Vol.2021-DPS-186 No.22
Vol.2021-CSEC-92 No.22
2021/3/15

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

稲垣 勇佑[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：地理的分散した MEC (Multi-access Edge Computing) 環境におけるコンテナ配置を実現するには，遅延・帯域といったネットワーク情報を考慮することが重要である．本稿では，コンテナオーケストレーションシステムの 1 つである Kubernetes (K8s) を使用した MEC オフローディング機構，kube-mec アーキテクチャを提案する．kube-mec アーキテクチャにより，地理的分散な環境におけるコンテナオーケストレーションが，K8s に変更を加えることなく実現される．また，実装したオフローディング機構が機能していることを，評価を取り確認した．

1. はじめに

IoT (Internet of Things) の浸透により，インターネットに接続されるデバイスの数は大きく増加している [1]．そのため現在の一極集中型のクラウドコンピューティングではクラウド周辺のトラフィックが過度に増加し，ネットワークを逼迫することが懸念されている．クラウドコンピューティングのこのような問題点を解決するために，近年 MEC (Multi-access Edge Computing)[2] という，モバイル端末から計算量が大きい処理をユーザ近傍 (エッジ) のコンピューティング資源に実行させる (オフロードする) 概念が浸透している．MEC により，低遅延かつモバイル端末の負荷軽減が期待できる．

また，近年ではコンテナ技術が台頭している．コンテナはプロセスと同等に動作が軽量でメモリを少量しか使用せず，起動も短時間で完了するという利点がある．そのため，本稿では MEC におけるオフローディング機構をコンテナベースで実施することを考えている．また通常，単一のアプリケーションは多数のコンテナが連携して動作しているため，コンテナを集中的に管理するためのツールであるコンテナオーケストレーションシステムが必要になる．コンテナオーケストレーションシステムは主にコンテナの作成と配置，スケーリング，状態監視を自動的に行う．本稿ではコンテナオーケストレーションシステムのデファクトスタンダードである Kubernetes (K8s) を採用している．K8s では，1 つ以上のコンテナの集合体を Pod とし，最小の管理対象として扱う．また，データセンタなどのコンピュータクラスタ上に VM (Virtual Machine) として Master Node と Worker Node を配置し，1 つの Master Node と 1 つ以上の Worker Node で K8s クラスタを構成する．K8s クラスタ内では高可用性や負荷分散を考慮し，Master Node が Worker Node に Pod を配置する．

また本稿では，図 1 のようなモバイルキャリア網での MEC オフローディング機構を想定している．Core network には多数の基地局 (Base Station) が配置され，複数の基地局に対して拠点 (MEC Base) が設けられる．図は，UE (User Equipment: 端末機器) が拠点にアプリのオフロードをしている様子を表している．クラウドサーバは大規模データセンタであり多くの資源を保持しているが，UE との通信は高遅延になる．また本環境では，単一ネットワークオペレータが地理的分散した MEC 拠点を保持，管理している．これらの MEC 拠点の中には UE が計算処理をオフロードするための MEC サーバが 1 台以上収容され，UE は最寄りの基地局に接続しオフロード処理を開始

[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)] kuwaro@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

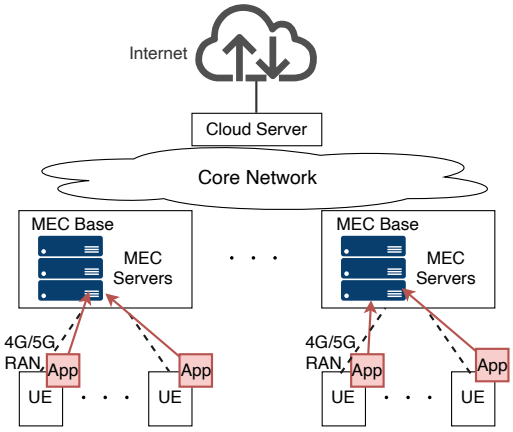

図 1: 本稿で想定している地理的分散な MEC 環境.

する．

従来の技術を利用して，上記のような地理的分散環境で MEC を実現するには以下の課題がある．

- MEC サーバは小規模であるため一般的に資源に制限がある．分散協調を実現するために，隣接する MEC 拠点の MEC サーバへの資源融通可能なアーキテクチャが必要である．
- K8s は基本的には集中環境を想定しており，地理的に分散した環境を考慮していない．そのため，クラウドに Master Node を設置してエッジに Worker Node を配置する，といった形で運用を進めるのは困難である．
- K8s は Pod の配置先ノードを決定する際に K8s クラスタ内の Worker Node の負荷を考慮するが，ネットワークに関する情報を考慮しないため通信遅延が大きくなる Worker Node に Pod を配置してしまう可能性がある．そのため，ネットワークに関する情報も考慮して Pod 配置先ノードを決定する機構が必要である．
- K8s に変更を加えてしまうと，K8s がアップデートされる度に変更部分のアップデートも必要になる．また，変更を加えずに想定環境に適した改良をすることで改良部分のデプロイが容易になる．そのため，K8s 自身に変更を加えずに想定環境の MEC へ K8s を適用する必要がある．

本稿では K8s を用いた新アーキテクチャを考案して上記の課題を解決し，想定している MEC 環境を実現することを目的とする．また，その新アーキテクチャに対して，アプリの起動・実行・終了時間，アプリ実行中の UE 上での CPU/RAM の使用量の測定など様々な面で評価する．

2. 関連技術・研究

2.1 ネットワーク資源を考慮したコンテナスケジューリング

エッジコンピューティング環境において，遅延の影響を受けやすいサービスに適切な資源割り当てスケジューリングを実装することはとても重要なことである [3][4].

そのため，文献 [5] では，スマートシティ展開におけるコンテナアプリケーションのネットワーク情報対応スケジューリングアプローチを提案している．具体的には，Worker Node に対し CPU, RAM などのリソースの容量や，事前に測定した Master Node からの RTT 値に応じたラベルを付与することにより，K8s のスケジューリング動作を微調整できる．これによりスケジューリングにおいて遅延制約，つまりネットワーク情報を取り扱うことが可能である．

さらに文献 [5] では，各 Worker Node に割り当てられた RTT ラベルを利用して，Pod の構成ファイルで指定されたロケーションにおいてアプリケーションの展開先として最適なノードを決定するための NAS (Network Aware Scheduler) を実装している．結果として，NAS アルゴリズムにより K8s-native なデフォルトのスケジューリングアルゴリズムに対してネットワーク遅延を約 80 % 削減している．

しかし，NAS において使用している RTT ラベルは運用前に測定した静的なネットワーク情報であり，アプリケーションの展開中に RTT に変動があった場合でも，動的に対応することができない．また，K8s-native が NAS を呼び出すようにスケジューリングを上書きしているため，K8s を拡張している．また，Master Node を 1 つしか使用していないためスケーラビリティを担保できる Worker Node の個数が少なく，多くの Worker Node・UE を使用できない．それゆえに，地理的に分散している環境には不適切である．

2.2 レイテンシに配慮したアプリケーションの計算とデプロイの管理

産業自動化におけるエッジコンピューティングは，センシングデバイス，コンピューティングデバイス，及び作動デバイス間で転送されるデータ量と制御の遅延を最小限に抑える必要がある [6].

そのため文献 [7] では，K8s を使用してレイテンシに配慮した産業用アプリケーションの計算とデプロイを管理するアーキテクチャを提案している．適切な基準を用いて局所的に最適なケースを連続して選択する貪欲法アルゴリズムを採用することで，ノード間の通信遅延やネットワークに流れるデータ量を最小限にする．文献 [7] では，元々既存研究 [8] が提案しているコンテナ割り当てのための貪欲法アルゴリズムをより産業用自動化に適するように改善している．具体的には，すでに稼働している Worker Node に空き容量がある場合は同一ノードにコンテナを配置し，空き容量がない場合は任意の Worker Node にコンテナを配置する，というものである．結果として，K8s に大きな変更を加えることなく，リソース割り当ての最適化問題を動的に解決し，分散しているコンテナアプリケーションを

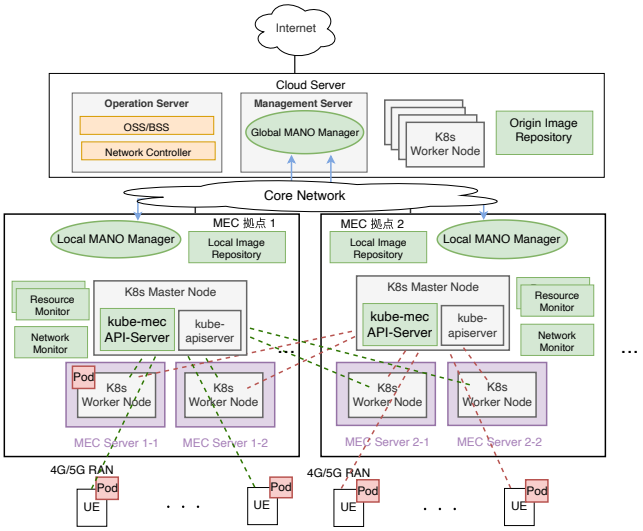

図 2: kube-mec アーキテクチャの全体図.

産業自動化システムネットワークに展開することを実現している.

しかし，文献 [7] では同一の Worker Node で処理できない場合は任意の Worker Node にコンテナを割り当てるため，任意に抽出された Worker Node がレイテンシを考慮したときに最適であるという保証はない．元々の K8s-native なコンテナスケジューリングアルゴリズムよりも都合の悪い Worker Node を選択してしまう可能性もある．そのため，実際の提案がレイテンシに適切に配慮したアプローチであるとは言えない．よって，地理的に分散した環境にこのアプローチを採用するのは適切でない.

また，上記研究は想定環境をプライベートな環境内に閉じてしまっている．キャリア網においての MEC 機構はあらゆるユーザが利用するため，エッジデバイスが MEC サーバの資源にアクセスする際には認証認可が必要になる.

3. kube-mec アーキテクチャ

本章では，第 1 章で述べた課題を解決するために考案した kube-mec アーキテクチャの全体像とそれぞれのコンポーネントを説明する.

図 2 は，kube-mec アーキテクチャの全体図である．単一の携帯電話網がインターネットに接続している環境を想定する．図の一番上がクラウドであり，Core Network を通じて各 MEC 拠点と繋がっている．各 MEC 拠点は 1 台以上の MEC サーバを保持する．MEC サーバは物理マシンであり，MEC 拠点内の Master Node と Worker Node は一般的に VM として稼働している.

また，オレンジ色のコンポーネントは 5G-NEF (Network Exposure Function)[9] という，5GC (5th Generation Core network) を構成するネットワーク機能を外部アプリに公開するためのコンポーネントである．5GC は 5G 無線を収容するモバイルコアネットワークシステムを指す．緑色のコンポーネントは K8s には存在せず，kube-mec が導入したコンポーネントである．MEC 拠点の構成や主要なコンポーネントについて詳細に説明する.

3.1 kube-mec アーキテクチャの MEC における構成

kube-mec においては，MEC 拠点ごとに Master Node を用意するマルチマスター構成となっている．これにより，K8s クラスタを各 MEC 拠点内で一つずつ構築し，UE からのオフロード要求を MEC 拠点ごとに処理することが可能になり，スケーラブルな Pod 配置を行うことが可能になる.

また，1 台の Worker Node が複数の Master Node に属する構成となっている．これにより，それぞれの MEC 拠点の Master Node が隣接 MEC 拠点に属する MEC サーバに Pod を配置することが可能になる．すなわち 1 台の Worker Node を複数の K8s クラスタが共用することができ，資源の利用効率を向上させる．また，複数 MEC 拠点に Pod を配置する場合でも，オフロード要求を受けた MEC 拠点からなるべく近い拠点に Pod を配置することが可能になり低遅延を実現する.

また，UE は MEC 拠点における K8s クラスタに Worker Node として参加する．これにより，UE の K8s 的な資源情報・アプリの稼働状況などを透過的に監視することが可能になる.

3.2 Image Repository 機構

kube-mec においては，クラウドと各 MEC 拠点に kube-mec のアプリのコンテナイメージを保持するレポジトリである *Image Repository* を配置する．クラウドには，全ての kube-mec 対応アプリのコンテナイメージを保持するレポジトリである *Origin Image Repository* が，各 MEC 拠点には *Local Image Repository* が設置され，ファイルサーバの役割を果たす．Worker Node からのコンテナイメージ要求を受けたとき，Local Image Repository 内にコンテナイメージが存在する場合はそのままファイルの URL を返す．存在しない場合は Local Image Repository が Origin Image Repository に該当コンテナイメージを要求する.

3.3 MANO (Management and Network Orchestration) 機構

想定環境における MEC の要求事項の 1 つは，ネットワーク情報を含めて Pod の配置先を決定することである．それを実現するために，kube-mec では独自の MANO (Management and Network Orchestration) 機構を採用する．MANO は MEC 拠点内の Worker Node の CPU/RAM の使用率やネットワーク情報を基に，Pod の配置先として最良な MEC サーバを算出することができる.

各 MEC 拠点内には，MEC 拠点内における Pod 配置

を算出する *Local MANO Manager* が設置されている．MEC 拠点内での Pod 配置が難しい場合は，Local MANO Manager はクラウドの *MANO Manager* に問い合わせる．MANO Manager は最適な Pod 配置先 MEC 拠点を算出し，Local MANO Manager に返答する．

算出に必要な資源・ネットワーク情報は，定期的に *Resource Monitor*，*Network Monitor* から収集する．これらの Monitor は MEC 拠点に設置されている．Resource Monitor は MEC サーバの物理的な計算資源情報 (空き仮想 CPU 数, 空き RAM 容量, 空き Storage 容量) を定期的に収集する．Network Monitor は MEC 拠点間のネットワーク情報 (RTT，使用帯域) を定期的に収集する．Network Monitor には 5GC 機能を組み込むことで実現を目指す．

3.4 kube-mec API Server

kube-mec に K8s を適用する上で, 第 1 章で述べた検討事項を実現するため, また独自実装する MANO との情報のやりとりをする機能を追加する必要があるため kube-mec では UE と K8s-native な API Server である kube-apiserver を仲介する *kube-mec API Server* を導入する．kube-mec において，UE はアプリの起動・終了要求を送信するとき，kube-mec における独自 API である *kube-mec API* を使用する．kube-mec API Server は，UE から要求を受信した後に独自の MANO や Local Image Repository との通信を行ったあと，諸々の要求を K8s-native な API に翻訳し kube-apiserver に要求を送信する．この kube-mec API Server 導入によって，前述の要求事項は実現される．

また，UE や MANO との通信処理を kube-mec API Server が全て請け負い，それらから受け取った情報を全て K8s-native な API に翻訳して kube-apiserver に転送する．これにより，kube-mec が導入した kube-mec API Server や MANO は全て K8s のサポート範囲外で稼働する．よって，K8s に変更を加えることなく，MANO との通信も実現することができるようになる．本稿では kube-mec API Server の実装，評価を取ることを主な目的としている．

3.5 UE 上のコンポーネント

UE には，*Kick App* と *App Binary* というコンポーネントが置かれている．Kick App は UE から kube-mec アプリの起動・終了要求を送信するための機能である．アプリケーション実行中の操作は Kick App 上では行わず，起動要求承認後に kube-apiserver が UE 上に，ユーザとの入出力を担う UI Pod を UE 上に起動する．これにより，アプリを構成する全ての Pod の起動・終了操作の権限を kube-mec 運用側が保持することが可能になる．

App Binary は UE 上で起動するコンテナのイメージである．アプリをインストールする際にアプリに必要なコンテナイメージもダウンロードするため，キャッシュミスが起きることはない．

3.6 アプリ起動手順

本節では，kube-mec におけるアプリの起動・終了シーケンスを，複数拠点間に跨って Pod 配置をする場合を例として説明する．

図 3 は，Pod を複数拠点において起動した時のシーケンス図である．UE 上の機能である Kick App は kube-mec API Server に，kube-mec アプリの起動要求を送信する (図 3-(1))．

ソースコード 1 は, 図 3-(1) で送信する API の形式を表している．1 行目の 「–H “content-type: application/json” 」という部分で，送信するデータの型を指定している．この例では json 形式である．3 行目の 「–X POST」 という部分が，kube-mec API Server に送信するメソッドの型を示している．kube-mec においては，アプリの起動要求時は POST, 終了要求時は DELETE を使用する．5 行目で添付しているデータである userPreference の入れ子には offload, highAvailability, highQuality の 3 種類がある．offload はオフロードの on, off を指定する．on にした場合は演算 Pod は MEC 拠点の MEC サーバに配置され，off にした場合は UE 上に配置される．highAvailability はアプリの高可用性の on/off を決定するためのプリファレンスである．highQuality は，アプリの質の設定をするためのプリファレンスである．gold, silver, bronze の順に質が高く，質が高いほど最大遅延が小さくなり割当帯域が大きくなる．7 行目の 「http://<kube-mec API Server IP>:8888/app/81-456-789/nelio-ar/makecastle」 という URL 部分が，curl コマンドの対象 URL になる．app に続く <ueID>/<appName>/<appOperate>で，起動要求しているユーザとアプリのリソース情報を定義する．ここでは “81-456-789” がユーザを一意に識別するための ueID，“nelio-ar” がアプリ名を表す appName，“makecastle” がアプリ内操作を表す appOperate に対応する．また，図 3-(1) の通信には REST API[10] を採用している．REST API の採用理由としては，kube-mec 対応のアプリ作成者は K8s に精通している可能性が高いため，親和性を持たせるためにも K8s と同様に REST API を採用した方がシステムを扱いやすくなると考えたためである．

kube-mec API Server はアプリ起動要求を受信すると，UE の認証認可を行う (図 3-(2))．5G における認証情報を kube-mec に紐づけることで認証を行う．また，UE に権限のある要求のみを受諾し，権限のない要求 (Pod の起動・作成以外の要求) はこの時点で拒絶することで認可を行う．

kube-mec API-Server は Local Image Repository に UE から起動要求を受けたアプリのコンテナイメージの名前，URL，資源情報を要求する (図 3-(3))．送信内容には，アプリの名前とそのアプリの中で行う操作を含む．Local Image

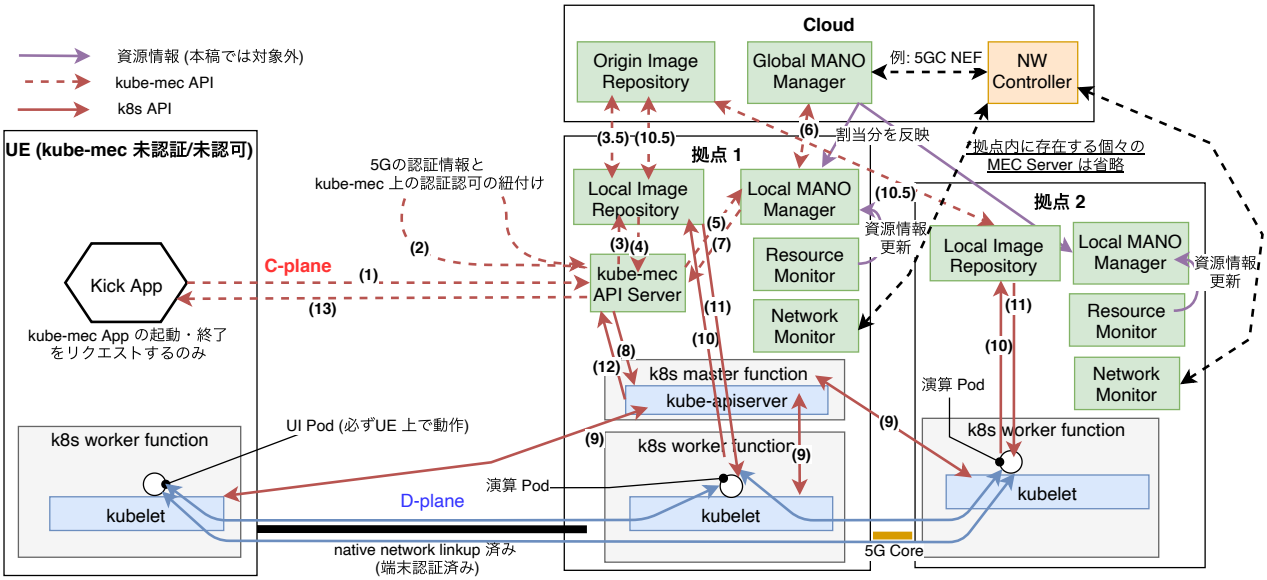

図 3: 複数拠点間におけるアプリ起動のシーケンス.

ソースコード 1: UE ⇨ kube-mec API Server への kube-mec API.

```
curl -H "content-type: application/json"
# メソッドを指定
-X POST
# ユーザプリファレンスに関するデータをjson 形式で添付
-d '{"userPreference": {"offload":"on","
    highAvailability": "on","highQuality": "
    gold"}}'
# 送信先のURL にリソース情報を既述
http://<kube-mec API Server IP>:8888/app
    /81-456-789/nelio-ar/makecastle
```

Repository は kube-mec API Server から受け取ったアプリ名と，アプリ内操作に対応するコンテナ構成情報を返答する (図 3-(4))．Local Image Repository が要求されたコンテナ構成情報を保有していない場合は図 3-(3.5) のように，クラウドに設置されている Origin Image Repository に対応するファイルを要求する．Origin Image Repository は全てのコンテナ構成ファイルを保有している．送信するデータは，コンテナイメージの数，名前，イメージの URL，UI Pod か否かの情報，資源情報，トポロジ情報が主な構成要素を含む．資源情報は，必要な仮想 CPU 数，RAM 容量および Storage 容量が設定されている．

次に kube-mec API Server は，Local MANO Manager へアプリを利用するユーザの ueID, 新規に作成する Pod の数，Pod の詳細 (名前，UI Pod か否か，資源情報)，Pod 配置における条件 (オフロードの有無，トポロジー情報) を送信する (図 3-(5))．これらの情報は MANO 機構が最適な Pod 配置先を決定する上で全て必要な情報である．MEC 拠点内のみでの Pod 配置がリソース上の関係で困難な場合，Local MANO Manager はクラウドの Global MANO Manager に配置可能拠点先を問い合わせる (図 3-(6))．Global MANO Manager は，Local MANO Manager から受信した配置対象の Pod の詳細情報 (資源情報，トポロジ情報) を基に Pod の最適な配置先 MEC 拠点を算出し，算出結果を Local MANO Manager に返答する．Local MANO Manager は Global MANO Manager からの算出結果を受け取り次第，kube-mec API Server に Pod の配置先 MEC サーバを伝える (図 3-(7))．UE 上で起動される Pod に関しては ueID，MEC サーバ上で起動される Pod に関しては対応する Worker Node 名が返却される．

kube-mec API Server は UE，Local Image Repository，Local MANO Manager から受信した諸々の情報を統合して，最終的に K8s-native で扱われる yaml フォーマットに整形し，K8s-native な API を用いて kube-apiserver に送信する (図 3-(8))．kube-apiserver は kube-mec API Server から要求を受け取った後，Pod 配置先の Worker Node 上の kubelet に Pod 作成命令を出す (図 3-(9))．kube-let は，K8s に参加しているノードを管理するためのコンポーネントである．kubelet はそれに従って自リソースを使用して Pod 作成を開始する．作成が開始された演算 Pod は，kube-mec API Server から与えられた imageURL にアクセスしてコンテナイメージのダウンロードを要求する (図 3-(10))．Local Image Repository は Pod からのコンテナイメージダウンロード要求に応答する (図 3-(11))．次に kube-apiserver は，全 Pod 配置先ノードにおいて Pod の起動が完了したことを kube-mec API Server に伝達する (図 3-(12))．アプリの起動要求が正しく実行されたことを UE に伝達し (図 3-(13))，一連のシーケンスが完了する．

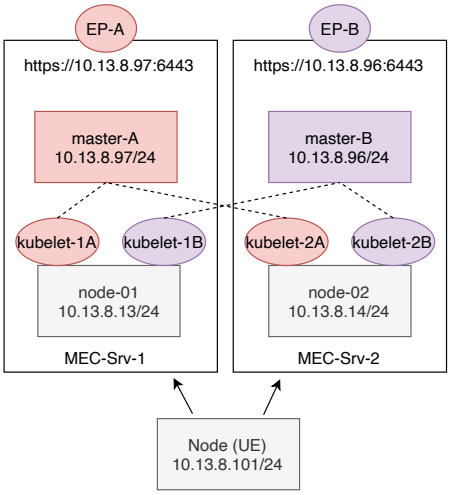

図 4: 本稿における kube-mec の実装環境.

表 1: 実装で用いた VM のスペック (全 VM 共通).

項目	内容，バージョン
OS	Ubuntu 18.04 LTS
CPU	vCPU × 4
RAM	4 GB
ストレージ	32 GB
Docker	19.03.6
K8s	v1.19.1
kubeadm	v1.19.1

4. kube-mec の実装詳細

図 4 は本稿における kube-mec アーキテクチャの実装環境の概略図である．MEC 拠点が 2 つあり，それぞれに MEC サーバが 1 台ずつ設置されている想定で，それぞれの MEC サーバには Master Node 1 台，Worker Node 1 台がそれぞれ用意されている．それぞれの Worker Node は 2 つの Master Node の配下に入っているため Master Node は自 MEC 拠点，他 MEC 拠点問わず Pod を配置することが可能である．UE も Worker Node の 1 つとして K8s クラスタに組み込まれる．kube-mec アプリを使用する場合，UE はどちらかの MEC サーバのクラスタに参加する．

UE と kube-mec API Server 間の通信に用いる REST API には Python のウェブアプリケーションフレームワークである Flask[11] を使用している．

実装で用いた VM のスペックを表 1 に示す．K8s クラスタの構築には kubeadm[12] という K8s が公式に提供している構築ツールを用いている．kubeadm を使用することで K8s クラスタに最低限必要なコンポーネントを自動的にデプロイすることが可能になる．

また，本稿の実装においては，CNI (Container Network Interface) として Flannel[13] を展開している．基本的に，

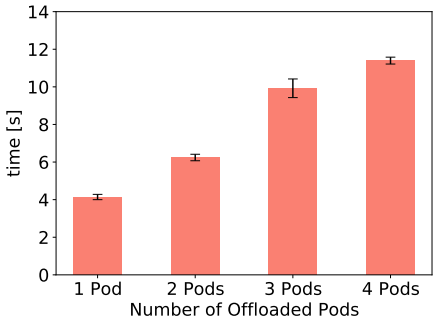

図 5: Pod (1 GB) の起動時間.

Docker で起動したコンテナに付与される IP アドレスはコンテナが起動している Worker Node の外からは疎通性のない Internal IP アドレスであるため，ノードをまたいでコンテナ同士が通信することができない．Flannel はノード間に仮想的なトンネルであるオーバーレイネットワークを構成することで，K8s クラスタ内の Pod 同士の通信を可能にする．

Python 3.6.9 を使用して，kube-mec API Server とダミーの Local Image Repository，Local MANO Manager を実装した．正式な MANO 機構と Repository 機構は今後実装する予定である．

5. 評価

本章の測定においては，Pod が UE の ディスプレイをマウントすることによりアプリの実行結果を表示しているため，UI Pod を使用していない．また，拠点 と UE 間，また拠点間においては MEC において一般的な 5 ms の遅延を挿入している．

5.1 Pod の起動時間

kube-mec のシーケンスによる Pod 起動時間を測定した．また，Pod のサイズは 1 GB，数は 1 〜 4 個に変更し測定した．

正式な Local Image Repository は未完成であるため，本測定ではアプリの名前を受信するとそれに対応するコンテナイメージ名とトポロジー情報を返却するダミーの Local Image Repository を用いる．また，MANO 機構も未完成であるため本測定における Local MANO Manager は Pod 配置要求を受信すると，静的に配置先ノードの名前を返却するダミーの Local MANO Manager を用いる．

本測定では Pod の起動時間を，「UE が kube-mec API Server に Pod 起動要求を送信した時点から，UE が kube-mec API Server から起動完了を受信するまでの時間」と定義する．

図 5 は，Pod の起動時間の計測結果である．あらかじめコンテナイメージが Worker Node や UE にダウンロードされている場合は，Pod の個数が 1 個増えるにつれて起

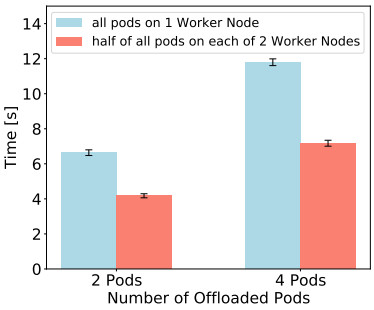

図 6: Pod 分散配置時の起動時間.

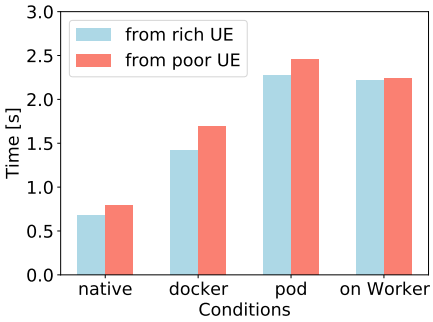

図 7: OpenCV アプリの起動時間.

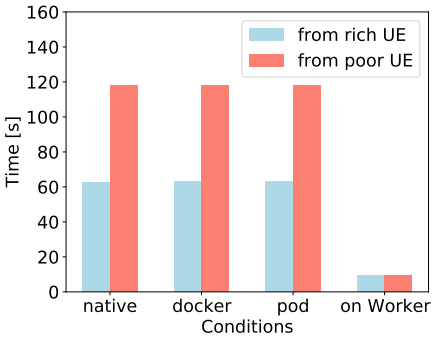

図 8: パラメータ検出アプリの実行時間の比較.

動時間は約 2.6 秒増えることが結果として得られた．これは，K8s の Pod デプロイがシーケンシャルに行われることが理由であると考えられる．

また，今回は kube-mec による起動シーケンスにおける起動時間だけを提示し，k8s-native なシーケンスにおける起動時間を提示していない．測定に使用している Local Image Repository と Local MANO Manager がダミーであり，kube-apiserver との通信以外に要する時間は微小 (約 0.03 秒) であるためである．

また，K8s においては Pod の起動シーケンスがノードごとにシーケンシャルに行われることから，1 つの Worker Node に全ての Pod をオフロードした時と，2 つの Worker Node に半分ずつ分散して Pod をオフロードしたときの Pod の起動時間を総 Pod 数 2，4 の場合で計測した結果を比較する (図 6)．結果として，分散配置は 1 台への配置に比べて総 Pod 数 2 の場合は約 2.51 秒程度，総 Pod 数 4 の場合は約 4.67 秒程度 Pod の起動時間が削減できることが得られた．これは，オフロード先の Worker Node の数だけ Pod の起動処理が並列して行われるためであると考えられる．kube-mec では Worker Node が複数 MEC 拠点の Master Node の傘下に入るため，Master Node 視点ではオフロード先の Worker Node の選択肢が MEC 拠点内のみで K8s クラスタを閉じる場合に比べて多くなる．そのため，Pod の起動を複数 Worker Node 上に分散することで所要時間を削減できるという結果は kube-mec の優位性を高めるといえる．

5.2 アプリの起動・実行時間

次に，OpenCV[14] による特徴点検出アプリを使用してアプリの起動・実行時間を測定した．UE から OpenCV アプリの操作選択画面が表示されるまでの時間を起動時間とする．また，UE から特徴点検出アプリの実行要求を送信した後計算処理されて結果が表示されるまでの時間を実行時間とする．また，本測定で用いる Worker Node のスペックは vCPU × 64, RAM 128 GB と，強力にしている．

図 7，8 は，左から UE 上で python プログラムとしてアプリを起動した場合 (native)，UE 上で docker コンテナを起動し，その中でアプリを起動した場合 (docker)，UE 上で K8s の Pod としてアプリを起動した場合 (pod)，Worker Node 上で K8s の Pod としてアプリを起動した場合 (on Worker) でアプリの起動・実行時間を計測し rich UE と poor UE で比較したものである．また，測定における rich UE, Worker は vCPU × 4，RAM 4 GB を有し，poor UE, Worker は vCPU × 2，RAM 2 GB を有するというスペックに設定している．

図 7 から，native と pod の起動時間を比較すると約 1.6 秒の差が生じていることがわかる．オフロードを利用するためにはアプリを pod 化する必要があるため，この時間がオフロード使用のためのオーバーヘッドになると言える．

また，図 8 より，オフロードを使用した場合は UE 上で実行する場合に比べて実行時間を Rich UE の場合は約 53.3 秒，Poor UE の場合は約 108.6 秒削減できることが結果として得られた．このことから，オフロードを使用するためのオーバーヘッドは約 1.6 秒ほど発生するが，処理負荷の重いアプリであればオフロードの使用によって実行時間を大幅に削減できることがわかる．

5.3 アプリの起動・実行中の CPU・RAM 使用量の測定

次に，OpenCV による共通物体検出アプリを使用してアプリの起動・実行中の CPU の使用率・RAM の使用量の推移を測定した．

図 9，10 は，本測定シーケンスにおける CPU，RAM の使用量を計算したものである．なお，RAM は初期値に差

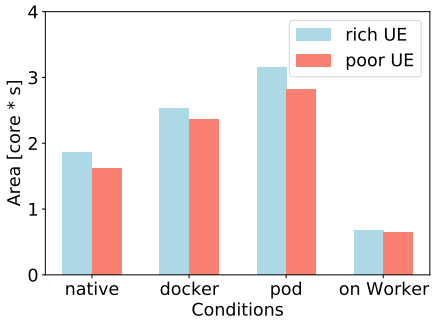

図 9: CPU 使用量.

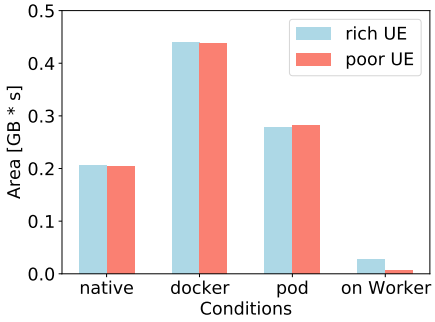

図 10: RAM 使用量の増え幅.

が生じているため測定開始時の RAM の使用量からの増え幅を計算した.

同じアプリでも native での実行，docker での実行，pod での実行の順でランタイム規模が大きくなるが，図 9 より，UE 上で計算処理を行う場合はランタイムの規模の拡大につれて CPU の使用量が大きくなっていることがわかる．pod をオフロードしている on Worker では，UE 上で計算処理を行う場合と比べて使用量が Rich UE では約 78.5 %，Poor UE では約 77.1 % 削減できていることがわかる．CPU と RAM に関して，pod をオフロードしている on Worker では UE 上で計算処理を行う場合と比べて使用量が Rich UE では約 90.2 %，Poor UE では約 97.7 % 削減できていることがわかる.

6. おわりに

本稿では，地理的分散な環境において K8s を使用した MEC オフローディング機構である kube-mec アーキテクチャを提案し，プロトタイプを実装した．kube-mec の導入により隣接拠点との分散協調とマルチマスター構成が実現され，遅延や帯域といったネットワーク情報を含めて Pod の配置先を決定することが可能になった.

また，kube-mec API Server の導入により UE に対する認証認可技術を自由に選定可能になり，K8s に変更を加えることなく MANO 機構や Local Image Repository を kube-mec に組み入れることが可能になった.

kube-mec API Server を含んだ kube-mec アーキテクチャにおいて Pod の起動・終了時間，アプリの起動・実行時間，UE 上における CPU/RAM の使用率の推移に関する評価を行った．この評価により，実装したオフローディング機構が正しく機能していることを確認した.

今後は，MANO 機構と MANO 機構を実現するための情報収集機構，Image Repository 機構の実装を目指す.

参考文献

[1] Meulen, R.: Gartner says 8.4 billion connected “Things” will be in use in 2017 up 31 percent from 2016, *Gartner Newsroom* (2017).
[2] Hu, Y. C., Patel, M., Sabella, D., Sprecher, N. and Young, V.: Mobile edge computing—A key technology towards 5G, *ETSI white paper*, Vol. 11, No. 11, pp. 1–16 (2015).
[3] Velasquez, K., Abreu, D. P., Goncalves, D., Bittencourt, L., Curado, M., Monteiro, E. and Madeira, E.: Service orchestration in fog environments, *Proceedings of 2017 IEEE 5th International Conference on Future Internet of Things and Cloud (FiCloud)*, pp. 329–336 (2017).
[4] Agarwal, S., Yadav, S. and Yadav, A. K.: An efficient architecture and algorithm for resource provisioning in fog computing, *International Journal of Information Engineering and Electronic Business*, Vol. 8, No. 1, p. 48 (2016).
[5] Santos, J., Wauters, T., Volckaert, B. and De Turck, F.: Towards network-aware resource provisioning in kubernetes for fog computing applications, *Proceedings of 2019 IEEE Conference on Network Softwarization (NetSoft)*, pp. 351–359 (2019).
[6] Karamoozian, A., Hafid, A. and Aboulhamid, E. M.: On the fog-cloud cooperation: How fog computing can address latency concerns of IoT applications, *Proceedings of 2019 Fourth International Conference on Fog and Mobile Edge Computing (FMEC)*, pp. 166–172 (2019).
[7] Eidenbenz, R., Pignolet, Y.-A. and Ryser, A.: Latency-Aware Industrial Fog Application Orchestration with Kubernetes, *Proceedings of 2020 Fifth International Conference on Fog and Mobile Edge Computing (FMEC)*, IEEE, pp. 164–171 (2020).
[8] Suter, M., Eidenbenz, R., Pignolet, Y.-A. and Singla, A.: Fog application allocation for automation systems, *Proceedings of 2019 IEEE International Conference on Fog Computing (ICFC)*, pp. 97–106 (2019).
[9] Bruschi, R., Bolla, R., Davoli, F., Zafeiropoulos, A. and Gouvas, P.: Mobile edge vertical computing over 5G network sliced infrastructures: an insight into integration approaches, *IEEE Communications Magazine*, Vol. 57, No. 7, pp. 78–84 (2019).
[10] Masse, M.: *REST API Design Rulebook: Designing Consistent RESTful Web Service Interfaces*, O’Reilly Media, Inc. (2011).
[11] Grinberg, M.: *Flask web development: developing web applications with python*, O’Reilly Media, Inc. (2018).
[12] Kubernetes: Kubeadm, `https://kubernetes.io/ja/docs/reference/setup-tools/kubeadm/`.
[13] Park, Y., Yang, H. and Kim, Y.: Performance analysis of cni (container networking interface) based container network, *Proceedings of 2018 International Conference on Information and Communication Technology Convergence (ICTC)*, pp. 248–250 (2018).
[14] OpenCV: OpenCV, `https://opencv.org`.