Vol.2018-IOT-40 No.2
2018/3/5

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

笠原 義晃[1,a)]　松本 亮介[2]　近藤 宇智朗[2]　小田 知央[2]　嶋吉 隆夫[1]　金子 晃介[3]　岡村 耕二[1]

概要：インターネットを介して多種多様なサービスが提供されるようになり，そのサービス基盤となるホスティングやクラウドサービスには高効率化，耐障害性，負荷変動への耐性，柔軟性，セキュリティなどさまざまな要件が求められている．本研究では，これらの問題を解決するため FastContainer と呼ぶ軽量コンテナに基づくシステムアーキテクチャの改良を進めており，特にオートスケーリングのために必要な状況検知やリソーススケジューリング機能の研究開発のため，パブリッククラウド上にテスト環境を構築している．本論文では FastContainer の概要とテスト環境の詳細，現状と今後の課題について述べる．

Flexible Hosting/Cloud Platform Based on Light-Weight Containers and its Evaluation and Stress Test Environment

Yoshiaki Kasahara[1,a)]　Ryosuke Matsumoto[2]　Uchio Kondo[2]　Tomohisa Oda[2]
Takao Shimayoshi[1]　Kosuke Kaneko[3]　Koji Okamura[1]

Abstract: As basic components of ever-expanding internet services, hosting services and cloud computing platforms are expected to satisfy various requirements, such as efficiency, fault-tolerance, resiliency against high load, flexibility, and security. We are trying to solve these problems with a light-weight container based architecture (called FastContainer). To improve FastContainer (especially its auto-scaling function), we are creating an environment for performance and stress tests on a public cloud platform. In this paper, we describe the overview of FastContainer and details of the test environment, its current status, and future plans.

1. はじめに

ウェブホスティングやクラウドサービス [1] の一般化・大衆化により，だれでも簡単にインターネット上にサーバを持つことができるようになった．また，オープンソースソフトウェアの活発な開発と利用拡大，関連する技術情報や利用ノウハウの透明化と共有が進み，個人レベルでインターネットを介した情報発信や多様なアイデアによる新規サービスの構築と提供が可能となっている．大学や研究所などにおいても，研究成果を社会に還元するため積極的な情報発信が求められているなど，教育研究のために情報発信する機会は多い．情報発信や情報サービス提供手段としてはウェブサービスを利用する物が多く，その基盤としてのウェブホスティングやクラウドサービスには高効率化，耐障害性，負荷変動への耐性，柔軟性，セキュリティなどさまざまな要件が求められている．

ハードウェアやネットワークの高性能化と低価格化により，個人や小規模な企業・組織による情報発信やサービス提供の場合，平常時には単一の物理サーバでは能力が余るようになり，複数のユーザ環境が単一のサーバに集約されるようになった．現在では，セキュリティの担保とリソース管理のため，複数のユーザ環境をセキュアかつ安定的に単一のサーバ上に集約できるような，OS の仮想化技術が

[1] 九州大学 情報基盤研究開発センター
Research Institute for Information Technology, Kyushu University
[2] 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
[3] 九州大学 サイバーセキュリティセンター
Cybersecurity Center, Kyushu University
[a)] kasahara@nc.kyushu-u.ac.jp

活用されている [2]．ウェブサービスのようなあまりハードウェアの構成に依存しない限定的な機能を集約させる場合，ハードウェアを含む仮想化は必要ないため，プロセス単位でユーザ領域を隔離・制御するコンテナ型の仮想化技術が活用されている [3]．コンテナ型の仮想化は，ハードウェアを含む仮想化よりオーバーヘッドが少ないことからサービスの収容効率を高めることができ，また仮想環境の起動も高速である．

OS の仮想化技術による物理リソースの効率的利用は，サービスの低価格化と普及に寄与している．一方，個人レベルでの情報発信などにおいても，テレビ番組で取り上げられたり，著名人のブログで紹介されるなどといった外部の要因により，突発的なアクセス集中が発生するような事象が起こる．大企業による情報提供ではアクセスの変動に十分対応できるようなサービス基盤が設計され，突発的なアクセスにも耐えるような対策がなされるだろう．しかし個人レベルではコンテンツは作れても，それを提供するサービス基盤についての知識はない事が多く，一般に提供されているホスティングサービス等を利用するしかない．もちろん負荷変動に対する対策も難しい．低価格ゆえにホスト・ドメインの集積度が高いようなウェブホスティングサービスでは，ホスティングサービス提供側からの人手による個別の技術支援・運用支援も難しい．突発的な負荷変動に対応するにはサービス基盤側にリソース量の迅速な自動調整（オートスケール）の仕組みが必要と考えられる．

またその他の課題として，集積度が高いホスティングサービスでは特に，影響を受けるユーザが多いためにメンテナンス等での停止コストが高くなり，脆弱性対応やバージョンアップの頻度が低くなったり，サービスの安定性・稼働率が低下する問題もある．

このような課題を解決するサービス基盤として，筆者の松本らは FastContainer アーキテクチャを提案している．これは軽量コンテナに基づくホスティング・クラウド基盤で，サービスを提供する単位であるコンテナの起動・複製・終了・リソース割当の変更などの状態変化が高速で効率的という特徴を持つ．この特徴により，外部からのリクエスト単位でコンテナの構成を適応的に決定したり，逆に不要になったコンテナは次にリクエストが来るまで停止してリソースの利用効率を上げるといったことが実現できる．

この FastContainer アーキテクチャは商用ウェブホスティングサービスの基盤としても活用され始めているが、実運用や考察を通してさらなる課題が明らかになってきている．改良のための研究開発は実運用環境でも行なわれているが，高負荷時の挙動確認など内容によっては実運用環境でのテストや評価が難しい場合が考えられる．そのため，現在本研究において評価や改良のためのテスト環境を別途パブリッククラウド上に構築している．本論文では，テスト環境の構築と，FastContainer で解決すべき今後の課題についてまとめる．

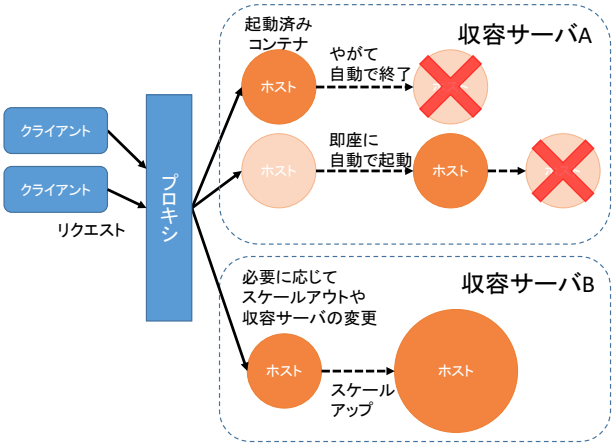

図 1 FastContainer のアーキテクチャ概念図
Fig. 1 Concept Design of FastContainer Architecture

本論文の構成を述べる．2 章では，FastContainer アーキテクチャの概要について述べる．3 章では，現在構築しているテスト環境について述べる．4 章では，FastContainer の研究開発における今後の課題について述べ，5 章でまとめとする．

2. FastContainer アーキテクチャの概要

本章では，本研究で改良を進めている FastContainer アーキテクチャの概要について述べる．なおより詳しい議論や実装の詳細については [4] を参照されたい．

本研究では一般的なウェブ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意識せずとも使えるサービスの実装を目指している．そのためには，基本的なサーバ運用に加えて，突発的なアクセス集中への対応や脆弱性対応やバージョンアップをサービス基盤で支援する必要がある．

特に，突発的なアクセス集中への対応については自動的な負荷対策としてオートスケールの機能が必要だが，一般的な仮想化基盤では以下の課題があると考えている．

(1) インスタンス追加処理が低速
(2) ハードウェアリソースの利用効率が低い
(3) スケーリングすべき状況検知の即時性が低い
(4) 空きリソース確認のためリソーススケジューリング処理の遅延

これを解決するために提案しているのが，コンテナ型仮想化を活用したサービス提供基盤である FastContainer アーキテクチャである．各ユーザの環境は個別のコンテナとして隔離されており，そのユーザの情報発信のためのコンテンツや，サービス提供のためのアプリケーションを提供するホストとして動作する．ただし，一般的なコンテナ基盤やクラウド基盤と違い，ユーザコンテンツなどの永続的な情報はコンテナからは完全に分離されていて，コンテ

ナ外のデータベースや共有ストレージだけに保持されている．

外部から特定のホストに対するリクエストが到着すると，その時点で対応するコンテナが反応的に起動され，リクエストが処理される．起動したコンテナは一定期間はそのまま起動し続け，続くリクエストを通常通り処理するが，その後自動で停止する．あるホストへのリクエストが増えて 1 つのコンテナでは対応できなくなると，コンテナに割り当てられるリソースの上限を動的に上げたり（スケールアップ），コンテナの同時起動数を追加したり（スケールアウト）する．リクエストが減ればリソース割当の上限は戻され，増えたコンテナは自動的に停止する．この動作の概念図を図 1 に示す．ウェブサービスではリクエストにドメイン名が含まれており，それに対応するコンテナの決定が容易であることから実装が比較的しやすい．IP アドレスとポートの組み合わせを各ユーザで変えることにより TCP/UDP に一般化する事も可能と考えている．

FastContainer アーキテクチャでは，コンテナ型仮想化技術を利用することでインスタンスの追加処理を高速化するとともに，オーバーヘッドが小さいというコンテナの特徴と，不要になるとコンテナが停止するという FastContainer の特徴により，(1) と (2) の課題を解決できると考えている．また (3) と (4) の課題については現在研究を進めている．関連する課題のいくつかは 4 章で述べる．

一般的なサービス基盤では，サービスの継続性から一度起動したサーバやサービスプロセスは基本的に止めない事が要求される．一方 FastContainer では，コンテナ内のアプリケーションとデータを分離することで，コンテナを使い捨てにできるにもかかわらず外部からサービスは継続しているように見せている．コンテナは自動で停止・再起動を繰り返すため，ライブラリ等のバージョンアップは自然にユーザ環境反映され，定期的な再起動によりシステムの安定性も向上すると考えられる．コンテナが実行される物理環境も自然に移動でき，メンテナンス性も向上する．

ここで，Serverless アーキテクチャ [5] と FastContainer アーキテクチャとの違いについて述べる．Serverless アーキテクチャは基本的にアプリケーションの運用を基盤側で行わないのに対し，FastContainer アーキテクチャでは，WordPress 等の一般的な CMS を配置・オートスケール可能とし，アプリケーションの運用まで一部基盤側で対応する．また，AWS lambda[6] に類する Serverless アーキテクチャでは利用者による所定のコーディングが必要であるため，基本的にはエンジニアや研究者向けである．一方で，FastContainer アーキテクチャは汎用的な Web アプリケーションを利用でき，アクセス集中時には基盤側でスケーリングできるため，エンジニアの専門知識を持たない個人向けにも提供できる．ただし，FastContainer アーキテクチャにおけるデータベースのスケーリングは，基本的にコンテナへのリソース追加によるスケールアップが前提となる．

Heroku[7] の無料プランは，無料プラン利用によるサービスリソースの使用量を低減するために，リクエスト単位で反応的にインスタンスを起動した後，インスタンスの稼働時間制限やリクエストを処理できないスリープ時間を設けている．Heroku のインスタンス上で状態を持つデータ，例えばブラウザからアップロードしたファイルやセッション情報は，インスタンス再起動時に揮発し，永続化されない．Heroku の有料プランではスリープ時間がないことからも，定期的な再起動を行いながら，基本的にはコンテナを起動させ続けるアプローチによってリクエスト処理を安定的に処理するアーキテクチャと考えられる．

一方で，FastContainer アーキテクチャはリソース効率化に加えて，停止や起動，スケーリング処理などの状態の変化を高速化し，反応的に状態を変化させることによって常にインスタンスを循環させ，変化に強い恒常性を生み出すことが主目的である．また，コンテナに置かれた状態を持つデータは全て永続化され，スケールアウト時であっても，複数のコンテナ間で共有される．さらに，サービスレベルとアクセス数からコンテナのライフタイムを適応的に決める手法の検討や，コンテナの投機的起動により変化に強い恒常性を目指している．

3. テスト環境

本章では，FastContainer アーキテクチャの改良のため現在構築しているテスト環境について述べる．これは「平成 29 年度国立情報学研究所クラウド利活用実証実験」において提供されたクラウド資源を利用している．FastContainer アーキテクチャを利用したウェブホスティング環境を本クラウド資源上に構築し，実運用システムではサービスユーザへの影響などから難しいような高負荷試験などのテストを通して，FastContainer アーキテクチャを改良することを目的としている．

実体は Amazon Web Service (AWS) 上の IaaS インスタンスで，FastContainer 基盤としては，コンテナ収容サーバとして r4.16xlarge 2〜3 台，NFS サービスによる共有ストレージサーバとして r4.2xlarge 2〜3 台，その他アーキテクチャ上必要な機能を搭載するサーバ群として c4.xlarge 20 台前後で構成され，他に負荷試験等に利用する検証用クライアントとして c4.xlarge 10 台程度を利用する．各インスタンスタイプの仕様は表 1 の通りである．

表 1 利用しているインスタンスタイプの仕様
Table 1 Specification of Instance Types

r4.16xlarge	Xeon E5-2686v4 2.3GHz × 64・メモリ 488GB
r4.2xlarge	Xeon E5-2686v4 2.3GHz × 8・メモリ 61GB ストレージ各 2TB(予定)
c4.xlarge	Xeon E5-2666v3 2.9GHz × 4・メモリ 7.5GB

もともと FastContainer アーキテクチャは高集積・高効率なホスティング環境を対象とするため，比較的大規模な計算機資源上でもオートスケールやリソース制御が適切に動作するか，また極端な高負荷下でも期待される性能が出せるかなどをテストする必要がある．このため，インスタンスの数や規模については，実運用に供されている環境と同等の処理能力を持たせることを考慮してサイズ設計している．

また，NII の実証実験で提供された資源は実験期間がは終了すると全資源が利用できなくなる予定となっている．その後も構築情報の再利用により開発やテストを進められるよう，仮想環境全体の構築は Terraform[8]，各インスタンスの環境構築は Chef[9] を利用するなどして自動化し，汎用性を持たせている．この自動環境構築は，洗練させれば組織内で利用するためのプライベートな FastContainer 環境構築に利用できるかもしれない．

なお，クラウドでの自動環境構築機能を提供するシステムとしては AWS であれば CloudFormation[10] も利用できるが，Terraform は特定のクラウド環境に依存しておらず，組織内で利用するにあたり利用環境等に柔軟に対応可能である点から採用した．また，インスタンスごとの環境構築に Chef を利用した理由は、その他の構成管理ツールと比べ中央サーバまたはノードごとへのデプロイが必ずしも必要でないため小規模〜中規模の環境に向いていること、中央サーバを利用した構成にも対応するため将来の大規模化にも耐えられること、オープンソースとして周辺のエコシステムが競合ツールより整っていることなどである．

4. FastContainer 研究開発における今後の課題

FastContainer アーキテクチャは商用ウェブホスティングサービスの基盤としても活用され始めているが、実運用や考察を通してさらなる課題が明らかになってきている．本章では，FastContainer アーキテクチャ研究開発における今後の課題について述べる．

4.1 リソース効率化の実証実験

FastContainer アーキテクチャにより，ハードウェアリソースの利用効率改善が可能と考えているが，定量的な評価ができていない．

今後，本論文で述べたテスト環境を用い，FastContainer によりインスタンスが不要な時に停止・循環させた場合と，従来型の永続的なインスタンスの場合を比較して，どの程度リソースが効率化されるかを定量的に評価・検証していく．現在のテスト環境の構築状況では，利用しているクラウド環境の利用期間の関係で実ユーザを付ける事は困難と考えているが，実際のウェブホスティングサービスのアクセスパターンを用いた実証的な負荷検証などを検討している．

4.2 サービスレベルの適切な導出方法

FastContainer アーキテクチャではインスタンス（コンテナ）の起動処理が高速である事を前提としているが，現状では利用するミドルウェアの種類によっては起動に時間がかかるため，最初のアクセスに対し応答が返るまでに待ち時間が生じる．

例えば，Apache httpd をウェブサーバとした標準的な WordPress 環境では起動に 3 秒程度の時間が必要であった．また初回起動に 10 秒かかるようなウェブアプリを用意した場合コンテナが起動してレスポンスを返すまでに 10 秒かかることになる．

このため，アクセス対象ドメインのコンテナが 1 つも起動していない状態で到着したリクエストは応答までに通常より長い待ち時間が生じ，そのタイミングに当たるとサービスレベル（そのリクエストによってサービスを受ける利用者のユーザ体験）が損なわれることになる．

一度コンテナが起動すると，そのコンテナの起動継続時間の間は通常の応答時間となるため，そのコンテナで提供されるサービスを利用する側全体から見ると，応答時間が長いタイミングに当たる確率によってサービスレベルが変わる事になる．このため，サービスレベルは起動時間，起動継続時間，アクセス頻度の関数になると考えられる．このことから，例えばこれぐらいのアクセス数であれば，起動時間に数秒かかってもサービス全体から見た時にはほとんど無視できる，というようなサービスレベルの関係を定量的に導けるようにすべきである．この関係式が定義できれば，アクセス数に応じて適応的にコンテナ起動時間を設定することも可能だろう．また，サービス・プロバイダ側は起動時間にかかる許容時間も設定できるだろう．

4.3 コンテナ起動時間の影響低減

4.2 節の議論に関連して，コンテナの起動時間はできるだけ短い方がよいと考えられる．コンテナ内部で利用するミドルウェア等のチューニングで改善できる場合もあるが，ユーザに高度な技術を要求しないという要件を満たすためには，サービス基盤側により汎用的な対策が必要である．

起動時間自体が短縮できないとした場合に，それでも起動時間の影響を減らす（アクセスに対する応答を早くする）方法としては

(1) 最低 1 つのコンテナを常に起動
(2) コンテナプロセス実行状態のダンプ・リストア等による起動状態の凍結と再開
(3) コンテナの事前起動からの切り換えによる循環
(4) 過去のアクセス履歴から将来のアクセス頻度を予測し投機的にコンテナを起動

が考えられる．

(1) は非常に単純な方法だがコンテナ循環による利点は失われ，リソース効率も下がるため，サービスとして提供する場合は時間単価を上げるような対応になるだろう．

(2) は CRIU(Checkpoint and Restore in Userspace)[11] 等，実行中のプロセスの状態を凍結してストレージに保存する技術を利用することで実現可能である．ただし，この手法ではプロセスのメモリ状態も含め保存されることからインスタンスの永続性が残ってしまい，FastContainer でコンテナを循環することによる利点が失われる．コンテナ内のライブラリ等を更新するために別途完全なコンテナ再起動のタイミングが必要となるだろう．

(3) では現在起動しているコンテナの起動継続時間が来た時に先に次のコンテナを起動し，リクエストの振り先を変更してから現在のコンテナを停止する．リソースの利用効率は下がるが，コンテナ視点では循環することからそれによる利点は継承できる．次に起動するコンテナを一旦起動してから凍結・停止しておくといった (2) と (3) を組み合わせた実装も可能かもしれない．

(4) は，予測が的中すれば続くアクセスへの応答は高速化されるが，予測が外れれば起動したコンテナにはアクセスが来ず，無駄になってしまう．

いずれの対策もある程度のリソース効率の低下を伴なう一方，サービスレベルの向上に寄与するため，ホスティングサービスユーザの希望により追加の費用負担で利用するような形になるだろう．

4.4 要スケーリング状態の迅速な検知と誤検知の扱い

FastContainer の特徴を生かすためには，スケーリングすべき状況検知のリアルタイム性が重要となる．いくらコンテナの増減・拡縮が高速でも，負荷の増減に対する反応が遅ければ効果が無くなってしまう．

現在運用中の実サービスでは，データ収集は 15 秒単位だが，集計は 3 分毎のためスケーリングのトリガ検知も 3 分の粒度となっていて，リアルタイム性が低い．しかし単純に集計時間を短くするだけでは，システムの負荷とは直接関係がない外部ネットワークの状況などによる外れ値の影響が増加することで誤検知が増加し，コンテナの起動や状態変化が無駄に発生する可能性がある．

これらをうまく適応的に解決するためには，トリガ検知のためのルールベースの設定，長期的な需要予測，短期的な変化点検出，そしてそれぞれのパラメータや閾値の適応的な設定変更のような観点について，サービス・製品の課題や基盤の仕様に合わせて検討する必要がある．

一方で，FastContainer はもともとコンテナを高速で効率的に状態遷移できることに着目していることから，誤検知によりコンテナの状態遷移が短時間で頻発したとしても，システム全体としては許容できるのではないかという観点もある．永続性が高く状態遷移のコストの高い旧来のシステムでは，誤検知を減らすことを正としてきたと考えられるが，FastContainer ではこのようなエラーを許容できるシステムとして改善していく方針もあり得る．4.3 章で述べた投機的起動と関連して，過去のアクセス履歴に基づき投機的に追加コンテナを起動したりリソース割当を追加することも可能かもしれない．

5. おわりに

本論文では，インターネットサービスの基盤としてのホスティングやクラウドサービスに対するさまざまな要求を解決するアーキテクチャとして提案している FastContainer について，その概要と，これを改良する研究開発のために構築しているテスト環境について紹介した．また，FastContainer の今後の課題についてまとめた．

本論文で述べたテスト環境は現在も構築中で，まだ実際に FastContainer の評価や改良に利用できるには至っていない．今後 4 章で述べたような課題について，本テスト環境を活用して開発を進め，より利用者目線で使いやすいホスティングサービスの構築基盤を目指す．

謝辞 本研究で使用したクラウド資源は，平成 29 年度国立情報学研究所クラウド利活用実証実験において提供された．また，本研究は，以下の研究助成を受けている．

松本亮介, 小田知央, 近藤宇智朗, 笠原義晃, 岡村耕二, 嶋吉隆夫, 金子晃介, mruby を利用した軽量コンテナクラウド基盤の研究開発を介した mruby の大規模・高負荷テスト, 2017 年度 Ruby Association 開発助成, 2017 年 10 月．

参考文献

[1] Prodan, R. and Ostermann, S.: A survey and taxonomy of infrastructure as a service and web hosting cloud providers, *2009 10th IEEE/ACM International Conference on Grid Computing*, IEEE, pp. 17–25 (online), DOI: 10.1109/GRID.2009.5353074 (2009).

[2] Che, J., Shi, C., Yu, Y. and Lin, W.: A Synthetical Performance Evaluation of OpenVZ, Xen and KVM, *2010 IEEE Asia-Pacific Services Computing Conference*, IEEE, pp. 587–594 (online), DOI: 10.1109/APSCC.2010.83 (2010).

[3] Soltesz, S., Pötzl, H., Fiuczynski, M. E., Bavier, A. and 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 (online), DOI: 10.1145/1272998.1273025 (2007).

[4] 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，pp. 89–97 (2017).

[5] Roberts, M.: Serverless Architectures, (online), available from ⟨https://martinfowler.com/articles/serverless.html⟩ (accessed 2018-1-31).

[6] Amazon Web Services: AWS Lambda, (online), available from ⟨https://aws.amazon.com/lambda/⟩ (accessed 2018-1-31).

[7] Middleton, N. and Schneeman, R.: *Heroku: Up and*

Running: Effortless Application Deployment and Scaling, " O'Reilly Media, Inc." (2013).
[8] HashiCorp: Terraform by HashiCorp, (online), available from ⟨http://www.teraform.io/⟩ (accessed 2018-1-25).
[9] Chef Software: Chef - Automate IT Infrastructure, (online), available from ⟨http://www.chef.io/chef/⟩ (accessed 2018-1-25).
[10] Amazon Web Services: AWS CloudFormation, (online), available from ⟨https://aws.amazon.com/cloudformation/⟩ (accessed 2018-1-31).
[11] CRIU: CRIU, (online), available from ⟨https://criu.org/Main_Page⟩ (accessed 2018-1-25).