Vol.2021-DPS-187 No.9
Vol.2021-MBL-99 No.9
Vol.2021-ITS-85 No.9
2021/5/27

地理的分散環境を想定したMECオフローディング基盤におけるネットワーク・計算資源管理機構

安森 涼[1,a)]　渡邊 大記[1,b)]　稲垣 勇佑[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[2,h)]

概要：地理的分散した MEC (Multi-access Edge Computing) 環境において，遅延，帯域などのネットワーク情報を考慮したスケーラブルなサービス展開が求められる．筆者らはコンテナオーケストレーションシステムの一つである Kubernetes (K8s) を利用し MEC オフロードを実現する基盤である kube-mec を提案している．本稿は kube-mec における MANO (Management and Network Orchestration) 機構を提案する．ネットワーク，計算資源情報収集と配置計算をネットワークエッジとクラウドで階層的に実施する MANO 機構により，K8s では取り扱えなかったネットワーク情報を考慮した Pod 配置が可能となる．5 つの計算拠点にオフロードする際，オフロードする Pod の数に関わらず K8s のデフォルトスケジューラと比べて，kube-mec MANO は kube-mec App を構成する Pod 間の総通信遅延を削減することが出来る．

Management and Network Orchestration Mechanism in MEC Offloading Infrastructure Considering Geographically Distributed Environment

1. はじめに

スマートフォンなどのユーザ端末 (UE: User Equipment) が普及し浸透するにつれ，ストリーミング，拡張・仮想現実，ゲームなどの多くのモバイルアプリケーションが台頭している．これらのアプリケーションを処理するには高い計算能力が必要となる．一方で，UE には限られた計算能力や記憶容量などの計算資源制約がある．UE で高い計算能力を必要とするアプリケーション要求を実現するために無線ネットワークのエッジに計算資源を用意し，UE の近くで処理を可能とする MEC (Multi-access Edge Computing) [1] という技術が注目を集めている．UE は MEC により計算処理を低遅延に実現できる．

筆者らはコンテナオーケストレーションシステムの一つである Kubernetes (K8s) [2] を用いた MEC オフローディング基盤である kube-mec [3] を提案している．kube-mec では機能ごとに分割された多数のコンテナが連携して単一のアプリケーション (kube-mec App) を構成しており，K8s を利用して MEC 基盤へのコンテナ作成と配置，スケーリング，状態監視などを自動的に実施する．

kube-mec はクラウドと地理的に分散した MEC 拠点，および MEC 拠点が収容する複数の MEC サーバによって構成される．K8s では単一の Master Node の制御により複数の Worker Node に Pod と呼ばれるコンテナの集合体を配置，管理することでオフロードを実現する．一方で kube-mec では地理的分散された計算拠点に存在する kube-mec API-Server が UE からの Pod 起動要求を仲介した後に，K8s の API サーバである kube-apiserver に送

[1] 慶應義塾大学大学院理工学研究科
Graduate School of Science and Technology, Keio University
[2] 慶應義塾大学理工学部
Faculty 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) moririn@inl.ics.keio.ac.jp
b) nelio@inl.ics.keio.ac.jp
c) kuwaro@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

信することで多くの Pod 起動要求を許容できる設計となっている．また，kube-mec は 1 つの Worker Node が複数の Master Node に所属することで近隣の計算拠点に資源を融通可能とし，限りある資源を最大限に活用できる構造となっている．K8s では Worker Node の負荷情報 (e.g., CPU, RAM, ストレージ) のみから配置を計算するため Pod 間の通信遅延が大きくなる可能性があるが，kube-mec は 別途 MANO (Management and Network Orchestration) [4] (kube-mec MANO) 機構を用意することで動的に変化するネットワーク情報を考慮した配置計算を実現する．

既存研究における MEC オフロード先決定手法は中央集権的な配置決定と分散的な配置決定の 2 種類ある．前者は地理的分散環境では資源配置をスケーラブルに実現できず，後者は UE に一番近い MEC 拠点内でオフロード先を決定するため，ユーザ数の増加に伴い，利用できる計算資源の上限に到達しユーザの満足度が下がる可能性がある．kube-mec MANO は MEC 拠点とクラウドで階層的に資源情報収集と kube-mec App の配置計算をすることで複数の拠点を網羅したスケーラブルなオフロードを実現する．

本稿は kube-mec MANO と kube-mec App の配置問題を数理最適化モデルとして設計し，実装，評価した．

2. 関連研究

2.1 ネットワーク情報に配慮した K8s の Pod 配置

K8s はデータセンタ内での運用を目的としたものであり，地理的分散環境を考慮していないため，ネットワーク情報を考慮した資源割当が出来ない．そのため，文献 [5] では，スマートシティ環境で K8s を用いてコンテナアプリケーションを管理する際にネットワーク情報を考慮した資源割当手法 NAS (Network Aware Scheduler) を提案している．K8s が利用する Worker Node の CPU, RAM などの計算資源情報に加え，事前に計測した Master Node からの RTT 値に応じたラベルを Worker Node ごとに付与することにより，K8s でネットワーク情報を利用した資源割当を実現する．IDLab Virtual Wall [6] の環境でスマートシティのコンテナアプリケーションを配置して評価を取った結果，NAS は K8s のデフォルトスケジューラによる Pod 配置に対してネットワーク遅延を約 80 % 削減できることを確認した．しかし，NAS にて使用している RTT ラベルは運用前に計測した静的なネットワーク情報であり，RTT に変動があった場合に動的に対応することができない．

文献 [7] では，K8s を使用してノード感の通信遅延やネットワークに流れるデータ量を最小化するための Pod 配置手法 K8s-GBA を提案している．K8s-GBA は ネットワーク情報からコンテナ配置を実現する貪欲法アルゴリズム GBA (Greedy Border Allocation) [8] をベースにネットワーク情報を利用せずに局所的に最適な配置を選択する手法である．結果として，K8s に変更を加えることなく，資源割当の最適化問題を動的に解決している．しかし，K8s-GBA はアプリケーションを構成するコンテナを原則同一の Worker Node に配置し，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アルゴリズムであるため，任意に抽出された Worker Node が遅延を考慮したときに最適であるという保証はない．つまり，K8s のデフォルトスケジューラによる Pod 配置と比較して高遅延な Worker Node を選択してしまう可能性がある．

文献 [5] [7] は小規模でかつ単一のユーザが利用するシングルテナントを想定している．モバイルキャリア網を想定した MEC は地理的に分散した大規模な環境で複数のユーザが複数のアプリケーションを利用するマルチテナントを想定しているため，上記の手法を適用するのは不適切である．

2.2 MEC におけるアプリケーション配置先決定手法

MEC オフロードの際は遅延の影響を受けやすいアプリケーションを適切な計算拠点に配置する必要がある．MEC におけるアプリケーションの配置先決定手法は大きく 2 つに分類できる．第一は中央集権的に配置先を決定するものである．文献 [9] は MEC 対応の IoT (Internet of Things) プラットフォームのためのマイクロサービス配置手法 Dyme を提案している．マイクロサービスのタスク実行のためのネットワーク遅延と価格を非凸最適化問題として定式化し，ユーザに公平な QoS (Quality of Service) と満足度を提供する．Dyme は タスク失敗率，リソース使用率，平均価格の観点で優れた性能を発揮している．文献 [10] は MEC におけるタスクオフロードフレームワーク HyFog を提案している．タスク実行コストの合計を最小化する問題をローカル，エッジノード，クラウドの 3 層グラフマッチングアルゴリズムに変換し解くことですべての機器がローカルでタスク実行するときと比較してコストを 50 % 削減している．文献 [9] [10] は中央集権的手法のため，計算には集中環境が必要となる．一方，本稿では地理的に分散した MEC 拠点を運用するモバイルキャリア網を想定しており多数の UE からオフロード要求が発生する．中央集権による制御では単一障害点になりうる点や，複数拠点を網羅した資源配置をスケーラブルに実現できないという課題がある．

第二は分散的に配置先を決定するものである．文献 [11] はナッシュ均衡の考えを利用したクラウド，エッジノードの階層的なオフロード先決定手法を提案している．ユーザ満足度を最適化したい IoT ユーザ間の競争をモデル化し解くことでオフロードを低遅延に実現する．文献 [12] はメタ強化学習と Seq2Seq (Sequence2Sequence) ニューラルネットワークを組み合わせたタスクオフロード手法 MRLCO

Vol.2021-DPS-187 No.9
Vol.2021-MBL-99 No.9
Vol.2021-ITS-85 No.9
2021/5/27

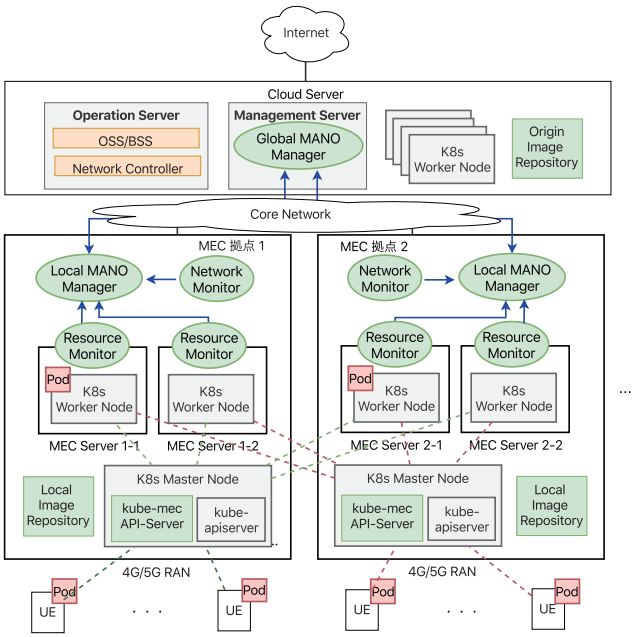

図 1: kube-mec アーキテクチャの全体図.

を提案している．既存のヒューリスティックなオフロード先決定手法と比べて少ない学習ステップ数で遅延を削減している．文献 [11]，[12] は最も近い計算拠点内の計算資源のみにオフロード可能である．しかし分散した計算拠点の計算資源には限界があることが想定される．上限のある計算資源を対象とした配置では多数の UE からのオフロード要求に対してユーザ満足度を維持し続けるのは困難である．

kube-mec の MANO 機構は拠点とクラウドで階層的に kube-mec App の配置を計算することで複数の拠点を網羅したスケーラブルなオフロードを実現する．UE に最も近い MEC 拠点の計算資源が不足している際，他拠点から資源融通を受けることでユーザ満足度を維持しながらのオフロードが可能である．

3. kube-mec の概要

3.1 kube-mec アーキテクチャ

kube-mec は UE からのオフロード要求を MEC 拠点で処理するためのアーキテクチャである．図 **1** は，kube-mec アーキテクチャの全体図である．単一のモバイルキャリア網がインターネットに接続している環境を想定する．クラウドが Core Network を通じて各 MEC 拠点と繋がっており，各 MEC 拠点は 1 台以上の MEC サーバを保持する．MEC サーバは物理または仮想マシンであり，計算資源情報を管理する最小の単位である．

オレンジ色のコンポーネントは 5G-NEF (Network Exposure Function) 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ケーション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 kube-mec 特有のコンポーネントである．

3.1.1 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 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であり，スケーラブルに Pod を配置できる．

kube-mec は 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 サーバの Worker Node に Pod を配置を可能となることで，資源の利用効率を向上することができる．

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 資源情報やアプリケーションの稼働状況などを K8s を介して管理することが可能である．

3.1.2 Image Repository 機構

kube-mec においては，クラウドと各 MEC 拠点に kube-mec のアプリケーションのコンテナイメージを保持するレポジトリを配置する．クラウドに *Origin Image Repository* と呼ばれる全ての kube-mec 対応アプリケーションのコンテナイメージを保持するレポジトリを，各 MEC 拠点にはキャッシュサーバの役割を果たす *Local Image Repository* を配置する．オフロード要求を受けたとき，kube-mec App を構成する複数のコンテナイメージが Local Image Repository 内に存在する場合はその URL を返し，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．

3.1.3 kube-mec API Server 機構

UE と K8s の API サーバ である kube-apiserver を仲介する *kube-mec API-Server* により，独自実装する MANO や Image Repository との情報のやりとりをする．

3.2 kube-mec MANO アーキテクチャ

クラウドと各 MEC 拠点にネットワーク・計算資源情報の収集と kube-mec App の配置先 MEC サーバの選定を担う kube-mec MANO 機構を配置する．

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, 空きストレージ容量) を定期的に収集し，クラウドに送信する．

Network Monitor は MEC 拠点間のネットワーク情報 (RTT，使用帯域) を定期的に収集し，クラウドに送信する．

Local MANO Manager は Network Monitor，Resource Monitor から各 MEC 拠点間のネットワーク情報および各 MEC サーバの計算資源情報を収集し，*Global MANO Manager* は Local MANO Manager 経由でネットワーク・計算資源情報を収集する．Local MANO Manager はその

Vol.2021-DPS-187 No.9
Vol.2021-MBL-99 No.9
Vol.2021-ITS-85 No.9
2021/5/27

計算資源情報をもとに MEC 拠点内で kube-mec App を構成する各 Pod の適切な MEC サーバ配置を計算する．MEC 拠点内で Pod を配置しきれない場合はクラウドの Global MANO Manager に問い合わせる．Global MANO Manager はネットワーク・計算資源情報をもとに複数の MEC 拠点に跨った Pod 配置を計算する．kube-mec App の配置先が決まると，kube-mec API Server に Pod の配置を要求する．

4. kube-mec MANO による最適配置手法

4.1 kube-mec アーキテクチャのモデル化

モデルで使用する記号の定義を表 1 に示す．kube-mec App のデベロッパーはユーザがアプリケーションを実行する際にユーザ満足度向上のために一定の通信速度やネットワーク遅延を保証したいと考えると想定される．デベロッパーは kube-mec App を構成する Pod の数，kube-mec App の名前，構成する Pod の名前，必要な計算資源情報，UI Pod の有無を設定することができる．UI Pod はユーザとの入出力を担う Pod を指す．Pod 間における許容遅延や必要帯域などの接続情報は DOT 言語で記述される．デベロッパーが指定した情報に加え，UE の ID やユーザのオフロード要求の有無などの UE からの情報を合わせた設定ファイル (図 **2**) を kube-mec MANO に送信することで配置を実現する．

$A_{l,m}$ や $P^{band}_{l,m}$，$P^{delay}_{l,m}$ における l，m とは Pod を表しており，行列によって kube-mec App の構成や Pod 間の接続に対する遅延，帯域の要求を設定できる．

MEC サーバの計算資源としては仮想 CPU (vCPU) のコア数とメモリ (RAM) 容量，ストレージ容量を想定し，アプリケーションに割当可能な最大容量と割当済みの容量を定義した．またネットワーク資源としては同一 MEC 拠点内の MEC サーバ間の遅延は無視でき，帯域は十分にあると想定し，MEC 拠点間のリンクのみを考慮する．コアネットワーク上では QoS の保証が可能であるという前提のもと，最大帯域や予約済み帯域，ネットワーク遅延を定義した．

4.2 kube-mec App の配置問題

kube-mec MANO における kube-mec App の MEC サーバへの配置問題を Pod $l \in P$ を入力とし，Pod を配置する MEC サーバ $x^M_{l,i}$ を出力するモデルで表す．最終的に Pod l を $x_{l,m} = 1$ となるような MEC サーバ m に配置する．

4.2.1 Local MANO Manager による拠点内配置

拠点内配置における目的関数を式 (1) に示す．第 1 項は，MEC サーバ i に配置されている Pod の vCPU 要求 $x^M_{l,i} \cdot P^{cpu}_l$ を MEC サーバ i の残り vCPU $C_i - C^{used}_i$ で割ったものである．第 2 項は，MEC サーバ i に配置されている Pod の RAM 要求 $x^M_{l,i} \cdot P^{ram}_l$ を MEC サーバ i の残り RAM $R_i - R^{used}_i$ で割ったものである．式 (1) を最小化することで，より残り vCPU 数と残り RAM 容量が

図 2: kube-mec App 設定ファイルの例.

```
{
    "podNum" : 2,
    "ueID" : "81-456-789",
    "appName" : "nelio-ar.makecastle",
    "podDetail": [
        {
            "podName": "display",
            "ram": "128Mi",
            "cpu": "500m",
            "storage": "100Mi",
            "uiPod" : "yes"
        },
        {
            "podName": "rendering",
            "cpu": "500m",
            "ram": "128Mi",
            "storage": "100Mi",
            "uiPod" : "no"
        }
    ],
    "conditions": {"offload" : "on"},
    "dot": "display -> rendering [maxDelay = '60
        m', minBw = '10Mi']; rendering ->
        display [maxDelay = '30m', minBw = '20Mi
        '];"
}
```

表 1: モデルで使用する記号の定義.

記号	説明
$l \in P$	Pod l
P^{cpu}_l	Pod l の vCPU 要求量
P^{ram}_l	Pod l の RAM 要求量
P^{store}_l	Pod l のストレージ要求量
$A_{l,m} \in \{0,1\}$	Pod l，m 間の接続の有無
$P^{band}_{l,m}$	Pod l，m 間の必要帯域
$P^{delay}_{l,m}$	Pod l，m 間の許容遅延
$A^{UE}_l \in \{0,1\}$	UI Pod であるか
$i \in M$	MEC サーバ i
C_i / C^{used}_i	MEC サーバ i の vCPU 最大数 / 使用数
R_i / R^{used}_i	MEC サーバ i の RAM 最大量 / 使用量
S_i / S^{used}_i	MEC サーバ i の ストレージ最大量 / 使用量
$A^{fail}_i \in \{0,1\}$	MEC サーバ i における障害の有無
$x^M_{l,i} \in \{0,1\}$	MEC サーバ i に Pod l の配置の有無
H	MEC 拠点 H
U	オフロード元 UE U
$N = H \cup U$	MEC 拠点とオフロード元 UE の和集合 (拠点)
$j \in N$	拠点 j
$(j,k) \in L$	拠点 $j \rightarrow$ 拠点 k 間のリンク
$B_{j,k}$ / $B^{used}_{j,k}$	リンク (j,k) の最大帯域 / 予約帯域
$D_{j,k}$	リンク (j,k) の遅延
$x^N_{l,j} \in \{0,1\}$	拠点 j に Pod l の配置の有無

Vol.2021-DPS-187 No.9
Vol.2021-MBL-99 No.9
Vol.2021-ITS-85 No.9
2021/5/27

多い MEC サーバに Pod が配置されるようになり，MEC サーバの負荷分散につながる．

$$\min \sum_{i \in M} \sum_{l \in P} \left(\frac{x_{l,i}^{M} \cdot P_{l}^{cpu}}{C_i - C_i^{used}} + \frac{x_{l,i}^{M} \cdot P_{l}^{ram}}{R_i - R_i^{used}} \right) \tag{1}$$

制約条件を式 (2)〜(7) に示す．式 (2)，(3)，(4) は kube-mec App を構成する Pod の vCPU，RAM またはストレージの要求量の合計が，各 MEC サーバにおいて許容量を超えないための条件である．式 (5) は障害のある MEC サーバに配置をしないための条件である．式 (6) はオフロード対象 Pod を必ずいずれかの MEC サーバに配置することを示す条件であり，式 (7) は UI Pod が UE 上に配置されることを示す条件である．

$$\sum_{l \in P} x_{l,i}^{M} \cdot P_{l}^{cpu} \leq C_i - C_i^{used} \quad \forall i \in M \tag{2}$$

$$\sum_{l \in P} x_{l,i}^{M} \cdot P_{l}^{ram} \leq R_i - R_i^{used} \quad \forall i \in M \tag{3}$$

$$\sum_{l \in P} x_{l,i}^{M} \cdot P_{l}^{store} \leq S_i - S_i^{used} \quad \forall i \in M \tag{4}$$

$$A_i^{fail} = 1 \quad \Rightarrow \quad \sum_{l \in P} x_{l,i}^{M} = 0 \quad \forall i \in M \cup U \tag{5}$$

$$\sum_{i \in M \cup U} x_{l,i}^{M} = 1 \quad \forall l \in P \tag{6}$$

$$x_{l,|M \cup U|}^{M} = A_l^{ue} \quad \forall l \in P \tag{7}$$

これらの目的関数と制約条件をもとに，Local MANO Manager は Pod の MEC サーバへの配置を計算する．

4.2.2 Global MANO Manager による拠点間配置

MEC 拠点内で Pod 配置が完結しないとき，Global MANO Manager に拠点間に跨った Pod の配置を実現する．拠点間配置には 2 段階の手順を踏む．第一に Pod を配置する MEC 拠点を決定し，第二に選択した MEC 拠点から Pod を配置する MEC サーバを決定する．MEC 拠点の決定における目的関数を式 (8) に示す．式 (8) は kube-mec App を構成する Pod 間のネットワーク遅延 $x_{l,j}^{N} \cdot x_{m,k}^{N} \cdot A_{l,m} \cdot D_{j,k}$ をすべての Pod，MEC 拠点について合計したものを表している．

$$\min \sum_{\substack{l,m \in P \\ :l \neq m}} \sum_{j,k \in N} x_{l,j}^{N} \cdot x_{m,k}^{N} \cdot A_{l,m} \cdot D_{j,k} \tag{8}$$

制約条件を式 (9)〜(12) に示す．式 (9) は，各 MEC 拠点間リンクの遅延が kube-mec App を構成する Pod 間接続の許容遅延を超えないための条件である．式 (10) は kube-mec App を構成する Pod 間接続の帯域要求の合計が, 各 MEC 拠点間リンクの許容量を超えないための条件である．式 (11) はオフロード対象 Pod を必ずいずれかの MEC 拠点に配置することを示す条件であり，式 (12) は UI Pod が UE 上に配置されることを示す条件である．

$$x_{l,j}^{N} \cdot x_{m,k}^{N} \cdot A_{l,m} \cdot P_{l,m}^{delay} \geq D_{j,k} \quad \forall l, m \in P,\ \forall (j,k) \in L,\ l \neq m \tag{9}$$

$$\sum_{l \in P, m \in P, l \neq m} x_{l,j}^{N} \cdot x_{m,k}^{N} \cdot A_{l,m} * P_{l,m}^{band} \leq B_{j,k} - B_{j,k}^{used} \quad \forall (j,k) \in L \tag{10}$$

$$\sum_{j \in N} x_{l,j}^{N} = 1 \quad \forall l \in P \tag{11}$$

$$x_{l,|N|}^{N} = A_l^{ue} \quad \forall l \in P \tag{12}$$

これらの目的関数と制約条件をもとに Pod 配置先 MEC 拠点を選択する．図 2，式 (9)，式 (10) で示したように，遅延，帯域に関する要求は kube-mec App の各 Pod の接続ごとに設定することができる．Pod 配置先の MEC 拠点を決定した後，それぞれの MEC 拠点について第 4.2.1 項で示した Local MANO Manager による拠点内配置と同様の手順を踏むことで Pod 配置先の MEC サーバを決定する．

図 **3** に k 個の オフロード Pod を構成する kube-mec App をオフロードするときの拠点間配置のシーケンスを示す．拠点内配置が失敗した際，クラウドの Global MANO Manager に拠点間配置要求を送信する (図 3-(1))．配置要求送信元の MEC 拠点 (自拠点) とそれ以外の隣接した MEC 拠点 (他拠点) の 1 つにそれぞれ $k-1$ 個，1 個の Pod を配置するように計算する．成功したら拠点間配置成功として Pod の配置結果を自拠点の Local MANO Manager に送信する (図 3-(2-1))．他拠点で配置が失敗した際，別の拠点を選択し，再度拠点間配置を試みる (図 3-(2-2))．自拠点で配置が失敗した際は自拠点に $k-2$ 個，他拠点に 2 個として再度拠点間配置を試みる (図 3-(2-3), 図 3-(3))．上記の手順を繰り返した後，自拠点に Pod を 1 つも配置できなかった場合，隣接した MEC 拠点のうち一番近い拠点を自拠点とみなして，自拠点に $k-1$ 個，他拠点に 1 個の Pod を配置するよう計算を実行する (図 3-(4))．すべての拠点で配置が出来なかった場合は，拠点間配置失敗として自拠点の Local MANO Manager に送信する (図 3-(5))．

5. 実装

5.1 資源情報の収集機構

kube-mec MANO アーキテクチャを構成する 3 種類のコンポーネントを実装した．すべてのコンポーネントを Python 3.6.9 で実装した．ネットワーク・資源情報送信や kube-mec App の配置計算要求などの kube-mec コンポーネント間の通信には REST API [13] を利用している．REST API サーバ構築には Python の Web フレームワークである FastAPI [14] を利用した．

Vol.2021-DPS-187 No.9
Vol.2021-MBL-99 No.9
Vol.2021-ITS-85 No.9
2021/5/27

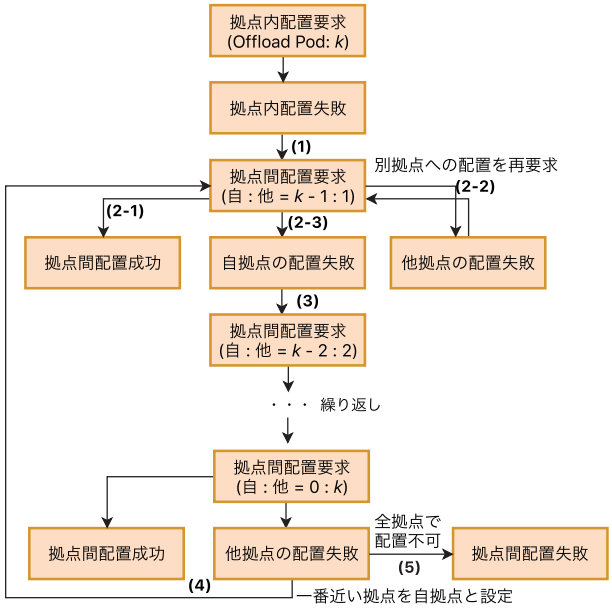

図 3: 拠点間配置シーケンス.

5.1.1 Resource/Network Monitor

Resource Monitor は MEC サーバ上で動いているデーモンであり，定期的に計算資源情報を収集し，Local MANO Manager に送信する．Network Monitor は 5GC を利用することを想定し，MEC 拠点間のネットワーク資源情報を収集し，Global MANO Manager に送信する．

5.1.2 Local MANO Manager

単一の MEC 拠点内で完結する kube-mec App の配置計算と MEC 拠点内の MEC サーバの計算資源管理を単一のプログラムとして実装している．起動時に指定した IP アドレス，ポート番号で Resource/Network Monitor, kube-mec API Server の接続要求を待つ．Local MANO Manager は以下の役割を担っている．

- Resource Monitor と Network Monitor から送信されたネットワーク・計算資源情報を MEC 拠点ごとに集約し，クラウド上の Global MANO Manager に送信する．
- kube-mec API Server からの kube-mec App 配置要求を受け取り，拠点内に完結した配置を実現する．
- 計算資源不足により拠点内の配置ができない場合は Global MANO Manager に複数拠点に跨った kube-mec App の配置計算を要求する．

5.1.3 Global MANO Manager

複数の MEC 拠点に跨った kube-mec App の配置計算と配下すべての MEC 拠点の MEC サーバの計算資源管理を単一のプログラムとして実装している．起動時に指定した IP アドレス・ポート番号で Local MANO Manager の接続要求を待つ．Global MANO Manager は Local MANO Manager からの kube-mec App 配置要求を受け取り，拠点間に跨った配置を実現する．

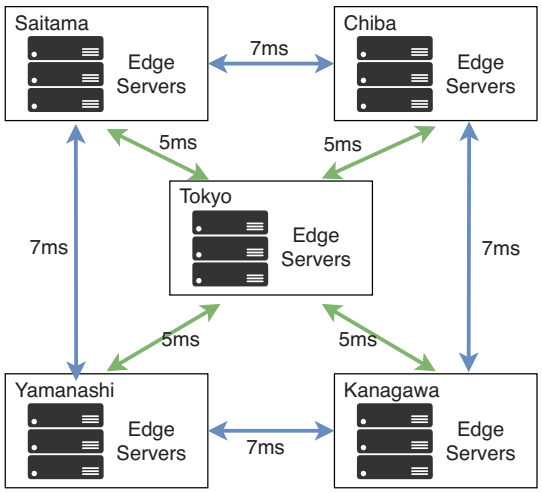

図 4: MEC 拠点間の通信遅延.

表 2: 配置計算における評価環境.

HV (Hypervisor)	
OS	VMware(R) ESXi(TM) 6.7.0
CPU	Intel(R) Xeon(TM) Gold 6248 2.50GHz × 40
RAM	400GB
VM (Virtual Machine)	
OS	Ubuntu Server 18.04 LTS
vCPU	Intel(R) Xeon(TM) Gold 6248 2.50GHz × 32
RAM	32GB

5.2 最適化手法による配置計算

最適化手法とは，式 (1)〜(12) で定義した目的関数・制約条件を利用し，ソルバーによって解を求める手法である．モデリング言語として Python で数理最適化モデルを記述可能なライブラリ Pyomo [15] を使用し，ソルバーとして IBM ILOG CPLEX Optimizer 12.10.0 [16] を使用した．

6. 評価

kube-mec MANO (Global MANO Manager および Local MANO Manager) に関して，Pod の配置計算時間と配置計算結果を評価した．図 **4** に MEC 拠点間のネットワークトポロジを示す．東京と隣接した 4 県の計 5 都県に MEC 拠点を配置した．東京と他の拠点との間の遅延は一定で 5ms，東京を経由しない通信における拠点間の遅延は一定で 7ms，東京を経由する場合の遅延は一定で 10ms とした．また，UE と最も隣接した MEC 拠点との遅延，および同一 MEC 拠点内の遅延は 1ms とした．評価環境として，表 **2** に示す HV (Hypervisor) 上の VM を使用した．図 **5** にオフロードする kube-mec App の構成を示す．kube-mec App は UI Pod 1 つと 1 つ以上のオフロード対象 Pod によって構成されている．オフロード対象 Pod 数が 1 つ増加すると Pod 間の相互接続の数も 1 つ増加するようになっている．

Vol.2021-DPS-187 No.9
Vol.2021-MBL-99 No.9
Vol.2021-ITS-85 No.9
2021/5/27

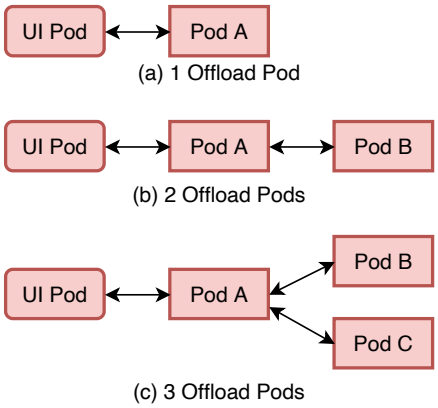

図 5: kube-mec App の構成.

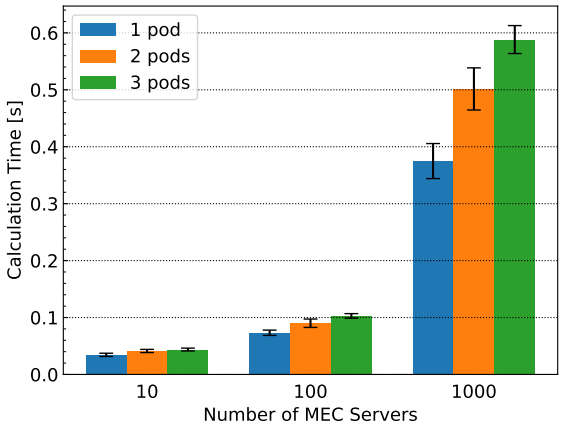

図 6: 拠点内配置計算時間.

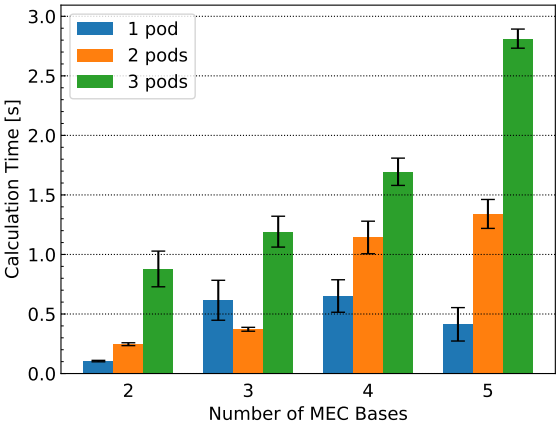

図 7: 拠点間配置計算時間 (ベストケース).

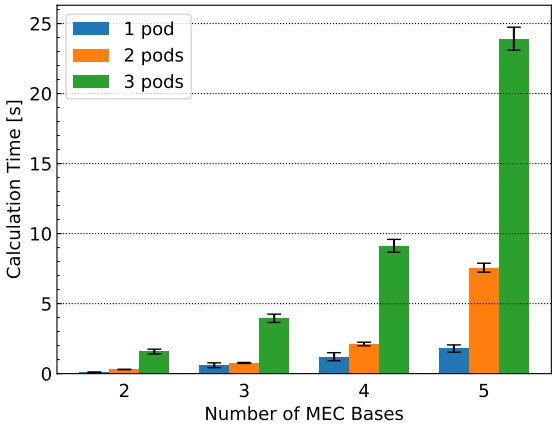

図 8: 拠点間配置計算時間 (ワーストケース).

6.1 拠点内配置計算時間

図 **6** に拠点内の MEC サーバ数に対する拠点内配置計算時間を 100 回計測して算出した平均と標準偏差を示す．オフロード対象 Pod 数や配置候補の MEC サーバ数が増加するほど，配置計算時間も増加する．1,000 台の MEC サーバに対して 3 つの Pod をオフロードする際も 0.6 秒程度で配置先 MEC サーバを決定可能である．実運用される MEC 拠点の MEC サーバ収容数は高々 100 台程度と想定される．その際，拠点内配置計算時間は約 0.1 秒となっている．以上より Local MANO Manager は即応性の高い拠点内配置計算を実現していることが確認できた．

6.2 拠点間配置計算時間

図 **7**，図 **8** に MEC 拠点数に対する拠点間配置計算時間のベストケースとワーストケースをそれぞれ表す．100 回計測を繰り返し平均と標準偏差を算出した．図 7，図 8 はともに 1 MEC 拠点あたり 100 台の MEC サーバを収容しているときの計測結果である．2 拠点のときは埼玉，東京，3 拠点のときは埼玉，神奈川，東京，4 拠点のときは埼玉，千葉，東京，神奈川を選択して評価した．

6.2.1 ベストケースにおける配置計算時間

ベストケースは拠点内配置が失敗した後に，自拠点と他拠点にそれぞれ $k-1$ (k は kube-mec App を構成する総 Pod 数) 個と 1 個の Pod の拠点間配置要求が送信され，1 度も失敗することなく成功したとき (図 3-(1)，図 3-(2-1)) を指す．オフロード対象 Pod 数や配置候補の MEC 拠点が増加するほど，配置計算時間も増加する．5 つの MEC 拠点に対して 3 つの Pod をオフロードする際も 3 秒以内で配置先 MEC サーバを決定可能である．

6.2.2 ワーストケースにおける配置計算時間

ワーストケースは拠点間配置が失敗し続け，一番遅延が大きい MEC 拠点にすべての Pod が配置されることになったときを指す．ベストケースと同様にオフロード対象 Pod 数や配置候補の MEC 拠点が増加するほど，配置計算時間も増加する．5 つの MEC 拠点に対して 3 つの Pod をオフロードする際，25 秒以内で配置先 MEC サーバを決定可能である．

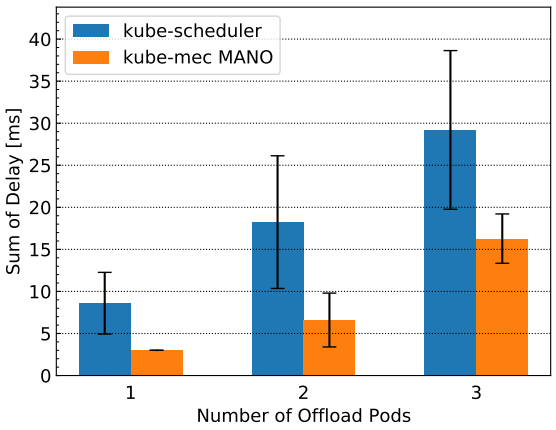

図 9: kube-mec App を構成する Pod 間の総通信遅延.

6.3 拠点間配置結果

図 **9** に K8s のデフォルトスケジューラである kube-scheduler による kube-mec App の MEC サーバへの配置結果に基づく Pod 間の通信遅延の総和と，kube-mec MANO による配置結果に基づく Pod 間の通信遅延の総和の比較を示す．埼玉，千葉，東京，山梨，神奈川の 5 つの MEC 拠点が 1 台ずつの MEC サーバ (RAM 最大量: 約 1GB) を収容し，東京の MEC 拠点からオフロード要求があるときを想定する．各 MEC 拠点の使用中の RAM 容量を 100 〜 500 MiB の乱数で与え，オフロードする Pod の要求 RAM 容量を 100 〜 300 MiB の乱数で与えた．50 回計測を繰り返し平均と標準偏差を算出した．kube-scheduler と比較して，kube-mec MANO は オフロードする Pod の数に関わらず kube-mec App を構成する Pod 間の総通信遅延を 44.2 % 〜 63.8 % 削減している．また，kube-mec MANO の kube-mec App 配置に基づく Pod 間通信遅延の総和における標準偏差が小さいことから，MEC 拠点の資源情報の容量に大きく左右されず，Pod 間の通信遅延を削減するような配置を実現している．

7. おわりに

本稿では kube-mec における MANO 機構を設計・実装，動的に変化するネットワーク情報を考慮した kube-mec App の MEC サーバへの配置計算手法を提案した．kube-mec MANO は資源情報収集と配置計算が MEC 拠点とクラウドで階層的に実施されるスケーラブルな設計となっており，遅延や帯域などの動的なネットワーク情報を考慮した Pod の配置先決定を実現した．評価結果から kube-mec App の MEC サーバへの配置手法に関して，5 つの MEC 拠点で各 MEC 拠点に 100 台の MEC サーバが収容されている環境で，遅くとも 25 秒以内で解を算出できることを確認した．また，kube-scheduler と比較して，kube-mec MANO は オフロードする Pod の数に関わらず kube-mec App を構成する Pod 間の総通信遅延を削減できることを確認した．今後は複数 UE からのアクセスを想定した kube-mec MANO におけるスケーラビリティについて評価を取っていきたい．

参考文献

[1] 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).
[2] Cloud Native Computing Foundation (CNCF): Kubernetes. https://kubernetes.io.
[3] 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基盤，研究報告マルチメディア通信と分散処理 (DPS)， Vol. 2021-DPS-186, No. 22, pp. 1–8 (2021).
[4] ETSI: Network Functions Virtualisation (NFV); Management and orchestration, *ETSI GS NFV-MAN 001*, Vol. V1.1.1 (2014).
[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] imec and University, G.: The virtual wall emulation environment. https://doc.ilabt.imec.be/ilabt-documentation/index.html.
[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)*, 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] Samanta, A. and Tang, J.: Dyme: Dynamic Microservice Scheduling in Edge Computing Enabled IoT, *IEEE Internet of Things Journal*, Vol. 7, No. 7, pp. 6164–6174 (2020).
[10] Chen, X. and Zhang, J.: When D2D meets cloud: Hybrid mobile task offloadings in fog computing, *Proceedings of 2017 IEEE International Conference on Communications (ICC)*, pp. 1–6 (2017).
[11] Shah-Mansouri, H. and Wong, V. W. S.: Hierarchical Fog-Cloud Computing for IoT Systems: A Computation Offloading Game, *IEEE Internet of Things Journal*, Vol. 5, No. 4, pp. 3246–3257 (2018).
[12] Wang, J., Hu, J., Min, G., Zomaya, A. Y. and Georgalas, N.: Fast Adaptive Task Offloading in Edge Computing Based on Meta Reinforcement Learning, *IEEE Transactions on Parallel and Distributed Systems*, Vol. 32, No. 1, p. 242–253 (2021).
[13] Fielding, R. T. and Taylor, R. N.: Architectural Styles and the Design of Network-Based Software Architectures, PhD Thesis (2000).
[14] Ramírez, S.: FastAPI. https://fastapi.tiangolo.com.
[15] COIN-OR: Pyomo. http://www.pyomo.org.
[16] IBM: ILOG CPLEX Optimization Studio. https://www.ibm.com/products/ilog-cplex-optimization-studio.