東京大学大学院　情報理工学系研究科
電子情報学専攻
修士論文

WebSocket 通信における
緊急メッセージの優先配送手法

−Method of Rapid Delivery of Prioritized Message

over WebSocket Protocol−

48 – 116449
李　聖年
SeongNyeon Lee

指導教員　江崎 浩　教授

2016 年 2 月

概要

近年，リアルタイムの双方向通信を実現する WebSocket 通信が注目を集めている．WebSocket 通信は，IoT(Internet of Things)と呼ばれる多種・多用な従来のコンピュータ以外のディジタル聞きの間で直接通信を行う M2M(Machine-to-Machine)の分野における活用が期待されており，特に，サーバからクライアントへ直接メッセージを送る PUSH 通知が提供されている点が評価されている．また，M2M においては，安価で容易に導入可能な 3G 回線の利用が増えている．しかし，安価な 3G 回線は，帯域に制限がある場合が少なくなく，WebSocket をそのまま用いた場合，大きいサイズのメッセージを送信する時や通信回線が混雑・輻輳している時には緊急的なメッセージをすぐに送信できない状況が発生してしまう．そこで，本研究では，WebSocket 通信において小さい遅延で送信すべき緊急メッセージを優先的に配送するシステムアーキテクチャを提案・実装し，メッセージの遅延特性やオーバーヘッドの評価を行い，その有効性を検証した．

Abstract

WebSocket, which can deliver full-duplex real-time communication, is becoming popular. WebSocket is expected to be used for Machine-to-Machine communication(M2M), which will be widely used in the IoT(Internet of Things) systems accommodating a lot of and wide variety of non-computer digital devices and it is highly evaluated the direct push notification from server node to client node. And, in the IoT system, people start to use 3G wireless network, because of it's less communication cost and easier system configuration. However, some 3G service have restriction on the available bandwidth M2M communication. As a result, WebSocket communication may have a problem that the prioritized messages, that should be transmitted with smaller delay, may experience large delay due to the head of line blocking by some big message or network congestion by heavy traffic load. In this paper, we propose and implement a mechanism and system that delivers prioritized message transmission and evaluate the delay of messages and system overhead so as to show the effectiveness of the proposed system.

Keywords M2M(Machine to Machine Communication)， Internet of Things(IoT)， WebSocket，Cellular Network, MVNO，Bi-Directional Full-Duplex Communication, HTML5, Server Push，QoS，Priority Control，Narrow Link，Single Session，Queueing，TCP Head of Line Blocking，MSS(The Maximum Segment Size)

目次

概要 I

目次 II

図目次 IV

第１章　序論 1

1.1 本研究の背景 1
1.2 本研究の目的 5
1.3 本研究の構成 5

第２章　関連研究 7

2.1 SSE: SERVER SENT EVENT 7
2.2 MQTT: MQ TELEMETRY TRANSPORT 10

第３章　提案手法 17

3.1 システム構成 17
3.2 PQ-PC: USE PRIORITIZED QUEUE FOR PRIORITY CONTROL 19
3.3 DMPQ-PC: DIVIDE A MESSAGE AND PQ-PC 21

第４章　パフォーマンス・モデル 27

4.1 従来手法 27
4.2 PQ-PC 29
4.3 DMPQ-PC 30

第５章　評価実験 32

5.1 実験の詳細 32
5.1.1 実験の構成や仕様 *32*
5.1.2 実験環境の構築 *33*
5.1.3 実験パラメータ *34*
5.2 ファイルの実装 34
5.2.1 従来手法 *34*
5.2.2 PQ-PC *36*
5.2.3 DMPQ-PC *37*

5.3 配送遅延の評価 37
5.3.1 手順 *37*
5.3.2 結果 *38*
5.3.3 評価 *42*
5.4 通信量・パケット数の評価 44
5.4.1 手順 *44*
5.4.2 結果 *44*
5.4.3 評価 *47*
5.5 分割による遅延抑制の評価 49
5.5.1 手順 *49*
5.5.2 結果 *49*
5.5.3 評価 *53*

第 6 章　考察 54

6.1 クライアント・サーバの双方実装 54
6.2 ラベルの導入 54
6.3 実際のマシーンとの相違点 55
6.4 ポートを用いた優先制御 55
6.5 TCP バッファの利用 56
6.6 ETHERNET JUMBO FRAME 56
6.7 他の通信による影響 56
6.8 パケットロスによる待機遅延の改善 56
6.9 優先度の管理 57
6.10 SPDY との比較 57

第 7 章　結論 59

発表文献と研究活動 60

参考文献 61

謝辞 64

付録 65

図目次

図 1.1 従来の HTTP 通信 1
図 1.2 サーバからの PUSH 通知 2
図 1.3 WEBSOCKET の通信 2
図 1.4 TCP に依存した WEBSOCKET の順次的配送 3
図 1.5 WEBSOCKET 通信の緊急メッセージの配送遅延 4

図 2.1 SERVER SENT EVENT の動作 7
図 2.2 POLLING 方式のサーバ通知 8
図 2.3 LONG POLLING 方式のサーバ通知 9
図 2.4 MQTT の動作 11
図 2.5 MQTT の WILL 13
図 2.6 MQTT の RETAIN 13
図 2.7 MQTT の CLEAN SESSION 14
図 2.8 MQTT のメッセージの階層型トピック 14
図 2.9 MQTT の QUALITY OF SERVICE 15

図 3.1 システム構成 17
図 3.2 WEBSOCKET 通信のヘッダー 19
図 3.3 キューイングを用いた手法 PQ-PC 19
図 3.4 QUEUE CONTROL の手順 20
図 3.5 回線の混雑による遅延の改善や分割によるオーバーヘッド 21
図 3.6 キューイングと分割を用いた手法 DMPQ-PC 22
図 3.7 分割メッセージにラベルをつけるプロセス 22
図 3.8 クライアントにおけるメッセージ処理 24
図 3.9 分割による遅延の抑制効果 25
図 3.10 メッセージの断片化やパケットの利用効率 26

図 4.1 PM 配送遅延の構成 28

図 5.1 評価実験の構成 32
図 5.2 PM と SM の構造 35
図 5.3 メソッド PC_FLUSH の動作 36

図 5.4 PM の配送遅延の結果 38
図 5.5 PM の配送遅延の累積分布(10[KBPS]) 39
図 5.6 PM の配送遅延の累積分布(11[KBPS]) 39
図 5.7 PM の配送遅延の累積分布(13[KBPS]) 40
図 5.8 PM の配送遅延の累積分布(15[KBPS]) 40
図 5.9 PM の配送遅延の累積分布(17[KBPS]) 41
図 5.10 PM の配送遅延の累積分布(20[KBPS]) 41
図 5.11 通信量の比較 45
図 5.12 パケット数の比較 46
図 5.13 異なる DV 値における PM の配送遅延 50
図 5.14 DMPQ-PC における PM の配送遅延の累積分布(7[KBPS]) 51
図 5.15 DMPQ-PC における PM の配送遅延の累積分布(9[KBPS]) 51
図 5.16 DMPQ-PC における PM の配送遅延の累積分布(11[KBPS]) 52
図 5.17 DMPQ-PC における PM の配送遅延の累積分布(15[KBPS]) 52
図 5.18 DMPQ-PC における PM の配送遅延の累積分布(20[KBPS]) 53

図 6.1 二つのポートを用いたアプローチ 55

第1章　序論

1.1 本研究の背景

HTTP 通信の制約と PUSH 通知

従来の HTTP 通信のプロトコル[1]は，以下の図 1.1 のようにクライアントのリクエストが起点になる仕組みで作られたプロトコルであり，Web が登場した当時はこのような方式でもユーザの需要に十分対応してきた．一方，スマートフォンの普及に伴う大きいモバイル環境の拡散によってユーザの需要は様々な形に増え，これは Rich Interactive Application(RIA)[2]の発展に繋がった．そして，その様々なユーザの需要の一つがサーバからの PUSH 通知である．

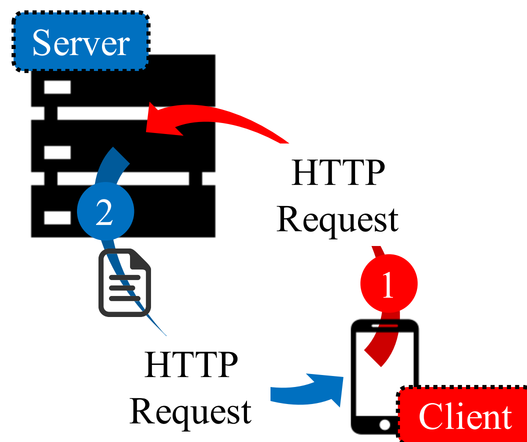


図 1.1 従来の HTTP 通信

PUSH 通知とは，以下の図 1.2 のように，クライアントからのリクエストがない時でも，サーバ側においてある特定のイベントが起きた際に，これをサーバから知らせてくれることを意味する[3]．代表的な例としては，防災速報や株価の更新など，サーバでデータの分析が行われ，その結果が直ちにクライアントへ通知される必要のあるサービスが挙げられる[4][5].

一方，既存の HTTP 通信の仕組みにおいては，サーバから開始する通信を含めた双方向通信は考えられていなかったため[6][7]，この PUSH 通知を実現するためには別の工夫が必要であった．しかし，その結果として生まれた様々なプロトコルは，従来の「クライアント起点」という制約から脱却できず，HTTP ヘッダーによる通信量のオーバーヘッドが発生することや，多数のセッションが必要であることなど，ネットワークリソースの利用効率が低下する課題を抱えた状態である[8]．また，このような課題はショートパケットを頻繁にやり取りする Machine to Machine/Internet of Things(M2M/IoT)環境においてより大きい問題となる[9][10].

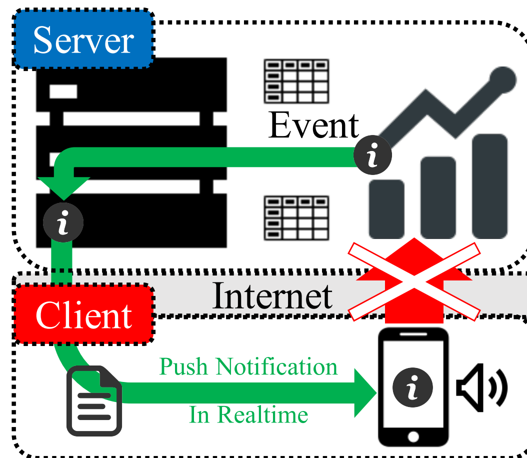


図 1.2 サーバからの PUSH 通知

WebSocket 通信の特徴

上に述べたような背景から生まれた WebSocket[11]は，根本的に HTTP 通信のやり方を捨てた新しいプロトコルで，以下の図 1.3 のように，クライアントとサーバがお互いに全二重双方向(Full-duplex & Bi-directional)でリアルタイム通信が可能であることから，近年，注目を集めているプロトコルである．

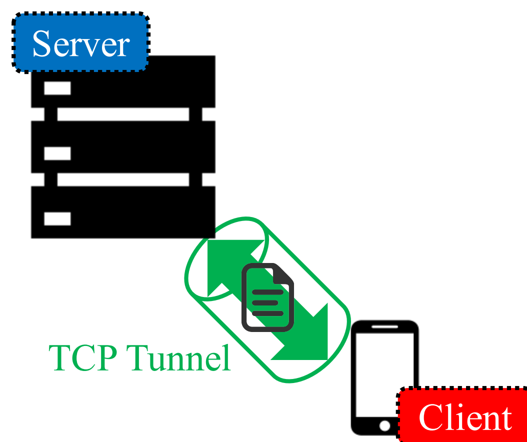


図 1.3 WebSocket の通信

この WebSocket は，HTTP 通信に沿った形のリクエストから接続を開始し，いったん接続が開始されたらその接続を維持したまま，TCP トンネルを通して継続的にメッセージのやり取りを行う．その結果，HTTP ヘッダーが途中では発生せずにその分通信のパフォーマンスが高い[12]．また，クライアントとサーバが双方向に通信を行うための TCP セッション一つのみであるため，全体的にネットワークリソースを効率的に利用しながら PUSH 通知が実現できる技術といえる．

このような WebSocket は，接続を終了せずにメッセージのやり取りが行われることから頻繁にメッセージをやり取りしても比較的に負荷が少なく，ヘッダーが非常に小さいことから短いメッセージを運ぶのに有利である．このような二つのメリットから，M2M/IoT 環境において有効的に活用できる技術であり[13]，スマートメーターと電力会社間やセンサ・アクチュエータ間の通信における活用が想定できる[14].

M2M/IoT における 3G 回線の活用

従来のインターネットの世界では，人と人，あるいは人とサーバとを繋ぐ情報交換ネットワークを実現してきた．これが更に発展し，自動車や家電など，現実世界のあらゆるものが，人の手を介することなくネットワークに繋がる M2M の世界の到来が期待されている[15]．物と物とがネットワークを用いて繋がりあう仕組みが M2M であるが，実世界上のあらゆる物がインターネットに接続される IoT も，同様の意味で用いられている[16].

近年，このような M2M/IoT 環境において，3G などの Celullar Network の活用が増えており[17][18][19]，特に MVNO で代表される安価な回線の登場が，このような動きを加速化させている[20][21]．関連するデバイスが設置される場所は至るところであり，将来的にその数が 500 億にも及ぶと予想される[22]．M2M/IoT 業界にとって，Celullar Network の「広いカバー範囲」・「簡単な導入」・「低いコスト」は，非常に魅力的な要素であるためである[23]．しかし，このような安価な回線は帯域に制限がある場合が多く[24]，TCP セッションの数が増えるとその分多く制限がかかる可能性もある．その結果，「回線の数や速度」と「導入や運用のコスト」がトレードオフの関係になり，コストを安価に抑えるためには，回線の速度が遅い状況でも正常に運用ができるシステムにする工夫が必要となる.

WebSocket 通信の課題

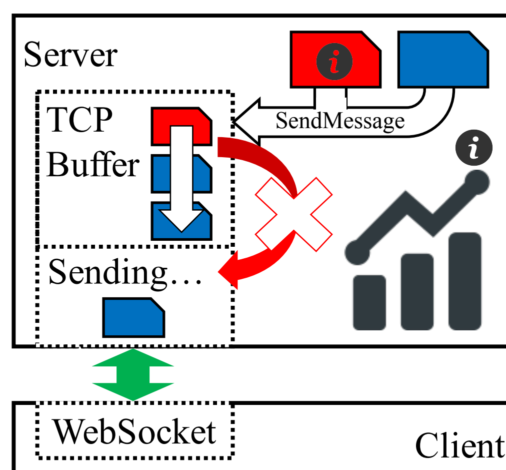


図 1.4 TCP に依存した WebSocket の順次的配送

通常，WebSocket の実装[25]では，以上の図 1.4 のように，サーバ側のアプリケーションが生成して送信したメッセージがすぐに送出されず，バッファに蓄積されて順次に送出されていく．このように，TCP 通信の特性に依存し，メッセージを優先的に送る仕組みが存在しないことにより，緊急的なメッセージが発生する状況において，その活用の幅が狭まると思われる[26][27].

そのような状況を具体的に想定すると，次のようである．以下の図 1.5 のように，家庭内のダッシュボードが家の内部や外部との情報を WebSocket 通信でやり取りしていると想定する．そうすると，WebSocket 通信におけるサーバ側は交通会社・気象庁・電力会社などで，クライアント側はダッシュボードとなる．この時，地震速報のような一刻を争う緊急的なイベントがサーバ側において予測されたとしても，回線が大量のメッセージで混雑したり大きいメッセージで詰まっていたりしてバッファ長が長い時は，緊急メッセージでもその分まで待機してしまう．

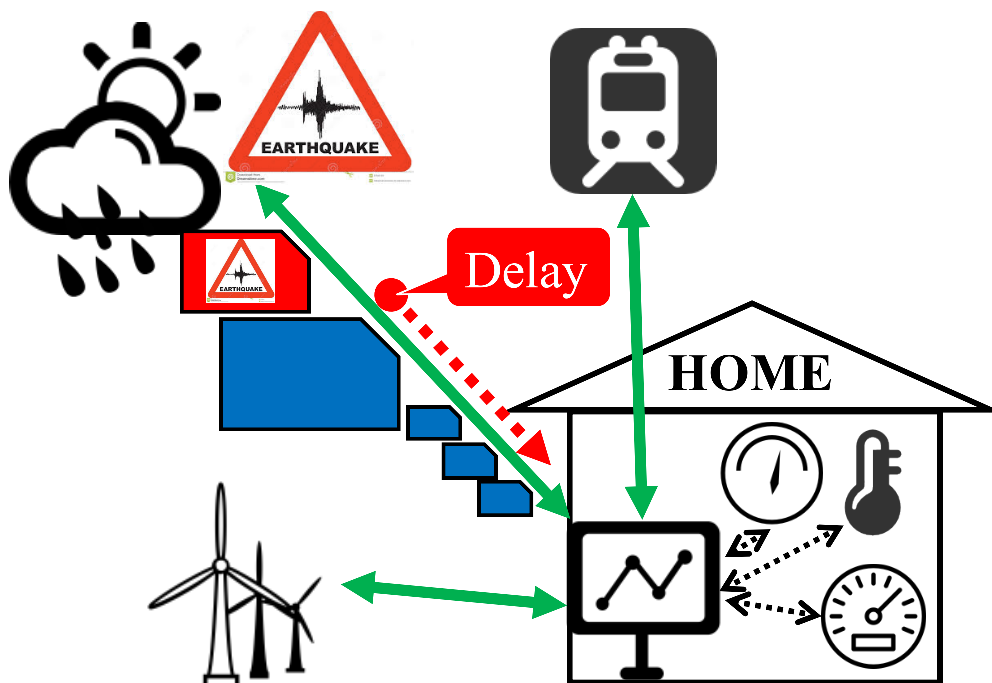


図 1.5 WebSocket 通信の緊急メッセージの配送遅延

コストを抑えるために MVNO などの帯域速度に制限のある回線を使った場合，こういった遅延がさらに長くなるため，WebSocket 通信においては緊急メッセージリアルタイム性を失わないためには回線の帯域速度が保証されるべく，その分高いコストがかかる．一般的な状況ではなく緊急的な状況を想定して回線を確保しているため，普段の回線使用率が低下してしまう．特に，緊急メッセージがほぼ遅延なく送信できるほど小さいメッセージであるとしたら，より非効率的な設計となってしまう．すなわち，WebSocket 通信で緊急的なメッセージの優先配送が可能になれば，今よりその活用の幅が広がると思われる．

1.2 本研究の目的

本研究の最終的な目的は，WebSocket を用いたクライアント・サーバ間の通信において突発的に発生する緊急メッセージの優先配送を可能にすることで，より多くの場面に WebSocket 通信が活用できるようにその柔軟性を高めることにある．その時，帯域や TCP セッションのネットワークリソースをできるだけ有効的に活用し，コストを最大限に抑えた環境を想定する．本研究では，このような目的を果たすための仕組みを二つ提案し，その評価を行う．

一つ目の提案は，サーバ側においてメッセージを送出する仕組みを工夫したものであり，クライアント側は変化させる必要のないことが特徴である．具体的には，サーバ側のアプリケーションが生成したメッセージを優先度によって別々にキューイングし，優先度の高いキューのメッセージから送出する仕組みである．そのために，制御が難しい TCP バッファではなく違う場所にメッセージをキューイングし，帯域が空いた後にメッセージの選定して TCP バッファに渡す制御を行う．

二つ目の提案は，クライアント・サーバの双方に工夫を行ったものであり，一つ目の提案にさらなる処理を加えた仕組みである．具体的には，サーバ側でメッセージをキューイングする前に分割・ラベル追加の作業を行い，クライアントはメッセージのラベルによって異なる処理を行う仕組みである．

本研究の評価のために，サーバのアプリケーションからメッセージを周期的に発生させ，狭帯域回線で WebSocket 通信を行う．このような環境において，サーバのアプリケーションからランダムに発生するメッセージの遅延を求め，提案手法による抑制効果を評価する．また，通信量やパケット数から提案手法によるオーバーヘッドを求め，これを効率的に抑えられたかについての評価も行う．

1.3 本研究の構成

第 1 章 序論
第 2 章 関連研究
第 3 章 提案手法
第 4 章 パフォーマンス予測
第 5 章 評価実験
第 6 章 考察
第 7 章 結論

本論文は，以上の各章により構成される．第 1 章では，本研究の背景，目的，構成について示す．第 2 章では，サーバ通知を実現する Server Sent Event や双方向通信の MQTT について紹介し，WebSocket との相違点から提案手法の必要性を明らかにした．第 3 章では，キューイン

グやメッセージ分割を用いて緊急メッセージの配送遅延の最大値を抑制する，二つの手法を提案する．第 4 章では，提案手法のパフォーマンスを予測するために，従来手法や提案手法における緊急メッセージの配送遅延の理論的モデルを，いくつかの条件のもとで近似式で表す．第 5 章では，緊急メッセージの配送遅延や各手法における通信量・パケット数を測定する．また，実験の詳細，結果，そして結果の評価を述べる．第 6 章では，本研究に関する考察や今後の課題について述べ，第 7 章で本論文の内容をまとめる．

第2章　関連研究

2.1 SSE: Server Sent Event

WebSocket 以外にも，サーバからの PUSH 通知を実現できる方式はいくつかある．その中でも，Ajax や Comet という技術を応用して改良を行った Server Sent Event(SSE)が代表的な例である[28][29][30]．その名前からわかるように，サーバ側において起きたイベントを通知することを主な目的としており，HTML5 に含まれている仕様である[31]．これは，「クライアント起点」という HTTP の制約をそのまま持っていながらも，リアルタイムで通知が可能であり，以前の技術と比べてオーバーヘッドも軽減させた．

SSE の通信

以下の図 2.1 は，Server Sent Event の通信手順を示したものである．以下においてその具体的な手順を示し，どのようにサーバから PUSH 通知を送るかその動作原理を説明する．

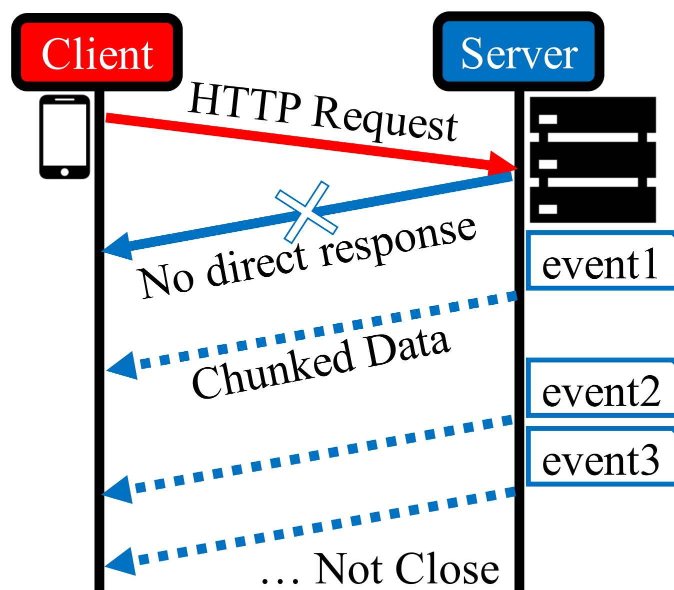


図 2.1 Server Sent Event の動作

(1) クライアントは，従来と同様にサーバへ HTTP リクエストを送る．
(2) サーバは，このリクエストに対してすぐにレスポンスをせずに，待機状態でいる．
(3) サーバ側においてイベントが発生したら，最初のリクエストに対したレスポンスとしてクライアントへ通知を送る．
(4) 接続を終了せずに維持したまま，イベントが起きる度に(3)の動作を繰り返す．

SSE の特徴

この Server Sent Event は，HTTP 通信のやり方を大きく変えずに，少しだけ工夫を加えてサーバ通知を実現した技術である．Server Sent Event における従来の HTTP 通信との相違点は，以下の二つである．

(1) リクエストが来た直後ではなく，イベントが起きた直後にレスポンスをする．

(2) イベントが起き，通知を行うためにレスポンスを送った後にも，接続を維持させる．

最初に，リクエストに対してすぐにレスポンスをしないことは，「リアルタイム」で通知を行うことを目的とした工夫である．以下の図 2.2 のように，従来のすぐにレスポンスを返す方式を Polling 方式といい，この方式ではクライアントがサーバの状況をアップデートするために周期的にレスポンスを送る必要がある．その結果，サーバがリクエストを返した直後に発生したイベントは，次のクライアントからのレスポンスが届くまで通知されなくなってしまう．

そこで，レスポンスを送るタイミングをリクエストが来た時点ではなくイベントが起きた時点とし，イベントの発生と通知を一致させた．

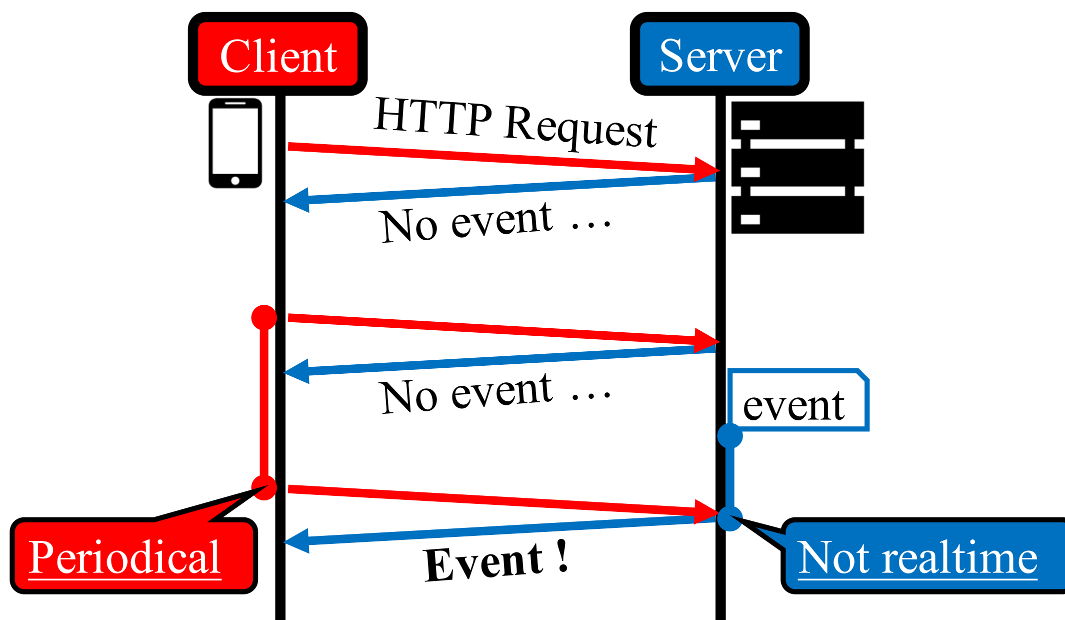


図 2.2 Polling 方式のサーバ通知

次に，通知を含めたレスポンスを送った後にも接続を維持させる目的や，そのために行われた工夫を述べる．以下の図 2.3 のように，イベントが起きる度にサーバがレスポンスを送って接続が終了・再開される方式を Long Polling 方式といい，この方式で通知が起きる度にレスポンスと新たなリクエストが発生する．Server Sent Event で接続を持続させることは，このクライアントからのリクエストの数を減らすことや，同時接続によるサーバにかかる負担を少なくすることを目的とした工夫である．

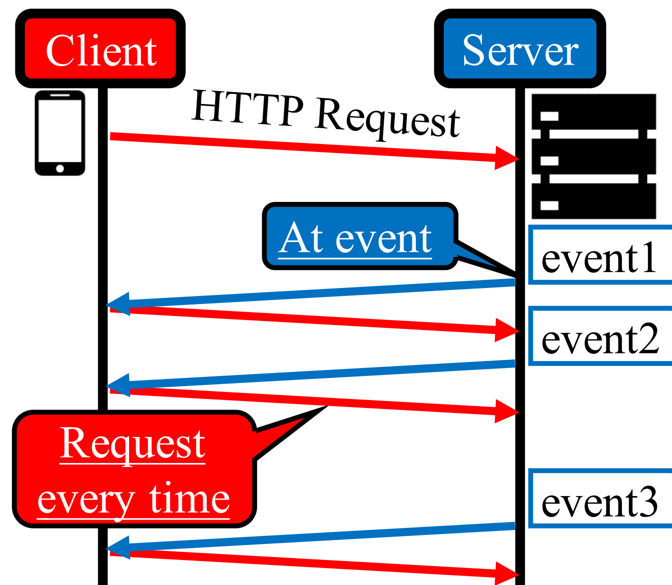


図 2.3 Long Polling 方式のサーバ通知

もし，M2M のように頻繁に通知が行われるような状況であれば，その度に発生するリクエストは，通信量が多くなってしまう原因となる．さらに，メッセージのサイズが小さいショートパケットなどの場合は，ヘッダーの通信量の合計がメッセージの通信量の合計を超えてしまうことも想定され，通信全体のパフォーマンスが落ちてしまう．

また，接続の開始・終了が多発すると，サーバ側にかかる負荷も増加してしまう．通常，一つのサーバ側に接続されているクライアントの数は複数存在しており，C10K 問題などが挙げられている[32]．そのような環境においてクライアントが毎回接続を更新するとしたら，通知が終わった直後に多数のクライアントが一斉にサーバ側に接続を要求してくることになり，これはサーバ側にとって大きな負荷となってしまう．

そこで，Server Sent Event では，クライアントとの接続を終了させずに維持したまま，その接続をサーバからの通知のみに用いることにより，レスポンスのヘッダーによる通信量のオーバーヘッドを減少させ，同時接続によるサーバ側の負荷も軽減させている．

SSE vs WebSocket

上に述べたように，Server Sent Event は既存に挙げられていた「リアルタイム性」やオーバーヘッド・負荷」といった課題を解決し，サーバの PUSH 通知を実現した．さらに，従来の HTTP のやり方に沿った通信であるため，通信の互換性が高く，従来のセキュリティモデルをそのまま使うことができるとの利点を持っている．

しかし，HTTP のやり方を完全に捨ててなかったことにより，ある程度の制約も存在したままになってしまい，以下の三つの理由から，Server Sent Event が WebSocket より有効的ではない

場合が存在する．

(1) 最大 2 倍の TCP セッションが消費される．

(2) HTTP レスポンスのヘッダーは比較的に重い．

(3) クライアント起点が前提なので，双方向の実装が異なる．

最初に，クライアントからサーバにメッセージを送るためにもう一つの TCP セッションが必要となることである．一つの TCP セッションで常に全二重・双方向のメッセージのやり取りが可能である WebSocket と比較したら，TCP セッションというネットワークリソースを最大で二倍を消費してしまうことを意味し，そのままコストが上昇する原因となる．その結果，多くのクライアントなどが理由で，よりコストを抑えたい環境においては，WebSocket の方が有利となる．

次に，HTTP 通信のメッセージに含まれるヘッダーが比較的に軽くないことである．WebSocket の場合，メッセージごとに追加されるヘッダーの容量が 2-14[Byte]で，HTTP のレスポンスヘッダーが約 200[Byte]であること比べたら，何十倍も軽い．その結果，一つのメッセージが軽く，さらにそのやり取りが頻繁な環境においては，WebSocket が有利となる．

最後に，サーバ側とクライアント側の実装が異なることである．WebSocket においては，一つの動作に対し，サーバ側とクライアント側がほぼ同様な実装となるため，Server Sent Event と比べて実装のコストが半分になる．その結果，サーバとクライアントがお互いにメッセージを送る可能性のある環境においては，WebSocket が有利となる．

以上の三つことから，WebSocket は，通信コストを抑えながら軽いメッセージを頻繁に行う環境において，Server Sent Event より魅力的な手段である．特に，その数が 5 億個にも及ぶと予想され，比較的に軽いメッセージを，相互にやり取りする，IoT/M2M 環境においてより多く活用されると考えられる．

2.2 MQTT: MQ Telemetry Transport

Web の発展と伴い，上位層プロトコルは「リッチメディアを効率よく使うための高速化」や「M2M/IoT のような低機能な端末を対象とした軽量化」という二つの方向性で進化している[33]．この中でも，M2M や IoT における通信デバイスには省電力といった，これまでとは異なった要求条件が存在する．このような条件を満たすため，各種プロトコルをシンプル化・軽量化する取り組みが進み，CoAP，MQTT，AMQP などが代表的な例である[34][35][36]．その中でも MQTT は，以上に述べた SSE のような HTTP のリクエスト・レスポンスの方式を用いず，ブローカー(Broker)を用いたパブリッシュ／サブスクライブ(Publish/Subscribe)の方式でメッセージのやり取りを行う，M2M/IoT 指向のプロトコルである．

MQTT の通信

MQTT も HTTP と同様に TCP/IP プロトコルの上で動作するが，中には ZigBee といった非 TCP/IP ネットワーク上において動作する MQTT-SN というものも存在する[37]．以下の図 2.4 は，MQTT の通信の概略を示したものである．

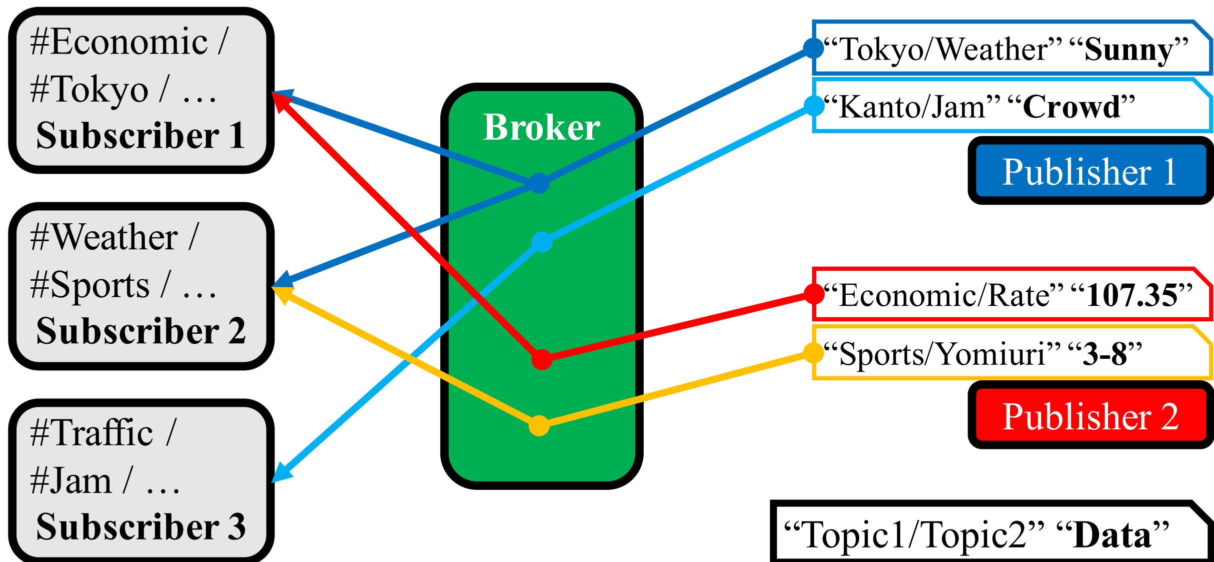


図 2.4 MQTT の動作

従来の HTTP では，多くのデバイスに情報を配信したい時，各デバイスと個別に通信を行っていた．一方，MQTT においては次のように一斉に送信が可能である動作を行う．情報を発信する側である「パブリッシャー」が，情報を中継する側である「ブローカー」(MQTT サーバ)にメッセージを配信する．この時，メッセージはトピックごとで階層構造化したものである．ブローカーは，受け取った全てのメッセージに対し，情報を受信する側である「サブスクライバー」のトピックに基づいて受信すべきデバイスを判断し，情報を転送する．そのために，サブスクライバーは受信したいメッセージのトピックを事前にブローカーに登録しておき，トピックの指定は「一部の情報を含んでいれば受信する」といったような指定も可能である．

その結果，発信側は一度の送信のみで情報を求めるデバイスの全てに情報の配信が可能で，受信側の要求に対してデバイスごとに行う必要がなく，一斉に送信が可能である．さらに，送信側と受信側の立場が逆転させることが可能であり，そうすることによって双方向通信を実現できる．

MQTT の特徴

これまでのインターネットのプロトコルは，主にエンドユーザとサーバの間を接続することを前提として作られており，そのエンドユーザが利用するデバイスとして PC やスマートフォンといった高性能なデバイスが想定されていた．これに比べ，MQTT は M2M/IoT 環境で用いられるデバイスの「低機能」や「省電力」といった特性を考慮して設計されたプロトコルであり，そのことから以下のような特徴を持っている[15].

(1) 固定ヘッダーが短く，ショートパケットによるオーバーヘッドが少ない.

(2) 一つのセッションで双方向通信を実現し，その接続を維持したままで動作する.

(3) 突発的な接続の切断に対し, Will・Retain・CleanSession といった対策が用意されている.

最初に，MQTT は固定長ヘッダーが 2[Byte]のみでオーバーヘッドが少なく，プロコトルそのものも比較的に単純なものである．それにより，従来の HTTP と比べた時，帯域の利用効率や処理速度の面においてメリットがある[38].

さらに，処理が少ないということから消費電力も少なくなっており，そのことから，モバイル機器にも向いているプロトコルである.

次に，HTTP のように通信の度に接続を終了する方式ではなく，一定間隔で Heart Beat を送り合うことで接続を維持し，双方向通信ができる．このことから，メッセージをリアルタイムに処理する能力が高い通信とも言える[39].

Will・Retain・CleanSession といった機能は，以下に図を用いて具体的に述べる．Will は「遺言」の意味で，以下の図 2.5 のように，パブリッシャーがブローカーに最初に接続した時に追加しておく情報のことである．このパブリッシャーとブローカーとの通信が突発的に切断された場合，ブローカーはこの Will によって指定されたトピックとメッセージをサブスクライバーに送信する．このメッセージにより，サブスクライバー側はパブリッシャーの接続が途切れたことが判断できる．また，パブリッシャーとの接続を確認するために，一定間隔ごとに Ping が送られることになり，この間隔も接続する毎に決められる．この Ping の間隔を調整することにより，デバイスのバッテリー消費をある程度制御することができ，省電力に繋がる.

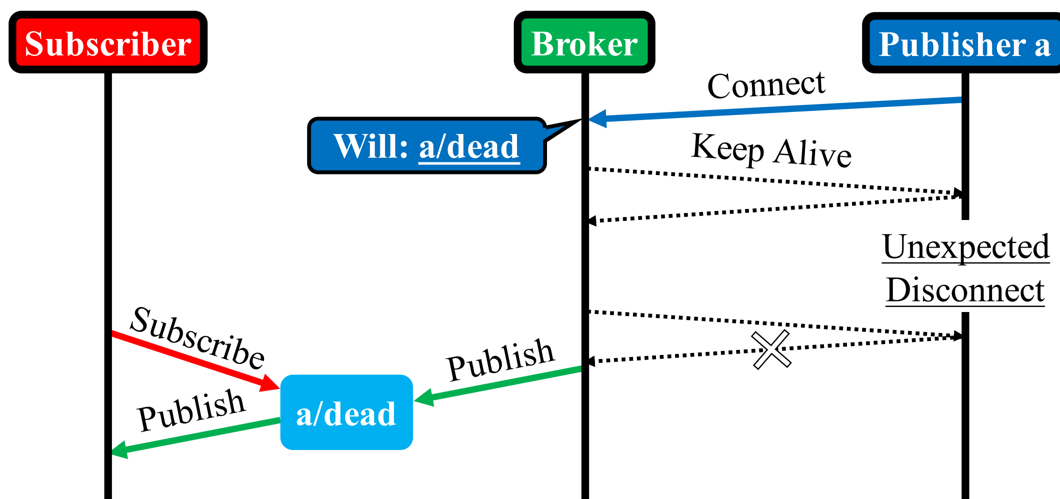


図 2.5　MQTT の Will

Retain とは，以下の図 2.6 のように，パブリッシャーの最後のメッセージをブローカーが保持しておき，新しくサブスクライバーが現れた時にそのメッセージを渡す機能のことである．パブリッシュ／サブスクライブの方式においては，パブリッシュの時にサブスクライブしていたクライアントにしかメッセージを送ることができない．しかし，Retain 機能により，新しいサブスクライバーでも最新の情報を得ることができる．

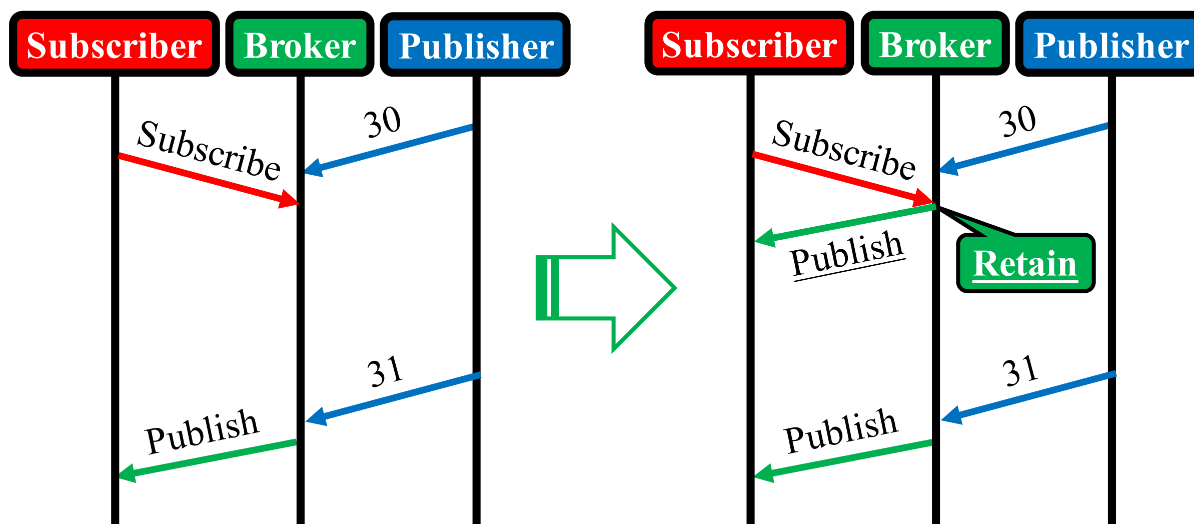


図 2.6　MQTT の Retain

CleanSession とは，パブリッシャーから送られたメッセージをブローカーが一定の期間保持しておく機能のことを意味する．MQTT は，ネットワークが不安定な状況を想定しており，Will はパブリッシャーの接続が突発的に切断された時の対策である．一方，CleanSession はサブスクラ

イバーの接続が突発的に切断された時(DISCONNECT，UNSUBSCRIBE 以外の切断)の対策であり，このような場合に再接続してきたサブスクライバーへその間のメッセージを再送する機能である．

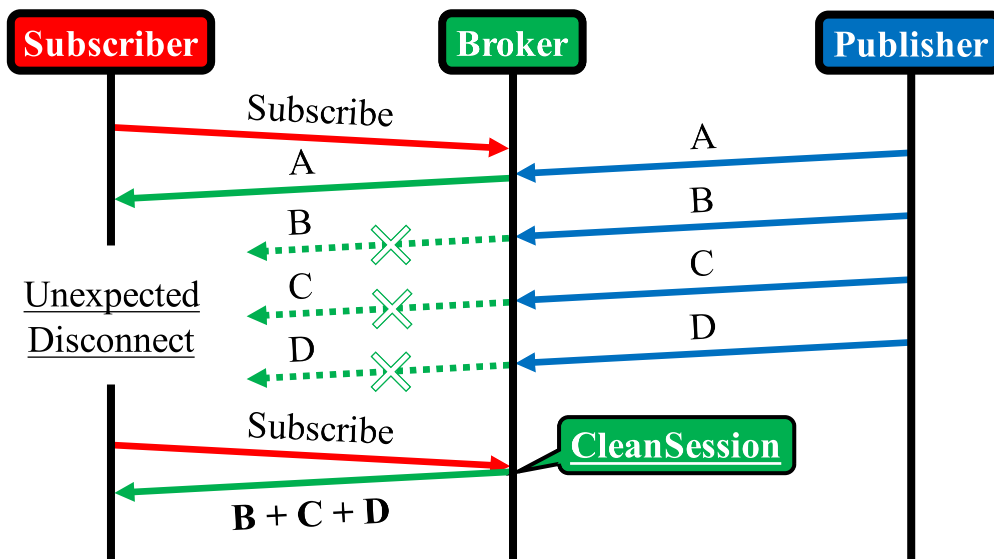


図 2.7　MQTT の Clean Session

これら以外にも，以下の図 2.8 のようにメッセージの持っているトピックが「／」の区切りで階層化されているおり，これらのトピックをワイルドカードで選択できることも大きい特徴である．また，以下の図 2.9 のようにメッセージの重要度によって配信の QoS が設定できることも MQTT の機能の一つである．

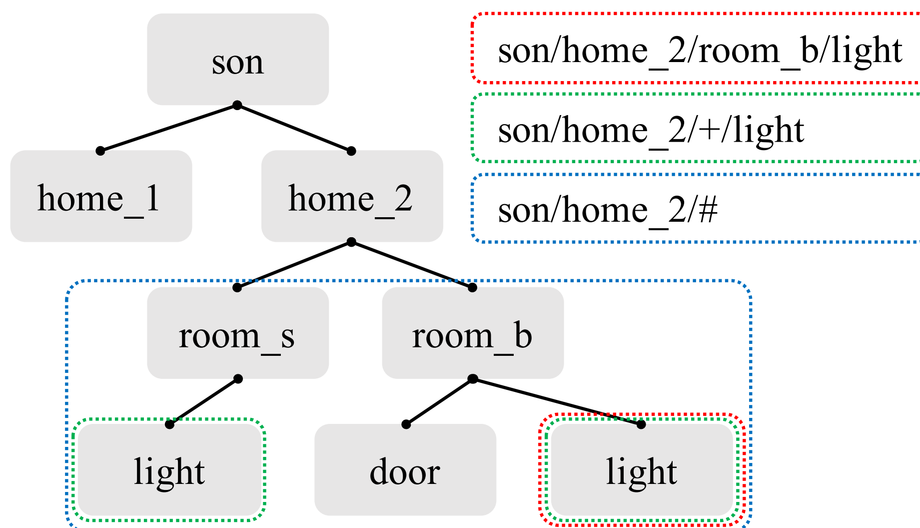
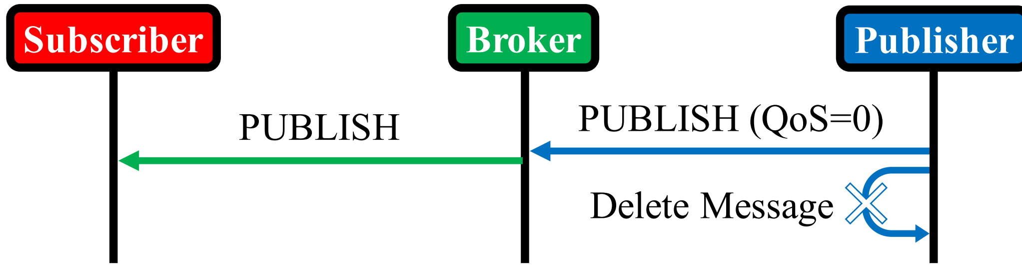
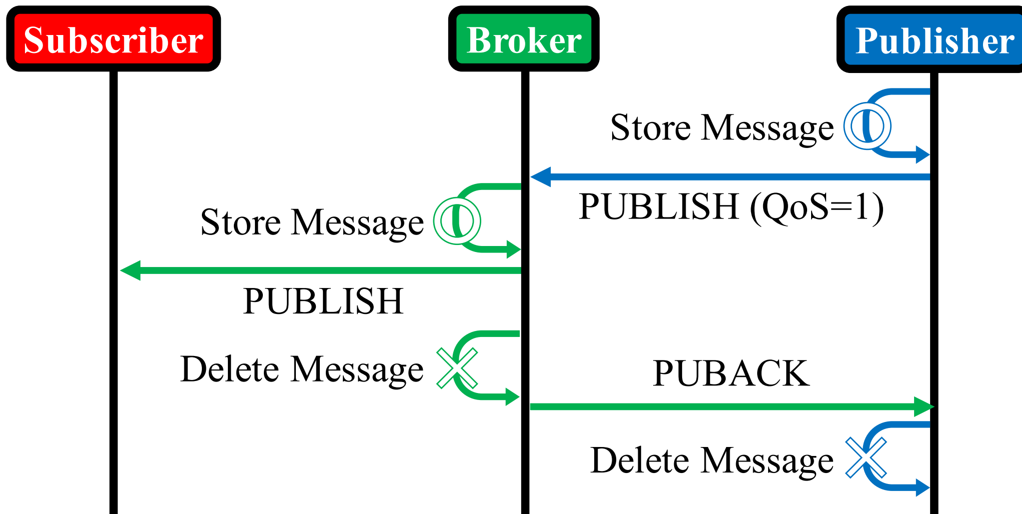


図 2.8　MQTT のメッセージの階層型トピック





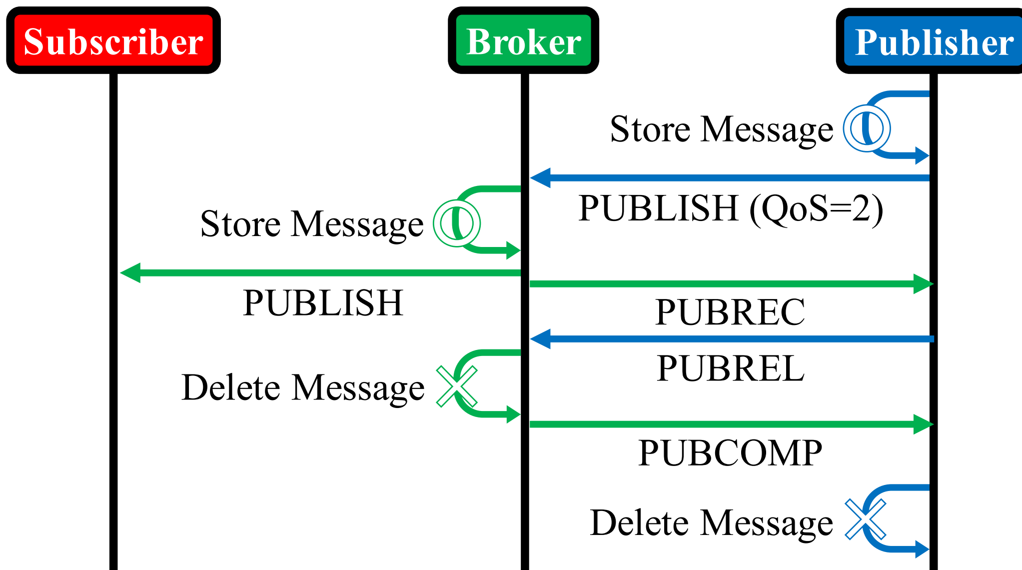


図 2.9　MQTT の Quality of Service

MQTT vs WebSocket

MQTT は双方向通信や軽量化されたプロトコルといった点で WebSocket と類似している面もあるが，構造や目的によってはやや相違点があると思われ，大きく以下の三つが挙げられる．

(1) MQTT は中継の役割を果たすブローカーが存在し，構造から大きく異なる．

(2) HTTP 通信への互換性の面において WebSocket の方が有利である．

(3) MQTT における QoS 機能は，メッセージ到達の信頼性に基づいたものであり，メッセージの順番に関する優先度ではない．

最初に，MQTT と WebSocket の二つのプロトコル間の相違点として，想定しているシステムの構成そのものが異なるということが挙げられる．WebSocket は，以前 HTML5 の仕様の一つであり，「Web ブラウザとサーバ間における通信」というのが前提であった．例えば，クライアントとクライアントの間の通信という状況のみに，その間でサーバが中継の役割を果たす構成である．一方，MQTT は，最初からブローカーという中継の役割をするものが存在し，パブリッシャーやサブスクライバーという二つのクライアントがブローカーに接続して通信を行う．すなわち，WebSocket の基本構成が一つのサーバと一つのクライアントであり，MQTT の基本構成は一つのサーバと二つのクライアントである．このような構成の相違により，システムの構成によって向き不向きが発生し，どちらかがはっきりと優位があると言い難いことになる．

次に，HTTP 通信との互換性が挙げられる．HTTP のやり方から脱却した WebSocket 通信であるが，通信の開始と終了を HTTP に沿った形としているため，MQTT と比べて HTTP 通信を行うシステムとの互換性が比較的に高い．一方，M2M/IoT 環境のみに想定した場合は，Zigbee などの非 TCP/IP 環境まで対応しているなど，M2M/IoT 環境に特化した MQTT の方が優位であると思われる．

最後に，MQTT における QoS 機能は，メッセージが相手に到達したかどうかについての QoS であり，本研究の目的とは異なるものである．本研究の目的は，「他のメッセージより到達の確率を高める」ものではなく，「他のメッセージより優先的に送信を行う」というもので，到達の確率ではなく時間に焦点を当てている．

このように，MQTT と WebSocket は類似している部分もあるが，目的や状況によってその差が存在する場合もある．例えば，一対一の通信，サーバとの連携，Web との関わりなどといった場面においては WebSocket が有効的であると思われ，その分野を M2M/IoT など，Web 以外に限定するのであれば，MQTT を用いた方が有効的であると思われる．

第3章　提案手法

3.1 システム構成

本研究では，以下の図 3.1 で示すようにシステムを構成した．サーバはクライアントに対し，アプリケーションから生成された優先度の異なる二種類のメッセージを WebSocket 通信で送信する．優先度の高いメッセージを Prioritized Message(以下，PM)，優先度の低いメッセージを Standard Message(以下，SM)と表記し，PM はランダム，SM は周期的に一定の間隔で生成されるとする．

サーバ・クライアント間の帯域は十分でなく，TCP セッションを一つのみを用いることで，最大限にコストを削減する．このような回線において，混雑が発生することや，送信に長時間がかかる程大きいメッセージが発生することを想定する．

今回の実装に用いられたもの[S]は，SendMessage というメソッドによって生成されたメッセージが WebSocket 通信に流し込まれ，WebSocket 通信の内部バッファに格納され，帯域が空き次第で順次に送出される仕様である．

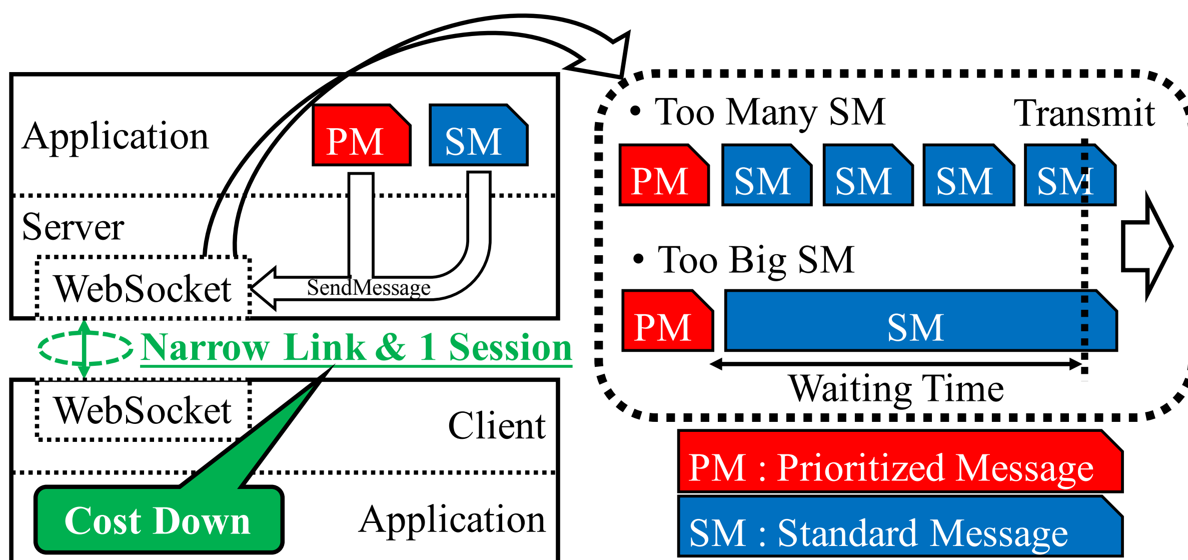


図 3.1　システム構成

WebSocket 通信の詳細

ここで，WebSocket 通信は「ハンドシェイク」と「メッセージのやり取り」というの二つのパートで構成されており，以下にその具体的な手順を述べる．

(1) 接続を確立するために，クライアントからサーバに以下のように HTTP リクエストを送り，ヘッダーでこの通信が WebSocket であることを要求する．

```
GET /chat HTTP/1.1
Host: server.example.com
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Key: dGhlIHNhbXBsZSBub25jZQ==
Origin: http://example.com
Sec-WebSocket-Protocol: chat, superchat
Sec-WebSocket-Version: 13
```

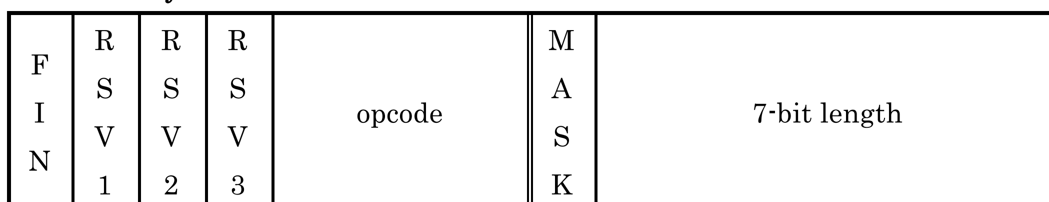
(2) リクエストを受け取ったサーバは，クライアントへ以下のようなレスポンスを返し，接続が確立される．

```
HTTP/1.1 101 Switching Protocols
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Accept: s3pPLMBiTxaQ9kYGzzhZRbK+xOo=
Sec-WebSocket-Protocol: chat
```

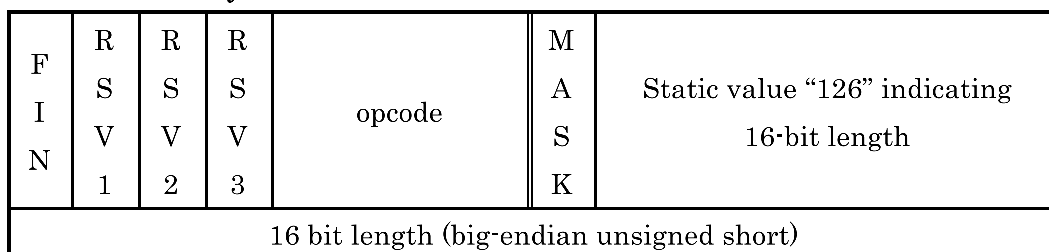
(3) (1)と(2)のハンドシェイクが成功したら接続が確立され，接続が終了される前までに以下の図 3.2 のような WebSocket ヘッダーが加わったメッセージのやり取りができる．ここで，ヘッダーの種類はフレームの大きさによって三つに分けられる．

[bit] 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16

Payload: 0-126 [Bytes]



Payload: 127-65535 [Bytes]



Payload: 65536- [Bytes]

F I N	R S V 1	R S V 2	R S V 3	opcode	M A S K	Static value “127” indicating 63-bit length
0	63 bit length (big-endian unsigned)					
63 bit length (continued)						
63 bit length (continued)						
63 bit length (continued)						

図 3.2 WebSocket 通信のヘッダー

本研究では，このような WebSocket 通信において，アプリケーションから生成されたメッセージは一本の TCP Stream 上にひとかたまりのメッセージとしてシリアライズ（送出）されることに着目し，バッファやメッセージ分割を用いた以下の二つの手法を提案する．

3.2 PQ-PC: use Prioritized Queue for Priority Control

本提案手法は，サーバ側の工夫のみで改善を図るものである．従来の WebSocket の仕組みでは生成されたメッセージがそのまま TCP のバッファに格納されて順次に送出される．一方，PQ-PC は，以下の図 3.3 に示すように，アプリケーションから生成された PM と SM を Prioritized Queue(以下，PQ)と Standard Queue(以下，SQ)にという別々のバッファにキューイングし，回線が利用可能となったら PQ から先にメッセージを送出する制御を行う．

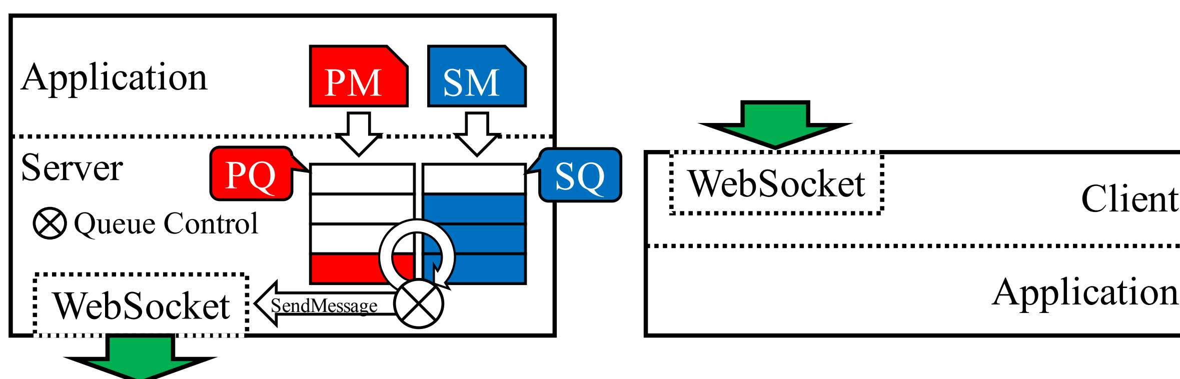


図 3.3 キューイングを用いた手法 PQ-PC

ただし，TCP のバッファに格納してからは制御することが難しいため，以下の図 3.4 のように，TCP のバッファではない場所にバッファでメッセージをキューイングし，一つのメッセージの送信が完了した時点におけるバッファのメッセージの中から次に送出するメッセージを選出するようにした．そうすることによって，TCP のバッファは常に送信中のメッセージのみが入ることになる．その Queue Control におけるアルゴリズムを以下に示す．

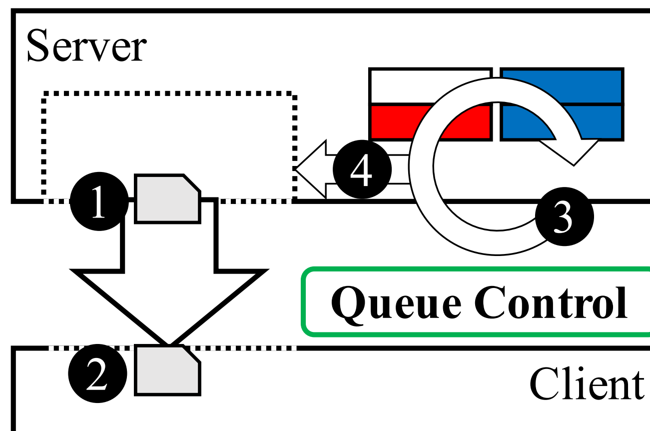


図 3.4 Queue Control の手順

(1) Queue から送出されたメッセージの送信を行っている．
(2) 一つのメッセージがクライアントに到達し，TCP Buffer は空き状態となる．この時，帯域も使われていない状態となる．
(3) 優先度の高いキューから次に送出するメッセージの選出を行う．
(4) WebSocket 通信部(TCP Buffer)に選出されたメッセージを送出する．

Algorithm 1. Queue Control

1: **while** ($PQ \neq \emptyset$ & $SQ \neq \emptyset$) **do**
2: **if** ($TCP\ Buffer \neq \emptyset$) **then**
3: **if** ($PQ \neq \emptyset$) **then**
4: sendMessage (PQ.get)
5: **else if** ($SQ \neq \emptyset$) **then**
6: sendMessage (SQ.get)
7: **end if**
8: **end if**
9: **end while**

PQ-PC においては，ある PM より先に生成された SM があるとしても，PM は回線の空き次第で優先的に送出されるため，以下の図 3.5 のように，SQ のバッファ長と関係なく送出が行われる．その時，PM が生成されてから送出されるまでの待機時間は最大でも SM 一つの送信時間のみになる．この方式は多くのメッセージの発生によって回線が非常に混雑している時に効果を発揮すると考えられる．

しかし，この手法のみでは大きいメッセージによる遅延が改善されない．これはメッセージの分割によって解決することができるが，それをクライアント側において元の形に復元するためには，以下の図 3.5 のように，そのための情報が分割メッセージにラベルとして追加される必要がある．この追加した情報量が多い場合，その分，通信量のオーバーヘッドが増加してしまう課題が残る．

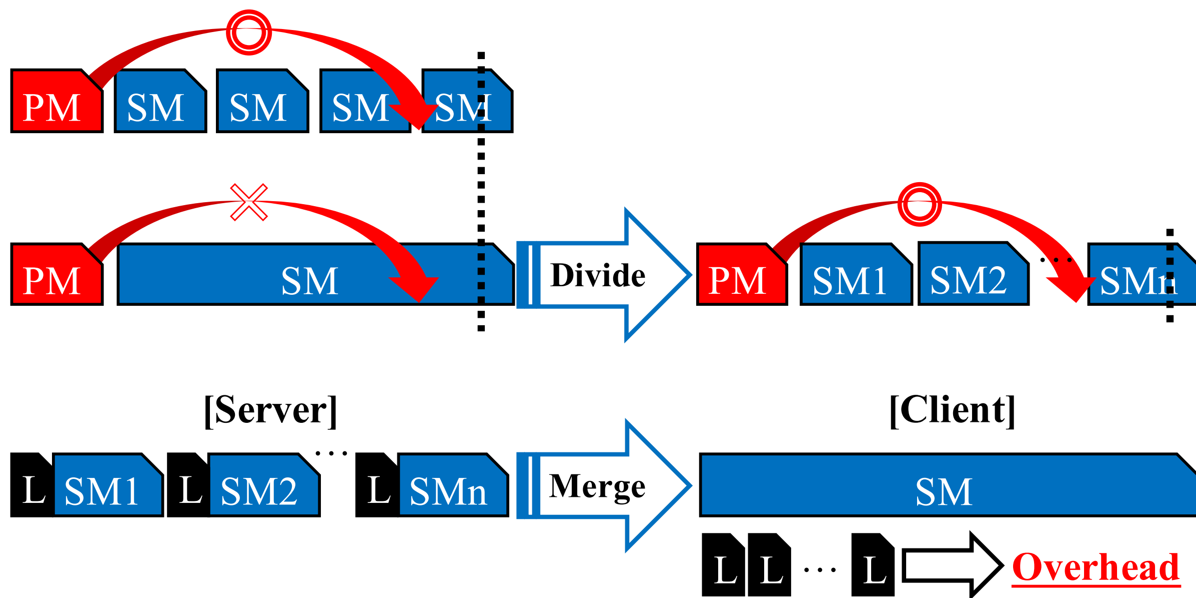


図 3.5 回線の混雑による遅延の改善や分割によるオーバーヘッド

3.3 DMPQ-PC: Divide a Message and PQ-PC

本提案手法は，サーバ側・クライアント側の双方に工夫を行って改善を図るものである．上に述べた PQ-PC は，複数のメッセージが詰まっている時の遅延の改善には有効的であるが，帯域を長時間占有してしまうほど大きいメッセージが発生した場合の解決策にはならない．そこで，DMPQ-PC は，以下の図 3.6 のように，アプリケーションからの SM をそのまま SQ に格納するのではなく，ある一定のサイズ DIVISOR(以下，DV)の大きさで分割して格納する方式である．この時，分割された SM の数 N_{DV} は，SM のサイズが $SIZE_{SM}$ である時，以下の式 3.1 のように定義できる．

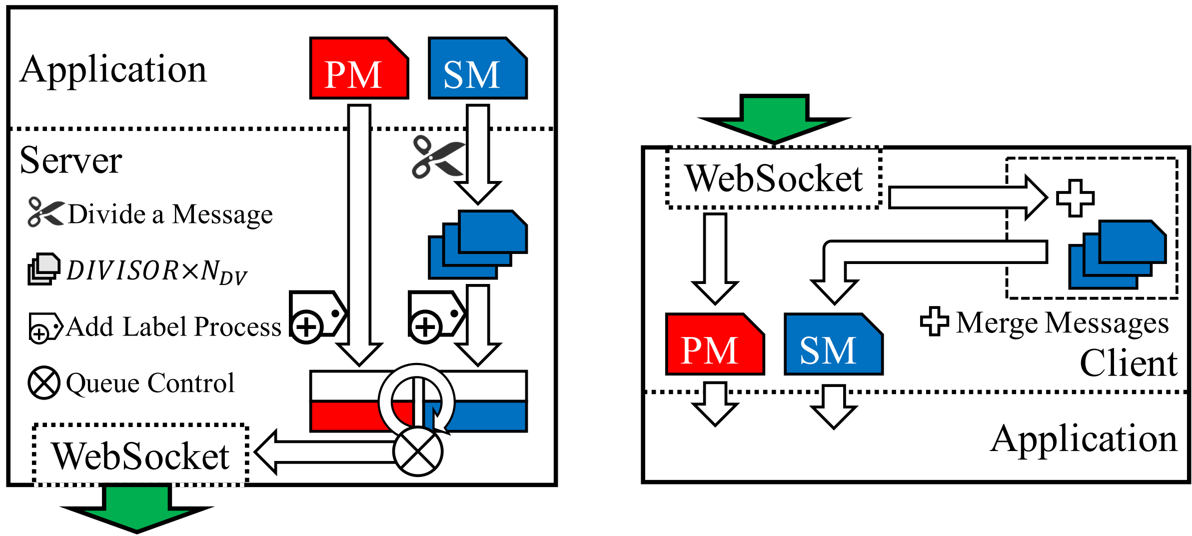


図 3.6 キューイングと分割を用いた手法 DMPQ-PC

$$N_{DV} = \left\lceil \frac{SIZE_{SM}}{DV} \right\rceil \qquad \cdots (3.1)$$

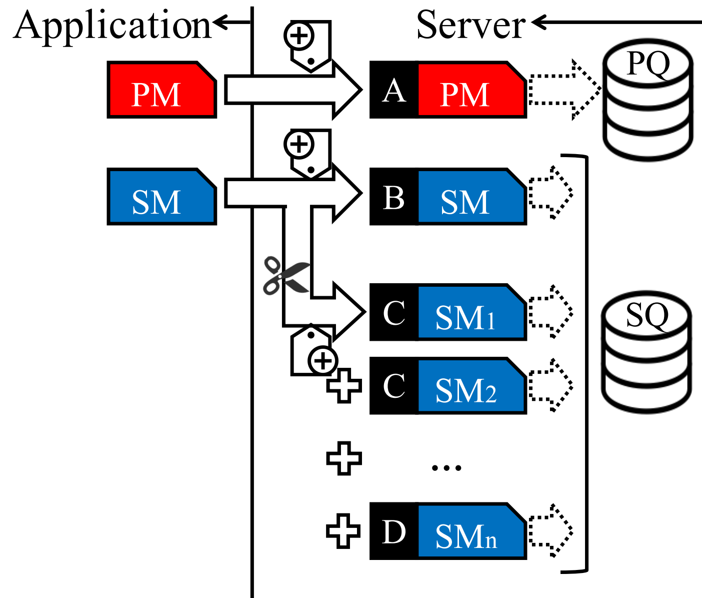


図 3.7 分割メッセージにラベルをつけるプロセス

一方，この手法を実現するためには，以上で述べたようにメッセージを分割する際にクライアントにおいて元に復元できるためのラベルが必要であり，その情報量をいかに小さくできるかが非常に重要である．そこで，本提案手法では，送出の順番が変わらない TCP 通信の特性に着目

し，以上の図 3.7 のように A，B，C，D の 4 種類のラベルを用いたたったの 2[bit]の情報で，メッセージの分類や復元を行う．その具体的な手順は，以下のようである．

(1) PM なら，ラベル A をつける．
(2) SM で DV より小さいサイズは，分割せずにラベル B をつける．
(3) SM で DV より大きいサイズは，DV の大きさで分割していきながら，ラベル C をつける．
(4) (3) において分割された最後のメッセージには，ラベル D をつける．

以上に述べたメッセージ分割やラベル追加, すなわち, 以上の図 3.6 における Divide a Message や Add Label Process のアルゴリズムを以下に示す．ここで，msg はメッセージ(Message)を，LM はメッセージの長さ(Length of Message)を意味する．

Algorithm 2. Divide message & Add label

```
 1:  if ( msg = PM ) then
 2:     msg = 'A' + msg
 3:     PQ.put ( msg )
 4:  else if ( msg < DV ) then
 5:     msg = 'B' + msg
 6:     SQ.put ( msg )
 7:  else
 8:     while ( i + DV > LM ) do
 9:        SQ.put ( 'C' + msg [ i:i+DV ] )
10:        i = i + DV
11:     end while
12:     SQ.put ( 'D' + msg [ i:LM ]
13:  end if
```

一方，クライアント側では，図 3.7 のように分割されてラベルが追加されたメッセージを受け取り，以下の図 3.8 のようにメッセージを分別してアプリケーションに渡したり一時的にキューに格納しておいて元の形に復元したりする．その具体的な手順を以下に示す．

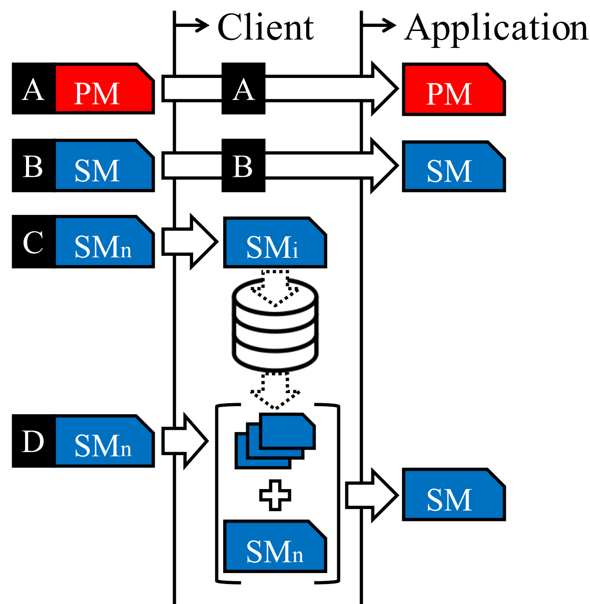


図 3.8 クライアントにおけるメッセージ処理

(1) ラベルが A なら，ラベルをはがしたメッセージをアプリケーションに渡す.
(2) ラベルが B なら，ラベルをはがしたメッセージをアプリケーションに渡す.
(3) ラベルが C なら，ラベルをはがしたメッセージを一時的にキューに格納しておく.
(4) ラベルが D なら，ラベルをはがしたメッセージを一時的にキューに格納し，キューに入っているメッセージ全てを一つのメッセージとしてアプリケーションに渡す.

また，以上の図 3.8 の示すメッセージ復元のアルゴリズムを，以下に示す．ここで，rmsg は受信したメッセージ(Received Message)，LRM は受信したメッセージの長さ(Length of Received Message)，TQ は分割されているメッセージを一時的に保管するためのキュー(Temporary Queue)を意味する.

Algorithm 3. Mange message at client side

```
1:  if ( rmsg[0] = 'A' ) then
2:      msg = rmsg [ 1:LRM ]
3:      relay msg to Application
4:  else if ( rmsg[0] = 'B' ) then
5:      msg = rmsg [ 1:LRM ]
6:      relay msg to Application
7:  else if ( rmsg[0] = 'C' ) then
8:      TQ.put ( rmsg [1:LRM] )
```

```
9:   else if ( rmsg[0] = 'D' ) then
10:    TQ.put ( rmsg [1:LRM] )
11:    while ( TQ  ≠ ∅ ) do
12:      msg = TQ.get
13:    end while
14:    relay msg to Application
15:  end if
```

一方，この手法においては，以下の図 3.9 のように，送出されるメッセージのひとかたまりの大きさが DV という値に縮小され，結果として SM が部分的に送出される途中にも PM を送出できる隙間が生まれる. SM の大きさに依存せずに PM の配送遅延を抑えることができる. さらに，DV が小さければ小さいほどその隙間の間隔が小さくなり，PM の配送遅延は小さくなる.

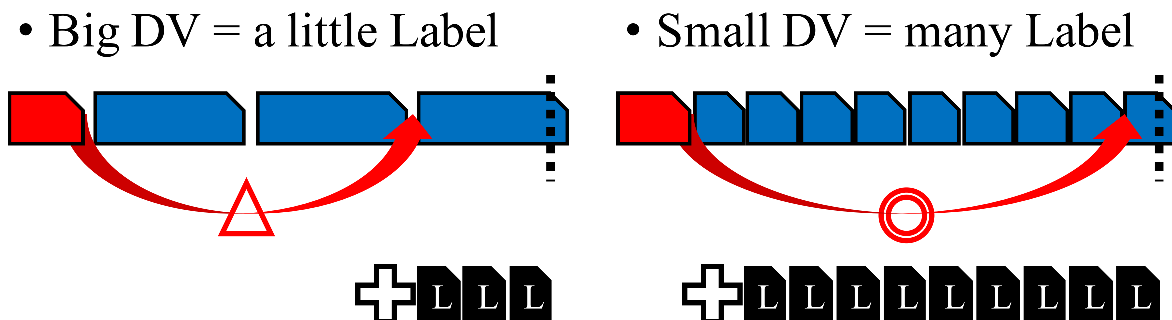


図 3.9 分割による遅延の抑制効果

しかし，どのような大きさで分割するかを考慮する必要がある．なぜなら，ラベルの情報量を小さく抑えたとしても，以下の図 3.10 のように，メッセージはパケットに断片化されて送られ，どのような大きさで分割するかによってパケットの利用効率が変わり，この利用効率が悪い場合パケットのヘッダーが多く発生してしまうためである．例えば，図 3.10 のような場合は，DV_1 と DV_2 との差が大きくなければ，DV_2 より DV_1 の単位でメッセージを分割した方がパケットの利用効率の面において良いと言える．

そこで，本研究では一つのパケットに入るデータの最大値である 1444[Bytes]からラベルの大きさ 1[Bytes]を引いた値である 1443 [Bytes]を DV の最小値として設定し，実際にもこの値において通信量のオーバーヘッドが抑制されるかを評価する．

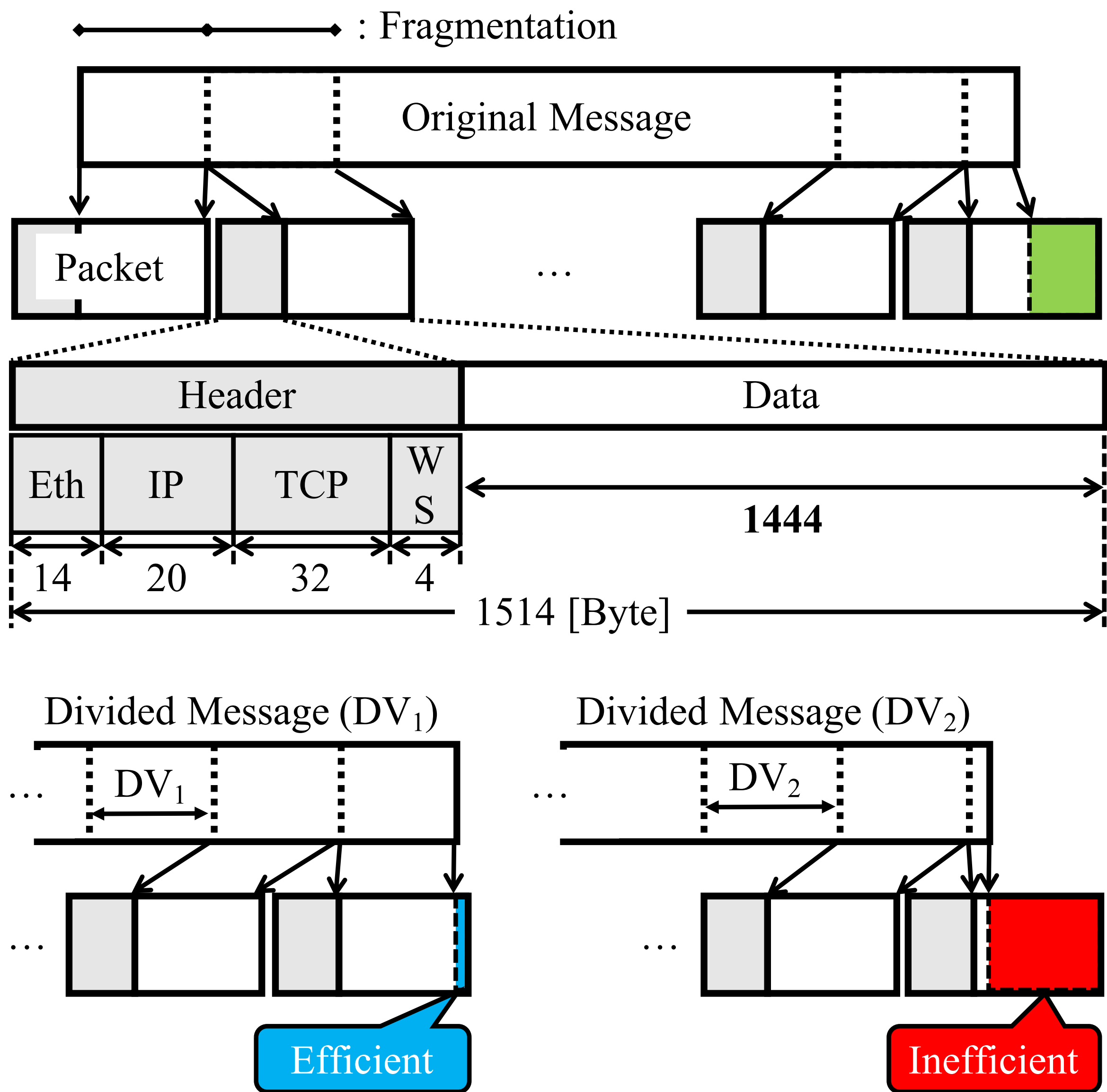


図 3.10 メッセージの断片化やパケットの利用効率

DMPQ-PC においては，送出されるメッセージのひとかたまりが小さくなり，SM の送出が完了されていなくても PM の送出が可能である．この方式は回線が非常に混んでいる時のみならず，大きいメッセージで回線が詰まってしまった時も効果を発揮すると考えられる．

第4章　パフォーマンス･モデル

4.1 従来手法

PM の配送遅延の分解

最初に，従来手法と提案手法の PM の配送遅延を予測するために，これを式で表す．PM が生成されてから送出される直前までの時間を T_{PM1}，PM が送出されてからクライアントに届くまでにかかる時間を T_{PM2} とすると，PM の配送遅延 D_{PM} を以下の式 4.1 で表すことができる．

$$D_{PM} = T_{PM1} + T_{PM2} \qquad \cdots (4.1)$$

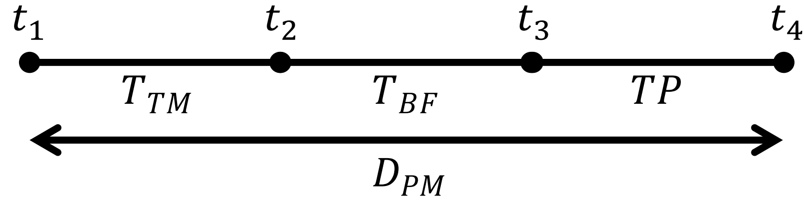
以上の式 4.1 の二つ目の項である T_{PM2} は，PM の大きさ $SIZE_{PM}$ を帯域速度 Bandwidth(以下，BW)で割って決まる定数であるため，式 4.1 は以下の式 4.2 に書き直すことができる．

$$D_{PM} = T_{PM1} + \frac{SIZE_{PM}}{BW} \qquad \cdots (4.2)$$

また，T_{PM1} は送出中のメッセージが送出完了となるまでの時間 T_{TM} と, PM より先に送られる待機中のメッセージが全て送信される時間 T_{BF}，二つを足したものであり，式 4.2 の二項は定数 TP で置き換えられるので，式 4.2 は以下の式 4.3 に書き直すことができる．ここで，TP は一つの PM を送る時間と定義する．

$$D_{PM} = T_{TM} + T_{BF} + TP \quad , \qquad \left(\frac{SIZE_{PM}}{BW} \stackrel{\text{def}}{=} TP\right) \qquad \cdots (4.3)$$

以上の式 4.3 を用いて従来手法と二つの提案手法のパフォーマンスを考えるとする．また，この時間関係を図で示すと以下の図 4.1 のようになる．



t_1 ：　PM が発生した時刻

t_2 ：　t_1 に送出中であったメッセージの送出が完了された時刻

t_3 ：　t_1 で発生した PM の送出が始まる時刻

t_4 ：　t_1 で発生した PM がクライアントに届いた時刻

図 4.1 PM 配送遅延の構成

送出中のメッセージが送出完了となるまでの時間 T_{TM}

従来手法の場合，送出中のメッセージは PM や SM そのものであるため，式 4.3 の一項である T_{TM} の最大値は一つの SM を送信するのにかかる時間となり，最小値は 0 である．

バッファ長が 0 の場合の T_{BF} および PM の配送遅延 D_{PM}

式 4.3 の二項である T_{BF} は，バッファ長が 0 である場合と 0 でない場合にわけて考えることができる．無論，バッファ長が 0 である場合は T_{BF} が 0 となり，式 4.3 は以下の式 4.4 に書き直すことができる．ここで，TS は一つの SM を送る時間と定義する．

$$
\begin{aligned}
D_{PM} &= T_{TM} + T_{BF} + TP \\
&= T_{TM} + TP \quad (\because\ T_{BF} = 0) \\
&\leq TS + TP \quad \left(\because\ \frac{SIZE_{SM}}{BW} \stackrel{\text{def}}{=} TS\ , T_{TM} \leq TS\right) \\
&\lesssim TS \quad (if,\ SIZE_{PM} \ll SIZE_{SM})
\end{aligned}
\quad \cdots (4.4)
$$

バッファ長が 0 でない場合の D_{PM} および PM の配送遅延 D_{PM}

一方，バッファ長が 0 でない場合，T_{BF} は PM が生成された時点において WebSocket の内部バッファに入っているメッセージを全て送る時間となる．これを式で表すと以下の式 4.5 になる．ここで，n_{WPM} は WebSocket の内部バッファに存在する PM の数を，n_{WSM} は WebSocket の内部バッファ内に存在する SM の数を意味する．

$$
\begin{aligned}
D_{PM} &= T_{TM} + T_{BF} + TP \\
&\leq TS + T_{BF} + TP \qquad (\because\ T_{TM} \leq TS) \\
&\leq TS + (n_{WSM}TS + n_{WPM}TP) + TP \\
&\leq (n_{WSM} + 1)TS + (n_{WPM} + 1)TP \\
&\lesssim (n_{WSM} + 1)TS + n_{WPM}TP \ (if,\ SIZE_{PM} \ll SIZE_{SM}) \\
&\lesssim (n_{WSM} + 1)TS \qquad (if,\ n_{WPM} \lesssim n_{WSM}) \qquad \cdots (4.5)
\end{aligned}
$$

すなわち，バッファ長が 0 ではない場合，$SIZE_{PM}$ が $SIZE_{SM}$ と比べて十分小さく，WebSocket の内部バッファ内の PM の数 n_{WPM} が SM の数 n_{WSM} より大きくないという条件が成り立つ時は，PM の配送遅延の最大値は「バッファに保存されている SM の数 n_{WSM}」に「一つの SM を送る時間 TS」をかけた値にほぼ等しくなり，その値はバッファ内に存在する SM の数 n_{WSM} に依存して増加する．

4.2 PQ-PC

送出中のメッセージが送出完了となるまでの時間 T_{TM}

PQ-PC の場合，送出中のメッセージは PM や SM そのものであるため，式 4.3 の一項である T_{TM} の最大値は一つの SM を送信するのにかかる時間である TS となる．

PM より先に送られるメッセージが送信される時間 T_{BF}

バッファ長が 0 である場合は T_{BF} が 0 となる. これまでは従来手法と同様であり, その際の PM の配送遅延も式 4.4 になる．しかし，バッファ長が 0 でない場合は従来手法と異なってくる．PQ-PC における T_{BF} は，バッファに入っているメッセージを全て送る時間ではなく，バッファ PQ に入っているメッセージのみを送る時間となり，その時の T_{BF} は以下の式 4.6 のようになる．ここで，n_{PM} はバッファ PQ に保存されている PM の数と定義し，T_{BF} は「バッファ PQ に保存されている PM の数 n_{PM}」に「一つの PM を送る時間 TP」をかけた値となる．

PM の配送遅延 D_{PM}

以上に述べたことを式で表すと以下の式 4.6 になる．

$$
\begin{aligned}
D_{PM} &= T_{TM} + T_{BF} + TP \\
&\leq TS + n_{PM}TP + TP \quad (\because\ T_{TM} \leq TS\ ) \\
&\leq TS + (n_{PM} + 1)TP \\
&\lesssim TS \qquad (\ if,\ n_{PM} \ll 1\ ,\ SIZE_{PM} \ll SIZE_{SM}\ ) \qquad \cdots\ (4.6)
\end{aligned}
$$

すなわち，バッファ PQ に入っている PM の数 n_{PM} が十分少なく，$SIZE_{PM}$が $SIZE_{SM}$と比べて十分小さいという条件が成り立つ時は，配送遅延の最大値を一定の値である TS に近い値で抑えることができる．ここで，$1 \gg n_{PM}$ は PM の発生周期より PM の送る時間が短ければ成り立つ条件である．

これは，バッファ長が存在する場合，配送遅延の最大値がそのバッファ長によって増加する従来手法と比べ，配送遅延の最大値がある一定の値で抑制できるとのことを意味する．しかし，バッファ長が 0 である時は従来手法の配送遅延と大きく変わらない．また，バッファ長の有無と関係なく配送遅延 D_{PM} が SM の大きさ $SIZE_{SM}$には依存したままなので，帯域が長時間占有されるほどの大きいメッセージが SM として現れた場合，その占有時間がそのまま配送遅延に反映されてしまう点がある．

4.3 DMPQ-PC

送出中のメッセージが送出完了となるまでの時間 T_{TM}

DMPQ-PC の場合，送出中のメッセージの大きさが以上に述べた従来手法や PQ-PC とは違ってくる．送出されるメッセージが PM や SM のそのままの形でなく，PM はそのままのメッセージにラベルが加わったものとなり，SM は DV という大きさで分割されてラベルが加わったものになる．その結果，DV がより $SIZE_{PM}$ より大きければ，式 4.3 の一項である T_{TM} の最大値は DV という大きさの分割メッセージにラベルが加わったものを送信する時間となる．これは，DV を調整することで配送遅延の最大値の一部の T_{TM} を制御できるとのことを意味する．

PM より先に送られるメッセージが送信される時間 T_{BF}

バッファ長が 0 である場合，T_{BF} は 0 となり，その際の PM の配送遅延も式 4.4 になる．バッファ長が 0 でない場合，T_{BF} はバッファ PQ に入っているものを送る時間となり，その中身は PM にラベルが付いたものである．その時の T_{BF} は「バッファ PQ に保存されている PM の数 n_{PM} 」に「一つのラベル付き PM を送る時間 TP 」をかけた値となる．

PM の配送遅延 D_{PM}

以上に述べたことを式で表すと以下の式 4.7 になる．ここで，$SIZE_{LB}$ はラベルの大きさを意味する．

$$
\begin{aligned}
D_{PM} &= T_{TM} + T_{BF} + TP \\
&\leq \frac{DV + SIZE_{LB}}{BW} + n_{PM} \times \frac{SIZE_{PM} + SIZE_{LB}}{BW} + TP \\
&\leq \frac{DV + n_{PM} SIZE_{PM} + (n_{PM} + 1) SIZE_{LB}}{BW} + TP \\
&\leq \frac{DV + (n_{PM} + 1) SIZE_{LB}}{BW} + (n_{PM} + 1) TP \\
&\lesssim \frac{DV + SIZE_{LB}}{BW} + TP \quad (\, if,\ n_{PM} \ll 1\,)
\end{aligned}
\quad \cdots (4.7)
$$

すなわち，従来手法や PQ-PC と違い，配送遅延の最大値が $SIZE_{SM}$ に依存しない値となり，その値も DV を調整することによって制御することが可能である．これは $SIZE_{SM}$ がいくら大きくても, PM の配送遅延を式 4.7 のように一定の値以下で抑えることができるとのことを意味する．

第5章　評価実験

5.1 実験の詳細

PM の配送遅延および通信量を求めるべく，そのための実験の構成や仕様，実験環境，そして実験に用いられたパラメータを以下に示す.

5.1.1 実験の構成や仕様

回線の混雑している環境において，提案手法の PQ-PC や DMPQ-PC によって PM の配送遅延がどれほど抑えられるか評価した．さらに，その時の通信量やパケット数から提案手法によるオーバーヘッドを評価した．この評価のために，本研究では以下の図 5.1 のように実験環境を構成した．また，図 5.1 におけるクライアントやサーバ，そしてハブの仕様を以下の表 5.1 に示す.

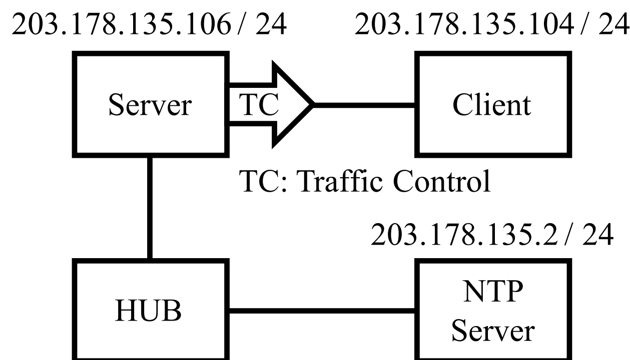


図 5.1 評価実験の構成

表 5.1 クライアント・サーバ・ハブの仕様

Server / Client	
Model	Lenovo ThinkPad X201i
HDD	250 [GB]
Memory	4 [MB]
Processor	Intel Core i3 CPU M 380 2.53 Ghz x 4
OS	Ubuntu 14.04 LTS Server 64-bit / Ubuntu 14.04 LTS Desktop 64-bit
HUB	
Model	Logitec LAN-SW08P / TAPA
Port	8
Connection	100 BASE-TX (Switching HUB)

5.1.2 実験環境の構築

実験環境を構成するための具体的な手順は以下のようである．

(1) LAN Cable を用いて図 5.1 のようにサーバ，クライアント，ハブをつなげる．

(2) クライアントやサーバのターミナル上で，以下のように ifconfig コマンドを用いてアドレスを設定する．

ifconfig eth0 203.178.135.104 [クライアント]

ifconfig eth0 203.178.135.106 [サーバ]

(3) クライアントとサーバ間に正常に通信ができるかどうかを，以下のように ping コマンドで確認する．

ping 203.178.135.104 [サーバ]

(4) 正確な配送遅延を求めるために，クライアントとサーバの時刻を同期する．そのために，NTP Server との時刻の差がほぼ 0 になるまで，双方のターミナル上で以下のように ntpdate コマンド実行させる．また，この時刻の同期は実験を行う度に実行し，時刻の非同期による誤差を最低限に抑える．

ntpdate 203.178.135.2 [クライアント・サーバ]

(5) Autobahn|Python[5]を用いてクライアントやサーバに WebSocket を実装する．その後，以下のファイルの中身でアドレスを指定する部分を図 5.1 のように変更し，サーバの方から実行して"Hello, world"が正しく出力されるか確認する．ポート番号は 9000 番を用いた．

python /example/twisted/websocket/echo/client.py [クライアント]

python /example/twisted/websocket/echo/server.py [サーバ]

(6) server.py および client.py ファイルを一部書き換えたファイル(5.2 節)を実行しながら，従来手法および提案手法の配送遅延と通信量を求める．その時，帯域速度の変更が必要な場合は，ファイルを実行する前に以下のような括弧の中の数値を変化しがら tc コマンドを実行し，サーバから送出されるパケットの帯域を制限した．

tc qdisc add dev ethic root tbf limit [ ]Mb buffer [ ]Kb rate [ ]Kbps

5.1.3 実験パラメータ

実験に用いられたメッセージや帯域速度に関する各パラメータを以下の表 5.2 に示す.

表 5.2 評価実験に用いられたパラメータ

Prioritized Messages	
Pattern	Random
Size	1 [Kbyte]
Period	1 [Second]
Probability	10 [%]
n	100
Standard Messages	
Pattern	Periodic
Size	100 [Kbyte]
Period	10 [Second]
n	100
Bandwidth	
Rate	4 – 20 [Kbps]
Buffer	Rate / 5 [KB]

具体的には，SM を 10[Second]毎に周期的に生成される 100[Kbyte]のメッセージとし，帯域速度はその SM のみで 100[%]の占有される 10[Kbps]の前後の区間である 4-20[Kbps]と設定した．PM は 1[Second]毎に 10[%]の確立でランダムに生成される 1[Kbyte]のメッセージとし，この値は式(4.4)と式(4.5)に示す， $SIZE_{PM}$ が $SIZE_{SM}$ と比べて十分小さく($SIZE_{PM} \ll SIZE_{SM}$)，WebSocket の内部バッファ内の PM の数 n_{WPM} が SM の数 n_{WSM} より大きくない($n_{WPM} \lesssim n_{WSM}$)，そしてバッファ PQ に入っている PM の数 n_{PM} が十分少ない($n_{PM} \ll 1$)という条件を満たす値となる．

5.2 ファイルの実装

評価実験に用いられたプログラムのメソッドなどの詳細を手法ごとに分けて以下に示す.

5.2.1 従来手法

以上の図 3.1 のように従来手法を実現するために，サーバにおける **server.py** ファイルの **MyServerProtocol** というクラスの中に, 以下の表 5.3 のように必要なメソッドを追加したファイル **default_server.py**[付録：ソースコード 1]を実装した.

表 5.3 default_server.py に加わったメソッド

メソッド名	役割
PM_create	表 5.2 で示す仕様の PM を生成してバッファに格納する.
SM_create	表 5.2 で示す仕様の SM を生成してバッファに格納する.

最初に PM と SM を生成してバッファに格納するメソッドを加えた．この時，メソッドが生成するメッセージは，表 5.2 に示した仕様のものである．ここで，PM と SM は以下の図 5.2 のように，先頭に 17 文字の生成された時刻情報を持つ文字列であり，この情報が後に配送遅延の計算に用いられる．また，PM と SM を区別するために，残りの文字列の部分は PM が‘1’，SM が‘2’の文字で詰められる．従来手法の場合，このように生成された二種類の文字列は，どちらも WebSocket の内部バッファに格納される．

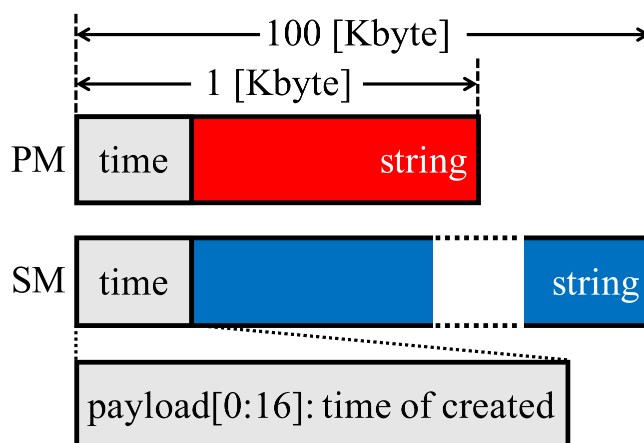


図 5.2 PM と SM の構造

このようにサーバの **default_sever.py** によって生成されて送出されたメッセージは，WebSocket 通信でクライアントに届き，クライアントは配送遅延の計算を行う必要がある．その時，配送遅延(**t_delay** in default_client.py)は受け取りを完了した時刻(**t_received** in default_client.py)から生成された時刻を引いた値となり，ここで生成された時刻は受け取ったメッセージの先頭 17 桁の文字列(**payload[0:17]** in default_client.py)である．この計算を実現するために，クライアントにおける **client.py** ファイルの **MyClientProtocol** というクラスの中にある **onMessage** というメソッドの中で，配送遅延を計算して結果を出力するファイル **default_client.py**[付録：ソースコード 2]を実装した．

5.2.2 PQ-PC

以上の図 3.3 のように PQ-PC を実現するために，以上で実装した **default_server.py** ファイルを元に，以下の表 5.4 のように PM_create_PQ と SM_create_SQ，そして新たに PC_flush というメソッドを追加したファイル **pq_server.py**[付録：ソースコード 3]を実装した．

表 5.4 pq_server.py に加わったメソッド

メソッド名	役割
PM_create_PQ	PM を生成してバッファ PQ に格納する．
SM_create_SQ	SM を生成してバッファ SQ に格納する．
PC_flush	クライアントへメッセージの送出が完了した時点で， PQ からメッセージを WebSocket 通信に流し込む．

新たなメソッド PC_flush は，バッファから WebSocket 通信にメッセージを流し込むタイミングを制御するために加えたメソッドである．今回用いた WebSocket の実装においては，メッセージの送信が完了して TCP のバッファが空きになったことが直接参照できず，図 3.4 のように Queue Control ができなかった．

そこで，UDP パケットを用いた PC_flush というメソッドでで図 3.4 の Queue Control を実現した．他の動作においては図 3.4 と同様であるが，送信が完了して TCP バッファが空きになったことを参照するために，以下の図 5.3 のように，クライアントからメッセージが到達した時点で UDP パケットをサーバへ返すようにした．この UDP パケットとしては，受信メッセージの先頭文字列の一文字（ラベルに相当する，1[Byte]）を用いた．

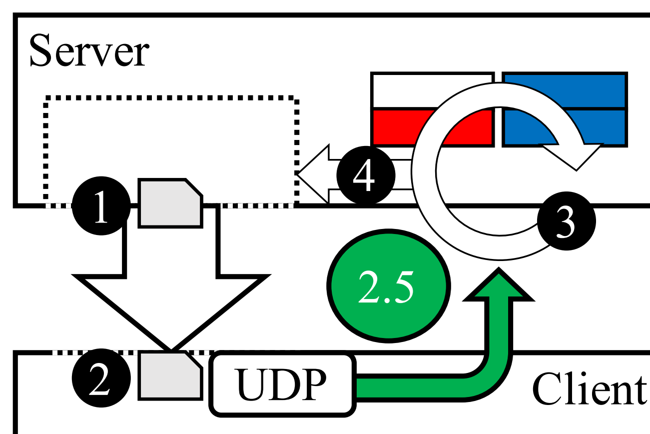


図 5.3 メソッド PC_flush の動作

UDP パケットを利用した理由としては，送信速度が速いためリアルタイムに近く，これは実際にサーバ側で TCP バッファを参照するのにかかる時間とほぼ近いと思ったためである．また，そのパケットの大きさは 1[Byte]しかないため，実験結果にほぼ影響を及ぼさないと判断した．

このメソッドは，**default_client.py** ファイルの中の，メッセージを受け取った瞬間呼び出される **onMessage** メソッドが始まると同時に，サーバ側へ UDP パケットでレスポンスを送るようにした **pq_client.py**[付録：ソースコード 4]ファイルで実装を行った．

5.2.3 DMPQ-PC

以上の図 3.6 のように DMPQ-PC を実現するためには，以上の図 3.7 のように「ラベル追加」や「メッセージ分割」の機能が必要である．そこで，以上に述べた Algorithm 2 を参考にし，以下の表 5.5 のように **pq_server.py** の PM_create_PQ と SM_create_SQ に変更を加えたメソッド PM_create_label と SM_create_label を作り，ファイル **dmpq_server.py**[付録：ソースコード 5]で実装した．

表 5.5 dmpq_server.py に加わったメソッド

メソッド名	役割
PM_create_label	PM を生成してラベル‘A’をつけてバッファ PQ に格納する．
SM_create_label	SM を生成して図 3.7 のように大きさによってメッセージを分割し，それぞれにラベル‘B’，‘C’，‘D’をつけて，バッファ SQ に格納する．

一方，クライアント側において図 3.8 のようにメッセージの分別かつ統合の処理が必要となる．そこで，以上に述べた Algorithm 3 を参考にし，このような処理をファイル **dmpq_client.py**[付録：ソースコード 6]に実装した．ここで，**received_msg** は，分割されたメッセージを後に統合させるためのクライアントの一時的なバッファである．

5.3 配送遅延の評価

提案手法によって PM の配送遅延がどれほど抑えられたかの評価を行った．以下には実験の手順やその結果，そして結果に関する評価を示す．

5.3.1 手順

従来手法および提案手法における PM の配送遅延を求める具体的な手順は以下のようである．

(1) サーバ側において，tc コマンドを用いて帯域速度を設定する．
(2) クライアントおよびサーバ側において，ntpdate コマンドを実行して時刻を同期する．
(3) サーバのプログラムを実行する．：default_server.py / pq_server.py / dmpq_server.py

(4) クライアントのプログラムを実行する．：default_client.py / pq_client.py / dmpq_client.py
(5) (1) ~ (4)を繰り返す．

5.3.2 結果

5.3.1 の手順に従って従来手法および提案手法における PM の配送遅延の評価を行い，その結果を以下の図 5.4 に示す．PM の配送を 100 回行い，生成された時刻からクライアントに届いた時刻までの時間を配送遅延とし，それをグラフにプロットしたものである．ここで，Default は従来手法の結果であり，TS は SM を送信するのにかかる時間の理論値である．また，結果を見易くするためにそれぞれの横軸のデータを少しずつずらして表記した．

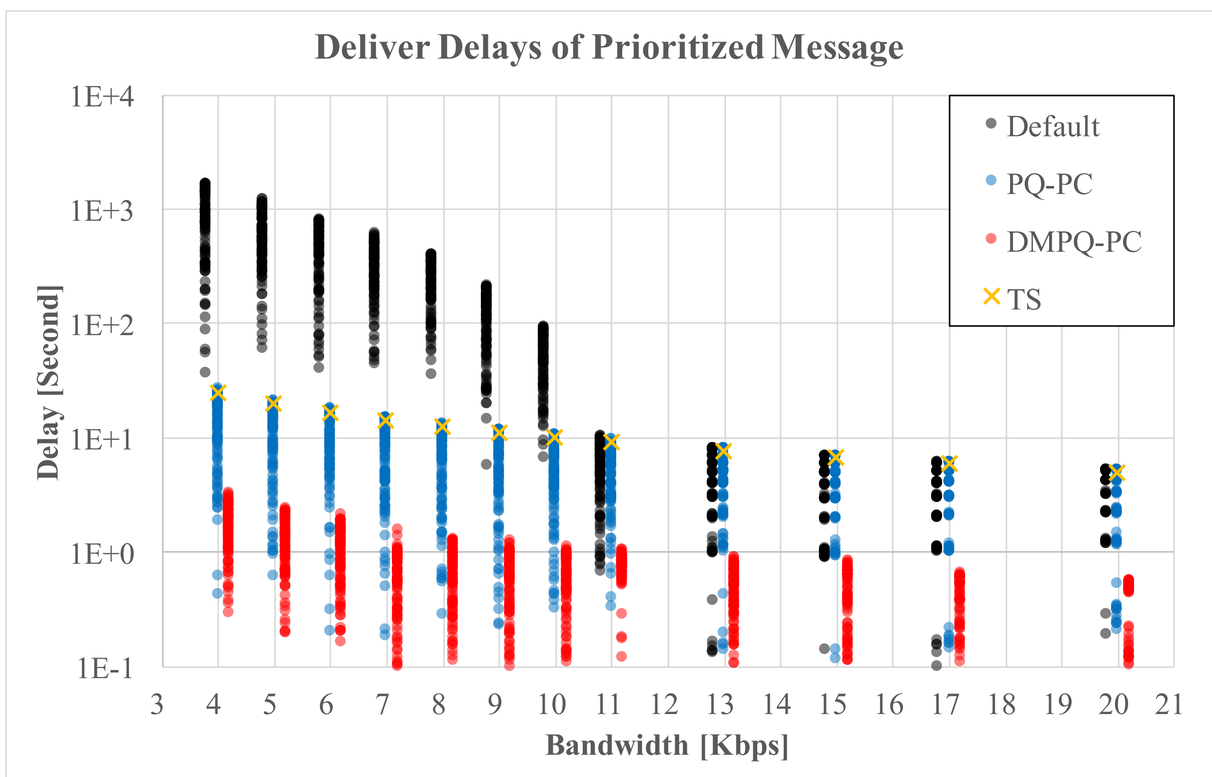


図 5.4 PM の配送遅延の結果

また，以下の図 5.5-5.10 は，帯域が 10-20 [Kbps]における配送遅延の累積分布である．以上の図 5.4 の結果からは，おおよその分布や最大遅延の評価のみが可能である一方，以下の図の結果からはより具体的な値の比較が可能である．ここで，10[Kbps]の結果のみ，帯域速度(x 軸)のスケールが異なることに注意する．

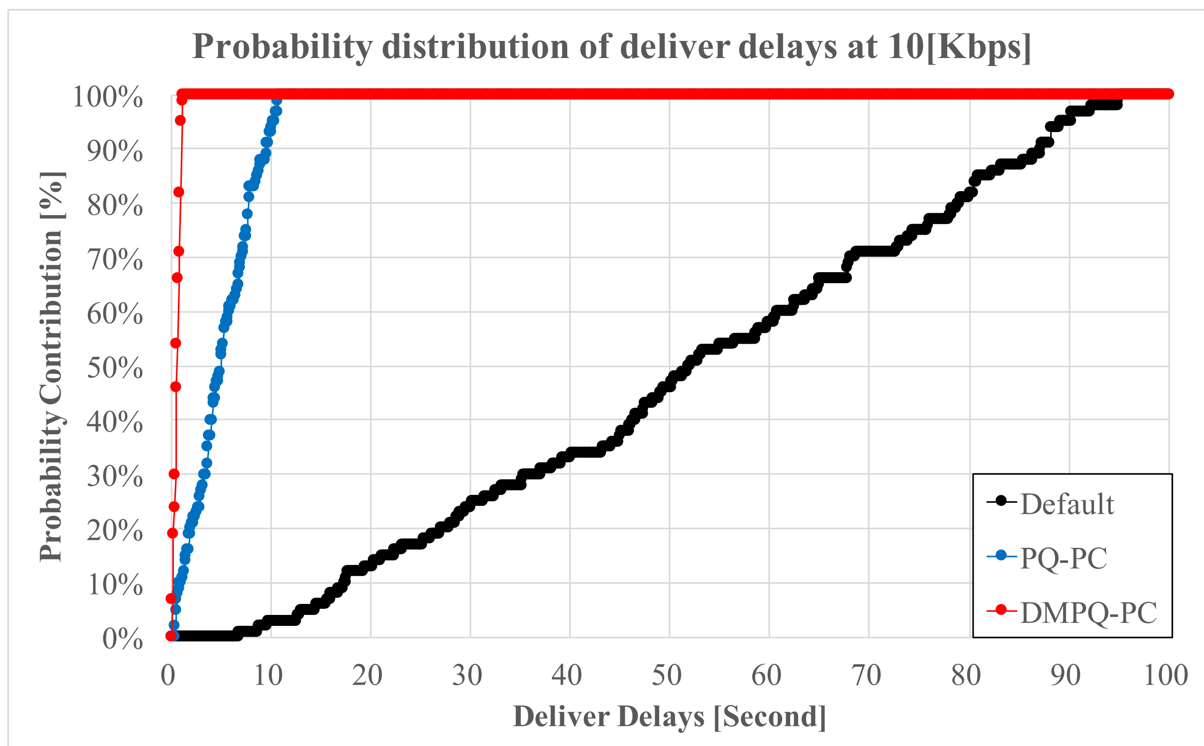


図 5.5 PM の配送遅延の累積分布（10 [Kbps]）

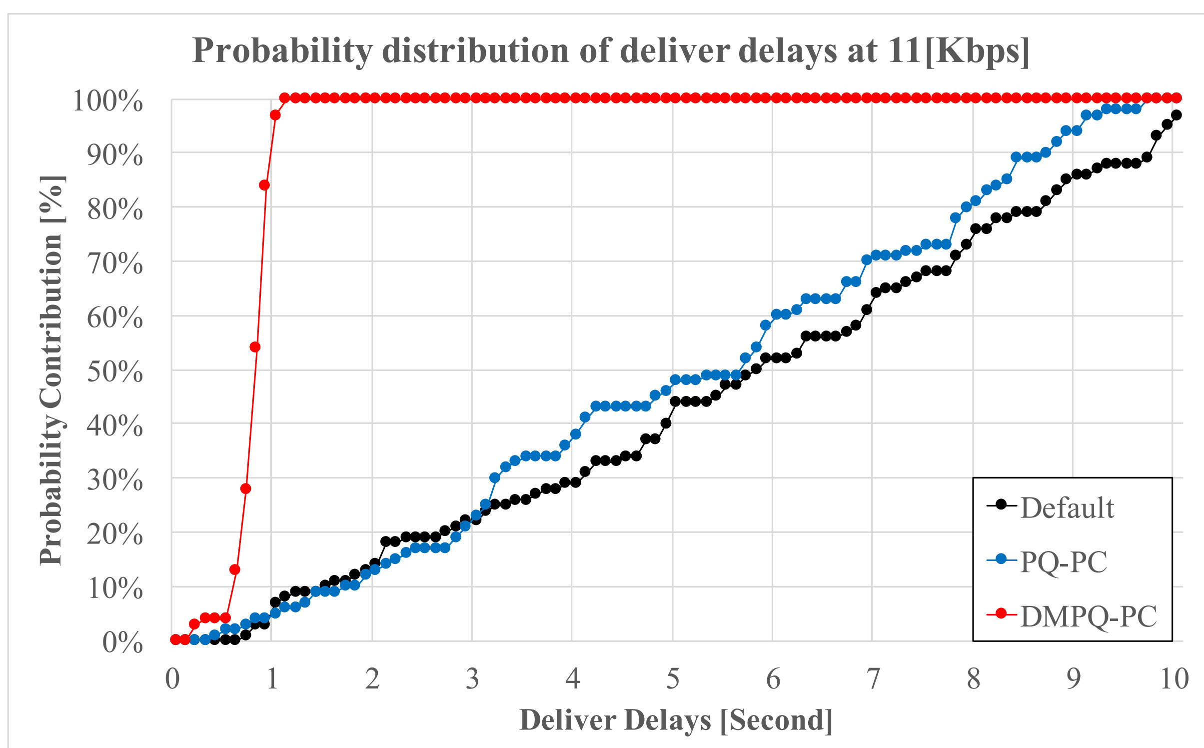


図 5.6 PM の配送遅延の累積分布（11 [Kbps]）

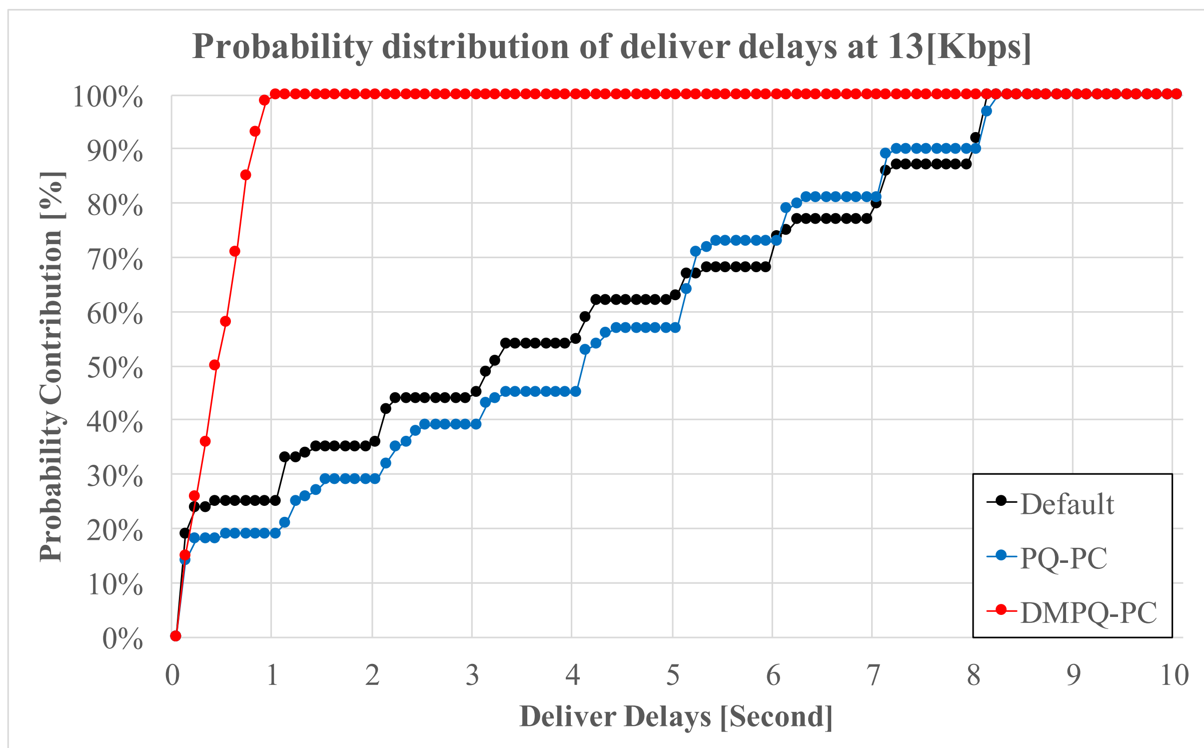


図 5.7 PM の配送遅延の累積分布（13 [Kbps]）

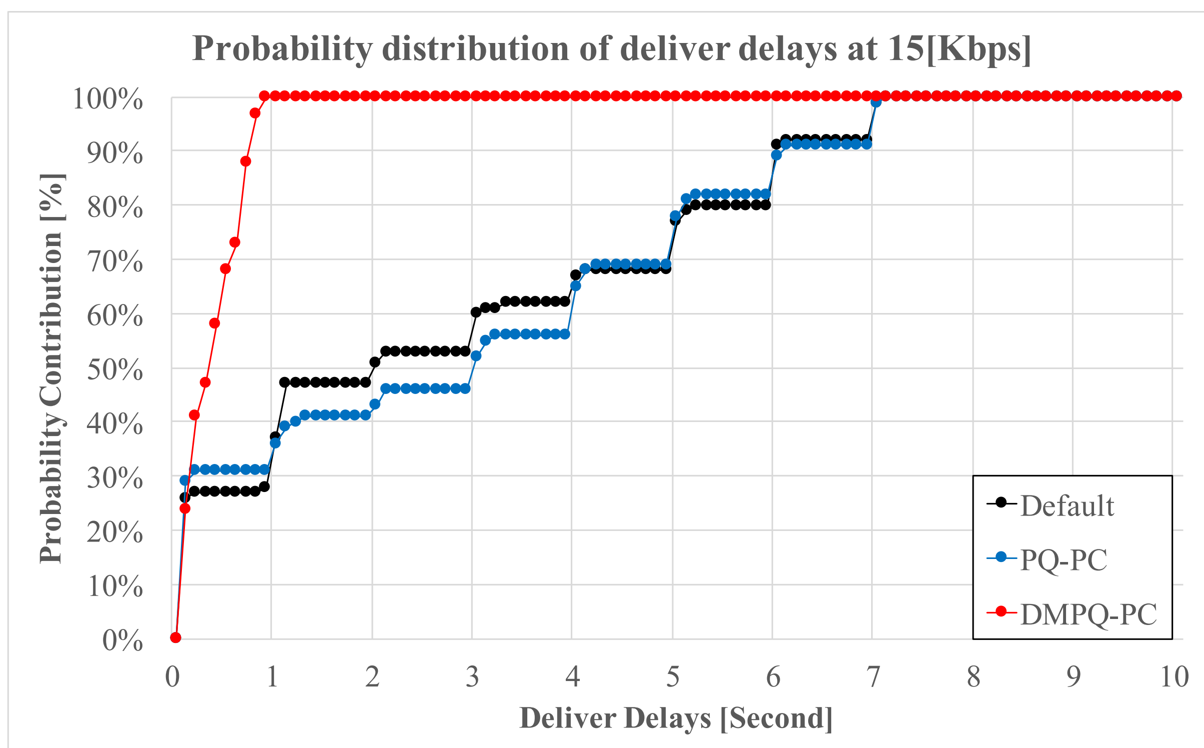


図 5.8 PM の配送遅延の累積分布（15 [Kbps]）

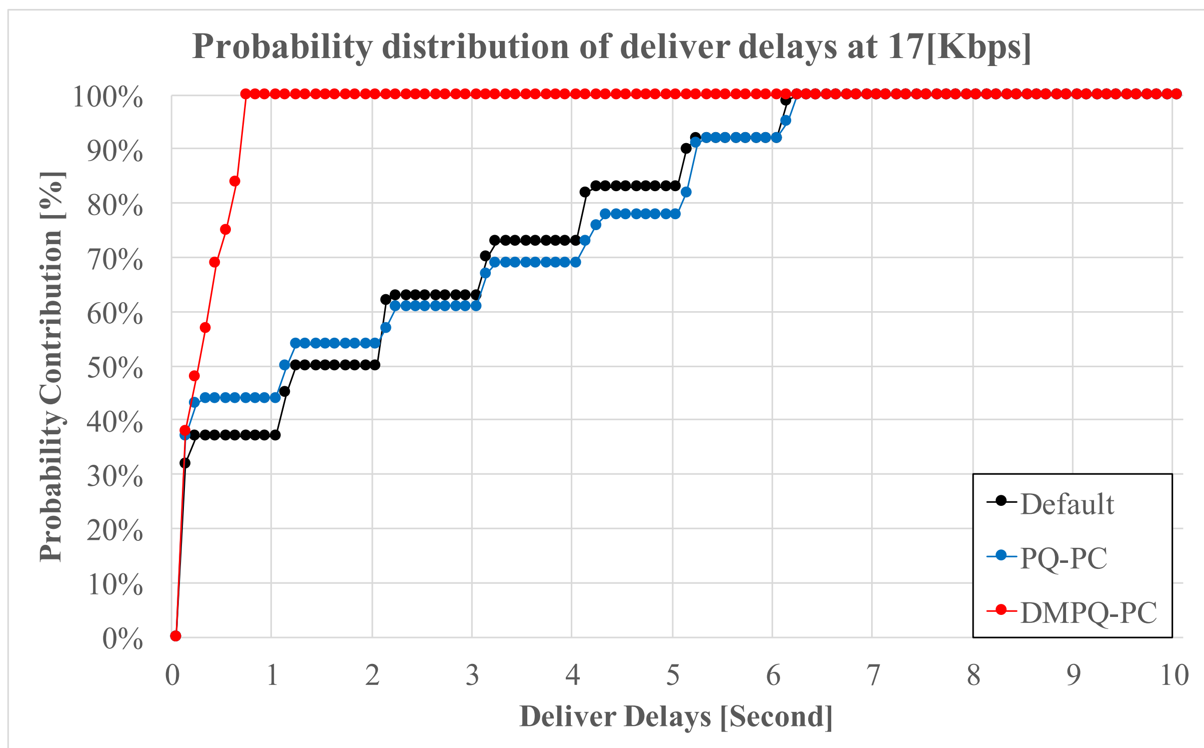


図 5.9 PM の配送遅延の累積分布（17 [Kbps]）

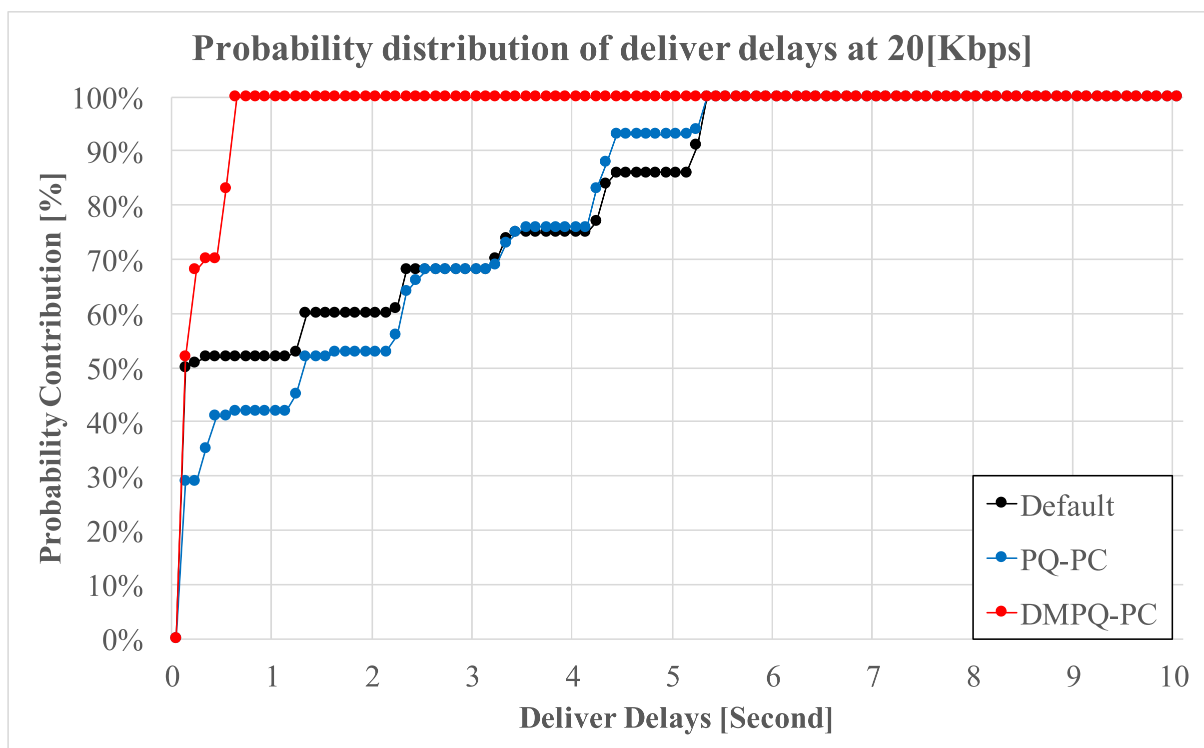


図 5.10 PM の配送遅延の累積分布（20 [Kbps]）

5.3.3 評価

図 5.4 – 図 5.10 の結果に関する評価を，手法ごとにわけて以下に示す．

(1) 従来手法の配送遅延

以上の図 5.4 の結果から注目すべき点は，帯域速度 10[Kbps]以下の場合，帯域速度が遅くなることにつれて遅延が急激に上昇し，その最大値も一定とならず上昇し続けた点である．この範囲は，以下に SM が回線を占有する占有率(Band Share: 以下，BS)を求める式(5.1)から，その割合が 100[%](帯域: 10[Kbps])を超える範囲であることがわかる．ここで，SM の生成される周期(Period)は PD とする．

$$BS\ [\%] = \frac{TS}{PD} \qquad \cdots\ (5.1)$$

SM による回線占有率が 100[%]を超えた範囲においては，一つの SM の送信が完了される前にさらなる SM が生成されてしまう．その結果，バッファの待機状態のメッセージが消費されていく速度より溜まっていく速度の方が早くなり，配送にかかる時間は増える一方となる．また，SM と PM の間に配送の優先度が存在しないため，その間に生成される PM は小さいメッセージであるにもかかわらず，その遅延は SM の遅延に依存して上昇される．この結果は，第 4 章における式(4.5)から見ても，実験が進むにつれてバッファの中で待機する SM の数 n_{WSM} が上昇するため，この結果は妥当であると考えられる．

一方，帯域速度が 10[Kbps]以上の場合，ある速度における配送遅延の最大値は一定となっており，その値は帯域速度が早くなることにつれて減少していくことがわかる．この範囲においては，SM の生成される時間より送出される時間が短く，それによって SM がバッファに溜まらずにそのまま送出される．その結果，PM の配送遅延に影響を与えるバッファ長は 0 となり，配送遅延の最大値は SM 一つの送信時間と PM の送信時間を足した時間となる．この結果は，第 4 章における式(4.4)から見ても，PM の最大遅延が TS に近い値となっていることから，妥当であると考えられる．

(2) PQ-PC の配送遅延

図 5.4 の PQ-PC における PM の配送遅延は，全範囲においてなめらかな曲線を描いており，その最大値は一定の値で抑えられていることがわかる．その結果，一定の帯域速度以下の範囲で遅延の最大値が急激に変化している従来手法と比べ，10[Kbps]以下の範囲においては遅延が抑制されている一方，それ以上の範囲においては最大遅延の面においてあまり改善が見られなかった．特に，図 5.5-図 5.10 からわかるように，最大値だけではなく遅延の分布もある帯域速度以上の範囲においては従来手法と比べて改善されていないこと

がわかる．

PQ-PC は PM が SM と違う独立したバッファに保存されているため，先に生成された複数の SM がバッファ SQ において待機状態でいるとしても，PQ に入っている PM は回線の空き次第で優先的に送られる．その結果，PM の配送遅延は SQ に入っている SM の数には依存せずに，回線の占有率と関係なく全範囲において SM の一つの送信時間のみに大きく依存する．この結果は，第 4 章における式(4.6)から見ても，PM の最大遅延が TS に近い値となっていることから，妥当であると考えられる．

このように PQ-PC は占有率が 100[%]以上に混む可能性がある回線においては有効的であり，その実装もサーバのみとなるため，手法の実現が比較的に容易であると考えられる．しかし，占有率が 100[%]を超えないほど混雑している回線においては改善の余地がない．また，式(4.6)からわかるように，SM 一つのサイズ $SIZE_{SM}$ がとても大きい場合を想定すると，その分 TS が大きくなり，PM の配送遅延の最大値も大きくなってしまう．

(3) DMPQ-PC の配送遅延

図 5.4 の DMPQ-PC 手法における PM の配送遅延は，全範囲においてなめらかな曲線を描いており，その最大値も従来手法や PQ-PC 手法より抑えられていることがわかる．つまり，全範囲において最大遅延が改善され，その時間もほぼ 1 秒前後である．図 5.5-図 5.10 の累積分布の結果を見ても，従来手法や PQ-PC より改善が見られるのはもちろん，帯域速度全てにおいて最大遅延が 1 秒前後で抑えられていることがわかる．また，その分布も 0 秒から 1 秒以内において比較的均等に分布されており，帯域速度が速くなることにつれてより改善されていることもわかる．

DMPQ-PC は PQ-PC と同様に，バッファ SQ に入っている SM の影響は受けない．さらに，SM がある一定のサイズ DV で分割されてバッファに保存されているため，元のサイズがいくら大きくても，送出中のメッセージの大きさは常に DV 以下となる（PM の大きさ $SIZE_{PM}$ が DV より小さいという前提である）．また，この DV が小さければ小さいほど PM を送ることのできる間隔が短くなり，その分 PM の配送遅延の最大値も減少する．具体的には，DV を調整することによって式(4.7)における T_{TM} が小さく制御できることを意味する．

5.4 通信量・パケット数の評価

クライアント側に届く通信量を測り，提案手法によってどれほどオーバーヘッドが生じたかの評価を行った．以下には実験の手順やその結果，そして結果に関する評価を示す．

5.4.1 手順

従来手法および提案手法にいて同じ数の PM と SM を送った時の通信量を求める具体的な手順は以下のようである．

(1) サーバ側において，tc コマンドを用いて帯域速度を設定する．

(2) クライアントおよびサーバ側において，ntpdate コマンドを実行して時刻を同期する．

(3) クライアント側において，以下のように tcpdump コマンドを実行する．

tcpdump -w filename.pcap

(4) 必要に応じて DV の値を変更した後，サーバのプログラムを実行する．

(default_server.py / pq_server.py / dmpq_server.py)

(5) クライアントのプログラムを実行する．

(default_client.py / pq_client.py / dmpq_client.py)

(6) tcpdump によって生成された pcap ファイルを，Wireshark から開く．

(7) 以下のように，Wireshark のフィルタを用いてアドレスを指定し，必要なパケットのみを選択する．

ip.dst == 203.178.135.104 && tcp.port == 9000

(8) フィルタリングされたパケットの結果における通信量(Bytes)やパケット数(Packets)を記録する．

(9) (1) ~ (8)を繰り返す．

5.4.2 結果

5.4.1 の手順に従って通信量やパケット数の評価を行い，その結果を以下に示す．サーバからの PM や SM の配送はそれぞれ 10 回と設定し，各帯域速度における通信量やパケット数をグラフにプロットした．特に，DMPQ-PC は DV が 10000[Byte]の時のみならず，5000[Byte]，1000[Byte]，1443[Byte]の時の結果もプロットし，そのグラフを以下の図 5.12，図 5.13 に示す．また，DMPQ (DV: 1000B)の結果は図の y 軸のスケールより大きい結果であるが，各データの相違がより明確にわかるように，スケールを縮小して表示した．さらに，再送されたパケットの通信量・パケット数の分については，グラフにおいて点線のボックスで表記した．

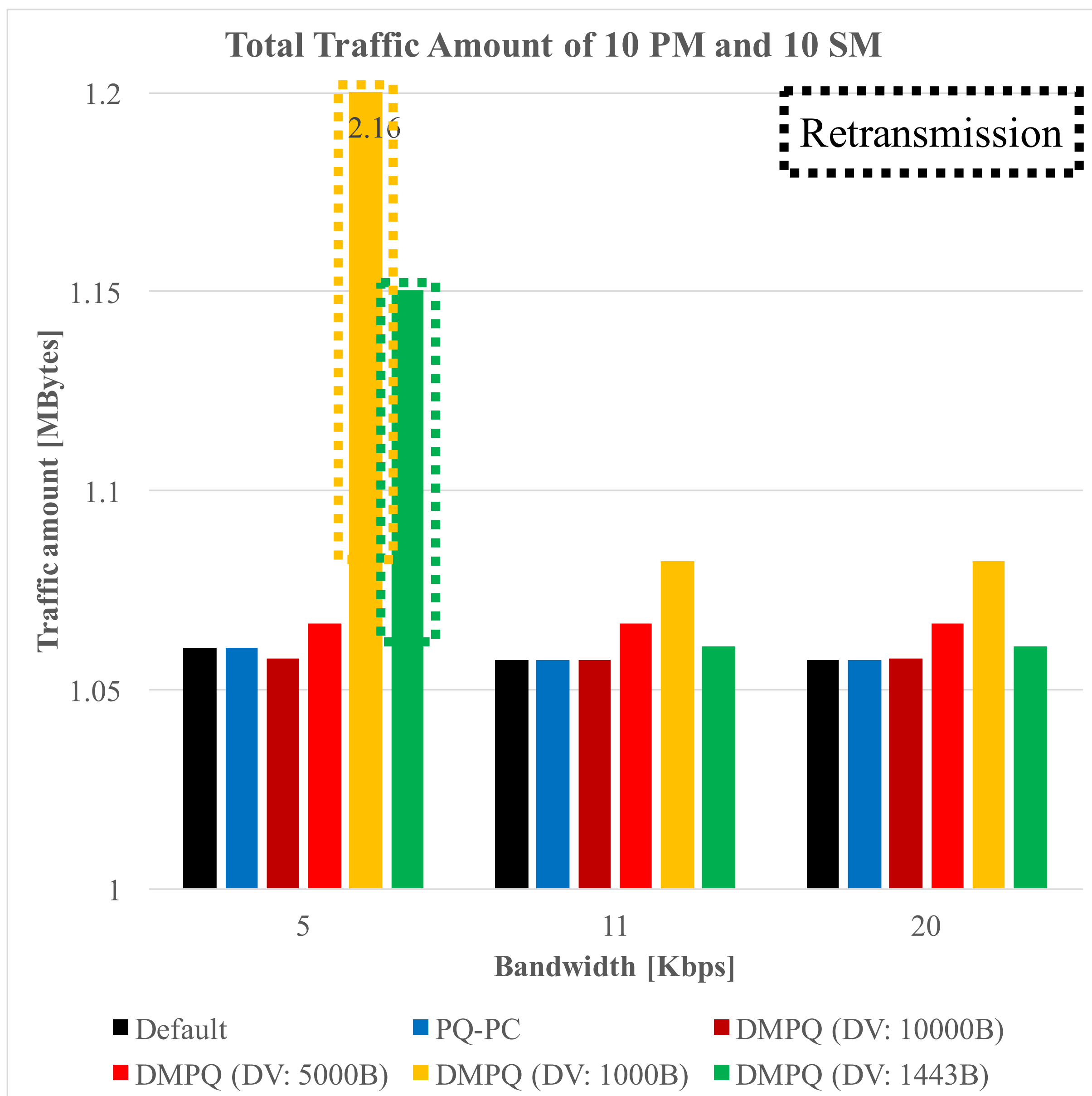


図 5.11 通信量の比較

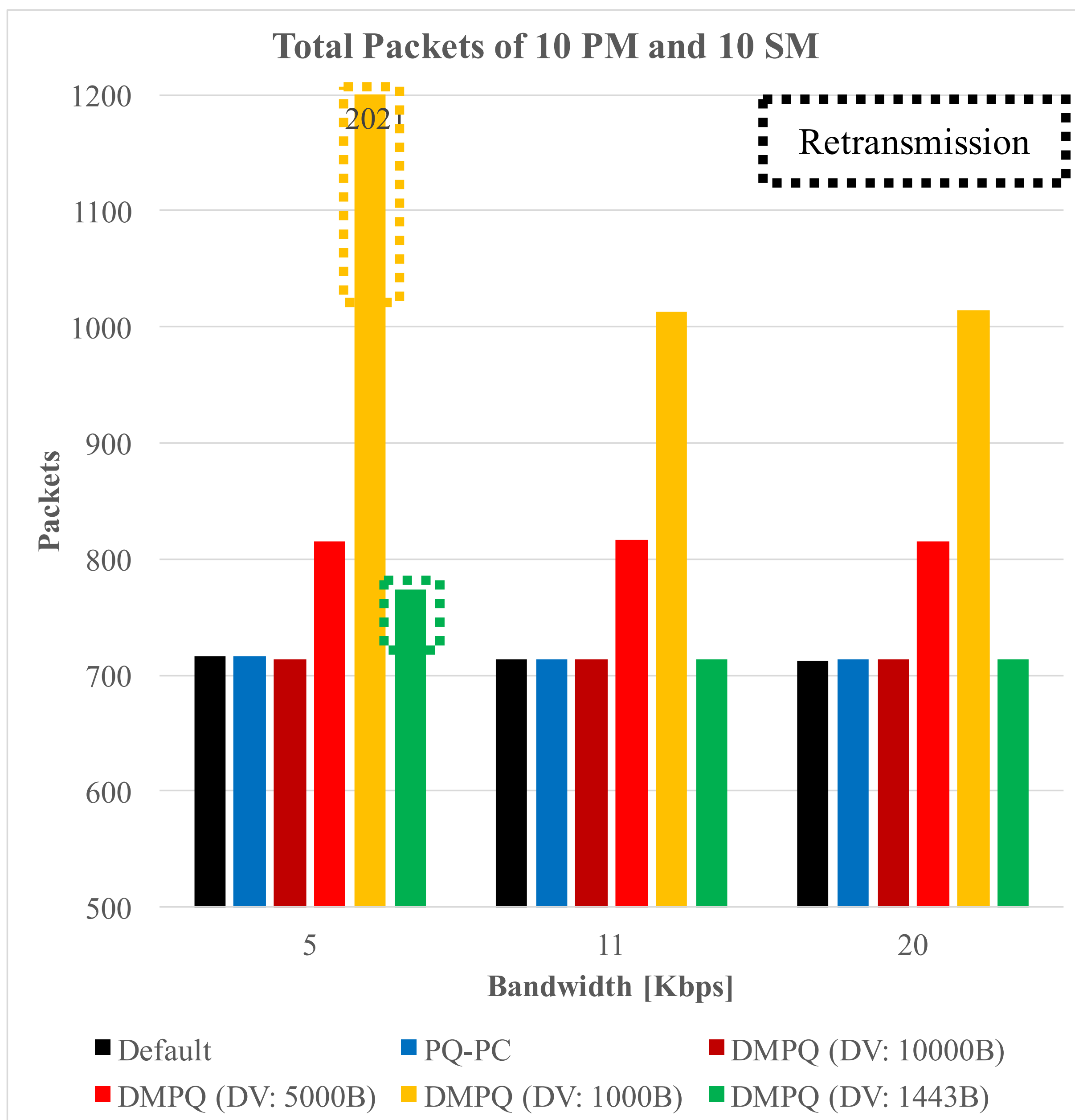


図 5.12 パケット数の比較

5.4.3 評価

図 5.11 – 図 5.12 の結果に関する評価を以下に示す.

(1) PQ-PC の通信量・パケット数

PQ-PC の場合，全範囲において従来と比べて通信量やパケットの増加分，すなわち，オーバーヘッドが発生しなかった．これは，従来とはメッセージの構造が変わらず，メッセージの送出順番が変わっただけなので，妥当な結果であると考えられる．また，帯域速度と関係なく同様なパケットの使い方をしていたと見られる．

(2) DMPQ-PC の通信量・パケット数

最初に，狭い帯域速度においては，DV 値が 1000 で一つの SM を 100 個に分けた場合，大量の再送が発生している．また，同様な帯域速度において，DV 値が 1443 の時も，1000 の時よりも少ないながら再送が発生している．一方，パケットロスを除外し，帯域速度ごとの結果を比べると，どの帯域速度においても通信量やパケット数においてほぼ同様な結果となっており，この結果から帯域速度と関係なく同様なパケットの使い方をしていたと見られる．次に，DV 値ごとに結果を比べると，DV 値が 10000 から 5000，1000 となることにつれて通信量・パケット数が上昇し，1443 の時は減少して 10000 の時とほぼ等しくなる．この結果に対する具体的な評価を DV 値ごとに分けて以下に示す.

DV 値が 10000 の時は，通信量・パケット数が従来と比べてほぼ変わらない結果となった．この結果を評価するために，「ラベルによるオーバーヘッド」と「パケットのヘッダーによるオーバーヘッド」に分けて考える．ラベルよるオーバーヘッドから考えると，一つの SM の分割された数は式(3.1)によって 10 個となり，ラベル一つは 1 [Byte]であるため，ラベルとして加わったデータが 10 [Byte]であることを意味する．これは 100 [Kbyte]の SM の 0.01 [%]に当たる大きさで，無視できるほど小さいものである．次に，パケットのヘッダーによるオーバーヘッドを考えるためには，必要なパケット数を求める必要がある．そこで，DV の長さで n 個に分割された一つの SM を送るに必要なパケットの数 n_{pk} を求めると，「分割メッセージに必要なパケット」に「分割された数 n-1」をかけ，「n 番目の分割メッセージに必要なパケット」を足した数になる．これを式で表すと，以下の式(5.2)のようになる．ここで，最大に収納できるデータはどのヘッダーが加わるかどうかによっても変わるので，ヘッダーが全て加わった時の最大のデータの長さである 1443[Byte]で計算し，$SIZE_{SM}$ はラベルを含めたものとした．また，DV が 10000，5000，1000，1443 [Byte]である時のパケットの数 n_{pk} を計算し，以下の表 5.6 にまとめた.

$$n_{pk} = \left\lceil \frac{DV}{1443} \right\rceil \times \left\lfloor \frac{SIZE_{SM}}{DV} \right\rfloor + \left\lceil \frac{SIZE_{SM}\%DV}{1443} \right\rceil \qquad \cdots (5.2)$$

表 5.6 DV 値による必要パケット数の変化

DV [Byte]	必要なパケット数
分割なし	70
10000	70
5000	80
1000	100
1443	70

すなわち，DV 値が 10000 であるの時は，メッセージは分割されたものの必要なパケットの数が同様であり，パケットのヘッダーによるオーバーヘッドが発生しなかったと考えられる．以上の結果から，通信量がほぼ変わらなかったと考えられる．

次に DV 値が 5000 の時は，通信量・パケット数が従来と比べて上昇した結果となった．ラベルよるオーバーヘッドは，一つの SM の分割された数はが(3.1)によって 20 個となり，ラベル一つは 1 [Byte]であるため，ラベルとして加わったデータが 20 [Byte]であることを意味する．これは 100 [Kbyte]の SM の 0.02 [%]に当たる大きさで，無視できるほど小さいものである．次に，パケットのヘッダーによるオーバーヘッドは，表 5.6 のように一つの SM に対して必要なパケットの数が 10 個増え，その分のパケットのヘッダーがオーバーヘッドになったと考えられる．

次に DV 値が 1000 の時は，通信量・パケット数が従来と比べて上昇し，DV 値が 5000 の時よりも上昇した結果となった．ラベルよるオーバーヘッドは，一つの SM の分割された数はが(3.1)によって 100 個となり，ラベル一つは 1 [Byte]であるため，ラベルとして加わったデータが 100 [Byte]であることを意味する．これは 100 [Kbyte]の SM の 0.1 [%]に当たる大きさで，無視できるほど小さいものである．次に，パケットのヘッダーによるオーバーヘッドは，表 5.6 のように一つの SM に対して必要なパケットの数が従来よりも 30 個増え，DV 値が 5000 の時よりも 20 個が増える．その分のパケットのヘッダーがオーバーヘッドになり，最も多い通信量・パケット数の結果になったと考えられる．

最後に，DV 値が 1443 の時は，通信量・パケット数が従来と比べて増えたものの，その差は大きくなく，DV 値が 5000 や 1000 の時より少ない結果となった．ラベルよるオーバーヘッドは，一つの SM の分割された数はが(3.1)によって 70 個となり，ラベル一つは 1 [Byte]であるため，ラベルとして加わったデータが 70 [Byte]であることを意味する．これは 100 [Kbyte]の SM の 0.07 [%]に当たる大きさで，無視できるほど小さい大きさである．また，パケットのヘッダーによるオーバーヘッドは，表 5.6 のように一つの SM に対して必要なパケットの数が従来と変わらず，この通りであれば従来との差はほぼないと考えられる．しかし，式(5.2)はパケットに全てのヘッダーを含まれた時の式であり，実際とは少し異なる．図 4.2 の WebSocket ヘッダーの場合，実際には分割メッセージの最初のパケッ

トにのみ加わることが，Wireshark の結果から分かった．その結果，DV 値が 1443 の場合は 4 [Byte]の WebSocket ヘッダーが全てのパケットに含まれる一方，DV 値が 10000 の場合は分割メッセージの途中の部分を運ぶパケットに WebSocket ヘッダーが含まれないため，その分の差が出てくる．実際に DV 値が 1443 と 10000 の時の結果をその数値まで具体的に比べると，例えば帯域速度が 20 [Kbps]の時，パケットの数は「714」で変わらない一方，通信量は「3000 [Byte] 」の差があり，一つの SM あたり「300 [Byte]」の差があることになる．これは，ラベル数の差 60 (60*1=60 [Byte])に，分割メッセージの数の差 60 (60*4=240 [Byte]) を足した値と等しく，結果は妥当であると考えられる．

以上の結果から，DV 値によって過剰なオーバーヘッドが発生することもあるが，パケットに入るデータの長さを考慮し，効率的にパケットを利用する DV 値を設定することにより，オーバーヘッドを抑えることが可能であることがわかった．

5.5 分割による遅延抑制の評価

分割単位である DV 値を変化させながら PM の配送遅延やクライアントへ届く通信量を測る．これにより，式(4.7)のように DV 値が小さくなればなるほど，PM の配送遅延が減少することを確認する．また，パケットサイズを考慮した DV 値を設定した時の通信量のオーバーヘッドの評価も行う．以下には実験の手順やその結果，そして結果に関する評価を示す．

5.5.1 手順

評価実験 5.3，評価実験 5.4 の手順を用いる．

(1) サーバ側において，tc コマンドを用いて帯域速度を設定する．
(2) クライアントおよびサーバ側において，ntpdate コマンドを実行して時刻を同期する．
(3) クライアント側において，以下のように tcpdump コマンドを実行する．
(4) 必要に応じて DV の値を変更した後，サーバのプログラム dmpq_server.py を実行する．
(5) クライアントのプログラムを実行する．
(6) tcpdump によって生成された pcap ファイルを，Wireshark から開く．
(7) Wireshark のフィルタを用いてアドレスを指定し，必要なパケットのみを選択する．
(8) フィルタリングされたパケットの結果における通信量(Bytes)やパケット数(Packets)を記録する．
(9) (1) ~ (8)を繰り返す．

5.5.2 結果

5.5.1 の手順に従って DV 値を変化させながら PM の配送遅延の評価を行い，その結果を以下に示す．PM の配送を 100 回行い，生成された時刻からクライアントに届いた時刻までの時間を配送遅延とし，それをグラフにプロットしたものである．ここで，結果を見易くするためにそれ

それの横軸のデータを少しずつずらして表記した．また，各帯域速度や DV 値を近似式(4.7)に入れた結果も一緒にプロットして比較を行った．

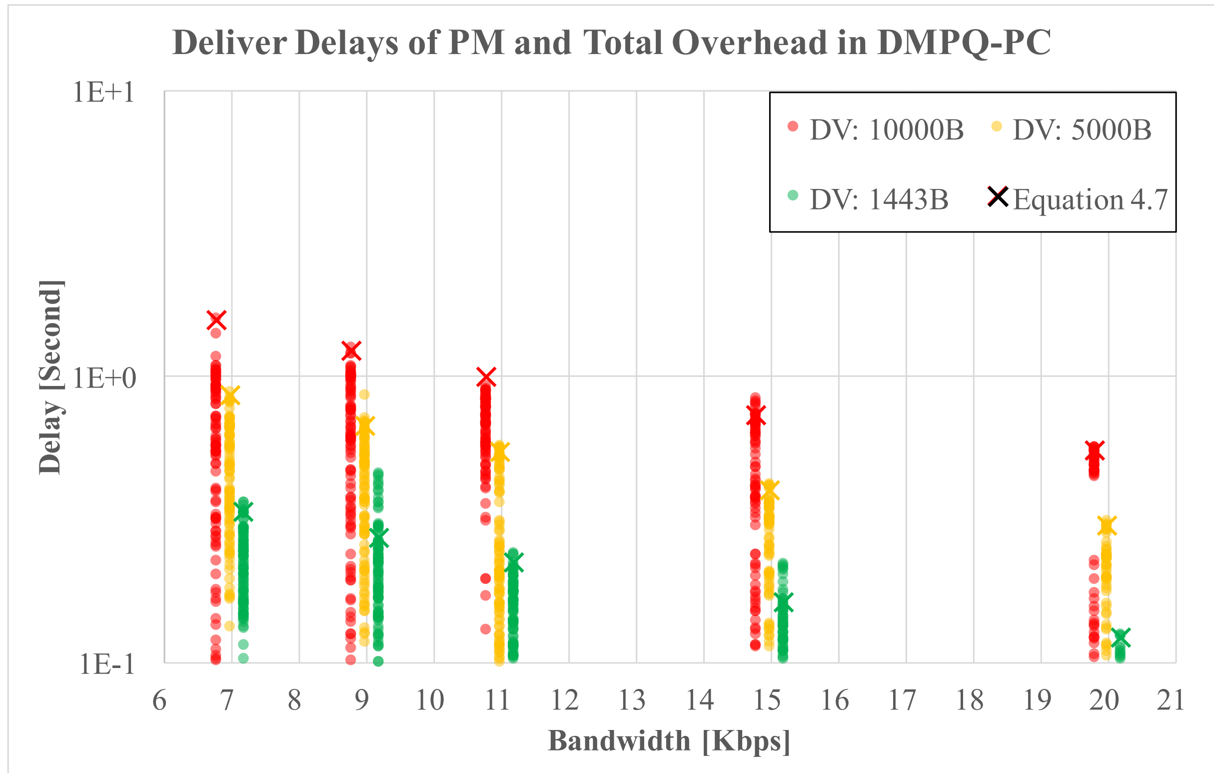


図 5.13 異なる DV 値における PM の配送遅延

また，以下の図 5.14-18 は，帯域が 7-20 [Kbps]における配送遅延の累積分布である．以上の図 5.13 の結果より具体的な値の比較が可能である．

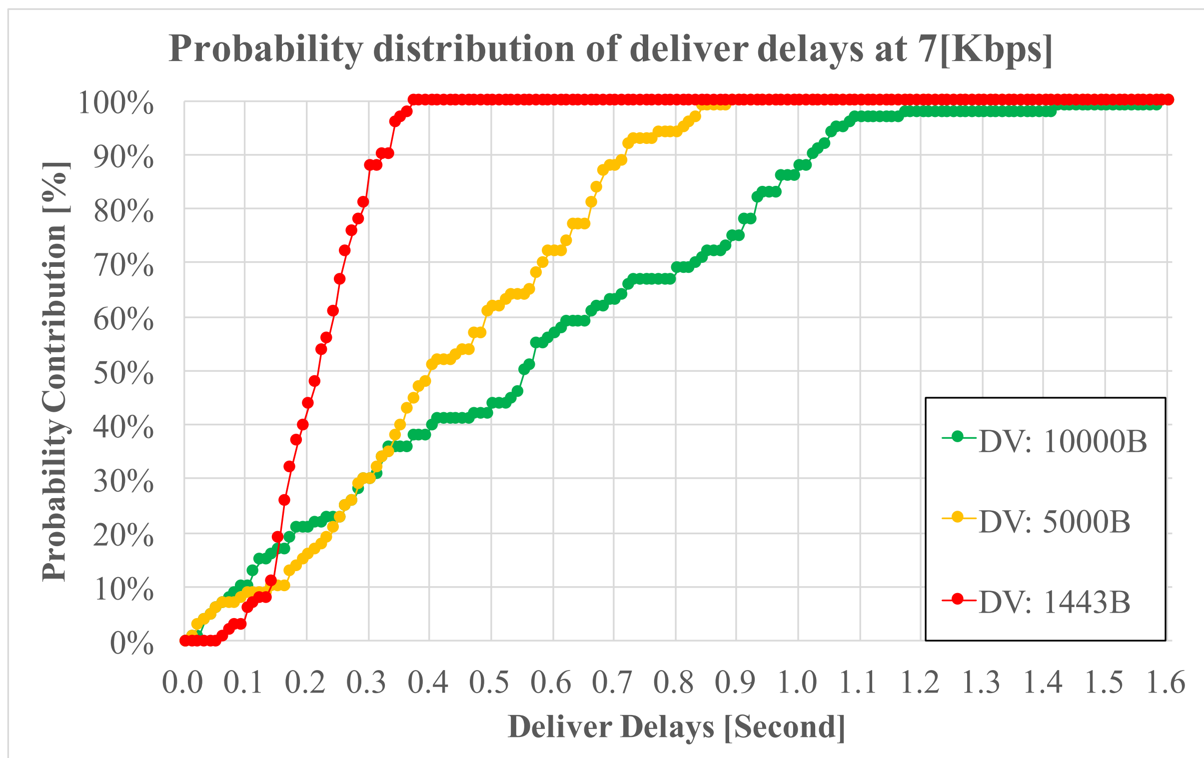


図 5.14 DMPQ-PC における PM の配送遅延の累積分布（7 [Kbps]）

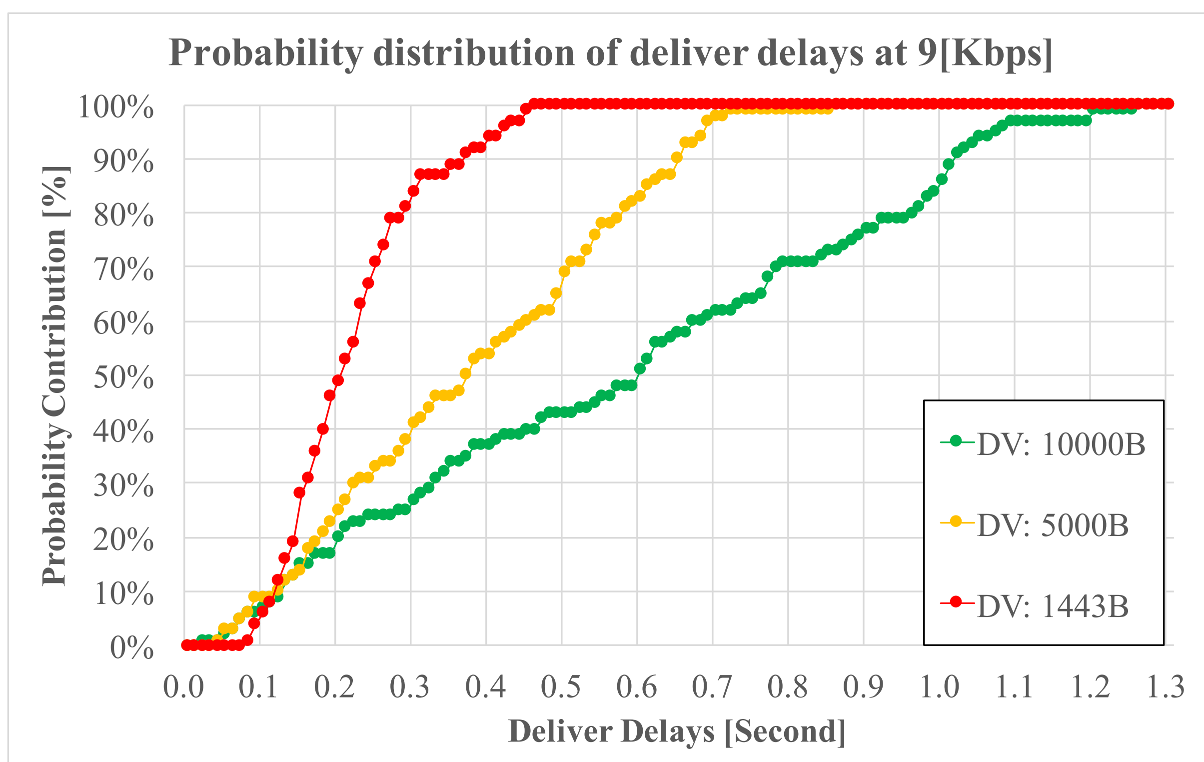


図 5.15 DMPQ-PC における PM の配送遅延の累積分布（9 [Kbps]）

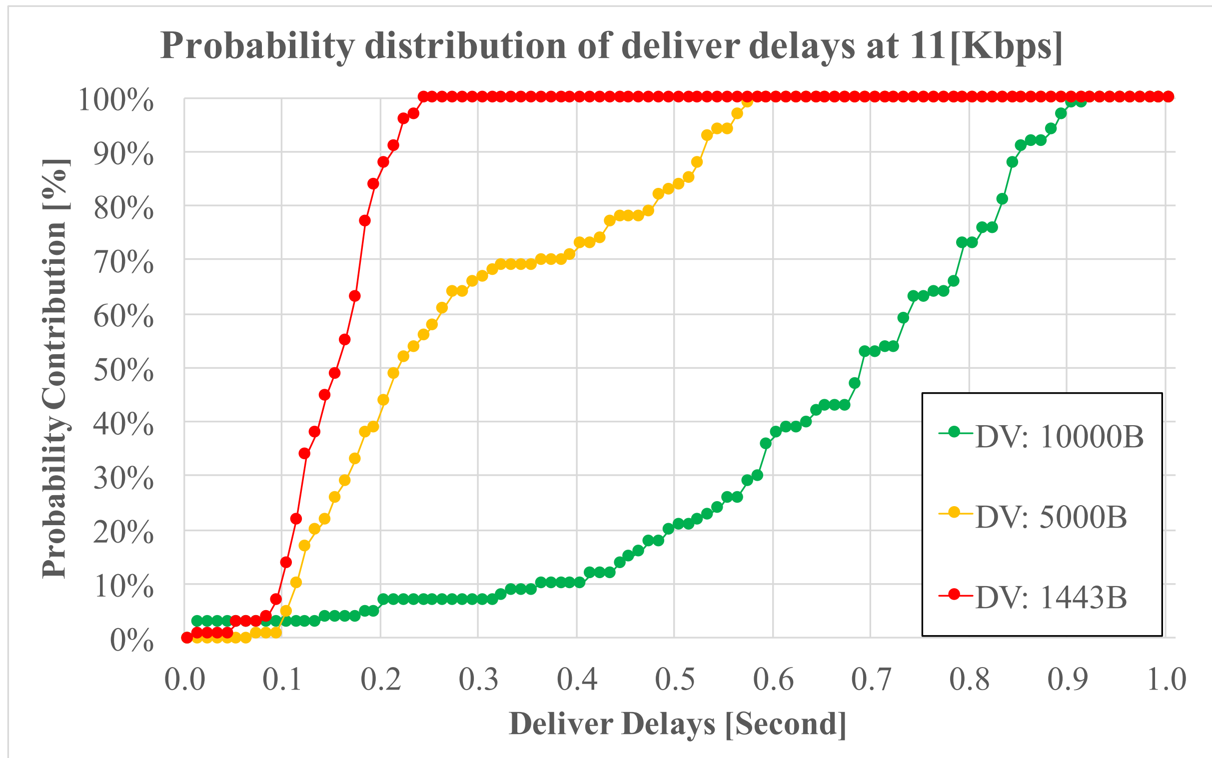


図 5.16 DMPQ-PC における PM の配送遅延の累積分布（11 [Kbps]）

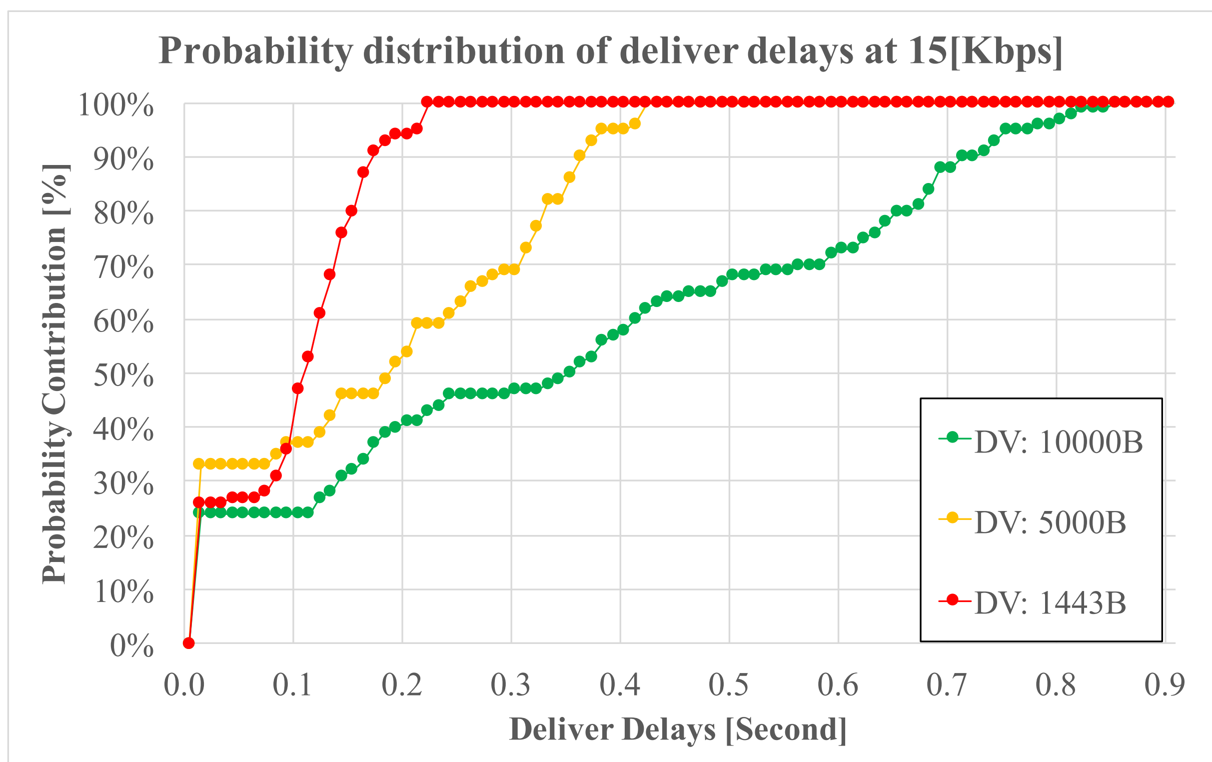


図 5.17 DMPQ-PC における PM の配送遅延の累積分布（15 [Kbps]）

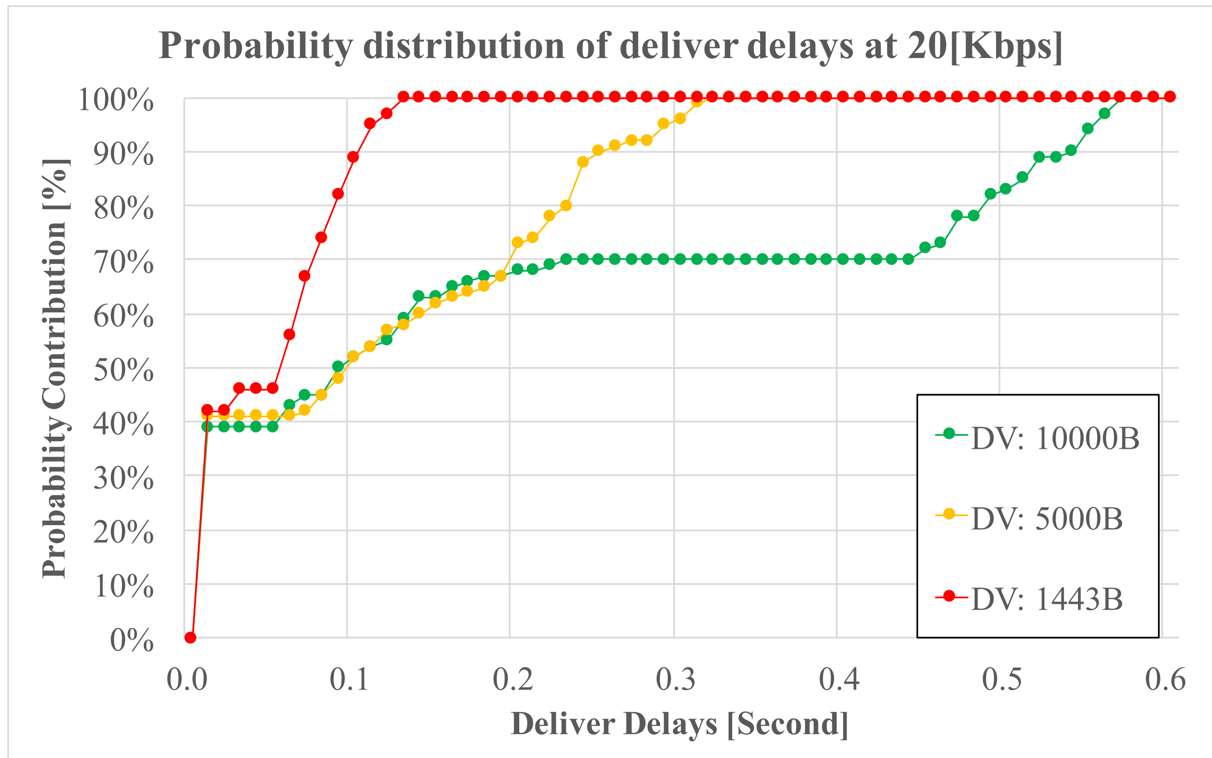


図 5.18 DMPQ-PC における PM の配送遅延の累積分布（20 [Kbps]）

5.5.3 評価

図 5.13-18 の結果から，DV 値が小さいほど PM の配送遅延が減少されていることがわかる．これは上で述べたように，SM の大きさに依存せずに，DV 値を変化させることによって配送遅延が制御できることを意味する．さらに，図 5.4 の結果と合わせて考えると，あまりオーバーヘッドを起こさずにも，配送遅延の抑制が可能であることを意味する．配送遅延の最大値も近似式(4.7)の値とほぼ一致し，結果は妥当であると考えられる．

第6章　考察

本研究では、第 5 章の評価実験から PQ-PC や DMPQ-PC による配送遅延の抑制を確認した．しかし，実際の導入環境や違うアプローチなど、他にも考慮すべき点がいくつかあり，それを以下に述べる．

6.1 クライアント･サーバの双方実装

「占有率」にも「SM の大きさ $SIZE_{SM}$ 」に依存せずに最大配送遅延の抑制ができる DMPQ-PC であるが，クライアント・サーバの双方の実装が必要になることが一つの問題点である．PQ-PC がサーバのみの実装で実現可能である一方，DMPQ-PC はクライアントの実装も必要となってくる分、実装に当たるコストが上昇してしまう．通常，一つのサーバに対して複数のクライアントが存在することを考えると，そのコストが大きい負担となる可能性がある．したがって、混雑する可能性がある回線であっても，通常やり取りするメッセージ SM の大きさがそこまで大きくならない回線の場合、PQ-PC の方がより効率的であると言える．

6.2 ラベルの導入

PQ-PC においてはメッセージがそのまま送られてくる一方，DMPQ-PC においては SM が一時的に分割されてから復元されるプロセスがあり、それには分割メッセージにラベルを加える方式を取っている。それは従来になかったラベルのオーバーヘッドにつながるが、提案する手法は 4 種類のラベルしか用いていないため、たった 2[bit]のみで実現できる。

これは以下の表 6.1 のように WebSocket ヘッダーの opcode の未使用部分を用いることも考えられ、そうすることによってラベル分の通信量のオーバーヘッドがほぼゼロに近い形で実現できる。

表 6.1 WebSocket ヘッダーの opcode

opcode (4bit)	意味・フレームのタイプ
0	Continuation frame
1	Text (UTF-8) frame
2	Binary frame
3-7	**Reserved for further non-control frames**
8	Connection Close
9	Ping
10	Pong
11-15	**Reserved for further non-control frames**

6.3 実際のマシーンとの相違点

提案手法は従来手法に対して新たな処理過程が加わっており，マシーンの処理能力や処理時間がもたらす影響については本研究で扱っていない．特に，M2M の通信におけるマシーンは本研究で使われた Laptop のようなマシーンではない小型の組み込み型のマシーンの場合も多い．このような場合，メモリ空間やプロセッサーなどに制約があり，本研究の提案手法の実現ができない可能性がある．具体的には，以下のようなものが考えられる．

(1) PQ を設けるための Storage が確保できるか.

本研究では評価を行うに十分な Storage があったが，実際の導入に当たってはこのような Storage が確保できない可能性もある．特に，DMPQ-PC の場合，サーバだけでなくクライアントにおいてもメッセージを復元するためのバッファが必要となり，M2M におけるクライアントはこういった Storage が確保できない小さいマシーンである場合もある．

したがって，帯域速度・メッセージ容量・メッセージ発生周期などによってどれほどの Storage が必要となるかを考慮しなければならない．

(2) メッセージを判別し，PQ や SQ に分別・分割・復元する処理が可能であるか.

本研究においてはサーバにおけるメッセージの分別・分割・復元といった処理が常に可能であり，その処理時間もほぼ 0 に等しいという前提で行った．しかし，処理能力やアーキテクチャの構造上，この処理を組み込めない可能性もある．

特に、DMPQ-PC のクライアントにおいては，WebSocket を用いて受け取ったメッセージを判別し，PQ や SQ に分別・復元する処理が必要となるが，M2M のクライアントのマシーンはこういった処理ができない小型マシーンも多い．

したがって，このような環境も想定したさらなる実証が行われる必要がある。

6.4 ポートを用いた優先制御

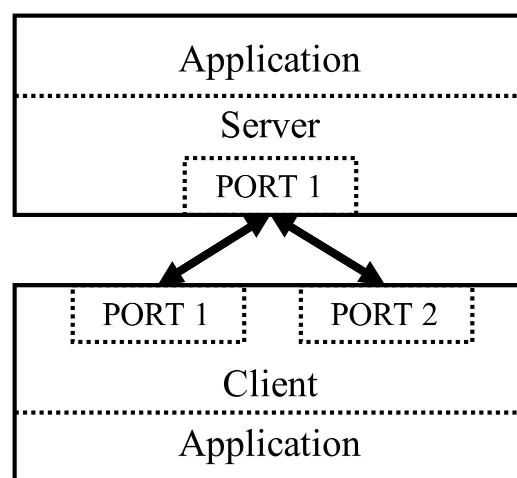


図 6.1 二つのポートを用いたアプローチ

本研究では追加のバッファを用いた優先制御を行ったが、WebSocket における優先制御を実現する方法として，以上の図 6.1 のようにサーバとクライアント間に複数のポートを用いた WebSocket セッションをはり，それぞれに優先度を割り当てる方式も考えられる.

しかし，この方式は以下の二つの課題があると考えられるため、本研究では違うアプローチを取った。

(1) TCP Session の数によってサーバにおける課金が増加するケースがある。

(2) アプリケーション層から TCP/IP 層への制御が必要となって仕組みが煩雑になる。

6.5 TCP バッファの利用

最初に、TCP カーネルのバッファ利用について検討する必要がある。TCP バッファが自由に制御でき、メッセージの送信状況もわかると、PQ-PC の実装がより簡単になる。少なくても、送信完了の情報さえ取得できると、クライアントから UDP パケットを返す必要がなくなる。ただし、他のアプリケーションからの TCP パケットをどう扱うべきかという議論は残る.

6.6 Ethernet Jumbo Frame

本研究の実験においては、Maximum Transmission Unit(MTU)が 1500[Byte]の時しか想定せず、これが 9000[Byte]となる Ethernet Jumbo Frame をサポートしていない環境において実験を行ったが、これをサポートする場合は、パケットに入る最大のメッセージの長さが変わる。その結果、メッセージに入る最大の長さで分割するという考え方は変わらないが、実験を行った 1443[Byte]より長くて効率的な分割単位が存在することになる。したがって、Ethernet Jumbo Frame をサポートするかしないかにより、分割単位を切り替えられる機能があるとより効率的な手法となる.

6.7 他の通信による影響

本研究においては、通信の途中でメッセージの順番が変わらない TCP 通信の特徴を用い，他の通信が実験の通信に影響を与えないという前提で行われた．しかし，実際の導入に当たっては他の通信のメッセージがこの回線を通る可能性もあり，TCP Urgent[40]のようなパケットは実験に大きく影響を与える可能がある。そういった環境においてどのように動作するかの実証が必要である.

6.8 パケットロスによる待機遅延の改善

本研究の提案手法では、パケットロスによる再送が原因である回線上の遅延というものは考慮していなかった。具体的な状況としては、普通メッセージのパケットロスが起きてその再送が繰り返されている時に、緊急メッセージが発生し、普通メッセージの再送より優先的に送る必要が出てくる。そのような状況では、TCP Buffer の送信状況を参照しながら、時には中身を直接取

り出す制御などが必要となり、今回の提案手法でそういった機能を追加した方が様々な状況に対してより柔軟に対応できるようになる。

6.9 優先度の管理

本研究においては、優先度の分類を「緊急」と「普通」の 2 種類でしか分類しなかった。しかし、実世界においては優先度をより細分化する必要がある状況も存在するため、優先度のレベルを増やすことが考えられる。その場合、ラベルの種類は増えるが、TCP に着目した今回の考え方を同様に適用した場合、ラベルの情報量が増えたとしても一桁のビット単位であると考えられる。提案手法のアーキテクチャからしても、TCP バッファの中にメッセージを置かない構造なので、異なる優先度を与えたキューを増やすことは容易であり、今後の課題の一つとして設定すべきことである。

6.10 SPDY との比較

QUIC[41]や SPDY[42]を用いたマルチストリームの制御により、本研究における目的である「メッセージごとの優先制度」は可能であり、それによる通信量の増加もほとんどない。しかし、以下ののように、本研究との相違点が存在する。

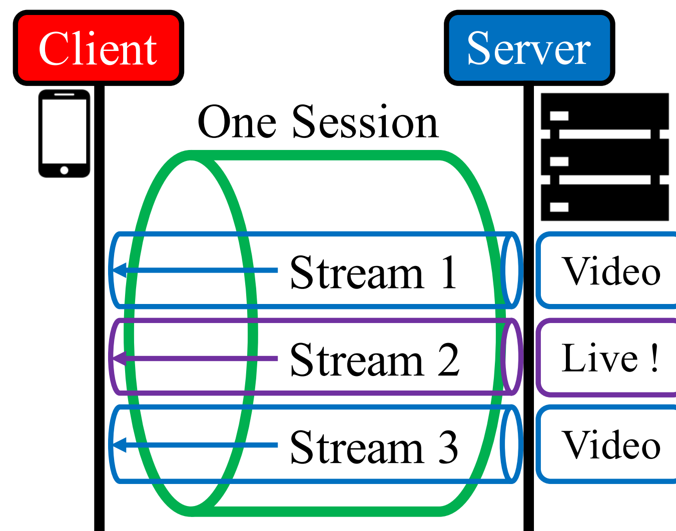


図 6.2 QUIC や SPDY のマルチストリームの制御

最初に、実装の面において異なる点が挙げられる。SPDY は M2M/IoT におけるデバイスのメッセージを想定していたものではなく、Web ページといったサービスの提供側であるCSP(Contents Service Provider)の運用環境や要求条件を満たせるために作られたものである。想定している環境が違うため、M2M 通信でよく使われる Block Message ではなく、Stream

Message が主に扱うメッセージとなっている。これにより、M2M 通信でこのプロトコルを使うためには、多くの場面においてアプリケーションを全体的に変更する必要あると考えられる。一方、本研究における提案手法は、アプリケーションレイヤーと TCP レイヤーに変化を加えていないため、TCP Socket と WebSocket の間に本研究の提案したモジュールをミドルウェアとして差し込むのみである。その結果、既存のアプリケーションをほぼそのまま利用することが可能であり、SPDY と比べて実装が比較的に用意であると考えられる。

次に、優先制御のやり方において異なる点が挙げられる。本研究においては、「緊急メッセージを送信している時は、一つのセッションを全て使って送る」という方式で、結果的には普通のメッセージはブロックされることになる。一方、SPDY の場合、優先度が低いものはブロックされるということが起きる単位が、セッション単位ではなくストリーム単位である。ストリームごとに 8[bit]の重み(Weight)をつけて優先度の管理は可能であるが、それは優先度の高いストリームが低いストリームを完全にブロックするような仕様ではない。すなわち、優先度の高いメッセージだとしても、一つのセッションを全て占有はしないといったことが異なる。

第7章　結論

本研究では，より多くの場面において，コスト抑えながら WebSocket 通信が活用できるように，WebSocket 通信における緊急メッセージの優先配送手法を二つ提案し，配送遅延や通信量の評価を行った．

一つの手法は優先度によって分けてキューイングを行う手法であり，もう一つの手法は最初の手法にメッセージの分割を加えた手法であった．

この時，TCP バッファが制御できない状況を考え，制御可能な別のキューを設け，TCP バッファが空いたらそこにメッセージを流す制御を行った．そうすることにより，TCP バッファにあるメッセージは送信中のメッセージのみとなり，このバッファが制御できない問題を解決した．

また，分割メッセージに追加するラベルの情報量を抑えるべく，TCP 通信の特徴に着目して 2[bit]の情報のみを用いた．さらに，パケットに入る最大のサイズで分割することにより，パケットヘッダーによる通信量の増加を防いだ．その結果，提案手法が緊急メッセージの遅延を抑制させながら，通信量のオーバーヘッドも少ないことを確認した．

二つの提案手法による遅延抑制の具体的な結果は, 以下の表 7.1 のようになった. 本研究では, 100[Kbyte]のメッセージが 10[Second]ごとに発生するシステムを想定し, その中でランダムに発生する 1[Kbyte]の緊急メッセージの遅延を 100[回]測定した.

具体的に結果を見ると，メッセージがバッファに蓄積されていく 7[Kbps]の時は，緊急メッセージの配送に最大 622.48[Second]以上かかった従来と比べ，PQ-PC では 15.20[Second]以内に配送ができ，分割単位が 10000[Byte]の DMPQ-PC では 1.59[Second]以内に配送できた．さらに，分割単位が 1443[Byte]の DMPQ-PC では，0.37 [Second]以内で配送でき，その時の通信量は 0.3[%]しか上昇しない結果となった．一方，メッセージが蓄積されない 20[Kbps]の時は，PQ-PC では改善が見られず，DMPQ-PC は分割単位が小さい場合に遅延がより抑制された．すなわち，パケットに最大に入る長さで分割することで，オーバーヘッドもほぼ発生させず，遅延を抑制することができた．

表 7.1 緊急メッセージ配送の最大遅延の結果［Second］

帯域速度［Kbps］	**従来手法**	**PQ-PC**	**DMPQ-PC (10000[Byte])**	**DMPQ-PC (1443[Byte])**
7	622.48	15.20	1.59	0.37
20	5.28	5.26	0.57	0.13
通信量の増加率[%]	100.00	100.00	100.00	100.33

発表文献と研究活動

(1) 李聖年，落合秀也，江崎浩．WebSocket 通信における緊急メッセージの優先配送の手法と評価．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（IA），2016.01.29 発表，学生研究奨励賞　受賞

参考文献

[1] "Hypertext Transfer Protocol -- HTTP/1.1," ed, June 1999.

[2] P. Fraternali, G. Rossi, and F. Sánchez-Figueroa, "Rich internet applications," Internet Computing, IEEE, vol. 14, pp. 9-12, 2010.

[3] K. M. Bell, D. N. Bleau, and J. T. Davey, "Push notification service," ed: Google Patents, 2011.

[4] (2016). Yahoo!防災速報（無料）. Available: http://emg.yahoo.co.jp/

[5] K. Richard Clark, "Stock Trading via WebSockets, Stomp, and JMS," 2016.

[6] S. Loreto, P. Saint-Andre, S. Salsano, and G. Wilkins, "Known issues and best practices for the use of long polling and streaming in bidirectional http," Internet Engineering Task Force, Request for Comments, vol. 6202, p. 32, 2011.

[7] P. Lubbers, B. Albers, F. Salim, and T. Pye, Pro HTML5 programming: Springer, 2011.

[8] N. Sharma, "Push technology–long polling," URL: http://www. ijcsmr. org/vol2issue5/paper398. pdf [cited 18 July 2013], 2013.

[9] A. Biral, M. Centenaro, A. Zanella, L. Vangelista, and M. Zorzi, "The challenges of M2M massive access in wireless cellular networks," Digital Communications and Networks, vol. 1, pp. 1-19, 2015.

[10] 藤田隆史, 後藤良則, and 小池新, "M2M アーキテクチャと技術課題 (< 特集> M2M サービスを支える情報通信技術)," 電子情報通信学会誌, vol. 96, pp. 305-312, 2013.

[11] I. Fette and A. Melnikov, "The websocket protocol," 2011.

[12] V. Pimentel and B. G. Nickerson, "Communicating and displaying real-time data with WebSocket," Internet Computing, IEEE, vol. 16, pp. 45-53, 2012.

[13] K. Shuang, X. Shan, Z. Sheng, and C. Zhu, "An efficient ZigBee-WebSocket based M2M environmental monitoring system," Dalian, 2014, pp. 322-326.

[14] D. Niyato, L. Xiao, and P. Wang, "Machine-to-machine communications for home energy management system in smart grid," Communications Magazine, IEEE, vol. 49, pp. 53-59, 2011.

[15] G. Wu, S. Talwar, K. Johnsson, N. Himayat, and K. D. Johnson, "M2M: From mobile to embedded internet," Communications Magazine, IEEE, vol. 49, pp. 36-43, 2011.

[16] L. Atzori, A. Iera, and G. Morabito, "The internet of things: A survey," Computer networks, vol. 54, pp. 2787-2805, 2010.

[17] K. Zheng, F. Hu, W. Wang, W. Xiang, and M. Dohler, "Radio resource allocation in LTE-advanced cellular networks with M2M communications," Communications Magazine, IEEE, vol. 50, pp. 184-192, 2012.

[18] J. Hämäläinen, "A measurement-based analysis of machine-to-machine communications over a cellular network," 2012.

[19] F. Ghavimi and H.-H. Chen, "M2M Communications in 3GPP LTE/LTE-A Networks: Architectures, Service Requirements, Challenges, and Applications," Communications Surveys & Tutorials, IEEE, vol. 17, pp. 525-549, 2015.

[20] (2016). MVNO の正体は「ちょっと特殊な携帯電話会社」. Available: http://www.itmedia.co.jp/mobile/articles/1509/18/news160.html

[21] (2016). 月額 280 円からの M2M 向け MVNO サービス提供開始. Available: https://www.akiba-holdings.co.jp/topic/topics/view/339

[22] L. M. Ericsson, "More than 50 billion connected devices," Ericsson White Paper, 2011.

[23] (2016). IoT／M2M の通信インフラは格安 SIM が解決する？. Available: http://www.atmarkit.co.jp/ait/articles/1511/30/news025.html

[24] (2016). 直近 3 日間の速度制限，MVNO の場合は？. Available: http://www.itmedia.co.jp/mobile/articles/1501/15/news142.html

[25] (2016). Autobahn|Python 0.12.1 documentation. Available: http://autobahn.ws/python/ (Open Source)

[26] J. Postel, "Transmission control protocol," 1981.

[27] M. Scharf and S. Kiesel, "NXG03-5: Head-of-line Blocking in TCP and SCTP: Analysis and Measurements," 2006, pp. 1-5.

[28] J. J. Garrett, "Ajax: A new approach to web applications," 2005.

[29] A. Russell, "Comet: Low latency data for browsers," alex. dojotoolkit. org, 2006.

[30] I. Hickson, "Server-sent events," W3C Working Draft WD-eventsource-20091222, latest version available at< http://www. w3. org/TR/eventsource, 2009.

[31] E. Estep, "Mobile HTML5: Efficiency and Performance of WebSockets and Server-Sent Events," 2013.

[32] D. Liu and R. Deters, "The reverse C10K problem for server-side mashups," 2008, pp. 166-177.

[33] NTT 技術予測研究会, 2030 年の情報通信技術：生活者の未来像: エヌティティ出版, 2015.

[34] Z. Shelby, K. Hartke, and C. Bormann, "The constrained application protocol (CoAP)," 2014.

[35] D. Locke, "Mq telemetry transport (mqtt) v3. 1 protocol specification," IBM developerWorks Technical Library, available at http://www.ibm. com/developperworks/webservices/library/ws-mqtt/index. html, 2010.

[36] S. Vinoski, "Advanced message queuing protocol," IEEE Internet Computing, pp. 87-89, 2006.

[37] U. Hunkeler, H. L. Truong, and A. Stanford-Clark, "MQTT-S—A publish/subscribe protocol for Wireless Sensor Networks," pp. 791-798.
[38] IBM. (2013). MQTT，IBM MessageSight 技術セミナー. Available: http://www.ibm.com/developerworks/jp/websphere/library/connectivity/ms_mqtt_ws/
[39] IBM. (2013, 2016-01-01). IBM MP0D: WebSphere MQ Telemetry V7.5 - Performance Evaluations - Japan. Available: http://www-01.ibm.com/support/docview.wss?uid=swg24034416
[40] F. Gont and A. Yourtchenko, "On the Implementation of the TCP Urgent Mechanism," RFC 6093, January2011.
[41] G. Carlucci, L. De Cicco, and S. Mascolo, "HTTP over UDP: an Experimental Investigation of QUIC," pp. 609-614.
[42] M. Belshe and R. Peon, "SPDY protocol," 2012.

謝辞

本論文を執筆するにあたり，大変多くの方からご指導，ご協力をいただきました．ここに心より感謝の意を示します．

最初に，常に愛情を持ってご指導してくださった東京大学報理工学系研究科教授江崎浩博士，講師落合秀也博士に感謝いたします．他国での研究生活は大変な部分も沢山ありましたが，日本における親のような存在でいつも支えてくださったお二人の方には，いくら感謝の気持ちを伝えても足りないような気がします．心から感謝の気持ちを申し上げます．

博士でありながらも，学生の目線からご指導・ご指摘をしてくださった，浅井大史博士，塚田学博士に感謝いたします．江崎・落合研究室の先輩として沢山の助言をくださった，中村遼さん，池上洋行さん，肥村洋輔さん，小林諭さんに感謝いたします．このような優秀な方々に囲まれ，常に刺激を受けながら研究生活を送ることができました．特に，池上君は長い間，後輩としても先輩としても，常に勇気付けてくれる存在でした．この場を借り，感謝の気持ちをお伝えしたいです．

江崎・落合研究室のメンバーとして一緒に過ごした，沼田進君さん，Tran Quoc Hoan さん，東角比呂志さん，中神啓貴さん，陶冶さん，田中晋太朗さん，平田歩さん，内海究さん，樋口海里さん，伊藤一さん，小林敦さん，井原大将さん，田口奨也さん，楊璞安さん，菰原裕さん，江間実さん，石嶋紘大さん，肥嶋達也さん，北里知也さん，坂本裕紀さん，塩川誠基さん，戸間明日香さんに感謝いたします．同期としても，先輩として不足な自分にいつも親切に教えてくれる人がいることは，とても心強かったです．ありがとうございました．

研究室生活を支援頂いた江崎研究室秘書の高橋富美さん，岩井愛映子さんに感謝致します．

最後に，常に息子のことを，信じて，心配して，支えてくださった父や母，そして愛する妹と弟，どんな悩みも打ち明けられる友人，お世話になった皆様に深く感謝いたします．

付録

ソースコード 1: 従来手法のサーバ側 **"default_server.py"**

```
 1: import random, threading, time
 2: from autobahn.twisted.websocket import WebSocketServerProtocol, \
 3:       WebSocketServerFactory
 4: from datetime import datetime
 5: from Queue import Queue
 6: from socket import *
 7:
 8: class MyServerProtocol(WebSocketServerProtocol):
 9:
10:       def onConnect(self, request):
11:             print("Client connecting: {0}".format(request.peer))
12:
13:       def onOpen(self):
14:             print("WebSocket connection open.")
15:
16:       ### added methods [Start] ###
17:
18:             # Length of PM and SM is 1 and 100
19:             # start of PM and SM is 17 string time data
20:             # remain string is "1" or "2"
21:             PMsg_ex = "1"*(1*1000-17)
22:             SMsg_ex = "2"*(100*1000-17)
23:
24:             # methods to create PM and SM
25:             def PM_create(sleep_time):
26:                   while 1:
27:                         if random.randint(1, 10) <= 1:
28:                               t_create_PM = datetime.now().strftime('%s.%f')
29:                               PMsg = t_create_PM + PMsg_ex
30:                               self.sendMessage(PMsg)
31:                               time.sleep(sleep_time)
32:                         else:
33:                               time.sleep(sleep_time)
34:             def SM_create(sleep_time):
35:                   while 1:
36:                         t_create_SM = datetime.now().strftime('%s.%f')
37:                         SMsg = t_create_SM + SMsg_ex
38:                         self.sendMessage(SMsg)
39:                         time.sleep(sleep_time)
40:
```

```
41:             # three methods are multi-threading
42:             th1 = threading.Thread(target=PM_create, args=([1.0]))
43:             th2 = threading.Thread(target=SM_create, args=([10.0]))
44:             th2.start()
45:             th1.start()
46:
47:             print '***** Message Sending Start *****'
48:
49:         ### added methods [end] ###
50:
51:         def onMessage(self, payload, isBinary):
52:             if isBinary:
53:                 print("Binary message received: {0} bytes".format(len(payload)))
54:             else:
55:                 print("Text message received: {0}".format(payload.decode('utf8')))
56:                 print("{0} bytes".format(len(payload)))
57:
58:             self.sendMessage(payload, isBinary)
59:
60:         def onClose(self, wasClean, code, reason):
61:             print("WebSocket connection closed: {0}".format(reason))
62:
63:
64: if __name__ == '__main__':
65:
66:         import sys
67:
68:         from twisted.python import log
69:         from twisted.internet import reactor
70:
71:         log.startLogging(sys.stdout)
72:
73:         factory = WebSocketServerFactory(u"ws://203.178.135.106:9000", debug=False)
74:         factory.protocol = MyServerProtocol
75:
76:         reactor.listenTCP(9000, factory)
77:         reactor.run()
```

ソースコード 2: 従来手法のクライアント側 **default_client.py**

```
 1: from autobahn.twisted.websocket import WebSocketClientProtocol, \
 2:         WebSocketClientFactory
 3: from datetime import datetime
 4: from socket import *
 5:
```

```
 6:  class MyClientProtocol(WebSocketClientProtocol):
 7:
 8:      def onConnect(self, response):
 9:          print("Server connected: {0}".format(response.peer))
10:
11:      def onOpen(self):
12:          print("WebSocket connection open.")
13:
14:      def onMessage(self, payload, isBinary):
15:          if isBinary:
16:              print("Binary message received: {0} bytes".format(len(payload)))
17:          else:
18:              # calculate the delay time (received time - created time)
19:              t_received = datetime.now().strftime('%s.%f')
20:              t_delay = float(t_received)-float(payload[0:17])
21:
22:              # save the delay time to file
23:              f = open("result.xls",'a')
24:              f.write("{0}\t{1:.6f}\n".format(len(payload),t_delay))
25:              f.close()
26:
27:      def onClose(self, wasClean, code, reason):
28:          print("WebSocket connection closed: {0}".format(reason))
29:
30:
31:  if __name__ == '__main__':
32:
33:      import sys
34:
35:      from twisted.python import log
36:      from twisted.internet import reactor
37:
38:      log.startLogging(sys.stdout)
39:
40:      factory = WebSocketClientFactory(u"ws://203.178.135.106:9000", debug=False)
41:      factory.protocol = MyClientProtocol
42:
43:      reactor.connectTCP("203.178.135.106", 9000, factory)
44:      reactor.run()
```

ソースコード 3: **PQ-PC のサーバ側 pq_server.py の一部**

```
1:       ### to this part, same with 'default_sever.py' ###
2:
3:          # for receiving UDP response from client after end of transmit
```

```
4:      svs = socket(AF_INET, SOCK_DGRAM)
5:      svs.bind(('203.178.135.106', 9000))
6:
7:      PMsg_ex = "1"*(1*1000-17)
8:      SMsg_ex = "2"*(100*1000-17)
9:
10:     # PM and SM putted in different Queue
11:     PQueue = Queue()
12:     SQueue = Queue()
13:
14:     def PM_create_PQ(sleep_time):
15:         while 1:
16:             if random.randint(1, 10) <= 1:
17:                 t_create_PM = datetime.now().strftime('%s.%f')
18:                 PMsg = t_create_PM + PMsg_ex
19:                 PQueue.put(PMsg)
20:                 time.sleep(sleep_time)
21:             else:
22:                 time.sleep(sleep_time)
23:     def SM_create_SQ(sleep_time):
24:         while 1:
25:             t_create_SM = datetime.now().strftime('%s.%f')
26:             SMsg = t_create_SM + SMsg_ex
27:             SQueue.put(SMsg)
28:             time.sleep(sleep_time)
29:
30:     # this methods flush message to websocket after end of transmit
31:     def PC_flush(sleep_time):
32:         while 1:
33:             if PQueue.empty() != True:
34:                 self.sendMessage(PQueue.get())
35:                 # send next message after UDP reaction from client
36:                 data = svs.recv(1)
37:                 time.sleep(sleep_time)
38:             elif SQueue.empty() != True:
39:                 self.sendMessage(SQueue.get())
40:                 # send next message after UDP reaction from client
41:                 data = svs.recv(1)
42:                 time.sleep(sleep_time)
43:             else:
44:                 time.sleep(sleep_time)
45:
46:     ### from this part, same with 'default_sever.py'###
```

ソースコード 4: PQ-PC のクライアント側 pq_client.py の一部

```
 1:  ### to this part, same with 'default_client.py'###
 2:
 3:    def onOpen(self):
 4:        print("WebSocket connection open.")
 5:
 6:    # socket for sending UDP Packet
 7:    cls = socket(AF_INET, SOCK_DGRAM)
 8:
 9:    def onMessage(self, payload, isBinary):
10:        # when message received, send UDP Packet to server
11:        self.cls.sendto(payload[0], ('203.178.135.106', 9000))
12:
13:  ### from this part, same with 'default_client.py'###
```

ソースコード 5: DMPQ-PC のサーバ側 dmpq_server.py の一部

```
 1:  ### to this part, same with 'pq_server.py'###
 2:
 3:        # Label A: Message is PM
 4:        def PM_create_label(sleep_time):
 5:            while 1:
 6:                if random.randint(1, 10) <= 1:
 7:                    t_create_PM = datetime.now().strftime('%s.%f')
 8:                    PMsg = t_create_PM + PMsg_ex
 9:                    PQueue.put('A'+PMsg)
10:                    time.sleep(sleep_time)
11:                else:
12:                    time.sleep(sleep_time)
13:
14:        # Label B, C, D
15:        def SM_create_label(sleep_time):
16:            while 1:
17:                t_create_SM = datetime.now().strftime('%s.%f')
18:                SMsg = t_create_SM + SMsg_ex
19:
20:                i = 0
21:                while i < len(SMsg):
22:                    # Label B: SM of len(SMsg) =< DIVISOR
23:                    if i == 0 and i + self.DIVISOR >= len(SMsg):
24:                        SQueue.put('B'+SMsg[i:i+self.DIVISOR])
25:                        i += self.DIVISOR
26:                    # Label C:
27:                    # SM of len(SMsg) > DIVISOR
28:                    # this part is not end of SM
```

```
29:                             elif i + self.DIVISOR >= len(SMsg):
30:                                 SQueue.put('D'+SMsg[i:len(SMsg)])
31:                                 i += self.DIVISOR
32:                             # Label D:
33:                             # SM of len(SMsg) > DIVISOR
34:                             # this part is end of SM
35:                             else:
36:                                 SQueue.put('C'+SMsg[i:i+self.DIVISOR])
37:                                 i += self.DIVISOR
38:                         time.sleep(sleep_time)
39:
40:     ### from this part, same with 'pq_server.py'###
```

ソースコード 6: DMPQ-PC のクライアント側 dmpq_client.py の一部

```
 1:     ### to this part, same with 'pq_client.py'###
 2:
 3:         def onOpen(self):
 4:             print("WebSocket connection open.")
 5:
 6:         cls = socket(AF_INET, SOCK_DGRAM)
 7:
 8:         # recieved_msg is temporary string for store divided messages
 9:         recieved_msg = ""
10:         recieved_header = 0
11:
12:         def onMessage(self, payload, isBinary):
13:             self.cls.sendto(payload[0], ('203.178.135.106', 9000))
14:
15:             if isBinary:
16:                 print("Binary message received: {0} bytes".format(len(payload)))
17:             else:
18:                 # there are 4 labels: A,B,C,D
19:                 # Label A: delete a labe and time calculate
20:                 if payload[0] == 'A':
21:                     t_received = datetime.now().strftime('%s.%f')
22:                     t_delay = float(t_received)-float(payload[1:18])
23:
24:                     f = open("result.xls",'a')
25:                     f.write("{0}\t{1}\n".format(payload[0],t_delay))
26:                     f.close()
27:
28:                 # Label B: delete a label and time calculate
29:                 elif payload[0] == 'B':
30:                     t_received = datetime.now().strftime('%s.%f')
```

```
31:                     t_delay = float(t_received)-float(payload[1:18])
32:
33:                     f = open("result.xls",'a')
34:                     f.write("{0}\t{1}\n".format(payload[0],t_delay))
35:                     f.close()
36:
37:                 # Label C: delete a label and stored in recieved_msg
38:                 elif payload[0] == 'C':
39:                     self.recieved_msg = self.recieved_msg + payload[1:]
40:                     self.recieved_header += 1
41:
42:                 # Label D:
43:                 # 1. delete a label and stored in recieved_msg
44:                 # 2. time calculate
45:                 # 3. recieved_msg is a merged message (original message)
46:                 elif payload[0] == 'D':
47:                     self.recieved_msg = self.recieved_msg + payload[1:]
48:                     self.recieved_header += 1
49:
50:                     t_received = datetime.now().strftime('%s.%f')
51:                     t_delay = float(t_received)-float(self.recieved_msg[0:17])
52:
53:                     f = open("result.xls",'a')
54:                     f.write("{0}\t{1}\n".format('C', t_delay))
55:                     f.close()
56:
57:                     # initialize a recieved_msg
58:                     self.recieved_msg = ""
59:                     self.recieved_header = 0
60:
61:                 else:
62:                     t_received = datetime.now().strftime('%s.%f')
63:                     t_delay = float(t_received)-float(payload[0:17])
64:
65:                     f = open("result.xls",'a')
66:                     f.write("{0}\t{1}\t{2:.6f}\n".format('E',len(payload),t_delay))
67:                     f.close()
68:         def onOpen(self):
69:
70:     ### from this part, same with 'pq_client.py' ###
```