Vol.2017-IOT-38 No.14
2017/6/24

FastContainer: Webアプリケーションコンテナの状態をリアクティブに決定するコンテナ管理アーキテクチャ

松本 亮介[1,a)]　近藤 宇智朗[2]　三宅 悠介[1]　力武 健次[1,3]　栗林 健太郎[1,b)]

概要：クラウドサービスやWebホスティングサービスの低価格化と性能の向上に伴い，コンテナ型の仮想化技術を活用することにより，複数のユーザ環境の収容効率を高めると同時に，セキュリティの担保とリソース管理を適切に行うことが求められている．一方で，障害時の可用性やアクセス集中時の負荷分散については依然として各システムに依存している．本研究では，HTTPリクエスト毎に，コンテナの起動，起動時間，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コンテナ管理アーキテクチャを提案する．提案手法により，アクセス集中時にはコンテナがHTTPリクエストを契機に，アクセス状況に応じて複製・破棄されることで，迅速に自動的な負荷分散が可能となる．さらに，コンテナが一定期間で破棄されることにより，収容効率を高め，ライブラリが更新された場合には常に新しい状態へと更新される頻度が高くなる．

FastContainer: Reacvtive Container Management Architecture with Lifespan Mortality for Web Applications

Ryosuke Matsumoto[1,a)]　Uchio Kondo[2]　Yusuke Miyake[1]　Kenji Rikitake[1,3]
Kentaro Kuribayashi[1,b)]

Abstract: Price reduction and performance improvement of cloud computing and Web hosting services lead to more demand on efficiency of highly utilized multi-tenant user environment while maintaining the security and appropriate resource management, by making use of container virtualization technology. On the other hand, maintaining the availability and load balancing on access congestion are still dependent on each system configuration. In this paper, we propose a container management architecture which reactively decides invocation, running periods, simultaneous running numbers and assigned resources of the containers in connection with the incoming HTTP requests. Our proposed architecture enables automatic and rapid load balancing by generating and discarding the containers following the access frequency in case of access congestion. The proposed method improves efficiency of resource utilization by automatically discarding the containers in a fixed period, which also contributes to increase the chance of reflecting the library updates.

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

[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 技術部 技術基盤チーム
Developer Productivity Team, Engineering Department, GMO Pepabo, Inc., Tenjin, Chuo ku, Fukuoka 810-0001 Japan
[3] 力武健次技術士事務所
Kenji Rikitake Professional Engineer's Office, Toyonaka City, Osaka 560-0043 Japan
a) matsumotory@pepabo.com
b) antipop@pepabo.com

になってきている．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，クラウド・VPS サービスなどが利用される [9].

Web ホスティングサービスやクラウド・VPS サービスの低価格化と性能の向上に伴い，Web アプリケーションの複数の実行環境を単一の Web サーバ上で安定かつセキュアに提供するために OS の仮想化技術 [3] が活用されている．OS の仮想化技術の中でも，プロセス単位でユーザ領域を隔離してリソース管理ができるコンテナ型の仮想化技術 [12] を活用することにより，仮想マシンと比べて，複数の実行環境の収容効率を高めることができ，プロセス単位でのセキュリティも担保できる．また，仮想マシンと比較して，高速に仮想環境を起動することができる．

単一のサーバに複数のユーザ領域を収容するような Web ホスティングサービスでは，一般的に，利用者の Web コンテンツが特定の Web サーバに紐づくため，負荷分散することが難しい．また，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ル [11] 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ルの機能を使う必要がある．AWS[1] などのサービス・プロバイダが提供しているオートスケールでは，サービス・プロバイダ自身が提供する監視項目に基づき，内部で自力構築したもの，または，外部のサービスを利用して仮想マシンを自動で起動させる必要があるため，突発的なアクセス集中に対して，スケール処理に時間がかかり負荷分散が間に合わない場合が多い [6]．また，迅速に負荷分散の仕組みを構築するのは，個人の利用者にとっては困難である．

本研究では，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 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にするために，Web アプリケーションコンテナの起動を HTTP リクエスト単位でリアクティブに決定するコンテナ管理アーキテクチャを提案する．ここでのリアクティブな決定とは，HTTP リクエスト単位で外的要因に基づく負荷の状態やレスポンス性能を検出し，状況に応じたコンテナの構成を迅速に決定することを意味する．Web アプリケーションの実行におけるユーザデータとアプリケーションの処理を明確に分離した上で，迅速に負荷分散処理を実現するために，HTTP リクエストを契機に，コンテナの起動処理，起動継続時間，リソース割り当ておよびコンテナ起動数を決定する．提案手法により，コンテナが一つ以上起動していれば即時レスポンスを送信し，コンテナが停止していた場合は，リクエストを契機にコンテナを一定期間起動させて複数のリクエスト処理を行う．これによって，コンテナの起動の所要時間が小さいため，リクエストを受け付けない時間が短くできる．また，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．アクセス集中時には，既に起動済みのコンテナの負荷やレスポンス性能の劣化を，リクエスト単位またはコンテナ自身の監視プロセス経由で検知することで，コンテナ自身を自動的に複製し，迅速に自動的な負荷分散ができる．また，コンテナが一定期間で破棄され，プロセスの起動時間を短縮することによって，マルチテナント方式 [7] のリソース効率を高め，同時に，ライブラリが更新された場合には新しい状態へと更新される頻度が高くなる．

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ルとその課題について述べる．3 章では，2 章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，2 章におけるオートスケールによる負荷分散の評価を行い，5 章でまとめとする．

2. 負荷分散と運用技術

最もアクセスが集中しており，かつ，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，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．本章では，Web ホスティングサービスやクラウド・VPS サービスにおける，負荷分散のためのオートスケールや関連する運用技術について整理する．負荷対応のためのスケール手法の分類は，大きく分けて，稼働している単一のインスタンス[*1]の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がある．

2.1 Web ホスティングサービスの特徴

Web ホスティングサービス [9] 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ルはデータの整合性の面で困難である．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 サーバで負荷分散を行う手法 [13] もあるが，仮想ホスト方式は複数のホストを単一のサーバプロセス[*2]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厳密に限定し区別することが困難 [15] であり，スケール時のコスト計算や必要な時に必要なだけリソースを追加するような方式が利用できない．また，負荷の応じた即時性の高いスケールアップ型の負荷対

[*1] 仮想的に構築されたサーバ環境
[*2] 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

応も困難である．

運用技術においては，ライブラリの更新の際に，沢山のホストを単一のサーバプロセスで処理している特性上，サーバプロセスの再起動時の影響が大きくなる．また，サーバ高負荷時には，リソースを適切に限定することが困難であるため，高負荷原因の調査と制御にコストがかかり [14]，サービス品質への影響も大きい．

サービス利用者の観点では，利用できる Web サーバソフトウェアをはじめとしたミドルウェア及び機能が全ホスト共通となり，システム構築の面で自由度が低い．

2.2 クラウド・VPS サービスの特徴

クラウドコンピューティング [8] とは，ネットワークやサーバといったコンピューティング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個別に行うことができる点において自由度は高いが，専門的な知識が必要となる．オートスケールについても，負荷に応じて増減させる機能が提供されている [2] が，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させたとしても，テレビ放映の影響のような突発的な高負荷時に，オートスケール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．また，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ルを行う条件を詳細に定義できるサービス[*3]もあるが，スケール時の判定を行う際に，外部サーバなどから 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，突発的な負荷に対して即時性が低くなる問題もある．そこで，事前にある程度想定される量の仮想マシンを起動させておくことによって対処する必要があるが，定量的な見積もりや事前のメンテナンスが必要であったり，限られたコストの兼ね合いから適切な見積もりをすることは困難である．

VPS サービスは，OS がインストールされた仮想マシンを利用者に貸し出すサービスである．VPS サービスを利用している場合，クラウドサービスと同様利用者が自ら Web コンテンツを配信するための構築および保守を行う必要がある．アクセス集中時には，仮想マシンに割り当てるハードウェアリソースを増強するスケールアップによって対応することができるが，一般的には仮想マシンのデータをスナップショット機能などにより保存しておいて，ベースの仮想マシンを作り直して再起動するといった対応が必要となり，即時性は低く，負荷に応じた対応が難しい．

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ルする機能[*4]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，処理の実行時間が限定的であるといった使用上の制限が大きい．このようなサービスを使う場合に，専門的な知識なく Web コンテンツを公開した上でオートスケールすることは困難である．

2.3 Web サーバ機能のプロアクティブ性とリアクティブ性

単一のサーバに高集積にホストが収容可能であり，ホスト単位でのリソース管理を適切に行いながら，セキュリティと性能および負荷に強い Web ホスティング環境を構築することを目的とした場合，Web サーバ機能をプロアクティブ性とリアクティブ性に基いて分類できる．以下に，Web サーバ機能のプロアクティブ性とリアクティブ性を定義する．

プロアクティブ性とは，Web サーバ機能を持つ仮想マシン，コンテナおよび Web サーバプロセスが予め起動しており，リクエストに応じて仮想マシンやコンテナの状態を即時変更できないが，常に起動状態であるため，高速にリクエストを処理できる性質とする．また，常時 Web サーバ機能を稼働させておく必要があるため，リソース効率が悪い．プロアクティブ性をもったオートスケールは，例えば 2.2 で述べたように，事前にアクセス頻度から予測を行い，予測に基づいた数だけインスタンスを起動させておくようなアプローチである．

リアクティブ性とは，CGI や FastCGI[4] のように，アプリケーションが実用上現実的な速度で起動可能であることを前提に，リクエストに応じてアプリケーションを起動する性質とする．リアクティブ性を持つ Web サーバ機能は，起動と停止のコストは生じるため，性能面はプロアクティブ性を持つ Web サーバ機能より劣るが，リクエストを受信しない限りはプロセスが起動しないため，リソース効率が良い．また，リクエストに応じて複数起動させるといった変更に強い処理が実装し易い．FastCGI はリソース効率と性能を両立するために，一定期間起動して連続するリクエストを高速に処理可能とするアーキテクチャをとっ

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

[*4] https://aws.amazon.com/lambda/

ている．これまでは，CGI のようなリアクティブ性に基づく処理は性能面の問題などから利用されなくなってきており，オートスケールについても，リクエスト単位で仮想マシンやコンテナを都度起動させるコストを考慮すると，実用的な性能を満たすことは困難である．

3. 提案手法

現状の各種サービスの特徴を考慮した場合，限られたリソースの範囲内で負荷に応じて即時インスタンスを制御するためには，リアクティブ性を持つ Web サーバ機能を前提に，インスタンスを柔軟に管理し，実用上問題にならない程度の性能を担保するアーキテクチャが必要となる．以下に要件をまとめる．

(1) HTTP リクエスト単位の粒度で迅速にインスタンスのスケールアウトとスケールアップできる
(2) HTTP リクエスト単位の粒度でインスタンスの監視を行い即時スケール処理の命令を出せる
(3) リソース効率化のため必要の無いインスタンスは停止可能であり，必要な時に HTTP リクエスト単位で起動できる

また，このアーキテクチャによって実現される Web ホスティングサービスとしては以下の要件が必要となる．

(1) OS やライブラリの更新作業のようなサーバ運用はサービス提供側が行う
(2) 広く使われる一般的な Web アプリケーション（WordPress など）を利用できる
(3) アクセスが集中した際に専門的な知識がなくてもオートスケールによる負荷分散が行われる
(4) Web アプリケーション実行時間程度の粒度での課金が可能である
(5) ホストの収容効率を高めることによってハードウェアコストを低減する
(6) 高頻度でセキュリテイを担保するための OS やライブラリの更新が行われる

そこで，本研究では，Web コンテンツを配信するために，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とするために，Web アプリケーションコンテナの起動，起動継続時間，起動数およびスケール処理の判定を HTTP リクエスト毎にリアクティブに決定するコンテナ管理アーキテクチャを提案する．このアーキテクチャを FastContainer と名付ける．

3.1 FastContainer アーキテクチャ及び関連技術

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 コンテナを利用する．Linux におけるコンテナ [5] 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．また，コンテナ環境単位であるため広く使われている Web アプリケーションを隔離して利用できる．

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，2.3 で述べた FastCGI のようにリソース効率を高めつつ，性能も担保するアーキテクチャを組み合わせた上で，コンテナ上で起動する Web アプリケーションの実行処理におけるデータとアプリケーションの処理を明確に分離する．さらに，HTTP リクエスト毎に負荷状態やレスポンス性能に応じて，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．

提案手法では，最低一つのコンテナが常に起動していることを前提に，コンテナが一つ以上起動していれば即時レスポンスを送信し，コンテナが停止していた場合は，リクエストを契機にコンテナを一定期間起動させて複数のリクエスト処理を行う．これによって，仮にコンテナが全て停止していたとしても，コンテナがリクエスト単位で起動するため可用性が高くなる．例えば，メモリリークが生じるような不完全なソフトウェアが動作していたとしても，定期的に循環が行われることにより，メモリが解放される．この特徴により，システム全体がメモリ不足にならないため，そういったソフトウェアを許容できる．ただし，システムとしては許容できても，ソフトウェアとして不完全であることを検知する必要がある．また，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．

アクセス集中時には，既に起動済みのコンテナがコンテナ自身の負荷状態およびレスポンス性能に基いて，Web サーバが自律的にスケール処理を開始する．2 つの方法で監視を行い，1 つはコンテナの管理マネージャによって，コンテナに割り当てることのできなかった CPU 使用時間を監視し，もう 1 つは，レスポンスタイムをコンテナの前段に配置した Web プロキシが監視する．これらを組み合わせることにより，コンテナのハードウェアリソースが不足しているのか，あるいは，単に同時接続数が多いことによって処理できないリクエストが発生しているのかを判定する．監視で設定した閾値を超えた場合には，自動的に新しいコンテナの構成情報をコンテナの収容情報を構成管理データベース（CMDB）に対して管理マネージャ経由で登録する．その後，コンテナの前段に配置されている Web プロキシが構成管理データベースに基づいて，新しいコンテナへとリクエストを転送する．コンテナが収容されているサーバ上で動作している Web ディスパッチャが，プロ

キシ先のコンテナが起動していればそのままリクエストを転送し，起動していなければ，CMDB からコンテナの構成情報を取得して，コンテナを先に起動しリクエストを転送する．ただし，コンテナが起動中の場合は既に起動済みのコンテナにリクエストを転送し，起動が完了してからリクエストが振り分けられるようにする．Web アプリケーションに関するデータは共有ストレージ上に配置し，コンテナを収容するサーバ群は同一領域をマウントすれば，どのサーバにコンテナが起動していても，CMDB 上にコンテナの構成情報に基づいて適切に動作可能となる．

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，および，オートスケールの監視手法がリクエスト単位での粒度で行われることにより，突発的な負荷に対しても迅速にオートスケールが可能となる．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[10] 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．また，コンテナが一定期間で破棄されることにより，不必要なプロセスの起動数を低減してコンテナの収容効率を高め，ライブラリが更新された場合には常に新しい状態へと更新されることが保証される．

3.2 Haconiwa: コンテナ管理ツール

FastContainer アーキテクチャをシステムとして実現するためには，コンテナを複雑に制御する必要がある．Haconiwa[*5]は筆者の一人である近藤らによって開発されており，コンテナのリソース割り当てやプロセス隔離の構成情報の設定だけでなく，コンテナ起動・停止時やコンテナのセットアップ時の各種フェーズで Ruby DSL[*6]を実行することにより，コンテナの振る舞いをプラガブルに定義できるソフトウェアである．コンテナを固定的な仮想環境としてのみ利用するだけでなく，プログラムで制御が可能なネットワーク上で動作するスレッドとみなせるような方針で開発されている．Haconiwa は，FastContainer アーキテクチャのように，リクエスト単位でリアクティブにコンテナを起動させることができ，かつ，一定時間起動した後に停止する処理や，Haconiwa で起動されたコンテナを管理するプロセスがコンテナのリソース使用状態を監視して，状態によって HTTP ベースの API にアクセスするといったような動的な処理を DSL で平易に記述できる．また，Haconiwa によって構築されたコンテナのリソース割り当ては cgroup を活用しており，コンテナのプロセスを停止させることなく，処理中であっても割り当てを即時変更することができる．**図 1** のように書かれた Haconiwa の設定を実行すると，bootstrap API により，git でコンテナのイメージを取得後，provision API によりコンテナに Ruby の環境を作り，コンテナ起動と同時にコンテナ内部で Ruby プログラムが sleep するだけの処理を実行する．その後，Haconiwa の add_async_hook API の定義に従い，30 秒後にコンテナを停止する．

```
Haconiwa.define do |config|
  config.name = "haconiwa-auto-droptest"
  # Ruby loop daemon:
  config.init_command = ["/usr/bin/ruby",
                         "-e",
                         "loop { sleep 1 }"]
  config.daemonize!

  root = Pathname.new("/var/lib/haconiwa/#{config.name}")
  config.chroot_to root

  config.bootstrap do |b|
    b.strategy = "git"
    b.git_url = "https://example.jp/haconiwa.image"
  end

  config.provision do |p|
    p.run_shell <<-SHELL
apk add --update bash
apk add --update ruby
    SHELL
  end

  config.add_async_hook(msec: 30 * 1000) do |base|
    Haconiwa::Logger.info("Process killed: #{base.pid}")
    ::Process.kill :TERM, base.pid
  end

  config.mount_independent "procfs"
  config.mount_independent "sysfs"
  config.mount_independent "devtmpfs"
  config.mount_independent "devpts"
  config.mount_independent "shm"

  config.namespace.unshare "mount"
  config.namespace.unshare "ipc"
  config.namespace.unshare "uts"
  config.namespace.unshare "pid"
end
```

図 1 Haconiwa の記述例
Fig. 1 Haconiwa example.

コンテナの構成を管理するソフトウェアとしては，LXC[*7]や Docker[*8]，rkt[*9]がある．FastContainer アーキテクチャを適用したシステムおよびツールは，多数のコンテナで構成されたシステムを自動で管理するためのコンテナオーケストレーションソフトウェアであり，Haconiwa は

[*5] https://github.com/haconiwa/haconiwa
[*6] Domain-specific language
[*7] https://linuxcontainers.org/
[*8] https://www.docker.com/
[*9] https://coreos.com/rkt/

単一のコンテナを管理するツールと定義できる．

3.2.1 既存のコンテナ管理ツール

Docker はコンテナによる仮想環境の独立性を重視しているため，コンテナを構成するプロセスの隔離技術やリソース割り当て技術が機能として密結合になっている．一方で Haconiwa は，必要なコンポーネントを組み合わせて，chroot() システムコールによる最低限のファイルシステム隔離に加え，CPU やメモリのリソース割り当てのみを適用した環境を DSL によって平易に記述することができる．そのため，構築したい仮想環境に応じて，仮想環境の独立性と運用性のバランスを検討しやすい設計になっている．

LXC は Haconiwa と同様に比較的プラガブルな作りになっているものの，コンテナの振る舞いを定義するためのフックフェーズが，コンテナ起動や停止時のみといったように限定的である．一方 Haconiwa は，シグナルハンドラや起動後の非同期遅延処理，さらには定期実行のタイマー処理を定義して実行することも可能となっており，コンテナを活用した様々なシステムに適用しやすくなっている．rkt は多くの設定をサポートしているものの，Ruby DSL のようにプログラマブルに記述することはできず，コマンドラインのオプション設定で実行する必要がある．

3.2.2 コンテナのオーケストレーションソフトウェア

代表的なコンテナのオーケストレーションソフトウェアとして Kubernetes[*10]がある．Haconiwa は，通常の Docker や LXC による単一あるいは複数の平易なコンテナ管理に加え，Ruby DSL を各コンテナの処理フェーズで記述可能で，任意の処理を実行可能であることから，独自で実装する場合に複雑になりがちなオーケストレーション層との連携を想定した作りとなっており，Haconiwa 自体がオーケストレーションソフトウェアとしても遜色ない使い方が可能である．その場合，Kubernetes は設定を単一の yaml ファイルで記述する必要があり，独自のオーケストレーション層を実装する場合において自由度が低い．今後コンテナを活用した複雑なコンテナ設定，あるいは，オーケストレーション層との連携が求められることが想定されるため，Haconiwa のようにコンテナの設定層およびオーケストレーション層との接続を DSL で統一的に記述でき，かつ，Ruby のような一般的に広く使われるプログラミング言語を利用できることにより，独自に定義された記述言語よりも自由度が高く，学習もしやすいと考えられる．

3.3 システム構成例

図 2 に FastContainer アーキテクチャのシステム設計例を示す．Internet からアクセスのあった HTTP リクエストは，UserLB から UserProxy に転送され，UserProxy は CMDB の情報に基いて，リクエストを転送すべきコンテナの収容ホストである Compute 上で動作している Web ディスパッチャにまずはリクエスト転送する．Web ディスパッチャは，リクエスト転送先のコンテナが起動していれば，そのままリクエスト転送し，起動していない場合は，CMDB 上から起動すべきコンテナの構成情報を取得し，Compute 上の Haconiwa に対してコンテナ生成の指示を出す．その後，Haconiwa がコンテナを起動させた後に，HTTP リクエストをコンテナに転送して処理を行う．また，CMDB 上に複数のコンテナ情報が登録されている場合は，転送先をランダムに決定する．既に 1 台のコンテナが起動済みで，アクセス集中などにより新たなコンテナを起動させる必要がある場合は，起動済みのコンテナの負荷状態やレスポンスタイムを，Web ディスパッチャや Haconiwa のコンテナ監視プロセスが監視し，スケール処理が必要と判定した際には CMDB に対して新たなコンテナの追加情報を登録する．登録後，新たに Internet から HTTP リクエストを受けた場合に，CMDB 上に登録されているがまだ起動していないコンテナの場合，起動処理の開始と同時に，CMDB に対して一時的に起動中のコンテナに対するリクエストの転送を停止し，既に起動しているコンテナのみに転送する．その後，コンテナが起動を完了しポートの Listen を開始した後に，CMDB に対して再度リクエスト転送の再開し，処理を開始する．これによって，起動中のコンテナに HTTP リクエストが滞留することを防ぐ．Web コンテンツは DataPool 上

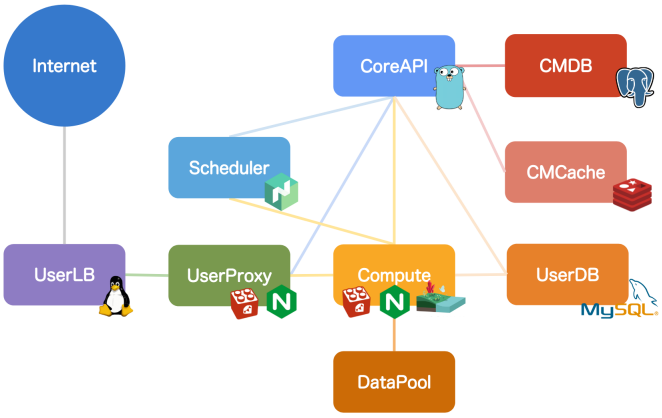

図 2 Haconiwa による FastContainer のシステム構成例
Fig. 2 Example of FastContainer System using Haconiwa.

表 1 FastContainer のシステムのコンポーネントの役割
Table 1 Roles of FastContainer System Compoments.

役割	概要
CoreAPI	コンテナの構成管理情報を制御
CMDB	コンテナの構成管理情報を保存
UserProxy	CMDB に基づきコンテナにリクエストを転送
Compute	コンテナが動作するサーバ
DataPool	コンテナのコンテンツを格納

[*10] https://kubernetes.io/

表 2　実験環境
Table 2　Experimental Environment.

	項目	仕様
Compute	CPU	Intel Xeon E5-2650 2.20GHz 12core
	Memory	39GBytes
	NIC	1Gbps
	OS	Ubuntu 14.04 Kernel 4.4.0
UserProxy	CPU	Intel Xeon E5620 2.40GHz 4core
	Memory	4GBytes
	NIC	1Gbps
	OS	Ubuntu 14.04 Kernel 4.4.0
CoreAPI	CPU	Intel Xeon E5620 2.40GHz 8core
	Memory	8GBytes
	NIC	1Gbps
	OS	Ubuntu 14.04 Kernel 4.4.0
CMDB	CPU	Intel Xeon E5620 2.40GHz 4core
	Memory	16GBytes
	NIC	1Gbps
	OS	Ubuntu 14.04 Kernel 4.4.0
DataPool	CPU	Intel Xeon E5620 2.40GHz 2core
	Memory	4GBytes
	NIC	1Gbps
	OS	Ubuntu 14.04 Kernel 4.4.0

に配置されており，各コンテナは NFSv4 を経由してデータを共有する．CMCache は CMDB のキャッシュ，UserDB はユーザが利用するデータベース，Scheduler はコンテナ初期化時のジョブ処理や環境構築の処理に利用される．

4. 実験

FastContainer アーキテクチャの有効性を確認するために，図 2 に示した FastContainer を用いたプロトタイプ環境を構築し，コンテナのスケールアウトの性能を評価した．図 2 において，実験で利用するサーバの各種役割は**表 1** である．**表 2** に実験環境を示す．実験では，表 2 のロールのみを構築し，UserProxy から Compute で起動しているコンテナに対してベンチマークを行う．コンテナ上には，Apache2.4.10 を 1 プロセスで起動させた上で PHP5.6.30 をインストールし，PHP の環境情報を取得する phpinfo() 関数を実行するだけのコンテンツを動作させる．また，コンテナの最大 CPU 使用量は，cgroup の機能により 1 コアの 30%に制限し，その設定のもと予備実験から CPU を 30%使い切ることのできるベンチマークの設定を同時接続数 100，総リクエスト数を 10 万に決定した．ベンチマークには ab コマンドを利用した．本実験では，コンテナの負荷に応じてコンテナ追加を，起動済みのコンテナが自身で CoreAPI を介してスケール処理を実施する処理は未実装であるため，コンテナ追加のための API へのリクエストは手動で実施した．

ベンチマークでは，1 秒間のレスポンスタイムの平均値

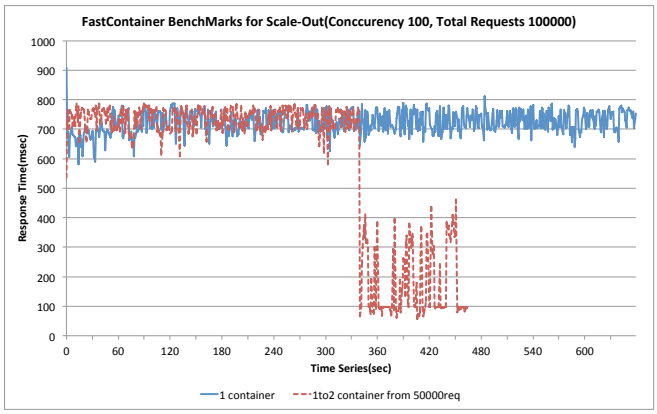

図 3　FastContainer のオートスケールアウト
Fig. 3　FastContainer Auto-Scale-Out.

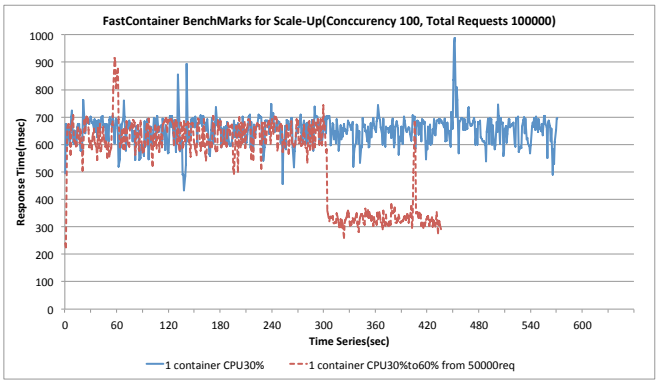

図 4　FastContainer のオートスケールアップ
Fig. 4　FastContainer Auto-Scale-Up.

を時系列データとして作成しグラフ化した．続いて，同様のベンチマークを実施し，処理リクエスト数が 5 万を超えた段階で，スケールアウト型とスケールアップ型それぞれの負荷対応を実施し，即時スケール処理が実施され，レスポンスタイムが短くなるかどうかを確認した．**図 3** にスケールアウト型，**図 4** にスケールアップ型の実験結果を示す．

図 3 の 1 container で示されるグラフは 1 コンテナに対するベンチマークの結果を示している．ベンチマークの間は，制限された最大 CPU 使用率 30%を常に使い切っている状態となっており，その状態で概ねグラフの通りレスポンスタイムは 700msec 前後程度になっている．また，これ以上同時接続数を増やすと，リクエストの処理に失敗する数が増えるため，この値が 1 コンテナがリクエストの処理に失敗することなく処理できる最大のレスポンスタイムである．

次に，横軸 340 秒あたりから下降する 1to2 container で示されるグラフは，5 万リクエストに到達した横軸で 330 秒の時点でオートスケールアウトを行った場合の結果である．グラフからも，コンテナが数秒で立ち上がることの高速性と，オートスケールアウトの処理がリクエストに対してリアクティブに起動することにより，迅速にスケールア

ウトし，リクエストを取りこぼすことなくレスポンスタイムが半分程度に短くなり，高速に処理できていることがわかる．また，レスポンスタイムが短くなったことにより，1コンテナの場合は 10 万リクエストを処理するのに 720 秒程度かかっていたが，グラフのように 470 秒程度で 10 万リクエストの処理が完了していることがわかる．

図 4 の 1 container は，図 3 の 1 container と同様に，スケール処理を何もしない場合のグラフであり，横軸 300 秒あたりから下降し始めている 1 container CPU30%to60% のグラフは，5 万リクエストに到達した横軸 300 秒の時点で，Haconiwa によって即時 CPU の最大使用率を 60%までに引き上げた場合のレスポンスタイムの遷移を示している．このグラフからも，リクエスト処理が 5 万リクエストまで到達した横軸 300 秒の時点から，大きなレスポンスタイムの遅延やレスポンス処理の失敗などなく，即時スケールアップ処理が行えていることが分かる．

5. おわりに

本研究では，Web ホスティングサービスにおいて，サービス利用者に専門的な知識を要求することなく，アクセス集中時には，HTTP リクエスト単位で迅速にコンテナで構成されたユーザ環境がオートスケールできる，コンテナ管理アーキテクチャの FastContainer を提案した．FastContainer アーキテクチャでは，HTTP リクエスト毎に，コンテナ上の Web アプリケーションの起動，起動継続時間，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．これによって，Web アプリケーションの負荷や性能劣化をリクエスト単位で検知し，オートスケールまでの処理を大幅に短縮して，迅速にオートスケールが可能となる．また，コンテナが一定期間で破棄されることによりリソース効率を高め，ライブラリが更新された場合には新しい状態へ更新される頻度も高くなる．

今後の展望として，さらに FastContainer アーキテクチャのシステム設計と実装を進めていき，実サービス展開した場合の実用的な環境における評価を継続的に行っていく．現在注目されているサーバレスアーキテクチャ[*11]との関係性も考慮していく必要がある．また，コンテナが収容される複数のホスト OS の集合体を一つの OS あるいはリソースプールと捉えた場合，コンテナはホスト OS のカーネルを共有する一種のスレッドと見立てることができ，FastContainer アーキテクチャに基いて動作する Haconiwa はスレッドを制御するシステムコール層であると見立てることができる．その方針に基づき，Web アプリケーション実行基盤の分散化に取り組んでいく予定である．

[*11] https://techcrunch.com/2015/11/24/aws-lamda-makes-serverless-applications-a-reality/

参考文献

[1] Amazon Web Services, https://aws.amazon.com/.
[2] Amazon Web Services: Auto Scaling, https://aws.amazon.com/autoscaling/.
[3] Che J, Shi C, Yu Y, Lin W, A Synthetical Performance Evaluation of Openvz, Xen and KVM, IEEE Asia Pacific Services Computing Conference (APSCC), pp. 587-594, December 2010.
[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] Mietzner R, Metzger A, Leymann F, Pohl K, Variability Modeling to Support Customization and Deployment of Multi-tenant-aware Software as a Service Applications, the 2009 ICSE Workshop on Principles of Engineering Service Oriented Systems, pp. 18-25, May 2009.
[8] 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.
[9] 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.
[10] Rosen R, Resource Management: Linux Kernel Namespaces and cgroups, Haifux, May 2013.
[11] 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.
[12] Soltesz S, Ptzl H, Fiuczynski M E, Bavier A, Peterson L, Container-based Operating System Virtualization: A Scalable, High-performance Alternative to Hypervisors, ACM SIGOPS Operating Systems Review, Vol. 41, No. 3, pp. 275-287, March 2007.
[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, 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).
[15] 松本亮介, 岡部寿男, リクエスト単位で仮想的にコンピュータ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, 情報処理学会研究報告 Vol.2013-IOT-23, No.4, 2013 年 9 月.