは，CPU, ディスク, メモリ，ネットワークからも影響を受ける [3]．しかし，kube-scheduler はデプロイ時のノード状況を見て配置を行う．そのため配置後の Pod の CPU 使用率におけるノードの影響を考えてはいない．

課題

kube-scheduler は Pod をデプロイする際の 1 度だけスケジューリングを決定するが K8s のノードの CPU 使用率は外部のトラフィックや同一ノードの Pod の処理により常に変化する．そのため kube-scheduler が決めた段階の状況とは変化するためその状況が考慮されていない．図 1 のように一時的に検索サービスにアクセスが増加した場合，同一ノード上のサービスは強制的に使用できる CPU が制限されてしまう．そのため，マルチテナントのように複数サービスが動いている状況では，一時的な特定の Pod へのアクセス増加が起きる．その場合，その Pod の使用する CPU は増加する．そうすると同一ノードにある Pod は強制的にその Pod が使う使用率を以外の空き容量で動かすため使用したい CPU 使用率が無理やり制限されてしまう．実際に Pod で設定した requests(要求量) よりも競合と干渉により CPU 使用率が低下する問題が報告されている [4].

[1] 東京工科大学大学院バイオ情報メディア研究科
コンピュータサイエンス専攻
〒 192–0982 東京都八王子市片倉町 1404-1

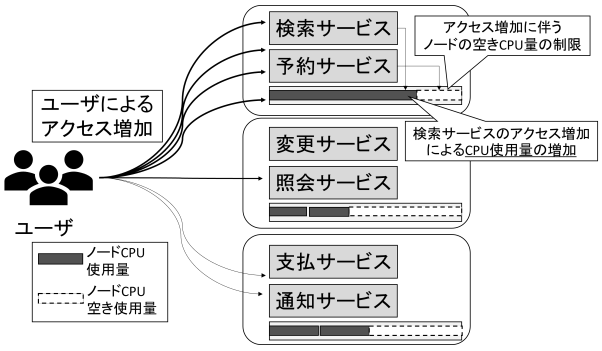

図 1 アクセス数増加によるノードにおける CPU 使用率の偏り

基礎実験

基礎実験として K8s のノードに WordPress が動いている Pod をデプロイする．その Pod があるノードで Stress-ng コマンドを実行し，ノードの CPU 使用率を 0-100%の範囲で 10%ずつ上昇させ負荷をかける．さらに Pod に HTTP リクエストを送信し，その時の応答平均時間の測定をする．送るリクエスト数は，Pod が Failures を出さずに処理できる上限である 150 (req/s) を固定して送信する．その結果を図 2 に示す．基礎実験の結果としてノードにかかる

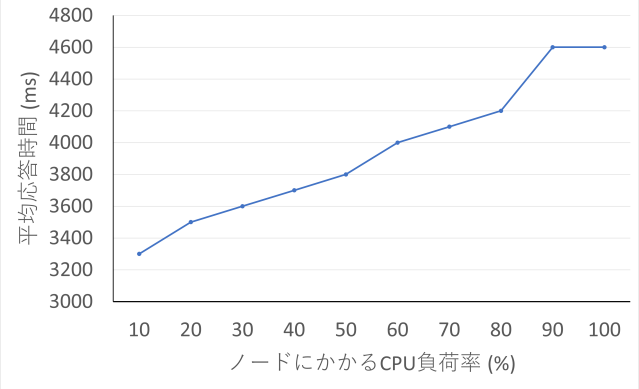

図 2 ノードにかかる CPU 負荷率変化による Pod の平均応答時間の結果

CPU 負荷率が上昇するとそれに伴いノード上の Pod の応答時間が遅い結果となった．ノードに対する CPU 負荷率が 10%の時，平均応答時間は 3300 (ms), 100%で Pod の平均応答時間は 4600 (ms) となり，ノードの CPU 負荷率によって 1300 (ms) も応答時間が遅くなった．以上のことからノードの CPU 負荷率によって Pod が使用できる CPU 使用率が強制的に制限され，応答時間に影響が出たことがわかる．

各章の概要

このテクニカルレポートは, 次のように構成される．1 章では, 本研究の背景・課題．2 章は, 関連研究の説明．3 章では, 本研究の提案について述べ.4 章で実装とその実験環境,5 章で評価と分析を行い, 6 章で今後の課題や議論・考察をする．最後に 7 章で, 最終的なまとめとする．

2. 関連研究

この研究では, K8s 上の Pod で実行中のジョブに ProCon という名前の進行状況ベースのコンテナ配置スキームを提案している [5]．実行中のジョブの進行状況を監視することにより Worker での即時の CPU 使用率も考慮されており，将来のリソース使用率の見積もりもしている．しかし，Pod 内のジョブは短い時間のものに限定されており再配置ではなくスケジューリングに Pod のジョブの進行状況が追加されている．

デフォルトの Kubernetes スケジューラ実装に匹敵するスケジューラパフォーマンスを備えたマイクロサービスベースのスケジューラフレームワークである Epsilon を提案 [6]. デフォルトの scheduler をマイクロサービスベースで構築された研究であるが配置後の Pod のパフォーマンスについては検証されていない．

トポロジーベースの GPU スケジューリングフレームワークを提案 [7]．この研究では K8s の GPU リソースの活用に焦点を当てている．しかし，デフォルトの scheduler に GPU のリソース活用を組み込んだものであり再配置については考察されていない．

K8s のスケジューリングに関する研究では，デフォルトの kube-scheduler に機能を追加している．しかし，スケジューリング後の Pod の実際の CPU 使用率を取得してその変化量を考慮して適応させていない．

3. 提案方式

提案方式

本提案では，Pod のデプロイ時から現在までの CPU 使用率の確立度数分布からスケジューリング後の Pod に影響度という独自の重みで再配置をする．そして応答時間向上とノードの CPU 使用率の効率化を目的とする．前提条件として kube-scheduler にてスケジューリングがすでに行われている Pod を対象とする．前提条件として，スケジューリングされてから動いている最新の期間で各 Pod の CPU 使用率を取得をする．その取得内容から度数分布が求めることができる状態であることとする．

再配置決定における影響度算出のための手順を下記に示す．

(1) Pod の CPU 使用率から度数分布の算出
(2) 度数分布から Pod ごとの CPU 使用率の相対度数の算出
(3) 相対度数から各 Pod の影響度の算出

影響度を出すために各 Pod における度数分布を算出する．その際に用いる要素を下記に示す．

- ノード X における Pod A の起動時から現在までの CPU 使用率の度数分布 XA_C

度数分布から相対度数を算出し，最も高い割合から最も

低い割合の差分を求める．CPU 使用率の度数分布からの差分は下記にように求める．

$$|XA_{C_{Max}} - XA_{C_{Min}}|$$

クラスタ内の Pod 総数から差分の大きい順に各要素を重み付けする．クラスタ内の Pod が 6 個であれば求めた度数分布の要素ごとに $1\tilde{6}$ までの重みをつける．この重みを影響度と定義する．Pod 数を P_N とすると影響度の範囲は

$$P_N \geq x \geq P_N$$

となる．

Pod を影響度の少ない順にノードに配置をした場合を計算する．影響度の合計をノード数で割ることで平均値を求めその平均値をもとに配置をする．図 3 は，6 つのサービスが動いており影響度算出し，再配置を行った例である．前提条件として 1 サービス 1Pod であるとする．今回の例の場合 $1\tilde{6}$ までの重みがついておりその合計数は，21 となる．それをノード数の 3 で割ると影響度の平均は 7 となり 1 ノードにおける影響度合計が 7 となるように再配置を行う．影響度の算出までの期間は 1 時間ごとに行い，再配置も同様のタイミングで行う．

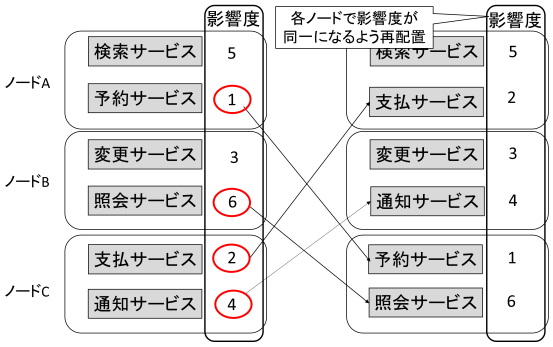

図 3　影響度算出後の再配置

ユースケース・シナリオ

ユースケースとして，41 個のマイクロサービスを含むマイクロサービスアーキテクチャに基づく列車のチケット予約システムを利用する運用する状況をユースケースとする [8][9]．このサービスは，チケットの照会, 予約, 支払い, 変更, ユーザー通知の一般的な列車チケット予約機能提供している．この列車チケット予約システムを K8s 上で運用する際には，サービスごとに Pod として配置をすることを前提とする．

ユーザが列車チケット予約システムの検索サービスにアクセスが集中して使用した際にアクセス増加した場合，同一ノードにある別のサービスはその影響を受け CPU 使用率が低下する．その場合すでにデプロイされているノード上の Pod を配置を変更する必要があり，その際にどのノードからどのノードへ移動するかは K8s を運用するユーザがクラスタ内のノードごとの CPU 使用率や Pod 数, Pod 単体の CPU 使用率を確認してどの Pod を再配置の対象とするかを決定する．

本提案方式によりユーザは管理対象が増加したとしても再配置の適応を自動化することが可能となる．

4. 実装と実験方法

実装

本研究の提案における再配置決定プログラムのプログラムコンポーネントは下記の 3 つからなる．

(1) Pod の CPU 使用率の取得
(2) 度数分布と影響度の算出
(3) 再配置の閾値決定と適応

実装における構成を図 4 に示す．

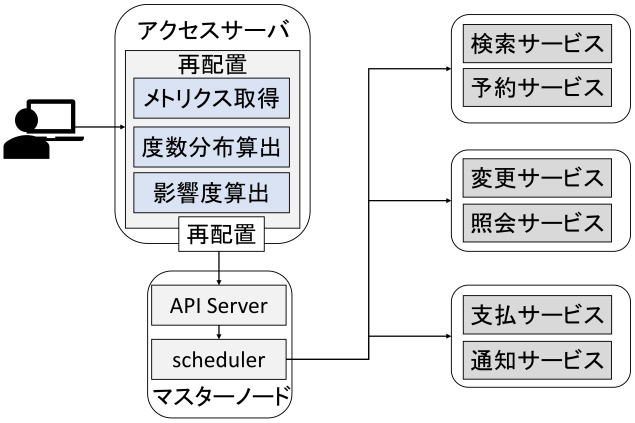

図 4　実装構成図

再配置決定プログラムはアクセスサーバにて構築及び起動する．Pod の CPU 使用率の取得は，Prometheus Operator を使用する．そこから Python3 による prometheus-api-client を用いてプログラムによる取得をする．提案手法に基づいて取得した CPU 使用率から度数分布を求め各 Pod ごとに影響度を算出する．再配置における移動はノードに事前にラベル付けを行い対応する Pod にラベルを対応させることで移動させる．

実験環境

実装環境は Ubuntu20.04 の VM3 台で構築した MicroK8s クラスタ (アクセスサーバ 1 台, マスターノード 1 台, ワーカーノード 2 台) を使用する．下記に VM の基本構成を示す．

- vCPU x 4
- RAM 4GB
- HDD 40GB
- MicroK8s ver.1.21

再配置対象とするシステムは，ユースケースと同じく

41 個のマイクロサービスを含むマイクロサービスアーキテクチャに基づく列車のチケット予約システムを利用する [8][9]．このサービスは実際にマイクロサービス開発の経験があるエンジニアによって作成されているため実際のサービス運用に近い状況を再現するために利用する．

5. 評価手法と分析手法

再配置対象とした 41 個のマイクロサービスに対して K8s のデフォルトの kube-scheduler で配置をした場合と再配置プログラムを適応させた場合との比較をする．どちらも共通の条件としてノード数とノードの基本構成は実験環境と同じとする．評価指標はノード上の Pod に HTTP リクエストを送信した際の応答時間と K8s ノード別の CPU 使用率が効率化できているかを評価する．また送信するトランザクションのパターンとして，チケットの購入・予約，列車チケット検索の処理の 3 つのパターンを送信する．送信数は，送信時に Failures による処理の失敗が起こるまでとする．評価項目としてトランザクション別のノード CPU 使用率と Pod 配置状況の変化をレスポンスタイムの測定から評価をする．

6. 議論

本研究の提案手法では，Pod ごとの CPU 使用率を取得し，Pod が起こり得るノードにおける影響度を用いて再配置を行うタイミングとして Pod が新たにデプロイされたときに Kube-relocation の実行及び適応を行った．しかし，K8s クラスタは Pod に対する外部アクセスや Pod で起動させるアプリケーションの内部処理の状況に応じてその CPU 使用率は動的に変化する．そして Pod の移動させるとその瞬間のサービスは停止してしまう．そのため再配置を行うタイミングはできるだけ Pod の稼働状況が引い時間帯を過去の度数分布の割合で CPU 使用率が相対的に低い状態のタイミングでの再配置を行うことでサービスの停止の影響を極力下げられると考える．しかし，再配置における無停止の再現も本研究の今後の課題となる．

7. おわりに

本提案では，Pod のデプロイ時から現在までの CPU 使用率の確立度数分布を算出し，スケジューリング後の Pod に影響度という独自の重みで再配置をする提案をした．これにより K8s のクラスタの CPU 使用における効率化に貢献することが可能となる．

参考文献

[1] El Haj Ahmed, G., Gil-Castiñeira, F. and Costa-Montenegro, E.: KubCG: A dynamic Kubernetes scheduler for heterogeneous clusters, *Software: Practice and Experience*, Vol. 51, No. 2, pp. 213–234 (online), DOI: https://doi.org/10.1002/spe.2898 (2021).

[2] Shah, J. and Dubaria, D.: Building Modern Clouds: Using Docker, Kubernetes amp; Google Cloud Platform, *2019 IEEE 9th Annual Computing and Communication Workshop and Conference (CCWC)*, pp. 0184–0189 (online), DOI: 10.1109/CCWC.2019.8666479 (2019).

[3] Townend, P., Clement, S., Burdett, D., Yang, R., Shaw, J., Slater, B. and Xu, J.: Invited Paper: Improving Data Center Efficiency Through Holistic Scheduling In Kubernetes, *2019 IEEE International Conference on Service-Oriented System Engineering (SOSE)*, pp. 156–15610 (online), DOI: 10.1109/SOSE.2019.00030 (2019).

[4] Liu, L., Wang, H., Wang, A., Xiao, M., Cheng, Y. and Chen, S.: *Mind the Gap: Broken Promises of CPU Reservations in Containerized Multi-Tenant Clouds*, p. 243–257 (online), available from ⟨https://doi.org/10.1145/3472883.3486997⟩, Association for Computing Machinery (2021).

[5] Fu, Y., Zhang, S., Terrero, J., Mao, Y., Liu, G., Li, S. and Tao, D.: Progress-based Container Scheduling for Short-lived Applications in a Kubernetes Cluster, *2019 IEEE International Conference on Big Data (Big Data)*, pp. 278–287 (online), DOI: 10.1109/BigData47090.2019.9006427 (2019).

[6] Jing Hui, A. N. and Lee, B. S.: Epsilon: A Microservices based distributed scheduler for Kubernetes Cluster, *2021 18th International Joint Conference on Computer Science and Software Engineering (JCSSE)*, pp. 1–6 (online), DOI: 10.1109/JCSSE53117.2021.9493827 (2021).

[7] Song, S., Deng, L., Gong, J. and Luo, H.: Gaia Scheduler: A Kubernetes-Based Scheduler Framework, *2018 IEEE Intl Conf on Parallel Distributed Processing with Applications, Ubiquitous Computing Communications, Big Data Cloud Computing, Social Computing Networking, Sustainable Computing Communications (ISPA/IUCC/BDCloud/SocialCom/SustainCom)*, pp. 252–259 (online), DOI: 10.1109/BDCloud.2018.00048 (2018).

[8] Zhou, X., Peng, X., Xie, T., Sun, J., Ji, C., Li, W. and Ding, D.: Fault Analysis and Debugging of Microservice Systems: Industrial Survey, Benchmark System, and Empirical Study, *IEEE Transactions on Software Engineering*, Vol. 47, No. 2, pp. 243–260 (online), DOI: 10.1109/TSE.2018.2887384 (2021).

[9] Zhou, X., Peng, X., Xie, T., Sun, J., Ji, C., Liu, D., Xiang, Q. and He, C.: Latent Error Prediction and Fault Localization for Microservice Applications by Learning from System Trace Logs, ESEC/FSE 2019, New York, NY, USA, Association for Computing Machinery, p. 683–694 (online), DOI: 10.1145/3338906.3338961 (2019).