Vol.2022-DPS-190 No.18
Vol.2022-CSEC-96 No.18
2022/3/10

分散 MEC 環境におけるコンテナを利用したクラウド基盤からのオフロード機構

柳澤 舜太郎[1,a)]　渡邊 大記[2,b)]　安森 涼[2,c)]　近藤 賢郎[3,d)]　熊倉 顕[4,e)]　前迫 敬介[4,f)]　張 亮[4,g)]
寺岡 文男[1,h)]

概要：筆者らは，コンテナクラスタとして構成された UE (User Equipment) 上のアプリケーションの一部を MEC (Multi-access Edge Computing) 環境にオフロードするためのアーキテクチャとして ContMEC を提案している．本稿では ContMEC を利用し，クラウド上のコンテナクラスタとして構成されるアプリケーションの一部を MEC 環境にオフロードするアーキテクチャを提案する．このアーキテクチャにより，クライアントとサーバ間の通信遅延が減少し，コアネットワークの消費帯域を減少させることができる．さらに MEC 環境にオフロードされたサーバは他のクライアントからの二次利用も可能になる．評価では提案アーキテクチャを利用し，オフロードサーバの二次利用によりアプリケーション起動時間を 100 分の 1 程度まで削減できること，サンプルアプリケーションの応答時間がクラウドからのオフロードにより 22 ミリ秒程度削減されることなどを確認した．

1. はじめに

現在のアプリケーションはクラウド上にサーバを配置し，ユーザが接続する集中型のクラウドコンピューティングが主流である．しかし，IoT (Internet of Things) 機器の普及等によるインターネットに接続する端末の増加 [1] や，モバイル端末における動画再生の増加 [2] によりコアネットワーク部分のトラフィックが増加している．これにより帯域不足による輻輳が発生する可能性がある．また，クラウドはユーザとの RTT (Round Trip Time) が大きく，低遅延を要求するアプリケーション (e.g., AR (Augmented Reality) / VR (Virtual Reality)) の処理を実行するのは適切ではない．

[1] 慶應義塾大学理工学部
Faculty of Science and Technology, Keio University
[2] 慶應義塾大学大学院理工学研究科
Graduate School of Science and Technology, Keio University
[3] 慶應義塾情報セキュリティインシデント対応チーム
Computer Security Incident Response Team, Keio University
[4] ソフトバンク株式会社
SoftBank Corp.
[a)] yana@inl.ics.keio.ac.jp
[b)] nelio@inl.ics.keio.ac.jp
[c)] moririn@inl.ics.keio.ac.jp
[d)] latte@itc.keio.ac.jp
[e)] ken.kumakura@g.softbank.co.jp
[f)] keisuke.maesako@g.softbank.co.jp
[g)] cho.ryo@g.softbank.co.jp
[h)] tera@keio.jp

これらの課題を解決するため，MEC (Multi-access Edge Compputing) [3] という概念が提唱されている．MEC ではクラウドと比較してユーザにより近い位置にエッジサーバを配置し，処理を転送 (オフロード) する．これによりユーザとの低遅延通信が可能となる．また，MEC の利用によりユーザとクラウドの間での通信が少なくなり，コアネットワーク部分の通信量が削減できる．

近年，高速かつ軽量な仮想化手法であるコンテナ仮想化技術を利用したマイクロサービスアーキテクチャが広く利用されている [4]．マイクロサービスアーキテクチャでは複数のコンテナがクラスタ (コンテナクラスタ) を構成することによりアプリケーションを実行する [5]．マイクロサービスアーキテクチャを MEC と組み合わせることにより，アプリケーションを構成するコンテナの一部分をエッジサーバへ転送することができる．

MEC 環境へのオフロードは図 1 に示すように 2 種類に分類できる．1 つ目は UE (User Equipment) 上のアプリケーションにおける高負荷な処理を MEC 環境へオフロードする UE サイドオフロードであり，2 つ目はクラウド上のサーバの役割の一部分を MEC 環境へオフロードするサーバサイドオフロードである．

分散 MEC 環境においてマイクロサービスアーキテクチャを利用して処理をオフロードするには以下のような課題がある [6][7]．

- MEC 環境において，各エッジ拠点 (複数のエッジサー

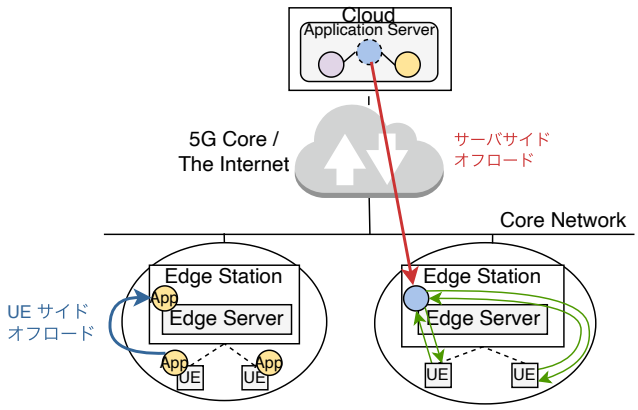

図 1: MEC 環境へのオフロードの種類.

バが存在する拠点) のサーバは小規模であり，利用可能なリソース (e.g., メモリ，CPU) が限定的であると想定される．

- 各エッジ拠点のネットワーク状況を考慮してアプリケーションのオフロード先を決定する必要がある．
- 接続する UE のスケーラビリティに対応する必要がある．

これらの課題を解消することを目的として，筆者らは ContMEC [7] を提案している．ContMEC はマイクロサービスアーキテクチャと MEC を組み合わせたオフロードアーキテクチャであり，以下の特徴がある．

- 分散 MEC 環境において，各エッジ拠点がコンピューティングクラスタ (コンテナが動作する複数のサーバマシンから構成されるクラスタ) を構成し，UE の増加に対するスケーラビリティ問題に対処する．
- クラウドとエッジ拠点で階層的にリソースを配置し，コンピューティングクラスタ間の効率的なリソース管理及びスケーラビリティを実現する．
- 隣接エッジ拠点と計算資源を共有するオーバーラップドコンピューティングクラスタの構成により，エッジ拠点でのリソースの有効活用を図る．

ContMEC では PoC (Proof of Concept) 実装において，コンテナを管理するコンテナオーケストレーションシステムとして K8s (Kubernetes) [8] を利用している．K8s は複数台の Node (マシン) からなる K8s クラスタを構成する．Node には Master Node と Worker Node がある．K8s の最小管理単位は Pod であり，コンテナを管理する．K8s Worker Node が Pod を管理し，K8s Master Node がクラスタ内の K8s Worker Node を管理する．各 Node への指示は K8s Master Node 上にある K8s API Server で行う．

ContMEC は UE サイドオフロードを対象としており，作成したコンテナを複数ユーザで共有することを考慮していない．また，ContMEC を利用したアプリケーションを実行するには UE がエッジ拠点のコンピューティングクラスタに参加する必要がある．そこで本稿では ContMEC を拡張し，これらの課題を解消するサーバサイドオフロードアーキテクチャを設計，実装した．また，設計したアーキテクチャを用いてアプリケーションの起動時間及び応答時間，サーバサイドオフロードによるコアネットワーク部分の通信量削減について評価をした．

2. 関連研究

2.1 クラウドとエッジで連携したマイクロサービスの利用

マイクロサービスアーキテクチャで構成されたアプリケーションにおいて，それぞれの構成要素は REST (Representational State Transfer) / HTTP (HyperText Transfer Protocol) を用いて通信することが多い [9]．そのためエッジ拠点に存在するマイクロサービスがクラウド上のマイクロサービスと通信する場合，マイクロサービスが動作するエッジサーバとクラウド上のサーバが同一のネットワークに所属する必要がある．また，マイクロサービスはサーバの場所 (e.g., エッジ，クラウド) に依存せずに配置可能である必要がある [9]．

これらの問題に対処するため，文献 [9] では K8s を用いたクラウドサーバとエッジノードを連携するアーキテクチャとして，KubeEdge を提案している．KubeEdge ではエッジ，クラウドの双方に存在する KubeBus がクラウドとエッジの通信を中継し，異なるネットワーク間での通信を可能にしている．

また，KubeEdge では K8s API Server と各エッジ拠点に配置されている K8s Worker Node が直接通信するのではなく，K8s API Server が指示した状態を記述したメタデータをエッジとクラウドで同期し，メタデータをもとにエッジ拠点上の K8s Worker Node が K8s API Server から受けた指示を実行する．これにより，クラウドとエッジの間の通信遅延に起因するクラスタ制御への障害 (e.g., ネットワーク状態の変動，限られた通信帯域) を回避できる．

KubeEdge では K8s Master Node をクラウド上に配置しており，エッジノード上の K8s Worker Node を管理している．しかし K8s ではクラスタに所属できる Node 数が最大 5,000 台程度に制限されているため [10]，クラスタに所属可能な K8s Worker Node 数が制限され，エッジ拠点の増加に伴いスケーラビリティの確保が困難になる．

2.2 分散 MEC 環境での複数 K8s クラスタの管理

フォグコンピューティングなどの流行により，地理的に分散した複数のコンピューティングクラスタを管理する要望が高まっている [11]．K8s を利用して複数のコンピューティングクラスタを管理する手法には KubeFed (Kubernetes Federateions) [12] が存在する．しかし KubeFed ではマニュフェストファイルを用いて手動で選択したクラスタにリソースを配置する必要があり，自動的なクラスタの

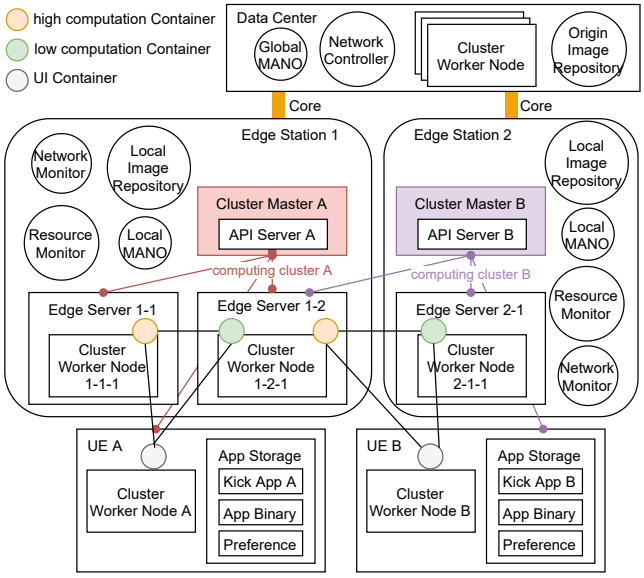

図 2: ContMEC アーキテクチャの構成要素 ([7])

管理は限定的である．また，KubeFed はクラスタでリソースが使用可能であるかの判定もできない [11]．

そこで文献 [11] では地理的分散環境における複数の K8s クラスタを管理するアーキテクチャとして，mck8s を提案している．mck8s では指定したポリシーに基づき自動的に複数のクラスタのスケジューリングが可能である．また，複数のクラスタ間でリソースの配置を自動的に管理し，必要に応じてクラウド上のリソースを提供する．

mck8s では management cluster が各 K8s クラスタを管理する．これにより地理的分散環境において複数のクラスタへのリソース配分を適切に行う．またリソースが不足しポリシーで指定した環境が作成できない場合，必要に応じてクラウド上に K8s クラスタを作成して不足しているリソースを提供する．作成した K8s クラスタは使用後に削除し，不要なリソースの消費を防ぐ．

mck8s ではエッジ拠点でのリソース不足に対し，クラウド上のリソースを共有することにより対処している．しかしクラウドとエッジでは距離が離れているため RTT が大きく，リソースの共有により MEC の利用による低遅延性が損なわれる．

3. ContMEC アーキテクチャの概要

ContMEC アーキテクチャは UE からのオフロード要求をエッジ拠点で処理するアーキテクチャであり，図 2 に示す構成となっている．

3.1 ContMEC アーキテクチャの構成

ContMEC はデータセンタとモバイルコアネットワーク，エッジ拠点から構成される．各エッジ拠点はコンピューティングクラスタを構成している．これによりエッジ拠点やエッジ拠点に接続する UE 数に対し，高いスケーラビリティを実現している．また，クラスタを管理する API Server が各エッジ拠点に配置されていることにより，クラスタを制御するトラフィックがコアネットワークに流れず，コアネットワーク部分の通信量を削減することが可能となっている．ContMEC において，Worker Node は複数のコンピューティングクラスタに所属することが可能である．これにより隣接エッジ拠点間で計算資源を共有できる．また，ContMEC では UE をコンピューティングクラスタに含むことにより，UE 上でもコンテナクラスタの一部の実行を可能にしている．

3.2 MANO (Management and Orchestration) 機構

ContMEC における MANO 機構ではエッジ拠点に Local MANO Manager，クラウド上に Global MANO Manager を配置し，ネットワーク状況に応じてオフロード配置先を管理する．また各エッジ拠点にはリソースモニタ，ネットワークモニタが配置され，リソースモニタがエッジサーバのリソース情報を収集し，ネットワークモニタがエッジ拠点間のネットワーク情報を収集している．

オフロード実行時，Local MANO Manager は 自拠点 (Local MANO Manager が存在するエッジ拠点) のリソースモニタと通信し，オフロード要求に対する自拠点でのコンテナ配置を最適化する．コンテナ配置が自拠点で完結しない場合 Local MANO Manager は Global MANO Manager と通信し，隣接エッジ拠点を含めてコンテナ配置を最適化し，オフロード先エッジ拠点を決定する．

3.3 Image Repository

ContMEC ではアプリケーションを構成するコンテナの情報およびコンテナイメージの取得元として，エッジ拠点上に Local Image Repository，クラウド上に Origin Image Repository を配置している．

オフロード要求を受けコンテナを作成する場合，Local Image Repository にコンテナ構成情報及びコンテナイメージを問い合わせる．エッジ拠点内にコンテナイメージ及び構成情報が存在しない場合は Local Image Repositroy が Origin Image Repository から取得し，キャッシュする．

3.4 ContMEC API Server

ContMEC API Server は UE からのアプリケーションオフロード要求を受け，MANO 機構や Image Repository 等と通信し，オフロード先のエッジ拠点にコンテナを作成する．

Vol.2022-DPS-190 No.18
Vol.2022-CSEC-96 No.18
2022/3/10

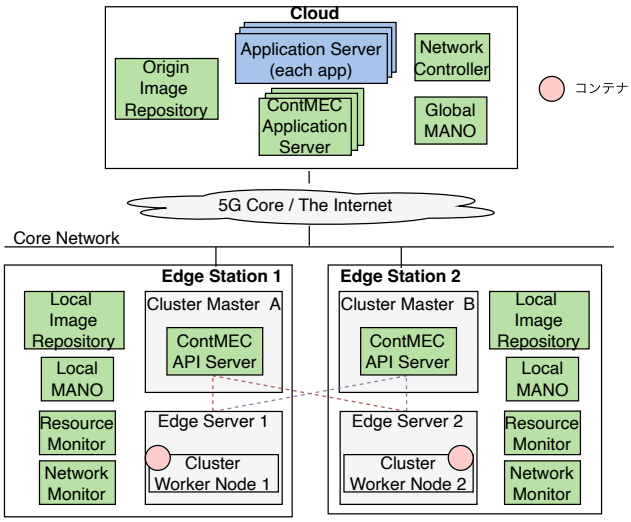

図 3: サーバオフロード型 ContMEC アーキテクチャの構成.

4. サーバオフロード型 ContMEC アーキテクチャ

サーバオフロード型 ContMEC アーキテクチャは ContMEC を拡張することにより第 1 章で述べた課題を解決し，サーバサイドオフロードを実行する．以下では UE サイドオフロードを対象とした従来の ContMEC を UE オフロード型 ContMEC と呼ぶ.

4.1 サーバオフロード型 ContMEC の構成

サーバオフロード型 ContMEC の構成を図 3 に示す．サーバオフロード型 ContMEC はクラウド及びモバイルコアネットワーク，各エッジ拠点から構成される．

4.1.1 UE オフロード型 ContMEC との共通要素

サーバオフロード型 ContMEC では UE オフロード型 ContMEC と同様に各エッジ拠点に ContMEC API Server が配置され，コンピューティングクラスタを形成している．また各エッジ拠点の Worker Node は隣接拠点のコンピューティングクラスタにも所属し，エッジ拠点間で計算資源が共有されている．MANO 機構及び Image Repository に関しても UE オフロード型 ContMEC と共通のものを用いている．

4.1.2 サーバオフロード型 ContMEC での変更要素

サーバオフロード型 ContMEC において，クラウド上にはコンテナクラスタで構成されたクライアントサーバ型アプリケーションのサーバが存在し，各アプリケーションごとに ContMEC Application Server が配置されている．また UE はサーバオフロード型 ContMEC の構成要素には含まれず，一般的なクライアントサーバ型アプリケーションにおけるクライアントと同様にクラウド上のサーバへ接続する.

4.2 ContMEC API Server

サーバオフロード型 ContMEC における ContMEC API Server は，オフロードで作成するコンテナにコンピューティングクラスタ外へのエンドポイントを追加する．また，作成したコンテナの情報 (e.g., コンテナの状態，エンドポイント情報) を ContMEC Application Server へ返答する.

4.3 ContMEC Application Server

ContMEC Application Server は各アプリケーションのサーバからのオフロード要求を管理し，またオフロードで作成したコンテナの情報を ContMEC API Server から取得している．これにより，各エッジ拠点に作成されたコンテナの情報をクラウド側で統一して管理することができる．よってオフロードを要求されたアプリケーションのコンテナがエッジ拠点において起動済みである場合，コンテナを再利用できる.

また，ContMEC Application Server は ContMEC API Server から取得したコンテナへの接続先情報を各アプリケーションのサーバへ送信する．この接続先はコンピューティングクラスタに所属していないクライアントでも接続可能であり，各アプリケーションのサーバはアプリケーションの一部の処理においてクライアントの接続をこの接続先に転送する．これによりコンピューティングクラスタに所属していない UE が ContMEC アーキテクチャで実装されたアプリケーションを利用可能となっている．UE がコンピューティングクラスタに所属しないことにより，コンピューティングクラスタのスケーラビリティが向上する．また，UE をコンピューティングクラスタに所属させるための追加設定が不要となる.

クラウド側でのコンテナ管理及びアプリケーションサーバへの接続先転送により，エッジ拠点でアプリケーションを初回起動したクライアントとは異なるクライアントによるオフロードサーバの利用が可能となっている.

4.4 エッジ拠点での初回アプリケーション起動シーケンス

サーバオフロード型 ContMEC において UE が所属するエッジ拠点にオフロードサーバが起動していない場合，図 4 に示すシーケンスでアプリケーションを起動する.

ユーザは UE 上のクライアントアプリケーション (クライアント) を起動し，各アプリケーションのサーバ (アプリケーションサーバ) に接続する (図 4-(1), (2))．アプリケーションサーバはクライアントのユーザ属性 (e.g., UE を収容するエッジ拠点，ユーザ認証情報) を取得し (図 4-(3))，ContMEC Application Server へオフロード要求を送信する (図 4-(4))．ContMEC Application Server は UE を収容するエッジ拠点上の ContMEC API Server へオフロード要求を送信する (図 4-(5)).

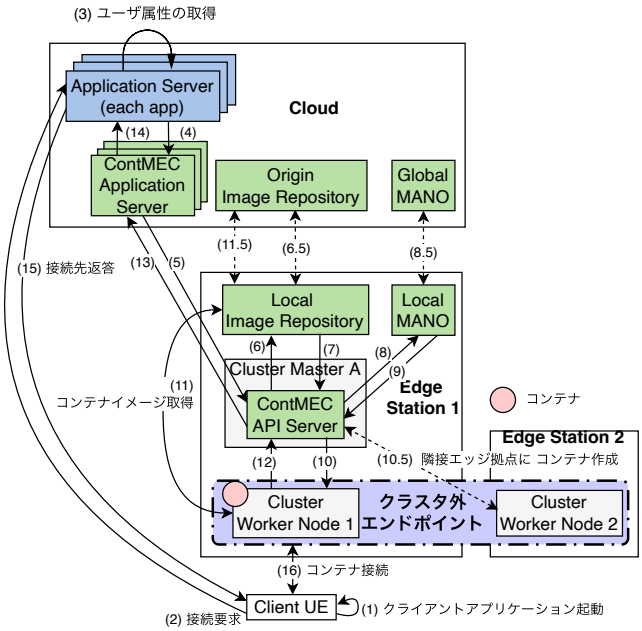

図 4: 初回アプリケーション起動時のシーケンス.

オフロード要求を受信した ContMEC API Server は，自身が所属するエッジ拠点 (自拠点) 上の Local Image Repository から起動するオフロードサーバが利用するコンテナ情報を取得する (図 4-(6), (7))．自拠点内に要求されたコンテナ情報がない場合，クラウド上の Origin Image Repository から取得する (図 4-(6.5))．次に ContMEC API Server は自拠点の Local MANO Manager と通信し，コンテナを配置するエッジ拠点を決定する (図 4-(8)，(9))．Local MANO Manager が自拠点内でコンテナの配置ができないと判断した場合，クラウド上の Global MANO Manager と通信して配置先エッジ拠点を決定する (図 4-(8.5))．

ContMEC API Server はコンテナを配置するエッジ拠点を決定した後，エッジ拠点上の Worker Node へコンテナの作成を要求する (図 4-(10)，(10.5))．次に Worker Node は自拠点にある Local Image Repository へコンテナイメージを要求し，取得する (図 4-(11))．自拠点にコンテナイメージがない場合，クラウド上の Origin Image Repository からコンテナイメージを取得し，自拠点内にキャッシュする (図 4-(11.5))．Worker Node はコンテナイメージを取得後コンテナを作成する．また同時にクラスタ外へのエンドポイントを作成し，コンテナを所属させる．コンテナの作成が完了し接続可能となった後，ContMEC API Server へコンテナ作成完了通知を送信する (図 4-(12))．この通知にはコンテナの情報 (e.g., コンテナの状態，接続先 IP アドレス，ポート番号) が含まれる．

ContMEC API Server はコンテナ情報を ContMEC Application Server へ送信する (図 4-(13))．ContMEC Application Server は 各アプリケーションのサーバにコンテナの接続先を返答する (図 4-(14))．作成したコンテナの情報は保存し，他のクライアントが利用できるようにする．

各アプリケーションのサーバはクライアントへコンテナの接続先を返答し (図 4-(15))，クライアントが接続先情報をもとにコンテナへ接続する (図 4-(16))．

4.5 オフロードサーバが起動済みである場合のアプリケーション起動シーケンス

UE が所属するエッジ拠点においてオフロードサーバが起動済みである場合，アプリケーション起動シーケンスではコンテナ作成のための ContMEC API Server との通信 (図 4-(5) 〜 (13)) が省略される．これはエッジ拠点での初回アプリケーション利用時に，起動したコンテナの情報を ContMEC Application Server が保存している (図 4-(14)) ためである．この情報により UE が収容されるエッジ拠点において，オフロードサーバが起動済みであると判断される．よって ContMEC Application Server は起動済みのオフロードサーバへの接続先情報を各アプリケーションのサーバへ返答し，各アプリケーションのサーバはクライアントへこの接続先情報を返答する．

5. 実装

PoC (Proof of Concept) 実装として Python 3.6.9 を利用して ContMEC Application Server 及び ContMEC API Server を実装した．MANO 機構及び Image Repository に関してはダミーとした．そのためコンテナは静的に配置し，コンテナイメージは取得済みとした．また各アプリケーションのサーバは 1 つのサーバに統合し，ContMEC Application Server も 1 つのサーバのみを利用した．

UE オフロード型 ContMEC では PoC 実装におけるコンテナオーケストレーションシステムとして K8s を利用している．そこでサーバオフロード型 ContMEC においても同様に K8s を利用して実装した．コンピューティングクラスタは K8s を利用して構成し，各エッジ拠点にクラスタを管理する K8s API Server を配置した．ContMEC API Server は各エッジ拠点の K8s API Server と通信し，コンテナの作成，管理及びエンドポイントの作成を行う．エンドポイントは K8s の機能である K8s NodePort Service を利用して作成した．エンドポイントへのコンテナの追加は時間を要するため，コンテナの作成完了後，接続可能と判断するまでの間に 250 ミリ秒の待機時間を挿入した．

ContMEC Application Server，ContMEC API Server 及び各アプリケーションごとのサーバは Python のライブラリである Flask を利用した HTTP (Hyper Text Transfer Protocol) Web サーバで実装した．Image Repository 及び MANO 機構と ContMEC API Server の通信はソケット通信を利用した．

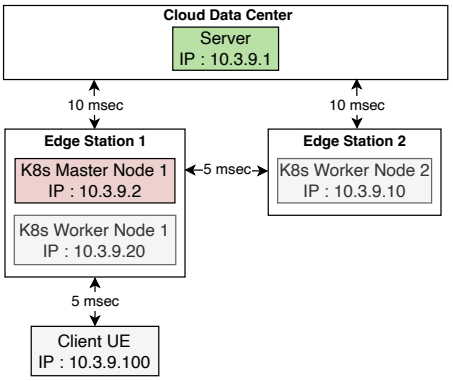

図 5: 評価環境.

表 1: 使用した VM のスペック.

項目	VM	サイズ，内容
OS	全 VM	Ubuntu 18.04 LTS
CPU	全 VM	vCPU × 2
RAM	Server	8 GB
	その他	4 GB
ストレージ	全 VM	32 GB
Kubernetes (Kubectl,	Master Node	1.21.2-00
Kubelet, Kubeadm)	Worker Node	1.22.1-00

6. 評価

評価では UE 上のクライアントアプリケーション (クライアント) 及びクラウド上の各アプリケーションのサーバ (アプリケーションサーバ)，エッジ拠点上のコンテナ (オフロードサーバ) から構成されるアプリケーションを利用した.

6.1 評価環境

評価は図 5 に示した環境で実行した．評価環境は Server, K8s Master Node 1, K8s Worker Node 1, K8s Worker Node 2, Client UE からなる 5 台の VM (Virtual Machine) で構成し，それぞれ図 5 に示す遅延を挿入した． Server VM には ContMEC Application Server 及び各アプリケーションのサーバを配置した．また，Local MANO Manager，Local Image Repository，ContMEC API Server は K8s Master Node 1 VM 上に配置した．Global MANO Manager 及び Origin Image Repository は PoC 実装のため使用していない．評価では表 1 に示すスペックの VM を利用した.

6.2 アプリケーション起動時間

ユーザが UE 上でクライアントを起動してからオフロードサーバへ接続可能となるまでの時間を計測した．オフロードサーバが利用するコンテナの数及びコンテナイメージサイズによるアプリケーション起動時間への影響を評価するため，イメージサイズが 250 MB，500 MB，1 GB のコンテナを 1, 2, 4 個作成し，それぞれ 20 回計測した.

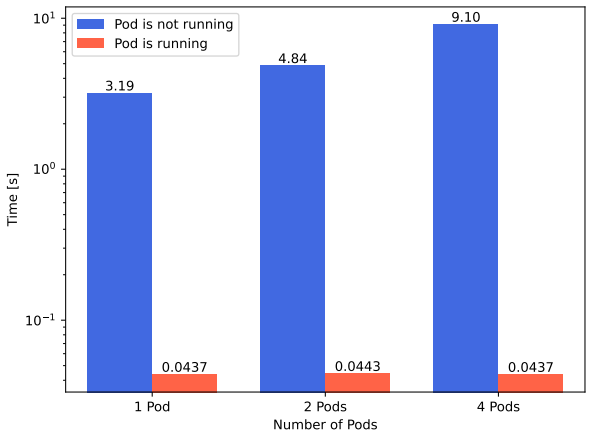

図 6: アプリケーション起動時間 (250 MB コンテナ).

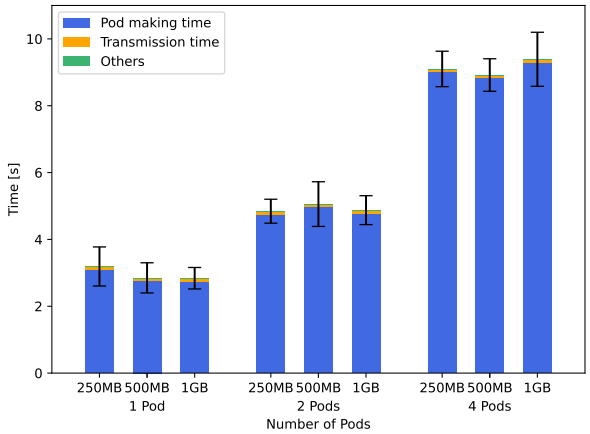

図 7: 初回アプリケーション起動時間.

6.2.1 オフロードサーバの二次利用によるアプリケーション起動時間の削減

コンテナイメージサイズが 250 MB であるオフロードサーバの利用時について，エッジ拠点での初回アプリケーション起動時及びオフロードサーバを二次利用した場合の測定結果を図 6 に示す．図 6 は片対数グラフとなっている．初回アプリケーション起動時では，オフロードサーバが利用するコンテナが 1 つの場合のアプリケーション起動時間が約 3.19 秒程度となり，以降はコンテナ数が増加する毎にアプリケーション起動時間も増加した．一方，オフロードサーバが起動済みである場合のアプリケーション起動時間はコンテナの数によらず約 0.044 秒程度であり，オフロードサーバの二次利用により 100 分の 1 程度までアプリケーション起動時間が削減できていることがわかる.

6.3 コンテナイメージサイズ及びコンテナ数によるアプリケーション起動時間への影響

エッジ拠点での初回アプリケーション起動時の各コンテナイメージサイズにおけるアプリケーション起動時間を図

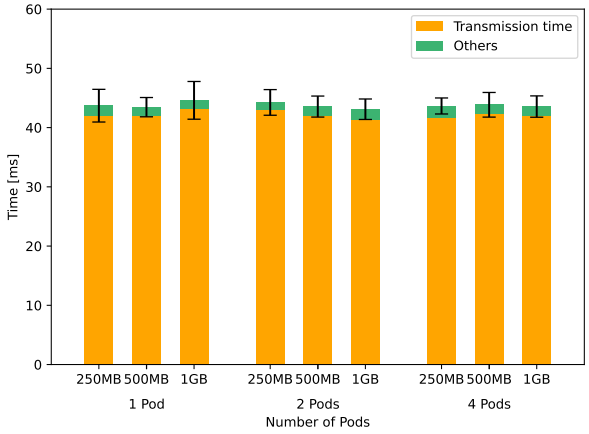

図 8: オフロードサーバ二次利用時のアプリケーション起動時間.

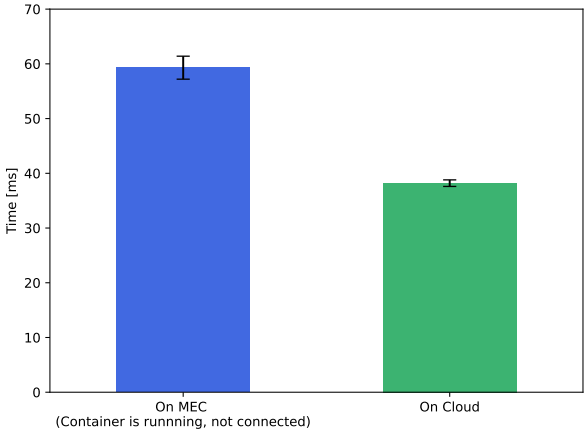

図 9: クライアント初回起動時の応答時間の比較.

7 に示す．アプリケーション起動時間はオフロードサーバが利用するコンテナの個数に応じて線形的に増加した．これはコンテナの作成時間がアプリケーション起動時間において支配的となっているためである．コンテナの作成時間はコンテナが 1 つ増加するごとに平均で 2.04 秒程度増加した．これは K8s が Pod (K8s においてコンテナを管理する単位) を 1 つずつ作成するためであると推測される．

エッジ拠点にオフロードサーバが起動済みである場合の各コンテナイメージサイズにおけるアプリケーション起動時間を図 8 に示す．アプリケーション起動時間はオフロードサーバが利用するコンテナのイメージサイズ及び個数によらず，平均して 43.7 ミリ秒程度となった．また起動時間において RTT が支配的となっており，サーバオフロード型 ContMEC での処理時間が短いことがわかる．

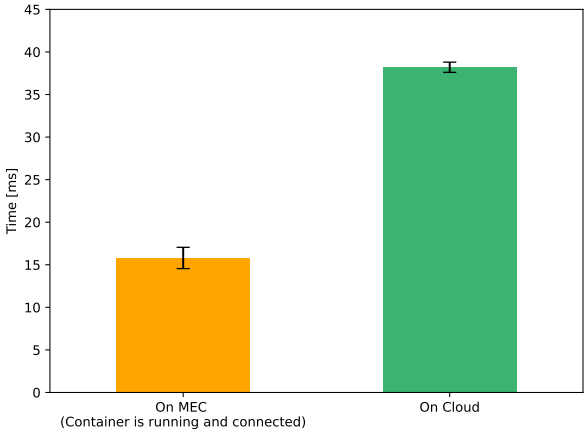

図 10: アプリケーション起動後の応答時間の比較.

6.4 アプリケーション応答時間

本評価ではクライアントサーバ型電卓アプリケーションを利用し，ユーザがクライアントを起動してからクライアントが計算結果を取得するまでの時間を 10 回計測した．オフロードサーバは 1 つのコンテナのみから構成され，既に他のユーザが利用しておりエッジ拠点上に起動済みであるとした．また，クラウド上のアプリケーションサーバは実装にコンテナを利用していない．

クライアント初回起動時とクラウド上のアプリケーションサーバを利用した場合のアプリケーション応答時間の比較を図 9 に示す． オフロードサーバを利用した場合の応答時間は平均 59.3 ミリ秒程度となった．クラウド上のアプリケーションサーバを利用した場合のアプリケーション応答時間は平均 38.2 ミリ秒程度であるため，エッジ拠点にオフロードサーバが起動している場合，初回クライアント起動時にオフロードサーバへの接続先を取得する際のオーバーヘッドは 21 ミリ秒程度であることがわかる．

クライアントが起動されオフロードサーバへ接続済みである場合とクラウド上のアプリケーションサーバを利用した場合のアプリケーション応答時間の比較を図 10 に示す．オフロードサーバを利用した場合のアプリケーション応答時間は平均して約 15.8 ミリ秒であった．初回アプリケーション利用時においてオフロードサーバに接続する場合の起動時間は 59.3 ミリ秒程度であるため，オフロードサーバへの接続先を取得済みであることにより約 44 ミリ秒程度の時間が短縮されている．

クラウド上にサーバを配置した場合のアプリケーション応答時間は平均 38.2 ミリ秒程度であるため，オフロードサーバの利用により約 22 ミリ秒の時間を削減できている．これは図 5 に示したようにクラウドと比較し UE と エッジ拠点との RTT を小さく設定しているためである．1 回の通信につきオフロードサーバとの通信はクラウド上のサーバと比較して 10 ミリ秒程度の時間が削減される．22 ミリ秒の時間削減は，サーバとの接続確立時の TCP 通信における 3 ウェイハンドシェイク通信及びアプリケーションデータ通信による RTT での差であると考えられる．

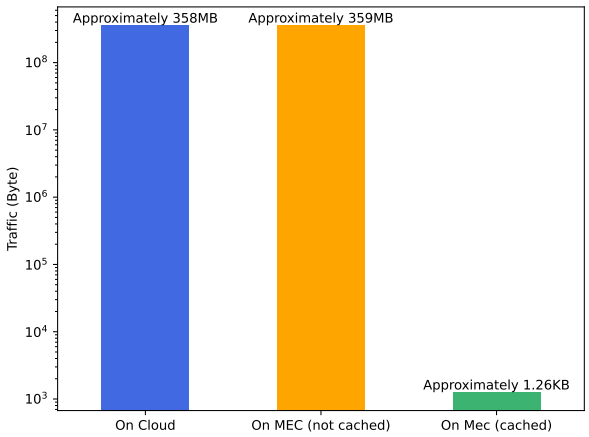

図 11: 動画再生時のコアネットワーク部分の通信量.

6.5 動画再生時の通信量

本評価では動画再生時のコアネットワーク部分の通信量について，ストリーミングサーバをエッジ拠点に配置する場合とクラウドに配置する場合を 10 回計測し，比較した．ストリーミングサーバには動画配信アプリケーションで用いられる MPEG-DASH 方式を利用し，356 MB の動画を再生した．エッジ拠点のストリーミングサーバでは初回動画再生後に動画がキャッシュされ，2 回目以降の動画再生時はコアネットワーク部分の通信量が削減される．

評価結果を図 11 に示す．図 11 は片対数グラフとなっている．エッジ拠点上のサーバを利用し，動画がキャッシュされていない場合の通信量は約 359 MB となり，動画サイズと比較して約 3 MB 増加した．これはパケットのヘッダや制御トラフィックによるものと考えられる．クラウド上のサーバを利用して動画を再生した場合の通信量に関しても約 358 MB となり，同程度の通信量となった．

一方，エッジ拠点のサーバに動画がキャッシュされている場合，通信量は約 1260 Byte となった．これはクライアントがアプリケーションサーバからエッジ拠点上のオフロードサーバへの接続先を取得する際の制御トラフィックにおける通信量であると推測される．

以上よりエッジ拠点上のオフロードサーバへのデータのキャッシュにより，クラウド上にサーバを配置する場合と比較してコアネットワーク部分の通信量を大幅に削減できることがわかる．

7. おわりに

本稿では ContMEC アーキテクチャを拡張し，サーバオフロード型 ContMEC アーキテクチャを提案，実装した．このアーキテクチャによりエッジ拠点にオフロードしたコンテナの再利用が可能となり，またコンピューティングクラスタに所属していない UE がエッジ拠点上のコンテナへ接続可能となった．これによりエッジ拠点上のオフロードサーバを複数のユーザが利用可能となった．

評価ではエッジ拠点での初回アプリケーション起動時間は利用するコンテナの数が 1 つ増加するごとに約 2.04 秒程度増加することを確認した．また，オフロードサーバの二次利用によりアプリケーション起動時間が 100 分の 1 程度まで短縮されること，オフロードによりサンプルアプリケーションでの応答時間がクラウドでの実行と比較し 22 ミリ秒程度短縮されることも確認した．さらに，動画再生時のコアネットワーク部分の通信量がクラウド上にサーバを配置した場合の 358 MB から，エッジ拠点へのキャッシュにより 1260 Byte まで削減されることも確認した．

今後は MANO 機構を実装し，複数エッジ拠点におけるコンテナ配置に関して評価を取ることを目標とする．

参考文献

[1] Chiang, M. and Zhang, T.: Fog and IoT: An Overview of Research Opportunities, *IEEE Internet of Things Journal*, Vol. 3, No. 6, pp. 854–864 (2016).
[2] Yang, S.-R., Tseng, Y.-J., Huang, C.-C. and Lin, W.-C.: Multi-Access Edge Computing Enhanced Video Streaming: Proof-of-Concept Implementation and Prediction/QoE Models, *IEEE Transactions on Vehicular Technology*, Vol. 68, No. 2, pp. 1888–1902 (2019).
[3] ETSI: Multi-access Edge Computing (MEC);Framework and Reference Architecture, *GS MEC 003 v2.2.1* (2020).
[4] Florio, L. and Nitto, E. D.: Gru: An Approach to Introduce Decentralized Autonomic Behavior in Microservices Architectures, *In Proceedings of 2016 IEEE International Conference on Autonomic Computing (ICAC)*, pp. 357–362 (2016).
[5] Nguyen, C., Mehta, A., Klein, C. and Elmroth, E.: Why Cloud Applications Are Not Ready for the Edge (Yet), *In Proceedings of the 4th ACM/IEEE Symposium on Edge Computing*, pp. 250–263 (2019).
[6] 稲垣 勇佑, 渡邊 大記, 安森 涼, 近藤 賢郎, 熊倉 顕, 前迫 敬介, 張 亮, 寺岡文男: 地理的分散環境を想定した MEC におけるオフローディング機構，情報処理学会研究報告, Vol. 2021-DPS-186, No. 22, pp. 1–8 (2021).
[7] Watanabe, H., Yasumori, R., Kondo, T., Kumakura, K., Maesako, K., Zhang, L., Inagaki, Y. and Teraoka, F.: ContMEC: An Architecture of Multi-access Edge Computing for Offloading Container-Based Mobile Applications, *In Proceedings of 2022 IEEE International Conference on Communications (ICC)*, pp. 1–7 (2022). (to appear).
[8] Cloud Native Computing Foundation: Kubernetes. `https://kubernetes.io`.
[9] Xiong, Y., Sun, Y., Xing, L. and Huang, Y.: Extend Cloud to Edge with KubeEdge, *In Proceedings of 2018 IEEE/ACM Symposium on Edge Computing (SEC)*, pp. 373–377 (2018).
[10] Kubernetes: 大規模クラスターの構築. `https://kubernetes.io/ja/docs/setup/best-practices/cluster-large/`.
[11] Tamiru, M. A., Pierre, G., Tordsson, J. and Elmroth, E.: mck8s: An orchestration platform for geo-distributed multi-cluster environments, *In Proceedings of ICCCN 2021 - 30th International Conference on Computer Communications and Networks*, pp. 1–12 (2021).
[12] Kubernetes SIGs: KubeFed. `https://github.com/kubernetes-sigs/kubefed`.