Vol.2021-DPS-189 No.8
2021/12/20

拡張性と柔軟性を両立したコンテナベースアドホックIoT基盤の一考察

戸河 圭太[†1,a)]　大畑 誠弥[†1]　片瀬 拓海[†1]　山本 隼矢[†1]　中沢 実[†1]

概要：近年，急速な IoT デバイスの増加や高性能化が進んでいる．これによりデバイス管理が複雑化し，全体的なリソースの利用効率が低下すると考えられる．IoT デバイスに限らず，自動運転普及による身近な計算リソースの増加が見込まれている．特に自動車に搭載されるリソースは走行中にのみ利用されると考えられる．そこで，本研究ではコンテナを用いた拡張性と柔軟性を持つ IoT 基盤を提案する．この提案では，余剰リソースを一時的に機械学習やセンシングデバイスとして活用する．これらを実現するために，コンテナオーケストレーターである Kubernetes を活用することで，余剰リソースの汎用性を向上させる．

キーワード：IoT，Container，Edge Computing，Ad-Hoc，Clustering

Container-based Ad-Hoc IoT System with Extendability and Flexibility

1. はじめに

昨今，IoT の普及によりエッジデバイスが増加している [1]．台数だけでなく目的も多様化しており，センシングや複雑な処理を行うため，デバイスの処理能力が向上するなど高度化している．これにより，利便性が向上して多くのユーザのニーズを満たしている一方で，デバイス管理が複雑になり，利用されないリソースが相対的に増えることが考えられる．

また，近年は自動運転の開発が進められている．自動運転には画像認識など高度かつ高速な計算が必須であり，自動車にはその要件を満たすコンピュータリソースが搭載される．クラウド上にあるコンピュータリソースを使用することも考えられるが，電波状況や通信遅延を考慮すると車載コンピュータで処理を行うほうが現実的である．これらの高性能リソースによる処理は走行中に限定され，駐車中は利用されないと考えられる．公共施設の駐車場は多くの高性能コンピュータリソースが集合するが，それらは一切使われないということになる．

このように，利用可能である身近なコンピュータリソースの増加に伴う，全体的な利用効率の低下が懸念される．これらの問題を解決すべく，本稿で提案するシステムでは自身のサーバを中心にその付近にある利用されていないコンピュータリソースを一時的にクラスタリングすることで余剰リソースの有効活用を行おうというものである．

具体的な余剰リソースの活用例として，IoT としてのセンシングデバイスや機械学習の計算リソースが挙げられる．管理下にない IoT デバイスのセンサーを一時的に用いることで，管理下にあるデバイスだけでは取得できない値を得たり，それをもとにした状況把握や分析に役立てたりすることができるのではないかと考えた．特に，自動車は自動運転を実現するため高性能なセンサーを多く搭載していると考えられるため，センシングデバイスとして有効である．また，大きなコンピュータリソースを搭載しているデバイスとして捉えることができ，エッジデバイスの限られたリソースでは行えない画像処理や機械学習，任意の演算処理のリソースとして活用することができるようになることを想定している．

本研究では上記で示した利用を実現するためのコンテナベースアドホック IoT 基盤を提案する．以下，2 章では既存手法，3 章では提案手法とその詳細，4 章では評価を行う．最後に本稿のまとめと今後の展望について述べる．

[†1] 現在，金沢工業大学
Presently with Kanazawa Institute of Technology
[a)] b1816123@planet.kanazawa-it.ac.jp

2. 既存手法

2.1 関連研究

IoT とは，Internet of Things の略称であり，あらゆるモノがインターネットに接続可能であることを指す．一般的には家電やカメラなどのセンサーデバイスがインターネットに接続されることで，リモートでの操作や状態のモニタリングが可能となった．また，近年では工場内の機械や端末を IoT 化するスマートファクトリーや自動車自身がインターネットに接続されるコネクテッドカーも存在する．スマートファクトリーでは機械の稼働状況を把握し，効率化を図ることに利用される．コネクテッドカーでは車両の状態や周囲の道路状況などコンテンツ配信が可能となる．IoT の普及が進み，これまで想定されていなかった使い方が試みられるなど注目されている．

IoT に関連するキーワードとして，Edge Computing が挙げられる．上記で述べた IoT ではセンシングやモノの制御が中心に行われていたが，Edge Computing ではインターネットに頼らず，ユーザに近いところで処理をすることを指す．これまではエッジデバイスでデータを収集し，インターネットを介してクラウドへ送信したのちにクラウド上で任意の処理を行ってから結果をエッジへ返すといった方法がとられてきた．しかし，これらの方法ではクラウドとの通信の際に遅延が生じたり，ネットワーク障害に影響されたりなどデメリットがある．これらの問題を回避するため，信頼性やリアルタイム性が高いエッジコンピューティグを用いることが多い．

本研究の関連事例として Azure IoT[2] や FogBus2[3], Cloud4IoT[4] などが挙げられる．Azure IoT は IoT デバイスプラットフォームとして Microsoft 社により提供されているサービスである．デバイスの管理やデプロイ，監視などの機能をサポートしている．プラットフォーム自体はクラウド上で稼働しており，エッジで動作するデバイス群を一括で制御することが可能となる．これを利用することで IoT 開発者のデバイス運用のコストを軽減することに繋がる．

FogBus2 ではコンテナベース分散フレームワークの提案と評価が行われている．スケジューリングやスケーラビリティのアルゴリズムを複数実装し，IoT 開発者のため新たなリソース追加を容易に行えるよう支援する機能や，クラウドとエッジでの自由なスケジューリングできる機能が実装されている．

Cloud4IoT はクラウドとエッジのハイブリッドクラスタを利用した IoT プラットフォームである．マスターとなるノードがクラウド上に存在し，エッジデバイスがワーカーノードとして参加する．マスターは必要に応じてノード間だけでなく，エッジデバイスからクラウド上のノードへコンテナを移行することが可能となっている．

2.2 問題点

ここで従来の IoT 基盤での問題点を 2 点挙げる．1 点目はひとつの目的に特化しており，拡張性が乏しいことである．機能拡張は管理対象となる IoT デバイスが増加するほど複雑になる．センシングデバイスで新たな機能を拡張する場合，以前存在していたデータのクリーンアップや新規のプログラムを配置，対象となるデバイスのハードウェアや OS を考慮するなどの必要性が生じ，デバイスによる環境依存の低減する手段を検討する必要がある．2 点目は各自が定義し，実装したプラットフォームを使用して IoT デバイスを管理すると汎用性が低下してしまうことである．前項で述べたいくつかの事例では独自で定義されたプラットフォームが使用されている．自身の管理下にあるデバイスのみを対象としている場合に問題は生じないが，前章で挙げた余剰リソースの有効活用に着眼すると実現が困難となる．余剰リソースは使用するハードウェアやソフトウェアに差異があり，デバイス同士が協調して動作することが難しくなるからである．協調するデバイス間では共通のプラットフォームを活用することが重要となる．

2.3 関連技術

2.3.1 コンテナ

コンテナ技術は昨今注目されており，現在の運用の主流となりつつある．仮想化技術の一種と呼ばれ，仮想マシンより軽量である点がメリットである．仮想マシンではハードウェアをエミュレートし，OS を実行することでホストマシンとは完全に分離した状態で仮想化を実現している．一方で，コンテナでは必要なライブラリやファイルのみを格納し，OS など一部のリソースをホストマシンと共有することで実現している．コンテナ自身が OS を持たないことでオーバーヘッドを減らし，起動時間の短縮や可搬性の向上につながっている．仮想マシン同様，環境依存を考慮する必要が少ないため，開発環境と本番環境で一貫性を維持しつつ，軽量なことを活かして開発サイクルの高速化を図ることができる．コンテナではホストのディレクトリや分散ストレージにボリュームをバインドしない限り，一時的なボリュームとして扱われるため，前項で懸念していた機能拡張の際のクリーンアップの問題は解消できる．また，コンテナからのホストマシンに接続されている物理センサーにもアクセスできるため，IoT 分野でも活用である．

2.3.2 Kubernetes

Kubernetes[5] はコンテナプラットフォームとして主流になりつつある Google 社が開発した OSS であり，現在ではクラウドベンダーがマネージド Kubernetes を提供するなど，サービスを運用する多くの組織で利用されている．前項で述べたコンテナは「1 コンテナ 1 アプリケーション」が

望ましいとされているため，仮想マシンに比べて管理する対象コンテナは増えてしまう．これを解決するため，コンテナオーケストレーションツールと呼ばれる，Kubernetes が存在する．Kubernetes は複数のノードでクラスタリングを行い，Docker コンテナを稼働させる．Yaml ファイルによる宣言的な管理が可能で，標準機能が豊富である．ここでは提案システムに関わる重要な機能を 3 点説明する．

1 点目はスケジューリング機能である．Kubernetes は宣言されたコンテナ数を自動的に維持する．この際，スケジューリングはノードのステータスをもとに自動で行われる．このほか，全ノードへの配置やラベルに基づいたスケジューリングを行うことも可能である．クラスタリングを行った各ノードにはあらかじめ任意のラベルを設定することができ，そのラベルをもとにコンテナの配置が行われる．一般的には，高速なストレージデバイスを搭載しているか，GPU を搭載しているかなど各ノードの特徴をもとにラベリングするといった使い方がされる．これによって，目的に応じた柔軟な運用が可能となる．

2 点目はオートヒーリング機能である．宣言したコンテナ数を下回った場合，宣言を満たすように自動的に再デプロイを行う．これは Kubernetes 内部のコンポーネントが定期的にコンテナのステータスを監視することで実現されている．ただし，コンテナはボリュームをバインドしていない限り一時的なストレージとなるため，コンテナ内のデータは維持されない状態で再デプロイが行われる．オートヒーリング機能により，安定してサービスの継続が可能となる．

3 点目はコンテナネットワークのサポートである．デプロイされたすべてのコンテナには IP アドレスが付与され，どのノードにコンテナがデプロイされたかに関わらず，コンテナ間の通信を担保する．また，Kubernetes 内部で稼働している DNS によるコンテナ間の名前解決や，コンテナ間の通信制御可能なネットワークポリシーの設定を行うこともできる．開発者はレイヤーが低いネットワーク部分を考慮する必要がなくなる．

3. 提案手法

3.1 概要

本稿では Kubernetes を活用したアドホック IoT 基盤の実装および評価を行う．前章で述べた 2 つの問題点を解決する．

1 つ目の拡張性に乏しいという問題は，前章でも紹介したコンテナ技術を IoT へ応用することで補う．提案システムの大きな特徴でもある「余剰リソースの有効活用」における手段としてもコンテナベースであることは重要である．余剰リソースのハードウェアはさまざまな OS やアーキテクチャであることが想定される．このような状況において，コンテナを活用することで環境依存を減らしながら

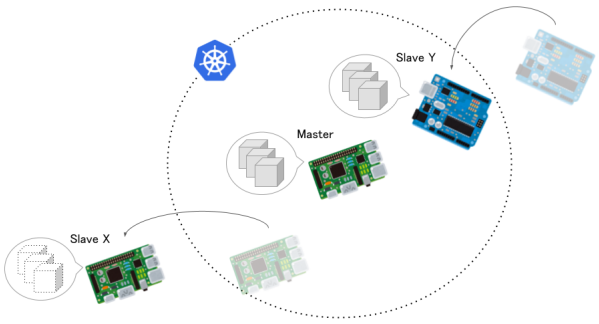

図 1 動作イメージ

機能拡張を行うことができる．また，任意の処理を追加する場合に新たに必要となるパッケージやライブラリの導入をコンテナを用いることで簡略化できるなど，多くのメリットをもたらす．

2 つ目の汎用性低下の問題は，現在さまざな場面で広く利用され，コンテナオーケストレーションツールとして主流となりつつある Kubernetes を採用することで解消する．前章で述べたように，独自で定義したプラットフォームではセットアップが行われていない余剰リソースに対してクラスタリングを行い，有効活用することは難しい．また，それらをコントロールする手段が煩雑になってしまう可能性もある．一方で，Kubernetes は Yaml ファイルによる宣言的な管理，豊富な機能をサポートしており，前述した問題を解消することができる．例えば，余剰リソースを Kubernetes ノードとしてクラスタリングする際に，「hasGPU=true」や「CPU=high」，「sensor=temperature」のようなラベリングを行う．このラベルに基づき，特定のセンサーを搭載したノードや GPU を搭載した機械学習向けのノードなど，目的に応じてスケジューリングを行うことができる．宣言する際の Yaml ファイルには使用するコンテナイメージや数，配置したいノードがもつラベルやノード名を記述する．

提案システムの動作イメージ図を**図 1** に示す．本システムはクラスタでの中心的な役割を担う Master と，ワーカーノードとして動作する Slave の 2 つから成り立つ．これらの役割は実行時に指定しておく．Slave は常に Master となるノードを探索し，通信可能範囲内に入ったときに自動的にクラスタリングを行う．反対に，Slave が Master から離れ，通信が不安定になった場合にはクラスタリングを解除する．このように動的にクラスタを構成するノードを変更することで余剰リソースの有効活用を試みるシステムを提案する．

3.2 実装

本節では実装の詳細について述べる．ソースコードは GitHub リポジトリ [6] で公開している．ワークフローの詳

表 1　Raspberry Pi 4 Model B

OS	Ubuntu Server 20.04.3 LTS
Kernel	5.4.0-1046-raspi
CPU	Quad-core Cortex-A72 ARMv8 SoC @1.5GHz
GPU	—
Memory	4GB
Bluetooth	5.0
Wi-Fi	IEEE802.11b/g/n/ac

細図を**図 2** 示す．本システムは大きく分けて，「デバイス探索」と「クラスタリング」の 2 つのフェーズで動作する．デバイス探索フェーズでは Master が，のちにクラスタリングフェーズで使用する Wi-Fi の情報を Bluetooth によってアドバタイズする．Slave はこのシグナルをスキャンし，Master が近くに存在していることを認識する．Slave は Master からアドバタイズされている情報を用いて，Master の Wi-Fi に接続する．Master がアドバタイズしている情報は SSID となる Bluetooth インターフェースの物理アドレスやパスワードとなる UUID が含まれる．Slave が Wi-Fi に接続後，Master では DHCP サーバが稼働しているため，Slave に対して IP アドレスの割当が行われる．

Wi-Fi 接続後，クラスタリングフェーズへ移行する．ここでは前過程で設定された IP を用いた通信が可能となっている．Slave は Microk8s[7] のインストール状況によって動作が分岐する．すでにインストールされていれば Master に対してクラスタリングの要求を行う．そうでなければ，まず Kubernetes の構築に必要なコンポーネントを Master に要求し，受信する．ここでのコンポーネントは Microk8s そのもののパッケージや Kubernetes クラスタの稼働に必要なコンテナイメージを指す．コンテナイメージにはコンテナネットワークの担保やノードのメトリクスを取得するためのイメージが含まれる．これらを用いて Microk8s の初期セットアップを行う．インストール後は Master へのクラスタリング要求を行う．クラスタリングが完了すると，Master から Yaml ファイルにより定義を行い適用行うことで，クラスタに対して任意の処理を行うことが可能となる．

クラスタリング後は，Master と Slave はそれぞれノードの正常性，Wi-Fi の RSSI を監視する．Master はクラスタ内のノードのステータスを監視し，一定時間”Not Ready” が続いた場合，そのノードを削除して利用しない．同様に Slave は RSSI を監視し，設定した閾値を一定時間下回るとノードの通信が安定していないと判定し，クラスタリング解除の要求を行う．これにより，移動や電波の混線が生じても，可能な限り安定したクラスタを維持する．

3.3　使用技術，環境

開発や検証に使用した環境の詳細を**表 1** と**表 2** に示す．Raspberry Pi と Jetson Nano の 2 種類のエッジデバイス

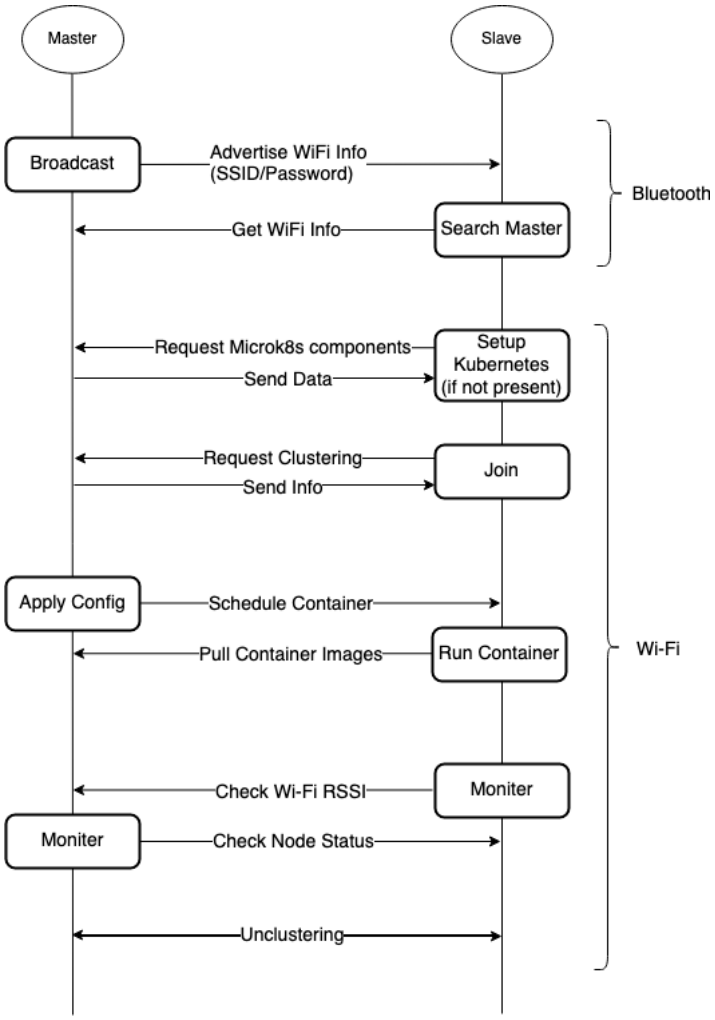

図 2　ワークフロー

表 2　Jetson Nano

OS	Ubuntu 18.04.6 LTS
Kernel	4.9.253-tegra
CPU	Quad-core ARM A57 @1.43GHz
GPU	128-core Maxwell
Memory	4GB
Bluetooth	4.2
Wi-Fi	IEEE802.11ac

表 3　使用ソフトウェア

Software	Version
Golang	1.17.3
Microk8s	v1.22.3 (Rev. 2628)
Kubernetes	v1.22.3-3+64e63223b19811
hostapd	v2.9
Bluez	5.53
iperf3	3.7

を用いて，異なるデバイスで動作可能であることを確認した．使用ソフトウェアを**表 3** に示す．Kubernetes はエッジデバイスのリソースが一般的なサーバと比較して小さいことを考慮し，Microk8s を採用した．k0s や k3s といったディストリビューションが存在するが，なかでも Microk8s

は GPU サポートや導入コスト，アドオンが利用可能であるという点から Microk8s に決めた．ほかにも，提案システムでノード間通信に利用する Wi-Fi のアクセスポイントとして稼働する hostapd[9]，Bluetooth の利用に必要な bluez[10] などを利用している．いずれも開発時点で最新のバージョンである．

4. 評価

ここでは複数台の実機を用いて，提案システムの実用性を評価する．余剰リソースに対して，任意の処理を実行するためのコンテナ配置までにかかる時間や通信速度，帯域に関しての検証を行った．

4.1 検証項目

検証は以下の項目で実施した．

- 各ノード間の通信速度
- クラスタリングに要する時間
- コンテナ展開速度

また，上記の各項目でクラスタに参加するノード数 2-5 へ変更しながら，ノード数による影響が生じるかを検証した．使用したハードウェアは Master, Slave1-3(Raspberry Pi 4) と Slave-jetson(Jetson Nano) である．詳細は前章の使用環境表 1，表 2 で示した．

4.2 検証結果

結果を**表 4** に示す．大きく分けて，通信部分とクラスタリング部分の 2 つについて述べる．

4.2.1 通信

通信に関する検証では，ノード間 (Master-Slave, Slave-Slave) と異なるノード上に配置されたコンテナ間で検証を行った．検証には iperf3[8] を用いた．ノード間，コンテナ間ともに，クラスタ参加ノード数が増えるほど低速になる傾向があった．参加ノードは Master が発している Wi-Fi に集中して接続しているため，ノード増加による帯域圧迫が影響していると考えられる．また，コンテナ間ではノード間と比較してオーバーヘッドが大きくなるため，通信速度が低下している．コンテナ間通信に関しては異なるノードに配置されている Pod 間で計測を行った．

4.2.2 クラスタリング

ここではクラスタリングや参加ノードでのコンテナ展開に要する時間の大きく 2 項目について検証した．1 つ目のクラスタリングに要する時間とは，Kubernetes がインストールされていないノードがデバイス探索を行い，Kubernetes のセットアップに必要なコンポーネントを受信し，インストールとクラスタリングが完了するまでを指す．検証対象は 2 または 5 台目のノードとしている．前項での「参加ノード増加により，通信帯域の圧迫と通信速度の低下が生じる」という結果からクラスタリングに要する時間は増加すると予想された．しかし，実際にはクラスタリング時間が短縮するという結果になった．事前に Microk8s のセットアップが完了しており，デバイス探索とクラスタリングのみを行う場合は 120-150 秒ほどで完了した．

表 4 検証結果の表

Number of Node	n = 2	n = 5
Master-Slave 間の通信速度	77.6 Mbps	74.7 Mbps
Slave-Slave 間の通信速度	-	47.8 Mbps
コンテナ間の通信速度	48.5 Mbps	47.9 Mbps
n 台目のクラスタリング時間	608 s	589 s
コンテナ展開 (Alpine + RaspberryPi)	2 s	6 s
コンテナ展開 (Python + RaspberryPi)	115 s	251 s
コンテナ展開 (Python + Jetson Nano)	67 s	221 s

2 つ目のコンテナ展開とはコンテナ配置に関する設定を適用してから，Kubernetes によるスケジューリング，各ノードがコンテナイメージを Pull し，Pod のステータスが Running になるまでを指す．各ノードはインターネットへの疎通性がないことを想定しており，コンテナイメージは Master 上で稼働しているコンテナレジストリにあらかじめアップロードしていたものを使用する．実際にコンテナ展開の検証に用いたイメージは Alpine(7.93MB) と Python(864MB) の 2 種類である．軽量なイメージである Alpine では展開がすぐに完了するものの，前項で述べたようにノード増加による通信が不安定になる影響を受けていると考えられる．比較的大きなイメージである Python の場合はハードウェアによって差が生じている．同じコンテナイメージにおいても異なるハードウェアに展開する場合，そのハードウェアの処理能力や他ノードとの通信状況に左右されると考えられる．

以上のように，参加するノードの処理能力やノード数による通信帯域圧迫の影響を大きく受ける．コンテナ間での通信はノード間に比べて実測値が低下するため，大きなデータのやりとりは不安定となってしまう．今回の評価では最大 5 ノードまでを検証したが，ノード数を大きく増やすことでよりクラスタリングやコンテナ展開にかかる時間が増えることが想定される．

5. おわりに

本稿では，機能拡張性の向上と余剰リソース有効活用を実現するためのコンテナベース IoT 基盤の提案とその評価検証を行った．Bluetooth によるデバイス探索からクラスタリングを行うことで余剰リソース上で任意の処理を行うことができる．しかし，Master から発する Wi-Fi への接続が集中するため，ノードが増加すると通信が不安定になり，コンポーネント転送やクラスタリング，新たなコンテナの配置に要する時間が大きく，改善の余地が多く見られた．

今後の課題として，1 つは高速化が挙げられる．デバイ

ス探索してからクラスタリングが完了するまでに要する時間が長いことが実用性を考慮した際に懸念点となる．あらかじめ必要なセットアップが行われている場合は数分でクラスタリングが完了する．しかし，ノードに Kubernetes が導入されていない場合は必要なコンポーネントの転送やインストール処理が生じるため，より長い時間を要する．クラスタリング高速化を図り，処理時間を短縮することができれば，余剰リソースを利用できる時間が増え，有効活用率向上につながる．参加ノードの十分なリソースやネットワークインターフェースを搭載することも必要となる．

また，提案システムを運用する場合はセキュリティの向上が必須となる．余剰リソースの有効活用は可能であることを示すことはできたが，現時点ではクラスタリングしたノードを自由に制御できてしまう．あらかじめ利用可能なリソースやソフトウェア的にアクセスできる範囲を制限する必要がある．コンテナの特徴であるホストとの分離性を活かすことができれば向上できると考えられる．ただし，IoT デバイスとしてセンサーを利用する場面が多く想定される．その場合，物理デバイスへのアクセスが必須となるため，セキュリティを向上させる手段を検討しなければならない．ほかにも，デバイス探索時に Slave は Master からの Bluetooth によるアドバタイズされた情報を利用しているが，一対一のコネクションを張り，セキュアな状態で情報のやりとりを行う方式に改善すべきである．

参考文献

[1] 総務省: 令和 3 年版情報通信白書, 入手先 ⟨https://www.soumu.go.jp/johotsusintokei/whitepaper/ja/r03/html/nd105220.html⟩ (2021.11.26)
[2] Microsoft Azure IoT, 入手先 ⟨https://azure.microsoft.com/ja-jp/overview/iot⟩ (2021.11.24)
[3] Qifan Deng, Mohammad Goudarzi, Rajkumar Buyya: FogBus2: A Lightweight and Distributed Container-based Framework for Integration of IoT-enabled Systems with Edge and Cloud Computing, 2021
[4] Corentin Dupont, Raffaele Giaffreda, Luca Capra: Edge computing in IoT context: horizontal and vertical Linux container migration, 2017
[5] Kubernetes, 入手先 ⟨https://kubernetes.io/⟩ (2021.11.24)
[6] cacis，入手先 ⟨https://github.com/Keita0805carp/cacis⟩ (2021.11.26)
[7] Microk8s，入手先 ⟨https://microk8s.io/⟩ (2021.11.24)
[8] iperf3，入手先 ⟨https://iperf.fr/⟩ (2021.11.24)
[9] hostapd，入手先 ⟨https://w1.fi/hostapd/⟩ (2021.11.24)
[10] Bluez，入手先 ⟨http://www.bluez.org/⟩ (2021.11.24)