# 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

安田 和磨[1]　石原 真太郎[1]　秋山 豊和[1]

**概要**：近年，Internet of Things (IoT) デバイスの普及に伴い，多種多様になる IoT アプリケーションの要求に応えるために，エッジコンピューティングが注目されている．一方で，センサから得られるデータストリームをマイクロサービスを組み合わせて処理する Dataflow platform の開発も進められている．エッジコンピューティング環境での Dataflow platform にはネットワーク間の地理的距離とコンポーネント配備先のリソース状況を考慮した通信が求められる．これまで，P2P 構造化オーバレイを活用し，コンポーネントに地理的な情報と値を付与し，送信元で明示的に配送先を指定による通信及び負荷分散をする手法が提案されているが，配送先の決定にリソース状況は考えられていない．配送先を適切に決定するには一度各配備先のリソース状況を収集する必要がある．本研究では，地理的距離を考慮した配送が可能な分散型の MQTT Broker PIQT において，収集した値をオーバレイ上で集約値として扱い，それを用いた条件付き Multicast を拡張することで，リソース状況を考慮し適切なコンポーネント選択を可能とする Anycast 機構を導入する．集約値はオーバレイを維持するためのメッセージと一緒にリソース状況を収集するため，必要なメッセージ数を削減できるが，情報の鮮度は更新頻度に依存する．リソースの状態が更新されない最悪のケースの Anycast 機構と一度問い合わせる配送方法で要したメッセージの総ホップ数を比較調査した．また，集約値の有無により増加すると考えられる維持メッセージのペイロードサイズも同様に調査した．それらの結果を用いて，集約値を用いた Anycast により総ホップ数が抑えられることと，維持メッセージのサイズにより負荷が少なくなったことを明らかにした．

# A Comparative Evaluation of a Component Selection Method using Distributed MQTT Broker

## 1. はじめに

IoT 機器が次々生成するデータを階層構造をもつネットワークに配置された計算機群で処理する場合，ネットワーク間の地理的距離を考慮してデータ配送先を決定することが重要となる．本研究では，実行ノードが配備された地理的状況と負荷状況を考慮しながらデータの配送先を決定する技術を提案する．

近年，Internet of Things（IoT）デバイスが普及し，それらから生成されるデータは都市のスマート化へと活用されている．これまで，IoT デバイスからセンシングされたデータは従来のデータセンタにリソースが集約されたクラウドコンピューティング環境での収集，分析が一般的となっていたが，生成されるデータ量の増加に伴い，データの生成源の近傍に新たな計算リソースを設置するエッジコンピューティングの研究開発が盛んに進められている [1]．エッジコンピューティングでは，エッジでアプリケーションの一部を処理することでクラウドへの通信量の削減や，クラウドを経由せずにデバイスとエッジ間で処理を完結させることで通信遅延の低減が期待されている．また，これまで様々なネットワークアーキテクチャが提案されており，いずれもデータの生成源からの距離に応じて，2 階層から 3 階層のネットワーク階層で定義されている [2]．一方で，Azure Stream Analyitcs [3] や Google Cloud Dataflow [4] など，IoT デバイスから生成されるデータストリームをマイクロサービスを組み合わせて処理する Dataflow platform の研究開発が進められている．本研究では，文献 [5] と同様に，この組み合わせを Dataflow application，組み合わせる 1 つ 1 つのマイクロ

[1] 京都産業大学 先端情報学研究科

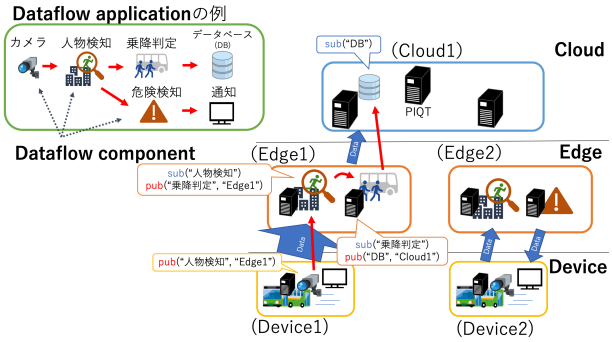


図 1　コンポーネントの配置及び通信

サービスを Dataflow component と呼ぶ．また，IoT デバイスが接続されるネットワークを “Device Network”, 基地局や局舎などに設置される小規模なネットワークを “Edge Network”, クラウドに展開される大規模なネットワークを “Cloud Network” と呼ぶ 3 階層の Dataflow platform を想定する．以降 “Device/Edge/Cloud” というネットワーク名を用いる．

図 1 では，我々の研究室で扱っているみなと観光バス株式会社のデータを用いた Dataflow application の例として，バス車内を映すドライブレコーダーで取得した動画データから人物を検知し，乗降の人数をデータベース (以降 DB) へ保存したり，人が倒れるような危険運転を検知し運転手に通知するアプリケーションを示している．定義された Dataflow application の要件に応じて，各コンポーネントを図 1 の “Cloud1”, “Edge1”, “Device1” のように，近傍の基地局や局舎をグループ化したネットワークのいずれかに配置し，コンポーネント間通信及びデータ処理を実行する．また，定義された各ネットワークは，“Cloud1”, “Cloud1/Edge1”, “Cloud1/Edge1/Device1”のようにネットワーク階層の関係を持つ．本稿では，以降階層関係として図 1 に示した例を用いて説明することとし，簡単のため階層関係は省略して各ネットワークの名称のみ記載する．文献 [5] では，Dataflow application の要件から各コンポーネントを適切なネットワークへ配置する手法が提案されており，本研究ではコンポーネント間の通信に着目する．

コンポーネント間の通信が満たすべき要件は (1) コンポーネントは異なるアプリケーション開発者によって開発されるため，多様な言語に対応した標準的な通信プロトコルを用いること，(2) 地理的に離れたネットワークで配置された同一のコンポーネント群から近傍のコンポーネントを選択するために，ネットワーク間の地理的距離を考慮した通信ができること，(3) 同一ネットワーク内に配置された同一のコンポーネント群から適したコンポーネントを選択するために，定義した Dataflow application の要件やリソース状況などの条件に応じて負荷分散ができること，の 3 つが考えられる．標準化された Pub/Sub プロトコルである MQTT を用いることで (1) の要件を満たすことができる．(2), (3) については，P2P オーバレイネットワークを活用することでネットワーク間の地理的距離を適切に扱いながら，リソース状況を考慮して分散配置された MQTT Broker 間のメッセージ配送を実現できる可能性がある．分散型 MQTT Broker PIQT [6] はこれらの機能をもつ MQTT Broker 実装の 1 つである．PIQT を用いる場合，図 1 に示すように人物検知や乗降判定などのコンポーネント名を topic とし，PIQT での接続及び通信を行うことになるが，PIQT では topic に基づいた Multicast しか考えられておらず，(3) が実現できない．(3) を実現する方法として，クラウドコンピューティングで用いられているロードバランサを活用する方法が考えられるが，本研究では各階層のネットワークが広域に分散することを想定しているため，均一なネットワーク環境を想定している既存のロードバランサで均等に負荷分散すると，性能低下をまねく可能性がある．一方で，下位層の情報をオーバレイ上で収集するアプローチとして，文献 [7] では，P2P オーバレイネットワークを維持するメンテナンス処理時に情報収集する機能と，収集した値に基づきメッセージのホップ数を増加させずにデータ配送が可能な条件付き Multicast が提案されている．本研究ではそれを Anycast に拡張し，さらに，PIQT が構築するオーバレイ上でメンテナンス処理時にリソース状況を収集することで，リソース選択に要する総ホップ数[*1]を削減しながら (3) を実現する．また，一度コンポーネントにリソース状況を問い合わせた上で適切なコンポーネントを選択する単純な配送方法と提案する Anycast 機構のそれぞれに要する総ホップ数とメンテナンス処理時のメッセージサイズの比較調査を行う．

以下，2 章でコンポーネントの選択手法について述べる．3 章ではリソース状況を考慮した単純な配送方法を述べ，4 章では提案する配送手法について述べる．5 章では評価について，6 章でまとめと今後の課題を述べる．

## 2. コンポーネントの選択手法

Dataflow application では，映像を用いた対象の動作検出などの前後のデータ比較が求められる場合，同一のコンポーネントで続けて解析することが望ましい．この場合，別途リクエストメッセージによって，処理に必要なコンポーネントを選出し，予約した上でデータの配送を行う．また，Dataflow application 終了時にはコンポーネントを解放する．以降ではこの実行方法を「予約実行」と呼ぶ．一方，前後のデータが独立な場合，データ配送時に活用するコンポーネントを決定することで負荷分散の柔軟性を高める．これを「非予約実行」と呼ぶ．図 1 の例では前後のデータ比較が必要なため予約実行となる．また，バスは運行により，対応する Device Network の近傍の Edge

[*1] 総ホップ数とは，オーバレイネットワークの論理リンクで転送されたメッセージ数の合計を指す．

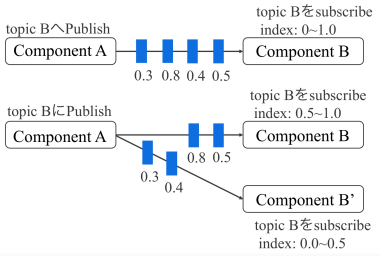


図 2　インデックス値を用いた配送

Network が変化する．このように動的に利用するコンポーネントの配置が変化する場合は，ストリームデータを途切れさせないために，変化に先立ってコンポーネントの予約を行い，スムーズな切り替えを行う必要がある．以下では，文献 [8], [9] の既存のコンポーネント選択手法の問題について紹介し，解決のアプローチを述べる．

### 2.1　エッジコンピューティング環境での動的な Dataflow platform

文献 [8] ではエッジコンピューティング環境において，分散型 Pub/Sub 基盤及び P2P 構造化オーバレイネットワークである Chord$^{\#}$ [10] を活用したコンポーネント間通信手法が提案されている．topic とネットワーク名の両方を用いてオーバレイネットワークを構築することで，地理的な距離を考慮したデータ配送と負荷分散を可能とする動的な Dataflow platform を実現している．図 2 で示すように Pub/Sub によりコンポーネント間を繋げており，subscribe 時に topic，ネットワーク名，インデックス値を指定する．インデックス値は 1 つのネットワークで 0.0〜1.0 の範囲を持っており，同じネットワーク及び topic を持つコンポーネントではインデックス値の範囲が分割される．publish 時に適切にネットワーク名とインデックス値を指定することにより，配送先を決定する．送信側でインデックス値をランダム等で選択することで，subscribe するコンポーネントへの負荷分散が可能であるが，リソース状況を考慮した適切なコンポーネントへの配送が実現できていなかった．

### 2.2　P2P を活用した期限付きワークフローの分散スケジューラ

文献 [9] では，大規模なシステムにおいて，P2P オーバレイネットワーク上で，実行ノードが活用可能な処理時間，メモリ，ディスク容量などのリソース状況を管理し，高速な応答時間と低いオーバーヘッドで期限付きのワークフローを実現する分散スケジューラが提案されている．ここでは，予約実行の場合のみを想定しており，オーバレイ上でリソース状況に基づき適切な実行ノードを探索し，予約している．また，ネットワーク間の地理的距離はオーバレイ上で IP アドレス順に並べることで考慮している．しかし，割り振る IP アドレスは一般的に本来の地理的な位置とは独立しており，また，マルチホームネットワークなどにより，IP アドレスを集約可能な形に割当することは困難であるため，単純に IP アドレスのみでネットワーク間の地理的距離を表現することは困難である．

### 2.3　解決のアプローチ

2.1 節の課題は，適切に配送先のコンポーネントを決定するには一度配送先のリソース状況を問い合わせた上でインデックス値を決定することで解決できる．しかし，この方法では問い合わせメッセージが余分に必要となるため，メッセージ数が増加してしてしまう．

2.2 節の課題は，PIQT に付与されるネットワークの階層関係を用いることで，ネットワーク間の地理的な距離を表現できる．

本研究では，2.1 節のアプローチを PIQT で実現し，提案する手法との比較を行う．次章では，提案方法の前提となる PIQT について述べる．

## 3.　PIQT でのリソース状況を考慮した配送

本章では，2.3 節で述べた PIQT でのリソース状況を考慮した配送方法について述べる．PIQT での配送は PIAX[11] が提供する Suzaku[12] の機構を用いており，Suzaku は Chord$^{\#}$ を拡張した構造化オーバレイネットワークである．以下では，はじめに 3.1 節で Chord$^{\#}$ について述べ，3.2 節で Chord$^{\#}$ を拡張した Suzaku 及び配送機構について述べる．その後，3.3 節で PIQT でのリソース状況を考慮した配送方法について述べる．

### 3.1　Chord$^{\#}$

Chord$^{\#}$ は key 順序型構造化オーバレイの代表例である．各ノードは key を持っており，リング状に key を辞書順で整列した状態で維持される．各ノードは自ノードの key の次に大きな key を持つノードのポインタ（Successor），次に小さい key を持つノードのポインタ（Predecessor）を保持する．ただし，トポロジがリング状であるため，最大の key を持つノードの Successor には最小の key を持つノードのポインタを，最小の key を持つノードの Predecessor には最大の key を持つノードのポインタを保持する．その他に key の検索を効率化するために複数のノードへのポインタを持つ FingerTable($FT$) を保持する．$FT$ に該当するノードは下記の式で求められ，$FT[i]$ は $2^i$ 個先のノードのポインタを示す．

$$FT[i] = \begin{cases} Successor & (i = 0) \\ FT[i-1] & \\ \quad \rightarrow getFinger(i-1) & (i \neq 0) \end{cases} \quad (1)$$

式 1 により，$i = 0$ の場合，$FT[0]$ は Successor を指す．あるノードを $p$, ノード $p$ の $FT[i-1]$ が指すノードを $q$,

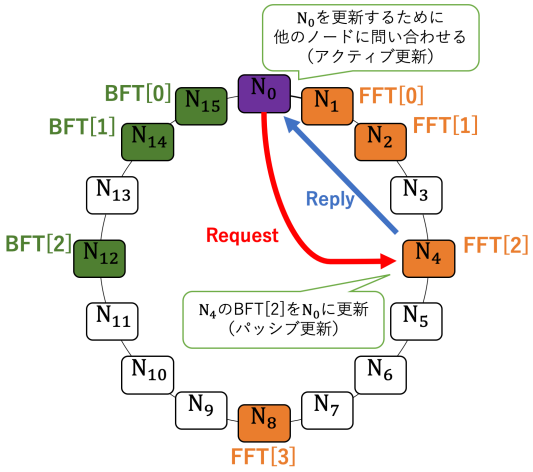


図 3　Suzaku の概要と更新

$getFinger(i-1)$ はあるノードの $FT[i-1]$ を取得する関数とすると，$i \neq 0$ の場合，ノード $p$ の $FT[i]$ には，ノード $p$ の $FT[i-1]$ が指すノード $q$ に Request を送信し，受信したノード $q$ は $getFinger(i-1)$ によってノード $q$ の $FT[i-1]$ をノード $p$ に Reply として返し，取得したポインタを指す．$i$ の最大はノードの総数が $n$ (ただし $n$ は自然数) の場合，$2^k < n$ ($n$=1 の場合，$k$ は 0 とする) を満たす $k$ の最大の整数となり，$FT$ は更新処理により鮮度が保たれる．各ノードの Successor と Predecessor は Chord[13] のスタビライズアルゴリズムによって，更新が行われる．$FT[i]$ $(i \neq 0)$ の更新は定期的に指定された時間間隔で $FT[0]$, $FT[1]$, ..., $FT[k-1]$ の順に Reqeust を送信し，$FT[1]$, $FT[2]$, ..., $FT[k]$ を更新する．また，検索の時に $FT$ を用いることで，$FT$ 収束時の最大ホップ数は $\lceil \log_2 n \rceil$ で収束する．

### 3.2　Suzaku

Chord$^\#$ を拡張した Suzaku では，$FT$ を双方向に保持し，各ノードが Chord$^\#$ と同様に時計回り方向にノードを指す Forward Finger Table($FFT$) の他に，反時計回り方向にノードを指す Backward Finger Table($BFT$) を持つ．$FFT$, $BFT$ の更新は，ノード挿入時，削除時に行われるが，それ以外にもノードの故障などで不整合が生じた場合，ポインタを修正するために定期的に行われる．ノード挿入時には $FFT[0]$, $BFT[0]$, $FFT[1]$, $BFT[1]$, $FFT[2]$, ..., $FFT[k]$, $BFT[k]$ の順に更新要求のための Request を送信し，更新が行われ，削除時には削除ノードを指し示すノードへと通知し，通知内容に含まれた代用のノードへとポインタを付け替えることで更新する．定期的な更新では，$FFT[0]$, $FFT[1]$, ..., $FFT[k]$ の順に指定された時間間隔で Request を送信し，更新する．Suzaku では，ノードを Request によって生存確認した後に更新するため，$FFT[k]$, $BFT[k]$ まで Request を送信する．次に，図 3 を用いて，$N_0$ の $FFT[2]$ の更新例を説明する．図 3 のノード数は 16

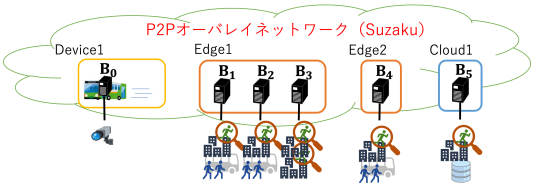


図 4　物理ネットワークの例

であり，各ノードは $N_i$ を key とする．更新には，アクティブ更新とパッシブ更新の 2 種類がある．アクティブ更新では $N_0$ は $FFT[2]$ に Request を送信し，Reply により $N_4$ を取得することで更新する．この Reply には $N_4$ の他に，$N_0$ が $FFT[3]$ を更新するために $N_8$ も入る．パッシブ更新では Request を受信した $N_4$ の $BFT[2]$ に $N_0$ で更新する．

Suzaku では，範囲検索により，センサデータの効率的な取得やメッセージの配送を実現する．メッセージの配送例として，$N_0$ から $N_2$, $N_3$, ...,$N_8$ の全てのノードにメッセージを配送する場合，検索する範囲として $[N_2, N_9)$ を生成し，範囲検索を行う．$[N_2, N_9)$ は $N_2 \leq$ key $< N_9$ を意味する．$N_0$ は生成した検索範囲を $FT$ が指すノードを元に分割して，各ノードに検索を委譲する．この場合，$N_2$ が担当する範囲は $[N_2, N_4)$，$N_4$ が担当する範囲は $[N_4, N_8)$，$N_8$ が担当する範囲は $[N_8, N_9)$ となる．$N_2$, $N_4$, $N_8$ では担当した範囲で再帰的に範囲検索が行われ，最終的に検索範囲内に該当する全てのノードにメッセージが行き渡る．配送結果は $N_0$ に直接返す方法と配送経路を元に集約しながら返す方法の 2 種類あり，本稿では $N_0$ への負荷分散を目的に後者で結果を返す．

### 3.3　PIQT

以下では，PIQT が形成する Suzaku，PIQT での配送方法を述べたのち，リソース状況を考慮した配送を述べる．

#### 3.3.1　PIQT における Suzaku

PIQT で形成される Suzaku のオーバレイネットワークでは，1 台の PIQT Broker(以下，Broker) 接続時と topic の subscribe 時にノードが生成される．Broker 接続時にはノードは Broker の ID を key として保持する．topic の subscribe 時にはノードは topic と subscribe 先の PIQT が保持するネットワーク名，$BrokerID$ を | で繋げた “topic|ネットワーク名 |$BrokerID$” を key として保持する．生成した key は辞書順に従い，整列に用いる属性の優先度は topic, ネットワーク名,BrokerID の順である．また，Broker に接続している複数のコンポーネントの topic が同じ場合，1 つのノードとしてまとめられ，異なる場合，新たにノードを生成する．そのため，1 つの Broker は複数の key を保持できる．

例えば，図 4 に示す各ネットワークを想定し，Dataflow application のコンポーネントとして人物検知，乗降判定，DB を各ネットワークに配置する．この場合，PIQT では，

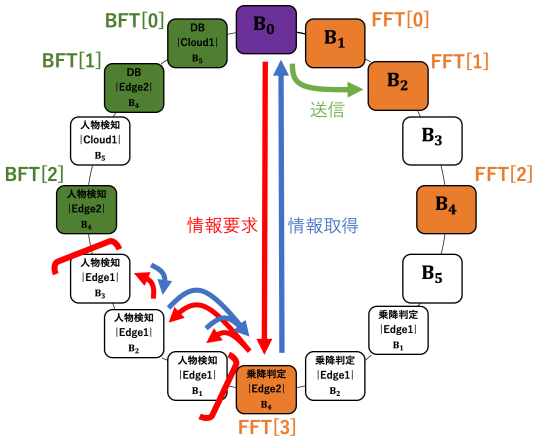


図 5　リソース状況を考慮した単純な配送方法

前段落で述べたことを踏まえて，図 5 に示すようなオーバレイネットワークが構築される．

PIQT では topic へのメッセージの配送を Suzaku の範囲検索により実現している．生成された $BrokerID$ の最小を $BrokerID_{min}$，最大を $BrokerID_{max}$ とすると，$B_0$ から Edge1 に存在する人物検知を持つ全てのノードにメッセージを配送する場合，[人物検知 $|Edge1|BrokerID_{min}$, 人物検知 $|Edge1|BrokerID_{max}$) が検索範囲として生成される．$B_0$ は，図 5 のように，はじめに乗降判定 $|Edge2|B_4$ に [人物検知 $|Edge1|BrokerID_{min}$，人物検知 $|Edge1|BrokerID_{max}$) を検索範囲とし，メッセージの配送を委譲する．乗降判定 $|Edge2|B_4$ は人物検知 $|Edge1|B_1$ が担当する範囲を [人物検知 $|Edge1|B_1$, 人物検知 $|Edge1|B_2$)，人物検知 $|Edge1|B_2$ が担当する範囲を [人物検知 $|Edge1|B_2$, 人物検知 $|Edge1|BrokerID_{max}$) とし，各ノードにメッセージの配送を委譲する．各ノードは担当した範囲で再帰的に範囲検索が行われ，人物検知 $|Edge1|$ を含むノードを持つ全ての Broker にメッセージを配送し，Broker が接続先のコンポーネントにデータを配送する．

### 3.3.2　PIQT におけるリソース状況を考慮した配送

本節で述べる配送方法は，文献 [8] の方法において 2.3 節で述べたコンポーネントの状態をモニタリングする方法と同様である．

リソース状況を考慮した適切なコンポーネントへの配送を実現するには，3.3.1 節で述べた範囲検索により各コンポーネント配備先のリソース状況を収集した上で適切なコンポーネントを保持する Broker に配送し，Broker からコンポーネントにデータを配送する必要がある．図 5 は $B_0$ が Edge1 の人物検知の中で適切なコンポーネントに配送する手順を示す．はじめに Edge1 の人物検知のリソース状況を収集するために，範囲検索で該当するノードにメッセージを送信する．その後，配送経路を元に集約しながら結果を取得する．$B_0$ は収集した情報を元に適切なノードを持つ Broker へとデータを配送し，Broker からコンポーネントにデータを配送する．

表 1　$B_0$ の $FT$

| $FT$ | 集約範囲 | 集約値 |
|---|---|---|
| $FFT[-1]$ | $[B_0, B_1)$ | null |
| $FFT[0]$ | $[B_1, B_2)$ | null |
| $FFT[1]$ | $[B_2, B_4)$ | null |
| $FFT[2]$ | $[B_4$, 乗降判定 $\|Edge2\|B_4)$ | 乗降判定 $\|Edge1\|$ : 2 |
| $FFT[3]$ | [乗降判定 $\|Edge2\|B_4$, $B_0$) | 乗降判定 $\|Edge2\|$ : 1<br>人物検知 $\|Edge1\|$ : 4<br>人物検知 $\|Edge2\|$ : 1<br>人物検知 $\|Cloud1\|$ : 1<br>DB$\|Edge2\|$ : 1<br>DB$\|Cloud1\|$ : 1 |
| $BFT[0]$ | [DB$\|Cloud1\|B_5$, $B_0$) | DB$\|Cloud1\|$ : 1 |
| $BFT[1]$ | [DB$\|Edge2\|B_4$, DB$\|Cloud1\|B_5$) | DB$\|Edge2\|$ : 1 |
| $BFT[2]$ | [人物検知 $\|Edge2\|B_4$, DB$\|Edge2\|B_4$) | 人物検知 $\|Edge2\|$ : 1<br>人物検知 $\|Cloud1\|$ : 1 |

## 4.　提案手法

本章では，提案する Anycast 機構について述べる．まず，はじめに 4.1 節で提案方式の概要について述べる．次に，4.2 節で集約値について述べ，4.3 節で提案する Anycast 機構について述べる．

### 4.1　提案方式の概要

提案方式では，2 章で述べたようにできるだけ短時間でコンポーネントを予約する必要があるので，配送に要するメッセージの総ホップ数を抑えるために，文献 [7] に基づきノードが保持する集約値を活用し，Anycast を実現する．集約値は，各ノードが保持するある値（value）を，定義した集約方法で予め一定の範囲のノードの value を $FT$ 上に集めた値のことである．文献 [7] では，集約値に基づき条件に合ったノードへと効率よく配送する条件付き Multicast が提案されている．提案する Anycast は条件付き Multicast を拡張し，各ノードで指定されたロジックに基づき，次の配送先を選択する．今回は，単純な選択ロジックとし，コンポーネントの予約までに必要な総ホップ数を計測するが，リソース状況を考慮した負荷分散をするためには，1 節の (3) の条件を満たすように収集する value，選択ロジックを今後考えていく必要がある．PIAX が提供する Suzaku には集約値と条件付き Multicast が実装されているため，以降での Suzaku は PIAX が提供する Suzaku を指す．ただし，文献 [7] は集約値を短時間で収束可能な $FT$ の更新方式 (連続的更新方式) も提案されているが，PIAX では提案された方式は実装されておらず，定期的に更新する方式 (離散的更新方式) しか実装されていない．以降ではすでに実装されている離散的更新方式を想定して議論を進める．

### 4.2　集約値

Suzaku には，$FFT[i]$ および $BFT[i]$ に Chord$^\#$ でのポインタに加えて，複数ノードを含む範囲である集約範囲と，集約範囲内に含まれるノードの値を定義された集約方法で

計算した集約値が含まれる．$FFT[i]$ には，$FFT[i]$ が指すノードから時計回りに $2^i$ ノードを含む集約範囲と集約値が登録される．$BFT[i]$ には，トポロジがリングの関係上，時計回りに $2^{i-1}$(ただし，$i=0$ の場合は 1) ノードを含む集約範囲と集約値が登録される．表 1 は図 5 の $B_0$ が保持する $FT$ を示し，保持する集約範囲と集約値の例である．例では，Broker に接続しているコンポーネント数を value とし，各ノードは topic| ネットワーク名 | の種類別に該当するコンポーネントの数を結びつけて保持する．集約値には，topic| ネットワーク名 | ごとに合計値を記録する．表 1 の $FFT[3]$ の場合，$FFT[3]$ の集約範囲に含まれる 8 ノードの内，人物検知 |Edge1| の 3 ノードが保持するコンポーネント数は 1, 1, 2 であり，その合計は 4 となる．集約値には他の 4 ノードが保持するコンポーネント数も同様に集約した値が $FFT[3]$ に登録される．また，$FFT[-1]$ は自ノードを指すとし，自ノードのみを含む集約範囲と自ノードの value が集約値として登録される．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，$B_0$ が $FFT[i]$ のアクティブ更新を行う時，$B_0$ は送信先のノードのパッシブ更新用に $FFT[i-1]$ までの範囲とその範囲内の各ノードから集約値を計算する．そして，その範囲と計算した集約値を Request に含めて，$FFT[i]$ に送信する．Request の受信ノードは自ノードから時計回りに $2^i$ ノードを含む集約範囲と集約した値を $B_0$ に Reply として送信する．また，受信ノードは Request に含まれたパッシブ更新用の情報を用いて，$BFT[i]$ のパッシブ更新を行う．ここで，$FFT[i]$ の集約値の更新には $FFT[i-1]$ までの集約値を用いるため，正しく $FFT[k]$ を更新するには，$FFT[0]$ から順番に更新する必要がある．$FT$ 更新の周期を $T$ 秒とすると，あるノードが $FT$ の集約値を更新するには $FFT[0]$, ..., $FFT[k]$ の 1 個の要素につき，最悪 $T$ 秒掛かるため，$(k+1)T$ 秒必要となる．さらに，自身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，$B_0$ の value に何らかの更新があった場合，$B_0$ の $FFT[2]$ の集約値を更新するには，$B_4$ の $FFT[1]$ までの集約値が更新されていなければいけない．この時，$B_4$ の更新がまだ $FFT[0]$ であった場合，$B_0$ の $FFT[2]$ の集約値を正しく更新するには $B_4$ の集約値の更新が順次実行され，$B_4$ の $FFT[1]$ までの集約値の更新が完了する必要がある．従って，$(k+1)T$ 秒では全ノードの集約値は $FFT$ の 1 個分しか更新されない．よって，$k+1$ 個の $FFT$ の全ての集約値は最悪 $(k+1)^2T$ 秒掛ければ正しく更新され，計算量は $O(T(\log n)^2)$ となる．

### 4.3 集約値を用いた Anycast 手法

提案する集約値を用いた Anycast のアルゴリズムをアルゴリズム 1 に示す．アルゴリズム 1 では，ノード $u$ が保持するアルゴリズムを示しており，ノード

**Algorithm 1** 集約値を用いた Anycast

```
1: ▷ u：実行するノード
2: ▷ r：Anycast を送信する範囲．ここでは [r_min, r_max) とする．
3: ▷ match：value を満たす条件を判定する関数
4: ▷ bestSelect：複数の FTE を 1 つに絞る関数
5: function u.condAnycast(r, match, bestSelect)
6:     ▷ matchE：r に集約する範囲が重なる かつ 条件に合う value を持つ FTE の集合
7:     matchE ← {e|(e ∈ ∀u.FTE) ∧ match(e.value) ∧ (e.range_min ∈ r ∨ e.range_max ∈ r)}
8:     if matchE ≠ null then
9:         ▷ selectE：matchE を 1 つに絞り込む
10:        selectE ← bestSelect(matchE)
11:        ▷ targetR：selectE に応じて，範囲の決定
12:        targetR ← shrink(selectE)
13:        if selectE.node = u then
14:            ▷ メッセージを受信
15:        else
16:            ▷ selectE のノードに委譲
17:            selectE.node.condAnycast(targetR, match, bestSelect)
18:        end if
19:    else
20:        ▷ 条件に合わない場合，r の最も近いノードに委譲
21:        p ← closestPrecedingNode(r_min)
22:        if p = u then
23:            ▷ 再送処理
24:        else
25:            p.condAnycast(r, match, bestSelect)
26:        end if
27:    end if
28: end function
29: ▷ e に応じて，範囲を決定する
30: function u.shrink(e)
31:    R ← [e.range_min, e.range_max)
32:    if e.range_min ∉ r ∧ e.range_max ∈ r then
33:        R ← [r_min, e.range_max)
34:    else if e.range_min ∈ r ∧ e.range_max ∉ r then
35:        R ← [e.range_min, r_max)
36:    end if
37: return R
38: end function
39: ▷ r に最も近いノードへ送る
40: function u.closestPrecedingNode(k)
41:    p ← {e.node|e ∈ ∀u.FTE ∧ e.node ∈ [u.key, k]} を満たす u から最も近いノード
42:    return p
43: end function
```

$u$ が $condAnycast(r, match, bestSelect)$ を実行する場合，$u.condAnycast(r, match, bestSelect)$ と表現する．各ノードは同様のアルゴリズムを保持する．$u$ は他のノードに範囲を狭めながらメッセージを送信し，$condAnycast$ を再帰的に実行する．ただし，Anycast を送信する範囲 $r$ は $[r_{min}, r_{max})$ とし，$r_{min}$ より $r_{max}$ の方が大きい key とする．ノード $u$ の $FT$ の各要素の集合を $FTE$，その 1 個の要素を $e$ とし，$e$ のノードを $e.node$, 集約する範囲を $e.range$, 集約値を $e.value$ とする．

$u.condAnycast(r, match, bestSelect)$ において，$matchE$ は $u$ が保持する $FTE$ から，検索する範囲 $r$ と $e.range$ が重なり，予め決められた $match()$ によって $value$ の条件を満たすものの集合である．$selectE$ は複数ある $matchE$ を $bestSelect()$ によって，1 つに絞り込む．絞り込むための選択ロジックは最小値, ランダムなど用途に応じて自由に変更可能である．最適な配送先のノードが選択されるかどうかは bestSelect() に依存しているが，前述のとおり本稿ではまずはホップ数削減効果の確認を目標とし，最適な負荷分散を実現する配送先選択ロジックは今後の課題とする．$targetR$ では，$shrink()$ により $selectE.range$ と $r$ が重なった範囲を作成し，$selectE.node$ に検索する範囲を $targetR$ とし，メッセージを委譲する．$selectE$ が $u$ の時，

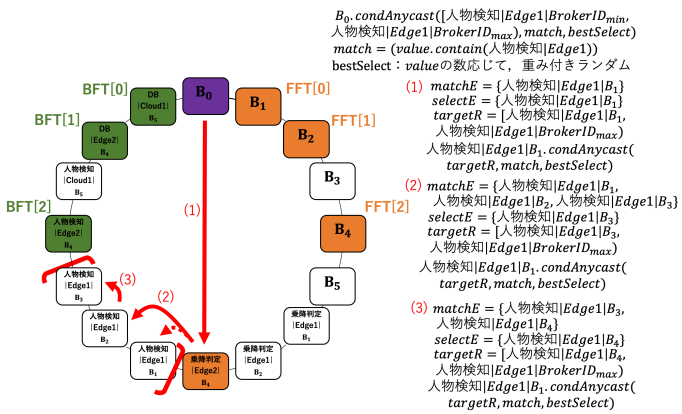


図 6　提案する Anycast 機構

$u$ がメッセージを受信する．

3.3.2 節で述べたメッセージ配送例と同様の環境で，Anycast を実行した配送例を図 6 に示す．配送例で活用する集約値は 4.2 節と同様とし，$match$ は人物検知 |$Edge1$ を保持しているかどうか，$bestSelect$ は各ノードの残り保持数をバランスさせるため，保持数に応じて重みを付けランダムに選択する．選択ロジックにより，保持数が多い $FTE$ が選択される可能性が高いため，配送例では図 6 のように配送される．

## 5. 集約値の更新状況に対する総ホップ数及び，Request/Reply のペイロードの比較調査

本章では，提案する Anycast は，文献 [14] で集約値の更新状況に応じての総ホップ数の傾向を調査した結果，$(k+1)T$ 秒間隔でのメッセージ配送に要するホップ数が安定することを明らかにしたが，妥当性が明らかにできていなかった．本評価では，3.3.2 節で述べた配送方法を用いて，要する総ホップ数を比較することで妥当性を明らかにする．また，2 章で述べたようにできるだけ短時間でコンポーネントを予約する必要があるため，今回は 1 つのコンポーネントの予約に要するメッセージの総ホップ数の比較調査を行う．さらに，集約値を用いる場合，$FT$ 更新のための Request/Reply のペイロードサイズが増加すると考えられるため，同様に調査する．

### 5.1　評価方法

本評価では，IoT デバイスとそれ以外に 1 つのコンポーネントから構成されるシンプルな Dataflow application を用いる．また，想定する Dataflow platform はコンポーネントを活用する場合のみとし，本来の value には，文献 [9] での実行ノードの処理時間や，メモリ，ディスク容量などのリソースが当てはまるが，今回は集約値の更新に掛かる伝搬遅延がメッセージの総ホップ数へ及ぼす影響を調査するために，シンプルにコンポーネントの予約状態を 1，空き状態を 0 で表現したものとする．コンポーネントの状態は 2 章で述べた別途リクエストメッセージによって，コンポーネントを予約し，0 に変更する．また，集約値は 4.2 節と同様の方法で集める．

メッセージの総ホップ数の比較調査では，3.3.2 節の配送方法と Anycast 機構の 2 つを用いる．また，集約値の更新状況によってはアルゴリズム 1 により，空き状態のコンポーネントを保持しない Broker に配送される可能性があり，その場合，受信した Broker から再送処理を行う．ここで，Anycast では，最悪のケースとの比較を考え，集約値の更新がほとんどされない連続的なメッセージ配送とする．物理環境は，1 つのネットワークに Broker を 10 台，9 種類のコンポーネントが一様に接続されている状態とし，1 台の Broker から 1 種類のコンポーネントにリクエストメッセージを送信する．1 計測はリクエストメッセージを全てのコンポーネントが予約されるまでとし，情報要求/取得メッセージ及び，メッセージ配送とそれに対する応答メッセージに要したノード間の総ホップ数を 50 回平均で計測した．

また，同様の評価環境で，集約値が完全に更新する最悪 $(k+1)^2T$ 秒内に発生した Request/Reply を集約値の有無でそれぞれ計測し，平均バイト数で比較する．

### 5.2　評価結果

図 7 はメッセージの総ホップ数比較の結果を示す．縦軸はメッセージの総ホップ数の平均値，横軸は 10 個のコンポーネントに順番に送信するため，何番目に送信したメッセージかを示し，3.3.2 節の集約値なしの配送方法，4.3 節の集約値ありの Anycast を示す．集約値なしの場合だと，10 個のコンポーネントに対して一度リソース状況を問い合わせる必要があり，約 20〜25 ホップ要することを確認した．一方で，集約値を活用する Anycast 機構では，残りコンポーネント数が少なくなるにつれて，総ホップ数が増加する傾向を示し，集約値を更新していないことと，残り少ないコンポーネントを探索するためにメッセージの再送処理が連続して発生したことが原因と考えられる．集約値の有無で比較すると，残りコンポーネントが 1 つの 10 番目のメッセージでは，約 32 ホップと集約値なしの総ホップ数を上回る結果となった．これはいくつか冗長のコンポーネントを配置することで，集約値の更新度の影響を抑えられると考えられる．

図 8 は Request/Reply のペイロードサイズ比較の結果を示す．縦軸はメッセージのペイロードのバイト数の平均を示し，横軸は Request と Reply を示し，凡例は先ほどと同様に集約値の有無を示す．結果から，Request の場合，約 0.1KB 増加し，Reply の場合，約 0.2KB の集約値分が増加することを確認した．

調査結果より，総ホップ数は集約値なしの配送方法より抑えられることを明らかにした．今回のペイロードサイズ

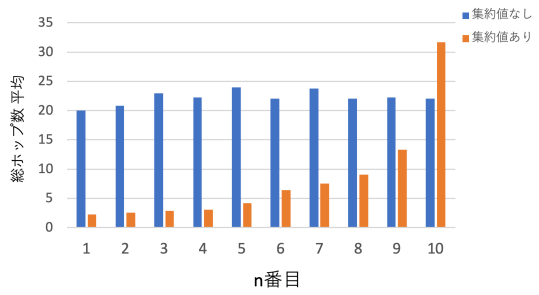


図 7　メッセージの総ホップ数比較

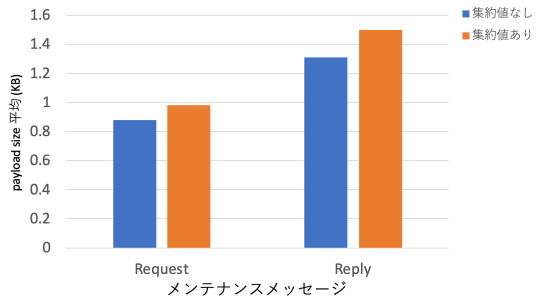


図 8　ペイロードサイズ比較

の調査はシンプルに 0/1 の値を用いた場合を示しており，ネットワークやコンポーネントの種類の増加に伴い集約値のサイズが大きくなると考えられる．また，ペイロードサイズが大きくなると，メッセージ処理にかかる負荷が増加すると考えられる．本稿では，大きな変更点である集約値の有無によるメッセージサイズの変動を調査したが，そのサイズは少なく，その変動に対するメッセージ処理にかかる負荷はそれほど上昇しないと考えられる．今後は，*FT* 更新だけでなく，全体でのメッセージ量の調査を行い，抑えられるホップ数に対する負荷を明らかにする．

## 6. まとめと今後の課題

本稿では，Dataflow platform のコンポーネント間通信において，ネットワーク間の地理的距離とコンポーネント配備先のリソース状況を考慮するためのルーティング手法を提案した．評価では，一度問い合わせる単純な配送方法と提案した Anycast 機構のメッセージの総ホップ数の比較と Request/Reply に要するペイロードサイズ比較調査を行い，コンポーネント選択に要するメッセージの総ホップ数を減らせることと，シンプルなリソース表現を用いた場合のペイロードサイズを明らかにした．今後の課題としては，提案した手法がユースケースの要件を満たしているかどうかの検討，文献 [8], [9] などの既存手法の調査を行いコンポーネントを予約しデータが流れるまでの時間などを比較，コンポーネントが活用可能な処理時間や，メモリ，ディスク容量などと value に用いる属性の検討と大規模化を含めた全体のメッセージ量の計測，そして，value に負荷情報を用いる場合，最適ではないが，処理可能なコンポーネントに配送可能な場合がある．その状況下でどれほどの精度で配送可能なのかの評価をする予定である．

**謝辞**　本研究開発の一部は JSPS 科研費 17K00143 の助成を受けたものである．本論文の作成にあたり，丁寧にご指摘を下さった大阪市立大学大学院 工学研究科の安倍 広多氏，情報通信研究機構の寺西 裕一氏に感謝する．

### 参考文献

[1] 飯田 勝吉：エッジコンピューティング研究開発の現状と今後の課題, IEEE Access, vol. 117, no. 187 ,pp. 25-30, Aug.2017.
[2] Klas, G.I.:Fog Computing and Mobile Edge Cloud Gain Momentum Open Fog Consortium ETSI MEC and Cloudlets, 2015 (online), 入手先 ⟨http://yucianga.info/?p=938⟩（参照 2019-11-06）.
[3] Microsoft, Inc.: Azure Stream Analyitcs (online), 入手先 ⟨https://azure.microsoft.com/ja-jp/services/stream-analytics/⟩（参照 2019-11-09）.
[4] Google, Inc.: Cloud Dataflow (online), 入手先 ⟨https://cloud.google.com/dataflow/?hl=ja⟩（参照 2019-11-06）.
[5] Ishihara, S., et al.: A Dataflow Application Deployment Strategy for Hierarchical Networks, IEEE Computer Society Signature Conference on Computers, Software and Applications, July COMPSAC2019.
[6] Teranishi, Y., et al.: Scalable and Locality - Aware Distributed Topic - based Pub/Sub Messaging for IoT, Proc. of IEEE GLOBECOM Dec.2015.
[7] 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, July DICOMO2019
[8] Teranishi, Y., et al.: Dynamic Data Flow Processing in Edge Computing Environments, IEEE 41st Annual Computer Software and Applications Conference, Jul.2017.
[9] Celaya, J., et al.: Dustributed Scheduler of Workflows with Deadlines in a P2P Desktop Grid, Proceedings of 18 IEEE Euromicro Conference on Parellel, Distributed, and Network-based Processing, 2010, pp. 69-73.
[10] Schutt, T., et al.: Range queries on structured overlay net- works, Computer Communications, vol.31, no.2, pp.280 - 291, 2008.
[11] 吉田幹ほか.：マルチオーバレイと分散エージェントの機構を統合した P2P プラットフォーム PIAX, 情報処理学会論文誌, Vol.49, No.1, pp. 402 - 413, Jan.2008
[12] Abe, K., et al.: Suzaku: a Churn Resilient and Lookup-Efficient Key-Order Preserving Structured Overlay Network, IEICE TRANSACTIONS on Communications, Mar.2019
[13] Stoica, I., et al.: Chord: A Scalable Peer-to-Peer Lookup Protocol for Internet Applications, IEEE/ACM Trans. on Net., Vol. 11, No. 1, pp. 17-32 (2003).
[14] 石原 真太郎ほか.：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, July DICOMO2019