Vol.2018-CSEC-81 No.13
Vol.2018-IOT-41 No.13
2018/5/18

HTTP リクエスト単位でコンテナを再配置する仮想化基盤の高速なスケジューリング手法

松本 亮介[1,a)]　田中 諒介[2]　栗林 健太郎[1]

概要：クラウドサービスの普及に伴い，個人の Web サイトでもクラウドサービスに類する機能を利用して，突発的なアクセス集中に耐性があり，かつ，利用したリソース使用量に応じて課金するサービスの提供が求められている．我々はその要求に応じるために，Web サイトをコンテナ上に構築し，コンテナの起動や停止，複製やリソース割り当てといった状態の変更を素早く実行できるコンテナ管理アーキテクチャ FastContainer を提案した．一方で，単一のコンテナが特定のサーバに収容されている状態で，サーバが過負荷に陥ったり停止したりするような状況においては，障害時にコンテナの収容サーバ情報の変更を手動で構成管理データベースに反映させる必要があった．本研究では，HTTP リクエスト処理時において，収容サーバ，および，その経路までの状態に応じて，自動的にコンテナを収容するサーバを決定し，サービスを継続させる，HTTP リクエスト単位でのコンテナスケジューリング手法を提案する．提案手法では，FastContainer の状態変化の高速性に基づいて，コンテナが頻繁に収容サーバを変更されてもサービスに影響がないことを利用する．それによって，プロキシサーバから収容サーバに 1 個の ICMP パケットで応答速度を計測し，少ないパケット数と短いタイムアウトで収容サーバの反応時間を計測できる．そのことで，HTTP リクエスト単位でのコンテナスケジューリングを実現する．

Fast Scheduler for a Cloud Platform to Relocate Containers at Each HTTP Request

Ryosuke Matsumoto[1,a)]　Ryosuke Tanaka[2]　Kentaro Kuribayashi[1]

1. はじめに

インターネットを活用した企業や個人の働き方の多様化に伴い，インターネット上で自らを表現する機会が増加している．特に個人にとっては，Twitter や Facebook を活用して，自身が作成したコンテンツを拡散させることにより，効率よくコンテンツへの訪問数を増やすことができるようになった．その結果，コンテンツの内容の品質が高ければ，さらに拡散され，コンテンツに紐づく個人のブランド化も可能になってきている．そのため，コンテンツ拡散時であっても継続的に配信可能な Web サイトの構築は，個人用途であっても非常に重要になってきている．

Web コンテンツを継続的に配信するためには，オートスケーリングのような負荷対策と，冗長構成による可用性の担保が挙げられる．一般的に，個人が Web コンテンツを配信するためには，Web ホスティングサービスやクラウドサービスなどが利用される [8]．特に，個人の利用を想定した場合，単一のサーバに高集積にユーザ領域を収容するような低価格 Web ホスティングサービスが利用される．そのような低価格ホスティングサービスでは，利用者の Web コンテンツが特定の Web サーバに紐づくため，ユーザ領域単位で突発的なアクセス集中に対して負荷分散することが難しい．クラウドサービスの場合は，利用者がアクセス集中に耐えうるオートスケーリング [9] の仕組みを作る必要があったり，各サービス・プロバイダが提供しているオートスケーリングの機能 [2] を使う必要がある．

[1] GMO ペパボ株式会社 ペパボ研究所
Pepabo Research and Development Institute, GMO Pepabo, Inc., Tenjin, Chuo ku, Fukuoka 810-0001 Japan
[2] GMO ペパボ株式会社 ホスティング事業部
Hosting Department, GMO Pepabo, Inc.
[a)] matsumotory@pepabo.com

可用性の面では，Web ホスティングサービスの方式もクラウドサービスも，仮想的に構築されたサーバ環境であるインスタンスを複数のハードウェア上に起動させることにより，特定のハードウェアに障害が置きてもサービスを停止せずに提供可能である [13]．しかし，複数台のインスタンスが必要であることに起因するコストの問題や，ハードウェアが停止すると縮退状態のインスタンスでサービスを提供することになり，場合によってはリソース不足に陥る．

我々は，従来の Web ホスティングサービスを利用できる程度の知識を持った個人が WordPress のような一般的な CMS を利用した Web コンテンツを配信することを前提に，サービス利用者が負荷分散のシステム構築やライブラリの運用・管理を極力必要とせず，迅速にユーザ領域を複数のサーバに展開可能な，実行環境の変化に素早く適応できる恒常性を持つシステムアーキテクチャの FastContainer を提案した [14]．FastContainer は，インスタンスとしてコンテナのように起動の所要時間が小さい実行環境を採用し，リクエスト単位でコンテナの状態を決定する．そのため，オートスケーリングの観点において，アクセス集中時にはコンテナの負荷やアクセス数に応じて，コンテナをリクエスト契機に自動的に複製，あるいはリソースの追加割り当てを行い，迅速に自動的な負荷分散ができる．一方，可用性の面では，単一のコンテナが収容されているサーバが稼働していることが前提であり，収容サーバが停止すると，手動によるコンテナ再配置が必要であった．そのため，可用性を担保するには，従来手法と同様にコンテナを複数の収容サーバに起動して配置しておく必要がある．

本研究では，HTTP リクエスト処理時において，単一のコンテナであっても，収容サーバの状態に応じて，自動的にコンテナを別の収容サーバに再配置し，サービスを継続させる，HTTP リクエスト単位でのコンテナスケジューリング手法を提案する．提案手法では，FastContainer の状態変化の高速性に基づいて，コンテナが頻繁に収容サーバを変更されてもサービスに影響がないことを利用する．それによって，プロキシサーバから収容サーバに 1 個の ICMP パケットで応答速度を計測し，少ないパケット数と短いタイムアウトで収容サーバの反応時間を計測できる．．そのことで，HTTP リクエスト単位でのコンテナスケジューリングを実現する．

本論文の構成を述べる．2 章では，Web ホスティングサービスやクラウドサービスにおけるオートスケーリング，FastContainer 等の特徴とその課題について述べる．3 章では，2 章で述べたインスタンスの再配置の課題を解決するための提案手法のアーキテクチャおよび実装を述べる．4 章では，HTTP リクエスト単位でのコンテナスケジューリング手法の評価を行い，5 章でまとめとする．

2. Web サービス基盤の負荷分散と可用性

最もアクセスが集中しており，かつ，Web コンテンツを幅広く閲覧される可能性が高い状況においては，サーバが高負荷状態となってアクセスが困難となり，貴重なコンテンツ拡散の機会を逃すことも多い．また，ホストが収容されているハードウェアに障害があった場合に，いかにサービスを継続するかといった可用性の観点でシステムを構築する必要もある．本章では，相互に関連の深いスケーリングの特性と可用性の観点から，Web ホスティングサービスやクラウドサービスにおける関連研究と課題について整理する．

2.1 Web ホスティングサービス

負荷対応のためのスケーリング手法は，大きく分けて，稼働している単一のインスタンスに割り当てるハードウェアリソースを増減させるスケールアップ型と，複数のインスタンスへと起動数を増減することによって負荷分散を行うスケールアウト型の 2 つに分類できる．

一般的に，個人が Web コンテンツを配信するためには，Web ホスティングサービスやクラウドサービスなどが利用される [8]．特に，個人の利用を想定した場合，仮想ホスト方式を利用して，単一のサーバに高集積にユーザ領域を収容するような低価格 Web ホスティングサービスが利用される [11]．仮想ホスト方式では複数のホストを単一のサーバプロセス[*1]で処理するため，リクエストは Web サーバプロセスを共有して処理される．そのため，ホスト単位で使用するリソースを適切に制御したり，その原因を迅速に調査することが困難である [16]．

Web ホスティングサービス [8] では，サービス利用者の Web コンテンツは特定の Web サーバに収容され，Web サーバと Web コンテンツが紐づくため，負荷に応じたオートスケーリングは，データの整合性の面と前述したホスト単位での適切な負荷計測・制御の面から困難である．このような場合に適用可能な手法として，ユーザデータ領域を共有ストレージにまとめた上で，仮想ホスト方式を採用した複数台の Web サーバで負荷分散と可用性を担保する手法 [13] がある．しかし，Web サーバを複数台配置して全ホストを同一の設定にすることが前提となる手法であるため，ホスト単位でのスケーリングや可用性の担保は対応できず，コストも高くなる．また，負荷に応じて，ホスト単位での即時性の高いスケールアップ型の負荷対応もできない．

[*1] ただし，ここでいう単一のサーバプロセスとは，ホスト毎にサーバプロセスを起動させるのではなく，複数のホストでサーバプロセスを共有することを示す

2.2 クラウドサービス

クラウドコンピューティング [7] とは，ネットワークやサーバといったコンピュータリソースのプールから必要な時に必要な量だけオンデマンドに利用可能とするコンピューティングモデルである．クラウドサービスはクラウドコンピューティングを各種サービスとして提供するサービスである．

クラウドサービスでは，Web コンテンツを配置するだけでなく，Web サーバソフトウェアやデータベースをサービス利用者が自ら構築する必要がある．そのため，負荷分散のためのシステム設計をホスト単位で個別に行うことができる点において自由度は高いが，前提として専門的な知識が必要となる．

オートスケーリングについては，負荷に応じてインスタンスを増減させる機能が提供されている [3]．しかし，その負荷の監視間隔が分の単位であり，突発的なアクセスに対して検知するまでの時間が長くなる．負荷状況に応じて仮想マシンを起動できたとしても，テレビ放映の影響のような突発的な高負荷時に，オートスケーリングのための処理自体が追いつかず，サービス停止に繋がることも多い．

仮想マシンの起動時間の問題を解決するためにコンテナを利用する手法 [6] や，外部サービス連携によってスケーリングを行う条件を詳細に定義できるサービス[*2]もある．しかし，スケーリング時の判定を行う際に，OS の負荷やプロセスの状態等を監視する方式がとられており，監視の時間間隔や取得できる情報の粒度が荒く，突発的な負荷やハードウェア障害に対する即時スケーリングと誤検知との両立が難しい．

高負荷状態に対して迅速に対応するためには，事前にある程度想定される量の仮想マシンを予測的に起動させておくことによって対処する手法がある [17]．しかし，本手法は突発的なアクセスに対するスケーリングを対象としておらず，予測の精度を保つための各種パラメータの選定に関する課題もある．

上記のような問題を解決するために，クラウドサービスプロバイダの AWS は，プロバイダ指定の記法によってアプリケーションを実装すれば，自動的にコンピュータリソースを決定し，高負荷時には自動的にプロバイダ側でオートスケーリングする機能 [2] を提供している．しかし，前提としてプログラミングができるエンジニアを対象としており，一般的な OSS として公開されている Web アプリケーションを利用できないことが多く，個人が Web コンテンツを配信する用途においては使用上の制限が大きい．

可用性の面では，Web ホスティングサービスの方式と同様に，インスタンスを複数のハードウェア上に起動させることにより，特定のハードウェアに障害が置きてもサービスを停止せずに提供可能である．しかし，複数台のインスタンスが必要であることに起因するコストの問題や，ハードウェアが停止すると縮退状態のインスタンスでサービス提供することになり，場合によってはリソース不足に陥る．

[*2] http://www.rightscale.com/

2.3 FastContainer アーキテクチャ

我々は，個人が Web コンテンツを配信する用途において，Web ホスティングサービスやクラウドサービスにおける突発的にアクセス集中の対応やオートスケーリングの課題を解決するために，FastContainer アーキテクチャを提案した [14]．FastContainer は，Web ホスティングサービスを利用できる程度の知識を持った個人が，WordPress のような一般的な CMS を利用した Web コンテンツを配信することを前提にしている．FastContainer は，そのようなサービス利用者が負荷分散のシステム構築やライブラリの運用・管理を極力必要とせず，迅速にユーザ領域を複数のサーバに展開可能な，実行環境の変化に素早く適応できる恒常性を持つシステムアーキテクチャである．その特性を利用して，我々はメールシステムにも応用する手法を提案した [12]．

FastContainer アーキテクチャでは，インスタンスとして，仮想マシンではなく Linux コンテナ [5] を利用する．Linux コンテナはカーネルを共有しながらプロセスレベルで仮想的に OS 環境を隔離する仮想化技術のひとつである．そのため，コンテナの起動処理は仮想マシンのようなカーネルを含む起動処理と比べて，新しくプロセスを起動させる程度の処理で起動が可能であるため，起動時間が短時間で済むという特徴がある．

FastContainer アーキテクチャでは，コンテナが仮想マシンと比較して速く起動できる点と，FastCGI[4] を参考に，サーバへの収容効率を高めつつ性能を担保するアーキテクチャを組み合わせる．さらに，HTTP リクエスト毎に負荷状態やアクセス数に応じて，Web アプリケーションコンテナの起動処理，起動継続時間，コンテナの起動数およびリソース割り当てをリアクティブに決定する．一定時間起動することにより，一度コンテナが起動してしまえば，起動時間に影響なくレスポンスを送信できる．この特性によって，ホストをコンテナ単位で個別に起動しながらも，リクエストの無い不要なコンテナは自然に停止するため，高集積に収容できる．

FastContainer は，コンテナの起動が一般的に高速であること，リクエストを契機としたリアクティブな起動処理であることにより，突発的な負荷に対しても迅速にリクエスト単位でオートスケーリングが可能となる．スケールアップについても，コンテナのリソース管理が cgroup によってプロセス単位で制御されており，cgroup の特徴を利用して，プロセスが処理中であっても CPU 使用時間などの割り当てを即時変更できる．

しかし，このアーキテクチャはコンテナが収容されるサーバが稼働していることが前提であり，収容サーバが停止すると，手動によるコンテナ再配置が必要であった．また，単一のコンテナで起動している場合は，その収容サーバが停止すると，手動で再配置が行われるまでには同様にサービスも停止してしまう．そのため，ハードウェア障害時の可用性については，Web ホスティングサービスやクラウドサービスと同様に，複数の収容サーバに横断的に複数のコンテナを立ち上げておく必要があった．

3. 提案手法

個人の利用者が WordPress のような一般的な Web コンテンツを配信する仮想化基盤において，収容効率を高めつつ，アクセス集中による負荷やインスタンスの収容サーバの障害に対する可用性を担保可能な基盤にするためには，以下の要件が必要である．

(1) インスタンスで WordPress のような一般的な Web アプリケーションが動作する
(2) 単一のインスタンスであっても収容サーバ障害時には別サーバへ自動的に再配置される
(3) インスタンスの再配置の実行時であっても数秒の遅延で HTTP タイムアウトすることなくオンラインでレスポンスを返せる

本研究では，上述した 3 つの要件を満たすために FastContainer を応用して，HTTP リクエスト単位でコンテナを再配置する仮想化基盤の高速なスケジューリング手法を提案する．提案手法は，HTTP リクエスト処理時において，収容サーバ，および，その経路までの状態に応じて，自動的にコンテナを収容するサーバを決定し，サービスを継続させる．FastContainer の状態変化の高速性に基づいて，コンテナが誤検知によって再配置されてもサービスが停止しないことを利用する．それによって，プロキシサーバから収容サーバに ICMP で接続を試みた上で，短いタイムアウトで収容サーバの反応時間を計測できる．そのことで，HTTP リクエスト単位でのコンテナスケジューリングを実現する．

図 1 に，コンテナスケジューリングのフローを示す．

Host OS が障害を起こしていない正常時のフローについて述べる．図 1 における Client から Container に対して HTTP リクエストがあった場合，Client から Web Proxy に一旦集約される．Web Proxy にリクエストが到達した段階で，Container の収容情報や構成管理情報を保持する CMDB API に情報を問い合せ，Container の収容サーバや IP アドレス/ポート情報を得る．Web Proxy は収容サーバである Host OS 上に存在する Container にリクエストを転送する前に，その Host OS が稼働しているかを ICMP によって確認する．Web Proxy は ICMP の疎通を確認した後に，いったん Host OS 上に稼働している Container Dispatcher にリクエストを転送する．Container Dispatcher はコンテナが稼働していればそのまま HTTP リクエストを転送し，稼働していなければ，再度 CMDB API 情報に基いてコンテナを起動させてからリクエストを転送する．

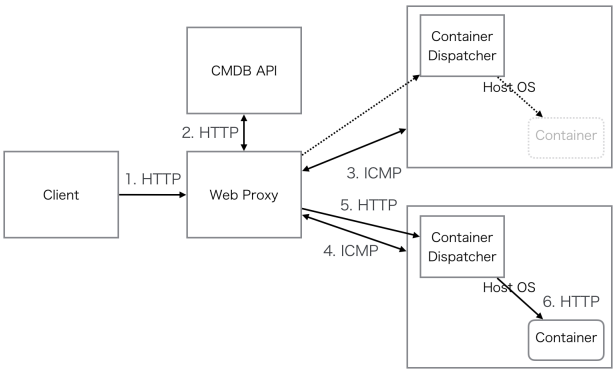

図 1 コンテナスケジューリングフロー
Fig. 1 The Flow of Contaienr Scheduling.

次に，コンテナが収容されている Host OS が突発的な障害で停止した場合のフローについて述べる．Client から CMDB API を介して Web Proxy に到達し，転送先の収容サーバに ICMP のチェックを行う．その際に，収容サーバがダウンしている場合は，ICMP のチェックタイムアウトを経て，Web Proxy が収容サーバのダウンを認識する．その後，Web Proxy は再度 CMDB API に接続を行い，コンテナの収容情報を他に起動している Host OS の情報に基いて再生成する．その収容情報に基いて，再度起動中の Host OS に ICMP チェックを行い，起動していることを確認した上で，Host OS 上で動作している Container Dispatcher にリクエストを転送する．この場合，収容サーバにはコンテナが起動していないため，Container Dispatcher によって該当コンテナを起動し，HTTP リクエストを転送する．

提案手法では，要件 (1) について，本手法は FastContainer アーキテクチャに基いており，複数の Host OS 間で共有ストレージによりデータを共有しているため，コンテナ上に WordPress のような Web アプリケーションが動作可能であり，かつ，別の Host OS 上でも同様に動作可能である．さらに，要件 (2) について，上記のスケジューリングフローによって，収容サーバがダウンしてもクライアントからの HTTP リクエスト処理の過程で再配置が行われる．要件 (3) を満たすためには，(1) で述べた通り，共有ストレージによってデータを Host OS 間で共有しているため，データのコピーは発生しないことから，ICMP のタイムアウトの時間と，コンテナの収容サーバ変更後のコンテナ再配置にかかる時間の合計値をできるたけ短くすれば良い．

ICMP のタイムアウト時間を短くするためには，Web Proxy と Host OS 間の通常の ICMP に必要とする時間にタイムアウト値をなるべく近くすることが必要となる．同時に，一時的な遅延によって，障害と至らないにも関わらず

表 1　実験環境
Table 1　Experimental Environment.

	項目	仕様
Client	ベンチマークを実施するクライアントサーバ	
	CPU	Intel Xeon E5-2650 2.20GHz 1core
	Memory	2GBytes
Compute	コンテナの収容サーバ	
	CPU	Intel Xeon E5-2650 2.20GHz 8core
	Memory	51GBytes
UserProxy	CMDB に基づきコンテナにリクエストを転送	
	CPU	Intel Xeon E5-2650 2.20GHz 1core
	Memory	2GBytes
CoreAPI	コンテナの構成管理情報を制御	
	CPU	Intel Xeon E5-2650 2.20GHz 1core
	Memory	2GBytes
CMDB	コンテナの構成管理情報を保存	
	CPU	Intel Xeon E5-2650 2.20GHz 1core
	Memory	16GBytes
DataPool	コンテナのコンテンツを格納	
	Storage	NetApp FAS8200A
	FlashPool	8.73 TB

タイムアウトした場合に再配置が発生したとしても，サービス継続への影響を最小化できれば良い．FastContainer のプロトタイプ実装と実験環境では，単一のコンテナを複数のコンテナにスケールアウトする場合に，レスポンスタイムが短縮されるまでにかかる時間が実験から数秒程度とわかっている [14]．スケールアウト型のスケーリングは，コンテナを追加で起動させる処理であり，コンテナの再配置の起動にかかる処理と同一の処理と言える．そのため，再配置に必要となる時間が論文の環境では数秒程度で完了することが見込める．また，コンテナの起動時間自体は FastContainer の課題である [10] ため，本手法のスコープは，再配置に要する時間を FastContainer におけるコンテナの起動時間にできるだけ近づけられることとする．

以上のことから，FastContainer の状態変化の高速性を活用することにより，積極的に ICMP タイムアウトを短くすることが可能である．また，Host OS が停止していない状況であっても，ICMP タイムアウトが生じた場合は何らかの負荷や異常が発生しているとも見なせる．それによって，別の Host OS にコンテナが再配置されることは，サービスが停止しない限りは有効なスケジューリングと考えられる．

4.　実験

本手法の有効性を確認するために，**図 2** に示す FastContainer を用いた実験環境を構築し，コンテナのスケジューリング処理を評価した．**表 1** に実験環境と各種ロールの役割を示す．実験環境の各ロールの NIC と OS は全て，NIC は 1Gbps，OS は Ubuntu16.04，カーネルは 4.4.0-59-generic

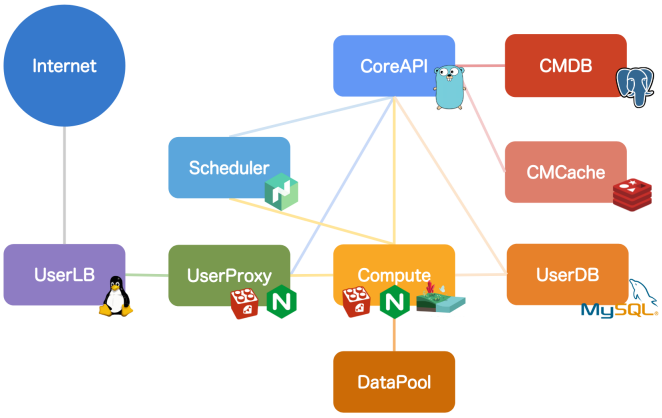

図 2　FastContainer のシステム構成例
Fig. 2　Example of FastContainer System.

表 2　ICMP の応答時間
Table 2　ICMP Response Time.

種類	応答時間
平均 (ms)	1.007
標準偏差 (ms)	1.499
最大値 (ms)	16.214
最小値 (ms)	0.0380

を利用した．

コンテナのデータは DataPool 上に保存し，Compute から DataPool に対して NFSv3 でマウントした．

ベンチマーク対象のコンテナは，40 バイトの静的ファイルを返すだけのコンテナと，WordPress4.9.5 をインストールしてデフォルトページを表示するコンテナを用意した．それぞれのコンテナには，Apache2.4.10 をインストールし，PHP のバージョン 7.1.11 を Apache モジュールとして組み込んで利用した．Apache 起動時には 5 プロセス起動し，アクセス集中時には 150 プロセスまで起動する設定を行った．

実験では，まず本実験のためのパラメータ設定と考察のために，予備実験を行った．その上で，本実験では，表 1 のロールを構築し，Client から UserProxy を介して，Compute に収容されているコンテナに対してベンチマークを行い，レスポンスタイムを計測した．

4.1　予備実験

4.1.1　ICMP タイムアウトに関する予備実験

最初に，ICMP チェックのタイムアウト時間を決定するために，UserProxy と Compute 間の ICMP の応答時間を計測した．計測には mruby-fast-remote-check を利用し，ICMP パケットを 1 パケット送信しその応答時間を測定した．測定は 1000 回行い，その平均と標準偏差，最大値，最小値を，小数点以下を有効数字 3 桁で計算した．

表 3　コンテナと Apache の起動時間合計
Table 3　Total Start Time of Container and Apache.

種類	応答時間
平均 (sec)	1.758
標準偏差 (sec)	0.417
最大値 (sec)	4.817
最小値 (sec)	1.045

表 2 に，ICMP 応答時間の結果を示す．実験環境では，ICMP 応答時間は平均 1msec 程度であることがわかった．また，最大値は 16msec 程度であることから，ICMP タイムアウトの値を 10msec 程度にすると，Compute が停止していない状態でも誤検知を起こす場合があると考えられる．そこで，本実験では UserProxy から Compute へ HTTP リクエストを転送する際の ICMP タイムアウトを，静的ファイルのような負荷の小さなコンテナの場合は 10msec，WordPress のような CPU を比較的多く利用するコンテナは 100msec で計測することにした．

4.1.2　Apache コンテナの起動時間に関する予備実験

続いて，実験環境において，Apache コンテナが最初に起動するのに要する時間を計測した．計測する理由として，Apache コンテナの起動時間を計測しておくことにより，実際に再配置が行われた場合のレスポンスタイムから，そのレスポンスタイムの妥当性を考察するためである．コンテナの起動時間自体は FastContainer の課題であるため，本実験のスコープは，再配置に要する時間を FastContainer におけるコンテナの起動時間にできるだけ近づけられることである．

FastContainer アーキテクチャによるコンテナ停止状態に対して HTTP リクエストを送信し，リクエスト契機にコンテナが起動してレスポンスを返すまでの時間を計測する．上述した，Apache と PHP が動作するコンテナを用いて，40 バイトの静的ファイルに対するレスポンスタイムを測定することで，全体のレスポンスタイムの内，コンテナと Apache の起動時間の合計が支配的になるようにした．また，40 バイトの静的ファイルに対するレスポンスタイムは，10msec 前後である．測定は 500 回行い，その平均と標準偏差，最大値，最小値を，小数点以下を有効数字 3 桁で計算した．

表 3 に，コンテナと Apache の起動時間合計を示す．コンテナの起動時間については，平均的には 1.758 秒であったが，最大値としては 4.817 秒，最小値としては 1.045 秒となった．つまり，理想的には，本手法によって収容サーバである Compute が停止した際には，ICMP タイムアウト値が 1msec 程度で，かつ，コンテナの再配置による起動時間が 1758msec であるため，それらの総和程度であることが見込める．

最大値が 4.817 秒かかっている原因の詳細は考察できていないが，Apache の起動時に設定ファイルやモジュールなどがキャッシュに載っているかどうかによって変わる可能性があるため，引き続き考察を行う．

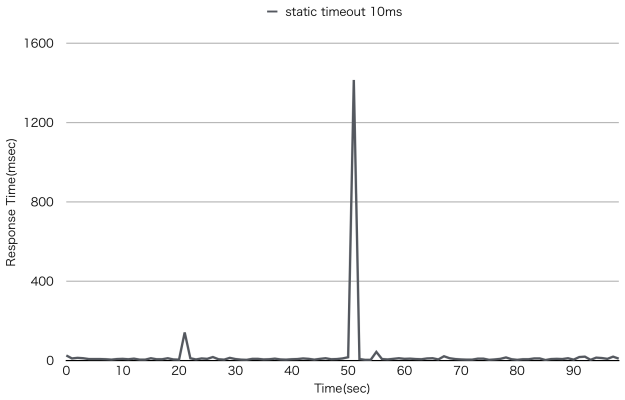

図 3　ICMP タイムアウト 10msec の場合の静的ファイルのベンチマーク
Fig. 3　Benchmark of Static Files for ICMP Timeout 10 msec.

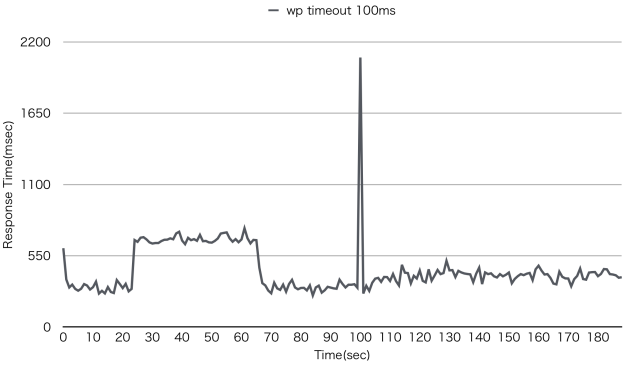

図 4　ICMP タイムアウト 100msec の場合の WordPress のベンチマーク
Fig. 4　Benchmark of WordPress for ICMP Timeout 100 msec.

4.2　再配置時のレスポンスタイムの本実験

予備実験の結果に基いて，静的ファイルと WordPress に対するレスポンスタイムを計測した．ベンチマーク測定中に，一定時間経過後に Compute 上で ipatables コマンドにより UserProxy からのパケットを drop し，別の Compute にコンテナが再配置されるようにした．

静的ファイルのコンテナの最大 CPU 使用量は，cgroup の機能により 1 コアのみ利用できるように割り当て，メモリは 512MB を割り当てた．また，WordPress のコンテナは，PHP の処理に CPU 使用時間を静的ファイルと比較して多く必要とするため，割り当てメモリは 512MB であるが，CPU は 4 コア利用可能となるように割り当てた．

静的ファイルを返すコンテナに対するベンチマークでは，ab コマンドで，同時接続数 10，総接続数 100000 で計測を行い，50 秒の時点でネットワークを切断した．この 50 秒は，静的ファイルに対する上述したパラメータによる総処

理時間が 100 秒程度であるため，前後の変化が明らかとなるようにその半分程度の時間に設定した．

また，WordPress のコンテナに対するベンチマークでは，ab コマンドで，同時接続数 10，総接続数 5000 で計測を行い，静的ファイルに対するコンテナと同様の理由から，100 秒の時点でネットワークを切断した．

ベンチマークでは，秒間のレスポンスタイムの平均値を計算し，時系列データとしてベンチマークが完了するまでのレスポンスタイムの変化を計測した．

図 3 に，ICMP タイムアウトが 10msec の場合の静的ファイルに対するベンチマーク結果を示す．実験環境においては，静的ファイルのベンチマークでは 1 リクエストにつき 10msec 前後でレスポンスを返せている．50 秒の時点で Compute が停止し，UserProxy が ICMP タイムアウト 10msec 経過後に，CoreAPI から再配置の情報を取得し，もう一台の正常に動作している Compute に再配置を行う．その際，ICMP タイムアウトに 10msec，再配置後，コンテナを起動させてレスポンスを返すまでに秒間平均 1400msec 程度時間を要している．その後 52 秒の段階では，既に 10msec 前後のレスポンスタイムとなっているため，再配置に要する時間は 1.5 秒程度であった．

図 4 に，ICMP タイムアウトが 100msec の場合の WordPress に対するベンチマ¿ーク結果を示す WordPress では，実験環境においては平均的に 400msec 前後のレスポンスタイムであった．100 秒の時点で再配置の処理が動作した際には，2080msec の処理時間を要した．ICMP のタイムアウト値が 100msec であることから，コンテナの再配置から Apache の起動とレスポンス生成までには概ね 1980msec で実現できている．そのことから，コンテナの再配置とコンテナ起動に要する時間は，WordPress のレスポンスタイム 400msec を除くと，1580msec となる．. また，22 秒の時点から 70 秒の時点まで一時的にレスポンスタイムが 2 倍に増加している期間があった．これは，実験では原因が特定できなかったため，引き続き調査予定である．

以上の実験結果について，4.1.1 節と 4.1.2 節の数値からも，再配置に要する時間は妥当であることがわかった．4.1.2 節の実験結果のコンテナの起動時間の平均や分散からも，再配置後のコンテナからのレスポンスに要する時間は，静的コンテンツも WordPress もコンテナの起動時間が支配的であり，本手法における再配置手法そのものの処理時間は影響がないことがわかった．また，誤検知による再配置は生じなかったが，1msec までタイムアウトを短くすると，実験環境においては大量に再配置が発生することがわかっている．

一方で，再配置時に 9 リクエストだけ，500 系のエラーを返していたことが分かった．これは，(同時接続数-1) の値であり，システム実装上の問題で，CoreAPI に対して同時に再配置の依頼が要求された際に，1 つを除いてエラーを返す実装にしていたためである．この問題については，1 つを除いて再配置の要求リクエストを一時停止させ，再配置が完了した時点で残りのコンテナへの HTTP リクエストを再配置後のコンテナに転送することで解決できる．

UserProxy から Compute にリクエスト転送が送信されて，Compute 上のコンテナから UserProxy にレスポンスデータが送信される前の状態で Compute が停止してしまった場合は，そのレスポンスを UserProxy が正しく受信できないため，レスポンスを一部取りこぼすこともありえる．しかし，これは従来方式において複数のホスト OS にそれぞれ分散してインスタンスを配置して可用性を担保する方式でも原理的に生じるため，本手法の特徴的な問題ではない．

本実験結果から，単一のコンテナインスタンスを単一の収容サーバに起動させておきながらも，その収容サーバの停止時には，迅速に再配置のスケジューリングが行えていることがわかった．また，実験環境程度のハードウェアスペックで，1.5 秒から 2 秒程度であれば，実用上は HTTP タイムアウトすることなくレスポンスを返せると見込めるが，提供サービスレベルにも依存する問題でもあるため検討が必要である．一方，FastContainer の課題のうち，コンテナのイメージ化によって起動時間を高速化する研究 [10] が改善されれば，再配置に占めるコンテナの起動時間も短縮できるため，よりこの手法を活かすことができる．

5. まとめ

本研究では，HTTP リクエスト処理時において，収容サーバ，および，その経路までの状態に応じて，自動的にコンテナを収容するサーバを決定し，サービスを継続させる，HTTP リクエスト単位でのコンテナスケジューリング手法を提案し，その有効性を評価した．

本研究により，WordPress のような一般的な Web アプリケーションが起動しているインスタンスにおいて，複数のホスト OS に複数のインスタンスを配置して可用性を担保することなく，単一のインスタンスという少ないリソースで，1.5 秒から 2 秒程度の再配置処理により，HTTP エラーを発生させることなく可用性を担保することができた．このようなリソース使用量の低減により，本手法は個人用途のホスティングサービスのような低価格が求められるサービスの設計に適している．

今後の課題として，従来からの課題である FastContainer アーキテクチャのコンテナ起動時間を更に短縮することや，負荷やレスポンスタイムに応じた HTTP リクエスト単位でのコンテナスケジューリング手法の実現が挙げられる．

参考文献

[1] Amazon Web Services, https://aws.amazon.com/.
[2] Amazon Web Services: Lambda, https://aws.amazon.

com/lambda/.
[3] Amazon Web Services: Auto Scaling, https://aws.amazon.com/autoscaling/.
[4] Brown Mark R, FastCGI: A high-performance gateway interface, Fifth International World Wide Web Conference. Vol. 6. 1996.
[5] Felter W, Ferreira A, Rajamony R, Rubio J, An Updated Performance Comparison of Virtual Machines and Linux Containers, IEEE International Symposium Performance Analysis of Systems and Software (ISPASS), pp. 171-172, March 2015.
[6] He S, Guo L, Guo Y, Wu C, Ghanem M, Han R, Elastic application container: A lightweight approach for cloud resource provisioning, Advanced information networking and applications (AINA 2012) IEEE 26th international conference, pp. 15-22, March 2012.
[7] P Mell, T Grance, The NIST Definition of Cloud Computing", US Nat'l Inst. of Science and Technology, 2011, http://csrc.nist.gov/publications/nistpubs/800-145/SP800-145.pdf.
[8] Prodan R, Ostermann S, A Survey and Taxonomy of Infrastructure as a Service and Web Hosting Cloud Providers,10th IEEE/ACM International Conference on Grid Computing, pp. 17-25, October 2009.
[9] Ferdman M, Adileh A, Kocberber O, Volos S, Alisafaee M, Jevdjic D, Falsafi B, Clearing the clouds: a study of emerging scale-out workloads on modern hardware, ACM SIGPLAN Notices, Vol. 47, No. 4, pp. 37-48, March 2012.
[10] 笠原 義晃, 松本 亮介, 近藤 宇智朗, 小田 知央, 嶋吉 隆夫, 金子晃介, 岡村 耕二, 軽量コンテナに基づく柔軟なホスティング・クラウド基盤の研究開発と大規模・高負荷テスト環境の構築, 研究報告インターネットと運用技術（IOT）, Vol.2018-IOT-40(2), pp.1-8, 2018 年 3 月.
[11] 松本 亮介, Web サーバの高集積マルチテナントアーキテクチャに関する研究, 京都大学博士 (情報学) 学位論文, 2017.
[12] 松本 亮介, 小田 知央, 笠原 義晃, 嶋吉 隆夫, 金子晃介, 栗林 健太郎, 岡村 耕二, 精緻に解析可能な恒常性のあるメール基盤の設計, 研究報告インターネットと運用技術（IOT）, Vol.2018-IOT-40(17), pp.1-8, 2018 年 3 月.
[13] 松本亮介, 川原将司, 松岡輝夫, 大規模共有型 Web バーチャルホスティング基盤のセキュリティと運用技術の改善, 情報処理学会論文誌, Vol.54, No.3, pp.1077-1086, 2013 年 3 月.
[14] 松本亮介, 近藤宇智朗, 三宅悠介, 力武健次, 栗林健太郎, FastContainer: 実行環境の変化に素早く適応できる恒常性を持つシステムアーキテクチャ, インターネットと運用技術シンポジウム 2017 論文集, 2017, 89-97（2017-11-30）, 2017 年 12 月.
[15] 松本亮介, 田平 康朗, 山下 和彦, 栗林 健太郎, 特徴量抽出と変化点検出に基づく Web サーバの高集積マルチテナント方式におけるリソースの自律制御アーキテクチャ, 情報処理学会研究報告インターネットと運用技術（IOT）,2017-IOT-36(26), 1-8, (2017-02-24).
[16] 松本亮介, 栗林 健太郎, 岡部寿男, リクエスト単位で仮想的にハードウェアリソースを分離する Web サーバのリソース制御アーキテクチャ, 情報処理学会論文誌, Vol.59, No.3, pp.1016-1025, 2018 年 3 月.
[17] 三宅 悠介, 松本 亮介, 力武 健次, 栗林 健太郎, アクセス頻度予測に基づく仮想サーバの計画的オートスケーリング, 情報処理学会研究報告インターネットと運用技術（IOT）,2017-IOT-38, Vol.13, pp.1-8, 2017 年 6 月.