平成 29 年度

修士学位論文

コンテナ型仮想化におけるスケーリング方式のヘテロジニアス IoT システムへの応用

Autoscaling of Container-Based Virtualization towards Heterogeneous IoT Systems

1205080　　小川　友暉

指導教員　　岩田　誠

2018 年 2 月 28 日

高知工科大学大学院 工学研究科 基盤工学専攻

情報システム工学コース

要 旨

コンテナ型仮想化におけるスケーリング方式のヘテロジニアス IoT システムへの応用

小川　友暉

近年, IoT の発展に伴ってフォグコンピューティングと呼ばれる新パラダイムが着目されている. 背景として, 従来のクラウドコンピューティングによるサーバでの集中処理方式は, IoT システムにおける多量のデータやネットワークの輻輳によって現実的では無く, またリアルタイム処理などの厳しいレイテンシ要件を満たすことができないためである. そのため, エッジデバイス側での処理が現在求められているが, エッジデバイスはサーバに比べて性能が非常に劣っており, 単体で集中処理方式と同じ要求を満たすことは不可能である. また, OS やライブラリの欠如などデバイスのヘテロジニアス性が多く存在するため, システムを構築することは非常に難しく, 管理も難しい.

そこで, 本研究ではヘテロジニアス IoT システムへのコンテナ型仮想化のスケーリングの応用を提案する. Kubernetes では, クラウドコンピューティングなどのホモジニアスな環境で使用されることが前提となっており, ヘテロジニアスなシステムに対しての最適化がなされていない. また, 配置するアプリケーションである Pod に対する, 要求リソースと制限リソースの割り当て量をあらかじめ決めておくことができるが, Pod 自体のリソース量を動的に変化させることができない. そのため, 負荷が上がってきたときに Pod はリソース量を調整することができず, 気温などの変化によって CPU 性能が相対的に変化することによって, 必要なパフォーマンスを発揮することが困難となる.

そこで負荷率によって Pod 自体の要求リソース量を動的にスケーリングさせ, 要求リソース量を増やすことで必要なパフォーマンスの維持を行なう. また, 現在のノードに必要とす

るリソースがない場合, 新しいノードへと再配置することによって, パフォーマンスの維持を行なうことを提案する. 評価では Raspberry Pi 5 台を用いて性能評価を行い，リソースの調整を行なった結果，リソース量の変更から Pod の再生成終了まで平均 7.238 秒の実行時間となった.

キーワード　IoT, コンテナ型仮想化, kubernetes, ヘテロジニアス IoT, スケーリング

Abstract

Autoscaling of Container-Based Virtualization towards Heterogeneous IoT Systems

In recent years, with the development of IoT, a new paradigm called fog computing has been payed attention to. As a background, the centralized processing method in the server based on the conventional cloud computing is not realistic due to a large amount of data and network congestion in the IoT system, and can not satisfy stringent latency requirements such as real time processing. For this reason, processing on the edge device side is currently required, but edge devices are poorer in performance than cloud servers and it is impossible to satisfy the same requirements as centralized processing on a standalone basis. In addition, because there are many heterogeneous properties of devices such as lack of neccesary library, it is very difficult to build and manage a system.

Therefore, we propose maintain stable performance by dynamically scaling the required resource amount of Pod itself by the load ratio, realizing an increase in the requested resource amount, and relocating to the new node when there is no resource required for the current node.

Performance evaluation was conducted using five Raspberry Pi's, and the average time of resource adjustment was 7.238 sec., including changing time of the resource amount and regeneration time of the Pod.

key words IoT, heterogeneous, scaling

目次

図目次

表目次

第 1 章

序論

近年, IoT(Internet of Things) と呼ばれ, モノのインターネットという新しい概念が普及しつつある. 様々なモノ, 機械, 人間の行動や自然現象は膨大な情報を生成しており, これらの情報を収集して可視化することができれば様々な問題が解決できるといわれている. 見ることや聞くこと, 触ることができる情報に加えて，加速度センサーなどのそれらに該当しないセンサー情報も数値化され収集可能になっている. 従来のように人間がパソコン類を使用してデータを入力するだけではなく，モノに取り付けられたセンサーが人手を介さずにデータを収集・入力し, インターネット経由で利用できるようになることが IoT の概念である [1].

IoT をビジネスに活用する企業は増加しており, 適応分野も多様化が進んできている. ITR による調査（図 1.2）によるとモノのトラッキング, 工場生産の最適化, エネルギー消費の最適化をはじめとして, 多岐に及ぶ分野への投資が進んでおり導入企業の多くが効果を実感してることが明らかになっている. IoT の市場は活用企業が増加するとともに, 1 社あたりの投資額も増大することから市場規模は 2017 年の約 4850 億円から 2020 年に約 1 兆 3800 億円へと急速に拡大すると予測している [2].

この IoT の主な活用事例として以下のような事例がある.

- **自動車分野**

  車同士が相互通信することによって衝突の防止を行う. V2V（Vehicle to Vehicle）を利用した, 車同士の通信によって状況を感知し情報を送受を行い, 運転で重要といわれる認知・判断を一瞬で行い衝突を防止することができる. クラウドを介さず, 個々の自

図 1.1　Internet of Things のイメージ図

動車が相互に状況を把握し, 危険を回避するように動くことで, 状況に合わせた早急な対応が可能になる.

- **医療分野**

  AppleWatch などのスマートウォッチによって装着者の健康状態を常時モニタリングし, 運動不足や睡眠時間, カロリーなどを計測し自動転送により見える化を行う. 病気の予防などに加えて潜在的な病気の早期発見なども提案されている. 他にも, 高齢者の自宅に設置することで, 室温などを検知, パソコンなどを通じて温度を調節することで, 熱中症や脱水症状などを未然に防ぐことができる [3].

- **農業分野**

  センサによって施設をモニタリングし, 施設の「今」「過去」のデータを見える化する. 最適な環境制御や作業検討を支援し, またデータを基にした栽培方法の評価・検証を行

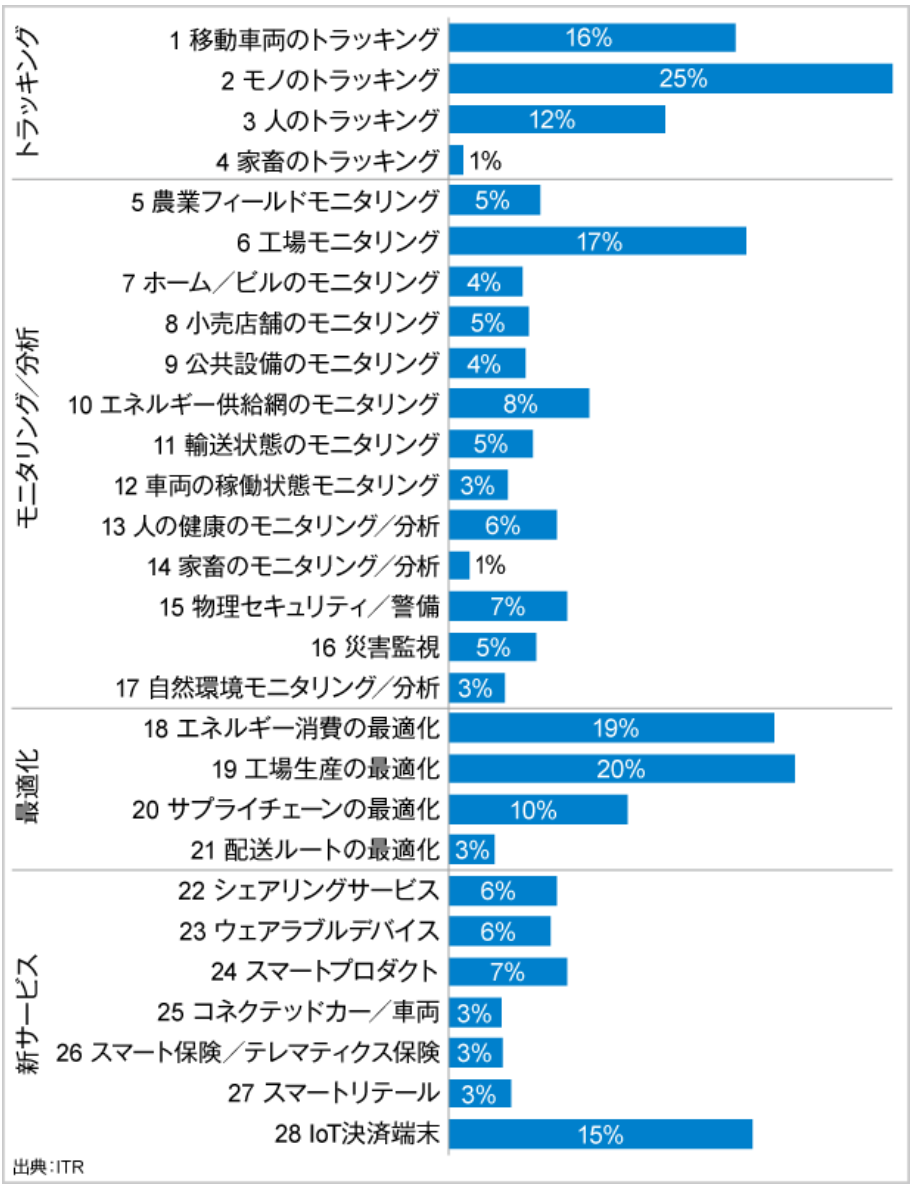

図 1.2　IoT 導入企業における分野ごとの実施率 [2]

うことにより, より高品質な作物栽培が可能となる. 他にも作業工程を見える化することによって, 作業の課題を発見し効率化・低コスト化に寄与することができる.

IoT はイベント発生の認識および対応の迅速化を行うことが可能で, 製造, 石油, ガス, 公益事業, 運輸, 鉱業, 公共部門などの業種では, 応答時間の短縮は生産性の向上, サービスレベルの向上, 安全性の向上に直結する. 工場の場合であれば, 重要なマシンの温度センサが故障に関連する計測値を送信することで, シャットダウン状態を避けるための技術者を修理の

ために派遣することが可能となる．シャットダウンは非常にコストが高く機会損失などを引き起こしてしまうため非常に有用である．石油やガスの調査では, オイルパイプラインのセンサが圧力変化を記録し, それに応じてポンプは自動的に減速し災害が発生することを回避することができる．IoT はあらゆる社会に浸透していき, 生産性の向上からリスク回避まで様々な問題を解決することができるといえる．

こういった中で, IoT の普及に伴いフォグコンピューティングに注目が集まっている [4]. シスコが定義するフォグコンピューティングは以下のようなモデルである．

- 膨大な量の IoT データをクラウドに送信するのではなく, データが生成された場所に近いネットワークエッジで, 最も時間制約の厳しいデータを分析
- ポリシーに基づいて, ミリ秒単位で IoT データに作用
- 選択したデータを履歴分析と長期保存用にクラウドに送信

こういったフォグコンピューティングが台頭してきた理由として, 従来のクラウドコンピューティングによるサーバでの集中処理方式は, 今日の IoT システムで発生する大量のデータやネットワークの輻輳を考慮して設計されていないため現実的では無いことがあげられる．以前では IoT に接続されていなかった数十億ものデバイスが毎日 2 エクサバイト以上のデータを生成しており, 2020 年までに 500 億近くの「モノ」がインターネットに接続されるといわれており膨大な帯域幅が必要となる．例えば，複数の店舗を持つ会社が，1 日の売り上げや顧客情報などをある決められた時刻に本社に送るようなシステムを考える．その場合，始業時間や終業時間といった決められた時刻にトラフィックが増大し，ネットワークの圧迫によって回線を占有することになる．また, 大量のデータが本社サーバに対して送られるため, このようなデータを一定時間内に処理するためだけに高性能なリソースを有するサーバを導入する必要があるなどといった, コスト面での懸念も発生する．このような場合，必要なデータのみを送ることが必要であり，各店舗で情報量を落として本社のサーバにデータを送るようなシステムが必要といえる．

またクラウドコンピューティングによる処理では，リアルタイム処理などの厳しいレイテ

ンシ要件を満たすことができない．エッジデバイス側のみならず, クラウドまでデータを転送する回線の帯域が非常に広く, またアクセス速度も高速であるといった, クラウドの処理能力とネットワークの回線についても考慮する必要が出てくる．そのため過剰なコストがかかってしまうことになる．それに対してフォグコンピューティングは, 少ないレイテンシと帯域幅効率を有し, 企業や事業がもし失敗した場合やエッジデバイスが故障した場合でも, 分散処理によって受ける影響の範囲を押さえることによってレジリエントなエンドユーザサービスを提供することが可能であり, インフラストラクチャの効率を高めることが可能となっている [5].

このため, 昨今, エッジ側での処理の分担が求められている．しかしながら, IoT などのデバイスはヘテロジニアス性を持っている．例えば, Raspberry Pi などを代表とした IoT デバイスはリソースに制約のあるデバイスで構成されている．互換性のないアーキテクチャ, オペレーティングシステムの違い, ライブラリの欠如などにより, サポートされているデバイスの種類ごとにプログラムを開発または微調整する必要があり, またそれぞれのデバイスごとに独自の更新や設定方法が混在するため, アプリケーションのデプロイメントやオーケストレーションも非常に困難であることが知られている [13].

また設置される場所は多種多様であり, また設置場所などの温度変化などによって性能にばらつきが発生し処理効率が変動する．そのため, ヘテロジニアスな環境でも, エッジ側で処理効率を高く維持する機構が必要である [7]．また, エッジデバイスで用いられる組込み機器では前述したリソース制約の観点から, 各アプリケーションが処理を完遂できるように, リソースの確保を行うことが重要である．そのため, 予想される負荷に対して必要なリソース量を見積ることで, 負荷に対応できるようにする.

しかしながら, リソースの必要な量（プロビジョニング量）と, 処理が行われず不必要で無駄になってしまう量（デプロビジョニング量）は時間とともに変化する．様々なアプリケーションには様々な用件があり, そのため, 図 1 のように，Pod ごとに作業負荷のピーク要件を満たすための十分なリソースを確保することはコストがかかることになる.

たとえば, 見積もり負荷が高すぎてしまうと, ピーク時以外には利用していないリソース

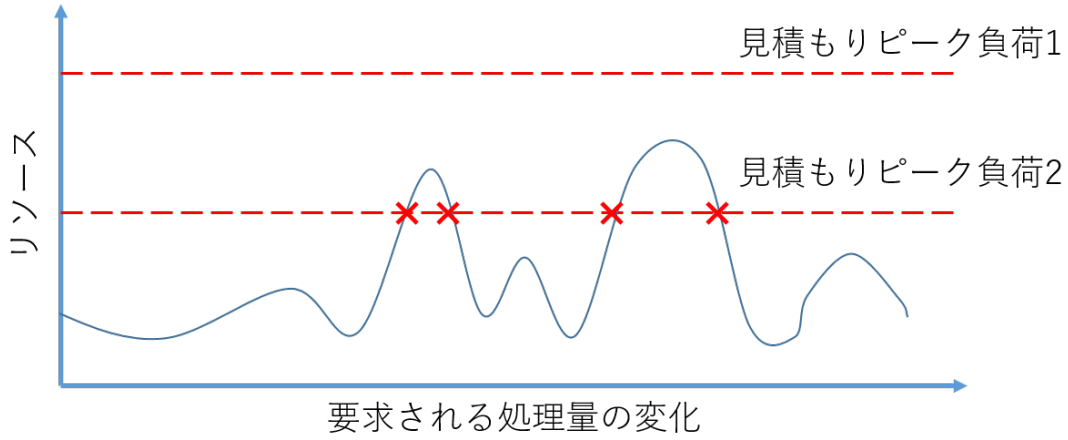

図 1.3　時間的推移とワークロード

が大量に発生してしまい全体の処理効率として無駄が多くなってしまう．しかしながら, 見積もり性能が少なすぎた場合, 負荷が見積もり負荷を超えてしまうと, その間システムは処理が追いつかず要求された時間内での処理が終わらない可能性が生じる．結果としてサービスレベルの低下を招くこととなる．Pod のアプリケーションが必要とするリソース量を活用するためには, Pod 自体のリソース量を動的に変化させ, ピーク時には必要なリソース量を増大させることで大きな処理負荷にも対応できるようにする仕組みが必要である．

コンテナ型仮想化を用いる理由として, ハードウェアリソースをソフトウェアから分離できることに加えて, パッケージ化されたソフトウェアを複数のハードウェアアーキテクチャ上で実行することが可能であることにある．コンテナ型仮想化はハイパーバイザ型仮想化に比較して, 少ないリソース消費, 非常に高速な構築やインスタンス化などの利点がある．加えて, コンテナは VM などの仮想イメージよりも小さいため, より高いアプリケーションおよびサービスの割り当てを可能としている．これらより, IoT およびフォグコンピューティングにおいて非常に有効であることが言える．

本論文ではこのような機構としてコンテナ型仮想化を用いた Kubernetes[8] でのスケーリング方式を提案し, ヘテロジニアスな環境での IoT システムにおいて, 各々のエッジデバイスがアプリケーションの負荷に合わせて動的にリソースの使用量を変更し, ヘテロジニアス IoT システム全体の可用性を向上させることを目標とした．

本章では, 研究背景と研究目的について述べた. 第 2 章では, IoT におけるヘテロジニアス性と仮想化を行うメリットおよび仮想化技術の説明と各メリットを述べる. 第 3 章では, 提案する動的スケーリング方式を構成するコンポーネントについて述べたのち, Web サーバをアプリケーションとして配置し, 評価を行う.

第 2 章

仮想化技術とヘテロジニアス IoT

2.1 緒言

本章では, 現在の IoT におけるヘテロジニアス環境の状況と本研究で用いる仮想化技術について使用される技術と用語についての概要について述べる.

第 1 節では, 現在の IoT システムの動向について述べ, そのシステムにおけるデバイスごとの環境や性能におけるヘテロジニアス性について述べた後, それらを解決する複数の手段とその問題点, 今回採用するコンテナ型仮想化を用いる理由などを示す.

2.2 IoT システムとヘテロジニアス性

IoT は様々な「モノ（物）」がインターネットに接続され, 情報交換することにより相互に制御する仕組みもしくは社会について表している. 従来のネットワーク機器（ルータ, スイッチなど）とは異なり, 物理環境（温度, 電磁波レベル, 動作など）を感知したり, 冷暖房や換気などの物理的な環境に影響を及ぼす動作を起こすことができる. そのほかにも生産ラインの圧力バルブの制御, 渋滞を緩和するための信号機のタイミングの変更など多岐に渡っている. 特に組込みシステムの進歩により, このビジョンが実現しようとしている. 組込みシステムに用いられるプロセッサや無線通信モジュールなどの電子部品はダウンサイジング, 価格の減少, 省電力化などのために, 我々の生活になじみつつある. IoT デバイスが偏在することによって人間がトラフィックの発生と受信を行うのは少数になると推定されており, このような観点からエンドユーザの端末を接続するネットワークインフラストラクチャとしての

従来のインターネットのコンセプトから, 広範なコンピューティング環境を形成する相互接続された「スマート」オブジェクトになるといわれている [11][12]. その結果として膨大なデータ量が生成されており, データの有効活用を行うためにはシームレスかつ, 効率的な方法で収集され, 処理される必要がある.

近年の IoT デバイスの処理能力は集中処理方式のジョブを実行できるレベルに達しており, 1 つの例として Raspberry Pi があげられる. このボードは 1000 万台以上が販売されており, 5000 円程度で非常に安価でありながらも, コンピュータとして十分な性能を保有している. しかしながら, クラウドサーバなどと比較すると, Raspberry Pi などの IoT デバイスは非常にヘテロジニアスでリソースに制約のあるデバイスで構成されている. 互換性のないアーキテクチャ, オペレーティングシステムの違い, ライブラリの欠如などにより, サポートされているデバイスの種類ごとにプログラムを開発または微調整する必要があり, またそれぞれのデバイスごとに独自の更新や設定方法が混在するため, アプリケーションのデプロイメントやオーケストレーションも非常に困難であることが知られている [13].

IoT アプリケーションを開発してデプロイするためにはオーバヘッドを最小限に抑えながら, ヘテロジニアス性を抽象化し, 効率的な管理と操作を可能にするための一般的なソフトウェアインフラストラクチャを共有する必要がある. 組込みデバイスを含む分散アプリケーションのための統合開発環境を提供するために, いくつかのソフトウェア抽象化が開発されている. 1 つのアプローチは, 実際のハードウェアから抽象レイヤー, プラットフォームに依存しない API を介して実行するソフトウェアプログラムをデカップリングできる, オペレーティングシステムを開発することである. しかしながら, これらのシステムでは独自のプログラミング言語を使用する必要があり, 既存のプログラムを書き換えずに使用する可能性を排除することになってしまうため再利用が困難となる [14].

もう一つのアプローチは特定のプラットフォーム用に設計された Linux を用いることによって解決する方法がある [15]. しかしながらこれらはシングルボードコンピュータなどの計算能力が低いデバイスを対象としていない場合が多く，また, これにより新規のソフトウェア統合が特定のオペレーティングシステムごとにケースバイケースで行われなければな

らないことに加えて, この手法による標準化は互換性を提供するためのものでしかなく, 予期せぬ動作やバグにつながる可能性を孕む.

これらの問題を解決する手段として仮想化技術を用いることが解決策である．Virutal Machine はマシンに依存しない形式（バイトコード）にアプリケーションをコンパイルし，インタプリタによって実行される. 同じアプリケーションは，適当なインタプリタを備えていれば，任意のハードウェアプラットフォームで実行可能であり, Android などは例の一つである．Android のコアは Linux カーネルに準拠しており, アプリケーションは Java で記述されている.

クラウドコンピューティングでは, ソフトウェアコンテナに基づく軽量な仮想化が複数のデータセンタで分散アプリケーションを簡単に, 作成, デプロイ, スケーリングできるようになるため, 近年推進されている．これらの軽量な仮想化を用いるためには Linux カーネルに組み込まれている機能を使用することであり, 仮想マシンやハイパーバイザのオーバヘッドなしに仮想環境を作成することが可能となる．アプリケーションはすべての依存関係とともに, サポートされている Linux 環境でインスタンス化できるイメージにパッケージ化されている．IoT の分野では, これらの方法が最近になって検討され始めており, プラットフォームのヘテロジニアス性を克服しつつ, 低オーバヘッド実現するための有望な手段であるといえる.

2.3　仮想化技術の概要

仮想化とは, コンピュータ内の CPU やメモリ, ストレージディスクなどのハードウェアリソースを, 物理的な構成を無視して, 論理的に統合および分割できる技術のことである．仮想計算機（Virtual Machine：VM）とは物理的な計算機と同様の機能ソフトウェアにより仮想的に実現する技術, またその技術により実現された仮想的な計算機である．1 台のコンピュータをあたかも複数台のコンピュータであるかのように論理的に分割し, それぞれに別の OS（Operating System）やアプリケーションを動作させることができるようになる.

従来の技術では, 単一の物理計算機以上では単一の OS しか動作させることができなかったが, この仮想化技術を用いることで, 単一の計算機上に複数の VM を作り, 複数の OS を同時に実行させることが可能となった. 仮想化技術が出現した背景として, 近年, ハードウェアが高性能になり, サーバの処理能力の飛躍的な向上により, その結果としてリソースを効率的に活用することができず, 余剰リソース化を引き起こしてしまっていることにある [16].

仮想化を行う理由とそのメリットについて以下に示す [18].

- 仮想マシンを使用することで, 使用率の低い幾つかのサーバのワークロードを, より少ないマシンへと統合することができる. 場合によっては, 単一のマシンにも統合することができ, そのメリットとして, サーバインフラストラクチャのハードウェア, 環境コスト（冷却, 設置個所の確保）, 管理, および管理コストの節約を行うことができる.
- レガシーアプリケーションを実行する場合に, 仮想マシンを用いることで, 適切な処理を行える. 従来のアプリケーションは新しいハードウェアやオペレーティングシステムでは動作しない可能性があり, 前述のようにサーバを十分に活用できない場合でも, 複数のアプリケーションのコンソリデーションをすることは理にかなっている. このようなアプリケーションはハードコード化されており, 通常は単一の実行環境に共存するようには作成されていないため, 仮想化なしでは実行が難しい.
- セキュリティ上のリスクのある信頼できないアプリケーションを実行するための安全な独立したサンドボックスを構築することができる. インターネットからダウンロードしたファイルを実行する場合でも, 高速に実行環境を作成することができる. 仮想化は安全なコンピューティングプラットフォームを構築する上で重要な概念である.
- 仮想マシンを使用することで, リソース制限が課せられているオペレーティングシステムまたは実行環境を作成し, 適切なスケジューラ, リソースの保証が与えられる. パーティショニングは, QoS 対応のオペレーティングシステムを作成する際に, QoS と同時に使用される.
- 仮想マシンはハードウェアや, ハードウェア構成（SCSI デバイスや, マルチプロセッサ

など）の仮想環境を構築することができる．仮想化は独立したマシンのネットワークをシミュレーションするためにも利用できる．

- 仮想マシンを使用することで, 複数のオペレーティングシステムを同時に実行することができる．異なるバージョンやまったく異なるシステムであっても, ホットスタンバイ状態となることも可能である．このようなシステム中には, 新しいハードウェアで実行するのが困難, 不可能なものが存在する．
- 仮想マシンでは仮想マシンモニタに設置することで, 強力なデバッグとパフォーマンスのモニタリングが可能である．オペレーティングシステムは自身でモニタリングを行なわないため, 生産性を損なうことなく, または, より複雑なデバッグシナリオを設定することなく, デバッグすることが可能である．
- 仮想マシンは, 実行されているプログラムを切り分けることができるため, 障害やエラーの抑留を提供する．ソフトウェア障害を事前に引き起こし, システムの後続動作を調べることも可能である．
- 仮想マシンにより, ソフトウェアの移行が容易になり, アプリケーションとシステムのモビリティを支援する．
- 仮想化により, システムの移行, バックアップ, リカバリなどのタスクを簡単かつ管理が容易になる．

次節で主な仮想化手法について説明する．

2.3.1　ハイパーバイザ型仮想化

ハイパーバイザ型仮想化は一般的にベアメタル仮想化とホスト型仮想化に分別される．ベアメタル型仮想化はタイプ 1 ハイパーバイザとも呼ばれており, 図 2.1 左のようにホスト OS の介在なしで仮想化を実現する仮想化手法である．ESXi や Hyper-V, Xen などが分類される．ホスト OS を必要としないため, ハイパーバイザが直接ハードウェアを制御することができ, 仮想マシンの速度低下を最小限に抑えることができる．

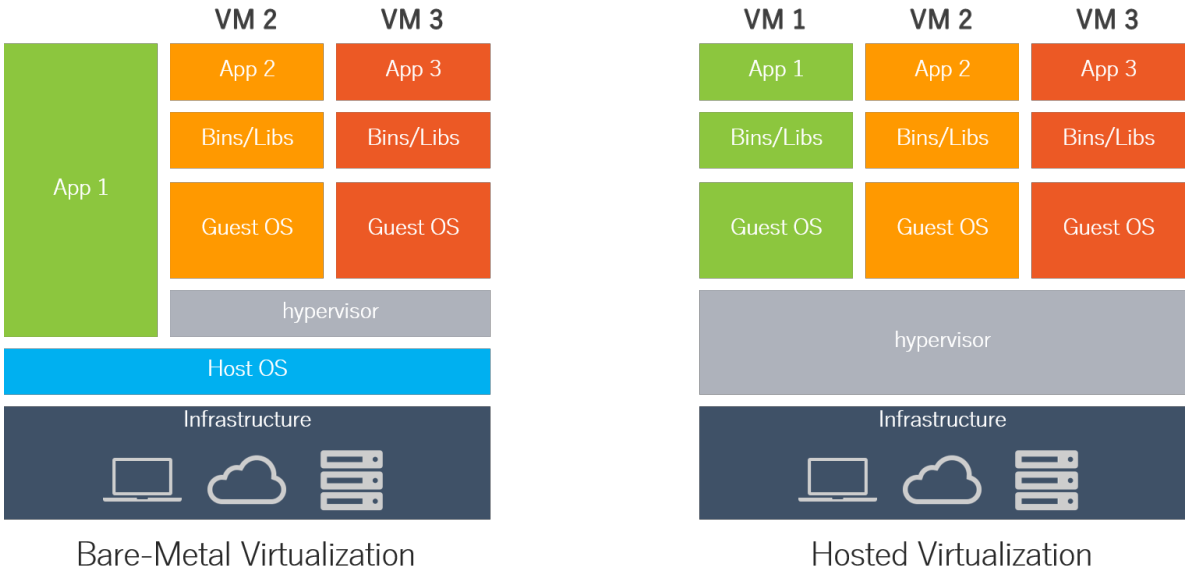

図 2.1　ベアメタル仮想化とホスト仮想化

ホスト型仮想化はタイプ 2 ハイパーバイザとも呼ばれており, 図 2.1 右のように, 通常の OS 上に土台となるソフトウェアをインストールし, そのソフトウェア上で仮想マシンを稼働させる方式である. VMware Player, や VMware Fusion などがホスト型に分類される. ホスト型はすでに利用しているサーバやコンピュータにもインストールすることができるため, 手軽に導入することができるが, ハードウェアへアクセスするにはホスト OS を経由しなければならないため, 余計なオーバーヘッドがかかり十分な性能が出ない場合がある.

2.3.2　コンテナ型仮想化

コンテナ型仮想化とはカーネルの機能を利用してプロセスの独立した環境を構築する方法である. 前述のハイパーバイザ型の仮想化とは対照的に, コンテナは独自の仮想化されたハードウェアを構築するのではなく, ホストシステムのハードウェアを使用する. そのため, 別々のインスタンスを必要とするハイパーバイザ型よりもコンテナを効率的に使用することができる. 図 2.2 左のコンテナイメージには, ファイル, 環境変数, ライブラリなど, 目的のソフトウェアを実行するために必要なコンポーネントが格納されており, また, ホストはコ

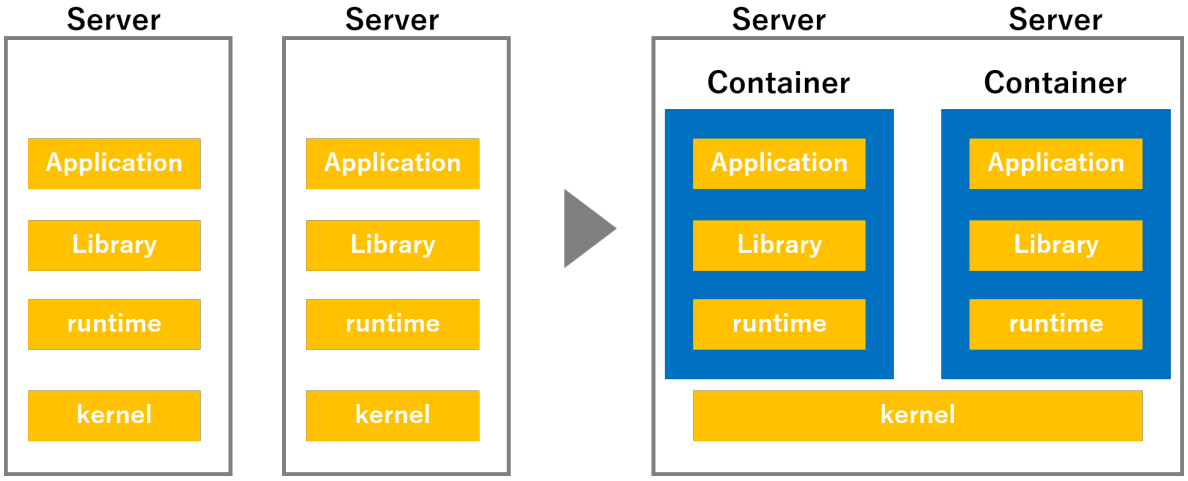

図 2.2　コンテナによる依存関係のパッケージング

ンテナの物理リソース（CPU やメモリなど）へのアクセスを制限しているため, 単一のコンテナがホストのすべての物理リソースを消費することはない. また, コンテナイメージは, オペレーティングシステム（デバイスドライバやカーネル, init など) を実行するための完全なツールチェーンを必要としないため, コンテナイメージは通常の仮想マシンのイメージと比較すると小さくなる. このことから, リソースが少ないためフィンガープリントも小さく, 規模が大きくなるにつれてパフォーマンスの向上およびセキュリティのメリットが非常に大きくなる.

このように, コンテナ型仮想化は従来の仮想化と比較した主なメリットとして, メモリ, CPU, およびストレージの効率の向上が可能であり, コンテナはハイパーバイザ型の別の OS インスタンスに必要なオーバーヘッドを持たないため, 同じインフラストラクチャ上でより多くのコンテナをサポートすることができることにある.

コンテナを利用することによる一般的なメリットを以下に示す.

- アジャイルアプリケーションの作成と配置

  VM イメージの使用と比較して, コンテナイメージを作成するための容易性と効率性を, 向上できる.

- 継続的な開発, 統合, 配置

  信頼性が高く, 頻繁なコンテナイメージのビルドとデプロイメントを, 迅速かつ容易なロールバックで提供できる.

- DevOps における分離

  デプロイメント時間ではなくビルド/リリース時にアプリケーションコンテナイメージを作成することで, アプリケーションをインフラストラクチャから切り離すことが可能.

- 可観測性

  OS レベルの情報とメトリックだけでなく, アプリケーションの健全性やその他の信号も表示される.

- 開発/テスト, およびプロダクション全体の環境の一貫性

  ラップトップでもクラウドと同様に動作する.

- クラウドと OS のディストリビューションポータビリティ

  Ubuntu, RHEL, CoreOS, オンプレミス, Google Kubernetes Engine などあらゆる場所で実行できる

- アプリケーション中心の管理

  仮想ハードウェア上で OS を実行する抽象化レベルを高め, 論理リソースを使用して OS 上でアプリケーションを実行する

- 疎結合, 分散, 伸縮性, リベレートされたマイクロサービス

  アプリケーションは, より小さい独立した部分に分割され, 1 つの大きい単一目的のマシン上で実行される, 肥大化してしまったモノリシックなスタックではなく, 動的に展開および管理できる

- リソースの分離

  アプリケーションのパフォーマンスが予測可能

- リソース利用率

  高効率および高密度なリソースの使用が可能

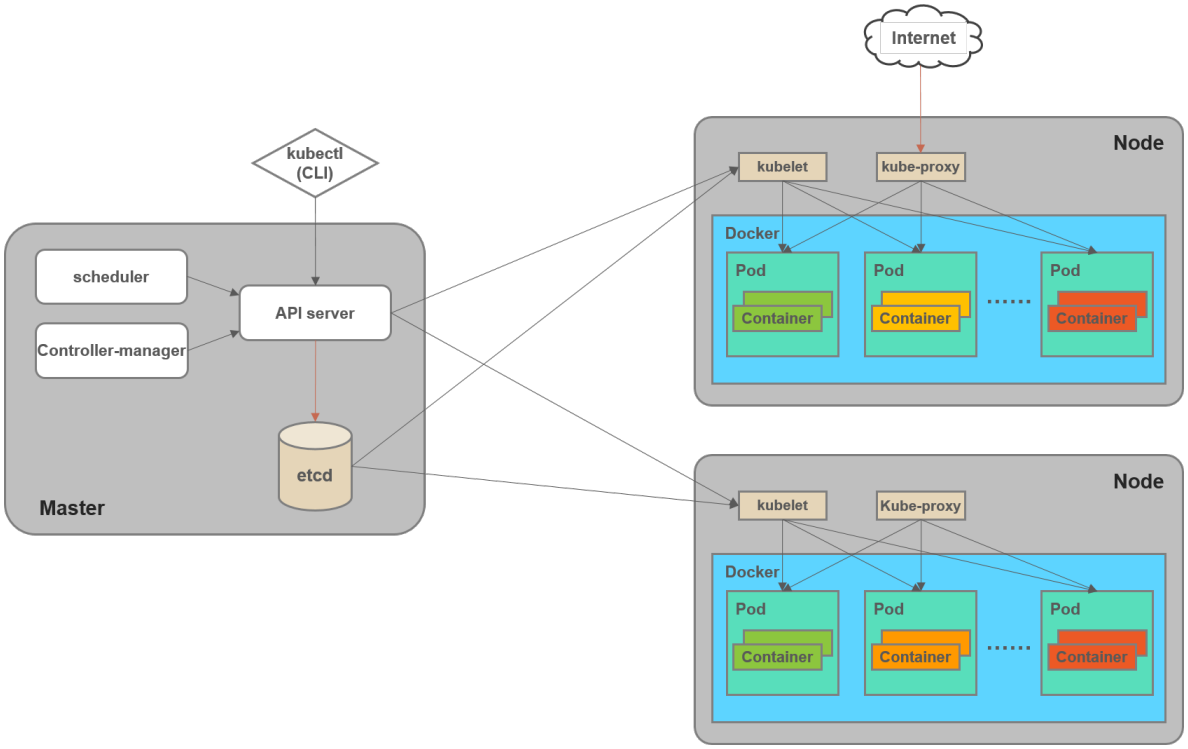

図 2.3　Kubernetes のアーキテクチャ

2.4　kubernetes

Kubernetes とは, Google 社が開発したアプリケーションコンテナのデプロイ, スケーリング, マネジメントを行うオープンソースソフトウェアである. さまざまなインフラストラクチャにわたる分散したホストのクラスタ間で, アプリケーションコンテナのデプロイ, スケーリング, マネジメントを自動化するためのプラットフォームとして提供することを目的としている. Kubernetes の概念を図 4.1 に示す.

Kubernetes はアプリケーションのデプロイ, スケーリング, マネジメントを行うためのメカニズムをまとめて提供するために, 一連の基本要素を定義している. 図 4.1 のアーキテクチャに基づき, 使用した基盤技術について説明する.

2.5　マスタのコンポーネント

Kubernetes では制御プレーンのコンポーネントを Master と呼んでいる．これはノードに対してクラスタ全体のシステム提供と管理のために，管理者への主要なコンタクト先として機能している．Master は単一及び複数のマシンに分散することもできる．このコンポーネントを実行しているサーバーにはクラスタの作業負荷を管理し，システム全体と通信するために使用されるいくつかのサービスがある．

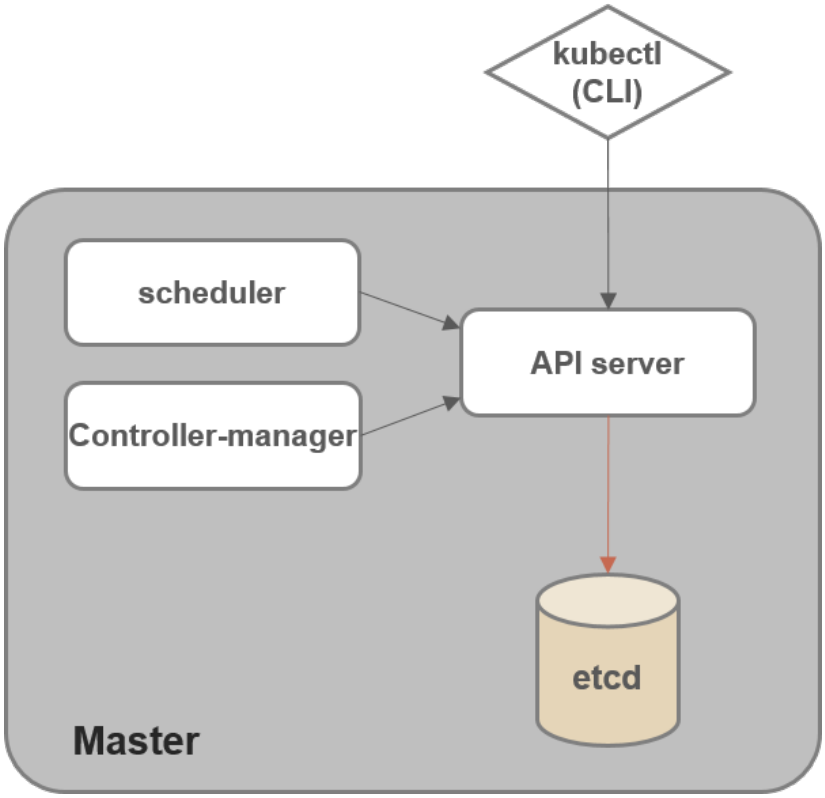

図 2.4　マスタの概要図

2.5.1　API サーバ

API サーバはユーザが多くの Kubernetes のワークロードと組織単位で構成できるようにするためクラスタ全体の主要な管理ポイントになっている．また etcd の情報をもとにデプロイされたコンテナとサービスの詳細が一致しているかどうかを確認する仕組みも持っている．クラスタ全体の正常な状態を維持し，情報とコマンドの伝搬を行うためのコンポーネン

ト間ブリッジとして機能する．API サーバは Restful インタフェースを実装しているため, 様々なツールやライブラリとも通信が容易である．

2.5.2　etcd

etcd は軽量な分散 Key Value Store（KVS）である．Kubernetes では, クラスタ内の各ノードが使用する構成データを格納するために使用する．これはサービスディスカバリにも使用できる上, 各コンポーネントが自身を構成および再構成するために参照するクラスタ状態を持つことができる．単純な HTTP/JSON API によって, 値を設定または取得するための簡易インターフェスが提供されている．マスタ内のほとんどのコンポーネントは etcd 同様に単一のマスタサーバ上に構成することも複数のマシンに分散することもできる．

2.5.3　scheduler

scheduler はポリシー, 豊富なトポロジを意識したワークロード固有の機能であり, 可用性, パフォーマンス, および容量に大きな影響を与える．スケジューラは, この集合的なリソース要件, サービス品質要件, ハードウェアソフトウェアポリシー制約, 類似性および非類似性の仕様, データ局所性, ワークロード干渉, 期限などを考慮する必要がある．

2.5.4　controller-manager

複雑性の解消のためにそれぞれのコントローラーのプロセスは分離されており，すべてが 1 つのバイナリにコンパイルされ，1 つのプロセスで実行されている．これらのコントローラーは Node Controller，ReplicationController，EndpointsController，Service AccountToken Controllers を含んでいる．ノードがダウンした際に反応して応答を行なう．

- Node Controller

  ノードコントローラはノードに CIDR 割り当て, 内部ノードリストの状態監視, ノード状態の監視を行う．

- ReplicationController

  システム内のすべてのレプリケーションコントローラオブジェクトに対して正しい数のポッドを維持する.

- EndpointsController

  Endpoints オブジェクトを生成することによって，サービスとポッドを結合する.

- Service Account & Token Controller

  新しい名前空間の規定のアカウントと API アクセストークンを作成する.

2.6　ノードサーバのコンポーネント

Kubernetes では, ワークロードを実行するサーバは「Node」および「Worker」と呼ばれる. Node は Master コンポーネントとの通信, コンテナのネットワーキングの構成, およびそれらに割り当てられた実際のワークロードの実行に必要な要件を持つ.

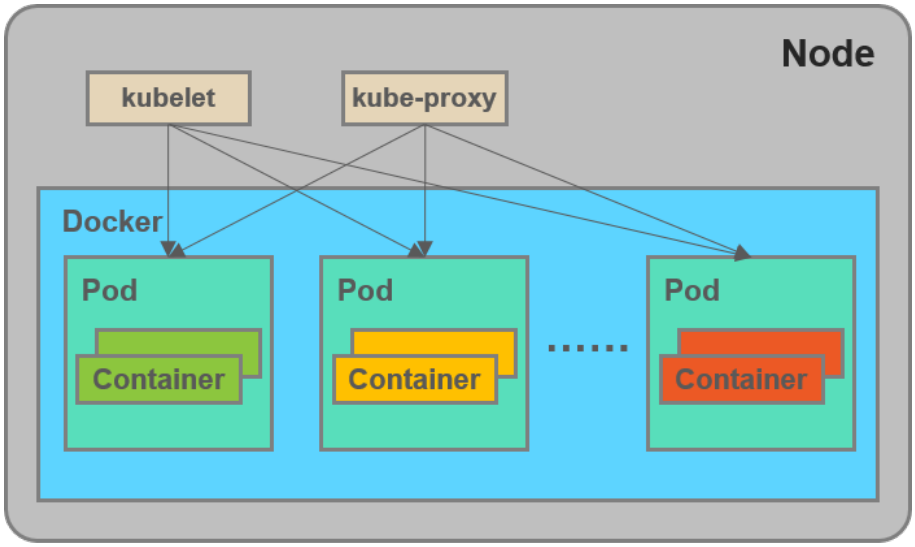

図 2.5　ノードの概要図

2.6.1　kubelet

kubelet はクラスタ内のそれぞれのノードで動作しており，これはコンテナが Pod 内で実行されているかを確認するモジュールである．kubelet は，さまざまなメカニズムによって提供される一連の Pod の状態を取得し，それらの Pod の状態に記述されているコンテナが実行中であること，および正常な状態であることを保証する．

2.6.2　kube-proxy

kube-proxy はホスト上で接続フォワーディングの実行とネットワークルールの管理を行うことによって，kubernetes サービスの抽象化を行うものである．kube-proy は各ノード上で実行されている．これは，各ノードの kubernetes の API で定義されているサービスを反映し，単純な TCP/UDP ストリームの転送とバックエンドでの転送を実行できるようになっている．サービスクラスタの IP アドレスとポート番号は，サービスプロキシによって公開された，ポート指定のための環境変数である Docker-link-compatible によって検出される．これらのクラスタ IP に対してクラスタ DNS を提供するオプショナルアドオンもある．プロキシを設定するためには API server から API を使用してサービスを作成する必要がある．

2.6.3　DNS

kubernetes DNS はクラスタ上で DNS の Pod と Service をスケジュールし，個々のコンテナに DNS サービスの IP アドレスを使用してドメインネーム解決を行なうように kubelet を設定する．DNS 自身を含め，クラスタ内のすべてのサービスはドメインネームが割り当てられ，デフォルトでは，クライアントの DNS 検索リストには，Pod 固有の名前空間とクラスタのデフォルトドメインが含まれている．例として，namespace ”bar ” 内の Service ” Foo ” があるとして，namespace ” quux ” から namespace ” bar ” 内の Service ” foo ” にアクセスしたい場合，DNS のクエリは”foo.bar”となる．

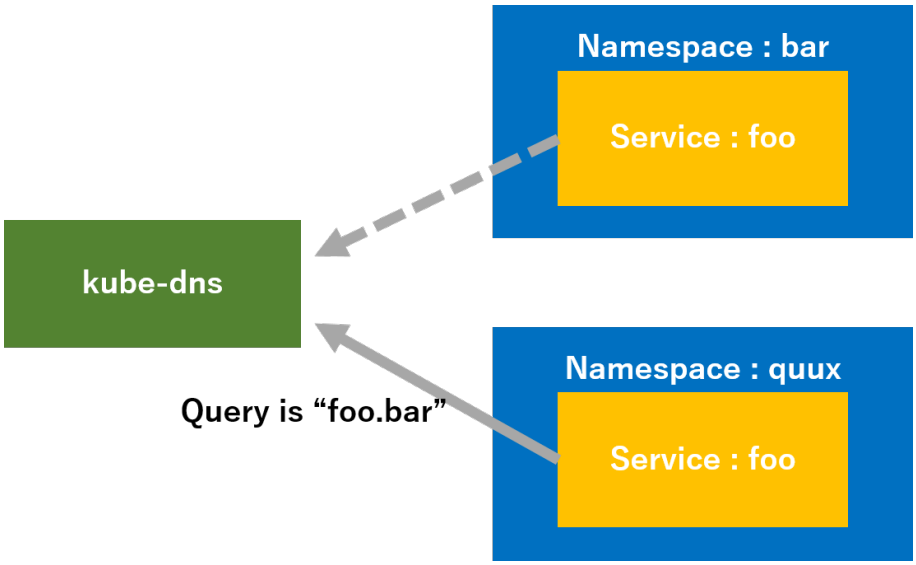

図 2.6　kube-dns クエリ例

また，kubernetes はマイクロサービスアーキテクチャを採用しているため，各サービスに接続するための IP アドレスやポート番号をアクセス元がどのように判断するかという問題がある．この問題を解決する仕組みとしてサービスディスカバリの仕組みが備わっており，DNS の機能を利用して実現している．例として，図 2.7 のように，myapi という名前の Service であれば，同じ名前空間であれば `http://myapi` という URL で Service にアクセスすることができる．図 2.7, 2.8 のようにテスト用の名前空間とプロダクト用の名前空間といったように，別の名前空間で動作している Pod であれば，同じ Service 名であってもアクセスすることが可能であるため，複数の環境の構築が容易になるといったメリットもある．

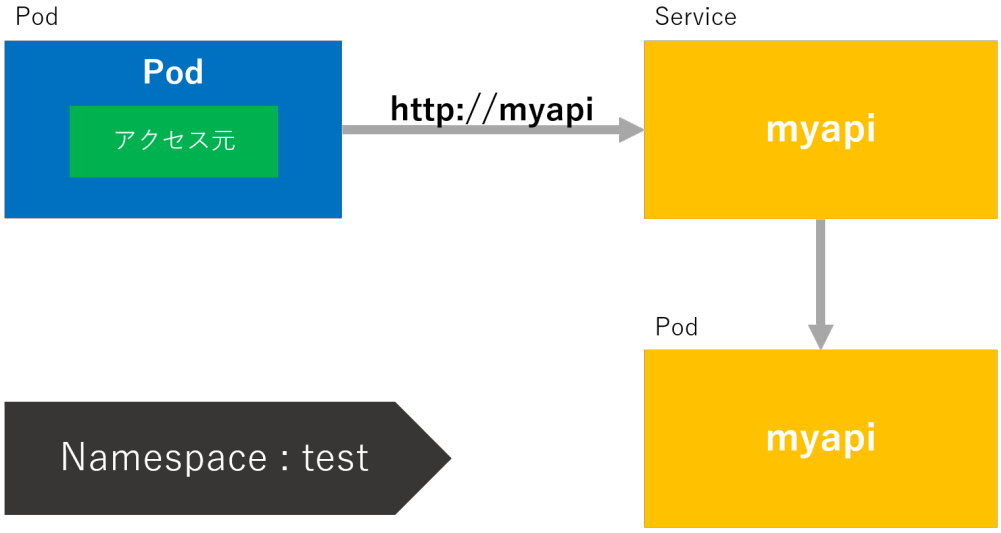

図 2.7　test 空間におけるアクセス

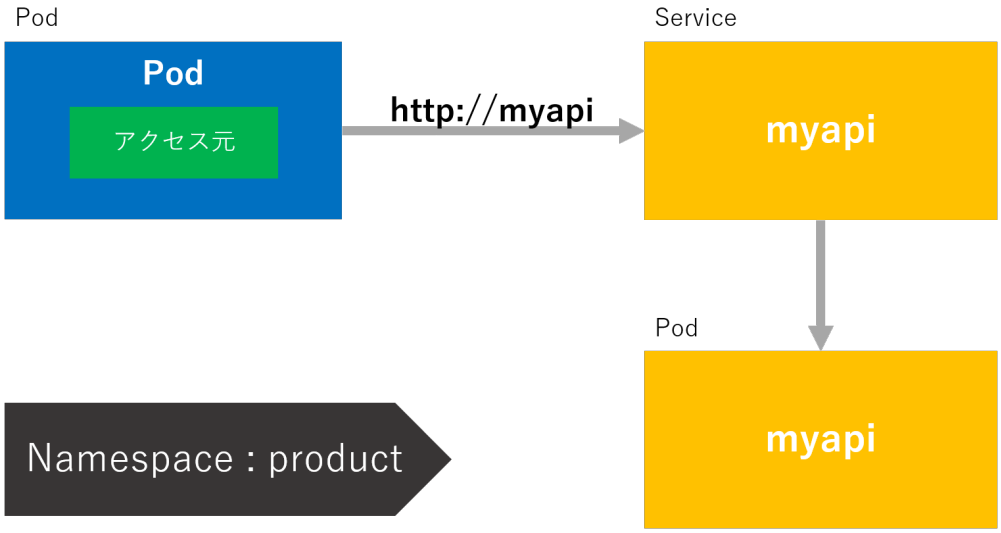

図 2.8　product 空間におけるアクセス

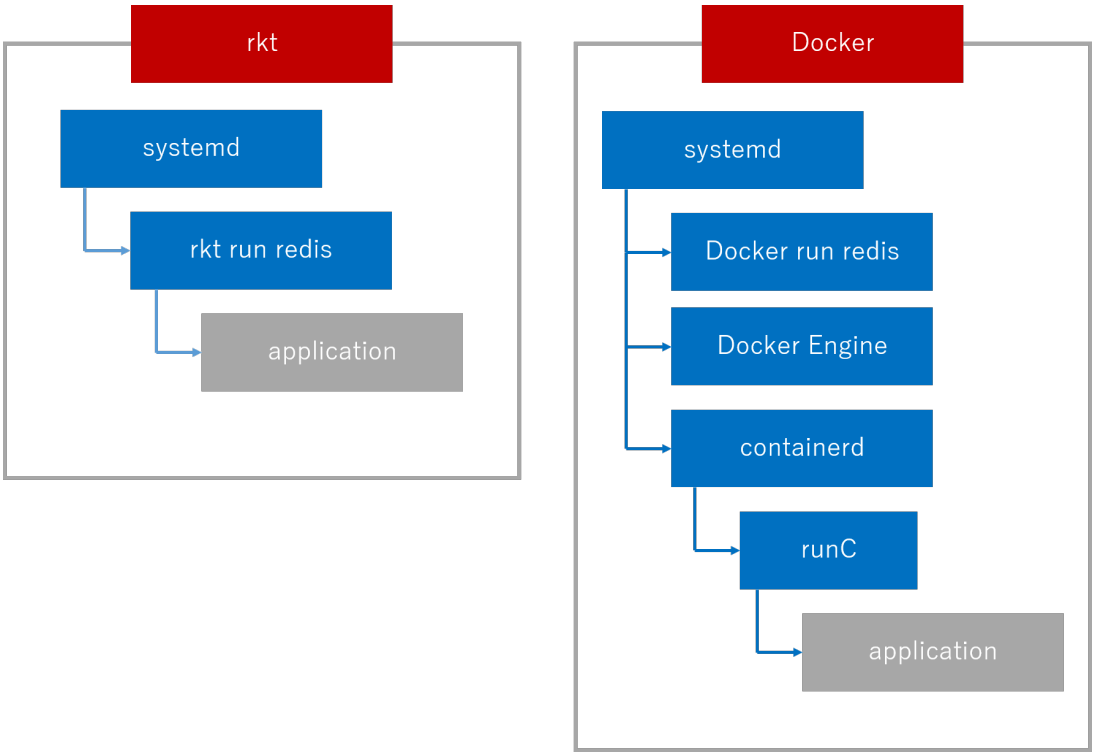

図 2.9　rkt と docker のプロセスモデル

2.6.4　container runtime

Kubernets でのノードの最下位レイヤーにはコンテナの起動および停止を行なうソフトウェアがあり，これをコンテナランタイムと呼ぶ．2018 年現在最も普及しているコンテナランタイムは Docker であるが，目的に合わせて必要なコンテナランタイムを選択することができるようになっている．Kubernetes では異なるコンテナランタイムにも容易にアクセスできるようにすることができる CRI（Container Runtime Interface）を備えており，再コンパイルを行なう必要なく，様々なコンテナランタイムを使用することが可能となっており，Docker を用いる場合よりも他のコンテナランタイムを用いた場合が良いケースも存在する．kubernetes は現在，Docker と rkt の 2 つのランタイムをサポートしている．Docker と rkt のプロセスモデルの違いを図 2.9 に示す．

Docker デーモンはコンテナ自体の実行は処理を行わず，containerd が代わりに処理しており，API 呼び出しによって runC を用いてコンテナを開始する．rkt は Docker に対して

init デーモンが存在せず，クライアントサイドからコンテナを直接起動するため，systemd，upstart などの init システムと互換性がある．

Pod

Kubernets での基本的なスケジューリング単位は「Pod」と呼ばれる．これはコンテナ化されたコンポーネントに高い抽象性を持たせることができる．コンテナ自体がホストに割り当てられることはなく，代わりに密接に関連した Pod がまとめられることになる．Pod は，一般に単一の「アプリケーション」として制御されるべき 1 つ，または複数のコンテナを表している．

この関連付けにより，関連するすべてのコンテナが同じホスト上でスケジュールされる．それらはユニットとして管理され，環境を共有しており，例えば図 2.10 のようにボリュームと IP 空間を共有しているため，単一のアプリケーションとしてデプロイおよびスケールすることができる．

例えば Wordpress アプリケーションの場合であれば，nginX と MySQL，Wordpress のコンテナが Pod として扱われる．

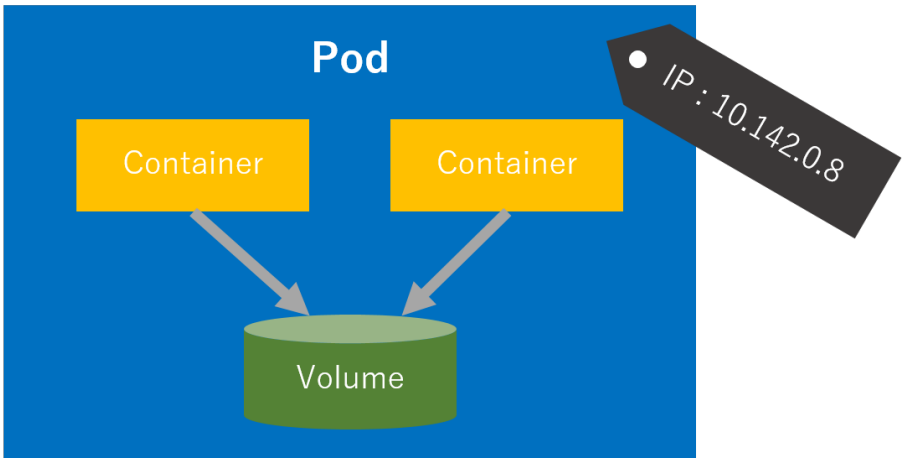

図 2.10　Pod

この Pod の生成方法として，マニフェストファイルと呼ばれる構成ファイルから生成す

ることが可能で, 生成時にリソースの使用量を指定することができる．CPU リソースの使用量には Request と Limit の 2 つを使用することができ, Request には Pod が使用できる最低のリソース使用量（保証量）を指定することができ, Limit には使用できる限界値を指定する．リソースの制限を行なうことで, 他 Pod の処理を圧迫せず, リソースの占有を防ぐことができる.

Service

Kubernetes 上での基本的なロードバランサと他のコンテナのアンバサダーとして機能するユニットとして「Service」が定義されている．Service は同じ機能を実行して単一のエンティティとして提示する論理的な Pod 群をまとめてグループ化する．これにより, すべてのバックエンドのコンテナを認識しているサービスユニットを配置して, Service がコンテナに対してトラフィックを割り当て流すことができる．外部アプリケーションは, 単一のアクセスポイントについてのみ心配する必要があるが, スケーラブルなバックエンドまたは, 必要に応じてスワップアウトできるバックエンドの恩恵を受ける．サービスの IP アドレスは安定したままで, ノードが消滅したり Pod が再スケジュールされたりしても Pod の IP アドレスの変更は抽象化される．サービスはコンテナグループへのインタフェースであり, ユーザは単一のアクセス場所以外を考慮する必要がない．サービスを展開することによって, 発見容易性を向上させ, コンテナ設計を簡素化することができる.

Replication Controller

Kubernetes には Pod のレプリカを生成し維持する「Replication Controller」の機能がある.

Pod が多すぎる場合，ReplicationController によって追加された Pod の終了が行なわれ,

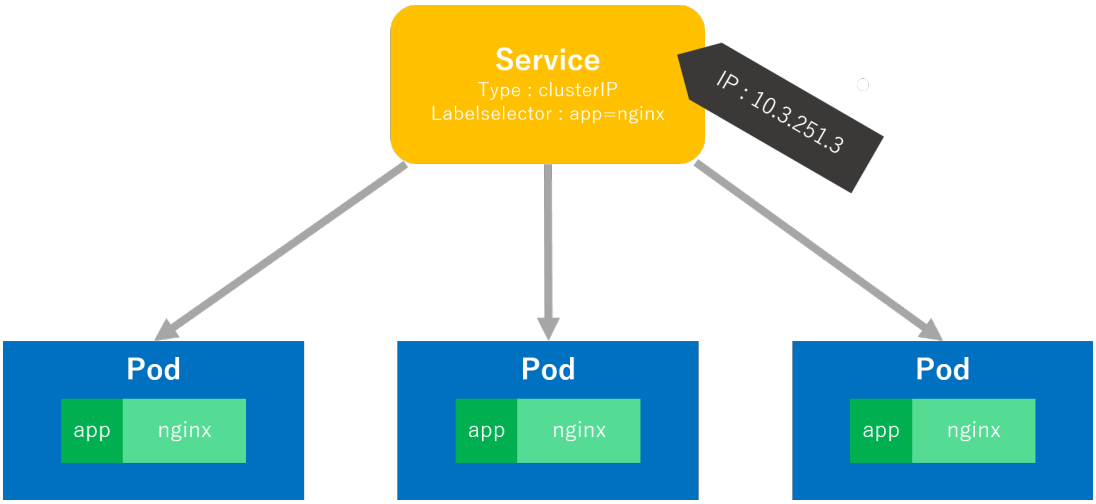

図 2.11　Service

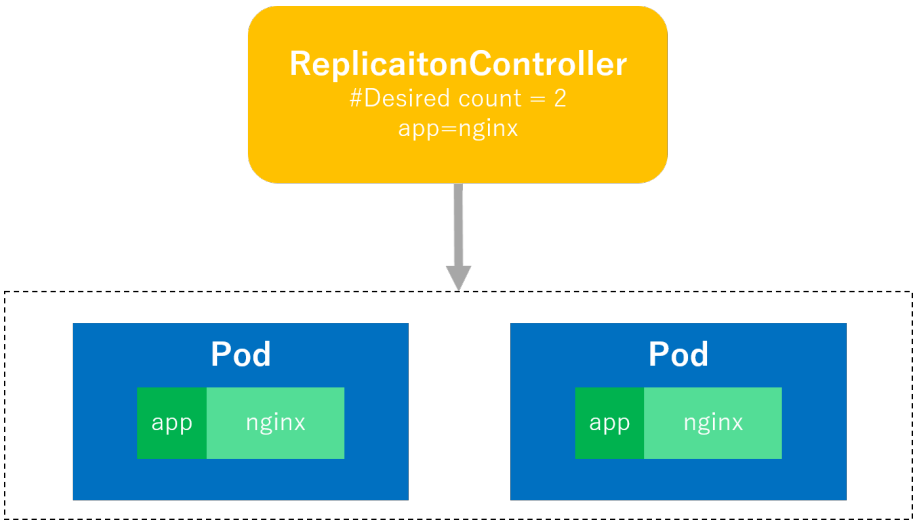

図 2.12　replication cotroller

少なすぎる場合は規定数になるように Pod の生成が行なわれる．手動で作成された Pod と異なり，Pod が失敗，削除，終了などによって消失すると自動的に再生成される．例としてカーネルのアップグレードなどのディスラプティブな行為の後であってもノード上で再生成される．この機能によって, 特定の Pod がもし消滅しても予め設定しておいたレプリカ数まで Pod が再生成され Node にデプロイされるため, アプリケーションの冗長性が保たれている．もし消滅した Pod が再起動しオンライン状態になると, Replication Controller がコンテナを 1 つ削除し, レプリカ数を保つ [19].

図 2.12 のように ReplicationCotroller に対して app=nginx のラベルを持つ Pod に Desired count として，2 を与えると 2 つの Pod が生成され，常時 2 つの Pod が起動している状態となる．

2.7　Kubernetes におけるスケーリング

Kubernetes におけるスケーリングは Horizontal Pod Autoscaler と呼ばれる Pod のスケーリング方式が存在する．これはノードを監視し，その CPU 使用率に基づいてノードのポッド数を自動的に調整するものである．Pod が処理できる量には限りがあるため，Pod を増減させることによって負荷分散を行うための仕組みである．

しかしながら，以下のケースにおいて水平スケーリングが有効ではない場合が存在する．1 つは，図 2.13 のように，Pod が分離されたノードに配置されることによって，通信のオーバヘッドが生じる場合である．ノード数に比例して通信コストが増大していくため，リソースに余裕があるのであれば，ノード内で Pod のスケーリングを行うほうが良いといえる．

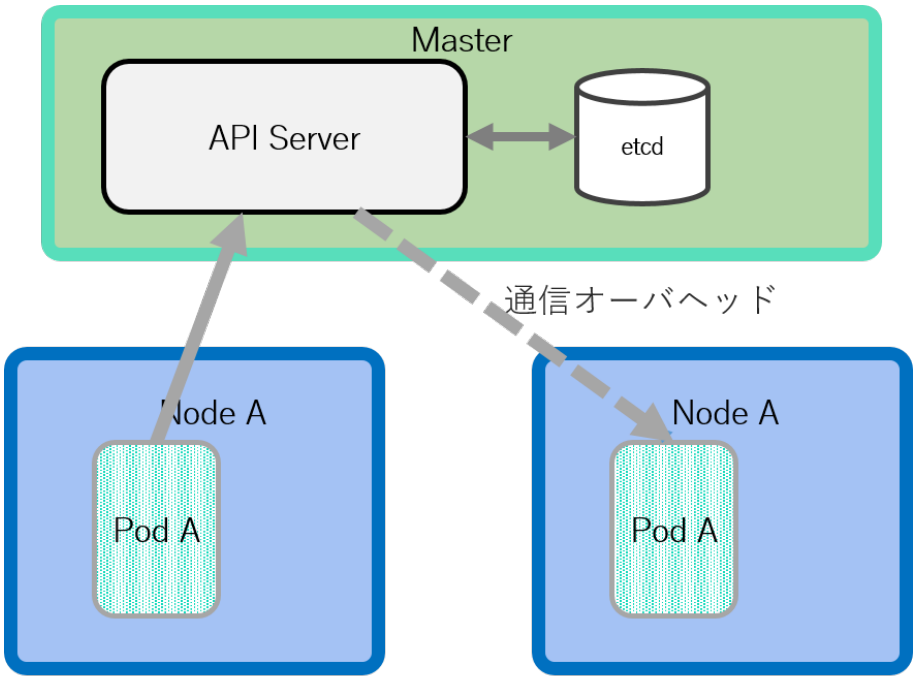

図 2.13　通信オーバーヘッドが多くなる Pod 配置のケース

2 つめは図 2.14 のように特殊なハードウェアを備えたノードでの処理が必要な Pod が通常のノードにも生成される場合である．GPU などの（CPU に対して）特殊なハードウェアを備えたマシンでの処理が行なわれることを期待した場合，kubernetes では割り振りはラウンドロビンであるため，GPU が搭載されていないノードにも Pod が複製される可能性がある．そのため，このような場合でもリソースに余裕があるのであれば，ノード内で対象 Pod のスケーリングを行うほうが良い．

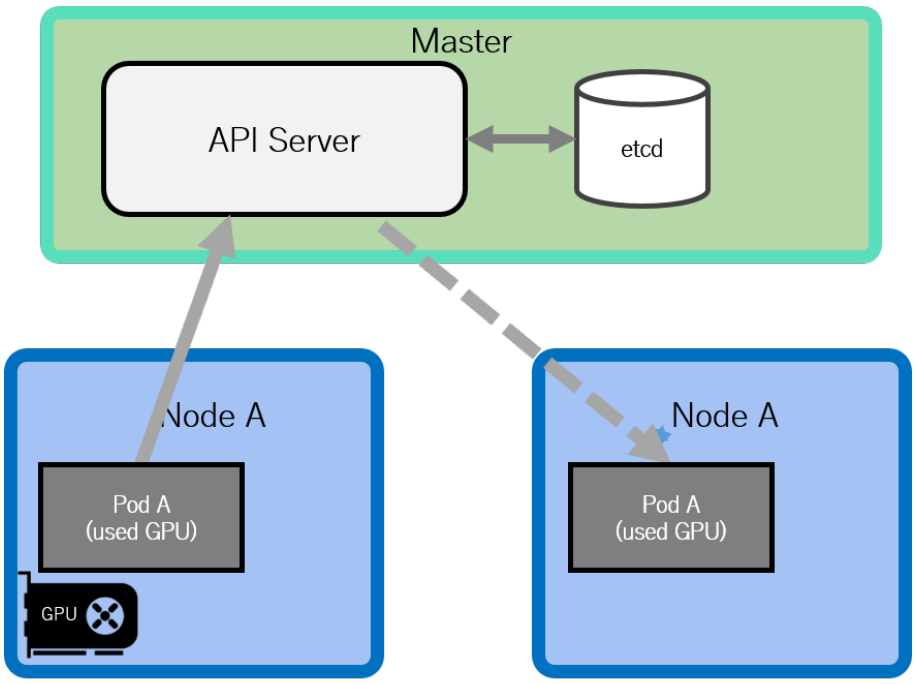

図 2.14　特殊なハードウェアを使用する Pod 配置のケース

2.8　結言

本章では, 現在の IoT におけるヘテロジニアス環境についてと本研究で用いる仮想化技術について使用される技術と用語についての概要について述べた．また，kubernetes におけるスケーリングの問題点について述べた．次章では, 基盤技術を利用して提案するスケーリング方式について述べる．

第 3 章

動的スケーリング方式

3.1　緒言

第 2 章で述べた Kubernetes には水平スケーリングと呼ばれる Pod を複数生成することによって, 負荷分散を行える仕組みがあるが, そのような状況が芳しくないケースが存在する. そのため, ノード間で負荷分散を行なうのではなく, Pod 単体のリソースのスケーリングをする必要性が出てくる. 本章では第 2 章の技術をもとに Kubernetes を用いたヘテロジニアス IoT システム向けの, Pod スケーリングを行うための実装アーキテクチャと用いる機能について記す.

3.2　スケーリング方式の構成

提案するスケーリング方式の構成を図 3.1 に示す

各コンポーネントの説明は以下である.

マスタ

クラスタを構築する際に全ノードを管理するために必要なコントロールプレーンである. マスタはマニフェストファイルに基づいて Pod を生成する.

ノード

クラスタ内でユーザアプリケーションの Pod が実行されるホストである. 実行できる Pod 数自体に限度はなく, Pod のリソースの要求量を満たすのであれば, 配置される.

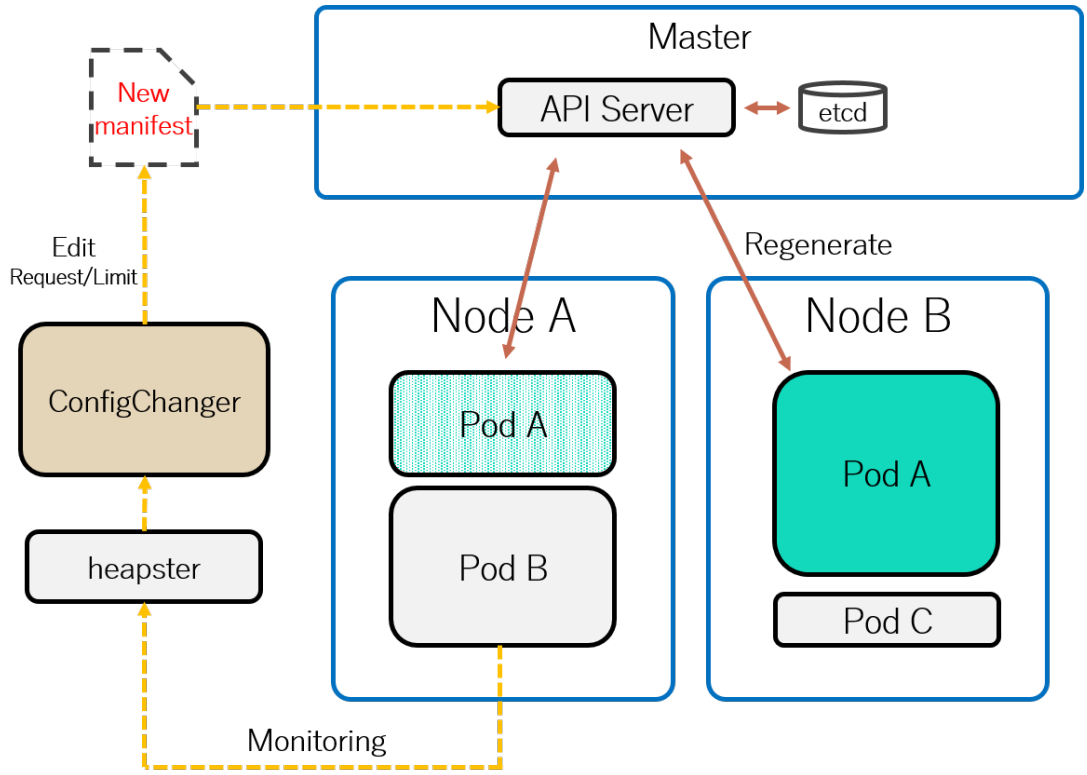

図 3.1　提案スケーリング手法の実装図

Pod

Pod はアプリケーションコンテナをまとめた単位で配備される.

Heapster

マスタおよび各ノードのモニタリングを行う Pod である. また Pod ごとのモニタリング情報も管理しており, CPU 使用率やメモリ使用率などのメトリクスを定期的に取得して, データベースへと保存を行っている. CPU 使用率を取得する際に, API Server 経由で Heapster へと Curl を用いてアクセスすることとし, CPU 使用率の取得を行う.

ConfigChanger

Heapster から各ノードの CPU 使用率を定期的に取得し, 負荷の高いノードの Pod の再生成を行う. 再生成する際には, スクリプトを用いて, 元となるマニフェストのリソースパラメータである Request と Limit を変更し, 新規マニフェストと再生成命令を APIServer に与えることでノードへの Pod 再配置を行う.

以上のコンポーネントによる処理を行うことによって, スケーリングを行い, 負荷に対してよりリソースを活用できるようにする.

クラスタリングを行う際には, taint 機能と Toreration 機能は連携して Pod が不適切なノードへのスケジュールが行われないようにする. これは 1 つまたは複数の Taint をノードに適用することで, ノードが Taints を許容しない Pod を受け入れるべきではないことを表している. 許容値は Pod に適用され, Pod に一致する Taint を持つ, ノードにスケジュールすることを許可する.

今回, Master の冗長化を行う際に Master と Node の機能の多重化は行わない. その理由として Master にはクレデンシャル情報があるため, 外部からのアクセスがされるノードと同じ機能を有した場合, セキュリティ上の欠陥となる可能性がある. 外部からノードにアクセス後, コンテナからカーネルに対してアクセスされてしまうとマスタの全権限が奪われてしまう可能性もあるため, Taint 機能を利用してマスタへは通常の Pod の配備は行わないようにしなければならない. また, そのほかにも GPU などの特殊なリソースを持つノードへの Pod の配置は GPU を用いる処理を行う必要のある Pod を配置するほうが効果的であるため, セキュリティの観点だけではなくリソースの有効活用にも有用である.

3.2.1　リソースの割当ての変更方法

Kubernetes は Pod を生成する際にコンテナイメージを利用する方法に加えて, yaml ファイルによる定義に基づき Pod の生成を行うことができる. これを Manifest と呼び, Replicaset など, どのタイプのアプリケーションとして生成するかを記述できる. この Manifest 内にリソースの要求量である Request, リソースの限界値である Limit を記述することで, 使用できるリソースの保証と, 使用できるリソースの限界量を決める. CPU における Request と Limit は 100m 単位で表し, 100m は 1 コアあたりの 10%という意味である. そのため 8 コアの場合であれば, 使用できるリソース量は 8000m となる.

CPU における 100m の正確な意味を以下に示す [8].

- 1 AWS vCPU
- 1 GCP Core
- 1 Azure vCore
- 1 Hyperthread on a bare-metal intel processor with Hyper threading

0.5 であれば, 1 CPU あたりの半分が要求量として保証されることになる. ここで用いられる精度の単位としては 1m が最小単位である. また, CPU は常に絶対量として要求され, 0.1 が表すのは, シングルコア, デュアルコア, または 48 コアマシン上の CPU での量は同じである.

割り当て変更後に Pod が再生成, 配置される際には, Request の値を参照して, 各ノードにそのリソース要求を満たすことが可能なノードへと Pod の配置が行なわれる.

3.2.2　CPU 使用率の取得

CPU 使用率の取得は API Server 経由による CPU 使用率のメトリクス取得を行う. API Server 経由で各ノードに以下の形式でアクセスし, その応答を切り出して数値として取得する.

```
curl http://127.0.0.1:8001/api/v1/proxy/namespaces/$NAMESPACE_NAME \
/services/heapster/api/v1/model/nodes/$NODE_NAME/metrics/cpuinfo
```

また, DashBoard などでの確認に備え, CPU 使用率などのメトリクスは Pod 毎に Heapster によって 1 分ごとに記録し, influxDB に記録するようにしている.

3.2.3　リソースの割り当てイベントの判断

Heapster では CPU 使用率を 100ms 毎に取得しており, その値の合計を平均した結果を 1 分間ごとにデータベースへ保存するようになっている. 急激な負荷がかかっても平均でのサンプリング結果を出すことによって, リソースの割り当てが必要以上に発生しないように留意している. リソースの割り当てを変更するイベントの条件では, 1 分間のデータをもとに

Pod に割り当てられている CPU の Limit 値に対して, CPU 使用率が超えてしまうと Pod が削除されてしまうため, Pod の Limit 値に対して使用している CPU の値が 90%以上の値になった場合, Limit 値を 2 倍に変更する．また, Pod 生成時や Pod のアップデート時などは負荷が上がる可能性があり, CPU 使用率の結果として正しくない可能性がある．そのため, Pod が生成されたタイムスタンプから 1 分後のデータから取得することによって, 同じようにリソースの割り当て量の変更が頻繁に発生してオーバヘッドにならないように変更を行う．

3.3 結言

本章では第 2 章の技術をもとに Kubernetes を用いたヘテロジニアス IoT システム向けの, Pod スケーリングを行うための実装アーキテクチャと用いる機能について述べた．次章では, 提案する動的スケーリング方式の評価について述べる．

第 4 章

評価

4.1 緒言

本章では提案したスケーリング方式の評価を行うために, Raspberry Pi 5 台を用いて Kubernets を用いたクラスタ環境を構築し, スケーリング方式のについての性能評価を述べる.

4.2 評価方法

ヘテロジニアス環境の模擬として処理負荷を与えることによって実際の性能変化を実現する. また, 評価アプリケーションとしては Web サーバを Pod として配置させ, Web サーバへのアクセスを処理負荷として与えることによって, スケーリングが行われているかどうかを確認する.

4.3 評価環境

本実験を行うにあたって用意した環境は表のとおりである. Kubernetes でのヘテロジニアス IoT システムのスケーリング方法を提案するにあたり, プロトタイプのクラスタシステムを構築し, 実装を行った. プロトタイプの構成図を図 4.1 に示す. プロトタイプの構成では, 5 台の Raspberry Pi を使用してマスタとノードを構成しており, この 5 台は Category 5e の LAN ケーブルによって接続されている. ネットワークはすべて DHCP によって IP アドレスの割り振りを行っている. 電源には USB3.0 ポートを 5 つ有する電源タップを，DC

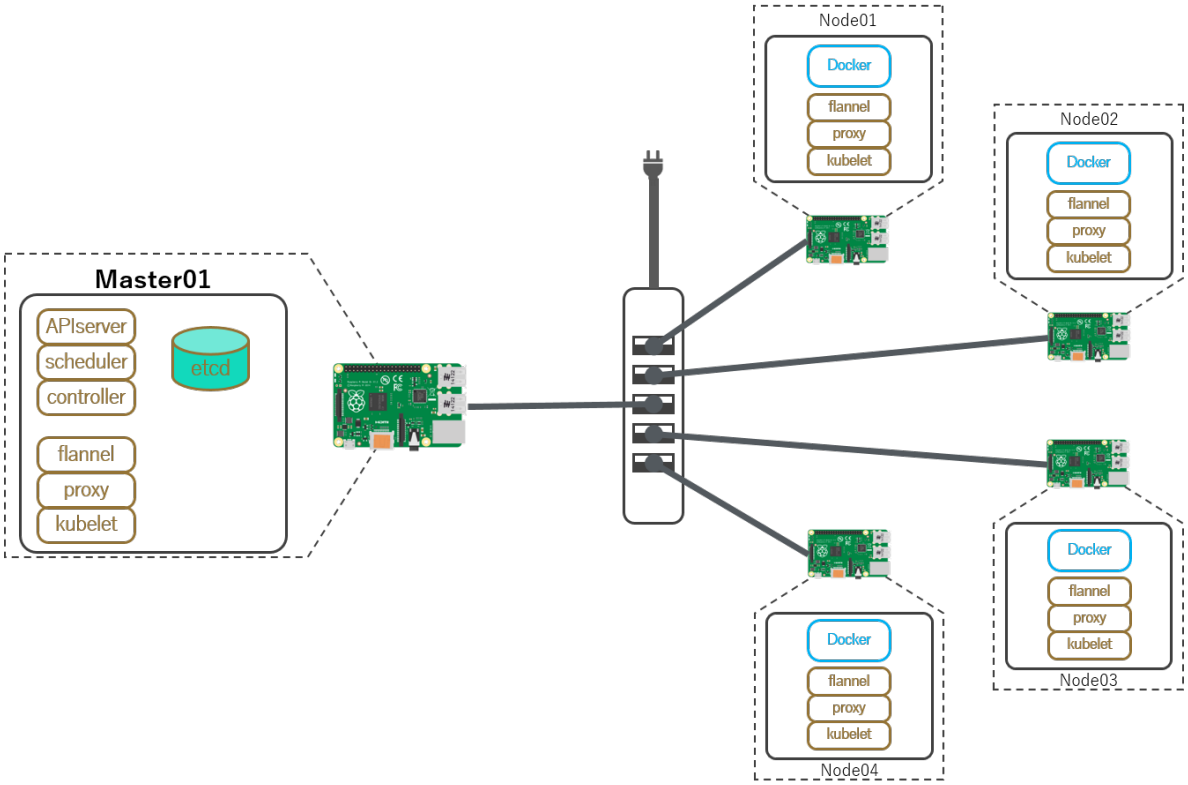

図 4.1　評価用テストベッド

電源として 5V を用いる．これは低電圧状態によって周波数が低下してしまい, 性能が発揮できない状態を防ぐ目的がある．

評価に用いるマスタで動いているモジュールは以下である．

- API server
- Scheduler
- Controller
- flannel
- proxy
- kubelet

ノードで動いているモジュールは以下である．

- Docker

- flannel
- proxy
- kubelet

表 4.1　使用機器の諸性能

CPU	ARM Cortex-A53
メモリ	1 GByte
ネットワーク	1000Mbps
電源	DC 5V
OS	HypriotOS

評価に使用する Pod の Manifest は以下である.

```
kind: ReplicationController
metadata:
  name: nginx
spec:
  replicas: 2
  selector:
    app: nginx
  template:
    metadata:
      name: nginx
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx
        resources:
          requests:
            cpu: "250m"
          limits:
            cpu: "500m"
        ports:
        - containerPort: 80
```

4.4　評価結果

評価結果を表 4.2 に示す.

表 4.2　実行時間

real time	user time	system time
7.238	5.2583	0.5324

また再配置された結果を図 4.2, 4.3 に示す. テストベッドにおける, 状態の検知, 削除,

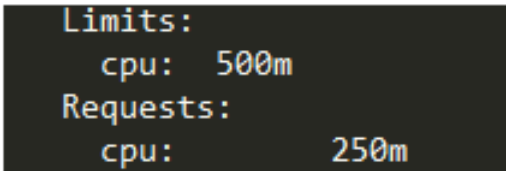

図 4.2　スケーリング前の状態

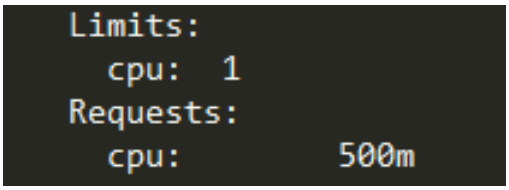

図 4.3　スケーリング後の状態

マニフェストの再生成, Pod の再生成の一連のプロセスにおける平均実時間は 7.238 秒となった.

4.5　考察

今回の実験における評価として, Raspbrry pi での実行では, 平均 7.238 秒程度という結果になった. 実際のシステムで考慮した場合, 約 7 秒では非常に遅いといえる. ハードリアルタイムシステムと呼ばれるような, 処理がデッドライン以内に終了しなかった場合に, システム全体もしくは対象物に対して致命的なダメージを与えるようなシステムである場合, マイクロ秒からミリ秒オーダーの速度が求められる. そのため実時間処理に 7 秒では満たせていないため, ハードリアルタイムのような処理には向いていないといえる.

本研究ではスクリプトベースでの環境模擬になっているため, ネイティブ言語での実装を行うことによってこの問題が改善されると考えている. 加えて, Pod を削除するのではなく, リソースの加算を kubernetes 上から追加していくような仕組みを導入することができれば,

この問題を解決できる．またライブマイグレーションのような仕組みがあれば, Pod を削除した際でも同一のリソース内でのスケーリングであれば, ローカルリソース内に処理の進行度を保存しておき, 再生成した際に処理を途中から再開することによって, より効率的 に処理を行うことができる.

4.6　結言

本章では, 提案スケーリング方式についての性能評価を行い, 結果の考察を行った．リソースのマニフェストの変更と Pod の再生成が行われていることを確認し, 約 7 秒で一連のプロセスが終了することを確認した.

第 5 章

結論

近年, IoT(Internet of Things) と呼ばれ, モノのインターネットという新しいアーキテクチャが普及しつつある. 様々なモノ, 機械, 人間の行動や自然現象は膨大な情報を生成しており, これらの情報を収集して可視化することができれば様々な問題が解決できるといわれている. 見ることや聞くこと, 触ることができる情報に加えて加速度センサーなどのそれらに該当しないセンサー情報も数値化され収集可能になっている. 従来のように人間がパソコン類を使用して入力したデータ以外にモノに取り付けられたセンサーが人手を介さずにデータを入力し, インターネット経由で利用できるようになる [1].

IoT をビジネスに活用する企業は増加しており, 適応分野も多様化が進んできている. ITR による調査によるとモノのトラッキング, 工場生産の最適化, エネルギー消費の最適化をはじめとして, 多岐に及ぶ分野への投資が進んでおり導入企業の多くが効果を実感してることが明らかになっている. IoT の市場は活用企業が増加するとともに, 1 社あたりの投資額も増大することから市場規模は 2017 年の約 4850 億円から 2020 年に約 1 兆 3800 億円へと急速に拡大すると予測している [2].

こういった中でフォグコンピューティングに注目が集まってきている. その理由として, 従来のクラウドコンピューティングによるサーバでの集中処理方式は, 今日の IoT システムで発生する大量のデータやネットワークの輻輳を考慮して設計されていないため現実的では無いことがあげられる. 以前では IoT に接続されていなかった数十億ものデバイスが毎日 2 エクサバイト以上のデータを生成しており, 2020 年までに 500 億近くの「モノ」がインターネットに接続されるといわれており膨大な帯域幅が必要となる. またリアルタイム処理などの厳しいレイテンシ要件を満たすことができない. それに対してフォグコンピューティング

は, 少ないレイテンシと帯域幅効率, レジリエントなエンドユーザサービスを提供することでインフラストラクチャの効率を高めることを目指す新しいパラダイムだとされているためである [5]. このため, 昨今, エッジ側での処理の分担が求められている.

しかしながら, IoT などに用いられるエッジデバイスである組込みシステムなどはクラウドなどに用いられるサーバなどに比べると極端に性能が劣っており, リソースの制約が厳しいことが一般的に知られている. 少ないリソースを有効に活用するためには効率的なアプリケーションの配置, リソースアロケーションなどが求められている. しかしながら, デバイスのヘテロジニアス性に加えて, 温度などの外部環境によって CPU の性能が左右されるため, 想定しているパフォーマンスが維持されない可能性がある. また, エッジデバイスで用いられる組込み機器では前述したリソース制約の観点から, 各アプリケーションが処理を完遂できるように, リソースの確保を行うことが重要である. そのため, 予想される負荷に対して必要なリソース量を見積ることで, 負荷に対応できるようにする. しかし見積もり負荷が高すぎてしまうと, ピーク時以外には利用していないリソースが大量に発生してしまい全体の処理効率として無駄が多くなってしまう. しかしながら, 見積もり性能が少なすぎた場合, 負荷が見積もり負荷を超えてしまうと, その間システムは処理が追いつかず要求された時間内での処理が終わらない可能性が生じる. 結果としてサービスレベルの低下を招くこととなる. Pod のアプリケーションが必要とするリソース量を活用するためには, Pod 自体のリソース量を動的に変化させ, ピーク時には必要なリソース量を増大させることで大きな処理負荷にも応答できるようにする仕組みが必要である.

コンテナ型仮想化を用いる理由として, ハードウェアリソースをソフトウェアから分離できることに加えて, パッケージ化されたソフトウェアを複数のハードウェアアーキテクチャ上で実行することができる. コンテナ型仮想化はハイパーバイザ型仮想化に比較して, 少ないリソース消費, 非常に高速な構築やインスタンス化などの利点がある. 加えて, コンテナは仮想マシンなどの仮想イメージよりも小さいため, より高いアプリケーションおよびサービスの割り当てを可能としている. これらより, IoT およびフォグコンピューティングにおいて非常に有効であることが言える.

本論文ではこのような機構としてコンテナ型仮想化を用いた Kubernetes[8] でのスケーリング方式を提案し, ヘテロジニアスな環境での IoT システムにおいて, 各々のエッジデバイスがアプリケーションの負荷に合わせて動的にリソースの使用量を変更し, ヘテロジニアス IoT システム全体の可用性を向上させることを提案した.

ubernetes は Google 社が開発した, アプリケーションコンテナのデプロイ, スケーリング, マネジメントを行うオープンソースソフトウェアである. さまざまなインフラストラクチャにわたる分散したホストのクラスタ間で, アプリケーションコンテナのデプロイ, スケーリング, マネジメントを自動化するためのプラットフォームとして提供することを目的としている. 今回はコンテナ型仮想化を用いている kubernetes を用いてヘテロジニアス IoT システムの模擬を行なった.

提案手法では, Pod のモニタリングを常時行なっており, 100ms ごとにメトリクスとして CPU 使用率を取得する. その結果を 1 分ごとにデータベースに登録している. CPU 使用率が 90%以上になった場合に Pod の再生成を行なうこととし，値は 2 倍に上昇させることとした.

その結果として Raspberry Pi 5 台を用いて kubernetes クラスタを構築し, ヘテロジニアス IoT システムのプロトタイプのテストベッドを構築し評価した. Pod が必要とするリソースの Request を Pod の CPU 使用率に応じて変化させ, Request 値が再定義後の Pod が生成されることを確認した. テストベッドにおける, 状態の検知, 削除, マニフェストの再生成, Pod の再生成の一連のプロセスにおける平均実時間は 7.238 秒となった.

謝辞

本研究において，指導教員として日頃から懇切丁寧にご指導してくださった岩田誠教授に心より深謝申し上げます．学士時代からお世話になり，修士でも引き続き岩田研究室に在籍させていただけて非常にありがたく思います．修士 1 年の時には，PDPTA での論文作成の際に，お忙しいにも関わらず，日が変わっても添削につきあっていただきました．非常に申し訳ない気持ちと同時に，自分の英語における単語の選択能力やライティング能力の欠如が浮き彫りになり，今後の自分の英語能力の向上を目指す良い機会になりました．発表練習にも発音について細かくご指導をいただき, 英語に対しての姿勢を再考する良い機会でした．初めて海外へ行くということで, 身の回りのアドバイスから, 現地での翻訳までしてくださり, 大変お世話になりました．

また就職活動では，大学院に進学してから，いろいろと身勝手な行動が目立ったにもかかわらず，ご助力いただけたこと非常にありがたく感じています．新しい研究テーマの提案や, 吟味など, 自分の研究を進めるにあたって，専攻分野外にも関わらず，アドバイスを沢山いただいたりと感謝してもしきれません．

また，学士時の卒業論文時の副査をお引き受け頂き，引き続き本論文の副査をお引き受け頂いた，栗原徹准教授，並びに松崎公紀准教授に感謝の意を表します．研究テーマを変更したにもかかわらず，承諾して頂きありがとうございました．

研究室での同期として，日ごろからご協力を頂いた，梅嵜佑樹氏，渋田広樹氏，に感謝いたします．特に梅嵜氏には，学士 1 年時から数多度お世話になりました．特に, 卒業のために必要な科目を履修しておらず，梅嵜氏が気づいて連絡をしていただけたことによって今この修士論文を書くことができていると言っても過言ではありません．渋田氏には，研究室の用事を率先して行なっていただき, 本当にありがとうございました．手間がかかるような事務作業でも積極的にしてくださり, 研究室での生活が円滑に行えたのは渋田氏のおかげです．

また，研究室での後輩として，日頃からご協力を頂きました，修士 1 年の齋藤あかね氏，

田原匡浩氏，福田和馬氏，学士 4 年の穴井志穂氏，奥井里永子氏，山崎尚之氏，吉本大輔氏，和田悠伸氏，学士 3 年の汐見興明氏，楠田健太氏，長野寛司氏，下出千晴氏，柳田海志氏に感謝いたします．皆さんの今後のご活躍を期待しております．

最後にはなりましたが，常日頃支えてくださった，親愛なる皆様に心より御礼申し上げます．

参考文献

[1] https://mono-wireless.com/jp/tech/Internet_of_Things.html, "IoT とは? | IoT : Internet of Things（モノのインターネット）の意味," 2018 年 2 月 1 日.

[2] https://enterprisezine.jp/article/detail/9917, "2017 年の IoT 市場規模は 4,850 億円、2020 年には 1 兆 3,800 億円へと急拡大――ITR 予測, " 2018 年 2 月 2 日.

[3] https://www.asahi.com/articles/ASKBW4CQ4KBWPPTB008.html," ＩｏＴで高齢者見守り　空調調節・睡眠把握、遠隔でケア," 2018 年 1 月 30 日.

[4] https://www.cisco.com/c/dam/en_us/solutions/trends/iot/docs/computing-overview.pdf, "Fog Computing and the Internet of Things: Extend the Cloud to Where the Things Are," 2018 年 1 月 30 日.

[5] H. Chang, A. Hari, S. Mukherjee, T. V. Lakshman, "Bringing the cloud to the edge", Proc. IEEE Conf. Comput. Commun. Workshops (INFOCOM WKSHPS), pp. 346–351, May 2014.

[6] Maarten Botterman. Internet of things: an early reality of the future internet.In Workshop Report, European Commission Information Society and Media, 2009.

[7] Bruzual Balzan, Daniel, "Distributed Computing Framework Based on Software Containers for Heterogeneous Embedded Devices," Service Design and Engineering, Aalto University, October 2017.

[8] Google. Inc, "Kuberntes," https://kubernetes.io, December 2017.

[9] Asad Javed, "Container-based IoT Sensor Node on Raspberry Pi and the Kubernetes Cluster Framework," ICT Innovation, Aalt University, June 2016.

[10] 叶 如絵, 田胡 和哉, "コンテナ型仮想化環境向き負荷予測システム「Tetris」の開発," 第 78 回全国大会講演論文集, vol.1, pp. 11–12. May 2016.

[11] Maarten Botterman. Internet of things: an early reality of the future internet.In

Workshop Report, European Commission Information Society and Media, 2009.

[12] Daniele Miorandi, Sabrina Sicari, Francesco De Pellegrini, and Imrich Chlamtac, "Internet of things: Vision, applications and research challenges," Ad Hoc Networks, pp. 1497–1516, 2012.

[13] Mario Di Francesco, Na Li, Long Cheng, Mayank Raj, and Sajal K Das. "A framework for multimodal sensing in heterogeneous and multimedia wireless sensor networks," In World of Wireless, Mobile and Multimedia Networks, IEEE International Symposium, pp. 1–3. 2011.

[14] P. Levis, S. Madden, J. Polastre, R. Szewczyk, K. Whitehouse, A. Woo, D. Gay, J. Hill, M. Welsh, E. Brewer, and D. Culler, "TinyOS: An Operating System for Sensor Networks," pp. 115–148. Springer Berlin Heidelberg, Berlin, Heidelberg, 2005.

[15] Adam Dunkels, Bjorn Gronvall, and Thiemo Voigt, "Contiki - a lightweight and flexible operating system for tiny networked sensors," In Proceedings of the 29th Annual IEEE International Conference on Local Computer Networks, LCN 04, pp. 455–462, Washington, DC, USA, IEEE Computer Society, 2004.

[16] http://www.vsolution.jp/vmware/virtualization_guide/01/, " 仮想化とは何か," 2018 年 2 月 2 日.

[17] http://www.kernelthread.com/publications/virtualization/, "An Introduction to Virtualization, "2018 年 1 月 30 日.

[18] https://www.digitalocean.com/community/tutorials/an-introduction-to-kubernetes, "Large-scale cluster management at Google with Borg," 2018 年 1 月 31 日.

[19] https://static.googleusercontent.com/media/research.google.com/ja//pubs/archive/43438.pdf, "Large-scale cluster management at Google with Borg, " 2018 年 1 月 30 日.

[20] https://mono-wireless.com/jp/tech/Internet_of_Things.html, ”IoT とは? | IoT : Internet of Things（モノのインターネット）の意味,” 2018 年 2 月 2 日.