複数ドメインにまたがる階層化ネットワークを考慮した **Dataflow** プラットフォームに関する研究

Studies on Dataflow Platform Considering a Hierarchical Network Over Multiple Domains

石原 真太郎

2020 年 12 月 23 日

概要

Internet of Things (IoT) デバイスから時々刻々と生成される大量の Dataflow を処理する Dataflow アプリケーションの研究開発が進められている．その中で，Dataflow アプリケーションの運用フェーズにおいては，デバイス・クラウド間での通信遅延，トラフィック量に対するクラウドでの課金やモバイル通信費用による運用コストの増大が問題となっている．一方で，デバイスの近傍にコンピューティングリソースを配置し，そこで一部の処理を実行することで低遅延応答を実現可能なエッジコンピューティング技術の研究開発が進められている．しかし，エッジ環境では使用可能なコンピューティングリソースが限られており，全ての処理をエッジに展開することは難しい．主要なエッジコンピューティングプロジェクトでは，2 から 3 層のネットワーク階層を想定したアーキテクチャとなっており，Dataflow アプリケーションにおいては，遅延，トラフィック量，リソース量を考慮して，適切な階層に Dataflow アプリケーションを構成するソフトウェアコンポーネントを展開することが求められる．

Dataflow アプリケーション開発者にとっては，その開発をサポートする Dataflow プラットフォームもまた重要な役割を持つ．Dataflow プラットフォームでは，既存のコンポーネントを組み合わせるだけなど，比較的簡単に Dataflow アプリケーションを開発できる．アプリケーション開発者の負担を減らすため，Dataflow プラットフォームにおいても，遅延，トラフィック量，リソースを考慮したアプリケーションの展開機能が提供されることが求められる．

これまでに，さまざまな遅延，トラフィック量，リソースなどを考慮したコンポーネント配置手法が提案されている．しかし，多くの場合，計算量を低減するため，ターゲットとする範囲の限定や遅延の除外などいくつかの条件が想定されており，そのままでは実環境では活用できない．本研究では，コンポーネント配置とコンポーネント間通信を分離することで，現実的に適用可能な手法を提案する．具体的には，コンポーネント配置先を地理的位置に基づいてクラスタ化し，クラスタ内における比較的短い通信遅延を省略することで，計算量を抑えながら，遅延，トラフィック量，リソースのすべてを考慮したコンポーネント配置を実現する．さらに，P2P オーバレイによる柔軟なコンポーネントのリソース情報の収集およびそれを活用した動的ルーティングを活用して，クラスタ付近におけるリソース状況や遅延を考慮したコンポーネント接続を実現する．結果として，Dataflow アプリケーションの運用時の課題となっている遅延やトラフィック量の削減が実現され，さらなる Dataflow アプリケーションの普及に貢献できる．

目次

表目次

図目次

1 緒論

1.1 研究の背景

世界に広く普及するインターネットは，より身近なものとなっており，パーソナルコンピュータのみならず，スマートフォンをはじめとして，環境センサ，自動車，家電など，さまざまな「モノ」がインターネットに接続されている．これらを実現する技術は，Internet of Things (IoT) と呼ばれ，医療，製造など，多くの分野で IoT に関する研究開発が進められている．

インターネットを通したサービスは日々増加しており，その中でも特にクラウド事業の発達はめざましく，IoT においてもクラウドサービスの利用が必要不可欠となってきている．クラウドサービスにおいては，これまで，1 つのシステムの中にすべてのモジュールが含まれるモノリシックなシステム開発が進められていた．しかし，このような開発形態では，モジュール同士が密結合状態となり，一部機能の変更が全体へ波及する可能性があり，開発速度の低下を招いていた．サービスの開発からリリースまでの速度向上のため，モノリシックアーキテクチャからマイクロサービスアーキテクチャへとサービスの開発形態が変わってきている．マイクロサービスアーキテクチャでは，小さなソフトウェアコンポーネントの集まりからシステムやアプリケーションが構成される．これにより，モジュールの変更，新規のアプリケーション開発，スケールイン・スケールアウトが容易となっており，さらなるクラウドサービスの機能向上が期待されている．

IoT においては，複数のソフトウェアコンポーネントからなるアプリケーション全体，またはその一部をクラウド上で実行する形態が採られる．センサやアクチュエータなどのエンドデバイスが配置されるデバイスネットワークと，クラウドが接続され，センサから出力される Dataflow とも呼ばれる，ストリームデータの収集や解析がクラウド上で行われる．本論文では，IoT アプリケーションを含む Dataflow を処理するアプリケーションを Dataflow アプリケーションと呼ぶ．スマートシティなど生活の向上や環境の改善を目的として，例えば，Augmented Reality (AR)，Mixed Reality (MR)，スマート交通，スマート家電など，さまざまな Dataflow アプリケーションの研究開発が進められている．その中で，Dataflow アプリケーションの運用フェーズにおいて，いくつかの課題が出てきている．AR, MR のストリーミングにおいては，没入感を阻害しないため，スマート交通では，事故を未然に防ぐため，それぞれ低遅延での応答が求められる．しかし，多くの場合，Dataflow アプリケーションはクラウド上に展開されるため，デバイス・クラウド間の遅延により，遅延要件を満たせない [44]．スマート交通では，ドライブレコーダを用いたバス運行状況の把握などの研究開発が進められている．IoT デバイスでは多くの場合，月額制の SIM などを活用してデータを転送するが，ドライブレコーダのキャプチャデータを転送する場合，必要な帯域が大きくなり，運用費用が増大する [24]．また，転送データ量の増加に合わせて，クラウド上において，入力データ量やストレージ量に対する課金額が増加し，アプリケーションの運用コストの増大が懸念される．さら

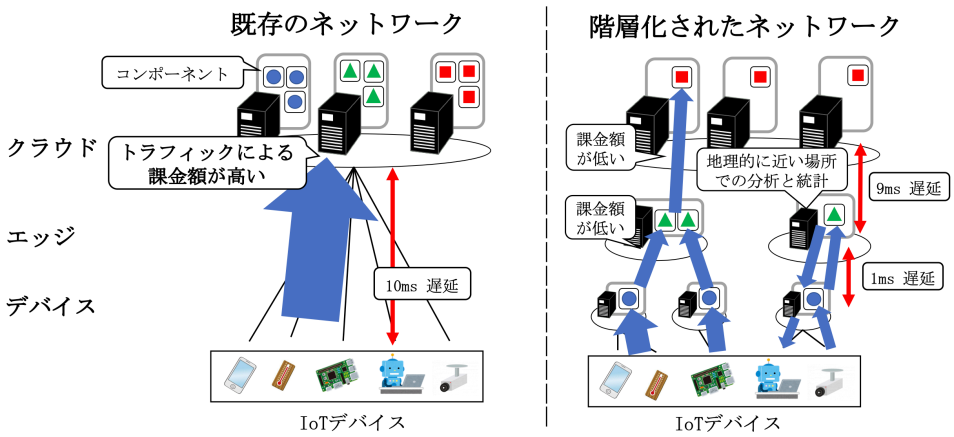


図 1: エッジコンピューティング環境を活用した遅延とトラフィック量低減

に，スマート交通やスマート家電を活用したアプリケーションなど，個人情報を扱う場合，プライバシの観点から，そのままのデータをクラウドに送信できない可能性がある．

一方で，IoT デバイスのインターネットへのアクセス部分や，モバイルトラフィックを収容する局舎など，デバイスの近傍であるエッジに，コンピューティングリソースを配備し，そこでアプリケーションの一部の機能を提供することで，遅延やトラフィック量の低減を目指すエッジコンピューティング技術の研究開発が進められている．Dataflow アプリケーションにおいても，これを活用し，本来クラウド上で実行されるマイクロコンポーネントのうち，一部のマイクロコンポーネントをエッジで，実行することにより地理的にデバイスに近い位置でのデータ分析や統計，応答処理が可能となり，長期間にわたって運用されるアプリケーションの運用コストを抑えることや，リアルタイムアプリケーションの遅延要件を満たすことができる．例えば，図 1 のように，クラウドとデバイスの中間であるエッジに一部のマイクロコンポーネントを配置することで，IoT デバイスで生成されたトラフィックをクラウド事業者によって課金される前に削減でき，アプリケーションの運用コストを下げられる可能性がある．また，地理的に近い位置でセンサデータを解析し，応答することで低遅延での制御が可能となる．以上から，エッジ/クラウドなど，階層化されたネットワークを考慮して，Dataflow アプリケーションを展開することは，これらの技術が生活に浸透する上で非常に重要となる．しかし，エッジやデバイス上で活用できるリソース量は限られており，すべてのコンポーネントをそこに配置することはできない．さらに，地理的に広く分散されているすべてのコンピューティングリソースに対して，遅延，運用コスト，リソース量を考慮して，Dataflow アプリケーションを構成するコンポーネントを展開することは，計算複雑性の観点から困難となる [54, 52]．また，アプリケーション開発者にとっても，これらの実現は容易ではなく，開発コストの増大を招く．

開発者の負担を抑えるため，アプリケーションの実装および展開をサポートする Dataflow プラットフォームの研究開発が進められている [2, 32, 16]．Dataflow プラットフォームでは，

Dataflow を処理するための既存のソフトウェアコンポーネントを GUI 上で組み合わせるだけで，比較的簡単に新しい Dataflow アプリケーションを構築・展開できる．このような Dataflow プラットフォームにおいて，遅延，運用コスト，リソース量を考慮して，階層化されたネットワーク上に Dataflow アプリケーションを展開できれば，アプリケーション開発者の負担を抑えつつ，エッジを活用して，アプリケーションの遅延要件および運用コストの要件を満たすことができる．

ここで，既に説明したように，エッジを活用することで，地理的に広く分散されている IoT デバイスから転送される大量の Dataflow を適切に処理できる可能性がある．しかし，エッジにあたる局舎や収容局などの Point of Presence (POPs) は，日本国内においては，NTT 東日本，NTT 西日本，米国においては，AT & T や Verizon など，異なるドメインによって管理されている．これらを柔軟に活用し，IoT デバイスの位置情報を考慮して，適切なエッジに Dataflow アプリケーションを展開するには，複数ドメインにまたがって Dataflow プラットフォームを構築することが求められる．

1.2　既存の Dataflow プラットフォームにおける課題

既存の Dataflow プラットフォームとして，Amazon Kinesis Data Streams [2]，Azure Stream Analytics [32]，Google Cloud Dataflow [16] などが挙げられる．また，Azure や Google では，Azure IoT Edge [31] や Google Cloud IoT Core [18] など，エッジでの IoT アプリケーション処理をサポートした Dataflow プラットフォームもクラウドの機能を拡張する形で提供されている．しかし，それぞれ特定のクラウドサービスに密接に結びついており，例えば，蓄積したデータを簡単に他のサービスに移行できないなど，サービス選択の自由度が制限される．現状は試行的なサービスのため，安価に利用することができるが，利用者の増加にともなう価格の上昇や，サービス提供者側の都合による API の変更などが発生する可能性があり，運用コスト面のコントロールが困難になるなど，ベンダロックインの課題を抱えている．

Satyanarayanan の提案手法 [43] では，オープンソースソフトウェアを活用するアプローチを採っており，ベンダロックインを回避しながら，プラットフォームを構築することを可能としている．しかし，遅延，運用コスト，リソース量を考慮したコンポーネントの配置はサポートされていない．Zhang ら [54]，Wang ら [52] によって，コンポーネント配置手法が提案されている．しかし，いずれの提案手法においても，計算量を抑えるため，遅延を考慮しないなどの条件が設定されており，現実的でない．本論文では，コンポーネント配置とコンポーネント間通信を分離し，コンポーネント配置においては，データセンタ内などの比較的短い遅延を省略し，Teranishi ら [48] によって提案されているネットワーククラスタのアイデアを活用し，クラスタ単位に抽象化してコンポーネントを配置することで，現実的な配置を実現する．

Teranishi ら [49] は，クラスタ構造を考慮した Dataflow プラットフォームを提案している．また，これは特定のクラウドにも依存しておらず，Pub/Sub 基盤を活用したコンポーネント間

通信手法を提案している．これにより，コンポーネント間が疎結合になっており，柔軟にスケールイン・スケールアウト可能な構成となっている．しかし，クラスタ情報を考慮したコンピューティングリソース管理方法およびコンポーネント管理手法は提案されていない．クラスタ単位に抽象化して，コンポーネント配置先を決定できたとしても，その配備先となるコンピューティングリソースが，クラスタ情報と結びつけて管理されていない場合，適切なクラスタのコンピューティングリソース上にコンポーネントを配備できない．加えて，CPU 使用率やメモリ使用量といったリソース情報を考慮した配送手法も提案されておらず，コンポーネント配置後のリソース量を考慮した配送には，そのままでは活用できない．リソース量を考慮した配送を実現する方法として，配送前にすべてのコンポーネントのリソース情報を収集し，その情報を基に配送することが考えられる．しかし，毎回リソース情報を集めると，その分トラフィック量や遅延が増大する．ここで，例えば，配送先候補となるクラスタにのみ絞って，リソース情報を収集するといったことも考えられる．一方で，Nakashima ら [36] によって提案されているバスサービスを対象としたアプリケーションの場合，配送先候補となるクラスタは，バスの経路によって異なるため，柔軟にリソース情報を収集する範囲を変更できることが求められる．

1.3 研究目的

本研究では，複数ドメインにまたがる階層化されたネットワーク環境を考慮した Dataflow プラットフォームを実現を目指し，階層化ネットワークを考慮したコンポーネント管理手法，遅延，運用コスト，リソース量を考慮し，適切なコンポーネント配置先ネットワークレイヤ決定する手法，コンポーネント間通信手法の実現を目指す．

本論文では，計算量を抑えつつ，現実的に活用可能な遅延，運用コスト，リソース量を考慮して，Dataflow アプリケーションを展開する手法として，コンポーネント配置と，コンポーネント間通信の 2 つに分離する手法を提案する．コンポーネント配置においては，データセンタ内など，比較的遅延の小さい箇所を省略することで，コンポーネントの配備先候補をクラスタ単位に抽象化する．，さらに，ネットワーククラスタのアイデアを活用し，計算対象となるコンピューティングリソース群から構成されるクラスタを 3 階層程度に限定することで，計算量を抑えながら，遅延，運用コスト，リソース量を考慮したコンポーネント配置を実現する．

ここで，コンポーネント配備先となるコンピューティングリソースが，クラスタ情報と結び付けられていない場合，適切なクラスタを決定できても，実際に配備することができない．それを解決するため，オープンソースソフトウェアである Kubernetes を用いて，クラスタ情報をラベルとして，コンピューティングリソースと紐づけて管理する手法を提案した．また，遅延，運用コスト，リソース量を考慮して，コンポーネント配備先ネットワークレイヤを決定するアルゴリズムを実装し，比較的単純なアルゴリズムでも，上記 3 点を考慮してコンポーネントを配置できることを確認した．

コンポーネント間通信においては，柔軟に収集範囲を変更可能なキャッシュ情報を活用して，リソース情報を考慮して，単一のコンポーネントにメッセージを配送する Anycast 手法を提案した．キャッシュ情報を用いて配送する場合，キャッシュが有効な場合は，少ないトラフィック量および低遅延での配送が可能であるが，キャッシュが無効だった場合，トラフィック量と遅延ともに増大する可能性がある．本論文では，コンポーネントの使用可否をキャッシュ対象とし，コンポーネント予約メッセージの送信頻度に対するキャッシュを用いた Anycast 手法の性能を評価した．結果として，コンポーネント予約メッセージ送信頻度がそれほど多くない場合であれば，トラフィック量および遅延を抑えながら，リソース量を考慮してコンポーネント間を接続できることを確認した．

これら 3 つの提案手法により，本論文でターゲットとしているようなバスサービスをターゲットとした Dataflow アプリケーションなど，アプリケーションの展開後，そのまま運用を続けるようなケース，つまり，展開/削除の頻度が低いアプリケーションに対しては，トラフィック量，運用コスト，リソース量を考慮したアプリケーション展開が実現できる．以上から，本研究により，現実的に実現可能な Dataflow アプリケーション展開が可能となり，アプリケーションの展開/削除の頻度が低いアプリケーションに限られるが，エッジ環境の活用が求められるリアルタイムアプリケーションの展開や，既存アプリケーションの運用コストの削減が実現でき，さらなる Dataflow アプリケーションの普及に貢献できる．

1.4　研究の位置づけ

さまざまな場所に設置された IoT デバイスから生成された Dataflow を処理する Dataflow アプリケーションの研究開発が進められている．その運用フェーズにおいては，遅延や運用コストが問題視されており，エッジコンピューティング技術の活用による，問題解決が期待されている．主要なエッジコンピューティングプロジェクトとして，Multi-access Edge Computing (MEC) [23]，Fog Computing [9]，Cloudlet [14] などが挙げられる．どのプロジェクトでも，計測・制御環境からの距離に基づいて，2 から 3 層のネットワーク階層を想定し，同じようなネットワークアーキテクチャを定義している．遅延や運用コストを抑えるには，これらのネットワークアーキテクチャと同じように階層化されたネットワークを考慮して Dataflow アプリケーションを展開することが重要となる．

一方で，Dataflow アプリケーションの開発をサポートする Dataflow プラットフォームの研究開発が進められている [2, 32, 16]．また，それらをエッジまで延伸したプラットフォームも提供されている [31, 18]．これらを活用することで，遅延や運用コストを抑えながら，比較的簡単に Dataflow アプリケーションを展開できる可能性がある．しかし，それらはオープンソースとしては提供されていないことや，特定のクラウドサービスに強く依存していることから，複数ドメインにまたがる広域なプラットフォームの構築には活用できない．Satyanarayanan [43] の提案ではオープンソースソフトウェアを活用するアプローチを採用しており，ベンダロック

インを回避しながら，プラットフォームを構築することを可能としている．本研究でも同様のアプローチを採用し，複数ドメインにまたがる階層化 Dataflow プラットフォームの実現を目指す．

これまでに，Zhang ら [54]，Wang ら [52]，Bahreini ら [6] によって，アプリケーション要件を考慮して，エッジコンピューティング環境上に Dataflow アプリケーションを展開する手法が提案されている．比較的短時間に移動するモバイル端末とエッジリソース間の遅延を考慮することは難しく，Zhang ら [54] や Wang ら [52] は，計算量を抑えて配置パターンを計算するために，ターゲットとするリソース数の限定，遅延の除外などの条件を加えている．Bahreini ら [6] は，ユーザの移動による遅延の変化を考慮した Dataflow アプリケーション展開手法を提案している．しかし，リソース上限を考慮していない．コンポーネント配置問題は整数計画法などを用いて解を得る場合が多く，遅延，運用コスト，リソース量を考慮した配置パターンの探索は計算が複雑になり，解を得ることが難しいと考えられる．それに対して本研究では，Teranishi ら [48] によって提案されている地理的位置に基づいたネットワークのクラスタリングに基づいて，データセンタ内における通信遅延などの些細な遅延を省略し，コンポーネント配置箇所をクラスタ単位に抽象化して計算することで，計算量を抑えながら，遅延，運用コスト，リソース量を考慮したコンポーネント配置を実現する．さらに，コンポーネント配置機能とコンポーネント間通信機能を分離し，コンポーネント間通信にて，モバイルデバイスの状態やその付近のクラスタのリソース状況をモニタリングすることで，それに基づいた動的なコンポーネント間でのルーティングを実現する．

コンポーネント間ルーティングにおいては，コンポーネントのリソース状況を把握し，適切にルーティングすることが求められる．しかし，リソース状況の把握が求められる範囲はターゲットとする環境によって異なる．例えば，Zhang ら [54] は無線アクセスネットワークにおける少数のサーバから構成されるエッジサーバクラスタをターゲットとしており，対象の範囲は比較的狭い範囲となる．Ascigil ら [5] は，単一ドメインで構成されるプライベートクラウドエッジ環境をターゲットとしており，Zhang ら [54] と比べると広域となる．さらに，我々がターゲットとする Nakashima ら [36] によって提案されているバスサービスの向上を目指した乗降者カウントアプリケーションのような地理的に広がって動作するアプリケーションの場合，モニタリング対象となる範囲は，バスの経路の近くに位置するクラスタとなり，バスによってその範囲が異なる．サービスメッシュにも活用される etcd [13] などの分散データベースを用いて，すべてのクラスタのクラスタで情報を収集し，全体で共有することで，すべての範囲の情報を取得できるが，デバイスネットワークやエッジネットワークでは，リソース量や通信帯域が限られる．本研究では，各ノードの負荷を抑えながら，適切な範囲でリソースを収集するために，P2P ベースのアプローチによる柔軟なリソース収集機能を実現する．

以上から，本研究の位置づけとして，Dataflow アプリケーションの運用フェーズにおける遅延や運用コストの低減を目的として，以下に示す提案により，階層化ネットワークを考慮した Dataflow プラットフォームにおける課題の解決を目指す．

- 複数ドメインにまたがる Dataflow プラットフォームのアーキテクチャの提案
- オープンソースソフトウェアとコンテナ技術を活用したクラスタを考慮したリソース管理方法の提案
- コンポーネント配備先ネットワークレイヤ決定手法の提案
- 柔軟なリソース収集とそれを用いたコンポーネント間通信手法の提案

1.5 論文の構成

既に述べたように，本論文では 4 つの提案をもって，複数ドメインにまたがる Dataflow プラットフォームを実現する．第 2 章では，提案する Dataflow プラットフォームのアークテクチャについて述べ，提案するコンポーネント配置および通信手法の概要を説明する．さらに，その実現に必要となるリソース管理手法について説明する．第 3 章では，第 2 章で説明した想定する階層化ネットワークにおいて，コンポーネントを配置するネットワークレイヤの決定手法について説明する．また，第 4 章では，コンポーネントのリソース情報の収集手法およびそれを活用したコンポーネント間通信手法について説明する．最後に第 5 章にて，本論文の結論を述べる．

2 階層化ネットワークを考慮した Dataflow プラットフォーム

2.1 緒言

第 1 章で述べたように，本研究では，計算量を抑えながら Dataflow アプリケーションを構成するコンポーネントの配置パターンを計算するため，Teranishi ら [48] によって提案されている地理的位置に基づいたネットワークのクラスタリングを用いて，クラスタ単位でコンポーネントを配置する手法を提案する．また，コンポーネント配置とコンポーネント間通信を分離することで，クラスタ付近のリソース状況に応じた動的なコンポーネント接続を可能とする．これらの実現には．クラスタ情報を考慮したコンピューティングリソースとコンポーネントの管理および，コンポーネント配置とコンポーネント間通信の連携機構が求められる．

Teranishi ら [49] によって，階層化ネットワークを考慮した Dataflow プラットフォームが提案されている．彼らの提案では，特定のクラウドサービスに依存せず，Pub/Sub 基盤を活用したコンポーネント間通信手法を提案している．Pub/Sub 基盤を活用することで，コンポーネント間が疎結合になっており，柔軟なコンポーネント間の繋ぎ替えを可能としている．本研究が提案するコンポーネント配置とコンポーネント間通信の分離でも活用可能なアーキテクチャとなっている．一方で，クラスタ情報を考慮したコンピューティングリソースおよびコンポーネント管理手法は考えられていない．

本研究では，Teranishi ら [49] の提案に基づいて，階層化ネットワークを考慮した Dataflow プラットフォームの実現を目指す．本章では．Teranishi ら [49] の提案に基づいた想定する階

層化ネットワーク，クラスタ情報を考慮したコンピューティングリソースおよびコンポーネント管理手法，コンポーネント配置とコンポーネント間通信の連携機構について説明する．以降，2.3 節では，Teranishi ら [49] の提案に基づいて，階層化ネットワークおよび本研究で想定するクラスタについて説明する．2.4 節では，Dataflow アプリケーションの定義，それを構成するコンポーネントについて，バスサービスに着目した IoT アプリケーションの例を用いて説明する．2.5 節では，Dataflow プラットフォームの概要およびコンポーネント配置とコンポーネント通信を連携させるために導入するオーケストレータについて述べる．2.6 節では，コンテナ管理技術を活用したクラスタ情報を考慮したコンピューティングリソースおよびコンポーネント管理手法について述べる．2.7 節では，Pub/Sub 基盤を活用したコンポーネント間接続について説明する．最後に 2.8 節にて本章のまとめを述べる．

2.2　既存の階層化ネットワークを考慮した Dataflow プラットフォーム

本節では，本研究がベースとする階層化ネットワークを提案している Teranishi ら [49] の手法について説明する．またその提案における階層化ネットワークについて説明する．そのあとで，既存手法における課題および本研究と異なる点について説明する．

Teranishi ら [49] は，既存の Dataflow プラットフォームがエッジコンピューティングのような環境を考慮できておらず，その利点を活用できないことに着目し，エッジコンピューティング環境における Dataflow の処理手法を提案している．具体的には，トピックベース Pub/Sub を拡張し，デバイスから生成されるデータストリームの処理をエッジなどに柔軟に割り当てる手法を提案している．提案手法では，アクセスネットワーク，エッジネットワーク，クラウドネットワークの 3 階層のネットワークを定義している (図 2)．彼らの提案では，それぞれのネットワークに属するコンポーネントは全てひとつの P2P ネットワークに属する．P2P オーバレイネットワークは，Chord$^{\#}$ [45] を用いて実装している．また，提案手法では，Teranishi ら [48] の提案手法を用いており，各ノードは，トピック情報とクラスタ情報の両方からなるキーを持ち，オーバレイ上では，そのキーが辞書順でソートされる．これにより，同一クラスタに属するノードはオーバレイ上では連続して並ぶようになっており，クラスタ間を跨いだホップを防ぐことができる．例えば，エッジネットワークに配備されているコンポーネント間で通信する際には，そのネットワーク内に閉じて通信可能となっている．

Teranishi ら [49] は，トピックベース Pub/Sub を拡張し，新たにインデックス値という概念を導入することで，それをベースとした柔軟なデータストリームの割り当てを実現している．この手法では，各ノードはインデックスの範囲を持ち，送信側でインデックス値を適切に設定することで，柔軟に送信先，つまりデータストリーム処理割り当て先を決めることができる．例えば，クラウドネットワーク，エッジネットワークに配備されているコンポーネントがそれぞれ 0.0 ～ 0.5，0.6 ～ 1.0 という範囲を持たせておくことで，クラウド/エッジ間のネットワーク帯域に余裕がない場合に，インデックス値を 0.6~1.0 の間の値にすることで，エッジネット

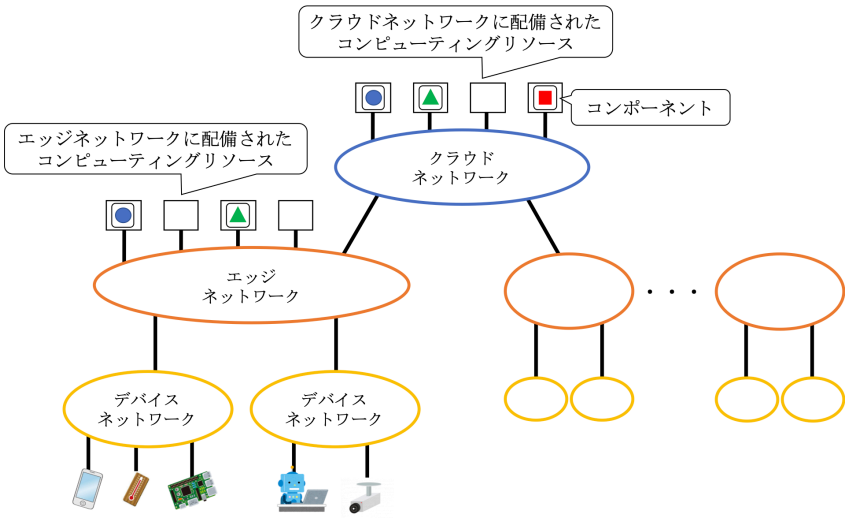


図 2: 既存手法における階層化ネットワークの例 [49]

ワークに配備されているコンポーネントへの処理のオフロードが可能となる．また，同様な手法で，あるコンポーネントへの処理の割り当てが過多となっている場合に，動的にスケールアウトすること，処理が少なくなった場合にスケールインすることができる．一方で，提案手法では，クラウドネットワークなどに配備されたコンピューティングリソースの管理方法，およびそのリソース上へのコンポーネントの配備手法については提案されていない．クラスタ情報とコンピューティングリソースの情報を結びつけて管理されていない場合，必要な箇所にコンポーネントを配備することができない．ここで，利用するコンポーネントをあらかじめすべてのコンピューティングリソース上に手動で展開しておくことが考えられるが，オーバヘッドが大きく現実的ではない．また，コンピューティングリソースについても，一般的には，クラウドサービスとして提供されている VM 環境やコンテナ環境を利用することが考えられる．この場合，再現なく使えるわけではなく，利用したコンピューティングリソースの分だけ課金され，運用コストが増大する．そのため，コンポーネントだけでなく，コンピューティングリソースについても同様にアプリケーション要件に基づいて，必要となるクラスタにおいて，スケールイン・スケールアウトすることが求められる．これらを実現する上で，コンピューティングリソースおよびコンポーネントをクラスタ情報を考慮して管理することが必要となる．一方で，コンピューティングリソースの管理においては，クラウドごとに API が異なるなど，プラットフォーム管理者にとって，それらをクラスタ情報を考慮しながら，管理することは煩雑となる．本研究では，インテグレーション層を導入することで，実装コストを抑えながら，クラスタ情報を考慮したコンピューティングリソースの管理手法を提案する．

2.3 階層化ネットワーク

本論文では，3 層のネットワークレイヤを想定しており，それぞれクラウドネットワーク，エッジネットワーク，デバイスネットワークとする．クラウドネットワークは，クラウドコンピューティング環境にあるサーバ群を接続する．また，計測，制御環境から遠く離れたデータセンタに位置する．エッジネットワークは，ネットワークサービスプロバイダが持つ POPs などにあるサーバ群を接続する．デバイスネットワークは計測，制御対象である IoT デバイスを接続する．それぞれのネットワークレイヤに接続するコンピューティング環境には，コンテナを動作させる環境があり，Dataflow アプリケーションを構成する各コンポーネントはコンテナとして展開される．

コンピューティングリソースはデバイスネットワークやエッジネットワークに地理的に分散して点在しており，コンポーネントも地理的に分散して配備できる．デバイスネットワークやエッジネットワークはクラウドネットワークに比べると通信遅延が短い．さらに，生成されるトラフィックをネットワーク内で収容できれば，実効帯域に対して課金される額を抑えられる．一方で，デバイスが持つコンピューティングリソースは限られており，すべてのコンポーネントをデバイスに配置することはできない．処理する内容やそれに必要なリソースに従って，コンポーネント配置箇所として適切なネットワークレイヤを考える必要がある．階層化されたネットワーク上にコンポーネントを分散配置させる詳細なメリットについては以下で説明する．

2.3.1 通信費用の低減

多くの場合，センサから生成される膨大な量のストリームデータは，解析され必要最低限のデータに集約される．このようなデータ量の削減はクラウド環境へ転送する際にかかる通信費用の低減にも活用できる．例えば，定額制のモバイル通信では，月に利用可能なデータ通信量に対して月の利用料を支払う．この場合，通信量が利用可能な量を超えると，自動的に課金される，または，通信速度が制限される．クラウドサービス [31, 18] では，クラウド環境から送信されるデータ量およびクラウド環境に送信されるデータ量に対して課金される．エッジコンピューティングサービスはまだ実証実験の段階であるが，通信事業者はおそらく同様な料金設定を採用するため，複数のネットワークレイヤにまたがる通信の低減により，IoT アプリケーションのランニングコストの削減が期待できる．

2.3.2 低遅延応答

IoT のひとつのユースケースとして低遅延応答が求められる [44]．例えば，機械制御では出来る限り早く発生したイベントを検知しなければならず，1 ms 以下のネットワーク遅延が求められる．他にも，AR/MR アプリケーションの場合，低遅延ネットワークによりユーザエクスペ

リエンスが向上するため，ユーザの利用時間，利用回数を増加させることができる．Dataflow プラットフォームは，Dataflow アプリケーションのニーズに合わせて，計測，制御対象であるデバイスの近くにコンポーネントを配置することで，アプリケーション開発者に低遅延応答のメリットを提供できる．階層化ネットワークの特性を活かすためには，Dataflow アプリケーションを構成するコンポーネントをアプリケーションの要求に応じて，適切なネットワークレイヤに配置する必要がある．Dataflow アプリケーションの要件の記述方法およびコンポーネントを適切な箇所に配置する方法は，第 3 章で述べる．

2.3.3 ネットワーククラスタとその構造

本研究では，地理的に分散した多くのクラスタを含む広域な環境にまたがってプラットフォームを構築することを目的としている．すべてのコンピューティングリソースから最適なノードを探索することは，計算複雑性の観点から，難しく思えるが，Teranishi ら [48] によって提案されているネットワーククラスタのアイデアを導入することで問題を簡単化できる．

クラスタの例およびネットワーククラスタについて説明する．想定されるクラスタの例を図 3 の左に示す．大阪や東京に位置するデータセンタに配備されたコンピューティングリソース群 (クラスタ) をそれぞれクラウド 1, クラウド 2 とする．また，神戸や京都の局舎や収容局に配備されたコンピューティングリソース群 (クラスタ) をそれぞれエッジ 1, エッジ 2 とする．さらに，神戸市内に展開されたバス内に配備されたセンサ群が接続するデバイスをそれぞれデバイス 1, デバイス 2 とする．ここでは簡単に 2 台のバスを想定している．日本の場合，図 3 の右に示すようなネットワーククラスタが考えられ，各クラスタには表 1 に示すようなクラスタ ID を付与することが考えられる．

例えば，ある会社が神戸市でのサービス提供を想定し，アプリケーションを展開する場合，ターゲットとなる範囲は図 3 の右の枠内となる．この場合，コンポーネント配置先候補となるクラウド，エッジ，デバイスネットワークのコンピューティングリソースは次のように限定することができる．

- デバイスネットワークのコンピューティングリソース: 「/JP/West/Osaka/Kobe」の条件でマッチするリソース
- エッジネットワークのコンピューティングリソース: 「/JP/West/Osaka/Kobe」の条件でマッチするリソース
- クラウドのコンピューティングリソース: 「/JP/West」の条件でマッチするリソース

この場合，配置を検討する 3 階層は，クラウド 1, エッジ 1, デバイス 1 またはデバイス 2 となる．デバイスについては，詳細は，次章で述べるが，入力情報に基づいて 1 つに限定される．ここで，クラウドネットワークのコンピューティングリソースについては，通信遅延の観点から「/US」でマッチするリソースは選択されるべきではないが，日本国内であれば，クラウド間ではそれほど遅延は大きくないため，「/JP」の条件でマッチするリソースでも良い可能性があ

表 1: クラスタ名とクラスタ ID の例

階層情報	クラスタ名	クラスタ ID
クラウド	クラウド 1	JP/East/Osaka
クラウド	クラウド 2	JP/West/Tokyo
エッジ	エッジ 1	JP/West/Osaka/Kobe
エッジ	エッジ 2	JP/West/Osaka/Kyoto
デバイス	デバイス 1	JP/West/Osaka/Kobe
デバイス	デバイス 2	JP/West/Osaka/Kobe

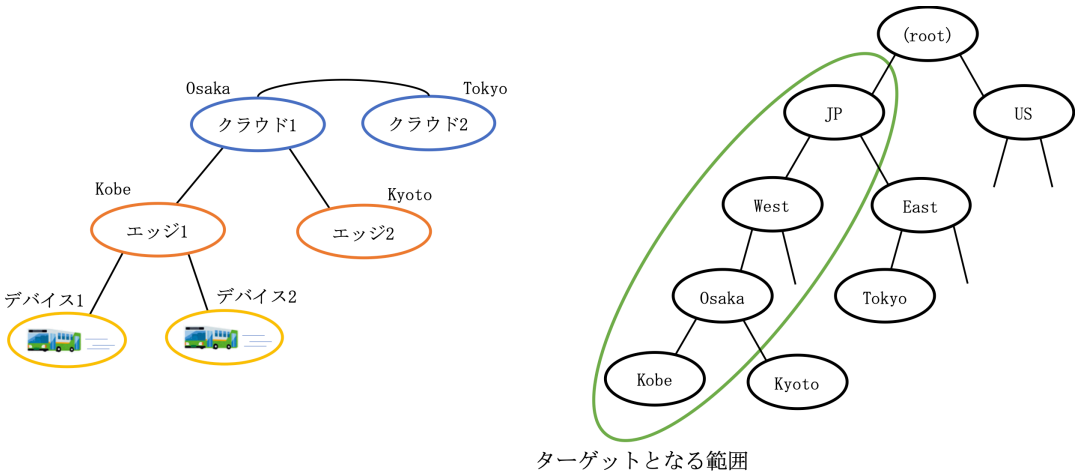


図 3: 日本でのネットワーククラスタの例

る．クラスタ分類の粒度など，クラスタ情報をどのように定義するかは別途議論が必要であるが，上記のようにネットワーククラスタを考慮することで，計算対象となる範囲を狭め，計算量を低減させることができる．これを実現する上で，プラットフォームでは，クラスタ ID とリソースを結びつけるなど，コンピューティングリソースが属するクラスタを管理しながら，広域に分散した複数のクラスタを統括的に管理することが必要となる．

2.4　Dataflow アプリケーションの例

本節では，Dataflow アプリケーションの構築方法について，バスサービスをターゲットとした IoT アプリケーションの例を用いて説明する．Dataflow アプリケーションは図 4 に示すような Dataflow グラフとして表せる．Dataflow グラフは複数の Dataflow アプリケーションを含められる．各ノード c_i は Dataflow アプリケーションを構成するコンポーネントを示す．

Dataflow グラフは分岐ノードと合流ノードにより，Dataflow の分割と複数の Dataflow の統合を表現できる．ただし，本研究では，計算の複雑化を防ぐために Dataflow グラフは有向非循環グラフ (DAG) であることを想定している．機械学習のようにデータベース (DB) に保存されたデータを再利用することで，結果の精度を向上させるためにループを含むことが望まれるケースもあるが，DB コンポーネントを入力ノードとした Dataflow を追加することで同様な処理を表現できる．

バスサービスで展開される IoT アプリケーションとして，Nakashima ら [36] によって提案されているシステムを想定する．ターゲットとするバスは神戸みなと観光バス株式会社 [57] によって運用されている．Nakashima ら [36] の提案で記述されているアプリケーションは図 4 に示す Dataflow グラフで表せる．このアプリケーションでは，バス内に設置されたドライブレコーダから生成される動画のストリームデータを用いてバスの乗降客数をカウントする．カウント処理の手順は，次の 4 つのコンポーネントに分割できる: (1) 乗降客数カウントの誤検知のうち False Positive を減らすためにバス停を検知するコンポーネント，(2) コンピューティングリソースの消費量を考慮して処理コストを削減するためにフレームレート変換を行うコンポーネント，(3) 動画から乗客の挙動を抽出するコンポーネント，(4) 乗客の乗降を推定し，それをカウントするコンポーネント．コンポーネント (3) はトラフィック量を抑えるためにバスがバス停付近に位置する場合にのみコンポーネント (4) に結果を転送する．ここで，コンポーネント (2) と (3) の出力データは他のアプリケーションに流用できる．例えば，乗客の急激な動きは急ブレーキや急ハンドルといった危険運転の検知に流用できる可能性がある．図 4 は上述の 2 つのアプリケーションをひとつの Dataflow グラフに含めている．

ターゲットとするバスの経路は神戸市から芦屋市にかけて設定されている．この経路の付近には計 11 つの NTT 西日本が持つ POP がある [47]．NTT DOCOMO のようなモバイル通信事業者は彼らのアクセスポイントを NTT 西日本が持つ POP に配置しており，アクセスポイントに流れるトラフィックは，NTT 西日本の持つ有線伝送路を活用して各都市にあるコアネットワークに伝送される [35]．5G サービスが可能になると，通信事業者から新たなサービスが提供されるだろう．モバイル通信は仮想回線上で転送されるが，新たなサービス上では，それらは NTT 西日本がもつ POP に配備されるサーバに直接転送される．これはローカルブレイクアウトとも呼ばれ，このようなサービスはエッジコンピューティングの可能性を広める．以降の節では，通信事業者の持つ POP でローカルブレイクアウトサービスが実現可能になることを想定し，議論を進める．

2.5 Dataflow オーケストレータ

本論文で議論する Dataflow プラットフォームのアーキテクチャを図 5 に示す．Dataflow プラットフォームは Dataflow アプリケーションを実行可能な環境を提供する．すでに議論したように，Dataflow オーケストレータではコンポーネント管理とコンポーネント間通信を連携さ

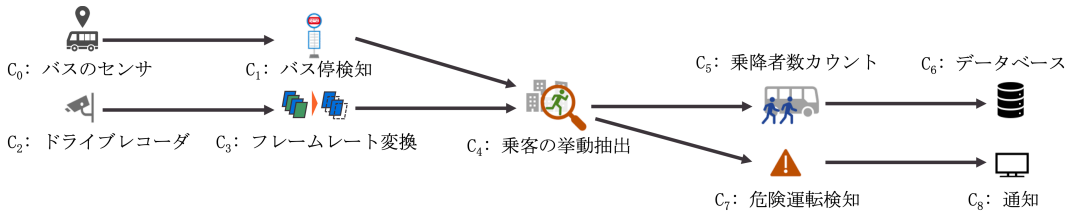


図 4: コンポーネント配置手法における Dataflow グラフの例

せる．既存のコンポーネント管理プラットフォームでは，コンテナイメージとしてコンポーネントの機能を柔軟にカスタマイズする機能や顕教に応じて挙動を調整するための起動オプションを提供している．しかし，既存ミドルウェアを組み合わせながら，Dataflow グラフを扱うにはオーケストレーションレイヤが求められる．本論文ではこれを Dataflow オーケストレータを呼ぶ．オーケストレータの主なタスクは次の 3 つである: (1) 配置先ネットワークレイヤの選択，(2) 階層化ネットワーク上への Dataflow アプリケーションを構成するコンポーネントの配備，また必要のないコンポーネントの解放，(3) Pub/Sub 基盤を用いた柔軟なコンポーネント接続．(1) の機能については第 3 章で説明する．ここで，上記のオーケストレータとしての機能の実装には，他にもヘテロジニアスなクラウド API の統合や，クラウド間メッセージングのサポートなどが求められ，開発コストが高くなる．そこで，既存の機能を組み合わせることで，実装コストを低減する．以降，2.6 節，2.7 節では，それぞれ (2)，(3) の機能の実現手法について説明する．

2.6　コンポーネント管理手法

本研究が対象とするネットワークアーキテクチャは，クラウドネットワーク，エッジネットワーク，デバイスネットワークの 3 つのネットワークレイヤを含むが，多くの場合，各ネットワークレイヤに属するコンピューティングリソースは，クラウドサービスプロバイダ，通信事業者，バス会社など異なるリソースプロバイダから提供される．適切なネットワークレイヤにコンポーネントを配置するためには，異なる管理ドメインをまたいで階層化ネットワークを構築し，それらのコンピューティングリソースを統括して管理することが求められる．Wang ら [53] は，エッジコンピューティングのアーキテクチャとして，MEC，Fog Computing，Cloudlet を比較している．MEC，Fog Computing，Cloudlet はすべて広くインターネット上で展開されるアプリケーションをターゲットとしている．しかし，MEC，Fog Computing は企業により推進されており，クラウド事業やネットワーク機器ベンダへのロックインが懸念される．反対に，Cloudlet はオープンソースソフトウェアを使うアプローチを採用しており，ベンダロックインを避けながら複数のエッジ/クラウドネットワークにまたがって，コンポーネント管理基盤を実現することができる．以上から，本研究では，Cloudlet のアーキテクチャを採用する．ここで，virtual machine (VM) based Cloudlets [42] や，複数クラウドにまたがって Cloudlet

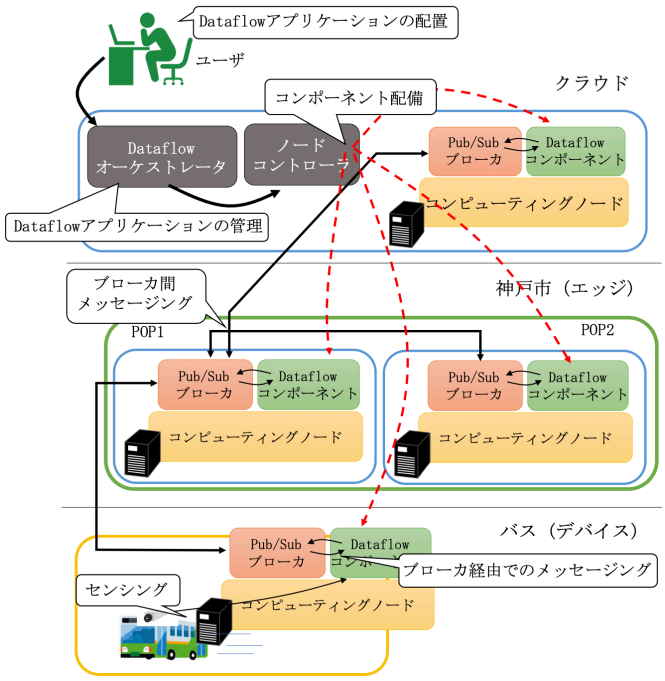


図 5: Dataflow プラットフォームのアーキテクチャ

環境を構築するための Cloudlet を適応するための OpenStack の拡張である OpenStack++ [21] が提案されている．一方でエンドノードデバイス上に VM をデプロイする場合，オーバヘッドが大きく，配備可能なコンポーネント数が制限される可能性がある [7]．本論文では，配備するサイズを低減するために，コンテナ技術を活用して，異なるリソースプロバイダネットワークにまたがる階層化ネットワーク上でコンポーネントを管理する手法を提案する．加えて，リソースプロバイダは基本的に彼らのリソースを管理するための専用の API をを持っているため，Dataflow プラットフォームでは，それぞれのプロバイダの API との互換性を保つ必要がある．例えば，Amazon Web Service [3]，Google Cloud Platform [19]，Microsoft Azure [33] はそれぞれ異なる API を用意しており，Dataflow プラットフォーム管理者にとってそれらを統合するタスクは煩雑である．それを避けるため，インテグレーション層として Kubernetes を導入し，共通したインタフェースを提供する．複数のリソースプロバイダネットワークにまたがって構築した Kubernetes 環境上にコンポーネントを展開するアプローチの例を図 6 に示す．このアプローチでは，単一の API によりコンポーネントの展開ができるため，Dataflow オーケストレータの実装コストが抑えられる．リソースプロバイダが提供する API を Kubernetes の API に変換する機能をもつ Rancher [40] や Virtual Kubelet [51] を活用することで，簡単に下位プラットフォームに分散コンピューティングリソースを追加することができる．

2.3.3 節で議論したように，広く分散されたリソース上にコンポーネントを展開するために，

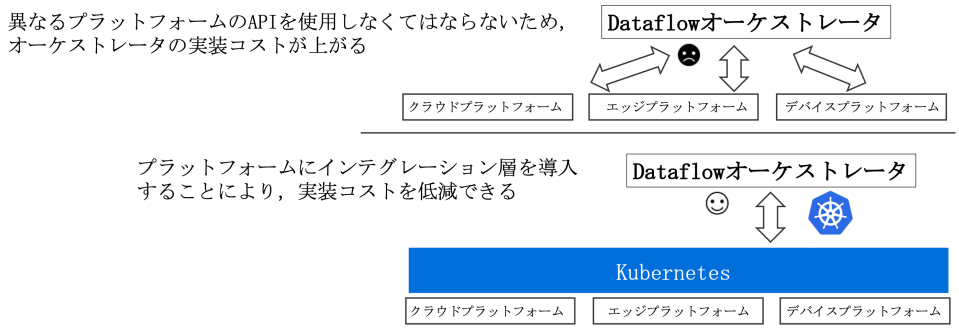


図 6: Kubernetes を用いた階層化ネットワークの集中管理

クラウド/エッジ/デバイスといったネットワーク層や，/JP/West や/JP/East などのネットワーククラスタを考慮する必要がある．しかしながら，もともとの Kubernetes ではそのようなリソースの属性は考慮されない．Kubernetes 上でそれらの属性をラベルとしてコンピューティングノードに追加することを提案する．Kubernetes では，アプリケーションコンポーネントはコンテナイメージとして登録され，そのインスタンスがコンピューティングノード上に展開される．ここで，Kubernetes は ラベルを含む条件を満たさないノードをフィルタしながら，適切なノードを探索してインスタンスを展開する手法を提供する．この機能に基づき，Dataflow オーケストレータは 3 節で説明する手法により決定された場所にコンポーネントを配備することができる．

2.7　分散 Pub/Sub 基盤を用いたコンポーネント間メッセージング

本論文では，コンポーネント間通信には．Teranishi ら [49] の提案手法で活用されているトピックベース Pub/Sub 基盤を用いることを想定している．彼らの提案では，メッセージを配送する際に，同じトピックをサブスクライブするコンポーネントの間でロードバランスする手法を提案している．各コンポーネントはインデックスの範囲を持ち，あるコンポーネントがメッセージ出力する場合，トピックに加えてインデックスを付与してパブリッシュする．例えば，パブリッシャはラウンドロビンなどのスケジューリング手法を選択できる．さらに，2 つのコンポーネントに分割したインデックス範囲を持たせることで，図 7 に示すように計算のオフロードやスケールアウトが実現できる．しかし，そのままではパブリッシャがインデックス値を指定する場合，CPU 負荷状況やメモリ消費量などの宛先コンポーネントの状態を参照できない．

しかし，インデックス値のみでは宛先コンポーネントの負荷状況を考慮した柔軟な配送は難しく，別途，参照可能な仕組みを用意する必要がある．

ここで，MQTT プロトコルを含む標準化された通信プロトコルに互換性をもつ分散 Pub/Sub ブローカ PIQT [39] が提供されている．PIQT は Suzaku [1] オーバレイを用いた柔軟なコン

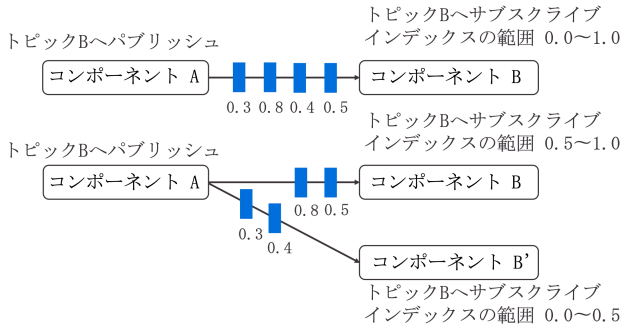


図 7: インデックス割り当てを用いたルーティング手法 [49]

ポーネント間通信を提供する．PIQT のメッセージング機能の詳細は第 4 章で述べるが，本論文で提案する Dataflow プラットフォームは，PIQT などのトピックベース Pub/Sub 基盤を活用することで実現される．

Node-RED [37]，Fluentd [15]，Apache Flink [4] などの既存の Dataflow コンポーネントをサポートするために，Wrapper インスタンスもコンポーネントと一緒に配備される必要がある．このような場合，各 Dataflow コンポーネントは直接サブスクライブ/パブリッシュする代わりに Wrapper インスタンスを通してサブスクライブ/パブリッシュする．ここで，CPU 負荷やメモリ消費量などのコンポーネントの状態を Wrapper インスタンスからオーバレイネットワークに登録することで，リソース状況を考慮した柔軟なメッセージングが実現可能となる．本章ではトピックベース Pub/Sub 基盤を通してコンポーネント同士が通信することを想定し，各 Dataflow アプリケーションコンポーネントを配備するネットワーク層を決定する手法に着目する．リソース状況を考慮したメッセージング手法は次章で述べる．

2.8　結言

本研究では，階層化ネットワークを考慮した Dataflow プラットフォームにおいて，実用的なコンポーネント展開の実現を目指し，クラスタ単位でのコンポーネント配置とコンポーネント間通信を分離する手法を提案する．

これまでに Teranishi ら [49] によって，エッジコンピューティングを考慮した Dataflow プラットフォームが提案されている．彼らの提案では，トピックベース Pub/Sub を拡張する形で，デバイスから生成される Dataflow の処理をエッジに配置されたコンポーネントに動的に割り当てることを可能としている．これにより，エッジ環境への Dataflow 処理のオフロードや，Dataflow の量に応じたスケールイン・スケールアウトを実現している．しかし，クラウドネットワークなどに配備されたコンピューティングリソースの管理方法およびそのリソース上

へのコンポーネント配備方法については提案されていない．クラスタ情報とコンピューティングリソースの情報を結びつけて管理されていない場合，適切なクラスタに含まれるコンピューティングリソース上にコンポーネントを配備できない．一方で，コンピューティングリソースの管理においては，クラウドごとに操作手順や API が異なるため，統合的に扱えるようにする必要がある．しかし，プラットフォーム管理者にとって，煩雑であり，実装コストも高い．

本研究では，インテグレーション層を導入することで，実装コストを抑えながら，クラスタ情報を考慮してコンピューティングリソースを管理する手法として，オープンソースである Kubernetes を活用して管理する手法を提案した．具体的には，クラスタ情報をラベルとして，コンピューティングリソースと紐づけて管理する．また，コンポーネントは，ラベル情報をもとに適切なコンピューティングリソース上に配備される．これにより，3 階層でのコンポーネント配置箇所が決まれば，提案した管理手法により，適切なクラスタのコンピューティングリソース上にコンポーネントを展開できる．

以降，第 3 章，第 4 章では，本章で述べた Dataflow プラットフォームをターゲットとして，それぞれコンポーネント配置手法，コンポーネント間通信手法について述べる．

3　コンポーネント配置先ネットワークレイヤ決定手法

3.1　緒言

Dataflow アプリケーションの運用時における遅延とトラフィック量の課題を解決するため，それらを考慮して適切なネットワークレイヤに Dataflow アプリケーションコンポーネントを配置することが求められる．本研究では，2.3.3 節で述べたように，ターゲットとなるクラスタが限定できることを想定し，そのクラスタに対応するデバイス/エッジ/クラウドなどのネットワークレイヤに基づいて，配置手法を検討する．

本章では，2.4 節で定義した Dataflow グラフに含まれるコンポーネントの配置先として適切なネットワークレイヤを決定する手法を提案する．各ネットワークレイヤへのコンポーネントの配備にかかるコストを定義できれば，コスト最小化問題として適切なコンポーネント配備先ネットワークレイヤを決定できる．以降では，コストが最小となる組み合わせを探索する問題を定式化し，想定するパラメータとともにアプリケーションの例を用いて配置を検証する．

これまで，谷田 [56] によって，上述のような Dataflow アプリケーションを構成するコンポーネント配備先ネットワーク決定手法が提案されている．しかし，ターゲットとなる Dataflow グラフには，分岐ノードや合流ノードが含まれておらず，汎用性に欠けていた．また，提案手法では CPU とメモリのみが対象となっており，他のリソース情報を計算に組み込めないなど，拡張性に欠けていた．さらに，コンポーネントによって消費されるリソース量の取得方法について検討できていなかった．本研究では，分岐ノードや合流ノードを含む Dataflow グラフをターゲットとし，その Dataflow アプリケーション展開手法を検討し，その適用性を示した．また，任意のリソースに対して適用可能なコスト関数を導入し，提案手法の拡張性を改善した．

さらに，コンポーネントが消費するリソース量の計測手法の検討した．

3.2　コンポーネント配置における想定環境

既に述べたように，Dataflow グラフは分岐ノードと合流ノードをもつことができる．最終的なゴールは任意のグラフを扱えることであるが，ほとんどのアプリケーションは分岐ノードや合流ノードを含まないシンプルな直線の Dataflow グラフ (以降では，Dataflow パスと呼ぶ) で表せる．Dataflow グラフが分岐ノードや合流ノードを持つ場合でも，深さ優先探索などを用いていくつかの Dataflow パスに分割することで扱える．Dataflow パス間に依存関係がある場合には，分割した複数のパスについてクリティカルになるパスを検討し，そのパスを中心として，残りのパスの配置箇所を検討することで扱える．また，配置時にデータ送信元が同一であるコンポーネントが既に配置されているかどうかを確認し，配置されている場合には，それを再利用することで，無駄なリソース消費を抑えられる．以降では，配置箇所決定手法について簡潔に説明するため，Dataflow パスについてのみ検討する．

センサのデバイスネットワークへの配置，過去のデータを多く含むようなデータベースのクラウドネットワークへの配置など，コンポーネントの要件に基づき，配置先ネットワークの制限される可能性がある．それらをサポートするため，Dataflow プラットフォームでは，Dataflow グラフの設定で配置先ネットワークを指定することを認める．上記の理由から，本アプローチでは Dataflow グラフ全体の完全に自動的な配備は想定しておらず，アプリケーション開発者は，入力されるグラフの全体もしくは一部のコンポーネント配置先ネットワークレイヤを指定できる．

以降では，コンポーネント配備にかかるコストに影響するデータ転送，通信/処理遅延，リソース消費などのパラメータを抽出したのちに，コスト最小化問題を定式化する．提案手法では上記 3 つであるが，同様な形式で関連するパラメータを抽出することで，式に組み込むことができる．また，3.3 節で説明するパラメータが提供されることを想定しており，一部のコンポーネントにおいて，パラメータが取得できない場合，それに関連するコンポーネントを配置するレイヤは手動で設定する必要がある．今後，パラメータをサポートするコンポーネント数が増えれば，十分なコンポーネント展開の自動化が可能となる．

3.3　コスト最小化問題

本章では，Dataflow アプリケーションを構成するコンポーネントの数を $N \in \mathbb{N}$ とする．また，レイヤ数は $M \in \mathbb{N}$ とする．図 8 の上部に示すように Dataflow グラフは，コンポーネント c_i $(i = 0, \ldots, N)$ の順序を示す．既に述べたように，少なくともデータの送信元となるコンポーネント $c_0 = c_s$ と，データの出力先となるコンポーネント $c_N = c_e$ (図 8 では $N = 5$) の配備先レイヤはアプリケーション開発者によって指定されることを想定する．その他のコンポーネント $(c_1 \cdots c_{N-1})$ の配備先レイヤは以下の節で述べるアルゴリズムにより決定する．

コンポーネント $(c_1 \cdots c_{N-1})$ の配備先レイヤを決定するために，2.3 節で議論した想定をベースとして配備にかかるコストを推定するために適切にパラメータを抽出する必要がある．図 8 は，階層化ネットワーク上に展開する与えられた Dataflow パスに対するコンポーネントの配置のひとつの候補を示す．コンポーネント c_i の配備箇所を単位ベクトル $\boldsymbol{x_i} \in \mathbb{R}^M$ 用いて表すと，決定変数である行列 $\boldsymbol{X}$ は次のように表せる．

$$\boldsymbol{X} = (\boldsymbol{x_1}\ \boldsymbol{x_2}\ \cdots\ \boldsymbol{x_i}\ \cdots\ \boldsymbol{x_{N-1}}) \tag{1}$$

各候補箇所へのコンポーネント配備には，展開コストがかかると考えられる．コスト推定手法が定義できれば，そのコストに基づいて配置箇所が決定できる．各コンポーネントのコストは階層化ネットワークの利点を考慮して，3 つのメトリクス: 転送データ量，通信/処理遅延，リソース消費量から推定する．これらのメトリクスは表 2 に定義したパラメータを用いて推定できる．コンポーネント c_i，c_{i-1} の配備先レイヤは $\boldsymbol{x}_i$，$\boldsymbol{x}_{i-1}$ のように表される．メトリクスの詳細の定義は以下で説明する．

3.3.1　データ転送

階層化ネットワークに関連するデータ転送のコストメトリクスはコンポーネントの入力と出力から推定される．Dataflow パスのはじめのコンポーネント c_1 はセンサデバイスからデータを受け取り，処理結果を次のコンポーネント c_2 へ転送する．c_1 と同じように，c_2 は c_1 からデータを受け取り，その処理結果を c_3 に転送する．このように，Dataflow は最後のコンポーネントがデータを受け取るまで継続して処理される．この場合，コンポーネント c_i への入力

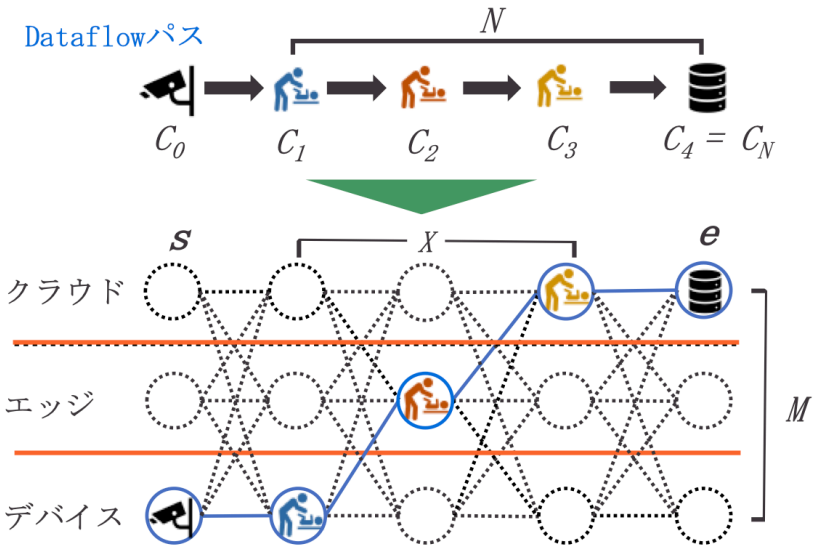


図 8: Dataflow パスの展開の全ての組み合わせとそれに対する配置例

表 2: 展開コストを推定するためのパラメータ

(a) Dataflow プラットフォームパラメータ

Name	Description
M	ネットワークレイヤ数
$\boldsymbol{A}_j$	各ネットワークレイヤにおけるすべての利用可能なリソース
$\boldsymbol{D}$	各ネットワークレイヤ間通信遅延

(b) Dataflow アプリケーションパラメータ

Name	Description
N	コンポーネント数
c_i	i 番目のコンポーネント ($i = 1, \ldots, N-1$)
$\boldsymbol{s}$	センサデバイスの配置先ネットワークレイヤ
$\boldsymbol{e}$	エンドコンポーネントの配置先ネットワークレイヤ
b_1	c_i の入力データ量を b_i ($i = 1, \ldots, N$) とした際の c_0 から c_1 への入力データ量
$\boldsymbol{B}_{\text{max}}$	各ネットワークレイヤ間の通信帯域の上限
d_{max}	Dataflow アプリケーション全体での許容可能な遅延の合計

(c) コンポーネント c_i に関連するパラメータ

Name	Description
r_i	単位時間あたりの $c_i (i = 1, \ldots, N)$ の入力データに対する出力データの比率
p_{ij}	入力データ量に対する推測される単位時間あたりの $c_i (i = 0, \ldots, N)$ のリソース消費
d_i	c_i ($i = 1, \ldots, N$) の処理遅延

データ量はコンポーネント c_{i-1} からの出力データ量と等しくなる．コンポーネント c_i への入力データ量を b_i とすると，Dataflow パスににおけるひとつ前のコンポーネント c_{i-1} からの出力データ量も同じ値 b_i となる．ここで，c_{i-1} からの出力データ量が c_{i-1} への入力データ量 b_{i-1} とコンポーネント c_{i-1} の入出力比 r_{i-1} から算出できるとすると，b_i は次のように表せる．

$$b_i \ = \ b_{i-1} r_{i-1} \quad (i = 1, \ldots, N) \tag{2}$$

また，ネットワークレイヤをまたぐ通信にかかる費用を考慮して，閾値が設定できれば，その閾値をコンポーネント c_i への入力データ量 b_i に対するコストメトリックの指標にできる．以下に示す行列 $\boldsymbol{B}_{\text{max}}$ は内積 $\boldsymbol{x}_{i-1}^{\text{T}} \cdot \boldsymbol{B}_{\text{max}} \cdot \boldsymbol{x}_i$ から計算される $\boldsymbol{x_{i-1}}$ と $\boldsymbol{x_i}$ の間の通信帯域上限を示す．

$$\boldsymbol{B}_{\text{max}} = \begin{pmatrix} \infty \text{ (Dev} \rightarrow \text{Dev)} & \text{Dev} \rightarrow \text{Edge} & \text{Dev} \rightarrow \text{Cloud} \\ \text{Edge} \rightarrow \text{Dev} & \infty \text{ (Edge} \rightarrow \text{Edge)} & \text{Edge} \rightarrow \text{Cloud} \\ \text{Cloud} \rightarrow \text{Dev} & \text{Cloud} \rightarrow \text{Edge} & \infty \text{ (Cloud} \rightarrow \text{Cloud)} \end{pmatrix} \tag{3}$$

他の通信に比べて必要な予算が低くなることを表現するため，同一ネットワークの通信帯域は無限大とする．$\boldsymbol{B}_{\text{max}}$ は，物理ネットワークの上限を示す場合もあるが，課金されるネットワークでの許容できる通信量を示す場合もある．これらはプラットフォームプロバイダかアプリケーション開発者によって与えられることを想定している．ここで，データ転送にかかる予算を計算するための以下に示すようなコスト関数 $bw()$ を導入する．この関数は，コンポーネント c_i がデータ受け取り時に消費する通信帯域 b_i と指定された $\boldsymbol{x_i}$，$\boldsymbol{x_{i-1}}$ 間で利用可能な通信帯域との比を返す．

$$bw(i, \boldsymbol{x}_i, \boldsymbol{x}_{i-1}) = \frac{b_i}{\boldsymbol{x}_{i-1}^{\text{T}} \cdot \boldsymbol{B}_{\text{max}} \cdot \boldsymbol{x}_i} \tag{4}$$

3.3.2　通信/処理遅延

コンポーネント間通信とコンポーネントでの処理遅延はリアルタイムアプリケーションにとって重要な要素となる．Dataflow パス内のコンポーネント c_i，c_{i-1} 間の通信遅延は，コンポーネントの種類には依存しないが，コンポーネントが配備されるネットワークレイヤに依存する．

ネットワークレイヤ間の通信遅延は，$\boldsymbol{B}_{\text{max}}$ と同様のフォーマットの行列 $\boldsymbol{D}$ で表せるとする．$\boldsymbol{D}$ は内積 $\boldsymbol{x}_{i-1}^{\text{T}} \cdot \boldsymbol{D} \cdot \boldsymbol{x}_i$ の計算から得られた $\boldsymbol{x}_{i-1}$，$\boldsymbol{x}_i$ 間の通信遅延を示す．ここで，コンポーネントの種類に依存する処理遅延 d_i が与えられるとすると，全体で許容可能な遅延は d_{max} として定義できる．最後に，コスト関数 *delay()* の導入により，通信/処理遅延によるコストは次のように表される．

$$delay(i, \boldsymbol{x}_i, \boldsymbol{x}_{i-1}) = \frac{\boldsymbol{x}_{i-1}^{\text{T}} \cdot \boldsymbol{D} \cdot \boldsymbol{x}_i + d_i}{d_{\text{max}}} \tag{5}$$

3.3.3　リソース消費

階層化ネットワークの利点を通信予算に反映するには，消費されるリソースも考慮しなければならない．実際のデプロイ段階では，コンポーネントは配備されるネットワークレイヤのリソースを消費する．また，その消費量はコンポーネントでの処理内容やコンポーネントへの入力データサイズに依存する．ここで，各ネットワークレイヤのリソースとしてプロセッサとメモリに着目する．ウィンドウサイズが大きいコンポーネントにとってはストレージリソースも重要な要素となる．例えば，特定期間に Dataflow の統計情報を計算するコンポーネントは，その期間内の Dataflow の蓄積に十分なストレージ容量を要求する．しかし，本論文では中間

ノードがストレージの所有を要求しないことを想定する．ストレージリソースの検討は今後の課題である．さらに，電源供給がないようなデバイスにとっては電源消費も重大な要素となる．バスサービスにおけるユースケースでは，バスのバッテリがセンサやアクチュエータデバイスに使用可能である．電源消費を抑えることは望ましいが，センサネットワークでのケースに比べてクリティカルではない．本論文では，CPU，メモリなどのコンピューティングリソースとネットワークリソースの消費量を抑えることに着目する．

3.3.4 コスト関数

コンポーネントによって消費されるリソースは一定ではないが，入力データサイズに依存するため，求められるリソース量の平均は予測できる．本プラットフォームでは，CPU 使用率やメモリ消費量などのコンピューティングリソースの使用量は計測環境でいくつかの入力データサイズに対して計測されることを想定する．その結果に基づいて，求められるリソースを推定する関数 *predictRequiredResources()* を導出する．デプロイ段階では，この関数を用いて求められるリソースを予測する．

一方で，リソース消費の程度はターゲットとなるレイヤ $\boldsymbol{x}_i$ に依存する．例えば．CPU，GPU などのプロセッサの種類や，そのクロック数はデプロイされたレイヤに依存する．クラウドネットワークでの 1GB メモリの消費はそこまで全体のリソース量に影響しないが，デバイスネットワークでは，リソースはそれほどないため，マイナス効果が大きくなる．それゆえ，配備先ネットワークレイヤの選択時には，これらはそれぞれ別で計算することが望ましい．提案手法では，コンポーネントに必要とされるリソースを各ネットワークレイヤの全体リソースに対する比率として表す．これにより，たとえ消費されるリソース量が同じであっても，ネットワークレイヤごとにその影響を適切に反映できる．CPU，メモリ，ストレージなどのリソースの種類数を $j\ (1 \leq j \leq J \in \mathbb{N})$ とすると，関数 *predictRequiredResources*$_j$*()* により，リソースの種類 j に対するコンポーネント c_i が必要とするリソース p_{ij} が算出できる．この関数はコンポーネントの種類，入力データサイズ，配備されるネットワークレイヤの情報から，コンポーネント c_i を実行する際の j のリソース消費量を予測する．

$$p_{ij} = \textit{predictRequiredResources}_j(c_i, b_i, \boldsymbol{x}_i) \tag{6}$$

ここで，j の種類の全ての利用可能なリソースを行列 $\boldsymbol{A}_j$ とする．

$$\boldsymbol{A}_j = \begin{pmatrix} \boldsymbol{a}_{\text{j\_device}} & \boldsymbol{a}_{\text{j\_edge}} & \boldsymbol{a}_{\text{j\_cloud}} \end{pmatrix} \tag{7}$$

$\boldsymbol{a}_{\text{j\_device}}, \boldsymbol{a}_{\text{j\_edge}}, \boldsymbol{a}_{\text{j\_cloud}}$ は，各ネットワークレイヤにおける j の種類の全ての利用可能なリソースを示す．Dataflow オーケストレータは，任意のリソースの種類に対して配備先ネットワークレイヤを計算できる．

このプラットフォームでは，各ネットワークレイヤにおいて，“4 コア/RAM 32GB” など利用可能なリソースのパターンが Dataflow プラットフォームプロバイダによって提供されるこ

とを想定する．そして，各レイヤにおいて，アプリケーション開発者が予約可能な合計リソースは，最大利用可能なリソース $\boldsymbol{A}_j$ として与えられる．デプロイされるネットワークレイヤにおいて，任意のリソースの種類に対して消費比率を計算するために，コスト関数 $resource_j()$ を導入する．

$$resource_j(i, \boldsymbol{x}_i) = \frac{p_{ij}}{\boldsymbol{A}_j \cdot \boldsymbol{x}_i} \tag{8}$$

3.3.5　定式化

提案手法では，$c_s = c_0$ から $c_e = c_N$ の各コンポーネントの配置コストを計算することで，Dataflow パスの各配置候補に対するコストを見積もる．その後，総当たりにより全ての候補から，合計のコストが最小であるものを探索する．ただし，条件に満たないパス，例えば，最終コンポーネントに辿りつくまでに利用可能なリソースの上限を超えたパスは計算中に候補リストから削除される．最小化される目的関数および制約を以下に示す．

$$min \sum_{i=1}^{N} f(i, \boldsymbol{x}_i, \boldsymbol{x}_{i-1}) \tag{9}$$

$$f(i, \boldsymbol{x}_i, \boldsymbol{x}_{i-1}) = \alpha\, bw(i, \boldsymbol{x}_i, \boldsymbol{x}_{i-1}) + \beta\, delay(i, \boldsymbol{x}_i, \boldsymbol{x}_{i-1}) + \sum_{j=1}^{J} \gamma_j\, resource_j(i, \boldsymbol{x}_i) \tag{10}$$

Subject to

$$bw(i, \boldsymbol{x}_i, \boldsymbol{x}_{i-1}) \leq 1,\ \forall i \in 1, \ldots, N \tag{11}$$

$$T_{\text{delay}}(i, \boldsymbol{X}) = \sum_{k=1}^{i} delay(k, \boldsymbol{x}_k, \boldsymbol{x}_{k-1}) \leq 1,\ \forall i \in 1, \ldots, N \tag{12}$$

$$T_{\text{resource\_j}}(i, \boldsymbol{X}) = \sum_{k=1}^{i} resource_j(k, \boldsymbol{x}_k) \boldsymbol{x}_k^T \cdot \boldsymbol{x}_i \leq 1, \forall i \in 1, \ldots, N,\ \forall j \in 1, \ldots J \tag{13}$$

式 (9) は最小化問題の目的関数を示す．式 (10) では，複数のメトリクスを同時に考慮するために導入した重み α，β，γ_j $(0 \leq \alpha,\ \beta,\ \gamma_j \leq 1)$ を用いてコンポーネント c_i の配置コストを計算する．式 (11)-(13) は各メトリックの制約を示す．

与えられた Dataflow パスに対して最小の配置コストとなるコンポーネント配置を探索するアルゴリズムを Algorithm1 と Algorithm2 に示す．提案アルゴリズムは，再帰を用いたシンプルな Brute Forth Depth First Search(総当たり深さ優先探索) であり，条件を満たさないケースはそれ以上計算しないように枝刈りすることで，ある程度，計算量を抑えている．アルゴリズムへの入力パラメータは 3.3 節の表 2 のとおりである．これらのパラメータは表 2(a) プラッ

トフォームプロバイダによって与えられる Dataflow プラットフォームパラメータ，表 2(b) アプリケーション開発者によって与えられる Dataflow アプリケーションパラメータ，表 2(c) コンポーネント開発者によって与えられる Dataflow アプリケーションコンポーネントパラメータの 3 つのタイプに分類できる．Dataflow アプリケーションパラメータはアプリケーション開発者によって設定可能だが，その他のパラメータは，それぞれプラットフォームプロバイダ，コンポーネント開発者から，コンフィグファイルなどのような形で配布される必要がある．*findMinimalCostDeployment* 関数はグローバル変数 *minCost* と *bestX* を用いて，コンポーネント配置 $\boldsymbol{X}$ の最小コストを出力する．*calcDeploymentCosts* 関数は最後のコンポーネントに辿り着くまで，各コンポーネントの配置コストを計算するために再帰的に呼び出される．次に，*calcDeploymentCosts* 関数の処理内容を説明する．Algorithm2 の *createNewPlacements* によってコンポーネントの新しい配置候補が生成される．生成されたそれぞれの配置候補は，Algorithm2 の *satisfied* 関数によりコンポーネント c_i の配置箇所 $\boldsymbol{x}$ が制約を満たすかどうか検証される．満たす場合には Algorithm2 の *merge* 関数により，これまでのコンポーネント配置にマージされ，配置コストの計算が進められる．満たさない場合，配置候補 $\boldsymbol{x}$ は削除される．それと同時に，その配置候補に依存する Dataflow パスの配置候補は削除される．

Algorithm 1 最小コストとなる Dataflow パスの探索 (1)

```
 1: function FINDMINIMALCOSTDEPLOYMENT(N, s, e, B_max, d_max)
 2:     X ← M 行 (N + 1) 列の行列を生成する
 3:     X ← x_0 = s, x_N = e, 他は 0 で初期化する
 4:     global minCost ← ∞ (十分に大きい値で初期化する)
 5:     global bestX ← NULL
 6:     CALCDEPLOYMENTCOST(X, 1, 0)
 7:     return bestX
 8: end function
 9: function CALCDEPLOYMENTCOSTS(X, i, cost)
10:     if i is not N then
11:         placements ← CREATENEWPLACEMENTS()
12:         for x in placements do
13:             if SATISFIED(i, MERGE(X, i, x)) is false then continue end if
14:             CALCDEPLOYMENTCOSTS(MERGE(X, i, x), i + 1, cost + f(i, x, x_{i-1}))
15:         end for
16:     else
17:         if SATISFIED(i, X) is false then return end if
18:         cost ← cost + f(i, x_N, x_{i-1})
19:         if minCost > cost then
20:             minCost ← cost
21:             bestX ← X
22:         end if
23:         return
24:     end if
25: end function
```

Algorithm 2 最小コストとなる Dataflow パスの探索 (2)

function CREATENEWPLACEMENTS
    return $\mathbb{R}^M$ の単位ベクトル
end function
function MERGE($\boldsymbol{X}, i, \boldsymbol{x}$)
    return $\boldsymbol{X}$ の i 列にベクトル $\boldsymbol{x}$ をマージした行列
end function
▷ SATISFIED() のコスト計算はさらなる最適化の可能性があるが，ここでは可読性のため上述の式を用いる
function SATISFIED($i, \boldsymbol{X}$)
    if not $bw(i, \boldsymbol{x}_i, \boldsymbol{x}_{i-1}) \leq 1$ **then return** false **end if**
    if not $T_{\text{delay}}(i, \boldsymbol{X}) \leq 1$ **then return** false **end if**
    for j in J that the number of resource kinds **do**
        if not $T_{\text{resource\_}j}(i, \boldsymbol{X}) \leq 1$ **then return** false **end if**
    end for
    return true
end function

提案したアルゴリズムは，与えられたパラメータと定義されたコストに基づいた最適な Dataflow パスの配置を提供する．しかし，アルゴリズムが実用性が明らかになっていない．次節では，提案したアルゴリズムを 2 つの典型的なユースケースに適用し，その適用可能性を検証する．

3.4　評価

本論文では，バスサービスのための Dataflow アプリケーションをターゲットとして，プラットフォームへの要件を検討してきた．本節では，深さ優先探索によって Dataflow グラフから抽出した 4 つの Dataflow パスに着目する．抽出したそれぞれの Dataflow パスに対して，提案手法を用いた適切なコンポーネント配置の求め方をデモンストレーションする．提案手法に必要となるパラメータを抽出するために，サンプルアプリケーションの実装も行った．以下では，はじめに実装したアプリケーションおよびパラメータの計測結果について説明する．

図 9 に示す 4 つの Dataflow パスを構成するコンポーネントのうち，4 つのコンポーネントを実装した．図 9 (a), (b) の c_1 はバス停を検出するコンポーネント，図 9 (c), (d) の c_1 はフレームレートを変換するコンポーネント，c_2 は人の骨組みを検出するコンポーネントである．c_3 はバスの乗降者をカウントするコンポーネントおよび危険運転を検知するコンポーネントである．ここで，Nakashima ら [36] の提案では，ランダムフォレスト回帰を用いて乗降者数をカウントするが，事前に生成された学習モデルを用いた検出処理はそれほど計算リソースを必要としないため，本論文では，シンプルな人数カウントコンポーネントを c_3 として使う．また，図 10 に示すように各コンポーネントは MQTT ブローカである Moquette [34] を通して通信することを想定する．

バス停検出では，バスのセンサから得られる GPS と速度のデータからバスがバス停の近くに位置するかを判定する．判定結果は true/false で “bus_stop” トピックとして MQTT ブローカにパブリッシュする．MQTT ブローカとの通信には，MQTT クライアントライブラリである Mosquitto [10] を用いる．GPS データはターゲットとなるバスの経路上にあるバス停の位置が保存されている Elasticsearch データベース [12] の検索機能を用いてバス停と一致させる．フレームレート変換では，video4linux を用いてカメラから得られる MPEG を JPEG に変換する．その後，JPEG データは “skeleton” トピックに紐づけて Mosquitto を用いて MQTT ブローカにパブリッシュされる．フレームレート変換コンポーネントでは，カメラからの出力レートである 30fps から 1fps に変換する．人の骨組みを検出するコンポーネントでは，MQTT クライアントライブラリ Paho [11] を用いて “skeleton” トピックをサブスクライブし，MQTT ブローカからイメージデータを受け取る．その後，OpenPose [38] を用いて受け取ったイメージから骨組みを抽出する．また，抽出した骨組みデータは “count” トピックにパブリッシュする．ここで，このコンポーネントが (a, c) の c_2 として使われる場合，つまり乗降者数をカウントするアプリケーションとして使われる場合，“bus_stop” トピックをサブスクライブし，受け取ったデータが true であった時にのみ，その結果をパブリッシュする．上記のコンポーネントと同じように，人数カウントコンポーネントは “count” トピックをサブスクライブし，カウント結果を “DB” トピックにパブリッシュする．

評価では，CPU 使用率およびメモリ使用量はリソースコストとして考慮される．つまり，リソースの種類 j は 2 である．CPU 使用率およびメモリ使用量はアプリケーション実行中に top コマンドを用いて計測した．また，各コンポーネント間のトラフィック量は Wireshark を用いて記録した．計測に使用した機器の仕様を表 3 に示す．各コンポーネントを用いて計測した

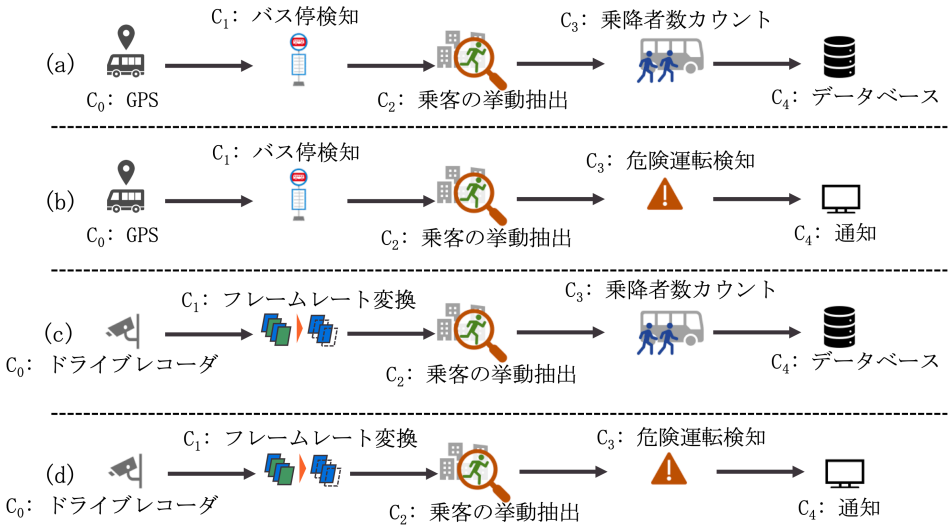


図 9: 4 つの Dataflow パス

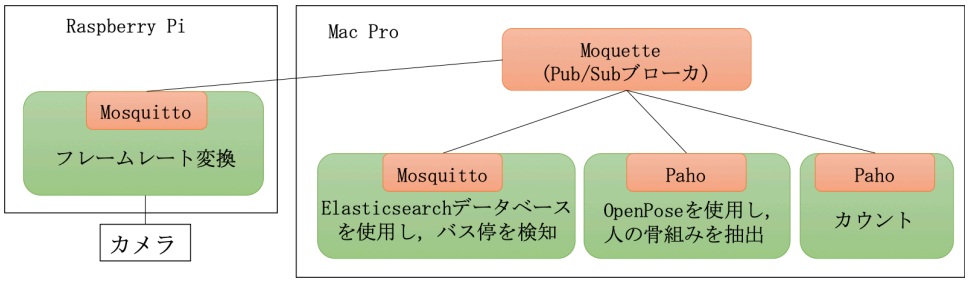


図 10: 計測環境

表 3: 計測に使用した機器の仕様

Name	Specification
カメラ (BSW20KM11BK)	MPEG, 30fps
Raspberry Pi	3 model B v1.2
Mac Pro	8-Core Intel Xeon E5, RAM 32GB

表 4: (a) と (c) のコンポーネントパラメータ ($N = 4,\ M = 3$)

(a)	c_1: バス停検知	c_2: 骨組み抽出	c_3: カウント	c_4: データベース
r_i	0.01	-	-	-
p_{i1}	0.69	0	0	0
p_{i2} [MB]	1340	0	0	0

(c)	c_1: フレームレート変換	c_2: 骨組み抽出	c_3: カウント	c_4: データベース
r_i	0.03	0.01	0.18	-
p_{i1}	0.1	0.4	0.1	0.5
p_{i2} [MB]	0.5	214	21	200

CPU 使用率，メモリ使用量，計算した入出比を表 4 に示す．簡単化のため，リソース消費量は入力データ量や割り当てられた CPU コアによらず一定であると想定する．そのため，本論文での評価では，処理遅延 d_i はネットワークレイヤの違いによる影響は受けず，配置コストの計算から省略される．処理遅延を含む検証は今後の課題である．以降では，抽出したパラメータを用いて，提案するコンポーネント配置手法を各ユースケースに適用する．

3.4.1 ユースケース 1: 乗降者数カウントアプリケーション

はじめに図 9 (a), (c) に示す “乗降者数カウントアプリケーション” について説明する．プラットフォームパラメータおよびアプリケーションパラメータは式 14 に示す．センサデバイスと出力先コンポーネントの配備先ネットワークレイヤはそれぞれデバイスネットワーク，クラウドネットワークであることを想定する．また，各ネットワークレイヤのリソースおよび通信帯域は，“デバイスネットワーク $<$ エッジネットワーク $<$ クラウドネットワーク” の条件を満たすように設定する．通信帯域の上限には LTE のようなモバイル通信を参考にした．本論文では 5G の特性が使える環境を想定しているが，本評価では通信帯域と予算を検討するために LTE を参考とする．ほとんどの格安 SIM では定額制の料金プランが提供されており，よく選択されるもののひとつとして月 3GB のプランがある．$\boldsymbol{B}_{\max}$ はその値を用いて計算した．また，はじめのコンポーネント c_i への入力データ量はケース (a)，ケース (c)，それぞれ 2.07 Mbps，0.01 Mbps である．

$$\boldsymbol{A}_1 = \begin{pmatrix} 1 & 4 & 20 \end{pmatrix},\ \boldsymbol{A}_2 = \begin{pmatrix} 500 & 4000 & 10000 \end{pmatrix}\ [\mathrm{MB}],$$
$$\boldsymbol{s} = \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix},\ \boldsymbol{e} = \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix},\ \boldsymbol{B}_{\max} = \begin{pmatrix} \infty & 0.01 & 0.01 \\ 0.01 & \infty & 1000 \\ 0.01 & 1000 & \infty \end{pmatrix}\ [\mathrm{Mbps}] \tag{14}$$

このユースケースでは，たとえ出力データがリアルタイムに使われる場合であっても，時間的制約はそれほど厳しいものではない．例えば，バスが次のバス停に到着するまでに処理が終われば良い．それゆえ，このデモンストレーションでは遅延条件は $\beta = 0$ と設定して省略する．

Dataflow パス配置コストは Algorithm 1，Algorithm 2 を用いて計算されるが，計算の過程をを確認するために表 5 のサンプルの計算結果では，中間コンポーネント c_i における計算結果も示す．

重みを $\alpha = 1$，$\beta = 0$，$\gamma_1 = 0.5$，$\gamma_2 = 0.5$ とした場合の結果を表 5 (a)，(c) に示す．表では，c_i の列に表示されるコンポーネントまでの Dataflow パスの配置コストを示す．“配置レイヤ” の列に書かれた文字は，コンポーネント c_1，c_2，c_3，c_4 が配備されるネットワークレイヤを示す．D はデバイスネットワーク，E はエッジネットワーク，C はクラウドネットワークであることを示す．また，“コスト” 列は配置コストの推定値を示す．提案手法では，3 節で定義した条件を満たさない組み合わせは除外されるが，表 5 では $\times$ マークを用いて表示する．

Dataflow パス (a) の c_1 から出力されるデータは，c_2 でのみ活用される．またその処理は Dataflow パス (c) で考慮される．それゆえ，表 5 では，c_1 の配置コストのみ計算される．結果から，パス (a) の c_1 はメモリが足りないため，デバイスネットワークには配置できないが，エッジネットワークか，クラウドネットワークには配置できる．次に，表 5 (c) について説明する．c_1 から転送されるデータ量が大きいため，人の骨組みを検出する c_2 はエッジネットワー

表 5: ユースケース 1 における計算結果

(a) パス (a) の結果 1
($\alpha = 1, \beta = 0, \gamma_1 = 0.5, \gamma_2 = 0.5$)

c_i	配置レイヤ	コスト
c_1	D	1.35 ×
	E	1.09
	C	1.02

(b) パス (a) の結果 2
($\alpha = 1, \beta = 0, \gamma_1 = 0.7, \gamma_2 = 0.5$)

c_i	配置レイヤ	コスト
c_1	D	1.48 ×
	E	1.12
	C	1.02

(c) パス (c) の結果 1
($\alpha = 1, \beta = 0, \gamma_1 = 0.5, \gamma_2 = 0.5$)

c_i	配置レイヤ	コスト
c_1	D	0.05
c_2	D-D	0.25
	D-E	6.31 ×
	D-C	6.27 ×
c_3	D-D-D	0.3
	D-D-E	0.33
	D-D-C	0.32
c_4	D-D-D-C	0.32
	D-D-E-C	0.34
	D-D-C-C	0.33

(d) パス (c) の結果 2
($\alpha = 1, \beta = 0, \gamma_1 = 0.7, \gamma_2 = 0.5$)

c_i	配置レイヤ	コスト
c_1	D	0.07
c_2	D-D	0.35
	D-E	6.35 ×
	D-C	6.29 ×
c_3	D-D-D	0.42
	D-D-E	0.43
	D-D-C	0.42
c_4	D-D-D-C	0.45
	D-D-E-C	0.45
	D-D-C-C	0.43

クやクラウドネットワークには配置できない．c_2 が実行されることで，それ以降のデータ転送量は十分に低減されるため，c_3 はどのネットワークレイヤにも配置できる．しかし，デバイスネットワークの外に配置すると，データ転送量の増加に伴いその配置コストも増大する．結果として D-D-D-C の配置コストが最小コストとなる．一方で，c_1 から c_3 がすべてデバイスネットワークに配置される場合，CPU 使用率が 80% となり，ほかのコンポーネントを同じネットワークに配置することが難しくなる．CPU 使用率のコストに対する優先度を上げることで，配置結果を調整する．パラメータを $\alpha = 1, \beta = 0, \gamma_1 = 0.7, \gamma_2 = 0.5$ とした場合の結果を表 5 (b) に示す．計算結果は D-D-D-C より D-D-C-C が良いことを示している．これらの結果から，リソース消費によるコストを重視させ，優先度を調整することでデバイスネットワークに配備されるコンポーネントの数を減らせることを確認した．

3.4.2 ユースケース 2: 危険運転検知アプリケーション

2 つ目のアプリケーションは図 9(b)，(d) に示す “危険運転検知” である．このアプリケーションもまた 4 つのコンポーネントから構成されるが，c_3 および c_4 がユースケース 1 と異なり，危険運転を検知し，運転手に通知する．このユースケースでは，出力先コンポーネントを

表 6: ユースケース 2 における計算結果

(a) パス (b) の結果
$(\alpha = 1, \beta = 1, \gamma_1 = 1, \gamma_2 = 1)$

c_i	配置レイヤ	コスト
c_1	D	2.69 ×
	E	1.22
	C	1.53

(b) パス (d) の結果
$(\alpha = 1, \beta = 1, \gamma_1 = 1, \gamma_2 = 1)$

c_i	配置レイヤ	コスト
c_1	D	0.1
c_2	D-D	0.5
c_3	D-D-D	0.6 ×
	D-D-E	0.64
	D-D-C	1.07
c_4	D-D-D-D	1.1 ×
	D-D-E-D	1.2
	D-D-C-D	2.08

デバイスネットワークに配置されること，短い時間で応答することが求められる．この要件を反映するために，新しく各ネットワークレイヤ間で発生する通信遅延を定義する．また，最大許容遅延 d_{max} を 20ms とする．ユースケース 2 に合わせて変更および追加したパラメータを式 (15) に示す．

$$d_{\text{max}} = 20\ [\text{ms}], \boldsymbol{e} = \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix}, \boldsymbol{D} = \begin{pmatrix} 0 & 1 & 10 \\ 1 & 0 & 7 \\ 10 & 7 & 0 \end{pmatrix} [\text{ms}] \tag{15}$$

重みを $\alpha = 1, \beta = 1, \gamma_1 = 1, \gamma_2 = 1$ とした場合の計算結果を表 6 (a)(b) に示す．表 6 (a) の結果はユースケース 1 の結果に似ているが，遅延を考慮したユースケース 2 の結果の方がエッジネットワークにコンポーネントを配置した際のコストが低くなっている．表 6 (b) では，ほとんどの配置候補が通信帯域の上限を超えたため除外されており，D-D-E-D と D-D-C-D の 2 つのみが残っている．D-D-D-D は最小コストとなる配置パターンとして計算されているが，使用する計算リソースの上限を超えたため除外されている．多くの場合，全てのコンポーネントをデバイスネットワークに配置することは難しく，遅延とリソースの両方を考慮して適切に配置コストを計算する必要がある．D-D-E-D が適切な配置であることを提案するが，D-D-D-D と D-D-E-D の配置コストがかなり近い値になっていることから，デバイスネットワークのリソース不足がパラメータで正しく考慮されていないと考えられる．これはプロセッサの種類を省略していることや，入力データ量をまるめて反映していることが原因だと考えられる．

上記の結果から，両方のユースケースにおいて重みを適切に設定することでそれぞれのコストの影響を調整できることを明らかにした．一方で，各ネットワークレイヤにおけるコンピューティング環境の違いは十分には反映されなかった．それゆえ，各ネットワークレイヤにある CPU や GPU といったハードウェアの違いは 3.3.2 節で述べたコンポーネントでの処理時

間 d_i と一緒に考慮しなければならない．また，必要に応じて消費電力も考慮することが求められる．しかし，ハードウェアの違いによる処理性能の変化を計測することは難しく，今後それらの反映手法の検討が求められる．

3.5 関連研究

Blackstock ら [8] は Node-RED [37] を活用して Dataflow プラットフォームの実現を検討している．Node-RED は GUI ベースで Dataflow アプリケーションを定義できるが，物理マシンなど複数のノードを跨いでコンポーネントを操作することができない．この問題を解決するために，Blackstock ら [8] は処理ノードを指定するために，MQTT ブローカを用いて通信機能を拡張している．ユーザに定義された Dataflow グラフをいくつかのサブグラフに分割できるが，コンポーネントをデプロイするノードの選択方法については議論されていない．

Azure IoT Edge [31] や Google Cloud IoT Core [18] では，クラウドとエッジをまとめて管理し，Dataflow グラフを展開できるが，ベンダロックインが回避できないため，柔軟なサービス構成が実現できない．

Zhang ら [54] や Bahreini ら [6] は MEC 環境におけるコンポーネント配置手法を提案している．Wang ら [52] は本論文に近い問題の解決に取り組んでいる．具体的には，CPU 負荷やネットワーク帯域などのアプリケーション要件を表現できるアプリケーショングラフを物理ノードやリンクのリソース状態を表現できる物理グラフに展開する問題を検討している．彼らの手法では物理ノードやリンクの最大のリソース使用率を最小化することで，ロードバランスを実現する．しかし，計算の複雑性を低減するために遅延を考慮していない．これはリアルタイムアプリケーションにとっては致命的である．我々の提案手法ではモデルを簡単化し，クラスタ内における些細な遅延を省略することで，通信遅延を考慮した配置を実現している．一方で，現状のアルゴリズムでは任意の分岐ノードや合流ノードを含む Dataflow グラフをカバーできていないため，今後拡張方法を検討する必要がある．Bahreini ら [6] では，ユーザの移動性を考慮し，ユーザとエッジサーバの距離に応じて通信コストを最小化するコンポーネント配置手法を提案している．しかし，利用可能なリソース量の上限を考慮できておらず，実際には配置できないケースが算出される可能性がある．我々の提案手法では事前の知識に依存せず，現実的な前提条件を想定するために，コンポーネント配置とコンポーネント間通信機能を分離している．モバイルデバイスを考慮して観測した状態に基づいて動的にコンポーネント間でルーティングすることを可能としている．あらかじめ冗長コンポーネントを用意しておき，ユーザの移動に合わせて，動的にコンポーネントを繋ぎ変えることで通信コストを抑えられる．

Harris ら [22] は MEC 環境において，全てのクライアントの遅延が最小となるように，ネットワーク機能を配置する手法を提案している．この手法では，最小化のために，全てのクライアントと基地局の最適な組み合わせを探索する．POP におけるデータ集約機能がアプリケーションから要求される場合，我々の提案においても，これと似たような組み合わせを検討する

必要がある．しかし，本論文の提案では，下位レイヤの 5G サービスによって基地局が選択されることを想定しており，アプリケーションレイヤでの制御に着目している．

Ren ら [41] は MEC 環境における全てのデバイスの遅延を最小化するための通信の接続とリソース割り当てに取り組んでいる．エッジとクラウドの連携は我々の提案と似ているが，ターゲットが異なる．我々は遅延と予算を考慮した個々のサービスのアプリケーションの展開をターゲットとしているが，彼らの提案では全体の遅延の最適化がターゲットとなっている．Ren ら [41] の提案で述べられているように，全てのデバイスの遅延を合計して計算しているため，各サービスの遅延要件を満たせない可能性がある．我々の提案ではアプリケーション要件を満たすことが求められるため，そのままでは適用できない．Dataflow プラットフォームプロバイダの観点からは，全体の遅延を抑えることは必要であるが，この手法は有用である．今後，目安として導入し，アプリケーションの展開時のパラメータ設定に取り入れるなど，提案手法の拡張が考えられる．

Anthos [17] は，Kubernetes [50]，Istio[28]，Knative[20] などを活用したサービスメッシュを実現するプラットフォームである．サービスメッシュは，Dataflow を処理する技術のひとつであり，Anthos では主に，Kubernetes が提供するコンポーネント管理機能，コンポーネントを接続するための Istio が提供するサービスディスカバリ機能，Knative による複数クラウド環境にまたがるプラットフォームの構築機能を用いてサービスメッシュを実現している．Anthos は Google によって提供されるため，クラウド機能のロックインが懸念される．一方で，Knative の機能により，オンプレミス環境などのクラスタを組み合わせることができるため，広域なネットワークの構築は可能である．Kubernetes では，ネットワーク階層を考慮したコンポーネントの配置を直接はサポートしていないが，Kubernetes で管理する物理マシンなどのノードにラベルを付与する機能を保持している．クラスタ ID などのネットワークレイヤの情報をラベルとしてノードに付与することで，識別できる．提供されるコンポーネント配置機能に，我々が提案するような配置レイヤ決定などの機能が拡張実装されることで，階層化ネットワークを考慮したコンポーネント配置が実現できる可能性がある．

3.6　クリティカルパス抽出における課題

本章では，ターゲットとする Dataflow グラフが分岐ノードや合流ノードを含む複雑なトポロジを持つとしても，遅延や運用コストが支配的となるクリティカルな Dataflow パスを抽出できることを想定している．また，この想定に基づいて，抽出したクリティカルパスから順にリソースを割り当てるシンプルなアルゴリズムを提案している．一方で，クリティカルパスが一意に決まらない場合，リソースを割り当てた結果，遅延，予算を考慮した最適な割り当てにならない可能性がある．我々の想定するユースケースでは，フレームレート変換コンポーネント，乗客の挙動抽出コンポーネントを含むパスが支配的となっており，それ以外の分割されたパスは，支配的なパスの一部を再利用する形で配置コストが計算されるため，後で割り当てら

れたパスによって，悪い結果が招かれることはない．しかし，今後，複雑な Dataflow の中には，クリティカルパスが一意に決まらないケースが含まれる可能性がある．これらのケースに対応するためには，クリティカルパスの候補を抽出する手法，それらへのリソース割り当て手法を検討する必要がある．

3.7　コンピューティングリソース不足時の配置における課題

本研究では，開発者によって，Dataflow アプリケーションが展開される際に，それを構成するコンポーネントの配置先ネットワークレイヤを決定し，その決定に従って，プラットフォームによりコンポーネントが配備されることを想定している．また，配備によって消費されたリソース量は，Dataflow プラットフォームパラメータである利用可能なリソース量に反映されるため，コンポーネント配置先ネットワークレイヤは，その時点での利用可能なリソース量に基づいて算出される．リソースの消費によって利用可能なリソース量がなくなった場合は，それ以上のコンポーネントを配置することができないため，配置不能となる．配置不能となる原因として根本的なリソース不足と，コンポーネントが消費するリソース量の予測誤りによるリソース不足の 2 種類が考えられる．3.7.1 節では，リソース不足となる場合に発生する問題と，それに対応する際の課題について説明する．3.7.2 節では，コンポーネントが消費するリソース量の予測精度により発生するリソース不足について説明する．

3.7.1　リソース不足に対応する際の課題

既に説明したように，コンポーネント配置先ネットワークレイヤは，その時点での利用可能なリソース量に基づいて算出されるが，複数の開発者によって同時に Dataflow アプリケーションが展開される場合，利用可能なリソース量がうまく反映されず，競合が発生する可能性がある．例えば，同時に 2 つのコンポーネントの配置先を算出した際に，リソース上限を超えて同じネットワークレイヤに配置するように決定する可能性がある．この場合，リソース上限を超えているため，どちらか一方のコンポーネントは配備できない．リソース上限に達するのがいずれかひとつのネットワークレイヤだけの場合，コンポーネント配置先の決定をシーケンシャルに行うことで競合の発生は防げる可能性がある．しかし，すべてのネットワークレイヤの利用可能なリソース量がなくなる場合や，指定されたネットワークレイヤ以外には配備できない場合は防ぐことができない．

上記のような問題の解決方法として，単純にコンピューティングリソースを追加する方法と，その時点で配備されているすべての Dataflow アプリケーションに対して，静的にリソース使用率を最大化するような配置パターンを算出し，その時点での最適な配置となるように再配置する方法が考えられる．後者の方法については，例えば，別途遅延を考慮する必要はあるが，Wang ら [52] の提案するオンラインアルゴリズムの比較対象として挙げられているオフラインアルゴリズムを用いることで，リソース使用率に関しては最適な配置を算出できる．オンライ

ンアルゴリズムは，単一のアプリケーションの配置に着目して，比較的短時間に近似解を得るアルゴリズムであるのに対して，オフラインアルゴリズムは，多少時間はかかるが，すべてのアプリケーションの配置など複雑な計算を対象としたアルゴリズムである．再配置のコストはかかるが，後者のオフラインアルゴリズムを用いることで，問題を解決できる可能性がある．

クラウドコンピューティングにおいては，利用可能なリソース量が減ってきた際に，自動でスケールアウトを行うオートスケーリングの機能が提供されている．エッジにおいても，同様な機能が提供されることが想定され，このような機能を使うことで，自動で利用可能なリソースを追加することができる．一方で，リソースが追加されてから，それを認識し，Dataflow プラットフォームパラメータである利用可能なリソース量に反映されるまでにはある程度の遅延が予想される．そのため，上記のような競合が発生する場合や，残りのリソースが少なくなってきた場合には，配置の方針を調整することが考えられる．方針を変えない場合，そのまま先着順でエッジのリソースから順に消費していき，エッジのリソースがなくなった場合，すべてのコンポーネントがクラウドに配備される．エッジに配置した場合と，クラウドに配置した場合で，大きく運用コストが変わるような場合，エッジのリソースが追加された時点で，エッジを含めて再配置を検討することとなる．そのままの方針で運用すると，リソースが追加されるまでの運用コストが大きくなること，別途，再配置のコストがかかることが懸念される．そのため，リソース上限にかかりそうな場合には，より運用コストに影響を与えるアプリケーションにエッジを使ってもらえるように，プラットフォーム側から，配置先ネットワークレイヤの決定におけるリソース量のコスト計算を調整できることが求められる．しかし，提案手法では，そのようなパラメータを導入できていない．このようなケースに対応するため，今後プラットフォーム側から配置先ネットワークレイヤを調整するためのパラメータの導入を検討する必要がある．

ここで，配置先ネットワークレイヤの決定において，配置不能となった際に，アプリケーションを展開する開発者からプラットフォーム側に利用可能なリソースを追加するように要求することが考えられる．この場合，リソースの追加に伴うコストと，それによって抑えられるコストを比較することが求められる．例えば，クラウドにはリソースがあるが，エッジにはリソースがない場合を想定すると，配置先ネットワークレイヤの決定ではエッジにコンポーネントを配置できないため，通信コストが大きくなり，配置不能と算出されることが考えられる．エッジにリソースを追加することで，通信コストや遅延が抑えられるが，その分，エッジへのリソース追加に伴い，運用コストが増大し，クラウドに配置した場合に比べて，全体的にみた場合にコストが大きくなる可能性がある．このようなケースに対応し，より汎用的に使用可能なプラットフォームとするため，今後，リソースの追加を考慮して配置パターンを算出する仕組みを導入する必要がある．

3.7.2 消費リソースの予測精度により生じるコンピューティングリソースの不足

既に説明したように，コンポーネントによって消費されるリソース量はそのコンポーネントへの入力データ量と関数 *predictRequiredResources()* から算出することを想定している．先頭のコンポーネントについては，Dataflow アプリケーションパラメータとして指定された入力データ量を用いて，そのコンポーネントの要求リソース量を算出する．それ以外のコンポーネントにおいては，コンポーネントのパラメータとして与えられる入出力比を用いてひとつ前のコンポーネントの出力データ量を予測し，その値を用いて対象のコンポーネントの要求リソース量を算出する．

一方で，入力されるデータの内容によって，出力データ量および要求リソース量が大きく変動する場合がある．例えば，人物検知コンポーネントなど，人物を検知しなかった場合は出力データ量が 0 となるような場合がある．このような要求リソース量の変動が大きなアプリケーションでは，割り当てたリソースが過剰となる場合や，逆にリソースが不足する場合が考えられる．コンポーネント配置後の要求リソース量の変動への対応として，実際の環境で一定期間動作させた後，実際に送受信されるデータ量や処理負荷に応じて，前節と同じようにコンポーネントを再配置することが考えられる．ここで，再配置の頻度を高くすることによりコンポーネント配置を最適化できる可能性がある．しかし，再配置処理による運用コストの上昇が懸念される．今後，入出力データ量の変動に伴う処理遅延や余剰リソースの発生に伴うコスト増加に対するユーザの許容量などを考慮した再配置機能が必要になると考えられる．

3.8 結言

本章では，コンポーネント配置先ネットワークレイヤ決定手法を提案した．また，配置パターンを評価するためにデータ転送，リソース消費，通信/処理遅延を考慮したコストの測定方法を定義した．また，コストを推定するために求められるパラメータを抽出/分類し，計算に反映することで，遅延/リソース消費/予算の観点で最小コストとなるコンポーネント配置パターンを探索する手法を提案した．最後に，シンプルな Dataflow アプリケーションを実装し，2 つのユースケースを用いて，提案手法の有効性を検証した．

4 コンポーネント間通信手法

4.1 緒言

本研究では，現実的に活用可能な遅延と運用コストを考慮した Dataflow アプリケーションの展開を目指し，コンポーネント配置手法と，コンポーネント間通信を分離した手法を提案している．コンポーネント配置手法については，既に述べたように，データセンタ内での通信など，比較的短い遅延を省略することで，計算量を抑えながらコンポーネント配置先ネットワー

クレイヤを決定する手法を提案した．本章では．決定されたネットワークレイヤに対応するクラスタ付近に配置されたコンポーネントのリソース状況を考慮したコンポーネント間通信手法を提案する．

リソース状況を考慮してコンポーネントをつなげるために，配置されたコンポーネントからリソース情報を収集する必要がある．一方で，その収集範囲はターゲットとするアプリケーションによって異なる．Zhang ら [54]，Ascigil ら [5]，Ishihara ら [25] は，通信遅延やトラフィック量などのアプリケーションの要件を考慮したコンポーネント配置手法を提案している．Zhang ら [54] の提案では，無線アクセスネットワークにおけるエッジサーバクラスタからプラットフォームが構成されることを想定しており，各エッジサーバをリアルタイムでモニタリングすることで，適切なコンポーネント配置を実現している．ターゲットとされるクラスタでは，エッジサーバの台数が限定されており，全てのノードのリソース情報が取得可能であると想定されている．Ascigil ら [5] は特定のプライベートエッジ・クラウドをターゲットとしたアプリケーションの展開手法を提案している．Ascigil ら [5] がターゲットとするクラスタは，Zhang ら [54] のものと比べて規模が大きく，ネットワーク全体のリソース情報が取得できることが求められる．我々がターゲットとする Nakashima ら [36] によって提案されているバスサービスのための乗降者数カウントアプリケーションのような地理的に広がって動作するアプリケーションの場合，コンポーネント配置のターゲットとなるクラスタは，バスの経路の近くに位置するクラスタとなり，その範囲でのリソース情報が必要となる．しかし，そのクラスタの範囲は各バスの経路によって異なる．適切な範囲でのリソース情報の収集を可能とする柔軟なリソース収集機能が必要となる．

一方，P2P オーバレイ技術を活用した PIQT [39] は，リソース情報の効率的な収集に活用可能な 2 つの機能を持つ．一つ目は，効率的なレスポンス収集のためのマルチキャストメッセージのレスポンスメッセージを集約する機能である．以降ではこれをリソース情報リクエストと呼ぶ．二つ目は，安倍 [55] によって提案されている効率的なメッセージルーティングを実現するための，リソース情報リクエストにより集められたリソース情報の集約機能である．これら二つの機能に基づいて，本論文では，2 つのコンポーネント選択手法 Multicast と Anycast を提案する．Multicast 手法は，シンプルにリソース情報リクエストを用いて全ての利用可能なコンポーネントの情報を収集し，その中からひとつを選択する．全てのノードへリクエストを送信するため，遅延が増大するが，信頼性のある選択が期待できる．Anycast 手法は，メンテナンスメッセージと共に更新される集約された値を参照するため，少しのメッセージおよび小さい遅延で適切なコンポーネントを選択できる．しかし，その手法では集約された値の鮮度に依存して選択ミスが発生する可能性がある．さらに，集約メッセージを活用する場合，メンテナンスメッセージがオリジナルのものより大きくなるため，トラフィックの増大が懸念される．これら 2 つの手法をコンポーネント選択に適用するには，集約された値の特徴を考えながら，メッセージング遅延やトラフィック量を調査する必要がある．

本論文の貢献は，PIQT における Suzaku を活用したキャッシュ情報を活用したトラフィック

量と遅延を考慮した配送を可能とする Anycast 手法の提案である．しかし，提案手法ではすべての情報をキャッシュとして保持するようになっており，トラフィック量が多くなることが課題として残る．また，提案手法はその特性上からキャッシュ情報が更新されていることが求められるが，既存の更新手法では，更新にかかる時間が長く，その適用範囲が狭いことが課題として残る．それに対して，安田 [58] は，Anycast 手法の特性を考慮して，キャッシュとして保持する範囲を限定することで，更新にかかる時間を短縮し，その適用範囲を広げている．以降では，プロトタイプの Anycast 手法について議論を進める．

4.2　コンポーネント間通信における想定環境

本研究では，Dataflow アプリケーションの例として，2.4 節で説明したようなものを想定するが，本章では，簡潔にコンポーネント間通信について説明するため，図 11 に示すような Dataflow グラフに着目する．

コンポーネント接続においては，図 11 に示すような Dataflow グラフに従って，単一のコンポーネントから単一のコンポーネントに接続することが求められる．Teranishi ら [49] によってコンポーネント接続手法が提案されている．2.2 節で説明したように，彼らの提案では，Chord# を用いて実装した P2P ネットワークを活用して，各コンポーネントを直接 P2P ネットワークに参加させることで，クラスタ情報を考慮したコンポーネント接続を可能としている．一方で，直接コンポーネントを P2P ネットワークに参加させる場合，その部分の実装が必要となり，コンポーネント開発コストが増大する．ここで標準化されたプロトコル [29] に対応した分散型 MQTT ブローカである PIQT [39] がある．本研究では，これを経由してコンポーネントを接続することで，コンポーネントの開発コストを抑えつつ，柔軟に接続する手法を検討する．PIQT では，ブローカ間の通信に P2P 構造化オーバレイである PIAX [59] を用いており，Teranishi [48] らの提案と同じようにクラスタ情報を考慮して，コンポーネント間を接続することができる．しかし，PIQT では，一般的な Pub/Sub のみサポートしており，我々が想定するような Dataflow 処理には，そのままでは活用できない．具体的には，コンポーネント間通信では，コンポーネント名をトピックとして活用して，コンポーネント間で通信を行うが，通常のトピックベース Pub/Sub におけるメッセージ転送では，ひとつのメッセージはそのトピックをサブスクライブする全てのノードに配送される．そのため，例えば，図 12 の左のように負荷分散したい場合にも，図の右のように複数の同一アプリケーションを展開する場合にも，同一

図 11: Dataflow グラフの例

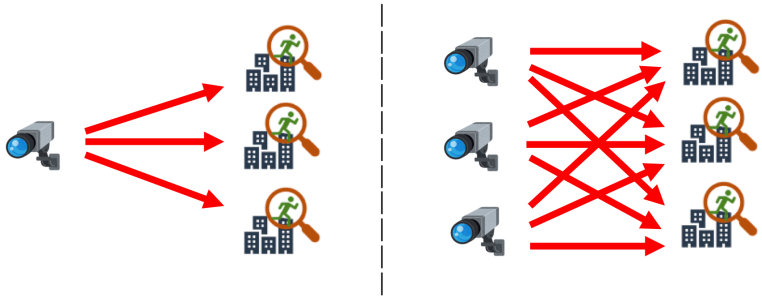


図 12: 一般的な Pub/Sub におけるメッセージ配送

名のコンポーネントのすべてに配送されるため，実現できないという問題がある．単一のコンポーネントに配送する手法として，Teranishi ら [49] によって提案されているように，サブスクライブ時にインデックス値の範囲を付与し，その値に対してラウンドロビンなどのロジックを適用してパブリッシュすることで，実現できる．一方で，CPU 使用率などの情報を活用して適切なコンポーネントに転送するには，それらのリソース情報を別途収集する必要がある．さらに，サブスクライブ時に振り分けたインデックス範囲に大きく依存し，柔軟にスケールさせることが難しい．他にも，アプリケーション ID などのユニークな ID を各コンポーネントに付与することで解決できるが，冗長化やリソース消費を抑えるためにコンポーネントを共有するといったことができない．本論文では，柔軟にコンポーネント間を接続するために，Pub/Sub でのメッセージ配信を拡張する形で，同一トピックをサブスクライブするノードのうち適切な 1 ノードにメッセージを配送する Anycast 手法を提案する．PIQT においては，送信時にオリジナルの配送方法か提案手法による配送かを指定することで切り替えを実現する．

ここで，コンポーネント接続形態には大きく 2 種類ある．ひとつは，キャプチャしたデータの比較など，ひとつ前のデータを参照しながらデータを処理するタイプのアプリケーションである．この場合，全てのデータは同一のコンポーネントに配送される必要があり．データ配送前に，コンポーネント予約メッセージによってコンポーネントを予約した上でデータを配送するような，予約ベースの処理が求められる．もうひとつは，個々のデータを独立して処理するタイプのアプリケーションである．この場合，それぞれのデータをどの共有コンポーネントにも配送できる．どちらの場合においても，配送先のコンポーネントをひとつ選ぶ必要がある．リソース状態を考慮してひとつのコンポーネントにメッセージを配送できれば．その配送手法は両方のケースで活用できる．以降では，予約ベースの処理のケースを想定して議論する．

次に，PIQT ブローカを活用したコンポーネント接続手法の概要を説明する．以降では，ひとつの PIQT ブローカを略してブローカとして表現し，その ID を B_i ($i \in N$) として表す．想定する Dataflow アプリケーションに対する提案手法に基づくコンポーネント配置の例を図 13 に示す．ブローカ (B_i) は全てのネットワークレイヤのコンピューティングリソース上に配置され，その後で適切なネットワークレイヤに位置するコンピューティングリソース上にコンポー

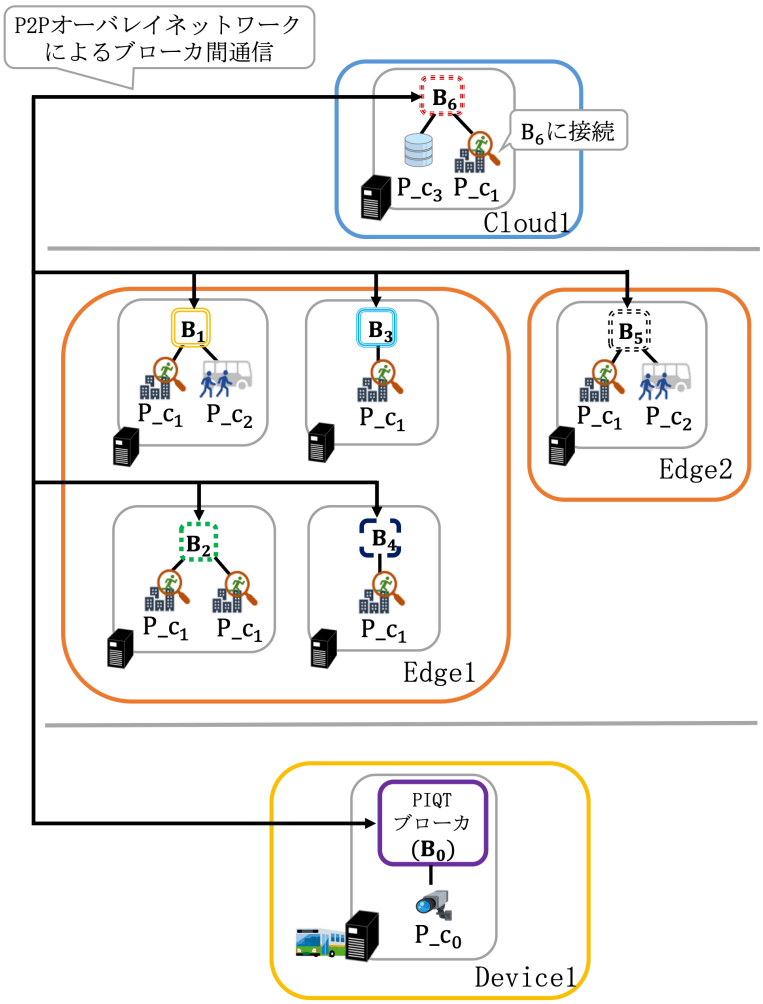


図 13: PIQT ブローカを用いたコンポーネント間通信

ネントのインスタンス (P_c$_i$) が配置される．P_c$_i$ は，同じコンピューティングリソースか，同じネットワークレイヤに位置するコンピューティングリソース上に配置された B$_i$ に接続し，B$_i$ を経由して，他の P_c$_i$ とトピックベース Pub/Sub で通信する．さまざまなネットワークレイヤに配備される全てのコンポーネントを柔軟に接続するには，各ネットワークレイヤに少なくとも 1 台のブローカが配置される必要がある．しかし，ブローカ間を地理的距離を考慮せず接続すると，遠くのクラスタに配備されたブローカを経由するような形でメッセージがルーティングされ，無駄なトラフィックが発生する可能性がある．各クラスタ内で，ローカルな通信を閉じるために，Teranishi ら [48] の提案では，地理的な情報を用いてオーバレイ上のノードをグループ化するクラスタリング手法を提案している．例えば，同じないし，近くの POP に位置するブローカは “Edge1”，“Edge2” などのようなラベルでグループ化し，このラベル

を用いて隣接関係を考慮し，オーバレイ上のトピックをソートする．ここで，“Edge/Edge1”，“Edge/Edge2” のように複数レベルの階層も定義でき，ターゲットとするクラスタの範囲は実際に配備したブローカに対して柔軟に定義できる [48]．ただし，本論文では，簡潔に説明するため，“Edge1” や “Edge2” などの単一階層を用いる．このようなラベルを考慮した通信は PIQT でもサポートされており，以下の議論ではラベルを用いた効率的な通信ができることを想定する．

4.3　提案する単一コンポーネント選択手法

既に述べたように，本論文では，2 つのコンポーネント選択手法 Multicast と Anycast を提案する．複数の同名のコンポーネントからリソース情報を考慮し 1 つを選択するには，単純に一度配送先のリソース情報を問い合わせた上でコンポーネントを選択することが考えられる．PIAX では，Abe ら [1] によって提案された Suzaku オーバレイを提供する．Suzaku オーバレイでは効率的にリソース情報を収集するリソース情報リクエスト機能をもつ．単純に全体に問い合わせを送るに比べて，応答を集約しながら，行うため，効率的に収集できる．これを用いた手法を以降では Multicast 手法と呼ぶ．一方で，Suzaku オーバレイでは CPU 使用率などの値の集約やその集約値を用いたメッセージのルーティングをサポートしており，CPU 使用率が一番低いコンポーネントへメッセージをルーティングすることができる．Anycast は Multicast に比べてメッセージ数を減らせる可能性がある．集約値は，P2P オーバレイを維持するための更新クエリにより更新される．そのため，この手法では，更新クエリの前にコンポーネント予約リクエストを送ると，集約値の更新が不十分となり，コンポーネントの選択ミスを招く．つまり，Anycast の性能は集約値の更新頻度とコンポーネント予約リクエストの頻度に依存する．本研究では，PIQT で実現した一度問い合わせる Multicast 方式と集約値を活用した Anycast 方式をいくつかのパラメータを変動させ，コンポーネント選択方式の特性調査を行う．

4.4　コンポーネント選択手順

本節では，提案する Anycast 手法および Multicast 手法の詳細な手順について説明する．はじめに，4.4.1 節では，PIQT ブローカと構築されるオーバレイネットワークの関係について述べる．4.4.2 節では，Suzaku におけるノード情報の保持方法およびそれを活用したオーバレイ上でのマルチキャスト機能とその効率性を説明する．4.4.3 節では，Suzaku が提供する集約メッセージング機能の仕組みについて述べる．4.4.4，4.4.5 節では，コンポーネント選択手法である Multicast と Anycast について述べる．

4.4.1　想定環境

図 14 は図 13 の物理環境で作成した Suzaku のオーバレイネットワークの一部を示している．各ブローカはオーバレイ上に複数のノードを持つ．ひとつは，図 14 のオーバレイに示

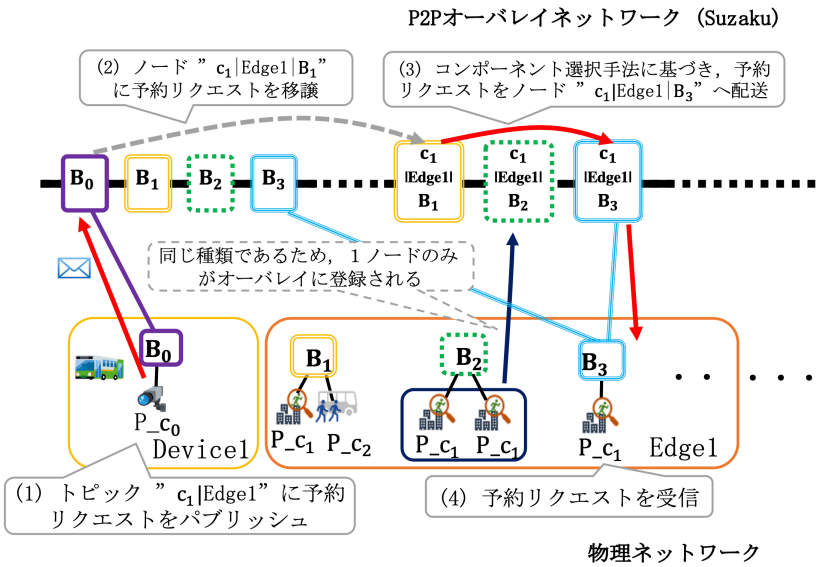


図 14: PIQT ブローカと構築されるオーバレイネットワークの関係

される B_3 などのブローカ自身を示すノードである．もうひとつは，ブローカに接続するサブスクライバの情報を示す “c_1|Edge1|B_3” のようなノードである．これは，“{topic}|{cluster name}|{broker ID}” という形式となっており，クラスタ名，ブローカ ID はコンポーネントがブローカにサブスクライブした時点で自動的に付与される．図 14 では，2 つの P_c_1 が B_2 に接続しているが，これらは同じ種類のコンポーネントであるため，“c_1|Edge1|B_2” のみがオーバレイに登録される．ここで，B_2 は直接接続しているため，2 つの P_c_1 の状態を正しく把握できる．B_2 がメッセージを受け取った際，B_2 はリソース状態を考慮して，適切な P_c_1 にメッセージを転送する．以降では，オーバレイ上でのコンポーネント選択に着目する．また，c_0 はサブスクライブせず，パブリッシュのみ行うため，オーバレイには登録されない．

図 13 での Pub/Sub によるコンポーネント間のメッセージ配送は，実際には図 14 のようにオーバレイネットワークを経由して行われる．Teranishi ら [48] によって，メッセージ配送処理をターゲットとなるクラスタに位置するノードに移譲し，移譲されたノードからメッセージを伝搬させる Delegate 手法が提案されている．これを用いることで，複数のクラスタにまたがる通信の増加を防ぐことができる．PIQT ブローカでも Delegate 手法はサポートされており，本提案においても Delegate 手法を用いてコンポーネント予約メッセージを配送する．図 14 は，P_c_0 がブローカ B_0 を経由して，“c_1|Edge1” というトピックを用いて，ノード “c_1|Edge1|B_1” にコンポーネント予約メッセージをパブリッシュする例を示している．はじめに，ブローカ B_0 が持つオーバレイ上のノード B_0 から Delegate 手法によって，リクエストの処理が “c_1|Edge1|B_1” ノードに移譲される．移譲されたノードは Multicast 方式か Anycast 方

式に従ってオーバレイ上の適切なノードへリクエストを配送する．その後，受信したノードは PIQT ブローカに組み込まれた MQTT ブローカを経由してコンポーネントへリクエストメッセージを送信する．Anycast 方式，Multicast 方式のいずれにおいても，Delegate 手法を活用して，メッセージ配送を行う．以降では，Suzaku におけるオーバレイ上でのノードの管理方法およびノード間のメッセージ配送方式について述べる．

4.4.2　Suzaku オーバレイ上でのブローカ間メッセージング

本節では，トピックとしてのコンポーネント名の保持手法およびそのトピックをサブスクライブするコンポーネントへのメッセージ配送手法について説明する．以下では，まず，Suzaku オーバレイにおけるノード管理について説明し，そのあとで，オーバレイが提供するクエリを用いたメッセージ配送について説明する．

Suzaku は，Chord$^{\#}$ を拡張したキー順序を保つ構造化オーバレイの 1 つである．各ノードは全順序集合の要素であるキーを保持し，ノードはそのキーの順序を保ちながらリング構造で整列される．各ノードは自身のもつキーの次に大きなキーを持つノードのポインタである Successor と自身のもつキーの次に小さいキーを持つノードのポインタである Predecessor を保持する．ただし，リング構造を保つため，最大のキーを持つノードの Successor は最小のキーを持つノードを指し示し，最小のキーを持つノードの Predecessor は最大のキーを持つノードを指し示す．さらに，キーの検索効率を向上するために，各ノードは 2 つの FingerTable(FT) を持つ．FT は，2 の累乗個先のノードへのポインタの配列で表現される．ひとつは時計回り方向にノードへのポインタを保持する Forward Finger Table (FFT) であり，もうひとつは時計回り方向にノードへのポインタを保持する Backward Finger Table (BFT) である．以降では，あるノード u が持つキー，FFT，BFT をそれぞれ u.key，u.FFT，u.BFT とする．ここで，u.FFT の i 番目の要素を u.FFT[i], u.FFT[i] が指すノードを u.FFT[i].node，リモートのノード u の i 番目の FFT を取得する関数を u.*getFFT*(i) とすると，u.FFT[i] は Chord$^{\#}$ を拡張した安倍 [55] の提案手法と同じように式 16 のように定義される．ただし，$i = -1$ の場合，自ノードを指す．また，最大の i を k，全ノード数を $n \in \mathbb{N}$ とすると，k は $\lceil \log_2 n \rceil - 1$ となる．

$$u.\mathrm{FFT}[i] = \begin{cases} u.\mathrm{node} & (i = -1) \\ \mathrm{Successor} & (i = 0) \\ u.\mathrm{FFT}[i-1].\mathrm{node}.getFFT(i-1) & (i > 0) \end{cases} \tag{16}$$

FT を保つために，ノード挿入時，削除時にターゲットのノードは FT に含まれるノードへ更新クエリを送信する．さらに，ノードの故障などを検知し，FT を修正するために各ノードは定期的に更新クエリを送信する．ノードが参加した場合，参加したノードは，FFT[0], BFT[0], FFT[1], BFT[1], ..., FFT[k], BFT[k] の順に更新クエリを送信する．脱退する場合，脱退するノードは，自身をポインタとして持つノードへ更新クエリを送信する．定期的な更新では，FFT[0], FFT[1], ..., FFT[k] の順に FFT に含まれるノードにのみ更新クエリを送信する．更新

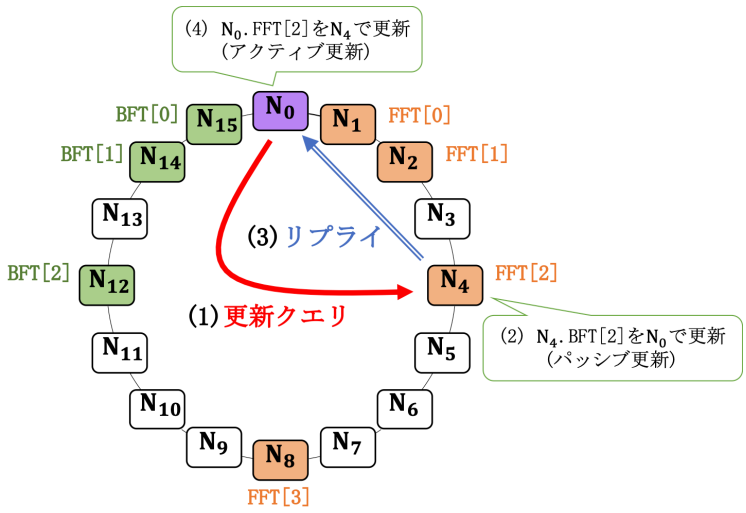


図 15: Suzaku におけるアクティブ更新とパッシブ更新の例

には，アクティブ更新とパッシブ更新の 2 種類がある．アクティブ更新とは自ノードの FT を更新するために他ノードに更新クエリを送信し，そのリプライを用いて更新することである．パッシブ更新とは更新クエリの送信元ノードの要素で受信ノードの FT を更新することである．図 15 に Suzaku におけるアクティブ更新とパッシブ更新の例を示す．図 15 は全体のノード数 n が 16 の時のリングを示す．各ノードは N_i をキーとしてもち，そのキーの辞書順で整列される．例えば， N_0 が FFT[2] を更新する場合，N_0 は N_4 に更新クエリを送信する．その後，クエリへのリプライを用いて N_0.FFT[2] を更新する．これがアクティブ更新である．一方で，パッシブ更新はノードが更新クエリを受け取った際に行われる．例えば，N_4 は，受け取った更新クエリに含まれる情報を用いて，送信元ノードである N_0 を指し示すポインタ N_4.BFT[2] を更新する．

Suzaku では，Chord$^\#$ と同様にレンジクエリを活用してメッセージを配送する．以下では，図 15 において，N_0 から $N_2, N_3, \ldots, N_8$ のすべてのノードにメッセージを配送する例を説明する．まず，N_0 は，[N_2.key, N_9.key) という範囲を生成する．[N_2.key, N_9.key) は N_2.key $\leq$ key $<$ N_9.key を意味する．N_0 は N_0.FT に含まれるノードのキー情報を元に生成した範囲を分割して，各ノードに分割した範囲を指定してレンジクエリを送信する．この場合，N_2 に与えられる範囲は [N_2.key, N_4.key)，N_4 に与えられる範囲は [N_4.key, N_8.key), N_8 に与えられる範囲は [N_8.key, N_9.key) となる．範囲の分割は再起的に行われ，最終的にはじめに生成した範囲に含まれる全てのノードにメッセージが配送される．リプライは，レンジクエリの配送経路を辿って，それぞれの範囲で集約されながら，N_0 へ配送される．

次に Suzaku のレンジクエリを用いた PIQT ブローカ間でのメッセージの配送について説明する．図 13 で示したようにコンポーネントのインスタンスとブローカを配置した場合，図

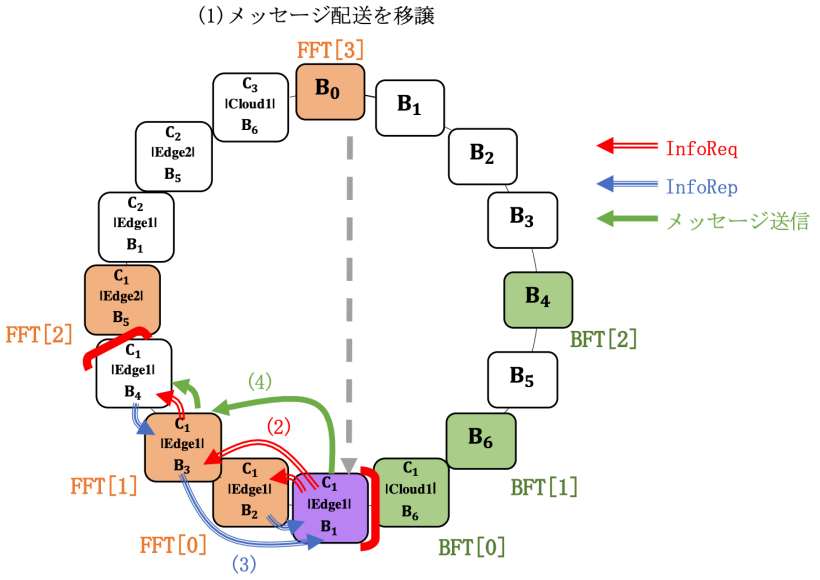


図 16: Multicast 方式の例

16 に示すようなオーバレイネットワークが構築される．4.4.1 節で述べたようにブローカは，$B_i(B_{min} < B_i < B_{max})$ というキーで登録される．図 16 における “c_1|Edge1|B_3” などはオーバレイ上のキーを示す．また，オーバレイ上のノードはトピック，クラスタ名，ブローカ ID の優先順で，キーを辞書順でソートし並べられる．

以下では，B_0 から Edge1 に位置する c_1 を持つ全てのノードにメッセージを配送する際の PIQT での配送例を説明する．はじめに，B_0 は検索範囲として [{c_1|Edge1|B_{min}}.key, {c_1|Edge1|B_{max}}.key) を生成する．つぎに，生成した範囲の処理をノード “c_1|Edge1|B_1” に移譲する．移譲されたノードは，受け取った範囲を [{c_1|Edge1|B_2}.key, {c_1|Edge1|B_3}.key) と [{c_1|Edge1|B_3}.key, {c_1|Edge1|B_{max}}.key) の 2 つに分割し，それぞれの範囲でのレンジクエリをノード “c_1|Edge1|B_2” とノード “c_1|Edge1|B_3” に送信する．これらの処理は再帰的に行われるため，前のノードと同じように，ノード “c_1|Edge1|B_3” は，ノード “c_1|Edge1|B_4” に [{c_1|Edge1|B_4}.key, {c_1|Edge1|B_{max}}.key) という範囲のレンジクエリを送信する．その後，オーバレイ上でメッセージを受け取ったブローカは，直接接続されるコンポーネントにメッセージを送信する．この時，既に説明したように，ブローカは接続されるコンポーネントのリソース情報を考慮して，適切なひとつのコンポーネントにメッセージを送信する．

4.4.3　Suzaku オーバレイの各 FT で保持される集約値

集約値は，Suzaku オーバレイ上の各ノードの各 FT[i] に保持される値である．指定された属性の値が指定された関数により集約される．各 FT[i] で集約されるノードの範囲は，[FT[i].node.key, FT[$i+1$].node.key) である．集約はオーバレイのメンテナンスに合わせて，FT

表 7: “c_1|Edge1|B_1” の FT

FT[i]	集約範囲	集約ラベル: 集約値
FFT[−1]	[{c_1|Edge1|B_1}.key, {c_1|Edge1|B_2}.key)	c_1|Edge1 : 1
FFT[0]	[{c_1|Edge1|B_2}.key, {c_1|Edge1|B_3}.key)	c_1|Edge1 : 1
FFT[1]	[{c_1|Edge1|B_3}.key, {c_1|Edge2|B_5}.key)	c_1|Edge1 : 2
FFT[2]	[{c_1|Edge2|B_5}.key, B_0)	c_1|Edge2 : 1 c_2|Edge1 : 1 c_2|Edge2 : 1 c_3|Cloud1 : 1
FFT[3]	[B_0, {c_1|Edge1|B_1}.key)	c_1|Cloud1 : 1
BFT[0]	[{c_1|Cloud1|B_6}.key, {c_1|Edge1|B_1}.key)	c_1|Cloud1 : 1
BFT[1]	[B_6, {c_1|Cloud1|B_6}.key)	null
BFT[2]	[B_4, B_6)	null

の更新時に集約結果が FT にキャッシュする形で行われるため，集約のオーバヘッドは，クエリを用いたものより小さい．以降では集約値を保持する仕組みについて説明する．

あるノード u が保持する特定の属性の値を u.value，FFT[i] の集約範囲を FFT[i].range，キャッシュされた集約値を FFT[i].value と表記する．FFT[i].range は FFT[i].node.key, FFT[$i+1$].node.key) であり，FFT[i].node から時計周りの方向に 2^i 個のノードを含む．反対に反時計回りの方向に 2^{i-1} 個のノードを含み，BFT[i].range は [BFT[i].node.key, BFT[$i-1$].node.key) となる．ただし，$i=0$ の時は，BFT[0].range は [BFT[0].node.key, FFT[-1].node.key) となり，ノード数は 1 となる．例として，表 7 は図 16 のノード “c_1|Edge1|B_1” の FT の集約範囲と集約値を示す．表 7 では，ブローカに接続するコンポーネント数を集約する値とし，各ノードは “{ トピック }|{ クラスタ名 }” というラベルをつけて，コンポーネントの数を集約する．FFT[-1] は自ノードを指し，その集約範囲は自ノードのみとなり，自ノードの値がそのまま FFT[-1].value に保持される．ここで，集約値は FT の更新と一緒に行うため，集約値の鮮度は FT 更新の頻度に依存する．例えば，{c_1|Edge1|B_1}.FFT[2] のアクティブ更新を行う時，リクエストの送信者であるノード “c_1|Edge1|B_1” は集約範囲 [FFT[-1].node.key, FFT[1].node.key) の生成および，宛先ノード “c_1|Edge2|B_5” のパッシブ更新のための FFT[-1], FFT[0], FFT[1] の集約値の用意を行う．そして，リクエストの送信者であるノードはその範囲と用意した集約値を含めて，宛先ノードへ更新クエリを送信する．更新クエリを受信したノードは，集約値を含めて送信者に更新クエリのリプライを送信する．それと同時に，受信したノード “c_1|Edge2|B_5” は，更新クエリに含まれる情報に基づき，{c_1|Edge2|B_5}.BFT[2]” の集約値を更新する．

次に集約値の鮮度について説明する．FFT[i] の更新クエリが送られる場合, 受信側は FFT[0],

. . . , FFT[i − 1] の集約値を用意する必要がある. FFT[-1] は常に最新の値を示すが，それ以外の値は参照する FT の状態に依存する．ここで，各ノードが更新クエリを送信する間隔を T 秒とすると，特定のノードの値が変更されてから，あるノードが全ての FFT の集約値を更新するまでに，FT[0], . . . , FFT[k] のそれぞれの更新に T 秒掛かかり，合計 $(k+1)T$ 秒必要となる．さらに，その集約値を正しく更新するには，参照する他ノードの集約値が正しく更新されている必要がある．例えば，{c_3|Cloud1|B_6}.value が変更された場合に，{c_1|Edge1|B_1}.FFT[2] を正しく更新するには，{c_1|Edge2|B_5}.FFT[0] と {c1|Edge2|B_5}.FFT[1] が更新される必要がある．以上から，特定のノードのターゲットとする値が変更されてから，全ノードの全集約値が正しく更新されるまでに，最悪 $(k+1)^2T$ 秒かかる．例えば，T を PIAX のデフォルト値である 60，ノード数を 100 とした時，全ての集約値を正しく更新するには，1 時間以上必要になる．ユースケースの要件に合わせて，適切に T を設定することが求められる．

4.4.4　Multicast 方式によるコンポーネント選択

本節では，レンジクエリを活用したコンポーネント情報集約方式である Multicast 方式を用いたコンポーネント選択について説明する．既に説明したように，Anycast 方式，Multicast 方式に関わらず，メッセージ配送では，クラスタにまたがる通信を抑えるために，Teranishi ら [48] によって提案されている Delegate 方式を活用する．両方式において，送信者ははじめにコンポーネント選択を検索範囲にあるノードに移譲し，移譲されたノードから各方式に基づいてコンポーネントを選択する．以降では，移譲されたノードを送信者と呼ぶ．

Multicast 方式は，送信者はコンポーネントに関係するリソース情報をレンジクエリを用いて収集し，収集した情報を考慮して適切なコンポーネントにコンポーネント予約リクエストを配送する．B_0 からリクエスト処理を移譲された c_1|Edge1|B_1 が Edge1 に位置する c_1 の考慮リストを収集し，選択したノードにリクエストを配送する手順を図 16 に示す．はじめに，ノード “c_1|Edge1|B_1” は，[{c_1|Edge1|B_{min}}.key, {c_1|Edge1|B_{max}}.key) の範囲で，リソース情報リクエストを送信する．その後，集約されたリプライを受信し，受信した情報を考慮して，コンポーネント予約リクエストを送信する．以降では，リソース情報リクエストとそのリプライを InfoReq および InfoRep と呼ぶ．Edge1 位置するの全ての c_1 のリソース情報を収集するために，範囲検索により各ノードにリソース情報を要求するメッセージを送信する．Mulicast 方式では，送信者が一度すべての候補ノードに InfoReq を送信する必要があるが，確実に適切なコンポーネントを選択することができる．

4.4.5　Anycast 方式によるコンポーネント選択

本節では，集約値を活用する方式である Anycast 方式の 2 つの方式を用いたコンポーネント選択について説明する．

安倍 [55] は，集約値を活用し，条件に合ったノードにメッセージを配送する条件付き Multicast を提案している．本論文で提案する Anycast 方式では，条件付き Multicast を拡張

Algorithm 3 集約値を活用した Anycast 方式

```
▷ r: 検索範囲
▷ r_min: r 内の最小のキー
▷ MATCH: value が条件を満たす場合に Boolean 型で true を返す関数
▷ SELECTONE: 候補ノードからひとつを選択する関数
▷ e: FT のひとつの要素
function CONDANYCAST(r, MATCH, SELECTONE)
    candidateElems ← [e|(e ∈ FT)∧ MATCH(e.value) ∧ (e.range ∧ r ≠ ∅)]
    if candidateElems is not empty then
        selectedElem ← SELECTONE(candidateElems)
        if selectedElem.node is FFT[−1].node then
            ▷ 選択を終了する
        else
            selectedElem.node. CONDANYCAST((selectedElem.range∧r), MATCH, SELECTONE)
        end if
    else
        r ← 検索範囲を初期化する
        FFT[−1].node. CONDANYCAST(r, MATCH, SELECTONE)
    end if
end function
```

し，条件に合ったノードから 1 つを選択し，メッセージを配送する．

Algorithm 3 は，Anycast 方式の手順を示す．r, r_{min}, e はそれぞれ検索範囲，検索範囲の最小のキー，FT の 1 つの要素を示す．MATCH 関数は $e.value$ が条件を満たすかどうかに基づいて，Boolean 値を返す．SELECTONE は候補ノードからひとつを選択する関数である．本稿では，具体的なアルゴリズムは指定せず，目的に応じて切り替えられるものとする．シンプルな戦略として，コンポーネント数に基づいて適切なコンポーネントを選ぶ手法があるが，それについては 4.5.2 節で述べる．はじめに送信者であるノード u は，CONDANYCAST(r, MATCH, SELECTONE) を呼び出す．*candidateElems* には，MATCH および $(e.range \wedge r \neq \emptyset)$ を満たす FT の要素のリストが入る．*selectedElem* は SELECTONE によって *candidateElems* から選ばれた要素である．選択されたノードが現在のノード *self*(*self*.FFT[-1]) である場合，そこで選択を終了する．そうでない場合は，現在のノードは *selectedElem.node* に *selectedElem.range* と r の重なった範囲である現在の範囲より狭い範囲を r として渡し，CONDANYCAST() を呼ばせる．このように再帰的に実行され，最終的に条件にあった単一のノードにメッセージは配送される．ターゲットとなるコンポーネントを持つ候補がなかった場合，選択に失敗したとみなされ，範囲を初期化した上で，現在のノードから再配送される．

4.4.2 節と同様の環境での Anycast 方式を用いた配送例を図 17 に示す．ここで，MATCH 関数で指定される条件は，集約ラベルに “c_1|Edge1|” が含まれることである．この例では，SELECTONE はターゲットラベルの集約値が大きいノード，つまり，多くの候補ノードを持つノードを選択するように定義する．移譲されたノード “c_1|Edge1|B_1” の FT を示す表 7 から，

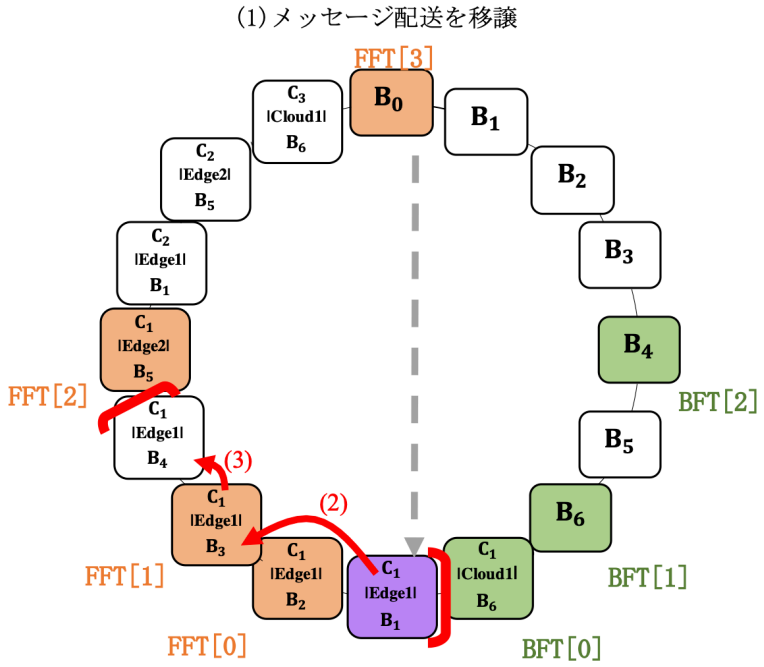


図 17: Anycast 方式での配送例

FFT[0].value はひとつ候補を持ち，FFT[1].value は 2 つ候補を持つ．この場合，FFT[1].node が選択され (図 17 の (1))，選択されたノード FFT[1].node は CONDANYCAST() を呼ぶ．その結果として “c_1|Edge1|B_3” か “c_1|Edge1|B_4” が選ばれる (図 17 の (2))．ここで，FT の要素がコンポーネントの候補をいくつか持っていた場合，それらのうちから 1 つがランダムに選択される．さらに，複数のノードが同時に CONDANYCAST() を呼び出す場合，配送が特定のノードに偏り，コンポーネント予約が衝突する可能性がある．これを回避する手法としては，SELECTONE では重み付きランダムアルゴリズムを用いることなどが考えられる．アルゴリズムの詳細は 4.5.2 節で述べる．

図 17 の場合，メッセージは 2 ホップで配送でき，Anycast は，Multicast に比べて半分のホップ数での配送を可能としている．しかし，Anycast の性能は，集約値の鮮度に依存する．値の更新が十分に行われていない場合や，ターゲットとなる値と選択ロジックがマッチしていない場合，性能低下が見込まれる．提案手法の有効性を検証するため，集約値の鮮度の影響や選択ロジックの性能を明らかにする必要がある．ここで，リソース情報を考慮した負荷分散を実現するには，CPU 使用率やメモリ使用量などを集約値として用いて，リソースの使用状況に合わせて適切なノードを選択することが求められる．しかし，本論文では，上記の点に着目し，シンプルに利用可能なコンポーネント数を集約値として採用し，Anycast 手法の評価を行う．

Anycast 方式の性能が低くなるケースとして，集約値が十分に更新されていない場合がある．

はじめはすべてのノードは利用可能であるが，予約が進むにつれて，残りの候補ノード数が少なくなる．集約値が十分に更新されていない場合，実際の残りの候補ノード数と，集約値ベースの残りの候補ノード数が異なり，選択ミスを招く．次節では，特に，検索範囲に含まれるノード数に対するホップ数の変化について調査する．ここで，オーバレイ上での転送では，全体のノード数もホップ数に影響する．一方で，既に説明したように提案手法では，Delegate 手法を用いて検索範囲にあるノードに処理を移譲した後で，Anycast 方式によりメッセージを配送する．そのため，全体のノード数ははじめの移譲する際のメッセージングのホップ数にのみ影響し，その性能はオリジナルの Suzaku でのメッセージングに相当する．これらの理由から，以下の評価では検索範囲内での移譲されたノードからのメッセージ配送に着目する．

4.5　提案するコンポーネント選択手法の比較

本節では，提案手法に対するホップ数とトラフィック量を計測し，それらの手法の適用可能性を検討する．4.5.1 節，4.5.2 節では集約値の想定と選択ロジックの詳細について説明する．4.5.3 節，4.5.4 節では評価環境および評価手順について説明する．各選択ロジックにおける検索範囲に含まれるノード数に対するホップ数を 4.5.5 節で述べ，その後，集約値の鮮度の性能への影響について 4.5.6 節で述べる．さらに，4.5.7 節では，集約値のサポートにより増加する更新クエリのメッセージサイズの全体のトラフィック量への影響について述べる．最後に，4.5.8 節で，2 つの方式を比較し，それらの方式の適用可能性について議論する．

4.5.1　想定する集約値とその更新頻度

本論文では，Ishihara ら [27] の提案手法と同様に利用可能なコンポーネントの数を集約対象とする．既に述べたように，Anycast 方式でのメッセージ配送のホップ数は集約値の鮮度に依存する．Ishihara ら [27] の提案手法では，コンポーネント予約リクエスト送信間隔 T_w が $(k+1)T$ より長ければ，ホップ数が十分収束することが明らかにされている．しかし，T_w が 0 から $(k+1)T$ の間となるような細かい粒度でのホップ数が計測できていないため，それより T_w が短いアプリケーションをデプロイする際に，そのアプリケーションの遅延要件を満たすかどうかの判断ができない．そのため，本論文では，T_w を $0 \leq T_w \leq (k+1)T$ の範囲で変更しながら計測を行う．ここで，更新クエリ送信間隔 T を短くすると，T_w が短いアプリケーションの遅延要件も満たせるが，同時にトラフィック量の増加も招く．遅延要件とトラフィック量の両方を考慮して，適切に更新クエリ送信間隔を設定することが求められる．

4.5.2　選択ロジックの詳細およびその特徴

本節では，選択ロジックの詳細について説明する．選択ロジックは，Algorithm 3 の SELECTONE 関数で使用される．FT から 1 つの要素を選ぶ戦略はいくつか考えられるが，本論文では Random，Many，Few，Close の 4 つの戦略を検討する．

Random は乱数で選択するロジックである．ここで，Algorithm 3 では，MATCH でフィルタした FT の要素である $candidateElems$ から選択するロジックとなっている．しかし，集約値を使用しない場合のホップ数を調査するために Random の場合のみ，MATCH でのフィルタリングを行わない．Many は，4.4.5 節で説明したロジックに似たアルゴリズムを用いてひとつの要素を選択する．違いは，重みつけランダムの部分である．その詳細は次の段落で説明する．Many では，集約範囲内に多くの候補をもつ FT の要素が選ばれるが，より集約範囲が広い要素を選ぶ傾向があるため，オーバレイ上で遠くに位置する候補が選ばれる可能性がある．Many とは対照的に Few では値が小さい要素が優先的に選択される．このロジックは、参照される集約値のキャッシュが無効である場合に，候補コンポーネントがない要素を選択する可能性があるが，キャッシュが有効な場合，集約範囲が小さい要素を選択し，少ないホップ数でメッセージ配送できる．Close は，オーバレイ上において近傍に位置するノードを指し示す要素を選択する．遠くのノードに比べて，近くのノードの集約値はより頻繁に更新されるため，近くの要素を選択する場合，参照する集約値が正確である可能性が高い．そのため，選択誤りによるホップ数の増加が抑えられる可能性がある．4 つの戦略の中では，Close がホップ数が小さくなり，かつ，なるべく近いノードに送られる可能性が高いことから，遅延とトラフィック量の両方を考慮した配送を実現ために，活用できる可能性が高いと考えられる．一方で，コンポーネント予約リクエストの送信間隔が短い場合，キャッシュミスにより，既に予約済みの近傍ノードへメッセージが転送され，他のロジックに比べ，Close の方が遅延とトラフィック量が大きくなる可能性がある．以降では，これら 4 つの戦略に対して評価を行い，遅延とトラフィック量を考慮した配送を実現するために，活用できる戦略を明らかにする．

4.4.5 節で説明したように，本稿での SELECTONE では重み付きランダムアルゴリズムを用いて，候補ノードからひとつを選択することを想定する．ただし，要素への重み付け方法は Many，Few，Close で異なる．本稿では，簡易的に累積和を用いた重み付きランダムアルゴリズムを実装した．SETWEIGHT によって与えられる重みは整数値としており，その累積和から Java における Rand 関数を用いて，ランダムな値を取得し，各ノードの値とランダム値を比較することで，ひとつのノードを選択する．例えば，候補となるノードを A, B, C, D の 4 ノードとし，A: 1, B:2, C: 2, D:1 のように重み付けされた場合，各ノードの範囲は，A: 1, B: 2∼3, C:4∼5, D:6 となり，累積和である 6 を用いて得られたランダム値が，2 であった場合，B を選択する．ただし，Rand 関数に 6 を渡す場合，その返り値は 0∼5 になるが，範囲を合わせるため，返り値にプラス 1 している．

Ramdom には重み付きランダムは導入しない．Algorithm 4 は，SELECTONE で使用する SETWEIGHT 関数を示す．Many では，集約値に対して降順で要素の重みをつける．反対に Few では，昇順に重みをつける．これは，最大の集約値に 1 を足した値から要素の集約値を引いた値を重みとすることで実現する．Close では，FT のレベルに対して昇順で要素に重みをつける．

Algorithm 4 各 SELECTONE に対する重みつけアルゴリズム

```
1: ▷ candidateElems: FT の要素の順序を保持したフィルタされた FT の要素
2: function SETWEIGHT(strategy, candidateElems)
3:     if strategy is Many then
4:         totalNum ← sum(e.value | e ∈ candidateElems)
5:         for e in candidateElems do
6:             e.weight ← e.value/totalNum
7:         end for
8:     end if
9:     if strategy is Few then
10:        maxNum ← max(e.value | e ∈ candidateElems)
11:        for e in candidateElems do
12:            e.weight ← (maxNum + 1) − e.value
13:        end for
14:        totalNum ← sum(e.weight | e ∈ candidateElems)
15:        for e in candidateElems do
16:            e.weight ← e.weight / totalNum
17:        end for
18:    end if
19:    if strategy is Close then
20:        maxLevel ← max(candidateElems.level(e) | e ∈ candidateElems)
21:        for e in candidateElems do
22:            e.weight ← (maxLevel + 1) − candidateElems.level(e)
23:        end for
24:        totalNum ← sum(e.weight | e ∈ candidateElems)
25:        for e in candidateElems do
26:            e.weight ← e.weight / totalNum
27:        end for
28:    end if
29:    return candidateElems
30: end function
```

4.5.3 評価環境

コンポーネントが配置されるネットワークレイヤは [25] で提案されたコンポーネント配置手法により決定される．バスサービスのための地理的に広がって展開されるアプリケーションにおいては，それを構成する Dataflow コンポーネントも地理的に分散して配置される．さまざまな場所に位置するコンポーネントを接続し，かつ各エリア内でのローカル通信をサポートするには，ブローカもまた地理的に分散して配置することが求められる．本論文では，全てのクラスタに均等にブローカが配置されることを想定する．シンプルな構成ではあるが，選択ロジックおよび集約値の基本的な性質の解析を簡単化するため，上記の構成を採る．本評価では，Dataflow アプリケーションの例に基づいて，3 つの種類のコンポーネントを用意する．また，トピック数は 3 つとなる．図 18 は評価環境の概要を示す．

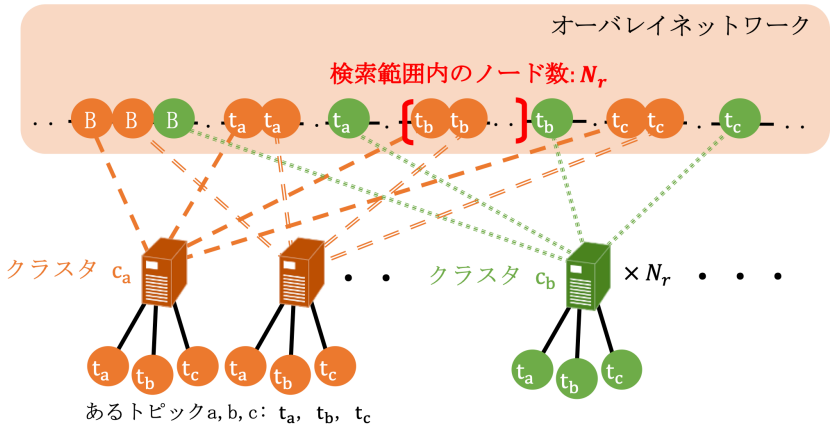


図 18: 評価環境の概要

前節で述べたように，提案手法の性能は検索範囲のノード数，つまり “{topic}|{cluster name}” をキーとして持つノード数 (N_r) に依存する．そのため，N_r を変更しながら評価を行うことが求められる．しかし，ブローカでは，複数の同一コンポーネントはひとつのノードとしてオーバレイに登録されるため，N_r を変更するには，同じキーをもつブローカ数を増やす必要がある．このような環境では同じキーをもつブローカ数も同様に N_r で表せる．以上からすべてのブローカ数が N_b で与えられるとすると，クラスタ数 N_c は次のように表せる．

$$N_c = \frac{N_b}{N_r} \tag{17}$$

また，シンプルに各ブローカが同じ 3 つのトピックを持つことを想定しているため，ブローカは自身を示すノードを含め，オーバレイ上に 4 つのノードを登録する．結果としてオーバレイ上のすべてのノード数 N_n は次のように表せる．

$$N_n = 4N_b \tag{18}$$

以上から，N_c と N_n は，N_b と N_r から算出できる．そのため，以降では N_b と N_r を変更しながら評価する．

本論文では，P2P フレームワークである PIAX を活用してオーバレイを構築しているが，PIAX のコードは広域分散環境で動作できるように設計されており，提案手法の検証には多くの計算リソースを必要とする．一方で，実際のネットワークを使わないメッセージレベルのイベントシミュレータが実装されている [1]．本評価では，そのシミュレータを用いて提案するアルゴリズムの評価を行う．

表 8: パラメータ

Name	Description
T_w	コンポーネント予約リクエスト送信間隔
N_r	検索範囲に含まれるノード数
N_b	ブローカ数

4.5.4 評価シナリオ

本節では，評価シナリオについて説明する．既に述べたように，コンポーネントの予約リクエストに着目して，予約に要したホップ数を計測する．4.4.4 節で述べたように，コンポーネント予約リクエストを移譲されたノードを送信者とする．評価でのホップ数は移譲後のメッセージ配送によるものを対象とする．また，利用可能なコンポーネント数が集約値として使われる．具体的には，コンポーネントの状態が集約値として登録される．状態 0 は使用済みを意味し，状態 1 は利用可能であることを示す．これらのコンポーネントの状態はブローカに登録され，この状態の合計が集約対象となる．

1 試行では，Multicast 方式，Anycast 方式のいずれかのコンポーネント選択手法に従って，指定されたトピックを持つ全てのコンポーネントが予約されるまで，コンポーネント予約リクエストを送信する．ただし，試行前には，$(k+1)^2T$ 秒待機し，全ての集約値を更新する．Multicast 方式では検索範囲に含まれるすべてのノードに InfoReq が送信され，結果を集約しながら送信者に InfoRep が返されるため，その経路は木構造で表せる．Multicast 方式において，InfoReq/InfoRep が送信される段階では，経路を示す木の辺の数を合計ホップ数として計測する．一方で，これらは並列に送信されるため，最大ホップ数は木の高さから計測する．その後でユニキャストによる予約段階のホップ数を計測する．4.5.5 節では，ホップ数により遅延を評価するため，Multicast 方式におけるホップ数を InfoReq/InfoRep が送信される段階の最大ホップ数と予約段階のホップ数の合計とする．4.5.6 節では，トラフィック量を評価するため，InfoReq/InfoRep が送信される段階の合計ホップ数と予約段階のホップ数の合計とする．試行回数を 50 回とし，表 8 に示すパラメータを変更しながら，2 つの方式でのホップ数を計測する．

また，比較のため，最小のホップ数で予約した場合のホップ数も計測する．以降ではこれを Optimal 方式と呼ぶ．この手法では，評価の前に全てのノードの状態を把握し，その情報を用いて送信者から最小のホップで到達可能なノードから順に全てのノードを予約する．以降では，Optimal，Multicast，Anycast のホップ数を比較する．

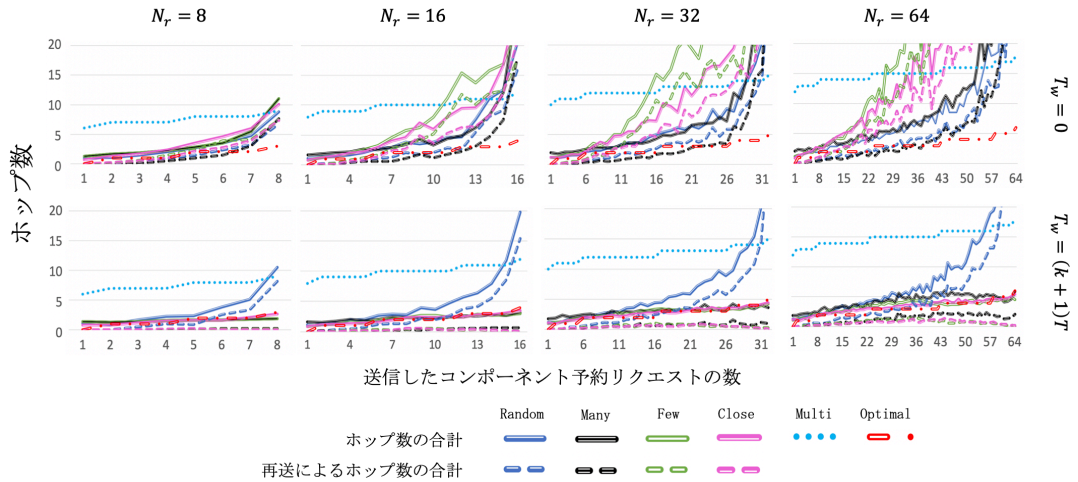


図 19: N_r の変化に対するホップ数の推移

4.5.5　選択ロジックと N_r に対するホップ数

本節では，はじめに N_r 変動しながら選択ロジックに対するホップ数を計測し，N_r の影響を調査する．そのあとで，選択ロジックごとのホップ数の違いを調査し，各 FT の要素の使用率を解析する．

$N_b = 128$ の際の計測結果を図 19 に示す．各グラフの縦軸は平均ホップ数を示し，横軸はコンポーネント予約リクエストの配送数を示す．凡例は各選択ロジックの総ホップ数および再送のためのホップ数を示す．また上段と下段はそれぞれ $T_w = 0$ と $T_w = (k+1)T$ の場合での結果を示す．N_r は，8, 16, 32, 64 のように FT と同じように 2 の累乗で変化させて計測した．ここで，$T_w = 0$ は集約値を更新しないことを意味する．この場合，更新せずホップ数を計測した．

結果から，Anycast 方式の場合，N_r の増加に伴いホップ数も増加し，$T_w = (k+1)T$ よりも $T_w = 0$ の方がより増加した．$T_w = 0$ と $T_w = (k+1)T$ の再送回数を比較すると，$T_w = 0$ の方が再送によるホップ数が多くなっている．これは集約値の更新が不十分であったため発生したと考えられる．$T_w = 0$ の場合，残りの利用可能なコンポーネント数が少なくなる後半のリクエストでは，Multicast や Optimal に比べると Anycast のホップ数が大きくなる．一方で，集約値の鮮度が不十分であっても，利用可能なコンポーネントが十分に残っている場合，少ないホップでターゲットのノードへ到達できている．特に，Close と Few ロジックでは，この傾向が強い．反対に $T_w = (k+1)T$ の場合，Few と Close の結果は Optimal に近いホップ数となった．

既に述べたように大きいレベルの FT ほど，その集約値の更新は遅くなる．選択ロジック間のホップ数の違いはこれに起因すると考えられる．図 20 では，上記の計測結果のうち $T_w =$

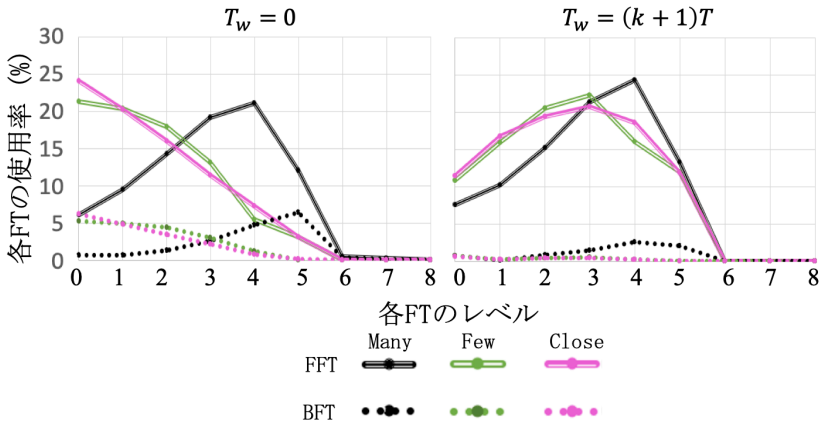


図 20: 選択ロジックごとの FT の各レベルの使用率 ($T_w = 0, (k+1)T$, $N_r = 64$)

$0, (k+1)T$ (N_r =64) の場合の結果における選択ロジックごとに各 FT の使用率を示す．各グラフの縦軸は FFT と BFT ノードの使用率を示し，横軸は選択ロジックで使われた FFT と BFT のレベルを示す．その割合は，試行回数 50 回の各 FT の使用率の平均した値となっている．

$T_w = 0$ では，ホップ数が大きくなる Few や Close はより低いレベルの FT を頻繁に使う傾向がある．低いレベルの FT は集約範囲が狭いため，候補となる要素数が少なくなる．そのため，不正確な集約値を参照し，頻繁に再送処理が行われ，ホップ数が増加したと考えられる．実装において，予約結果に基づいて使用済みの FT の要素をキャッシュすることで，ホップ数を減らせる可能性があるが，その拡張は今後の課題である．一方で，少ないホップ数を示した Many では，広い集約範囲を持つ高いレベルの FT を使用している．これは，誤った集約値を参照したとしても，代わりの候補が多くあるため，再送処理とホップ数が抑えられたと考えられる．

$T_w = (k+1)T$ では，全体的に BFT の使用が少なくなり，FFT を多用する傾向にある．$T_w = 0$ での結果に比べると，Few や Close は FFT[3] より高いレベルの FT を使っている．既に使用済みである低いレベルの FT は使っておらず，集約値の十分な更新によりコンポーネントの状態がキャッシュに正しく反映されていると考えられる．一方で，Many は $T_w = 0$ と $T_w = (k+1)T$ で同じような使用率を示しているが，ホップ数は増加している．Many ではより多くの候補をもつノードに配送しようとするため，たとえ選択されたノードが利用可能なノードであっても，より多くのノードをもつ高いレベルの FT に配送され，ホップ数が増加したと考えられる．例えば，現状のノードが条件を満たした場合，選択処理を終了するなど，アルゴリズムを拡張することで，ホップ数の増加を抑えられる可能性がある．

Ramdom は T_w が変わっても同じホップ数を示している (図 19)．Random は集約値を参照しないため，その傾向は変わらない．要素の選択に偏りがないため，$T_w = 0$ の場合，Few や Close よりホップ数が少なくなる場合がある．一方で，$T_w = (k+1)T$ の場合，集約値を使っ

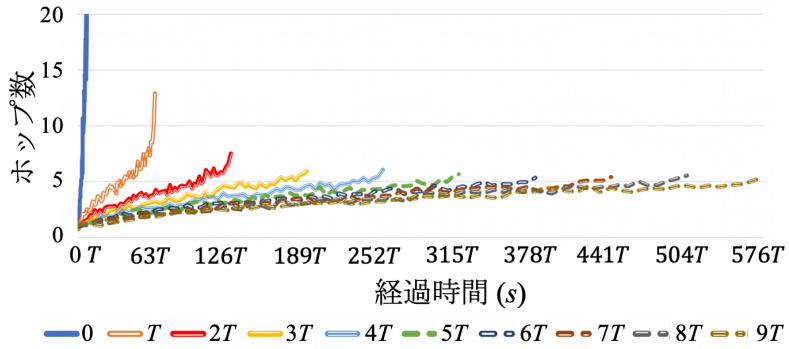


図 21: Close ロジックにおける T_w 対するホップ数 (N_r = 64, N_b = 128)

た選択ロジックの方がホップ数が少なくなる．今回のケースでは集約値を使わない場合の傾向を調査するために，Random において MATCH を用いたフィルタを省略したが，これを適用し，集約値を活用した上でランダムを採用した場合，他の選択ロジックより良い結果を示す可能性がある．今後調査することが求められる．

図 20 に示すように，FFT と比べると BFT の使用率は著しく低い．本評価では，Delegate 手法を活用した配送を想定しており，この手法を用いると，Teranishi ら [48] によって提案されたロジックに従い，送信者が必ず検索範囲の左端に位置することとなる．そのため，送信者は選択ミスが発生しないかぎり，常に FFT を使う．選択が失敗した場合，選択を失敗したノードが送信者となり，リクエストの再配送を行う．この場合においてのみ，送信者が左端意外に位置するため，BFT を使う可能性がある．それゆえ，$T_w = (k+1)T$ \$ より，$T_w = 0$ の方が再送回数が増加するため，BFT の使用率が高くなっている．

上記の結果から，更新が十分に行われる場合，Few と Close は利用可能なコンポーネントを持つノードに少ないホップ数で到達できる．Few は単純に狭い範囲のノードを使うが，常に近傍のノードを使うとはかぎらない．そのため，以降では，オーバレイ上の近くのノードを選択するロジックである Close にのみ着目し，Anycast 方式が Multicast 方式より良い性能を発揮するケースを調査する．

ここで，図 20 に示されるように，$N_r = 64$ の場合，FFT と BFT の両方において 6 以上のレベルをほとんど使っていない．これは $\log_2 N_r$ より高いレベルの FT は使われないことを示す．また，N_b の増加，つまり全体ノード数の増加は FT のサイズの増加につながるが，N_r が固定の場合，配送において活用される FT のレベルは変わらない．以上から，全体のノード数の変更は Anycast 方式と Multicast 方式のホップ数には影響しない．

4.5.6　集約値の鮮度に対するホップ数

本節では，集約値の鮮度が Anycast 方式の性能にどのように影響するかを調査するため，$0 \leq T_w \leq (k+1)T$ の範囲で T_w を変更し，ホップ数を計測する．

図 21 は，$N_r = 64$, $N_b = 128$, $T_w = 0$, T, $2T$, ..., $8T$, $9T$ とした時の，Close ロジックでの平均ホップ数を示す．縦軸は平均ホップ数，横軸は経過時間を示す．結果から，$T_w = (k+1)T$ (= $9T$) の場合，全てのコンポーネントが予約されるまでに，$64(k+1)T$ (= $576T$) 秒かかる．また，T_w が短いほど，ホップ数が増加する．$T_w = T$ の時，全てのコンポーネントを予約するために要したホップ数は，図 19 の Multiacst 方式の結果より少なくなる．以上から，$T_w \geq T$ の時，活用される集約値は十分に更新され，Multicast 方式より Anycast 方式の方が良い性能を示す．

一方で，Anycast 方式を使うには，オーバレイ上で集約値が保持される必要があり，オーバレイのメンテナンスのためのトラフィックが増加する．以降では，増加するトラフィック量を明らかにするため，集約値を維持するために追加されたメッセージサイズを調査する．

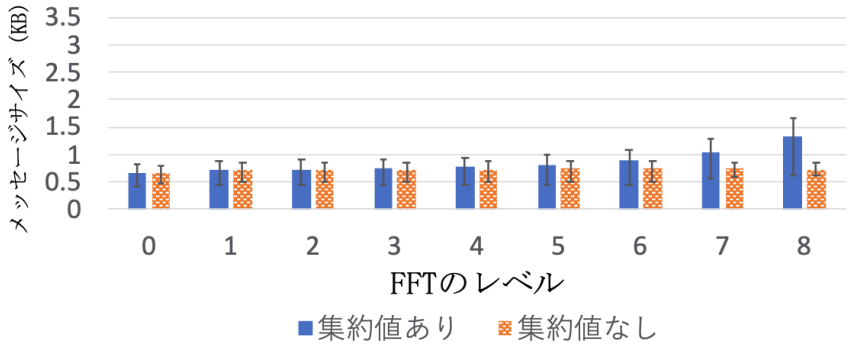
4.5.7　集約値を維持するために増加した更新クエリとリプライのメッセージサイズ

各 FT の集約値を更新するためのメッセージは，更新クエリとそのリプライに含まれる．4.4.3 節で述べたように，レベルの高い FT の要素は広い集約範囲を持ち，更新のためのメッセージサイズが大きくなる．本節では，4.5.4 節で述べたシナリオに沿って，$(k+1)^2T$ 秒間に生成された更新クエリとリプライのメッセージサイズを計測する．ただし，$N_b = 128$, $N_r = 8$ とする．

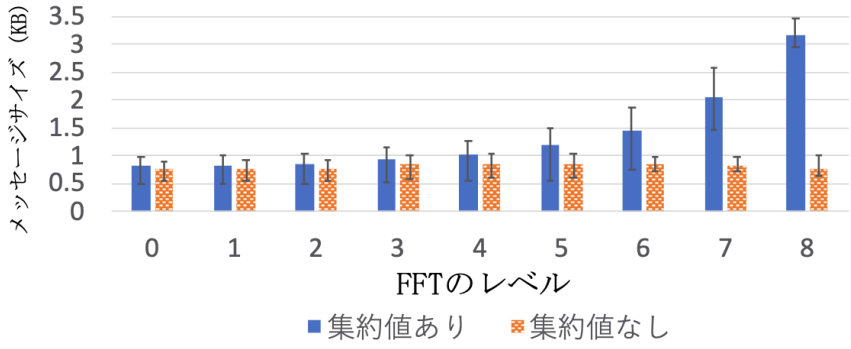
図 22 (a) と (b) に FT のレベルに対する更新クエリとリプライのメッセージサイズを示す．縦軸は $(k+1)^2T$ 秒間に送受信された平均メッセージサイズ，横軸は FT のレベルを示す．エラーバーは最大，最小の値，凡例は集約値の有無を示す．ここで，既に説明したように，集約範囲に含まれるノード数は，FT のレベルが上がるにつれて増加する．提案手法では，集約範囲に含まれるすべてのノードの値を集約するため，FT のレベルが上がるほど，集約ラベルおよび集約値の組みの数は増加し，当然メッセージサイズも増加する．

4.4.2 節で説明したように，FT の要素の更新クエリは T 間隔で FFT[0], ... FFT[k] の順で送信される．そのため，$(k+1)T$ (= $9T$) 秒間に生成されたトラフィック量は，図 22 (a) と (b) の k=8 の時のメッセージサイズを足し合わせることで計算できる．集約値を含む場合の総メッセージサイズは約 20.00KB となる．また，それ平均すると，T = 60.00 の場合，1 秒ごとに各ノードで生成される更新のためのトラフィック量は約 37.04 B/s となる．同様にして，集約値を含まない場合の総メッセージサイズは約 13.80KB となり，T = 60.00 の時，1 秒間の各ノードの更新トラフィック量は約 25.56 B/s となる．

メッセージサイズの増加は，トラフィック量の増加につながる．ここで，前節で説明したように，レベルが $\log_2 N_r$ 以上の FT 要素は予約リクエストの配送には使われない．例えば，FT のレベルが $\log_2 N_r$ 以上の場合は，集約をスキップするように実装することで，トラフィック量を抑えられる．機能拡張により，改善できる可能性があるが，今後の課題である．



(a) 更新クエリのメッセージサイズ



(b) 更新クエリリプライのメッセージサイズ

図 22: 集約値を含む場合と含まない場合での更新メッセージサイズの比較

4.5.8　コンポーネント選択手法の検討

Anycast 方式と Multicast 方式はそれぞれ利点，欠点を持つため，条件に応じてコンポーネント選択手法を選択することが望ましい．本節では，コンポーネント予約リクエストの送信間隔に着目して，その条件について議論する．

2 つの方式を比較するために，評価で得られた計測結果に基づいて，トラフィック量を計算する．はじめに計算で用いる変数について説明する．Anycast 方式において，すべてのコンポーネントを予約するために要した総ホップ数を H_a とする．また，Multicast 方式での総ホップ数を H_m とする．ここで，すべてのメッセージサイズは，集約値を含まない場合の更新クエリのサイズとほとんど同じである．計算の簡単化のため，ここでは，すべてのメッセージサイズを S とする．以上から，Anycast 方式，Mulsicast 方式におけるコンポーネント予約リクエストのための総トラフィック量はそれぞれ H_aS，H_mS から計算できる．

T_w を 0，T，$2T$，$3T$ と変化させた場合の Close ロジックでの Anycast 方式と Multicast 方

表 9: 予約に要したトラフィック量の比較

(a) H_aS における T_w と N_r に対するトラフィック量

N_r \ T_w	0	T	$2T$	$3T$
64	920.5 KB	283.3 KB	194.8 KB	182.1 KB

(b) H_mS における T_w と N_r に対するトラフィック量

N_r \ T_w	0
64	6450 KB

式のトラフィック量を計測する．ただし，S = 800.0 bytes，N_r = 6，N_b = 128 とする．H_a と H_m の計算には，図 19 および図 21 で計測した平均ホップ数を用いる．表 9 に H_aS と H_mS の計算結果を示す．

表 9 (a) の H_aS から，ホップ数は減少するため，T_w の増加に伴い，予約に要するトラフィック量も減少する．表 9 (b) の H_mS から，ホップ数は T_w には依存せず，予約のためのトラフィック量は一定となる．結果から，Multicast 方式ではすべての候補ノードにコンポーネント予約リクエストを送信するため，たとえ $T_w = 0$ の場合でも H_mS より，H_aS の方が小さくなる．したがって，予約に要するトラフィック量は Multicast 方式より，Anycast 方式の方が少なくなる．

ここで，4.5.7 節で述べたように，集約値を保持するため，Anycast 方式では，更新クエリとリプライのメッセージサイズが増加する．増加したトラフィック量を調査するため，両方式において，更新クエリとリプライのトラフィック量を算出する．計算では，表 9 の条件に加えて，いくつか変数を導入する．Anycast 方式，Multicast 方式において，1 秒間に更新クエリとリプライによって生成されたトラフィック量をそれぞれ U_a，U_m と表す．それぞれ 4.5.7 節で計測した更新トラフィック量と全ノード数 N_n から算出できる．以上から，Anycast 方式における 1 秒間の総トラフィック量を A_{total}，Multicast 方式にものを M_{total} とすると，それぞれ以下のように表せる．ただし，$T = 60.00$，$k = 8$ とする．

$$U_a = 37.04N_n \tag{19}$$

$$U_m = 25.56N_n \tag{20}$$

$$A_{\text{total}} = U_a + \frac{H_aS}{T_wN_r} \tag{21}$$

$$M_{\text{total}} = U_b + \frac{H_mS}{T_wN_r} \tag{22}$$

表 9 の条件に加えて，N_r を 8，16，32，64 と変化させながら，総トラフィック量を計算す

表 10: 1 秒間に生成されるトラフィック量の比較

(a) A_{total} における T_w と N_r に対するトラフィック量

N_r \ T_w	T		$2T$		$3T$	
	U_a	H_aS/T_wN_r	U_a	H_aS/T_wN_r	U_a	H_aS/T_wN_r
8	18.92	36.51×10^{-3}	18.92	13.76×10^{-3}	18.92	78.59×10^{-4}
16	18.92	46.89×10^{-3}	18.92	18.11×10^{-3}	18.92	10.84×10^{-3}
32	18.92	56.75×10^{-3}	18.92	21.83×10^{-3}	18.92	13.19×10^{-3}
64	18.92	62.08×10^{-3}	18.92	25.37×10^{-3}	18.92	15.81×10^{-3}

(b) M_{total} における T_w と N_r に対するトラフィック量

N_r \ T_w	T		$2T$		$3T$	
	U_m	H_mS/T_wN_r	U_m	H_mS/T_wN_r	U_m	H_mS/T_wN_r
8	13.08	20.18×10^{-2}	13.08	10.19×10^{-2}	13.08	67.27×10^{-3}
16	13.08	41.67×10^{-2}	13.08	20.83×10^{-2}	13.08	13.89×10^{-2}
32	13.08	84.03×10^{-2}	13.08	42.01×10^{-2}	13.08	28.01×10^{-2}
64	13.08	16.80×10^{-1}	13.08	83.98×10^{-2}	13.08	55.99×10^{-2}

る．表 10 は A_{total} と M_{total} の計算結果を示す．

表 10 は，それぞれ予約に要するトラフィック量と更新クエリとリプライのトラフィック量を示す．A_{total} と M_{total} は，ぞれぞれ上記の 2 つのトラフィック量の合計である．表 10 から，全てのケースで，A_{total} と M_{total} のうち，U_a と U_m が支配的となっている．つまり，ほとんどが更新クエリとリプライにより生成されたトラフィックとなっており，$T_w \geq T$ では，総トラフィック量は Anycast 方式の方が大きい．

ここで，更新クエリとリプライはオーバレイ上の全てのノード (N_n) を対象として計算しているが，予約リクエストは “{topic}|{cluster name}” のキーを持つノードのみを計算の対象としている．他のキーを持つノードへの予約リクエストも同時に行われることを想定すると，予約に要するトラフィックは検索範囲の数だけ増加する．例えば，$N_r = 8$ の場合，予約に要するトラフィックは 16 (クラスタ数) × 3 (トピック数) だけ倍増する．このような状況では，T_w が T に限りなく近いとき，Multicast 方式より，Anycast 方式の方が良い結果を示す．結果から，$T_w \geq T$ のとき，Anycast 方式は Multicast 方式より少ないホップ数での予約を可能とするため，トラフィック量を重視する場合，送信者は，コンポーネント選択手法として Anycast 方式を選択できる．しかし，$T_w < T$ では，両方式において，他の条件に比べ，トラフィック量と遅延が増大する．そのため，さらなる条件を検討する必要がある．

Anycast 方式はホップ数を抑えられるが，利用可能なコンポーネントが少ないような状況

ではホップ数が増加するため，Multicast 方式を使用することが望ましい．一方で，たとえ $T_w < T$ であっても，利用可能なコンポーネントが十分に残っていれば，Anycast 方式では，少ないホップでコンポーネント予約リクエストを配送できる．例えば，図 19 の $N_r = 64$, $T_w = 0$ の場合の結果から，利用可能なコンポーネントが 30 以上残っていれば，Close ロジックの Anycast は少ないホップを示している．以上から，残りのコンポーネント数に応じて，コンポーネント選択手法を切り替えることで，より効果的に実行できると考えられる．例えば，はじめは Anycast 方式を活用し，残りのコンポーネント数が減ってきた際に Multicast 方式に切り替えることが考えられる．しかし，切り替えの判断には，より具体的なアプリケーション要件，配備環境，予約戦略の想定が求められる．今後の課題として，$T_w < T$ の際のホップ数の調査や，実環境における Anycast 方式と Multicast 方式の組み合わせ手法の検討が求められる．

4.6 関連研究

Teranishi ら [49] の提案手法も Dataflow アプリケーションをターゲットとしており，同じトピックをサブスクライブするコンポーネント間でのロードバランシングをオーバレイ上で実現している．この手法では，各コンポーネントはインデックスの範囲を持っており，送信者がトピックに加えてインデックスを指定してメッセージをパブリッシュする．例えば，送信者はラウンドロビンなどのスケジューリング方法が選択できる．しかし，インデックスの割合としてしか分散できないため，CPU 負荷やメモリ消費などのリソース状態に応じたロードバランスの実現は難しい．本論文では，新たに集約されたコンポーネント情報を用いてコンポーネントを選択する手法を提案しており，リソース状態を考慮したロードバランスが実現できる可能性がある．

Shen ら [46]，Ren ら [30] によって，P2P ベースのリソースディスカバリ手法が提案されている．Shen ら [46] は，地理的距離を考慮した効率的なリソース探索手法を提案している．この手法では，ノードの CPU 使用率やメモリ利用量などのリソース情報をオーバレイのキーとして保持する．これにより，リソース情報を考慮したコンポーネント選択が可能となっているが，これらの情報は頻繁に変動するため，Churn が発生する可能性が高い．そのため，すべての情報をオーバレイ上にキーとして保持するのは難しい．我々の提案では，それらの情報は集約値として収集されるため，オーバレイの構造には直接は影響を与えない．Ren ら [30] は，複数のサービスから構成されるアプリケーションのための効率的なサービス検索手法を提案している．この手法では，各 P2P ノードは，ひとつのアプリケーション内でよく使われるサービスをグループ化された行列を持つ．この行列は一定の間隔でノード間を移動するエージェントによって更新される．我々の提案では，構造化オーバレイを用いており，Ren ら [30] の提案手法で指摘されている分散ハッシュテーブルではレンジクエリを実現できないという問題は，Suzaku で解決されている．

4.7　コンポーネント選択手法の切り替えの課題

最適なコンポーネント選択の検討には，2 つの方式を切り替えることが求められる．切り替え条件を明らかにするには，$0 < T_w < T$ などコンポーネント予約リクエストが高頻度で行われる状況での検証が求められる．さらに，更新クエリとリプライにより生成されるトラフィックを抑えるために，集約値のうちの使用しない部分を削除するなどの検討が必要となる．これらはモバイル環境など，より少ないトラフィック量での運用が求められるケースでの運用で重要となる．

4.8　CPU 使用率やメモリ使用量の集約値への適用の課題

集約値による負荷分散を実現するには，CPU 使用率，メモリ使用量やディスク容量等を集約させる必要がある．これらの情報は予約状態よりも変動が激しいことが予想される．そのため，ホップ数や，トラフィック量を考慮しながら，更新頻度や集約の粒度を検討する必要がある

4.9　選択ロジックにおける課題

提案したすべての選択ロジックでの Anycast 手法において，現状の実装では，検索範囲内の処理可能なコンポーネント数が少ない場合，コンポーネント選択に失敗し，複数回，再送処理が行われる可能性がある．これを防ぐため，Anycast 方式に TTL 等の仕組みを導入することが考えられる．例えば，ホップ数が TTL を越えた場合は Multicast 方式に切り替えることで，利用可能なコンポーネント数を正しく把握し，適切なコンポーネントにコンポーネント予約リクエストを配送できる．また，コンポーネント選択に失敗した際にそのノードの情報を保持しておくことで，再送時に同じノードに間違えて配送することを防げる．上記のようなホップ数の増加を抑える仕組みを導入することで，Anycast 手法の適用可能な範囲を広げられる可能性がある．その範囲については，今後調査することが求められる．

4.10　集約対象とする属性の限定の検討

コンポーネント予約リクエストの送信間隔が短くなると，Anycast 方式ではトラフィック量が多くなり，その適用可能な範囲が狭まる．これは全体の集約ラベルと集約値を含む更新クエリとリプライのメッセージサイズが大きいことに起因する．我々の提案では，Delegate 手法を採用しているため，ターゲットとする範囲内の集約ラベルと集約値のみを必要とする．例えば，集約対象となる属性を上記のようにフィルタリングすることでメッセージサイズを抑えられる．今後，これによる，全体のトラフィック量への影響を調査する必要がある．

4.11 結言

本章では，階層化ネットワークにまたがる Dataflow プラットフォームの実現に向けて，リソース情報を考慮したコンポーネント間通信手法の検討に取り組んだ．そして，Anycast 方式と Multicast 方式の P2P 技術を活用した 2 つのコンポーネント選択手法を提案した．

Multicast 方式は選択前に，該当するコンポーネントのリソース情報を収集するため，確実に利用可能なコンポーネントが選択ができる．一方で，Anycast 方式は，オーバレイのメンテナンスとともに更新される集約値を活用した効率的なコンポーネント選択を実現する．しかし，集約値に不整合がある場合，リクエストの再送が発生し，ホップ数が増加する．これらを使い分ける方法を明らかにするため，集約値のキャッシュの有効性に着目して，それぞれの手法において，メンテナンスメッセージを含めてホップ数とトラフィック量を評価した．

結果として，2 つのコンポーネント選択手法の特性を明らかにした．Anycast 方式は $T_w \geq T$ の時に Multicast 方式より少ないホップを示すため，トラフィック量を重視する場合，コンポーネント選択手法として Anycast 方式が使用できる．また，集約値が完全には更新されていない場合でも，利用可能なコンポーネント数が十分に残っている場合は Anycast 方式が適用できる．利用可能なコンポーネント数が少ない場合は，ホップ数の増加を回避するため，Multicast 方式を使うべきである．$T_w < T$ の条件下で適用する方式を選ぶには，残りのコンポーネント数を考慮する必要がある．

5 結論

本研究では，Dataflow アプリケーション運用時における遅延と運用コストの課題を解決するために，複数ドメインにまたがる階層化ネットワークを考慮した Dataflow プラットフォームにおいて，コンポーネント配置手法およびコンポーネント間通信手法を提案した．

第 1 章では，既存の Dataflow プラットフォームでは，エッジコンピューティング環境を考慮したアプリケーションの展開が難しいこと，既存の Dataflow アプリケーション展開手法では，実用的なアプリケーションの展開が難しいことを説明し，コンポーネント配置とコンポーネント間通信を分離する手法を提案した．コンポーネント配置においては，データセンタ内での短い遅延を省略し，クラスタ単位での配置パターンの計算を行うことで，計算量を抑えながら，適切なネットワークレイヤへのコンポーネント配置を実現する．コンポーネント間通信においては，ネットワークレイヤに関連するクラスタ付近でのリソース状況を考慮したコンポーネント探索を行い，適切にコンポーネント間を接続する．一方で，これらを実現するには，Dataflow プラットフォームにおいて，クラスタを考慮してコンピューティングリソースを管理する機能が求められることを説明した．

第 2 章では，本研究で想定する Dataflow プラットフォームのアーキテクチャを説明し，既存の Dataflow プラットフォームでは提供されていないクラスタを考慮したコンピューティン

グリソース管理手法を提案した．提案手法では，オープンソースとして提供されているコンテナ管理基盤上で，コンピューティングリソースをその地理的位置情報に結びつけて管理することで，コンポーネントの指定されたネットワークレイヤへの配置を可能とする．

第 3 章では，コンポーネント配置先として，適切なネットワークレイヤを決定する手法を提案した．提案手法では，コンポーネント配置に必要なコストを定義し，それをコスト最小化問題として解くことで，適切なコンポーネント配置パターンを算出する．そのために必要となるパラメータの抽出，コンポーネント配置にかかるコストの計算手法および，配置パターンを決定するためのアルゴリズムを提案した．また，提案したアルゴリズムを 2 つのユースケースに適用し，その有効性を示した．

第 4 章では，ネットワークレイヤに関連するクラスタ付近に配置されたコンポーネントのリソース情報を収集し，それに基づいて，適切にコンポーネント間を接続する手法を提案した．リソース情報の収集においては，収集対象となるコンポーネント数に応じて増加するトラフィック量が課題となる．これを解決するため，収集したリソース情報をキャッシュとして持ち，それを活用して接続先コンポーネントを選択する Anycast 方式と，効率的に範囲内のコンポーネントのリソース情報を収集する Multicast 方式を提案した．Anycast 方式においては，キャッシュの有効/無効における配送効率の変動が懸念される．本論文では，コンポーネント使用可否をキャッシュ対象とし，コンポーネント予約リクエストの送信間隔に対して，Anycast 方式でのメッセージ配送時のホップ数を計測した．結果として，コンポーネント予約メッセージの送信頻度が高くない場合は，提案手法により遅延およびトラフィック量を抑えながら，リソース情報を考慮した配送ができることを示した．以上から，コンポーネント予約後そのまま運用し続けるようなアプリケーション，つまり，バスサービスをターゲットとしたアプリケーションなど，高頻度でのコンポーネント配置および削除や繋ぎ変えが発生しないようなアプリケーションにおいては，トラフィック量と遅延を抑えながら適切なコンポーネントに接続することができる．

これらの提案手法を組み合わせることで，複数ドメインにまたがる階層化ネットワークを考慮した Dataflow プラットフォームにおいて，遅延，運用コスト，リソース量を考慮して Dataflow アプリケーションを展開することができる．ただし，コンポーネント間通信の性能の限界により，そのターゲットは比較的，アプリケーションの展開や削除，コンポーネント間の繋ぎ変えが低頻度であるものに限られる．例えば，VR，AR やバスサービス，家庭環境における IoT アプリケーションにおいては，一度，展開された後はそれほど頻繁にコンポーネントの繋ぎ変えやアプリケーションそのものの削除，再配置などは行われないため，本手法によりエッジの利点を活用できる．ここで，コンポーネント間通信においては予約ベースのコンポーネントの接続を想定している．この場合，予約の分オーバヘッドが発生するため，遅延が大きくなる．一方で，予約が発生するのは始めの一回のみであるため，上述のようなアプリケーションの場合，それ以降は遅延，トラフィック量を抑えつつ，通信できる．また，コンポーネント配置においては課題が残る．本論文の提案では，単一の Dataflow アプリケーションの展

開についてのみ着目しており，複数の Dataflow アプリケーションが展開される場合，先着順での配置となり，必ずしも最適な配置とはならない可能性がある．この場合，再配置により，ある程度コンピューティングリソースの空きが確保でき，最適な配置に修正できる可能性があるが，空きリソースがない場合，再配置時にすべてのアプリケーションが一旦止められる問題がある．これを解決するには，クラウドなど，比較的リソースが潤沢にあると想定されるクラスタにおいて，コンピューティングリソースを追加し，リソースが増設できないエッジの環境のみなど，ある程度範囲を限定した上で，カナリアアップデートなどの機能を用いて，アプリケーションを止めることなく，再配置を行うことが求められる．これを実現するには，既に展開されているすべての Dataflow アプリケーションに対して静的に最適な組み合わせを算出し，その時の配置と最適な配置を比較することで，再配置の対象となる Dataflow アプリケーションを抽出することが必要となる．また，コンポーネント通信においては，本論文では，コンポーネントの使用可否という比較的，キャッシュが無効になる可能性の高い情報を用いた手法となっており，CPU 使用率など，そこまで厳格でない情報とした場合に，より適用可能な範囲が広がる可能性がある．CPU 使用率やメモリ使用量などをキャッシュして，コンポーネント選択した場合について，その有効性をコンポーネント予約リクエストの送信間隔を変動させて，明らかにする必要がある．今後は，これらの課題を解決することで，さらに適用可能なアプリケーションが増加すると考える．

謝辞

本論文は，京都産業大学大学院先端情報学研究科先端情報学専攻博士後期過程の在学中における研究成果をまとめたものである．博士論文を提出するに当たって，多くの方にご指導，ご助力を頂き，深く感謝しております．特に，秋山豊和 教授には終始ご指導，アドバイス頂き，深く感謝しております．また，本論文を査読していただいた先生方にも感謝の意を表します．

参考文献

[1] Abe, K. and Teranishi, Y.: Suzaku: a Churn Resilient and Lookup-Efficient Key-Order Preserving Structured Overlay Network, *IEICE Trans. Communications* (2019).

[2] Amazon Web Services, Inc.: Amazon Kinesis Data Streams (online), available from https://aws.amazon.com/kinesis/data-streams/ (accessed 2020-10-01).

[3] Amazon Web Services, Inc.: Amazon web service (online), available from https://aws.amazon.com/ (accessed 2020-10-01).

[4] Apache Flink (online), available from https://flink.apache.org/ (accessed 2020-10-01).

[5] Ascigil, O., Phan, T. K., Tasiopoulos, A. G., Sourlas, V., Psaras, I. and Pavlou, G.: On uncoor-

dinated service placement in edge-clouds, *IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*, pp.41−48, (2017).

[6] Bahreini, T. and Grosu, D.: Efficient Placement of Multi-Component Applications in Edge Computing Systems, *Proc. 2nd ACM/IEEE Symposium on Edge Computing* (2017).

[7] Bernstein, D.: Containers and Cloud: From LXC to Docker to Kubernetes, *IEEE Cloud Comput.,* Vol.1, No.3, pp.81−84 (2014).

[8] Blackstock, M. and Lea, L.: Toward a Distributed Data Flow Platform for the Web of Things (Distributed Node-RED), *5th International Workshop on Web of Things,* pp.3439 (2014).

[9] Bonomi, F., Milito, R., Zhu, J. and Addepalli, S.: Fog Computing and its Role in the Internet of Things, *Proc. 1st Edition MCC Workshop Mobile Cloud Comput.*, p.1316 (2012).

[10] Eclipse Mosquitto: An open source MQTT broker (online), available from https://mosquitto.org/ (accessed 2020-10-01).

[11] Eclipse Paho (online), available from https://www.eclipse.org/paho/ (accessed 2020-10-01).

[12] Elastic, Inc.: Elasticesarch (online), available from https://www.elastic.co/ (accessed 2020-10-01).

[13] etcd (online), available from https://etcd.io/ (accessed 2020-10-19).

[14] Flinn, J. and Satyanarayanan, M.: Energy-Aware Adaptation for Mobile Applications, *Proc. 17th ACM Symp. Operating Systems Principles*, pp.4863 (1999).

[15] Fluentd (online), available from https://www.fluentd.org/ (accessed 2020-10-01).

[16] Google, Inc.: CLOUD DATAFLOW (online), available from https://cloud.google.com/dataflow/ (accessed 2020-10-01).

[17] Google, Inc.: Google Cloud Anthos (online), available from https://cloud.google.com/anthos (accessed 2020-10-01).

[18] Google, Inc.: Google Cloud IoT Core - Cloud IoT Edge (online), available from https://cloud.google.com/iot-core/ (accessed 2020-10-01).

[19] Google, Inc.: Google Cloud Platform (online), available from https://cloud.google.com/ (accessed 2020-10-01).

[20] Google, Inc.: Knative (online), available from https://cloud.google.com/knative/ (accessed 2020-10-01).

[21] Ha, K. and Satyanarayanan, M.: OpenStack++ for Cloudlet Deployment, *School of Computer Science Carnegie Mellon University Pittsburgh* (2015).

[22] Harris, D., Naor, J. and Raz, D.: Latency Aware Placement in Multi-access Edge Computing, *IEEE Conference on Network Softwarization and Workshops* (2018).

[23] 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 (2015).

[24] Internet Initiative Japan Inc., IIJmio IoT サービス (オンライン), available from https://www.iijmio.jp/mit/spec.html (accessed 2020-10-19).

[25] Ishihara, S., Tanita, S. and Akiyama, T.: A Dataflow Application Deployment Strategy for Hierarchical Networks, *IEEE 43rd Annual Computer Software and Applications Conference* (2019).

[26] Ishihara, S. and Akiyama, T.: Towards a Dataflow Platform in a Hierarchical Network: A Proposal for a Dataflow Component Management Method, *Journal of Information Processing*, Vol.28, pp.599-610 (2020).

[27] Ishihara, S., Yasuda, K., Akiyama, T., Abe, K. and Teranishi, Y.: A Proposal of an Inter Dataflow Component Communication Method using Distributed MQTT Broker, *Multimedia, Distributed, Cooperative, and Mobile Symposium*, pp.1571−1580 (2019).

[28] Istio - Connect, secure, control, and observe services (online), available from https://istio.io/ (accessed 2020-10-01).

[29] MQTT (online), available from http://mqtt.org/ (accessed 2020-06-20).

[30] Mastroianni, C. and Papuzzo, G.: A self-organizing P2P framework for collective service discovery, *Journal of Network and Computer Applications*, Vol.39, pp.214−222 (2014).

[31] Microsoft, Inc.: Azure IoT Edge (online), available from https://azure.microsoft.com/en-us/services/iot-edge/ (accessed 2020-10-01).

[32] Microsoft, Inc.: Azure Stream Analytics (online), available from https://azure.microsoft.com/en-us/services/stream-analytics/ (accessed 2020-10-01).

[33] Microsoft, Inc.: Microsoft Azure (online), available from https://azure.microsoft.com/en-us/ (accessed 2020-10-01).

[34] Moquette: Java MQTT lightweight broker (online), available from https://github.com/andsel/moquette (accessed 2020-10-01).

[35] NTT DOCOMO, Inc.: システムとしての信頼性向上（オンライン）, available from https://www.nttdocomo.co.jp/corporate/csr/disaster/reliable_system/index.html (accessed 2020-10-01).

[36] Nakashima, H., Arai, I. and Fujikawa, K.: Passenger Counter Based on Random Forest Regressor Using Drive Recorder and Sensors in Buses, *IEEE International Conference on Pervasive Computing and Communications Workshops*(*PerCom Workshops*), (2018).

[37] Node-RED (online), available from https://nodered.org/ (accessed 2020-10-01).

[38] OpenPose: Real-time multi-person keypoint detection library for body, face, hands, and foot estimation (online), available from https://github.com/CMU-Perceptual-Computing-Lab/openpose (accessed 2020-10-01).

[39] PIQT - distributed pub/sub broker (online), available from http://www.piqt.org/ (accessed 2020-10-01).

[40] Rancher Labs, Inc.: Rancher: Enterprise Kubernetes Management (online), available from https://rancher.com/ (accessed 2020-10-01).

[41] Ren, J., Yu, G., He, Y. and Li, G. Y.: Collaborative cloud and edge computing for latency minimization, *IEEE Trans. Vehicular Technology* (2019).

[42] Satyanarayanan, M., Bahl, V., Caceres, R. and Davies, N.: The Case for VM-based Cloudlets in Mobile Computing, *IEEE Pervasive Comput.,* Vol.8, No.4, pp.14−23 (2009).

[43] Satyanarayanan, M.: The Emergence of Edge Computing, *IEEE Computer*, Vol.50, No.1, pp.30−39 (2017).

[44] Schulz, P., Matthe, M., Klessig, H., Simsek, M., Fettweis, G., Ansari, J., Ashraf Ali, S., Almeroth, B., Voigt, J., Riedel, I., Puschmann, A., Mitschele-Thiel, A., Muller, M., Elste, T. and Windisch, M.: Latency critical IoT applications in 5G: Perspective on the design of radio interface and network architecture, *Proc. IEEE Communications Magazine*, Vol.55, No.2, pp.70−78 (2017).

[45] Schutt, T., Schintke, F. and Reinefeld, A.: Range queries On structured overlay networks, *Computer Communications*, Vol.31, No.2, pp.280−291 (2008).

[46] Shen, H., Li, Z. and Zhu, Y.: PIRD: P2P-based intelligent resource discovery in Internet-based distributed systems, *Proc. IEEE ICDCS*, pp.858−865 (2008).

[47] SoftBank, Corp.: NTT 西日本の POP の所在地 (オンライン), available from http://www.eaccess.net/cgi-bin/servicearea_ex.cgi?address=28 (accessed 2020-10-01).

[48] Teranishi, Y., Banno, R. and Akiyama, T.: Scalable and Locality-Aware Distributed Topic-Based Pub/Sub Messaging for IoT, *Proc. IEEE Globecom,* pp.1−7 (2015).

[49] Teranishi, Y., Kimata, T., Yamanaka, H., Kawai, E. and Harai, H.: Dynamic Data Flow Processing in Edge Computing Environments, *IEEE 41st Annual Computer Software and Applications Conference* (2017).

[50] The Linux Foundation.: Kubernetes: Production-Grade Container Orchestration (online), available from https://kubernetes.io/ (accessed 2020-10-01).

[51] Virtual Kubelet (online), available from https://virtual-kubelet.io/ (accessed 2020-10-01).

[52] Wang, S., Zafer, M. and Leung, K.: Online Placement of Multi-Component Applications in Edge Computing Environments, *IEEE Access,* Vol.5, pp.2514−2533 (2017).

[53] Wang, S., Zhang, X., Zhang, Y., Wang, L., Yang, J. and Wang, W.: A Survey on Mobile Edge Networks: Convergence of Computing, Caching and Communications, *IEEE Access,* Vol.5, pp.67576779 (2017).

[54] Zhang, W., Li, S., Liu, L., Jia, Z., Zhang, Y. and Raychaudhuri, D.: Hetero-Edge: Orchestration of Real-time Vision Applications on Heterogeneous Edge Clouds, *Proc. IEEE INFO-*

COM, pp.1270−1278 (2019).

[55] 安倍広多: 構造化オーバレイネットワークを用いた条件付きマルチキャストの提案，マルチメディア，分散協調とモバイルシンポジウム (2019).

[56] 谷田 智志: 階層化ネットワークを考慮した Dataflow Platform の設計とコンポーネント配置手法の検討，京都産業大学大学院先端情報学研究科修士論文 (未公刊) (2018).

[57] みなと観光バス株式会社，available from http://www.kobe-minato.co.jp/company.html (accessed 2020-10-19).

[58] 安田 和磨，分散型 MQTT Broker を用いたリソース情報の共有によるコンポーネント間通信の性能改善，京都産業大学大学院先端情報学研究科修士論文 (未公刊) (2020) (申請予定).

[59] 吉田 幹，寺西 裕一，春本 要，下條 真司: マルチオーバレイと分散エージェントの機構を統合化した P2P プラットフォーム PIAX，情報処理学会研究報告グループウェアとネットワークサービス (2006).

研究業績

論文誌

1. Shintaro Ishihara, Toyokazu Akiyama, Towards a Dataflow Platform in a Hierarchical Network: A Proposal for a Dataflow Component Management Method, *Journal of Information Processing*, Vol.28, pp.599-610 (2020).
2. 盛房 亮輔，坂野 遼平，安倍 広多，寺西 裕一，石原 真太郎，秋山 豊和, SDN を活用する Pub/Sub 基盤におけるオーバレイネットワーク管理方式の改善手法，情報処理学会論文誌，Vol.61，No.2，pp.326−338 (2020).

国際会議発表 (査読付)

1. Shintaro Ishihara, Satoshi Tanita, Toyokazu Akiyama, A Dataflow Application Deployment Strategy for Hierarchical Networks, *IEEE 43rd Annual Computer Software and Applications Conference* (2019).

国内会議発表

1. 鳥居 大輔，石原 真太郎，秋山 豊和，小林 和真，組織内ネットワークでの攻撃伝搬に対する既存のネットワーク機器を活用した監視手法の検討，マルチメディア，分散協調とモバイルシンポジウム (2020).
2. 安田 和磨，石原 真太郎，秋山 豊和，分散型 MQTT Broker を活用したコンポーネント選択手法の比較評価，インターネットと運用技術シンポジウム論文集 (2019).
3. 石原 真太郎，安田 和磨，秋山 豊和，安倍 広多，寺西 裕一，分散 MQTT Broker を活用した Dataflow コンポーネント間通信手法の提案，マルチメディア，分散協調とモバイルシンポジウム (2019).
4. 木村 智恒，石原 真太郎，秋山 豊和，土屋 樹一，環境音データによるイベント検知の取り組み，インターネット技術第 *163* 委員会第 *45* 回研究会 (2019).

ポスター発表

1. Shintaro Ishihara, Toyokazu Akiyama, A Tuning Method of a Monitoring System for Network Forensics in Cloud Environment, *IEEE 42nd Annual Computer Software and Applications Conference* (2018).

2. 石原 真太郎，秋山 豊和，クラウド/エッジを考慮した転送を可能とするメッセージング基盤の設計と実装，インターネット技術第 *163* 委員会第 *43* 回研究会 (2018).

第 2 章および第 3 章は，論文 “Shintaro Ishihara, Toyokazu Akiyama, Towards a Dataflow Platform in a Hierarchical Network: A Proposal for a Dataflow Component Management Method, *Journal of Information Processing*, Vol.28, pp.599-610 (2020).” を改訂したものである．

第 4 章は，論文 “Shintaro Ishihara, Kazuma Yasuda, Kota Abe, Yuuichi Teranishi, Toyokazu Akiyama, Comparative Evaluation of Dataflow Component Selection Methods in Distributed MQTT Broker Environment, *Journal of Information Processing*, (投稿予定).” を改訂したものである．