スケーラブルなMQTTブローカの実現に向けた分散連携プロキシの提案

坂野 遼平[1,2] 藤田 雅浩[3] 竹内 亨[1]

概要：IoT/M2M におけるデータ交換を支える技術として MQTT に注目が集まっている．MQTT では broker に負荷が集中することから，既存実装の中には複数の broker を連携させる機能を持つものが存在する．しかしながら，MQTT 仕様では単体 broker の動作のみを規定しているため，これらは仕様外の独自機能であり，環境に応じて異なる broker 実装を使い分けたい場合等にはスケーラビリティを得ることができない．そこで本稿では，任意の MQTT broker 同士を連携させることができる分散連携プロキシ DCP（Distributed Coordination Proxy）を提案する．DCP は client と broker の通信を中継しつつ，複数の DCP 同士で連携することで，複数 broker が論理的に 1 台の broker として振る舞うことを可能とする．DCP 間の連携方式として 2 つの方式を提案し，Java 言語にて実装したプロトタイプの設計について述べる．また，実装した DCP の性能検証のため，MQTT broker に対する汎用的なベンチマークシステムを設計した．実機環境における評価を行ない，DCP におけるオーバヘッドの確認や，配送遅延観点を交えた上記 2 方式の比較等を行なった．その中で，5 台の broker を連携させた場合に，DCP を用いない単体 broker と比べ最大でスループットをおよそ 3 倍に向上させられることを確認した．

A Proposal of Distributed Coordination Proxy for Scalable MQTT Brokers

RYOHEI BANNO[1,2] MASAHIRO FUJITA[3] SUSUMU TAKEUCHI[1]

1. はじめに

多様な IoT・M2M サービスの展開が進む中で，デバイス間のデータ交換を支える技術として MQTT [1] に注目が集まっている．MQTT は pub/sub 型の通信モデルに基づくプロトコルであり，図 1 に示すようにデータの送信者（publisher）と受信者（subscriber）をサーバ（broker）が仲介する構成をとる．

MQTT においては broker に負荷が集中することから，既存実装の中には複数の broker を連携させることで負荷を分散させる機能を持つものが存在するが，それらは実装依存の機能であり，MQTT 仕様としては単体の broker における動作のみが規定されている．このため，環境に応じて異なる broker 実装を使い分けたい場合や，付加機能やライセンス等の面から負荷分散機能を持たない broker 実装を用いたい場合に，複数台連携によるスケーラビリティ向上を行えないという問題がある．

そこで本稿では，任意の MQTT broker 同士を連携させることができる分散連携プロキシ DCP（Distributed Co-

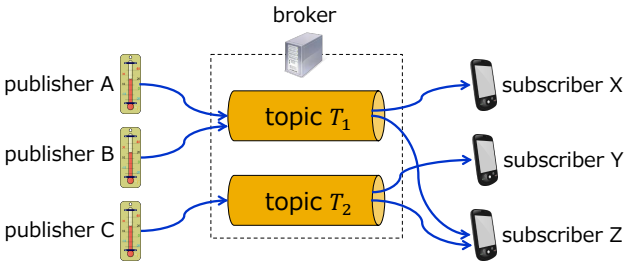

図 1 MQTT における通信の流れ

1 日本電信電話株式会社 NTT 未来ねっと研究所
NTT Network Innovation Laboratories, NTT Corporation
2 東京工業大学 大学院情報理工学研究科
Graduate School of Information Science and Engineering, Tokyo Institute of Technology
3 京都産業大学 大学院先端情報学研究科
Graduate School of Frontier Informatics, Kyoto Sangyo University

ordination Proxy）を提案する．DCP は MQTT client と MQTT broker の間に位置し，通信の中継を行なう．同時に，複数の DCP 同士が連携することで，broker 同士の連携を実現する．DCP 同士の連携には複数の方式が考えられる．本稿では Publication Flooding (PF) および Subscription Flooding (SF) の 2 方式を提案し，Java 言語による DCP のプロトタイプ実装について設計を述べる．

本論文の構成を以下に示す．まず，2 章で既存実装における複数 broker 連携について紹介する．次に，3 章にて提案手法である DCP のアーキテクチャを示した後，4 章にて DCP における 2 つの連携方式とその実装について説明する．5 章では，DCP の性能検証を行なうために構築した，MQTT broker のベンチマークシステムについて設計を述べる．6 章で実機環境における性能評価について述べ，最後に 7 章で総括する．

2. MQTT における複数 broker 連携

MQTT における broker への負荷集中の緩和や可用性向上のため，既存実装の中には複数の broker を連携させる機能を持つものが存在する．

Mosquitto [2], VernMQ [3] 等は bridge と呼ばれる機能を備えている．あらかじめコンフィグにて隣接 broker や共有するトピックの情報を指定することで，該当する publish 等のメッセージを隣接 broker へと転送させることができる．

Akane [4], HiveMQ [5] 等は cluster と呼ばれる機能を有しており，subscription 等のステータスを broker 間で動的に共有・同期することで，複数の broker が 1 台の論理的な broker として振る舞うことを可能としている．

その他，MessageSight [6] は HA 構成として 2 台の broker を協調させ，primary ノードの停止時には standby ノードが処理を継続することが可能であり，また分散エージェントプラットフォーム PIAX [7] では，オーバレイネットワーク上のノード群が仮想的に MQTT broker として動作する機能を取り入れている．

これら broker の連携は実装依存の機能であり，MQTT プロトコルとしては単体 broker の動作のみを規定している．このため，クラウドサーバとホームゲートウェイ等，環境に応じて異なる broker を利用したい場合や，付加機能やライセンス面等の要因により連携機能を持たない broker を用いたい場合に，スケーラビリティを得られない問題がある．

3. DCP のアーキテクチャ

本稿では，任意の MQTT broker 同士を連携させることができる分散連携プロキシ DCP（Distributed Coordination Proxy）を提案する．DCP を用いることで，異なる broker に収容された client 同士が，単一 broker を用いる場合と同様にデータ交換を行なうことが可能となる．DCP による複数 broker 連携の概要を，図 2 に示す．

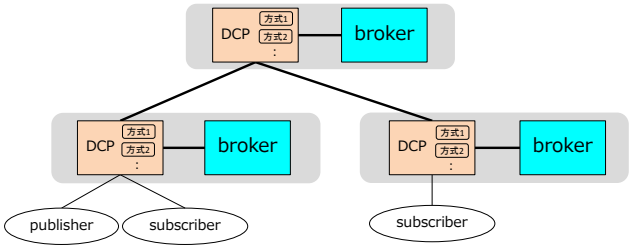

図 2 DCP による複数 broker の連携

DCP は client と broker の通信を中継しつつ，複数の DCP 同士で連携することで，複数 broker が論理的に 1 台の broker として振る舞うことを可能とする．以降では，DCP に直結している broker と client をそれぞれローカル broker，ローカル client と呼ぶ．また隣接する他の DCP をリモート DCP と呼び，リモート DCP のローカル broker やローカル client はリモート broker，リモート client と呼ぶ．

DCP を適切に連携させることにより，MQTT client を分散収容させ，全体としてのスループットを向上させることができると考えられる．DCP 間の連携には様々な方式が考えられ，各方式はプラグインとして DCP に配備し必要に応じて切り替えて利用する（図 2）．

DCP は，これら連携方式の比較基盤としても有用である．すなわち，連携機能を broker の外部に括り出すことで，方式の違いとは直接関係しない実装品質の差（メッセージ生成や subscription 管理等）を含めること無く，MQTT broker における連携方式間の比較が可能となる．

4. 連携方式と実装

本稿では，DCP における基本的な連携方式として Publication Flooding (PF) と Subscription Flooding (SF) の 2 方式を提案し，その設計について述べる．なお，これら 2 方式においては DCP はツリー状に接続されるものとし，閉路は想定しない．

4.1 PF 方式

PF 方式では，publish されたデータを DCP を介して全ての broker へ伝搬させることで，該当トピックの subscriber への配送を実現する．図 3 に動作例を示す．DCP は，subscribe メッセージについてはローカル broker とローカル client 間の中継のみを行う．publish メッセージについては，中継を行いつつ，リモート DCP へと転送を行う．転送を受けたリモート DCP は，当該 publish メッセージを自身のローカル broker へと送出しつつ，さらに隣接する他の DCP へと転送する．これにより，publish メッセージは接続されている全ての broker へと届けられ，各 broker の配下に当該トピックの subscriber がいれば，broker から

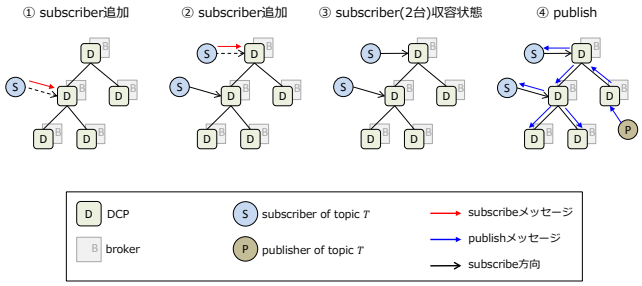

図 3　PF 方式による DCP 連携

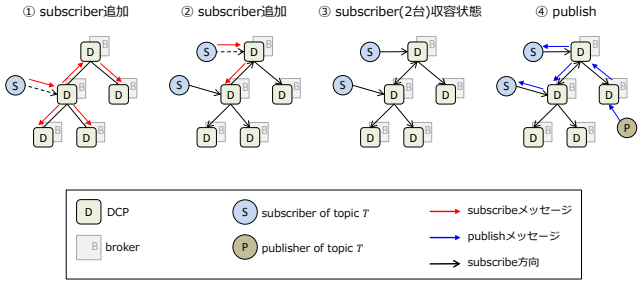

図 4　SF 方式による DCP 連携

subscriber へと転送される．

4.2　SF 方式

SF 方式では，subscription を全ての DCP へ伝搬させることで，publish されたデータを必要に応じて隣接 broker へ転送し，該当トピックの subscriber への配送を実現する．図 4 に動作例を示す．DCP は，subscribe メッセージを中継する際，当該 subscribe 情報を全てのリモート DCP へと転送する．即ち，配下に subscriber がいる場合には，DCP 自身がリモート DCP に対して当該トピックを subscribe する．このような動作を各 DCP が行うことで，subscribe 情報は接続されている全ての broker へと伝搬する．publish メッセージは，DCP 間の subscribe 関係に応じて転送されるため，当該トピックの subscriber を有さない部分木には転送されず，publish のトラフィックを最小限に抑える形となる．

4.3　連携方式による違い

PF 方式では，subscriber の有無に関係なく全ての broker に publish メッセージが伝搬するため，受け入れる publish メッセージの量という観点では単体 broker と同程度の負荷が各 broker にかかる．従って，broker が送出する publish メッセージの量が複数台の broker に分散しているか否かが，負荷分散の効率に大きく影響する．即ち，同一トピックの subscriber が多くの broker に収容されている状況下では効率が良く，逆に少数の broker にしか存在しない場合には効率が良くないと言える．

SF 方式では，publish メッセージは同じトピックの subscriber が存在している全ての broker に伝搬し，subscriber のいない部分木には伝搬しない．従って，同一トピックの publisher および subscriber が，小さな部分木に集中的に属している場合，効率が良い．一方で，DCP 内部にてローカル client の subscription 情報を管理する必要があり，PF 方式と比べ処理が複雑である．このため，特に前述のような，subscriber が多くの broker の配下に散在している状況下では，PF 方式が相対的に優位であると考えられる．

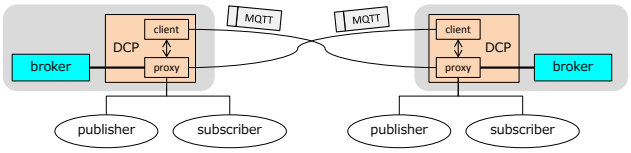

図 5　DCP 間のメッセージ交換

4.4　実装

Java 言語にて PF 方式/SF 方式を含む DCP のプロトタイプを実装した．対象とした MQTT 仕様のバージョンは 3.1.1 である．本プロトタイプにおいては，DCP 間の通信についても軽量な MQTT プロトコルを用いる形とした．図 5 に示すように，DCP は内部モジュールとして MQTT proxy と MQTT client を含み，DCP 内の client モジュールはリモート DCP の proxy モジュールを経由してリモート broker と接続する．

PF 方式においては，DCP は起動時にマルチレベルワイルドカード # をトピックとして指定しリモート broker に subscribe する．一方 SF 方式の場合，DCP はローカル client からの subscribe に連動して当該トピックをリモート broker に subscribe する．

なお，DCP 間にて相互に subscribe する関係が築かれた場合，publish メッセージのループが発生し得る．具体的には，DCP がリモート DCP から受信した publish メッセージをローカル broker へと転送し，ローカル broker が当該トピックを subscribe しているリモート DCP へと publish メッセージを転送することでループが生じる．このため，リモート DCP から publish メッセージを受信時，送信元リモート DCP の識別子とメッセージ（トピックおよびペイロード）のハッシュ値 [*1] およびタイムスタンプを記録しておき，ローカル broker から publish メッセージを受信時に重複チェックを実施することでループを検出・回避する仕組みを取り入れている．

MQTT プロトコル固有の機能として QoS レベルの設定が挙げられる．本プロトタイプ実装では，DCP 間の QoS については設定ファイルにて事前に指定した値に従う形とした．また，Will メッセージについては，通常の publish メッセージと同様の扱いにて配送可能である．なお，Retain お

[*1] MQTT 仕様におけるメッセージ ID は QoS レベルが 1 以上の時に可変ヘッダにて設定されるものであり，必ずしも常に利用できるものではないため，ハッシュ値を用いている

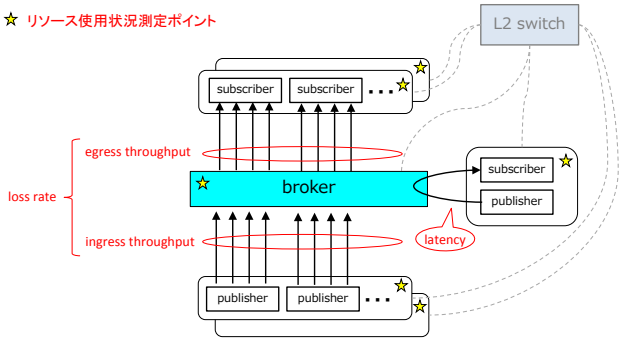

図 6　性能指標の測定構成

よび PersistentSession については，本プロトタイプでは未対応である．

5. ベンチマークシステムの設計

実装した DCP の性能検証のため，MQTT broker に対する汎用的なベンチマークシステムを設計・構築した．

5.1 概要

MQTT broker の性能指標としては，以下に記載の 4 項目を用いる．

- ingress throughput
  単位時間あたりに broker が受付可能なメッセージ数
- egress throughput
  単位時間あたりに broker が送出可能なメッセージ数
- latency
  publisher が発行した publish メッセージが subscriber に届くまでの所要時間
- loss rate
  subscriber が受信すべきメッセージ数に対する実際に受信したメッセージ数の比率

throughput については，MQTT は多対多の通信プロトコルであることから，入力側と出力側を分けて記載している．これら性能指標について測定するための構成を，図 6 に示す．

publisher および subscriber をそれぞれ複数用意し，broker に負荷をかけて ingress/egress throughput を測定する．これら publisher および subscriber の組み合わせ等の評価条件については，6 章にて述べる．また，throughput 測定用とは異なるマシン上に別途，publisher と subscriber のプロセスを立ち上げ，同一トピックを指定させることで，latency を測定する．並行して，各マシンでは CPU 使用率等のリソース使用状況を記録する．

5.2 ハードウェア環境

今回設計したベンチマークシステムでは，ノンブロッキングの L2 スイッチ配下に，表 1 に記載したスペックを有するサーバを接続して用いた．タイプ A を 10 台，タイプ B を 1 台接続し，評価条件に応じて適宜必要な台数を用いた．

表 1　サーバスペック

	タイプ A	タイプ B
processor	Atom C2750 (8core, 2.4GHz)	Xeon E5-2690V3 (12core, 2.6GHz) × 2
memory	16GB	256GB
OS	Ubuntu 14.04	Ubuntu 14.04
NW 帯域	1GbE	10GbE

なお，各サーバの省電力機能（Enhanced Intel SpeedStep Technology 等）については，意図しないパフォーマンス制御を避け，スペックとベンチマーク性能との関係性をより明らかにするために，無効化している．また，ingress/egress throughput の時間経過を重ねて可視化する等を可能とするため，同じセグメント内に NTP サーバを設置し，各サーバの時刻を同期している．

5.3 負荷ツール

publisher および subscriber として動作させる負荷ツールでは，高頻度な publish メッセージの送受信を行う必要があることから，高速な実装として知られる SurgeMQ [8] の client ライブラリを用いた．

throughput 算出のため，負荷ツールでは以下 3 点にてタイムスタンプの記録を行う．

(1) publisher における publish メッセージ送信実行時
(2) publisher における publish メッセージ送信完了時（QoS=1 では PUBACK，QoS=2 では PUBCOMP の受信時）
(3) subscriber における publish メッセージ受信時

ingress throughput については，上記 (1) または (2) により，「送信を実行したメッセージ数」もしくは「送信を完了したメッセージ数」のいずれかに基づいて，単位時間あたりのメッセージ数を算出する．egress throughput は上記 (3) から subscriber によるメッセージ受信数に基づいて算出する．

latency については，図 6 に示したように，同一マシン上の 2 つのクライアントプロセス（publisher および subscriber）にて publish メッセージの受け渡しをすることで測定を行う．具体的には，上記 (1) および (3) の差分を算出する．この時，クライアントおよびメッセージの識別子を publish メッセージのペイロードに記載することで，タイムスタンプの対応関係を判定する．

最後に，loss rate の算出について述べる．上記 (1) を元にした publish メッセージの送信数に対し，6 章にて述べる測定パターン（publisher 数，subscriber 数，トピック指定パターン）から計算することで，各 subscriber が受信すべき publish メッセージ数の和を求めることができる．この

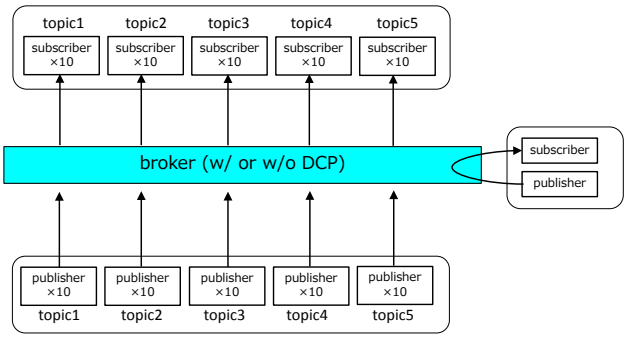

図 7　publisher および subscriber の測定構成

理論値に対し，実際に subscriber が受信した publish メッセージ数を，上記 (3) を元に計算することで，loss rate を算出する．

5.4　ボトルネックの検知

broker に負荷をかける際，負荷ツール側マシンやネットワークの処理性能がボトルネックとなっていると，broker の性能を適切に評価することができない．特に以下に挙げる事象は，broker 側での送信の抑制や再送処理を引き起こすため，考慮が必要である．

- subscriber の受信負荷過多による，TCP フロー制御
- subscriber 側ネットワーク帯域の不足による，L2 スイッチのフロー制御
- パケット損失による，TCP 再送制御

本ベンチマークシステムでは，各マシンのリソース使用状況を監視することに加え，TCP の ACK におけるウィンドウサイズや，PAUSE フレームの受信等を監視することで，上記事象の発生を検出可能としている．

6.　評価

5 章にて述べたベンチマークシステムを用いて，DCP の評価実験を実施した．

6.1　評価条件

本評価実験に共通する条件について述べる．MQTT broker としては，Mosquitto を用いた．QoS レベルは全て 0 とし，publish メッセージのペイロードサイズは 32byte とした．

publisher および subscriber については，図 7 に示す構成とした．5 つのトピックに対し，publisher と subscriber が 10 ずつ属している．従って，broker から subscriber へと送出される publish メッセージの数は，publisher から broker へと送られる数の 10 倍となる．

測定時間は 60 秒とし，1 秒に 1 回の頻度で latency 測定用の publish メッセージを送信する．throughput 測定用の publish メッセージについては，測定パターン毎に 7 種類の送信間隔（0ms, 1ms, 3ms, 5ms, 25ms, 50ms, 100ms）にてそれぞれ測定を実施し，性能上限となる送信間隔を特定する．

表 2　実験パターン毎の publish 送信間隔

パターン	概要	送信間隔
broker w/o DCP	タイプ A サーバによる単体 broker の性能評価	3msec
broker w/o DCP (HI)	タイプ B サーバによる単体 broker の性能評価	1msec
broker w/ DCP (SF)	broker1 台に DCP（SF 方式）を接続した際の性能評価	5msec
broker w/ DCP (PF)	broker1 台に DCP（PF 方式）を接続した際の性能評価	5msec
5 brokers w/ DCPs (SF)	broker5 台にそれぞれ DCP（SF 方式）を接続し連携させた際の性能評価	1msec

性能上限の判断基準としては，ingress throughput と egress throughput の比率を用いる．前述のように，本実験設定では broker において publish メッセージの量が 10 倍に増加するため，egress throughput が ingress throughput の 10 倍を下回っている場合，処理過多により broker 内部に滞留するメッセージが増加していく状況であると考えられる．そこで，誤差を考慮した下記の条件式を満たす最小の publish 送信間隔を求め，当該送信間隔における throughput を性能上限とする．

$$\frac{< egress\ throughput >}{< ingress\ throughput > \times 10} \geq 0.99 \qquad (1)$$

6.2　実験結果

5 つのパターンについて，前述の条件設定にて性能上限となる送信間隔を確認した（表 2）．

なお，表 1 に記載の 2 種類のサーバについて，broker w/o DCP (HI) における broker にのみタイプ B を用い，その他は全てタイプ A を用いた．また，5 brokers w/ DCPs (SF) においては，5 台の broker および DCP がリスト状に連なるトポロジとし，通信の局所性が大きい状況を模して，同一トピックの client がそれぞれ同一 broker の配下に収容される配置にて実験を行った．

表 2 の送信間隔にて各パターンの ingress/egress throughput および latency を測定した結果を，図 8 から図 12 に示す．throughput および latency のいずれも，値の変動はあるものの全体として増加あるいは減少の傾向は見られず，測定時間を通して一定程度の値を示している．latency については，DCP を用いることで値の変動が大きくなっている様子が見受けられる．なお，これらの測定において，loss rate はいずれも 0%であった．

続いて，評価指標ごとに各実験パターンの平均値および標準偏差を算出したものを図 13 から図 15 に示す．いずれも，単体 broker については性能が高いタイプ B においてパフォーマンスが向上しており，また DCP を用いることでオーバヘッドが生じてパフォーマンスが下がっていることがわかる．単体 broker に対しての適用については，

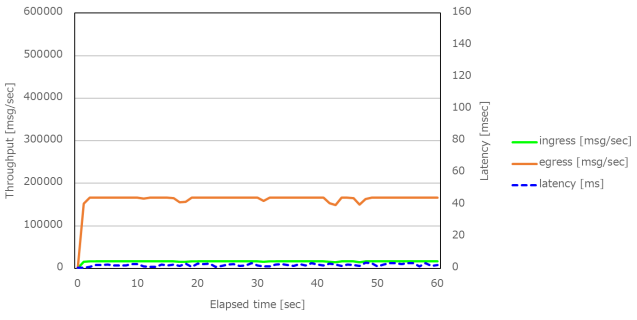

図 8　単体 broker の性能

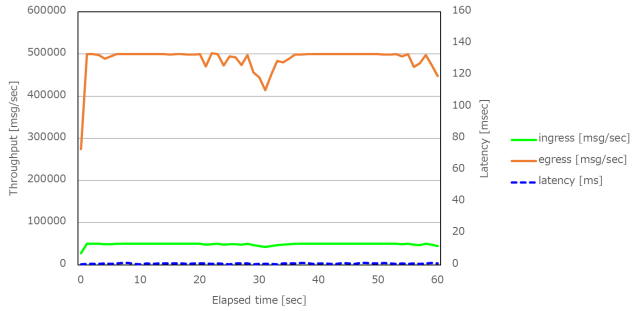

図 9　高性能サーバにおける単体 broker の性能

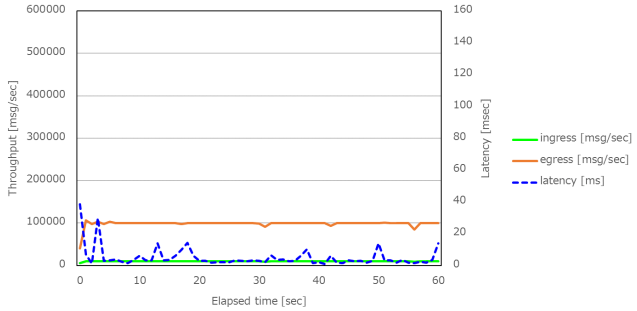

図 10　DCP（PF 方式）を適用した単体 broker の性能

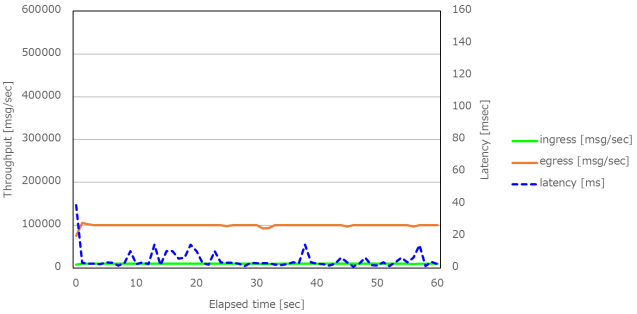

図 11　DCP（SF 方式）を適用した単体 broker の性能

SF 方式と PF 方式に大きな差は見られない結果となっている．また，DCP を用いて 5 台の broker を連携させることで，DCP を用いない単体 broker と比べ ingress/egress throughput がいずれも約 3 倍に向上している．これは，タイプ B サーバにおける単体 broker に匹敵する throughput 性能である．一方で，複数 broker を連携させることでホップ数が増加するため，latency については上昇する結果となっている．

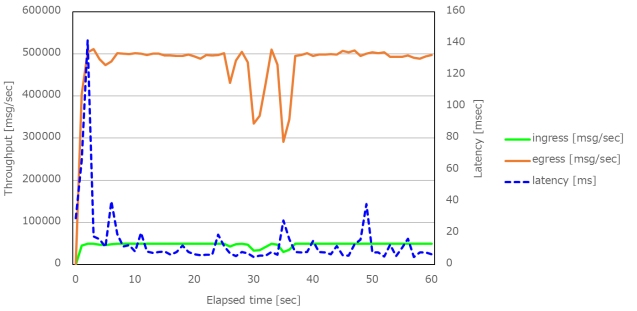

図 12　DCP（SF 方式）により 5 台の broker を連携させた際の性能

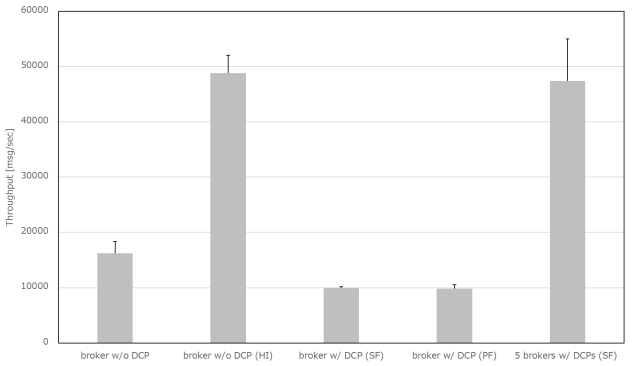

図 13　ingress throughput

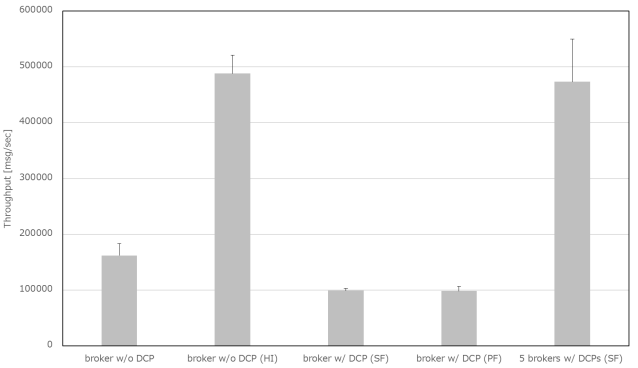

図 14　egress throughput

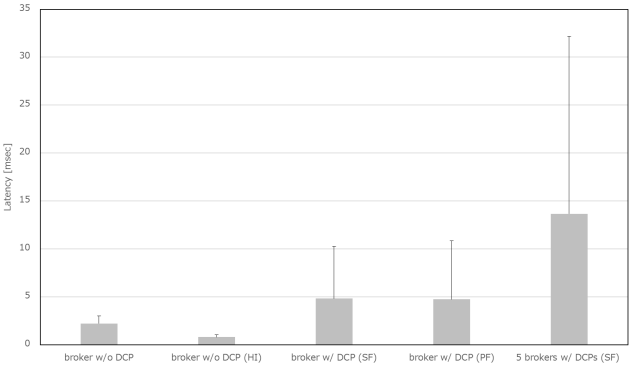

図 15　latency

7.　おわりに

本稿では，任意の MQTT broker を連携させスケーラビ

リティを得ることができる DCP のアーキテクチャを提案し，DCP 間の連携方法として PF 方式および SF 方式の 2 種類の設計を示した．Java 言語にて DCP のプロトタイプを実装するとともに，ベンチマークシステムを構築し，DCP のオーバヘッドの確認および複数 broker の連携による負荷分散効果の評価を行った．

DCP は MQTT broker における連携機能を外部に括り出したものであり，連携方式同士を公正に比較するための基盤としても有用である．また，異種 broker を連携可能であるため，サービスベンダー同士のデータ共用ハブとして位置付けることもでき，ミニマルな M2M platform とも捉えることができる．

今後の課題としては，retain や persistent session といった MQTT 機能の実現や，DCP 間の効率的な通信プロトコルの設計，障害時への対策等が挙げられる．

参考文献

[1] MQTT: `mqtt.org` (accessed 2016-05-08).
[2] Mosquitto: `mosquito.org` (accessed 2016-05-08).
[3] VernMQ: `vern.mq` (accessed 2016-05-08).
[4] Akane: `akane.shiguredo.jp` (accessed 2016-05-08).
[5] HiveMQ: `www.hivemq.com` (accessed 2016-05-08).
[6] MessageSight: `www-03.ibm.com/software/products/en/messagesight` (accessed 2016-05-08).
[7] 寺西裕一：MQTT on PIAX, 第 8 回広域センサネットワークとオーバレイネットワークに関するワークショップ (2016).
[8] SurgeMQ: `zhen.org/categories/surgemq/` (accessed 2016-05-08).