広島大学大学院先進理工系科学研究科情報科学プログラム修士論文

MQTT 共有サブスクリプションにおける サブスクライバ群の状態を考慮した動的負荷分散手法

広島大学大学院先進理工系科学研究科
先進理工系科学専攻情報科学プログラム

M222408

轟木 皓平

指導教員 近堂 徹

広島大学

Master's Thesis
Informatics and Data Science Program, Graduate School of Advanced Science and Engineering,
Hiroshima University

A Dynamic Load Balancing Method for MQTT Shared Subscription Considering the State of a Group of Subscribers

Informatics and Data Science Program
Graduate School of Advanced Science and Engineering
Hiroshima University

M222408

Kohei Todoroki

Supervisor: Tohru Kondo

Hiroshima University

要旨

IoT デバイスを利用したネットワーク経由でのデータ活用が積極的に行われている．IoT デバイスが利用される事例は様々であるが，広域に分散配置された不安定なネットワーク上の大量のデバイスからのデータ収集や，低遅延での通信がシステムに求められることから，IoT 向け通信プロトコルの MQTT とエッジコンピューティングが注目を集めている．

この 2 つの技術の組み合わせは広域分散環境におけるデータ基盤の実現に適しており，MQTT をエッジでの利用に最適化したアーキテクチャの提案も行われている．特に 2019 年に策定された MQTT Version5.0 の仕様には，メッセージを複数のサブスクライバで分散処理する共有サブスクリプションが追加され，MQTT を用いたデータ基盤のスケーラビリティとアベイラビリティを向上させることが可能になった．一方，現状の共有サブスクリプションにおけるサブスクライバ選択手法は実装依存になっており，多くのベンダ製品や OSS では静的負荷分散手法のみが実装されている．静的負荷分散手法ではエッジのような利用できるリソースに制限がある環境において，他のワークロードの影響によるサブスクライバのデータ処理性能の変化に対応することができず，結果としてワークロード全体の処理遅延が生じるリスクがある．

本研究では，ブローカが接続しているサブスクライバ群の状態をリアルタイムに管理することで共有サブスクリプションにおける動的負荷分散を実現する手法を提案する．提案手法では MQTT の制御パケットを拡張し，サブスクライバのメッセージ処理状況をブローカに随時伝達することで，保留中のメッセージ数に基づいた動的負荷分散を実現する．

評価では提案手法は既存の静的負荷分散手法と比較してワークロード全体のレイテンシを 98% 削減できることを確認した．

Abstract

IoT devices are being actively used to utilize data via networks. IoT devices are used in a variety of applications, but systems are required to collect data from a large number of devices on unstable networks distributed over a wide area and to communicate with low latency. For this reason, MQTT, a communication protocol for the IoT, and edge computing have attracted much attention.

The combination of these two technologies is suitable for realizing data infrastructures in wide-area distributed environments, and architectures have been proposed that optimize MQTT for use at the edge. In particular, the MQTT Version 5.0 specification, which was developed in 2019, adds shared subscriptions, in which messages are distributed processing by multiple subscribers, thereby improving the scalability and availability of data infrastructure using MQTT. On the other hand, the current subscriber selection method for shared subscriptions is implementation-dependent, and many vendor products and OSS implement only static load balancing methods. Static load balancing methods cannot deal with changes in subscriber data processing performance due to the influence of other workloads in an environment with limited available resources, such as the edge, and as a result, there is a risk of processing delays for the entire workload.

In this study, we propose a method to achieve dynamic load balancing for shared subscriptions by managing the status of a group of subscribers connected to the broker in real time. The proposed method extends MQTT control packets to transmit the status of subscriber message processing to the broker at any time. The broker performs dynamic load balancing based on the number of pending messages at the subscriber.

The evaluation confirmed that the proposed method reduces the overall workload latency by 98% compared to existing static load balancing methods.

目次

第 1 章　はじめに

1.1　研究背景と目的

IoT デバイスを利用したネットワーク経由でのデータ活用が積極的に行われている．IoT デバイスが利用される事例は様々であるが，工場や物流でのセンサデバイスによる業務自動化や故障検知を目的とした「産業用途」や，自動運転の実現が期待されている「自動車・宇宙航空」での活用が予想されている [1]．これらの事例では広域に分散配置された不安定なネットワーク上の大量のデバイスからのデータ収集や，低遅延での通信がシステムに求められることから，IoT 向け通信プロトコルの MQTT[2] とエッジコンピューティングが注目を集めている．

MQTT は OASIS[3] により標準化された Publish/Subscribe モデル [4] の通信プロトコルである．MQTT は多対多の通信を容易に実現できる Publish/Subscribe モデルによって，大量のデバイスからのデータを収集・処理するシステムを容易に構築可能であることや，指定した QoS による送信保証によって，ネットワーク品質が変動する環境でもデータの損失を防止可能であることから，IoT システムに適したプロトコルになっている．エッジコンピューティングはデータが生成されるデバイスに近い場所にデータを処理するリソースを配置する手法である．エッジコンピューティングを活用することで，低遅延での通信やネットワーク帯域幅の効率的な利用を実現することができる．

この 2 つの技術の組み合わせは広域分散環境におけるデータ基盤の実現に適しており，MQTT をエッジでの利用に最適化したアーキテクチャの提案も行われている [5][6]．特に MQTT Version5.0[7] で追加されたメッセージをサブスクライバで分散処理する共有サブスクリプションと，エッジノードの管理を自動化するオーケストレーションツール [8][9] を利用することで，デバイスやデータ量の増加にあわせてデータ処理を行うサブスクライバを自動でスケールするデータ基盤を仮想化基盤上に構築可能になっている．しかし，現在の共有サブスクリプションにおけるサブスクライバの選択手法は仕様として定義されておらず，既存の実装ではラウンドロビンやランダムなどの静的負荷分散手法の実装に留まっており [10][11]，サブスクライバの状態を考慮した動的負荷分散は実装されていない．仮想化基盤では複数のワークロードの実行が想定されており，静的負荷分散手法では他のワークロードの影響によるサブスクライバのデータ処理性能の変化に対応することができず，結果としてワークロード全体の処理遅延が生じるリスクがある．

そこで，本研究ではブローカが接続しているサブスクライバ群の状態をリアルタイムに管理する共有サブスクリプションにおける動的負荷分散手法を提案する．提案手法では MQTT の制御パケットを拡張し，サブスクライバのメッセージ処理状況をブローカに随時伝達することで，保留中のメッセージ数に基づいた動的負荷分散を実現する．実装および評価実験を通して，提案手法は複数のワークロードが動作し，他のワークロードの影響によるリソース変動や処理性能の変化が発生する可能性がある実行環境に対して，既存の静的負荷分散手法よりも堅牢であることを示す．

1.2　論文の構成

本論文 6 章から構成されている．2 章では MQTT や負荷分散手法などの要素技術について説明する．3 章では関連研究として共有サブスクリプションや動的負荷分散手法に関する研究について述べる．4 章では本研究で提案する動的負荷分散を実現するための設計と実装について説明する．5 章では提案手法と既存の静的負荷分散手法で比較評価を行い，提案手法の性能評価について述べる．最後に 6 章で本研究のまとめと今後の課題を述べる．

第 2 章　要素技術

2.1　メッセージングシステム

メッセージングシステムとは送信側のアプリと受信側のアプリがメッセージ指向ミドルウェア（MOM）を介してデータ送信を行う非同期通信が可能なアーキテクチャである．一般的な同期通信とメッセージングシステムによる非同期通信の比較を図 1 に示す．同期通信では，送信側と受信側は直接通信を行う．そのため，通信開始時に送信側と受信側は必ず動作している必要がある．また，送信側が複数の受信側に対して同一メッセージを送信する 1 対多の通信を行う場合，送信側は全ての受信側と相互接続する必要がある．これは 1 台の受信側の障害が送信側に影響を及ぼすため，可用性が低下する．一方で，メッセージングシステムの非同期通信の場合，送信側と受信側は MOM に接続し，送信側のデータは MOM を介して受信側に送信される．送信側は通信開始時に受信側の状態を認識する必要がなく，受信側も送信側が存在するかを認識する必要がない．送信側と受信側が疎結合になっているため，実装の変更を柔軟に行うことが可能になっている．また，受信側や送信側が増減しても実装を変更する必要がないため，スケーラビリティにも優れている．

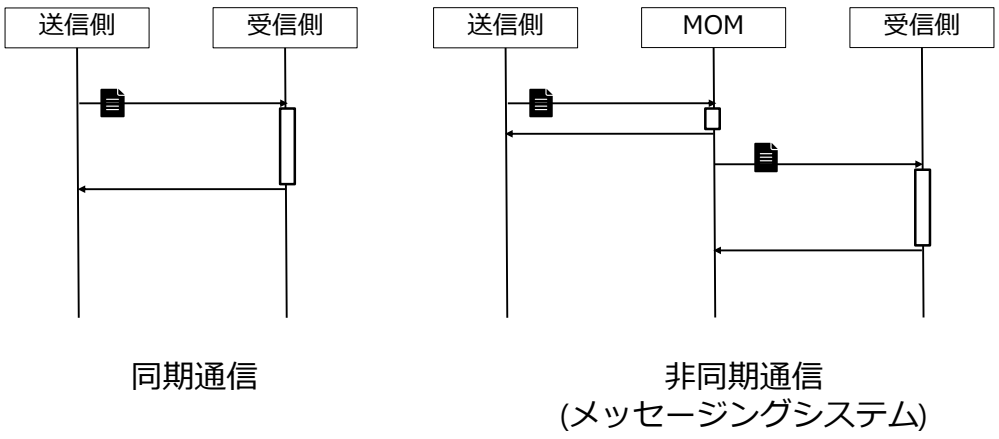

図 1　同期通信とメッセージングシステムの非同期通信の比較

2.2　Publish/Subscribe モデル

2.2.1　Publish/Subscribe モデルの概要

Publish/Subscribe モデルはメッセージングシステムの 1 種であり，データを生成して送信するパブリッシャ，データの分配を行うブローカ，データを処理するサブスクライバの 3 つのコンポーネントで構成される．Publish/Subscribe モデルの構成を図 2 に示す．Publish/Subscribe モデルではトピックと呼ばれる論理チャネルを介してデータの送信が行われる．

パブリッシャからサブスクライバにデータが送信される流れを説明する．まず，サブスクライバは受信したいトピックをブローカに通知する．この処理をサブスクライブと呼ぶ．この状態でパブリッシャが生成したデータをトピックを指定してブローカに送信する．この処理をパブリッシュと呼ぶ．ブローカはトピックにパブリッシュされたデータをトピックをサブスクライブしている全てのサブスクライバに複製して送信する．ブローカのトピックを介したデータ送信によって 1 対 1，1 対多の通信を容易に実現することが可能になっている．また，パブリッシャやサブスクライバの増減も既存の構成を変更することなく容易に実現できる．

Publish/Subscribe モデルには push 型と pull 型が存在し，それぞれにメリットとデメリットが存在する．次項からそれぞれの特徴を説明し，想定される用途を述べる．

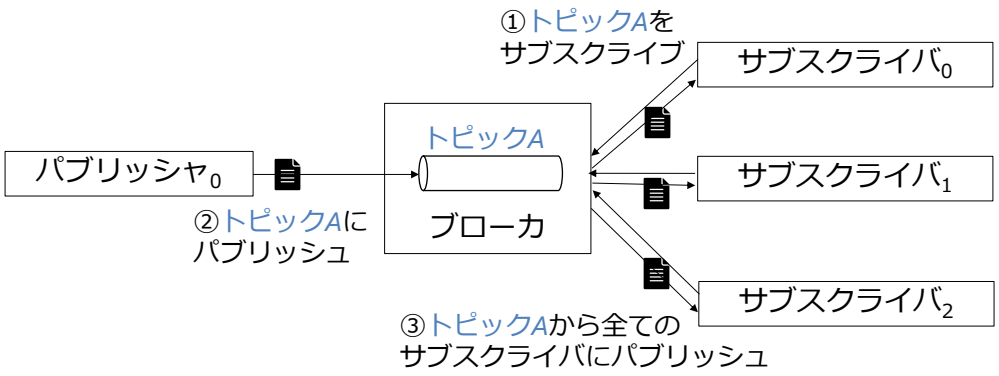

図 2　Publish/Subscribe モデル

2.2.2　push 型の Publish/Subscribe モデル

push 型の Publish/Subscribe モデルは，トピックにパブリッシュされた時点でサブスクライブしているサブスクライバにデータを送信する通信モデルである．push 型の Publish/Subscribe モデルの動作を図 3 に示す．トピックのデータはブローカからサブスクライバに即座に送信され，送信後にブローカから削除される．MQTT などで採用されている．

push 型のメリットは，パブリッシャで発生したイベントを即座にサブスクライバに通知することが可能な点である．そのため，push 型を採用することでリアルタイム性の高いシステムを構築することができる．

一方で push 型のデメリットは，パブリッシャの生成レートがサブスクライバの処理レートを上回ると処理が追い付かず，サブスクライバがダウンする可能性がある点である．この問題を回避するためには，パブリッシャの生成レートを抑制する実装を行うことや，サブスクライバに十分なリソースを割り当てる必要がある．

push 型の用途として，スマートフォンやアプリケーションのユーザに対する Push 通知や，IoT デバイスからの測定データに対して即座に反映を必要とするリアルタイム性の高い IoT システムでの利用が考えられる．

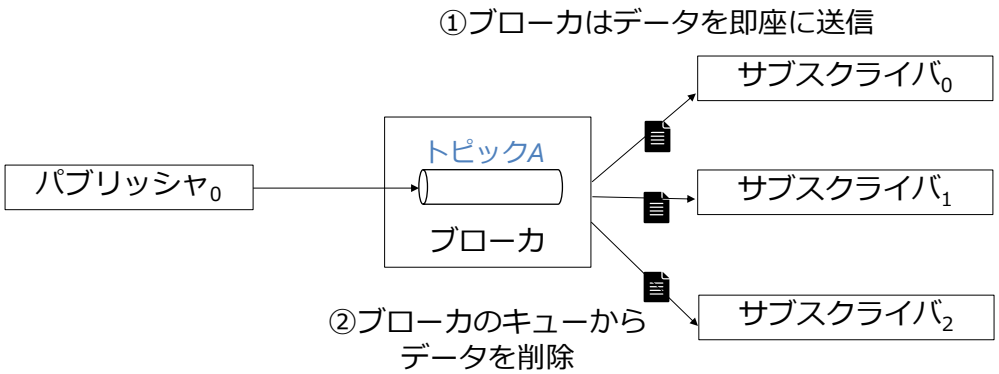

図 3　Push 型の Publish/Subscribe モデル

2.2.3　pull 型の Publish/Subscribe モデル

pull 型の Publish/Subscribe モデルは，トピックにパブリッシュされたデータをブローカで蓄積し，サブスクライバの要求に応じてデータを送信する通信モデルである．pull 型の Publish/Subscribe モデルの動作を図 4 に示す．トピックのデータはサブスクライバに送信されても削除されず，ブローカで一定期間保存される．Apache Kafka[12] などで採用されている．

pull 型のメリットは，サブスクライバが任意のタイミングでブローカにデータ要求を行うことができる点である．

そのため，パブリッシャでサブスクライバの性能に応じた生成レートの抑制などの実装も必要とせず，処理性能が異なる複数のシステムとの連携を容易に実現することができる．

一方で pull 型のデメリットは，サブスクライバの要求頻度によってシステムの特性が異なる点である．要求頻度を高くし過ぎるとブローカに大量のポーリングが発生し，ブローカの処理性能が低下する可能性がある．対して，要求頻度を低くし過ぎるとブローカでのデータの滞留時間が長くなり，データが生成されてから処理するまでのレイテンシが増加する．

pull 型の用途として，リアルタイム性が要求されないログデータの収集基盤や，一時的なアクセスの増加が発生するシステムでのバッファとしての利用が考えられる．

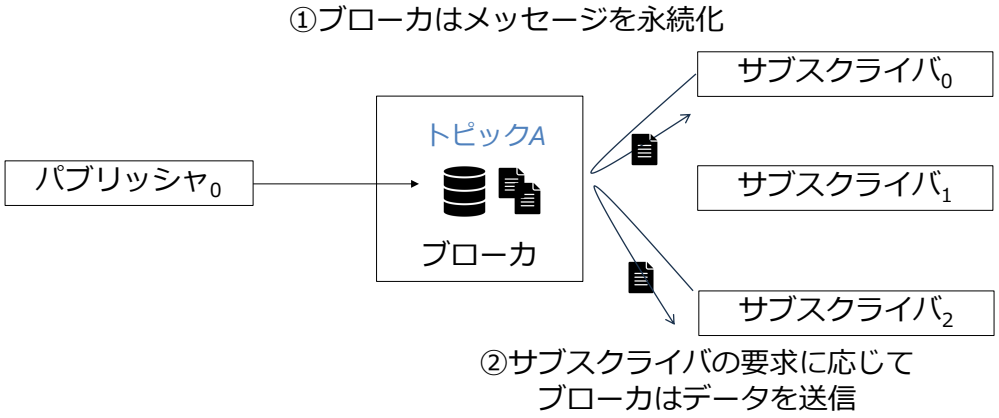

図 4　Pull 型の Publish/Subscribe モデル

2.3　MQTT

2.3.1　MQTT の概要

MQTT は push 型の Publish/Subscribe モデルを採用したアプリケーション層の通信プロトコルである．固定ヘッダが最小 2 バイトであるため通信のオーバーヘッドが小さいことや，QoS を指定することで不安定なネットワークでのデータの損失を防止できることから，リソース制限のある M2M や IoT での利用に適したプロトコルになっている．

MQTT では制御パケットと呼ばれるプロトコルで定義されたパケットを交換することでデータフローを実現している．制御パケットの構造を図 5 に示す．制御パケットは固定ヘッダ，可変ヘッダ，ペイロードの 3 つの領域で構成されている．制御パケットには複数の種類が存在し，固定ヘッダに含まれるパケットタイプで識別される．固定ヘッダのフォーマットは全てのタイプで同一であり，可変ヘッダとペイロードのフォーマットはタイプによって異なる．次節で制御パケットの詳細について説明する．

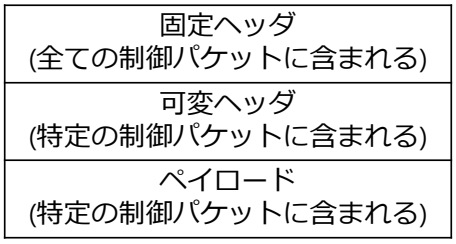

図 5　MQTT の制御パケット

2.3.2 制御パケット

2.3.2.1 固定ヘッダ

制御パケットにおける固定ヘッダのフォーマットを図 6 に示す．固定ヘッダは最小 2 バイトであり，パケット長によってサイズが 2 から 5 バイトまで変動する．固定ヘッダのサイズが HTTP などの既存のアプリケーションプロトコルと比較すると小さいため，センサデータなどのショートパケットの送信で生じるオーバーヘッドを軽減することができる．

固定ヘッダに含まれるパケットタイプは制御パケットの識別に利用される．制御パケットの一覧と対応するパケットタイプは表 1 の通りである．フラグは PUBLISH のみで意味を有し，それ以外の制御パケットでは決められた値が設定される．フラグの詳細は 2.3.3 項で説明する．パケット長には固定ヘッダを除いたパケットの残りの長さが設定される．パケット長は可変バイト数（Variable Byte Integer）で表現されており，最大 4 バイトまで拡張できる．

<table>
<tr><td></td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
<tr><td rowspan="2">1 Byte</td><td colspan="4" rowspan="2">パケットタイプ</td><td colspan="4">フラグ</td></tr>
<tr><td>DUP</td><td colspan="2">QoS</td><td>RETAIN</td></tr>
<tr><td>2(~5) Byte</td><td colspan="8">パケット長</td></tr>
</table>

図 6　制御パケットの固定ヘッダ

表 1　制御パケットタイプ

名前	パケットタイプの値	流れの方向
CONNECT	1	クライアント → ブローカ
CONNACK	2	ブローカ → クライアント
PUBLISH	3	クライアント → ブローカ or ブローカ → クライアント
PUBACK	4	クライアント → ブローカ or ブローカ → クライアント
PUBREC	5	クライアント → ブローカ or ブローカ → クライアント
PUBREL	6	クライアント → ブローカ or ブローカ → クライアント
PUBCOMP	7	クライアント → ブローカ or ブローカ → クライアント
SUBSCRIBE	8	クライアント → ブローカ
SUBACK	9	ブローカ → クライアント
UNSUBCRIBE	10	クライアント → ブローカ
UNSUBACK	11	ブローカ → クライアント
PINGREQ	12	クライアント → ブローカ
PINGRESP	13	ブローカ → クライアント
DISCONNECT	14	クライアント → ブローカ
AUTH	15	クライアント → ブローカ or ブローカ → クライアント

2.3.2.2 可変ヘッダ

制御パケットにおける可変ヘッダのフォーマットを図 7 に示す．可変ヘッダはパケットタイプ別に決められたフォーマットを格納する領域と追加のメタデータを格納するプロパティ領域で構成される．プロパティは MQTT Version5.0 で追加された領域であり，制御パケットにユーザーが任意の情報を追加することで，MQTT の機能を拡張可能にすることを目的としている．プロパティの一覧を表 2 に示す．制御パケットによって許可されているプロパ

ティは異なっている．

プロパティの内，最も拡張性の高い User Property を紹介する．User Property はキーバリュー形式の任意の文字列を設定可能なプロパティであり，多くの制御パケットで許可されている．例えばキーに client，バリューに client-1 を，キーに data，バリューに json を設定した User Property は図 8 に示すフォーマットで格納される．User Property の用途として，特定のサブスクライバのみに送信するルーティングや，メッセージ発信元の追加による広域分散環境でのトレーサビリティの向上が挙げられる．

<table>
<tr><td></td><td></td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
<tr><td rowspan="5">3(~6) Byte
~</td><td>制御パケット別の領域</td><td colspan="8">…</td></tr>
<tr><td rowspan="4">プロパティ領域</td><td colspan="8">プロパティ長</td></tr>
<tr><td colspan="8">プロパティID</td></tr>
<tr><td colspan="8">プロパティのデータ</td></tr>
<tr><td colspan="8">…</td></tr>
</table>

図 7　制御パケットの可変ヘッダ

```
Properties
   Total Length: 32
   ID: User Property (0x26)
   Key Length: 6
   Key: client
   Value Length: 8
   Value: client-1
   ID: User Property (0x26)
   Key Length: 4
   Key: data
   Value Length: 4
   Value: json
```

図 8　User Property の例

表 2　プロパティ一覧

プロパティ ID	名前	許可される制御パケット
1(0x01)	Payload Format Indicator	PUBLISH, Will Properties
2(0x02)	Message Expiry Interval	PUBLISH, Will Properties
3(0x03)	Content Type	PUBLISH, Will Properties
8(0x08)	Response Topic	PUBLISH, Will Properties
9(0x09)	Correlation Data	PUBLISH, Will Properties
11(0x0B)	Subscription Identifier	PUBLISH, SUBSCRIBE
17(0x11)	Session Expiry Interval	CONNECT, CONNACK, DISCONNECT
18(0x12)	Assigned Client Identifier	CONNACK
19(0x13)	Server Keep Alive	CONNACK
21(0x15)	Authentication Method	CONNECT, CONNACK, AUTH
22(0x16)	Authentication Data	CONNECT, CONNACK, AUTH
23(0x17)	Request Problem Information	CONNECT
24(0x18)	Will Delay Interval	Will Properties
25(0x19)	Request Response Information	CONNECT
26(0x1A)	Response Information	CONNACK
28(0x1C)	Server Reference	CONNACK, DISCONNECT
31(0x1F)	Reason String	CONNACK, PUBACK, PUBREC, PUBREL, PUBCOMP, SUBACK, UNSUBACK, DISCONNECT, AUTH
33(0x21)	Receive Maximum	CONNECT, CONNACK
34(0x22)	Topic Alias Maximum	CONNECT, CONNACK
35(0x23)	Topic Alias	PUBLISH
36(0x24)	Maximum QoS	CONNACK
37(0x25)	Retain Available	CONNACK
38(0x26)	User Property	CONNECT, CONNACK, PUBLISH, Will Properties, PUBACK, PUBREC, PUBREL, PUBCOMP, SUBSCRIBE, SUBACK, UNSUBSCRIBE, UNSUBACK, DISCONNECT, AUTH
39(0x27)	Maximum Packet Size	CONNECT, CONNACK
40(0x28)	Wildcard Subscription Available	CONNACK
41(0x29)	Subscription Identifier Available	CONNACK
42(0x2A)	Shared Subscription Available	CONNACK

2.3.2.3　ペイロード

ペイロードは一部の制御パケットの最後に含まれ，パケットタイプによってフォーマットが異なる．代表的な用途として，PUBLISH のペイロードにパブリッシャが生成したアプリケーションのデータを格納し，ブローカを介してサブスクライバに送信するのに利用される．

2.3.3 MQTT のデータフロー

MQTT の制御パケットによってパブリッシャが生成したデータがサブスクライバに送信されるまでのデータフローを図 9 を用いて説明する．MQTT のデータフローでは大きく 4 つの処理がある．

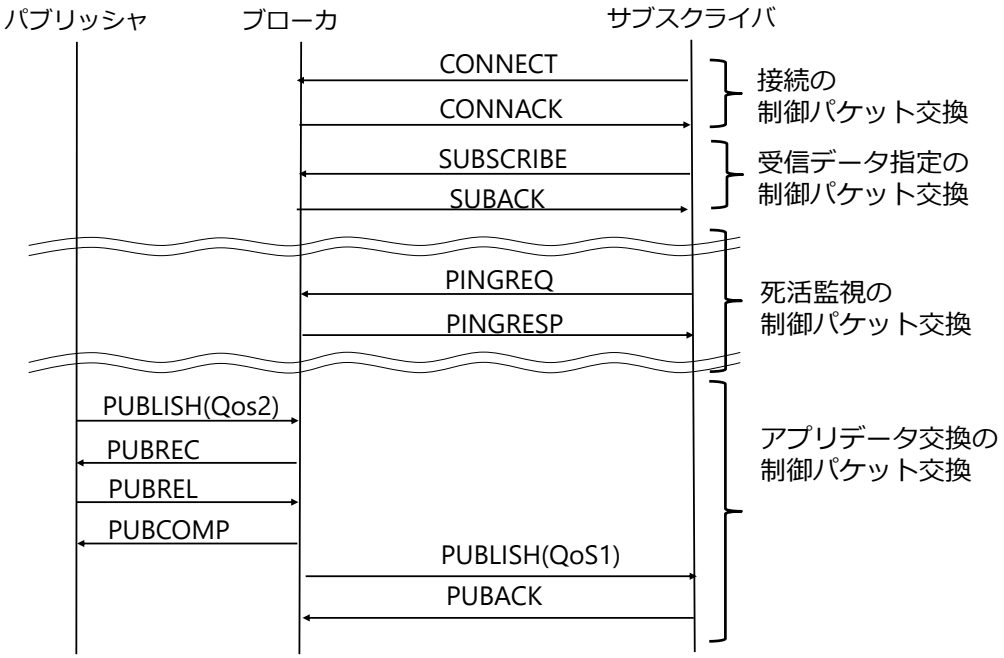

図 9　MQTT のデータフロー

1 つ目の処理はクライアント（パブリッシャもしくはサブスクライバ）とブローカの接続処理である．接続処理は TCP コネクション確立後にクライアントがブローカに CONNECT を送信することで開始される．MQTT では後述する TCP コネクションが正常に終了しなかった場合のデータ再送処理を実現するために，CONNECT（図 10）のペイロードに含まれるクライアント識別子（Client Identifier）によるセッション管理が実装されている．クライアントはシステム全体で一意となるクライアント識別子を CONNECT のペイロードに格納する必要があり，ブローカは受信したクライアント識別子とクライアントを関連付けて情報を管理する．MQTT ではセッションをクライアントとブローカの両方で管理する必要があり，セッションの情報をセッションステート（Session State）と呼ぶ．セッションステートには送信中のデータなどが保持されており，正常でない TCP コネクション切断でもクライアントが同じクライアント識別子を CONNECT で送信することで，送信が完了しなかったデータの再送が行われる．過去のセッションを破棄する場合，可変ヘッダの Clean Start を 1 にした CONNECT を送信することで新しいセッションが作成される．また，User Name Flag や Password Flag を 1 にし，ペイロードにユーザー名とパスワードを格納することで，ユーザー名とパスワードによる認証が可能になる．ブローカは CONNECT が正常であることを確認後，CONNACK をクライアントに返すことで接続処理が完了する．

		7	6	5	4	3	2	1	0
固定ヘッダ		0	0	0	1	0	0	0	0
		パケット長							
可変ヘッダ	パケットタイプの可変ヘッダ	プロトコル名							
		プロトコルバージョン							
		User Name Flag	Password Flag	Will Retain	Will QoS		Will Flag	Clean Start	Reserved
		Keep Alive							
	プロパティ領域	プロパティ長							
		...							
ペイロード		クライアント識別子（Client Identifier）							
		...							

図 10　CONNECT フォーマット

2 つ目の処理はサブスクライバのサブスクライブ処理である．サブスクライブ処理はサブスクライバが受信したいトピックを SUBSCRIBE（図 11）のペイロードに格納して送信することで実現される．サブスクライブ時，MQTT のトピックは「/」を区切り文字とした階層構造で表現できるため，複数のトピックにマッチするワイルドカード（+, #）を利用することができる．「+」は 1 階層にマッチし，「#」は複数階層にマッチする．例えば「/room/sensorA/temperature」,「/room/sensorB/temperature」,「/room/sensorA/humidity」の 3 つのトピックが存在しているとする．この状態で「/room/+/temperature」をサブスクライブした場合，2 階層目は任意の文字列とマッチするため，「room/sensorA/temperature」と「/room/sensorB/temperature」のトピックからデータを受信することができる．「/room/#」をサブスクライブした場合，「/room」を 1 階層目に持つトピックにマッチするため，全てのトピックからデータを受信することができる．SUBSCRIBE を受信したブローカは受信したトピック毎に正常に処理できたかを SUBACK のペイロードに含めてサブスクライバに返すことでサブスクライブ処理が完了する．

		7	6	5	4	3	2	1	0
固定ヘッダ		0	0	1	1	0	0	1	0
		パケット長							
可変ヘッダ		...							
ペイロード	トピックフィルター1	トピック長							
		トピック名							
		Reserved		Retain Handling		RAP	No Local	QoS	
		...							
	トピックフィルターN	トピック長							
		トピック名							
		Reserved		Retain Handling		RAP	No Local	QoS	

図 11　SUBSCRIBE フォーマット

3 つ目の処理は死活監視の処理である．死活監視処理は TCP コネクションは接続されているが，アプリケーションがハングアップしてデータの送受信が正確に行えない問題を回避するために，一定間隔毎にクライアントが PINGREQ を送信することで実現される．クライアントは接続時に CONNECT の可変ヘッダに含まれる Keep Alive（図 10）で PINGREQ の送信間隔をブローカに通知する．ブローカは Keep Alive の値を記録し，Keep Alive の 1.5 倍の時間以内に制御パケットの交換が行われていない場合，クライアントとの接続を切断する．MQTT Version5.0 では CONNACK のプロパティの Server Keep Alive（表 2）を利用することで，ブローカがクライアントの Keep Alive を上書きすることができる．クライアントは接続完了後，Keep Alive の時間毎に PINGREQ を送信し，ブローカは PINGRESP を返すことで死活監視が完了する．

4 つ目の処理はアプリデータの交換処理である．アプリデータ交換の処理はパブリッシャが PUBLISH（図 12）の可変ヘッダにトピック名を，ペイロードにアプリデータを格納して送信することで開始される．PUBLISH を受信したブローカは，そのトピックをサブスクライブしている全てのサブスクライバに PUBLISH を複製して送信することでアプリケーションの交換が完了する．

MQTT の PUBLISH では固定ヘッダに含まれる QoS を用いて，アプリデータ交換の信頼性を 3 つのレベルで指定することが可能になっている．QoS 別のデータフローを図 13 に示す．QoS 0 は At Most Once（最大 1 回）の送信を保証する．このレベルでは送信者が PUBLISH を送信後，確認応答を待つことなく PUBLISH をセッションステートから破棄するため送信効率は高いが，送信中に TCP コネクションが切断された場合にデータの損失が発生する可能性がある．QoS 1 は At Least Once(少なくとも 1 回) の送信を保証する．このレベルでは送信者は PUBLISH を送信後，受信者から確認応答である PUBACK を受信するまでセッションステートに PUBLISH を保持する．PUBLISH と PUBACK の紐づけにはパケット識別子が利用され，同じパケット識別子の PUBACK を受信するとセッションステートの PUBLISH は削除される．送信中に TCP コネクションが切断され，同じセッションで再接続が行われた場合，セッションステートに保持されている PUBLISH を再送することで少なくとも 1 回の送信を実現する．再送時は重複送信であることを示すために固定ヘッダの DUP を 1 にして再送される．PUBACK が損失した場合，同一の PUBLISH が複数回受信者に送信されるため，データの重複が発生する可能性がある．QoS 2 は Exactly Once（正確に 1 回）の送信を保証する．このレベルでは 2 回の確認応答が行われる．1 回目の確認応答は PUBLISH のメッセージを損失なく受信者に送信するための処理である．受信者は PUBLISH 受信時に PUBLISH のパケット識別子を確保し，PUBREC を返す．確保中に同じパケット識別子の PUBLISH を受信した場合，前回の PUBLISH を上書きすることで重複を回避する．送信者は PUBREC を受信後，セッションステートから PUBLISH を削除する．ここまでの処理で PUBLISH の交換が完了する．2 回目の確認応答はパケット識別子の解放を行うための処理である．送信者は PUBREL に PUBLISH と同じパケット識別子を格納して送信し，受信者は受信したパケット識別子を解放する．受信者は PUBCOMP に同一のパケット識別子を格納して返し，送信者は PUBCOMP 確認後にパケット識別子を解放する．データの損失，重複が発生しないため，正確な 1 回の送信が保証されるが，オーバーヘッドは大きくなる．

<table>
<tr><th colspan="2"></th><th>7</th><th>6</th><th>5</th><th>4</th><th>3</th><th>2</th><th>1</th><th>0</th></tr>
<tr><td colspan="2" rowspan="2">固定ヘッダ</td><td>0</td><td>0</td><td>1</td><td>1</td><td>DUP</td><td colspan="2">QoS</td><td>RETAIN</td></tr>
<tr><td colspan="8">パケット長</td></tr>
<tr><td rowspan="5">可変ヘッダ</td><td rowspan="3">パケットタイプのヘッダ</td><td colspan="8">トピック長</td></tr>
<tr><td colspan="8">トピック名</td></tr>
<tr><td colspan="8">パケット識別子</td></tr>
<tr><td rowspan="2">プロパティ</td><td colspan="8">プロパティ長</td></tr>
<tr><td colspan="8">...</td></tr>
<tr><td colspan="2">ペイロード</td><td colspan="8">データ</td></tr>
</table>

図 12　PUBLISH フォーマット

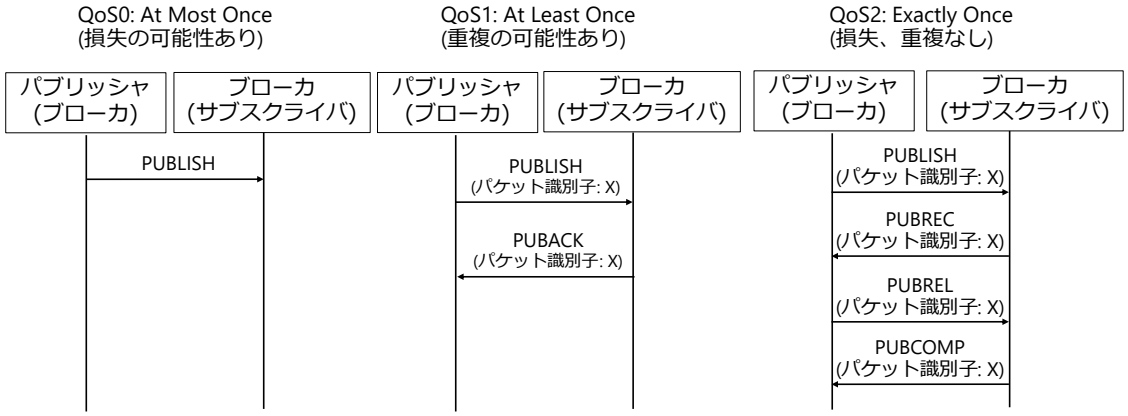

図 13　QoS 別のデータフロー

2.3.4　共有サブスクリプション

共有サブスクリプションは MQTT Version5.0 で追加された機能であり，トピックに紐づいた複数のサブスクライバを 1 つのグループとして関連付けることで，トピックへのデータをグループ内の複数のサブスクライバで分散処理することを可能にする．通常のサブスクリプションと共有サブスクリプションとの比較を図 14 に示す．通常のサブスクリプションでは全てのサブスクライバを独立して扱い，ブローカは Filter にマッチするトピックへのデータをサブスクライブしている全てのサブスクライバに送信する．通常のサブスクリプションは異なる処理を実行する複数のサブスクライバで同じデータを処理する場合に利用される．一方，共有サブスクリプションでは特殊なトピック「$share/{Share Name}/{Filter}」をサブスクライブし，トピックが完全に一致するサブスクライバでグループを形成する．Filter はワイルドカードを含むトピック名であり，ブローカは Filter にマッチするトピックへのデータをグループ内の 1 つのサブスクライバを選択して送信する．共有サブスクリプションは同じ処理を実行する複数のサブスクライバでトピックのデータを分散処理する場合に利用される．MQTT Version5.0 ではサブスクライバの選択方法は仕様として定義されておらず，ブローカの実装依存となっている．OSS の EMQX[10] ではランダム，ラウンドロビン，スティッキー，ハッシュの 4 つの静的負荷分散をサポートしている．HiveMQ[11] はブローカからサブスクライバへの送信はランダムになっている．

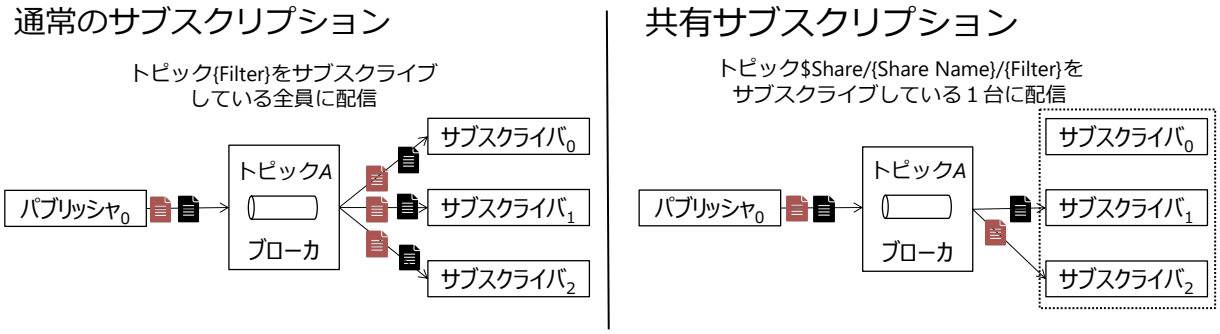

図 14　通常サブスクリプションと共有サブスクリプション

2.4　負荷分散手法

2.4.1　負荷分散手法の概要

負荷分散とはクライアントからサーバへのデータ送信をロードバランサと呼ばれる負荷分散装置を介すことで複数台のサーバに分散する手法であり，Web システムなどで利用されている．負荷分散の利点は，データ処理の負荷を複

数のサーバで分散することによるシステム全体のパフォーマンスの向上と，一部のサーバが停止してもサービスを継続できる高可用性の提供である．負荷分散の手法には静的負荷分散手法と動的負荷分散手法がある．

2.4.2 静的負荷分散手法

静的負荷分散手法はサーバの状態を考慮することなく事前に定められたルールに従って送信先を選択する手法である．一般的な Web システムでの静的負荷分散手法としてはラウンドロビン，重み付きラウンドロビン，IP ハッシュがある（図 15）．

ラウンドロビンは接続しているサーバを順番に選択する手法である．データは全てのサーバに均等に送信されるため，サーバの性能が均一であり，データの処理負荷が一定のシステムに適している．重み付きラウンドロビンはラウンドロビンの配信比率を指定可能な手法である．データは配信比率に従って送信されるため，サーバの性能が不均一であるがデータの処理負荷が一定のシステムに適している．IP ハッシュは送信元の IP アドレスをハッシュ関数を通して送信先のサーバとマッピングする手法である．クライアントによって送信先のサーバが固定されるため，ステートフルな通信が必要なシステムに適している．

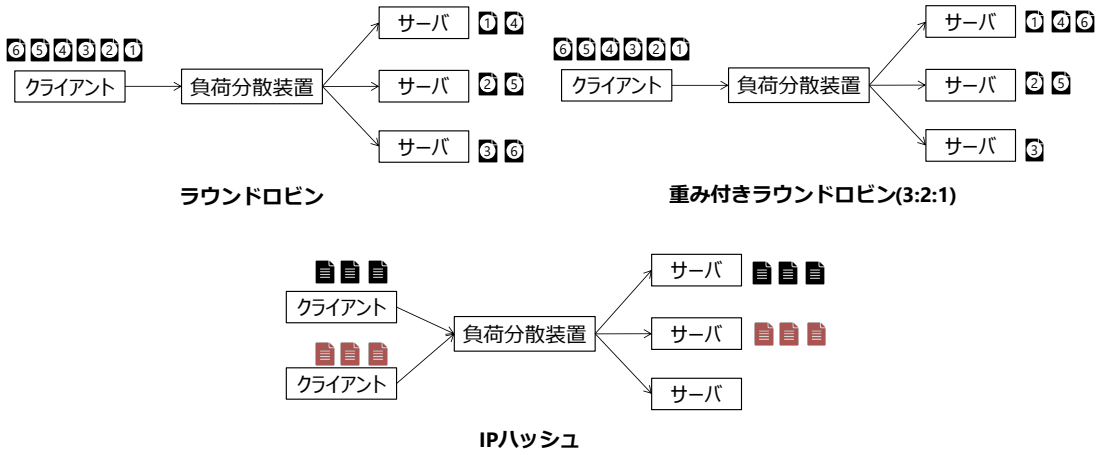

図 15　静的負荷分散手法

2.4.3 動的負荷分散手法

動的負荷分散手法はサーバの状態を随時考慮して送信先を選択する手法である．一般的な Web システムでの動的負荷分散手法としては最小接続数，最小応答時間，最小データ通信量，最小サーバ負荷がある（図 16）．

最小接続数は TCP コネクションが最も少ないサーバを選択する手法である．HTTP ではリクエストが終了する毎に TCP コネクションが切断されることから，コネクション数が現在のサーバの処理数となる．処理中のリクエストが少ないサーバを選択するため，サーバの性能が不均一な場合やデータによって処理時間が異なるシステムに適している．最小応答時間は応答時間によってサーバへの配信比率を変化させる手法である．HTTP ではリクエストに対してレスポンスが発生することから，応答が早いサーバほど性能が高くなる．処理性能の高いサーバへの配信比率を高めるため，サーバの性能が不均一な場合に適している．最小データ通信量はサーバに送信したデータ量が最も少ないサーバを選択する手法である．送信されるデータ量が均一になることから，入力データのサイズによって処理時間が異なるシステムに適している．最小サーバ負荷はサーバの CPU やメモリの使用率が少ないサーバを選択する手法である．この手法ではリソース状況を収集するエージェントをサーバ上に導入する必要がある．負荷分散装置が振り分ける処理以外の負荷上昇も検知することができるため，複数種類のデータ処理がサーバ上で発生するシステムに適している．

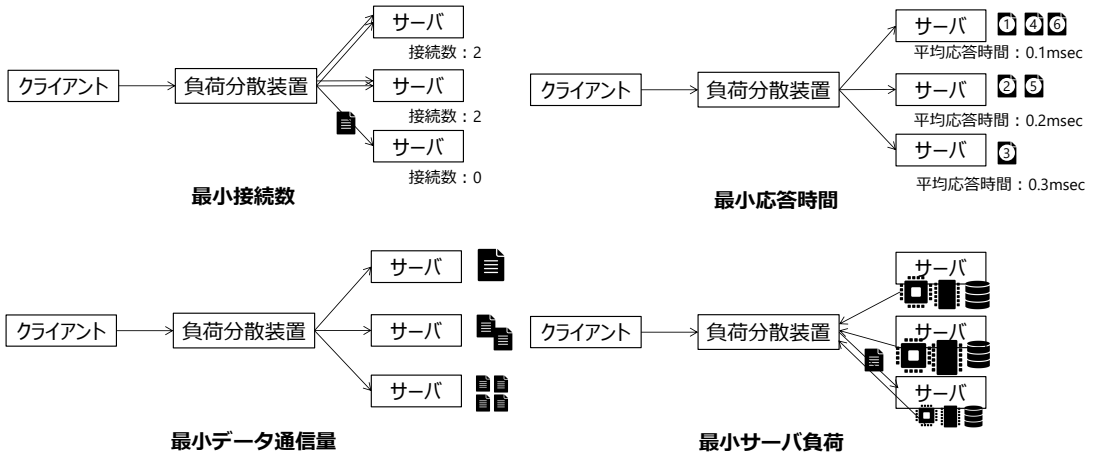

図 16　動的負荷分散

2.5　エッジコンピューティング

2.5.1　エッジコンピューティングの概要

従来の IoT システムでは，デバイス性能の制約からデータ処理リソースをクラウド上に配置し，全てのデータをクラウドに送信するクラウドコンピューティングが主流であった．しかし，近年の IoT システムでは，AR や自動運転など高度化したユースケースで課題となるデバイスとクラウド間の遅延，スマート家電などのデータ生成デバイスの増加によるネットワーク帯域の不足，ウェアラブルデバイスにおけるプライバシー情報の保護など，新しい問題が発生している．これらの問題を解決するために，データが生成される IoT デバイスの近傍（エッジ）にクラウド上の一部の処理を実行するリソースを配置する手法がエッジコンピューティングである [13][14]．クラウドコンピューティングとエッジコンピューティングの比較を図 17 に示す．ここでの「エッジ」はデータソースとクラウドの間の経路に存在するあらゆるリソースが該当する．例えば，工場のような環境では拠点内のサーバやゲートウェイが，スマートフォンや自動車などのモバイル環境では基地局などがエッジに該当する．

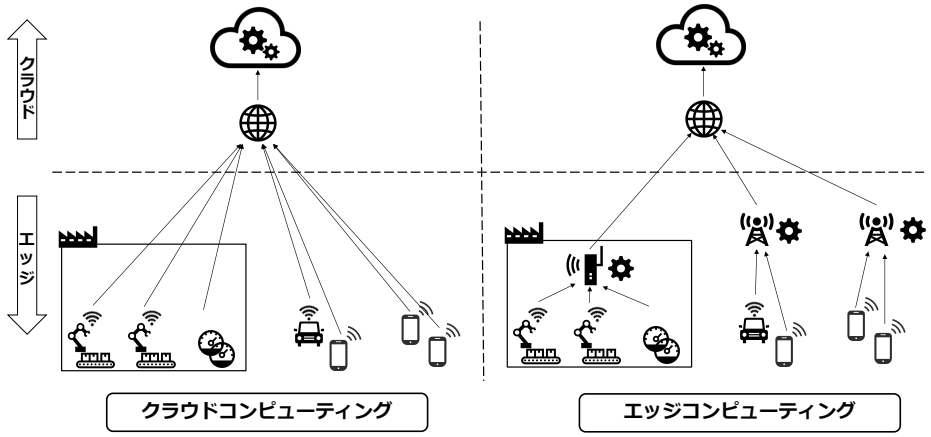

図 17　クラウドコンピューティングとエッジコンピューティング

エッジコンピューティングの利点として，物理的な距離の近さによる応答時間の短縮，データ前処理によるネットワーク帯域の効率化，伝送経路の短縮による信頼性の向上などが挙げられる．エッジコンピューティングで最適化可能な指標は遅延，帯域幅，エネルギー，コストなど様々であり，自身の最適化したい指標に合わせて処理リソースの配置を考える必要がある．

2.5.2 エッジコンピューティングを効率化するアーキテクチャ

エッジコンピューティングでは，計算処理が実行されるリソースのハードウェアや情報交換に利用される通信プロトコルの異種性から，アプリケーションの実装やデプロイ（プログラマビリティ）が困難である点が課題として指摘されている [13]．この問題の解決策として，コンテナ仮想化技術とマイクロサービスを組み合わせたアーキテクチャが主流となっている [15]．

コンテナ仮想化技術とはアプリケーションを実行するために必要なコードやランタイムなどをパッケージ化したコンテナを作成し，コンテナランタイム上で実行可能にする仮想化技術である．コンテナランタイムによって OS やハードウェアが抽象化されており，コンテナはあらゆるサーバのコンテナランタイム上で実行可能であるため，高いポータビリティを有する（図 18）．さらに，複数のサーバ上にあるコンテナランタイムを一元管理するオーケストレーションツール [8][9] の登場により，広域に分散したサーバへのコンテナのデプロイやスケーリングが容易になったことから，エッジコンピューティングに適した技術になっている．

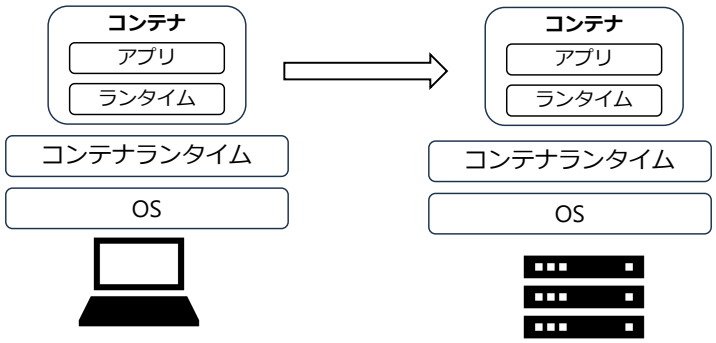

図 18　コンテナ仮想化技術のポータビリティ

マイクロサービスアーキテクチャとは独立した小さなサービスの組み合わせによって一つのアプリケーションを構築する手法である [16]．個々のサービスは独立したプロセスで動作し，サービス間の連携は API を介して実現される．従来の全ての機能を 1 つのプロセスが有するモノリシックアーキテクチャとマイクロサービスアーキテクチャの比較を図 19 に示す．マイクロサービスの利点はそれぞれのサービスを独立して開発することが可能な点にある．エッジコンピューティングではクラウドで実行されていたサービスの一部をエッジにオフロードするユースケースが主であり，これは従来のモノリシックアーキテクチャでは実現できない．また，サービスの負荷によってデプロイ先を柔軟に変更可能であり，これらの点からマイクロサービスはエッジコンピューティングと親和性が高い．

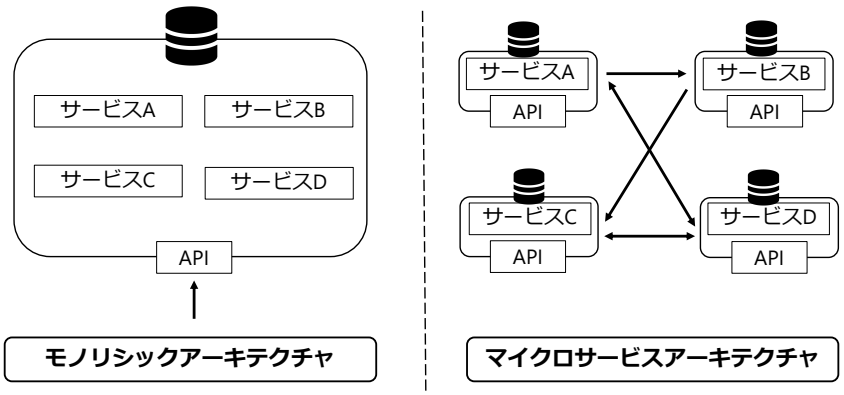

図 19　モノリシックアーキテクチャとマイクロサービスアーキテクチャ

[15] では，エッジコンピューティングのアーキテクチャとして，コンテナ仮想化技術の Docker[17] と Docker のオーケストレーションツールである Docker Swarm[18] を組み合わせた基盤にマイクロサービスを展開するアーキテクチャを構築している（図 20）．Docker Swarm にはコンテナを実行するワーカーノードと，ワーカーノードを管理するマスターノードがあり，マスターノードをクラウドに，ワーカーノードをクラウドとエッジに配置することで，クラウドからエッジの一元管理と計算処理の分散配置を実現している．計算処理はマイクロサービスによって分離され，サービス間の通信にはメッセージングシステムの OSS である NATS[19] を利用している．メッセージングシステムを利用することで通信プロトコルの複雑さを隠蔽するとともに，デバイスとクラウドの双方向通信を実現している．

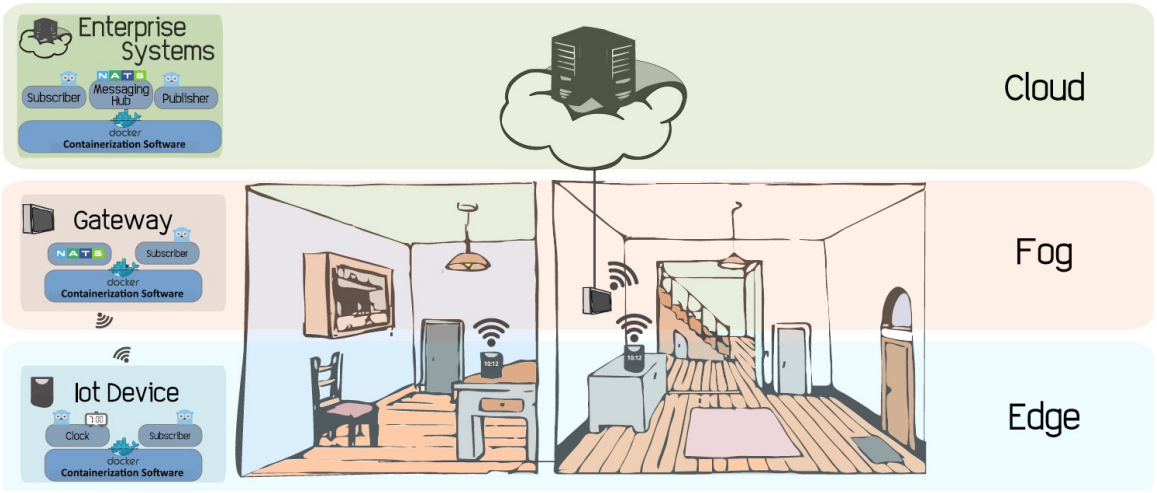

図 20　[15] が構築したアーキテクチャ

第 3 章　関連研究

3.1　共有サブスクリプションに関する関連研究

MQTT の共有サブスクリプションを利用した IoT データ基盤の検討を行った研究 [20] では，DNS ラウンドロビンによってパブリッシャの接続を複数のブローカに分散し，各ブローカに共有サブスクリプションのサブスクライバを接続させる構成を提案している．提案手法の概要を図 21 に示す．この構成によってパブリッシャからのデータ送信によるブローカとサブスクライバの負荷を軽減している．評価では以下の 4 つのパターンでパブリッシャとサブスクライバのスループット，パブリッシャがデータを生成してからサブスクライバで処理されるまでの時間であるレイテンシを測定し，比較評価を行っている．

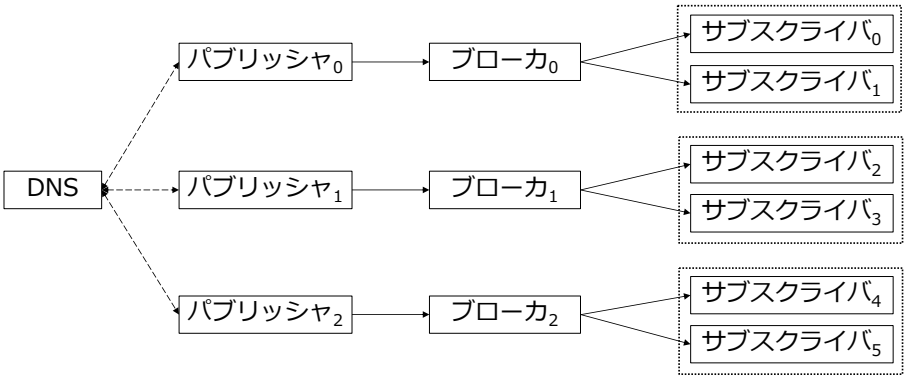

図 21　[20] の提案手法の概要

SGL-SGL
: パブリッシャ数 8, ブローカ数 1，サブスクライバ数 1.

SGL-SHD
: パブリッシャ数 8, ブローカ数 1，サブスクライバ数 2(共有サブスクリプション).

MLT-SGL
: パブリッシャ数 8, ブローカ数 2，サブスクライバ数 2．各ブローカにサブスクライバを 1 台ずつ接続.

MLT-SHD（提案手法）
: パブリッシャ数 8, ブローカ数 2，サブスクライバ数 4(共有サブスクリプション)．各ブローカにサブスクライバを 2 台ずつ接続.

評価結果を図 22，23 に示す．図 22 の結果から提案手法の構成である MLT-SHD ではシンプルな構成である SGL-SGL と比較してパブリッシャ, サブスクライバ共に高いスループットを提供可能であることを示している．一方で図 23 の結果から提案手法のレイテンシが増加する結果となっている．これはパブリッシャのスループットが向上したことでブローカが過負荷になり，メッセージがブローカで滞留していることが原因だと述べられている．

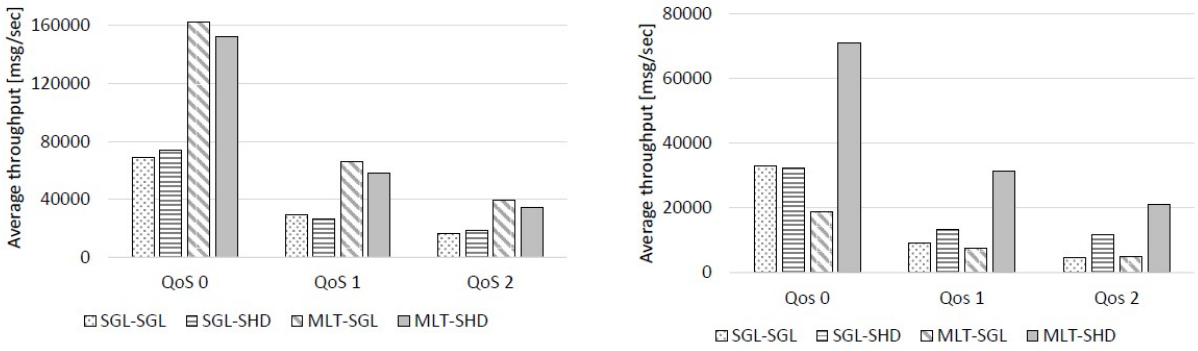

図 22　[20] のスループットの評価結果

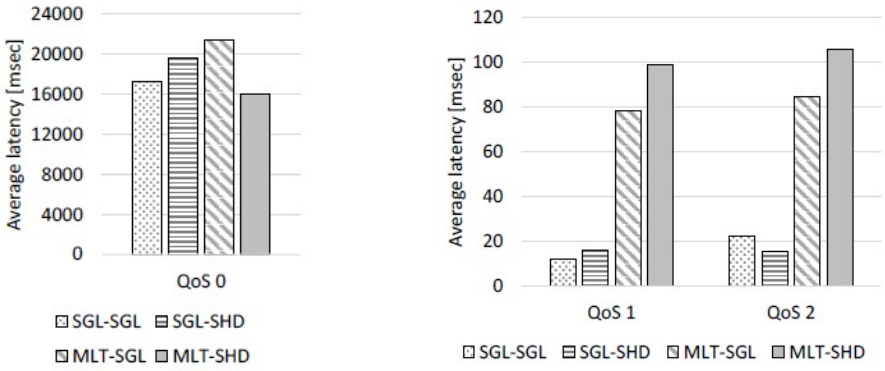

図 23　[20] のレイテンシの評価結果

クラスタ構成のブローカにおける共有サブスクリプションのメッセージスケジューリングを提案した研究 [21] では，サブスクライバの性能を定量的な値で定義し，ブローカの性能を接続しているサブスクライバの性能の総和とすることで，クラスタ内に送信されたデータに対して高性能のブローカに高確率で送信するリソース状況に基づいた動的負荷分散手法を提案している．提案手法ではサブスクライバの容量を t_i と定義し，各ブローカの容量 BR_k を以下の通りとしている．

$$BR_k = \sum_{i=1}^{m} t_i$$

m はブローカ k に接続しているサブスクライバの数である．ここでブローカ k にデータが送信される確率は以下の通りとなる．

$$BM_k = \frac{BR_k}{\sum_{i=1}^{n} t_i}$$

n は全てのサブスクライバの台数である．例えば図 24 の構成の場合，B に送信されたデータが B_3 に送信される確率は

$$BM_3 = \frac{t_4 + t_5}{t_1 + t_2 + t_3 + t_4 + t_5}$$

となる．データがサブスクライバに送信されるとブローカはサブスクライバの容量を減少させる．各サブスクライバの性能を時間に応じて増減させることでクラスタ内のデータ転送を効率化している．

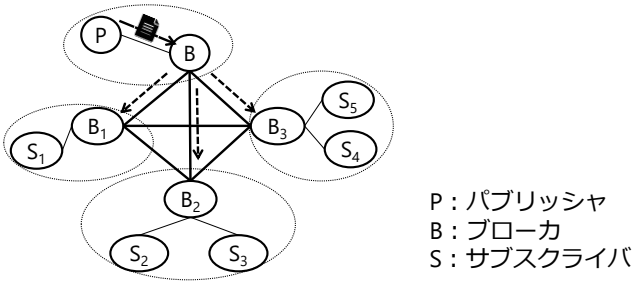

図 24　[21] の提案手法の概要

評価ではシミュレーション上で 3 台のクラスタ構成のブローカにサブスクライバ 20 台を分散接続させた環境を構築し，既存の静的負荷分散手法と比較を行っている．各サブスクライバの性能が全て同一の場合と異なる場合のそれぞれで比較を行い，提案手法が既存手法と比較して高速な処理を達成できることを示している（図 25）．

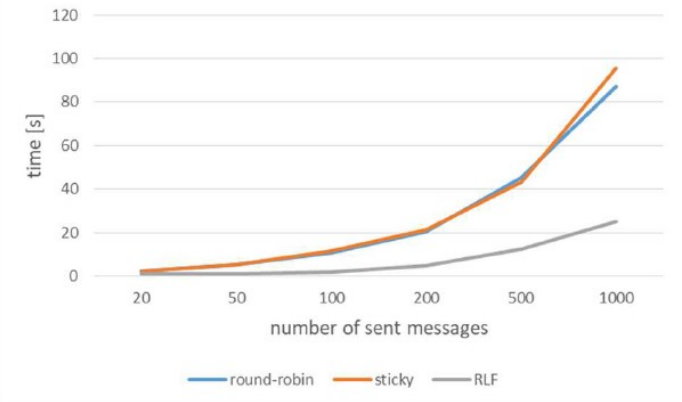

図 25　[20] の評価結果

3.2　動的負荷分散アルゴリズムに関する関連研究

近年でも多くの動的負荷分散アルゴリズムが提案されているが，その中でも保留中のリクエストに基づいた動的負荷分散のアルゴリズムが高いパフォーマンスを達成できることが示されている [22]．提案されているアルゴリズムは，負荷分散装置で HTTP リクエストとそのレスポンスをトレースすることで，振り分け先のサーバの保留中のリクエストを管理し，最も保留中のリクエストが少ないサーバを選択するアルゴリズムとなっている [23]．評価では性能の異なるサーバ 5 台に対して 100 から 10000 のリクエストが送信される環境で，既存のラウンドロビンアルゴリズムと提案手法の比較を行っている．評価指標として 1 秒あたりに処理可能なリクエスト数と全てのリクエストを処理するまでの時間を利用しており，評価結果から両方の指標で提案手法が優位であることを示している（図 26）．

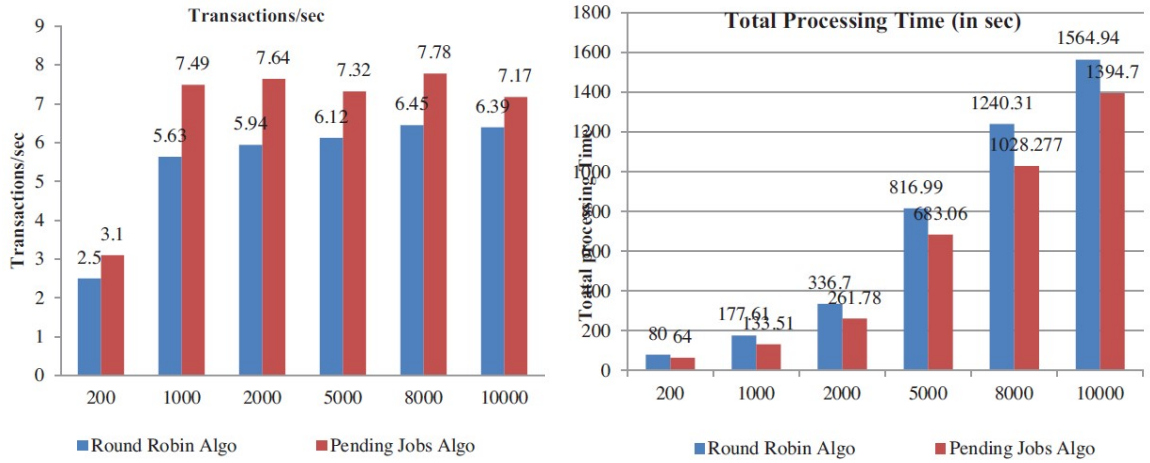

図 26　[23] の評価結果

3.3　エッジコンピューティングにおける MQTT に関する研究

エッジ環境での利用を想定したコンテナベースの MQTT ブローカに関する研究 [24] では，リソースやネットワーク性能が制限されているエッジノードにおけるブローカの過負荷が原因の障害を回避するために，エッジ環境でオーケストレーションツールによるクラスタを構築し，クラスタ上で水平方向にスケール可能な分散 MQTT ブローカを提案している．提案されているアーキテクチャを図 27 に示す．提案手法ではクラスタノードにブローカとパケットフィルタリング機能を持つ中間サーバ（Mid-Tier Server）をコンテナで展開する．中間サーバはノードが受信した MQTT パケットをデコードしてサブスクリプションを管理するとともに，隣接ノードとして指定されたクラスタノードとサブスクリプションの交換を行う．クラスタ全体でサブスクリプションを同期することで，受信した PUBLISH パケットの送信先ノードを決定するルーティングテーブルを構築する．クライアントはロードバランサによってクラスタ内のブローカに分散接続されるが，中間サーバのルーティングによって異なるブローカのメッセージも受信することが可能になっている．また，中間サーバは送信中のメッセージの管理も行っており，クラスタノードの停止に対しても再送制御が適用される．提案手法の利点は MQTT ブローカの実装を変更することなくブローカの台数を増減可能な点である．

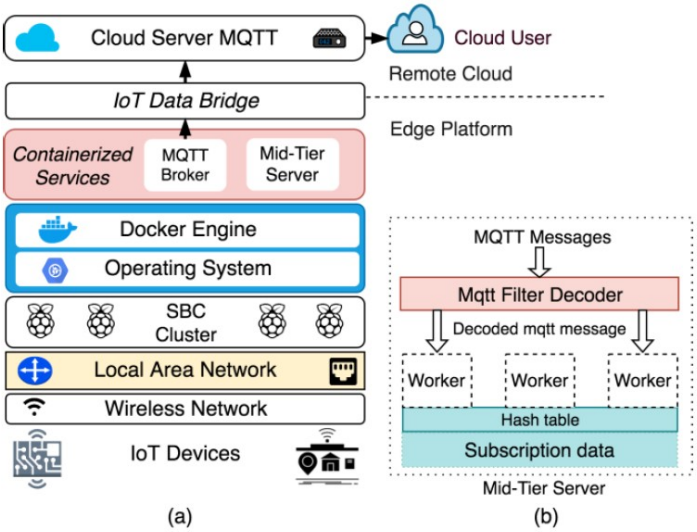

図 27　[24] が提案するアーキテクチャ

提案手法の動作例を図 28 に示す．Client 1 は Cluster Node A の Broker A に接続しており，「r/temp」をサブスクライブしている．Cluster Node A は Cluster Node B に「r/temp」のサブスクリプションを同期する．この状態で Client 2 が Cluster Node B の Broker B にメッセージをパブリッシュした場合，メッセージは Broker B だけでなく，Cluster Node B の中間サーバによるルーティングによって Cluster Node A の Broker A にも送信される．

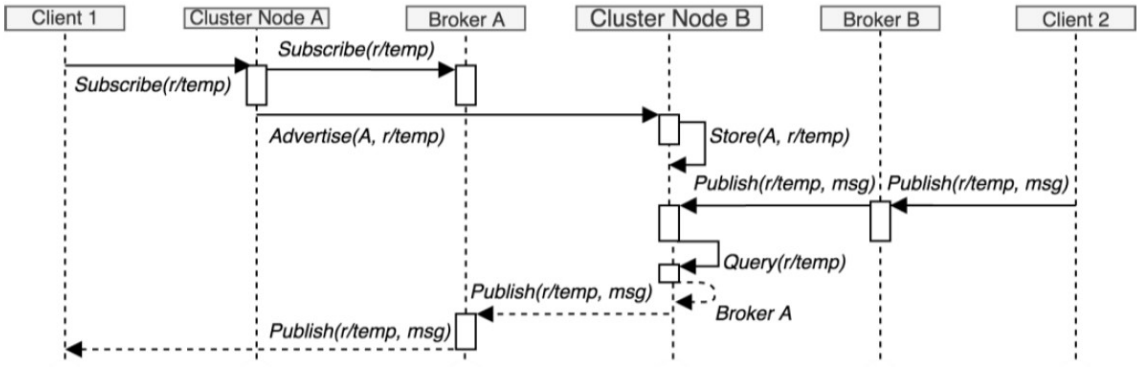

図 28　[24] の動作例

評価ではシングルボードコンピュータである Raspberry Pi[25]4 台のクラスタ上に，ブローカが 1 台の場合と 4 台の場合をそれぞれ構築し，最大 1000 組のパブリッシャとサブスクライバで負荷テストを行っている．評価指標としてはパブリッシャとサブスクライバのスループット，レイテンシ，CPU の使用率，RAM の使用量を用いている．図 29 からパブリッシャとサブスクライバのスループットでは 4 台より 1 台の方が高い結果となった．また，レイテンシも 4 台の方が高い結果となった．これはクラスタノード間のメッセージリレーが原因であることが論文内で指摘されている．CPU の使用率は図 30 からピークは 1 台の場合は 2.47%，4 台の場合は 3.65% となっていることが分かる．これは中間サーバのオーバヘッドであることが述べられている．RAM の使用量は図 31 より 1 台の場合が 320MiB 周辺を遷移している一方で，4 台の場合は 240MiB を下回っていることから，クラスタ構成による MQTT ブローカはメモリ使用のオーバヘッドを削減可能であることが示唆されている．

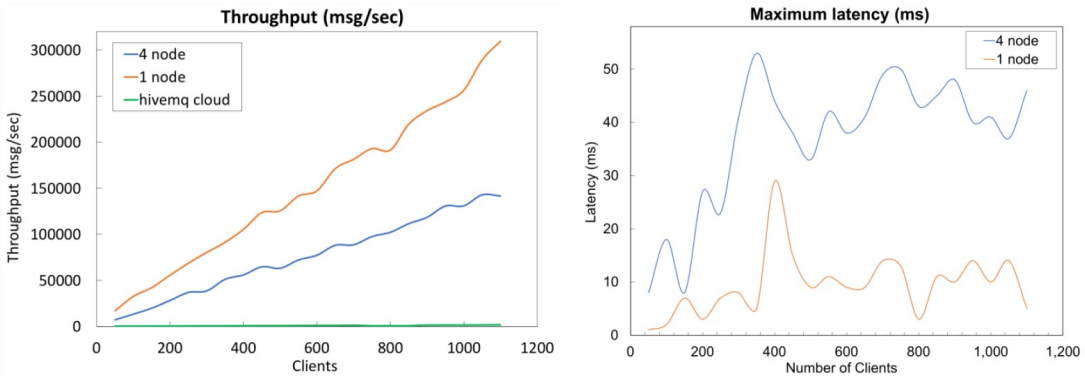

図 29　[24] のパブリッシャとサブスクライバのスループット結果

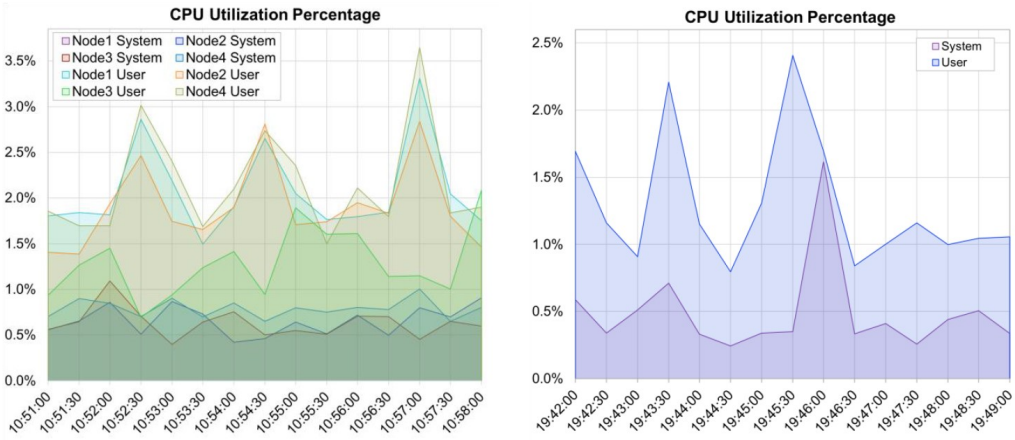

図 30　[24] の CPU 使用率の結果

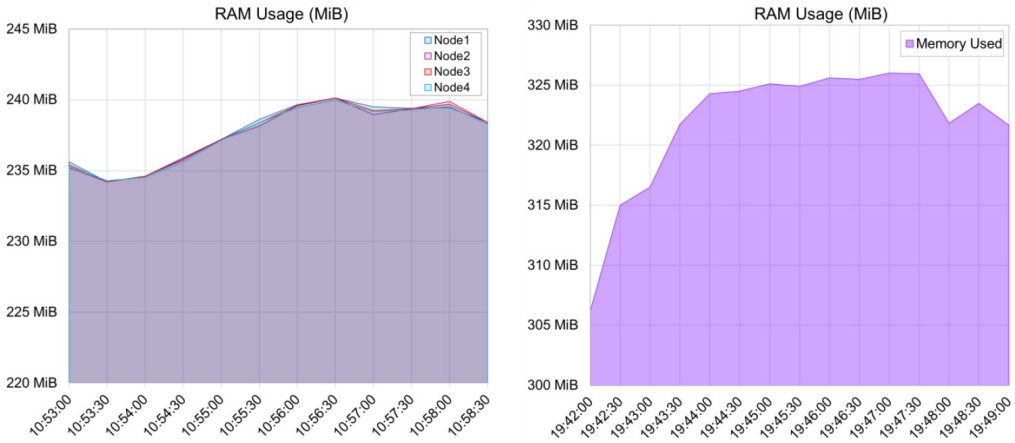

図 31　[24] のメモリ使用量の結果

3.4　関連研究における課題

本章で述べた関連研究における課題として，以下の 2 点が挙げられる．

1. サブスクライバの処理性能が考慮されていない
2. サブスクライバの情報をブローカに伝達する仕組みが存在しない

課題 1 について [20] では，パブリッシャからサブスクライバで処理されるまでの時間であるレイテンシを指標として評価しているが，実験ではサブスクライバはメッセージを受信するのみであり，サブスクライバでのメッセージ処理時間が考慮されていない．AR やビデオ分析などユースケースの高度化によってサブスクライバの負荷は高まっており，メッセージ処理の滞留はブローカだけでなくサブスクライバでも発生する可能性がある．また，[24] ではリソースが制限された環境におけるブローカの障害を回避するために，コンテナベースの分散 MQTT ブローカを提案しているが，エッジコンピューティングにおけるアベイラビリティとスケーラビリティを向上させるためにはサブスクライバの冗長化とスケーリングも必要となる．共有サブスクリプションとオーケストレーションツールの組み合わせはこれらのニーズに適しているが，既存の実装である静的負荷分散手法では図 32 に示すようなオートスケールによるサブスクライバの性能変化に対応することが出来ず，ワークロード全体に遅延が生じるリスクがある．

課題 2 について [21] では，ブローカがサブスクライバの処理容量から配信先のサブスクライバを選択する手法を提案しているが，評価ではブローカが全てのサブスクライバの処理容量のを事前に知っていることが前提になっている．

[21] ではサブスクライバの情報をブローカにリアルタイムで伝達する手段は提案されておらず，これは新しく接続されるサブスクライバに対応できないことや，入力されるデータによってサブスクライバの処理時間が異なる環境では対応することができない問題が生じる．

そこで，本研究では 2 つの課題を解決可能な共有サブスクリプションにおける動的負荷分散を実現する手法を提案する．提案手法では課題 2 のサブスクライバの情報をリアルタイムに伝達できない問題に対して，MQTT の制御パケットを拡張することで解決する．また，課題 1 のサブスクライバの処理性能が考慮されていない問題に対して，Web システムで高いパフォーマンスを達成した [23] を MQTT プロトコル上で実装することで解決する．

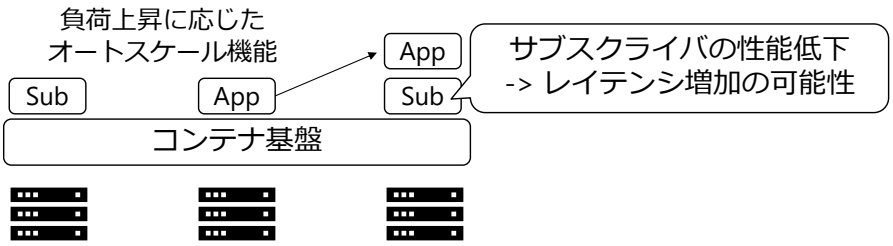

図 32　サブスクライバの性能変化に対する問題

第 4 章　提案手法

4.1　システム設計

4.1.1　サブスクライバの情報をリアルタイムで伝達可能にする設計

MQTT ではサブスクライバがブローカに情報を伝達可能な制御パケットとして CONNECT，SUBSCRIBE，UNSUBSCRIBE，PUBACK，PUBREL，PUBCOMP，PINGREQ がある．これらの制御パケットの内，サブスクライバとブローカでの定期的な交換が発生する制御パケットには PUBACK，PUBREL，PUBCOMP，PINGREQ がある．PUBACK，PUBREL，PUBCOMP のプロパティ領域には User Property が許可されているため，任意の情報を交換することが可能であるが，パブリッシャやサブスクライバが指定する QoS に依存することや，データを受信するまでブローカに送信することができない点からリアルタイムの情報交換には適していない．そのため，提案手法では死活監視のために利用される PINGREQ を拡張し，任意の情報を格納可能にすることでサブスクライバとブローカの情報交換を実現する．PINGREQ の既存のパケットフォーマットと拡張後のパケットフォーマットを図 33 に示す．既存の PINGREQ は固定ヘッダのみの制御パケットであるが，拡張後は可変ヘッダのプロパティ領域とペイロード領域が追加される．ペイロード領域にはサブスクライバの任意の情報を格納することでブローカでのサブスクライバの状態把握が可能である．プロパティ領域には User Property を許可することで，ペイロードに関するメタデータを格納可能である．

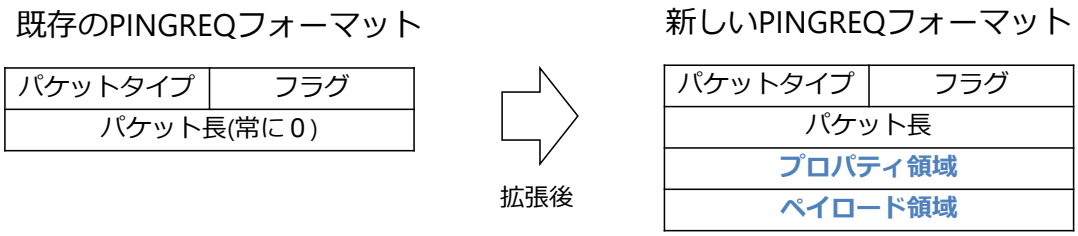

図 33　拡張した PINGREQ のパケットフォーマット

4.1.2　保留中のメッセージ数に基づいた負荷分散アルゴリズムの設計

関連研究 [23] では HTTP リクエストとそのレスポンスをトレースすることでサーバの保留中のリクエスト数の管理を行っていたが，MQTT では QoS 0 の場合にサブスクライバからの確認応答が存在しないため，ブローカでサブスクライバの保留中のメッセージ数を把握することができない．そのため，提案手法ではサブスクライバの保留中のメッセージ数と PINGREQ 送信間の平均処理時間を PINGREQ のペイロードに格納してブローカに通知することで，ブローカでのサブスクライバの状態把握を可能にする．ブローカが管理するサブスクライバの情報を表 3 に示す．ブローカは各サブスクライバごとに表 3 に示す情報を JSON 形式で管理する．提案手法ではサブスクライバのメッセージの平均処理時間には直近 5 回の PINGREQ 送信間の平均処理時間の平均を利用する．これは極端な外れ値の影響を小さくするためである．ブローカは共有サブスクリプションでサブスクライバ選択時に，これらの情報からサブスクライバごとに以下のスコア値を計算し，スコア値の最も高いサブスクライバを選択する．

$$
\begin{aligned}
Score &= t_1 - max((m_q + m_s)t_p - t_2) \\
t_1 &= t_n - t_s \\
t_2 &= t_n - t_r
\end{aligned}
$$

ここで t_n はサブスクライバ選択開始時刻，t_s は最後にメッセージを送信した時刻，t_r は最後に PINGREQ を受信した時刻，m_q は保留中のメッセージ数，m_s は PINGREQ 間に送信したメッセージ数，t_p はサブスクライバのメッセージの平均処理時間である．また，第 1 項 t_1 はサブスクライバに最後にメッセージを送信してからの経過時間であ

る．この値が大きいほどスコア値が高くなるため，送信から時間が経過しているサブスクライバの優先度を高くする設計となる．第 2 項 $max((m_q + m_s)t_p - t_2, 0)$ はサブスクライバが保留中のメッセージの処理に掛かる推定時間である．提案手法では PINGREQ 受信時の保留中のメッセージ数と PINGREQ 間に送信したメッセージ数の合計をサブスクライバが現在時刻までに処理する必要のあるメッセージ数とし，このメッセージの処理時間 $(m_q + m_s)t_p$ と最後に PINGREQ を受信してからの経過時間 t_2 の差分を保留中のメッセージの処理時間としている．この値が大きいほどスコア値が低くなるため，処理の滞っているサブスクライバの優先度を低くする設計となる．

スコア値の計算時に $t_p = 0$ の場合はグループ内の t_p の平均を利用する．これにより，新しく接続してきたサブスクライバはグループの平均として扱われる．全てのサブスクライバが PINGREQ を送信しない場合や，全てのサブスクライバで保留中のメッセージが存在しない場合は第 2 項の値が 0 となるため第 1 項に従いラウンドロビンとして動作する．

表 3　ブローカが保持する情報

保留中のメッセージ数	m_q
メッセージの平均処理時間	$t_p = \frac{t_{p1}+t_{p2}+t_{p3}+t_{p4}+t_{p5}}{5}$
最後に PINGREQ を受信した時刻	t_r
最後にメッセージを送信した時刻	t_s
PINGREQ 間に送信したメッセージ数	m_s
直近 5 回の PINGREQ 送信間の平均処理時間	$[t_{p1}, t_{p2}, t_{p3}, t_{p4}, t_{p5}]$

4.2　システム実装

提案手法を OSS の Mochi-MQTT Server[26] と paho.mqtt.java[27] を用いて実装を行った．Mochi-MQTT Server では PINGREQ で受信したサブスクライバ情報の管理と共有サブスクリプションでのサブスクライバ選択を実装し，paho.mqtt.java では PINGREQ のペイロードに保留中のメッセージ数と PINGREQ 間の平均処理時間を格納して送信する処理を実装した. 本節では各 OSS での提案手法の実装の詳細と動作概要について説明する．

4.2.1　Mochi-MQTT Server を利用した実装

Mochi-MQTT Server は Go 言語で実装された MQTT Version5.0 準拠の MQTT ブローカである．アプリケーションに組み込み可能な設計となっており, パッケージをインポートして数行記入するだけで MQTT ブローカの機能をアプリケーション上に実装することができる．Mochi-MQTT Server の特徴として，特定のイベント発生時に実行される関数をフックすることで, ブローカの機能を柔軟に拡張可能なイベントフック機能がある．フック可能なイベント一覧を表 4 に示す．提案手法では OnSessionEstablished，OnDisconnect，OnPacketRead，OnSelectSubscribers にフックする関数を実装することでブローカでのサブスクライバの情報管理と共有サブスクリプションでのサブスクライバ選択を実現した．

表 4　フック可能なイベント一覧

イベント名	イベントの発生タイミング
OnStarted	ブローカ正常に起動したとき
OnStopped	ブローカが正常に終了したとき
OnConnectAuthenticate	クライアント接続で認証するとき
OnACLCheck	トピックにメッセージがパブリッシュされたとき
OnSysInfoTick	$SYS から始めるトピックにパブリッシュされたとき
OnConnect	新しいクライアントが接続するとき
OnSessionEstablish	新しいクライアントが接続し，CONNACK を返す直前
OnSessionEstablished	新しいクライアントが接続し，セッションが確立されたとき
OnDisconnect	クライアントが何らかの理由で切断されたとき
OnAuthPacket	Auth パケットを受信したとき
OnPacketRead	制御パケットを受信したとき
OnPacketEncode	制御パケットがエンコードされクライアントに送信される直前
OnPacketSent	制御パケットが送信されたとき
OnPacketProcessed	制御パケットを受信してブローカで正常に処理されたとき
OnSubscribe	SUBSCRIBE パケットを受信したとき
OnSubscribed	クライアントのサブスクライブが正常に完了したとき
OnSelectSubscribers	トピックのサブスクライバを収集し，共有サブスクリプションのサブスクライバが選択される前
OnUnsubscribe	UNSUBSCRIBE パケットを受信したとき
OnUnsubscribed	クライアントのアンサブスクライブが正常に完了したとき
OnPublish	PUBLISH パケットを受信したとき
OnPublished	サブスクライバに PUBLISH パケットの送信が完了したとき
OnPublishDropped	サブスクライバへの PUBLISH パケットの送信が失敗したとき
OnRetainMessage	PUBLISH パケットが Retain されたとき
OnRetainPublished	Retain された PUBLISH パケットの送信が完了したとき
OnQosPublish	QoS 1 以上の PUBLISH パケットがサブスクライバに送信されたとき
OnQosComplete	QoS のフローが完了したとき
OnQosDropped	QoS のフローが完了する前に PUBLISH パケットが破棄されたとき
OnPacketIDExhausted	パケット識別子に割り当てる ID が枯渇したとき
OnWill	クライアントが切断され，Will メッセージが送信されるとき
OnWillSent	サブスクライバへの Will メッセージの送信が完了したとき
OnClientExpired	セッションの期限が切れ削除されるとき
OnRetainedExpired	Retain された PUBLISH パケットの期限が切れ削除されるとき

OnSessionEstablished にフックした関数のフローチャートを図 34 に示す．OnSessionEstablished はクライアントとのセッション確立時に発生するイベントである．この関数ではクライアント識別子をキーとしてハッシュ構造にサブスクライバの情報を初期化して格納する．サブスクライバ情報の各値として m_q, m_s, t_p は 0 に，t_r, t_s を接続時刻に初期化する．

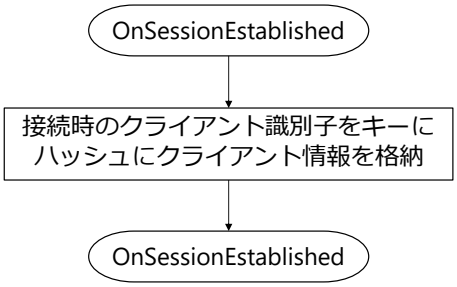

図 34　OnSessionEstabilished の関数

OnDisconnect にフックした関数のフローチャートを図 35 に示す．OnDisconnect はクライアントとのコネクション切断時に発生するイベントである．この関数では切断されるクライアント識別子をハッシュ構造から削除する．

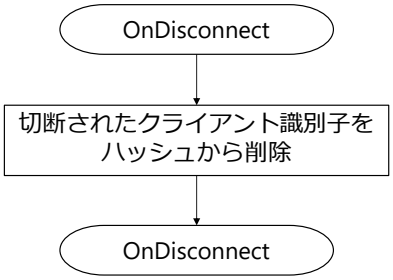

図 35　OnDisconnect の関数

OnPacketRead にフックした関数のフローチャートを図 36 に示す．OnPacketRead はクライアントから制御パケットを受信した時に発生するイベントである．この関数では拡張された PINGREQ を受信時にサブスクライバの情報を更新する．まず，受信した制御パケットが拡張された PINGREQ かを確認する．次に直近 5 回の PINGREQ 送信間の平均処理時間に受信した PINGREQ の値を追加する．ここで，直近 5 回分より多く格納されている場合は最も古い値を削除する．その後，サブスクライバ情報の各値として m_q を受信した値，t_p を直近 5 回の平均，m_s を 0，t_r を現在時刻に更新する．

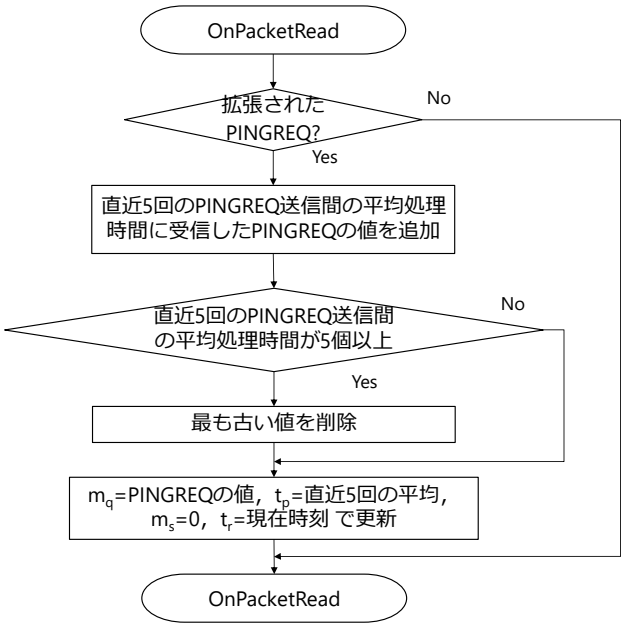

図 36　OnPacketRead の関数

OnSelectSubscribers にフックした関数のフローチャートを図 37 に示す．OnSelectSubscribers は PUBLISH のトピックにマッチした通常のサブスクリプションと共有サブスクリプションのサブスクライバが収集され，共有サブスクリプションのサブスクライバ選択前に発生するイベントである．この関数では共有サブスクリプションのトピック毎に最もスコア値の高いサブスクライバを選択する．サブスクライバの選択後，選択されたサブスクライバの情報を更新する．サブスクライバ情報の各値として m_s をインクリメントで，t_s を現在時刻で更新する．

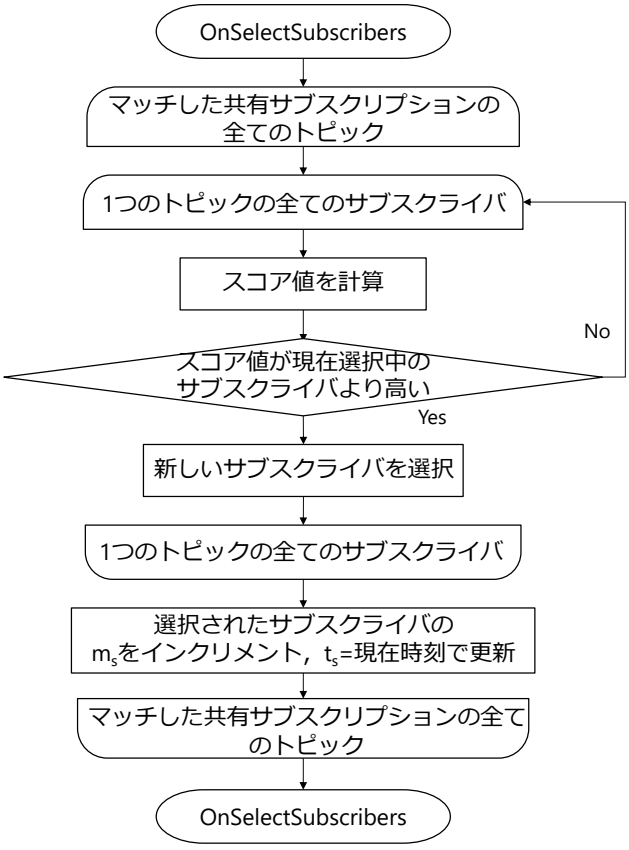

図 37　OnSelectSubscribers の関数

4.2.2　paho.mqtt.java を利用した実装

paho.mqtt.java は Java 言語で実装された MQTT Version5.0 準拠の MQTT クライアントである．パブリッシャとサブスクライバをアプリケーションに実装可能であり，メッセージ処理毎に処理をブロックする同期クライアントと処理をブロックしない非同期クライアントを提供している．paho.mqtt.java では PINGREQ を送信する処理を抽象化した MqttPingSender がインターフェースとして提供されている．本研究では MqttPingSender を実装し，任意のペイロードを格納した PINGREQ を送信可能な抽象クラス ExtendedPingSender を実装した（ソースコード 1）．ExtendedPingSender の動作について説明する．まず，start メソッドで内部クラスの PingTask の 1 回目の起動がタイマーで設定される．タイマーの起動時間には接続時の KeepAlive の値が利用されるが，独立した値を設定することも可能になっている．PingTask では PINGREQ が生成され，内部の comms.checkForActivity で PINGREQ の送信と schedule メソッドによる次の起動設定が行われる．PINGREQ の生成は抽象メソッド createPingreq によって行われるため，ExtendedPingSender を継承して createPingreq をオーバーライドすることで開発者が任意の PINGREQ を生成することができる．

ソースコード 1　ExtendedPingSender

```
1  package org.eclipse.paho.mqttv5.client;
2
3  import org.eclipse.paho.mqttv5.client.internal.ClientComms;
4  import org.eclipse.paho.mqttv5.client.logging.Logger;
5  import org.eclipse.paho.mqttv5.client.logging.LoggerFactory;
6  import org.eclipse.paho.mqttv5.common.MqttException;
7  import org.eclipse.paho.mqttv5.common.packet.MqttPingReq;
8
9  import java.util.Timer;
10 import java.util.TimerTask;
11 import java.util.concurrent.ScheduledExecutorService;
12 import java.util.concurrent.ScheduledFuture;
13 import java.util.concurrent.TimeUnit;
14
15 import static java.lang.Math.min;
16
17 public abstract class ExtendedPingSender implements MqttPingSender {
18       private static final String CLASS_NAME = ExtendedPingSender.class.getName();
19       protected Logger log = LoggerFactory.getLogger(LoggerFactory.MQTT_CLIENT_MSG_CAT,
             CLASS_NAME);
20
21       protected ClientComms comms;
22
23       private Timer timer;
24       private ScheduledExecutorService executorService = null;
25       private ScheduledFuture<?> scheduledFuture;
26       private String clientid;
27
28       private long pingIntervalMilliSeconds = -1;
29
30       @Override
31       public void init(ClientComms comms) {
32             final String methodName = "init";
33
34             if (comms == null) {
35                   throw new IllegalArgumentException("ClientComms␣cannot␣be␣null.");
36             }
37             this.comms = comms;
38
39             clientid = comms.getClient().getClientId();
40             log.setResourceName(clientid);
41       }
42
43       @Override
44       public void start() {
45             final String methodName = "start";
46
47             //@Trace 659=start timer for client:{0}
48             log.fine(CLASS_NAME, methodName, "659", new Object[]{clientid});
49
50             long delay = getDelay(comms.getKeepAlive());
51             if (executorService == null) {
52                   timer = new Timer("MQTT␣Ping:␣" + clientid);
53                   timer.schedule(new PingTask(), delay);
54             } else {
55                   schedule(delay);
56             }
57       }
58
59       @Override
60       public void stop() {
```

```
61              final String methodName = "stop";
62
63              //@Trace 661=stop
64              log.info(CLASS_NAME, methodName, "661", null);
65              if (executorService == null) {
66                    if (timer != null) {
67                          timer.cancel();
68                    }
69              } else {
70                    if (scheduledFuture != null) {
71                          scheduledFuture.cancel(true);
72                    }
73              }
74        }
75
76        @Override
77        public void schedule(long delayInMilliseconds) {
78              long delay = getDelay(delayInMilliseconds);
79              if (executorService == null) {
80                    timer.schedule(new PingTask(), delay);
81              } else {
82                    scheduledFuture = executorService.schedule(new PingRunnable(), delay,
                          TimeUnit.MILLISECONDS);
83              }
84        }
85
86        private class PingTask extends TimerTask {
87              private static final String methodName = "PingTask.run";
88
89              @Override
90              public void run() {
91                    Thread.currentThread().setName("MQTT␣Ping:␣" + clientid);
92                    //@Trace 660=Check schedule at {0}
93                    log.info(CLASS_NAME, methodName, "660", new Object[]{Long.valueOf(
                          System.nanoTime())});
94
95                    // Create PINGREQ before checkForActibity
96                    MqttPingReq pingReq = createPingreq();
97                    comms.updatePingCommand(pingReq); // update pingCommand of ClientState
98
99                    comms.checkForActivity();
100             }
101       }
102
103       private class PingRunnable implements Runnable {
104             private static final String methodName = "PingTask.run";
105
106             public void run() {
107                   String originalThreadName = Thread.currentThread().getName();
108                   Thread.currentThread().setName("MQTT␣Ping:␣" + clientid);
109                   //@Trace 660=Check schedule at {0}
110                   log.fine(CLASS_NAME, methodName, "660", new Object[]{Long.valueOf(
                          System.nanoTime())});
111
112                   // Create PINGREQ before checkForActibity
113                   MqttPingReq pingReq = createPingreq();
114                   comms.updatePingCommand(pingReq); // update pingCommand of ClientState
115
116                   comms.checkForActivity();
117                   Thread.currentThread().setName(originalThreadName);
118             }
119       }
120
```

```
121
122         public void setPingIntervalMilliSeconds(int pingIntervalMilliSeconds) {
123               this.pingIntervalMilliSeconds = pingIntervalMilliSeconds;
124         }
125
126         private long getDelay(long delayInMilliseconds) {
127               if (pingIntervalMilliSeconds == -1) return delayInMilliseconds;
128               return min(pingIntervalMilliSeconds, delayInMilliseconds);
129         }
130
131         abstract protected MqttPingReq createPingreq();
132   }
```

提案手法の実現にはクライアント内部のキューのメッセージ数とメッセージの平均処理時間を格納した PINGREQ を生成する StatusPingSender を実装した（ソースコード 2）．StatusPingSender ではメッセージ処理毎に updateProcessingTimePerMsg を呼び出し，メッセージ処理数とトータルの処理時間を更新する．createPingreq メソッドが呼び出されると，処理したメッセージ数とトータル処理時間から計算した平均処理時間と保留中のメッセージ数が JSON 形式でペイロードに格納された PINGREQ を生成する．PINGREQ 作成後は処理したメッセージ数とトータルの処理時間を初期化する．

ソースコード 2　StatusPingSender

```
1   package org.eclipse.paho.sample.mqttv5app.pingsender;
2
3
4   import com.fasterxml.jackson.core.JsonProcessingException;
5   import org.eclipse.paho.mqttv5.client.ExtendedPingSender;
6   import org.eclipse.paho.mqttv5.client.MqttPingSender;
7   import org.eclipse.paho.mqttv5.common.packet.MqttPingReq;
8
9   import com.fasterxml.jackson.annotation.JsonProperty;
10  import com.fasterxml.jackson.databind.ObjectMapper;
11
12  public class StatusPingSender extends ExtendedPingSender {
13        private static final String CLASS_NAME = StatusPingSender.class.getName();
14
15        private int messageCount;
16        private double totalProcessingTime;
17
18        private ObjectMapper mapper;
19
20        public StatusPingSender() {
21              messageCount = 0;
22              totalProcessingTime = 0;
23              mapper = new ObjectMapper();
24        }
25
26        public void updateProcessingTimePerMsg(double processingTime){
27              final String methodName = "updateProcessingTimePerMsg";
28
29              messageCount++;
30              totalProcessingTime += processingTime;
31        }
32
33        @Override
34        protected MqttPingReq createPingreq() {
35              final String methodName = "cratePingreq";
36
37              Payload payload;
```

```
38              if (messageCount == 0) {
39                    payload = new Payload(comms.getNumberOfMsgsUnprocessed(), 0);
40              } else {
41                    payload = new Payload(comms.getNumberOfMsgsUnprocessed(),
                          totalProcessingTime/messageCount);
42              }
43              MqttPingReq pingReq;
44              try {
45                    String jsonPayload = mapper.writeValueAsString(payload);
46                    pingReq = new MqttPingReq(jsonPayload.getBytes());
47                    log.info(CLASS_NAME, methodName, "Create␣PINGREQ␣payload:␣{0}.", new
                          Object[]{jsonPayload});
48              } catch (JsonProcessingException e) {
49                    throw new RuntimeException(e);
50              }
51              messageCount = 0;
52              totalProcessingTime = 0;
53              return pingReq;
54        }
55
56        class Payload {
57              @JsonProperty("numberOfMsgsInQueue")
58              public int numberOfMsgsInQueue;
59
60              @JsonProperty("processingTimerPerMsg")
61              public double processingTimerPerMsg;
62
63              public Payload(int numberOfMsgsInQueue, double processingTimerPerMsg){
64                    this.numberOfMsgsInQueue = numberOfMsgsInQueue;
65                    this.processingTimerPerMsg = processingTimerPerMsg;
66              }
67        }
68  }
```

4.3 動作概要

実装した提案手法の動作概要を以下に示す.

サブスクライバ

1. ブローカ接続時，PINGREQ の送信間隔を CONNECT の KeepAlive 領域でブローカに伝達する（図 38 ①）.
2. ブローカから PUBLISH 受信時，PINGREQ 送信間の平均処理時間を更新する（図 38 ⑤）.
3. 接続から KeepAlive で指定した時間が経過すると，PINGREQ のペイロードに「保留中のメッセージ数」と「PINGREQ 送信間の平均処理時間」を図 39 に示す JSON 形式の文字列で格納して送信する（図 38 ⑦）.
4. PINGREQ 送信後，PINGREQ 間の平均処理時間を初期化する（図 38 ⑧）

ブローカ

1. サブスクライバ接続時，CONNECT のペイロードに含まれるクライアント識別子をキーとしてハッシュ構造でクライアント情報 m_q, m_s, t_p を 0 で，t_s, t_r を接続時刻で初期化する（図 38 ②）.
2. パブリッシャから PUBLISH 受信時，マッチした共有サブスクリプションのトピックにに属するグループ内のサブスクライバのスコア値を計算し，最もスコア値が高いサブスクライバを選択する（図 38 ③）．サブスクライバ選択後，ハッシュに保存されているサブスクライバ情報の送信したメッセージ数 m_s をインクリメントで，最後にメッセージを受信した時刻 t_s を現在時刻で更新する（図 38 ④）.
3. サブスクライバから拡張された PINGREQ を受信時，ペイロードの値からサブスクライバ情報の保留中のメッセージ数 m_q とメッセージの平均処理時間 t_p を更新する．t_p の値は外れ値の影響を小さくするために直

近 5 回の平均で更新する．PINGREQ による更新処理終了後，最後に PINGREQ を受信した時刻 t_r を現在時刻で，PINGREQ 間に送信したメッセージ数 m_s を 0 で更新する（図 38 ⑦）

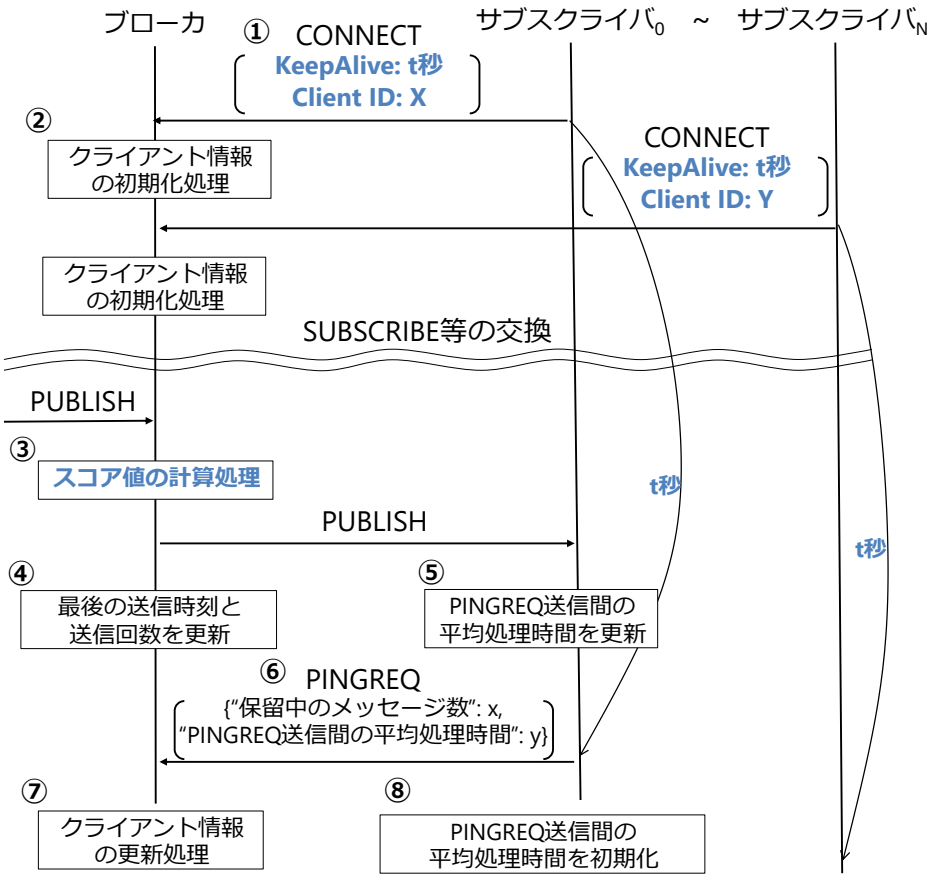

図 38　提案手法の動作概要

```
{
  "msgsInQueue": x,                  // 保留中のメッセージ数
  "processsingTimePerMsg": y         // メッセージあたりの処理時間
}
```

図 39　送信する PINGREQ のペイロードのフォーマット

第 5 章　評価

5.1　1 台の VM 上での評価

5.1.1　評価環境

本項では提案手法の評価を行うために，ローカルの VM 上で既存の静的負荷分散手法であるランダム，ラウンドロビンと提案手法の比較を行った．ランダムは乱数の値からサブスクライバを選択し，ラウンドロビンは最後にメッセージを送信した時刻が最も古いサブスクライバを選択するように実装した．評価を行ったシステム構成を図 40 に，評価環境を表 5 に示す．パブリッシャ，ブローカ，サブスクライバは全て同一の VM 上で実行した．パブリッシャ，サブスクライバは全て 1 つのプロセスからスレッドで生成され，サブスクライバは 3 台で共有サブスクリプションのグループを形成している．

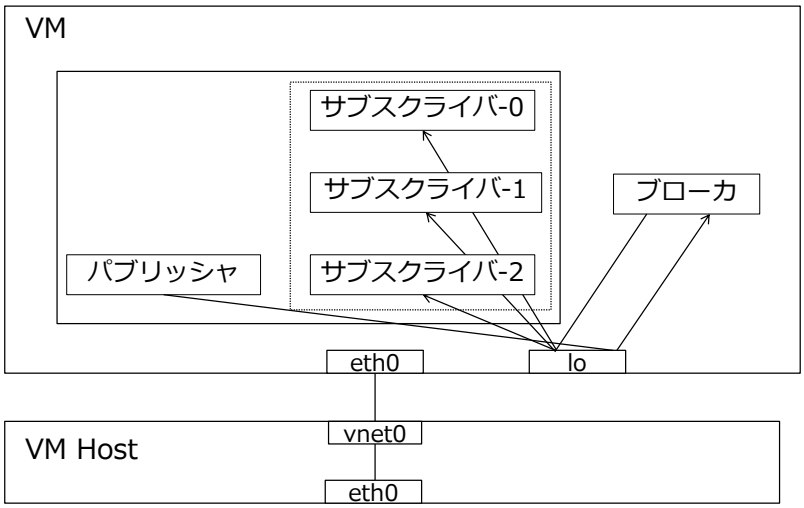

図 40　VM のシステム構成

表 5　VM の評価環境

マシン	スペック
VM Host	CPU: Intel(R) Xeon(R) CPU E5620 @2.40GHz Mem: 80GB OS: VMware ESXi 6.5U3
VM	vCPU: 8 Mem: 16GB OS: Ubuntu 20.04

評価では提案手法がサブスクライバの性能が均一な場合は既存手法と同等の性能を有しつつ，サブスクライバの性能が不均一な場合は既存手法と比較してレイテンシを抑制できることを示すために，以下に示す 2 つのシナリオを作成した．

シナリオ①

3 台の同一処理性能のサブスクライバで負荷分散

シナリオ②

3 台の処理性能に差があるサブスクライバで負荷分散

シナリオの詳細を表 6 に示す．シナリオ①はラウンドロビンでも処理が可能なシナリオである．パブリッシャは 10msec 間隔でメッセージを送信するため，ラウンドロビンの場合，サブスクライバは次のメッセージを 30msec 後に受信する（図 41）．全てのサブスクライバが 25msec でメッセージを処理可能なため，ラウンドロビンでは遅延のない負荷分散を実現できる．このシナリオで提案手法が既存手法と同等の性能を有することを示す．シナリオ②はラウンドロビンでは一部のサブスクライバで処理が滞るシナリオである．50msec でメッセージの処理を行うサブスクライバは，30msec 後に受信するメッセージまでに処理を完了することができないため，このサブスクライバではメッセージの遅延が蓄積していくことになる（図 42）．このシナリオでサブスクライバの性能が不均一な環境において提案手法が既存手法と比較してレイテンシを抑制できることを示す．評価指標にはメッセージがパブリッシャで生成されてからサブスクライバの保留中のメッセージ用のキューから取り出されるまでの時間をレイテンシとして利用した．また，どちらのシナリオでも各サブスクライバは 1 秒間隔で拡張された PINGREQ を送信するように設定した．

表 6　シナリオ詳細

共通パラメータ	
計測時間	15 秒
パブリッシャのメッセージ送信間隔	10msec
メッセージサイズ	100B
PINGREQ の送信間隔	1sec
QoS	0
シナリオ①	
サブスクライバ-0,1,2 の処理性能	25msec でメッセージを処理
シナリオ②	
サブスクライバ-0,1 の処理性能	25msec でメッセージを処理
サブスクライバ-2 の処理性能	50msec でメッセージを処理

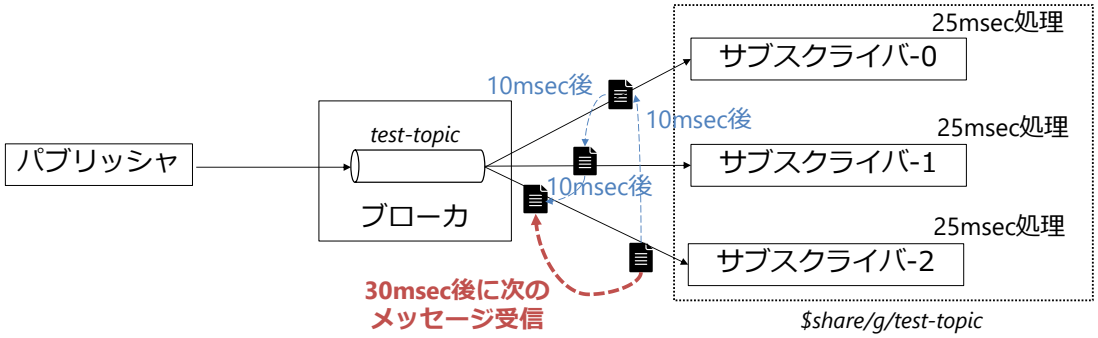

図 41　シナリオ①の概要

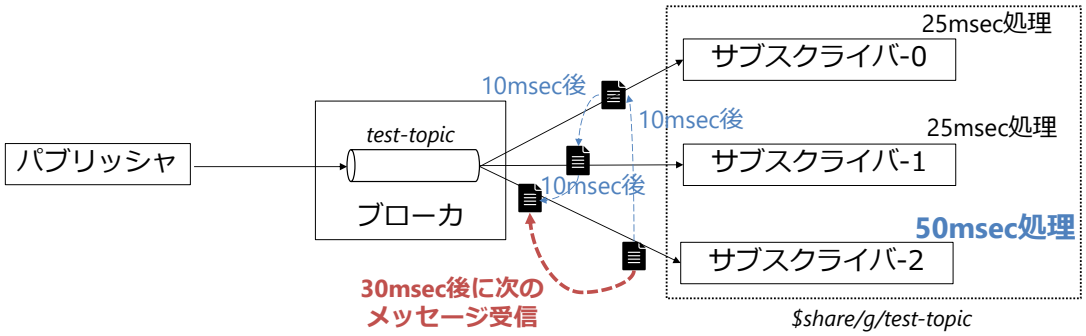

図 42　シナリオ②の概要

5.1.2　評価結果

5.1.2.1　シナリオ①の評価結果

シナリオ①の結果を図 43 に示す．図 43 はサブスクライバが受信した各メッセージのレイテンシの平均値を示している．この結果から，提案手法がラウンドロビンと同等のレイテンシを達成できていることが分かる．これは提案手法がラウンドロビンとして動作し，各サブスクライバが次のメッセージ受信までに前のメッセージを処理できているためだと考えられる．一方で，ランダムでは同一のサブスクライバにメッセージの連続した送信が発生することでメッセージ処理が滞留し，各サブスクライバでのレイテンシが増加している．

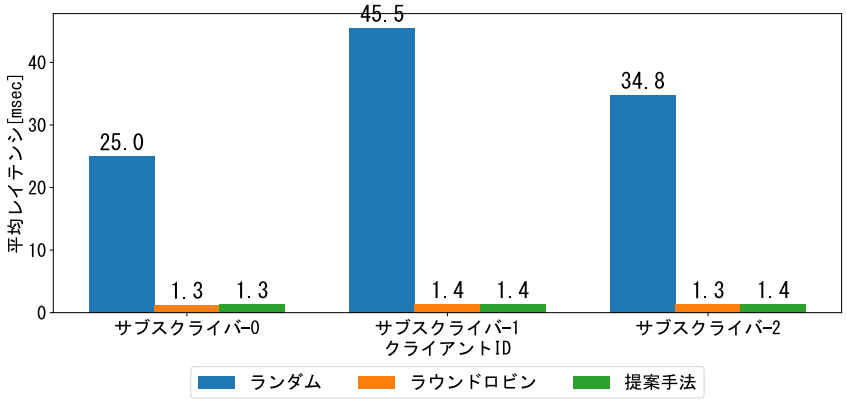

図 43　シナリオ①の各サブスクライバのレイテンシ

5.1.2.2　シナリオ②の評価結果

シナリオ②の結果を図 44 に示す．この結果から，提案手法がランダム，ラウンドロビンと比較して処理速度が遅いサブスクライバ-2 でのレイテンシを最大 98% 削減できていることが分かる．また，ワークロード全体でのレイテンシもランダムが 1400.7msec，ラウンドロビンが 1492.3msec であるのに対して，提案手法は 27.4msec と最大 98% 削減することができている．これは提案手法が PINGREQ で受信した保留中のメッセージ数とメッセージの平均処理時間から図 45 のように処理の遅いサブスクライバへの送信数を下げ，処理速度の速いサブスクライバ-0 と 1 への送信数を上げるためである．一方で，送信数が増加したサブスクライバ-0 と 1 のレイテンシはラウンドロビンと比較して最大 19.3msec 増加している．このことから，提案手法はレイテンシをグループのサブスクライバ全体で平滑化するような動作になることが分かる．

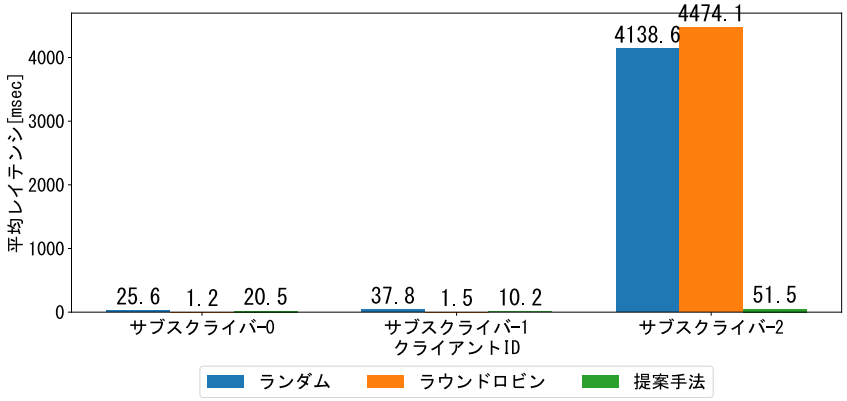

図 44　シナリオ②の各サブスクライバのレイテンシ

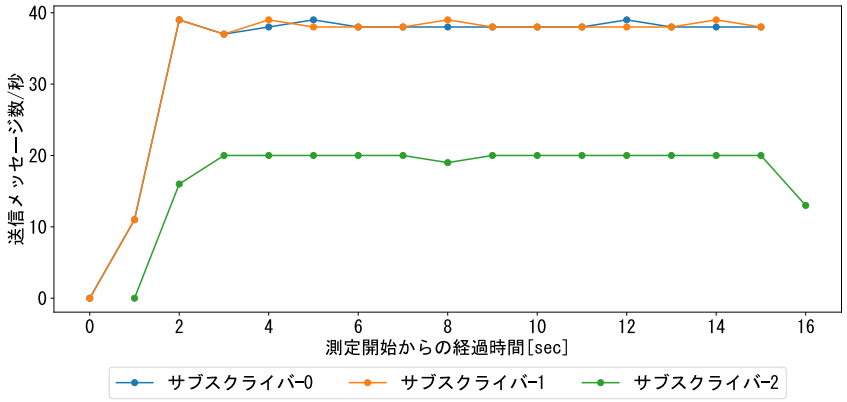

図 45　シナリオ②の提案手法での各サブスクライバへのメッセージ送信数/秒

図 44 の提案手法において，サブスクライバ-0 のレイテンシが同一性能のサブスクライバ-1 より 10msec 大きくなっている．この理由として，PINGREQ の送信タイマーはサブスクライバがブローカから CONNACK が返ってきたタイミングから起動するため，各サブスクライバの PINGREQ の送信タイミングが同期していないことが原因だと考えられる．本評価ではパブリッシャがメッセージの送信を開始する前にサブスクライバ-0 が PINGREQ 送信間の平均処理時間が 0 の PINGREQ を送信してしまい，ブローカがサブスクライバのメッセージの平均処理時間を実際の処理時間よりも短く見積もってしまっていることが図 46 から分かる．これを分析した結果を図 47 に示す．図 47 は提案手法でのメッセージ毎のレイテンシを示しており，横軸にメッセージ ID，縦軸にレイテンシを表したものである．この結果から，パブリッシャがメッセージ送信開始後 1 秒付近の 100 メッセージ前後でサブスクライバ-0 への送信過多が見られた．この影響を受けてサブスクライバ-0 と 1 のレイテンシに差が発生していると考えられる．一方で，図 47 から提案手法は 0~500 メッセージのレイテンシは不安定であるが，500~1000 メッセージのレイテンシは安定していること分かる．この不安定な区間を過渡期，安定した区間を安定期とし，安定期におけるラウンドロビンと提案手法の比較結果を表 7, 8 に示す．この結果から，安定期に入ればラウンドロビンと提案手法のレイテンシに大きな差は発生せず，サブスクライバ-0 と 1 のレイテンシにも差が発生しないことがわかる．

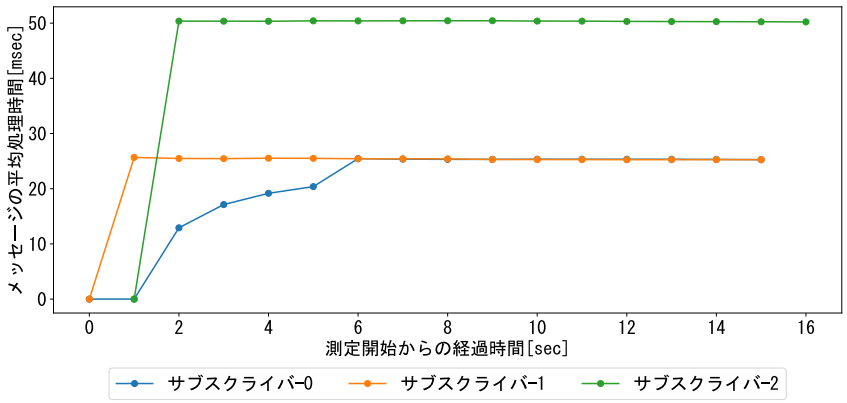

図 46　シナリオ②の提案手法での各サブスクライバのメッセージの平均処理時間

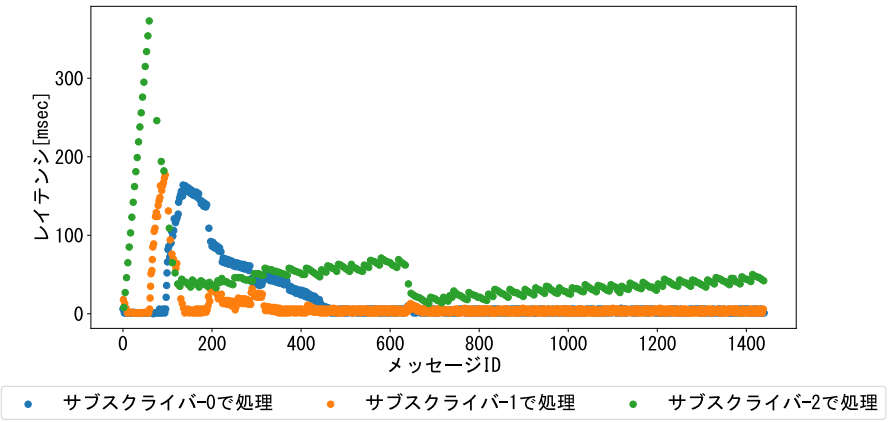

図 47　シナリオ②の提案手法でのメッセージ毎のレイテンシ

表 7　ラウンドロビンの安定期の各サブスクライバの統計情報

	処理データ数	最小 [msec]	最大 [msec]	平均 [msec]	標準偏差
サブスクライバ-0	167	0	6	0.9	0.7
サブスクライバ-1	166	0	8	1.0	0.8
サブスクライバ-2	167	3075	6250	4664.1	923.4

表 8　提案手法の安定期の各サブスクライバの統計情報

	処理データ数	最小 [msec]	最大 [msec]	平均 [msec]	標準偏差
サブスクライバ-0	199	1	10	3.4	2.4
サブスクライバ-1	199	1	13	3.7	2.5
サブスクライバ-2	102	13	71	34.8	17.8

5.2 考察

5.1 節の結果より提案手法について考察する．図 44 より，提案手法は性能が不均一なサブスクライバの共有サブスクリプションでもワークロード全体のレイテンシを抑制することができる．これは，ワークロード実行中のサブスクライバの性能低下に対しても有効であることから，性能低下を想定した過剰なリソース割り当てを回避することが可能である．また，処理性能に余裕のあるサブスクライバへの配信比率を動的に増加させる動作を行うことから，実行基盤のリソース効率を向上させることができる．一方，配信比率の増加によるレイテンシの増加をシステムが許容できるかは要件によって異なるため，導入時に検討が必要になる．

第 1 章で挙げたエッジ環境や広域分散環境での利用についても考察する．エッジ環境ではコンテナオーケストレーションツール [8] が積極的に開発されており，これらのツールの特徴である自動デプロイ機能やオートスケール機能から，実行基盤上ではデータ処理を行う複数のワークロードの実行が予想される．提案手法では多数のワークロードが実行されている環境でも，他のワークロードの影響を抑制できることから，これらの基盤上での利用に適した手法であると考えられる．

提案手法の課題として，サブスクライバの状態把握を PINGREQ に依存していることから，PINGREQ の送信間隔の間で局所的に発生する負荷上昇に対応できない点がある．PINGREQ の送信間隔には MQTT の仕様上の制限はないが，極端に短くするとネットワーク帯域の圧迫やブローカでのサブスクライバ情報の更新がスループットの低下を招く可能性がある．提案手法では PINGREQ の受信によって負荷の変動に対応していくことから，最適な送信間隔について今後検討が必要である．

第 6 章　まとめと今後の課題

本研究では，MQTT の制御パケットの 1 つである PINGREQ を拡張することで，共有サブスクリプションにおけるサブスクライバ群の状態を考慮した動的負荷分散手法を提案した．提案手法では，サブスクライバの保留中のメッセージ数とメッセージの平均処理時間からスコア値を計算することで，グループ内で最も配信に適したサブスクライバをリアルタイムに選択することができる．

評価では OSS の Mochi-MQTT Server と paho.mqtt.java を利用して実装を行い，サブスクライバの性能が均一な環境においては既存の静的負荷分散手法と同等の性能を有し，性能が不均一な環境においては既存手法と比較してレイテンシを大幅に削減できることを確認した．一方で，メッセージの平均処理時間の計算の影響で開始直後の性能が安定しないことや，PINGREQ の送信間隔の間に発生する局所的な負荷上昇に対応できない問題も残されている．これらの問題に対して，PINGREQ の送信タイミングの影響を小さくしたメッセージの平均処理時間の算出方法や，PINGREQ の送信間隔による性能変化の調査を行っていく必要がある．

今後の展望として，本研究では負荷分散の指標にサブスクライバの性能のみを利用したが，ブローカとサブスクライバとの間で発生するネットワーク遅延を考慮した指標を導入することで，より多様な環境で利用することが可能になる．また，ブローカをクラスタ構成にし，ブローカが保持するサブスクライバの状態をクラスタ間で共有することで，より広域な環境での効率的なルーティングを実現することが可能であると考える．

謝辞

本研究を進めるにあたり，広島大学情報メディア教育研究センター近堂徹教授には，日頃からの研究活動全般にわたる熱心なご指導と貴重なご助言を頂くとともに，本論文を作成するにあたり多大なるご支援を頂きました．心より感謝申し上げます．

広島大学情報メディア教育研究センター西村浩二教授，渡邉英伸准教授，岸場清悟助教，田島浩一助教，村上祐子助教，下地寛武特任助教，相原玲二上席特任学術研究員には数多くのご指導や研究方針へのご助言を頂きました．心より感謝申し上げます．

本研究を進める上で，広島大学高橋朋也さん，中野敦斗さん，森田崇大さん，浦野拓海さん，内藤岳人さん，原田啓佑さん，横尾和真さん，TWABI AHMED IMAN さん，石橋幸大さん，吉良有眞さん，黒川あゆみさん，谷口友浩さん，土居すみれさん，林鷹人さん，松下晃士さん，宮前輝大さん，安井朋輝さんには，日頃より研究室で共に活動し，本研究の議論にも積極的に寄与して頂きました．厚くお礼申し上げます．

最後に，ここまで研究を続けるにあたり，様々な面で支えて頂いた友人，そしで，常に陰ながら支えて頂いた両親・家族の皆様に心から感謝いたします．

業績リスト

1. 轟木皓平, 近堂徹，相原玲二．”Pub/Sub メッセージングシステムの耐障害性検証ツールの設計と実装”，情報処理学会 研究報告インターネットと運用技術（IOT）, 2022-IOT-58, pp.1-7, 2022.
2. Kohei Todoroki, Tohru Kondo, Reiji Aibara, “Design and Implementation of a Fault Tolerance Verification Tool for Pub/Sub Messaging Systems”, AINTEC 2022, Eikei University, Hiroshima, Dec.2022.
3. 轟木皓平, 近堂徹．”MQTT 共有サブスクリプションにおけるサブスクライバ群の状態を考慮した動的負荷分散手法”，情報処理学会 インターネットと運用技術シンポジウム (IOTS) 論文集 2023, pp.16-23, 2023.(優秀論文賞，優秀学生賞受賞)

参考文献

[1] 総務省. "IoT デバイスの急速な普及". https://www.soumu.go.jp/johotsusintokei/whitepaper/ja/r03/html/nd105220.html 2020. (参照 2023-12-19).

[2] "MQTT". https://mqtt.org/, 2019. (参照 2023-12-19).

[3] "OASIS". https://www.oasis-open.org/. (参照 2023-12-19).

[4] PATRICK TH. EUGSTER, PASCAL A. FELBER, RACHID GUERRAOUI, and ANNE-MARIE KERMARREC. "The Many Faces of Publish/Subscribe". *ACM Computing Surveys,*, Vol. 35, No. 2, pp. 114–131, 2003.

[5] Ryo Kawaguchi and Masaki Bandai. "Edge Based MQTT Broker Architecture for Geographical IoT Applications". *2020 International Conference on Information Networking (ICOIN)*, pp. 232–235, 2020.

[6] 上田 高寛, 片山 徹郎. "空間データを共有するための新たな分散 mqtt システム gamma の試作". 火の国情報シンポジウム 2022, 2022.

[7] "MQTT Version5.0". https://docs.oasis-open.org/mqtt/mqtt/v5.0/mqtt-v5.0.html, 2019. (参照 2023-12-19).

[8] "KubeEdge". https://kubeedge.io/. (参照 2023-12-19).

[9] "Red Hat OpenShift". https://www.redhat.com/ja/technologies/cloud-computing/openshift/edge-computing. (参照 2023-01-16).

[10] "EMQX Shared Subscriptions". https://www.emqx.io/docs/en/latest/configuration/mqtt.html. (参照 2023-12-19).

[11] "Hive MQ Shared Subscriptions". https://docs.hivemq.com/hivemq/latest/user-guide/shared-subscriptions.html. (参照 2023-12-19).

[12] "Apache Kafka". https://kafka.apache.org/. (参照 2023-01-16).

[13] Weisong Shi, Jie Cao, Quan Zhang, Youhuizi Li, and Lanyu Xu. "Edge Computing: Vision and Challenges". *IEEE Internet of Things Journal*, Vol. 3, No. 5, 2016.

[14] 寺西 裕一, 山中 広明, 河合 栄治. "IoT エッジコンピューティング技術". 情報通信研究機構研究報告, Vol. 64, No. 2, pp. 103–110, 2018.

[15] Muhammad Alam, João Rufino, Joaquim Castro Ferreira, Syed Hassan Ahmed, Nadir Shah, and Yuanfang Chen. "Orchestration of Microservices for IoT Using Docker and Edge Computing". *IEEE Communications Magazine*, Vol. 56, No. 9, pp. 118–123, 2018.

[16] "Microservices". https://martinfowler.com/articles/microservices.html. (参照 2024-01-16).

[17] "Docker". https://www.docker.com/ja-jp/. (参照 2024-01-16).

[18] "Docker Swarm". https://docs.docker.com/engine/swarm/. (参照 2024-01-16).

[19] "NATS". https://nats.io/. (参照 2024-01-16).

[20] 坂野遼平, 芳澤俊成. "MQTT ブローカと共有サブスクリプションを用いたスケーラブルな IoT データ収集方式の検討". 電子情報通信学会 インターネットアーキテクチャ研究会, Vol. 121, No. 201, pp. 1–5, 2021.

[21] Milica Matić, Marija Antić, Sandra Ivanović, and Ištvan Pap. "Scheduling messages within MQTT shared subscription group in the clustered cloud architecture". *2020 28th Telecommunications Forum (TELFOR)*, pp. 1–4, 2020.

[22] Ibrahim Mahmood, Siddeeq Yousif Ameen, Hajar Yasin, Naaman Omar, Shakir Fattah Kak, Zryan Najat, Azar Abid Salih, Nareen O. M. Salim, and Dindar Ahmed. "Web Server Performance Improvement Using Dynamic Load Balancing Techniques: A Review". *Asian Journal of Research in Computer Science*, Vol. 10, pp. 47–62, 2021.

[23] Deepti Sharma. "Response Time Based Balancing of Load in Web Server Clusters". *2018 7th International Conference on Reliability, Infocom Technologies and Optimization (Trends and Future Directions) (ICRITO)*, pp. 471–476, 2018.

[24] Zhong Ying Thean, Vooi Voon Yap, and Peh Chiong Teh. "Container-based MQTT Broker Cluster for Edge Computing". *2019 4th International Conference and Workshops on Recent Advances and Innovations in Engineering (ICRAIE)*, pp. 1–6, 2019.

[25] "Raspberry Pi". https://www.raspberrypi.com/. (参照 2023-01-16).

[26] "Mochi-MQTT Server". https://github.com/mochi-mqtt/server. (参照 2023-12-19).

[27] "paho.mqtt.java". https://github.com/eclipse/paho.mqtt.java. (参照 2023-12-19).