多数のクライアント間で画面共有を行う場合における WebSocket と WebRTC との比較

小林 幸司†1　小荒田 裕理†2　山之上 卓†3　下園 幸一†3
小田 謙太郎†3

多数のクライアント間で画面共有を行なう場合において，通信手段に WebSocket を使った場合と WebRTC を使った場合の性能を比較したことについて述べる．サーバ―クライアント間で双方向通信を実現する WebSocket を使った場合と，ブラウザ上で peer-to-peer で双方向通信を実現する WebRTC を使った場合の２つについて，それぞれ実時間画面共有システムを開発した．この２つのシステムの比較実験を行い，WebSocket と，WebRTC の RTCDatachannel の通信性能を比較・評価を行うと同時に，それぞれの技術の問題点やこれからの可能性を考えた．

Comparison WebSocket and WebRTC in screen share system and data sending to many Web clients

KOUJI KOBAYASHI[†1]　YUURI KOARATA[†2]
TAKASHI YAMANOUE [†3]　KOUICHI SHIMOZONO [†3]
KENTAROU ODA[†3]

This paper describes a comparison between WebSocket and WebRTC in screen share systems, which send the motion picture on a PC's desktop and sends it to many Web clients. We have made two real time screen share systems. One of which uses WebSocket, which is a bidirectional communication protocol between a server and a Web client. And another of which uses WebRTC, which is a peer-to-peer bidirectional communication protocol between two Web clients. We have made a comparison experiment these real time screen share systems to evaluate the performance and examined technical problems and possibility of future.

1. はじめに

近年，ウェブブラウザ上でのサーバ―クライアント間での双方向通信を可能にするWebSocket[1]や，ウェブブラウザ上でpeer-to-peer間の双方向通信を可能にするWebRTC[2]をはじめとした，新しい通信技術が登場し，それらを用いたアプリケーションの開発も盛んに行われている。

我々の研究室でも，WebSocketを用いた端末上の画面を実時間で共有するシステム(以下，実時間共有システムと呼称)である「Web Screen Share[3] [4][5]」を開発し，研究会や我々のゼミで運用・開発を行っている．このシステムは，WebSocketに対応したブラウザであれば，アプリ等をインストールすることなく，端末上の画面を実時間で共有することができるシステムであり，多数のクライアントに画面を配信し，共有することで，プロジェクタとスクリーンを用いることなく，プレゼンテ―ションを行うことができるシステムである．

本論文ではWebSocket技術を使った場合と，WebRTC技術を使った場合のそれぞれについて実時間画面共有システムを試作し，これらの通信速度等の性能を調査・比較を行った結果について報告する．

2. 実験目的

我々の研究室で運用している実時間画面共有システム「Web Screen Share」は，図1で示すように送信者が画面を送信する際に， WebSocketを用いてデータの送信を行っている．この手法では，データの送受信はWebSocketサーバを中継して行っている．

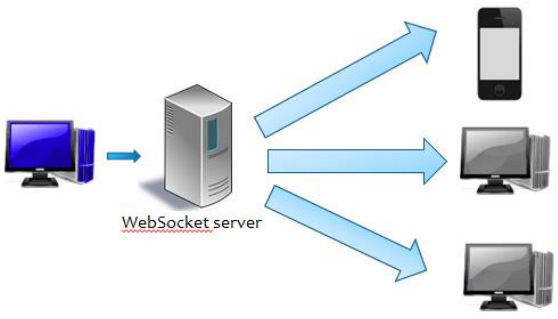

図 1　WebSocketによる画面共有におけるデータの流れ

しかし，WebSocketを用いて画面共有を行う場合は，データを一度WebSocketサーバに送信する必要があり，その部分の遅延が増加する．

そこで，我々はWebRTC技術に着目した．WebRTCはブラ

†1 鹿児島大学
Kagoshima University
†2 鹿児島大学大学院
Kyoto University Graduate School
†3 鹿児島大学学術情報基盤センター
Kagoshima University Computing & Communications Center

ウザ間でのpeer-to-peerの双方向通信を行うAPI群であるが，その中にpeer-to-peerでデータのやりとりを実現するRTCDatachannel[6]というものがある．この技術を用いて，多人数による実時間画面共有システムを実現させることができれば，図 2で示すようにサーバを中継することなくデータを送信することで，遅延の低下や，外部からの通信の傍受の可能性を低下させるといった効果を見込めるのではないかと考えた．しかし，一方で，WebRTCでの通信にはUDPプロトコルを用いるため，情報の信頼性という点で，TCPプロトコルを用いるWebSocketに劣るのではないか，とも考えた．

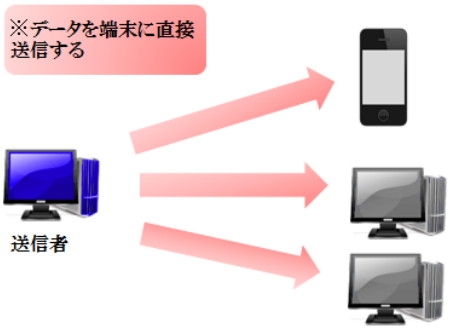

図 2　WebRTCによる画面共有におけるデータの流れ

この仮説を検証するため，WebSocket と WebRTC のそれぞれの技術を用いて新たに２つの実時間画面共有システム（パソコン間デスクトップ画面転送システム）を試作した．それらを使って実際に 1 台のクライアント上の動画を多数のクライアントに配信させ，通信性能を調査・比較することで，WebSocket と WebRTC の評価を行うと同時に，実時間画面共有システムの改良・性能向上の可能性を探った．

3. 関連研究

鈴木ら[7]はWebSocketの性能に関する研究を行っているが WebRTC との比較は行なっていない．WebSocket と WebRTC の RTCDatachannel によるデータ送信の比較に関する研究として，Gunay Mert Karadogan[8]の研究がある．この研究では IoT(Internet of Things)における WebSocket と WebRTC のデータ通信の詳細な調査を行っているが，本実験においては，上記の研究よりも端末や回線帯域の性能が高く，送信するデータ量も大きい場合について調査を行い，考察する．

4. 実験方法

4.1 実時間画面共有システムについて

WebSocket と WebRTC のそれぞれを使った，2 つの実時間画面共有システムを製作した． なお，プログラミング言語は JavaScript を用いた．

今回製作した実時間画面共有システムの大まかな仕組みを以下に述べる．

① Chrome desktop Capture API[9]を用いて，デスクトップ画面を動画として取得する．
② 取得した動画からスクリーンショットをキャプチャする．取得した画像は jpg に変換する．
③ 取得した画像を接続しているクライアントに送信する．
④ 画像を送信するごとに，画像送信時の時刻の取得，送信枚数のカウントを行い，その情報も送信する．
⑤ クライアントは，受信した画像データをブラウザ上で表示する．また，受信時の時刻の取得と，受信した画像の枚数をカウントする．
⑥ 10 秒ごとに，④で取得した時刻情報，送信枚数の情報と⑤で取得した時刻情報，受信枚数の情報から，フレームレートと遅延，送信したが受信できなかった画像の枚数を計算し，画面に表示する．その様子を図 3 に示す．

図 3　10秒毎のフレームレートと遅延の数値(拡大表示)

WebSocket による実装では，送信側 Web クライアントが同端末内の WebSocket サーバ(node.js)に画像を送信し，このWebSocketサーバが各WebクライアントにWebSocket を使って画像を送信する． WebSocket ではトランスポート層の通信に TCP プロトコルが利用される． WebRTC による実装では，最初に WebSocket サーバ(node.js)を用いてシグナリングを行い、互いの情報を交換し、その後、WebRTC の RTC Datachannel を使って送信側 Web クライアントが受信側 Web クライアントに直接画像を送信する． WebRTC ではトランスポート層の通信に UDP プロトコルが利用される．

この 2 つの実時間画面共有システムを利用し， 1 つの端末のデスクトップ画面の動画を同性能の他の複数の端末に送信し，その性能の評価を行った．

4.2 実験時の端末等の環境

送受信に用いた端末の環境，通信環境について述べる．

- CPU： Intel core I3-2700M 2.4GHz.
- OS： Windows7 64bit
- メモリ：4GB
- 通信回線の帯域幅：1Gbps(L2SW：D-linkDGS-3120-48tc)
- 使用ブラウザ：Google Chrome 37
- 使用 http サーバ： Apache 2.4.10
- 使用サーバサイド JavaScript：node.js 0.10.26
- その他使用ライブラリ：socket.io.js 0.9.16
- 接続台数：1 台，2 台，4 台，6 台，8 台，10 台

また，実験開始前に，サーバクライアント間の 1 対 1 における通信時の 1 分間の帯域幅の実測値を iperf[10]を用いて計測した．その結果を以下の表 1 と図 4 にて示す．なお，iperf で UDP の帯域幅を計測する際は，事前に iperf で使用する UDP の帯域幅の設定を行う必要があるが，今回はその設定値を 1Mbps から 100Mbps まで設定を変えて計測を行った．また，データグラムのサイズを 1470 バイト（初期設定），1024 バイト，512 バイトと設定を変えて計測を行った．その結果が図 4 である．

表 1 TCP,UDP での通信時の帯域幅

	帯域幅(Mbps)
TCP	255

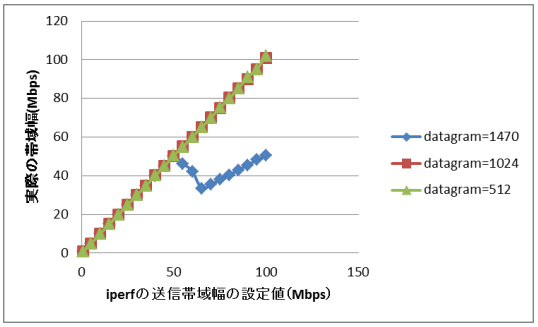

図 4 UDP の実測帯域幅について

TCP 通信時の帯域幅は 255Mbps であった．UDP 通信時の帯域幅は，データグラムが初期設定の 1470 バイトの際は，50Mbps までは設定値通りの帯域幅になるが，それ以上の帯域幅を設定しても，実測値は設定値を下回った．設定値を 70Mbps 以上にすると，設定値の上昇とともに実測値も少しずつ増加するが，設定値通りの帯域幅にはならなかった．一方、データグラムのサイズを 1024 バイト，512 バイトにして送受信を行った際は，設定値通りの速度が出た．

今回の環境における通信回線の帯域幅は 1Gbps であるが，実測値は TCP, UDP 共に公称値を下回り，特に UDP は TCP の約 5 分の 1 程度の帯域幅しか得られなかった．

4.3 送信する画面について

実際に共有を行う際に送信する画面についての設定を以下に示す．

- キャプチャ画面サイズ：1600*900
- 送信画像サイズ：320*180,640*360,854*480,1280*720
- UDP 時のデータ分割サイズ：64 キロバイト
- 画像形式：.jpg
- 送信フレームレート：10fps
- 元の動画の再生時間：30 秒(これを繰り返し再生)
- 元の動画のフレームレート：29.97fps
- フレーム幅：1280
- フレーム高：720
- データ速度：3597kbps
- 総ビットレート：3775kbps
- 再生方法：windows media player 12 で全画面再生
- 連続再生時間：2 分間

以上の条件のもと，実時間画面共有システムを稼動し，データを取得した．実際に稼働している様子を図 5 に示す．

図 5 実際の実時間画面共有の様子

4.4 データの収集，処理方法について

2 分間システムを稼働させ，4.1 にあるように 10 秒ごとにフレームレートと遅延の値を算出する．この場合，2 分間でフレームレートのデータが 12 個，遅延のデータが 13 個取得できる．取得できるデータ数が異なるのは，フレームレートのデータは送信開始直後に取得できないためである．

このデータを接続したクライアントの台数分収集し，そこからフレームレート，遅延の平均値と標準偏差を求めた．

たとえば，システムに 10 台接続している場合は，フレームレートの総データ数は 12*10=120 個，遅延の総データ数は 13*10=130 個である．

5. 実験結果と考察

5.1 実験結果

実時間画面共有システムの性能比較実験の結果を以下に示す．ここで菱形と四角形はそれぞれ WebSocket の場合と WebRTC の場合の平均値を表し，縦方向の細い線で示した範囲は標準偏差を表す．

(1) 320*180 のデータを送信した場合

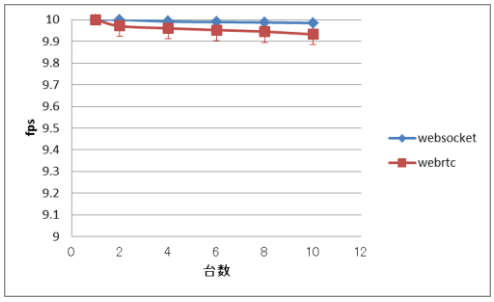

図 6　接続台数ごとのフレームレート(320*180)

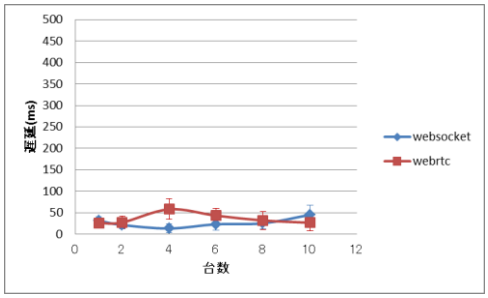

図 7　接続台数ごとの遅延 (320*180)

図 6，図 7 で示すように，画像サイズが 320*180 程度の小さいサイズである場合は，WebSocket，WebRTC の両方の実装でフレームレートは安定した．遅延も 100 ミリ秒未満に収まっており，グラフ図ではわずかに WebSocket の方が良い結果に見えるが，両者の差はほとんどないと考えられる．

(2) 640*360 のデータを送信した場合

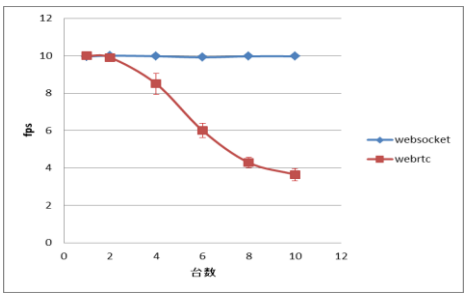

図 8　接続台数ごとのフレームレート(640*360)

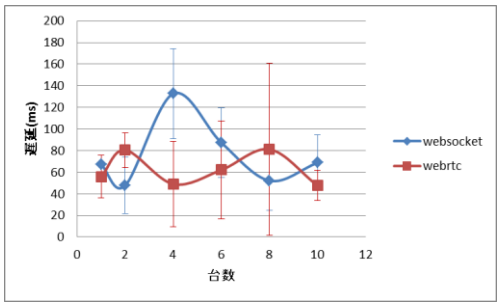

図 9　接続台数ごとの遅延 (640*360)

送信データサイズを 640*360 の大きさにして送信を行った場合，接続台数が 4 台になってから，WebRTC 側の実装は明確にフレームレートが低下した．接続台数が 10 台のとき平均フレームレートは 4.0 を下回り，目視でも明確にカクつきがわかるようになった．

また，遅延に関しては，送信データサイズを 320*180 にした場合に比べるとバラつきが大きくなったが，最大でも 200ms 程となっており，また，全てのデータにおいて遅延の急激な増加は見られなかった．

(3) 854*480 のデータを送信した場合

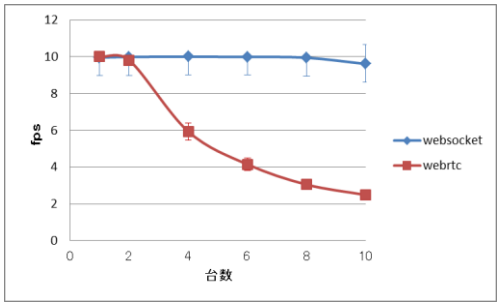

図 10　接続台数ごとのフレームレート (854*480)

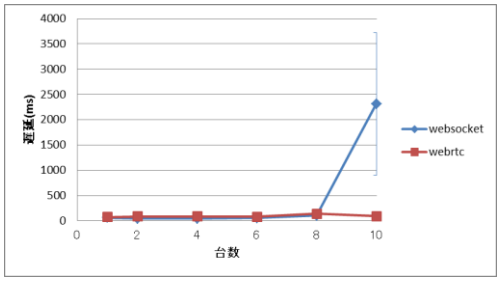

図 11　接続台数ごとの遅延 (854*480)

送信データサイズを 854*480 の大きさにして送信を行った場合，WebRTC 側の実装におけるフレームレートの低下がより顕著になった．遅延も 640*360 時と比較すると，平均して 25ms ほど増加している．

一方，WebSocket 側の実装では，接続台数を 10 台にしたところ，フレームレートの低下とバラつきが見られるようになった．また，それに伴って遅延も WebRTC 側に比べると急激に増加し，場面によっては 3 秒以上の遅延が出た．

(4) **1280*720 のデータを送信した場合**

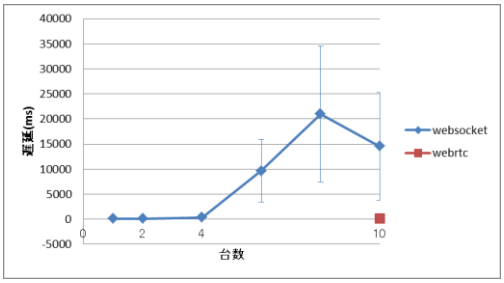

図 13　接続台数ごとのフレームレート (1280*720)

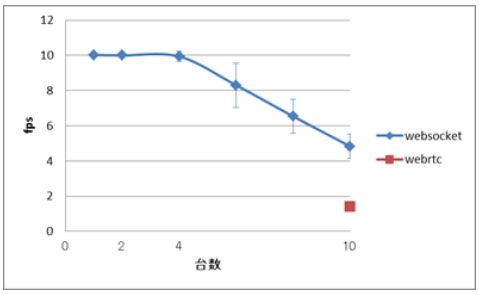

図 13　接続台数ごとの遅延 (1280*720)

送信データサイズを 1280*720 の大きさにして送信を行った場合，WebSocket 側の実装では，接続台数の増加によるフレームレートの大幅な低下と，遅延の急激な増加が見られた．接続台数を 10 台にした場合の遅延の値は，接続台数を 8 台にした場合より少なくなっているが，これは，データ量の増大により端末のバッファサイズを超過した結果，接続がリセットされ，画像の破棄が行われたためであり，実際の遅延の大きさは接続台数が 10 台のときのほうが大きい．

一方，WebRTC 側の方は，接続台数が 1 台，2 台，4 台の場合は，画面共有の途中で画像送信エラーが発生し，以後一切の画像が送信できなくなってしまった．接続台数が 10 台のときは 2 分間画像送信が続いたが，フレームレートは平均して約 1.39 程度だった．

なお，画像送信エラーに関して，接続台数が 4 台の場合は，遅延は発生したものの，20 秒程は画面共有が出来ていた．一方，接続台数が 1 台，2 台の場合は，画面共有を開始してからすぐに送信エラーが発生し，画像が送信できなくなった．

5.2 実験結果からの考察

性能比較実験の結果から，画面共有システムの通信手段として，WebSocket を用いて実装した場合と WebRTC を用いて実装した場合を比較したとき，WebSocket で実装した場合の方が WebRTC で実装した場合より高いフレームレートの数値を出した．しかし，送信する画像サイズを大きくすると，フレームレートの低下と共に遅延も増大した．

一方，WebRTC での実装では，送信する画像サイズが大きい場合のフレームレートの低下幅は WebSocket での実装に比べると大きいが，遅延に関しては一番大きい場合でも数百ミリ秒であり，WebSocket での実装に比べるとある程度は遅延が抑えられていた．

このような結果になった原因として，プロトコル及びアプリケーション側で設定された帯域幅が関係していると考えられる．

今回の実験に用いた映像データを全画面で 1 分間再生を行った際の，1 秒ごとに取得した jpeg データの合計サイズとその推移についてした次頁の図 14 のグラフにまとめた．この図を見ると，24 秒前後，53 秒前後のシーンがもっともデータサイズが大きく，1280*720 のデータの場合だと，4 メガバイト程の大きさになる．これを考慮した上で，10 台の端末に今回用いた映像を 1280*720 のサイズで不足なく送信することを考えると，必要な帯域幅は 4.0*8*10=320Mbps となる．　しかし，今回の WebSocket での実装で用いた TCP プロトコルの帯域幅は，表 1 に示すように 255Mbps しかないため，帯域幅が不足し，結果としてフレームレートの低下や遅延が発生したと考えられる．

同様に，WebRTC での通信を考える．今回のフレームレートおよび遅延の計測は 10 秒毎に計測を行ったが，その際に一番動画のビットレートが高く，送信量が多くなった箇所は再生元の 30 秒の動画の最後の 10 秒のところであった．この映像を送信している際の各端末のフレームレート値と図 14 の値から，送信者の総送信データ量をまとめたものを表 2，表 3 に示す．

表 2　WebRTC 画面共有のピーク時の総送信帯域幅(Mbps)

	1280*720	854*480	640*360	320*180
1 台	(エラー)	12.749	8.106	2.735
2 台	(エラー)	24.668	16.050	5.470
4 台	(エラー)	27.792	26.366	10.892
6 台	(エラー)	30.246	27.095	16.287
8 台	(エラー)	30.756	27.095	21.661
10 台	30.57065	29.740	27.764	27.179

表 3　WebRTC 画面共有のピーク時の 1 台あたりの送信帯域幅(Mbps)

	1280*720	854*480	640*360	320*180
1 台	(エラー)	12.749	8.106	2.735
2 台	(エラー)	12.334	8.025	2.735
4 台	(エラー)	6.948	6.591	2.723
6 台	(エラー)	5.041	4.516	2.714
8 台	(エラー)	3.845	3.387	2.708
10 台	3.057	2.974	2.776	2.718

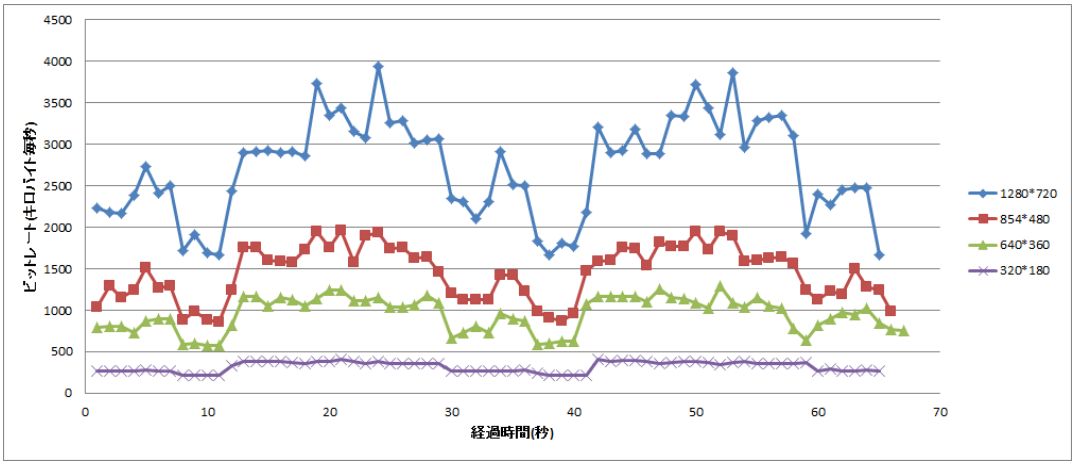

図 14　送信したデータサイズの推移

表 2，表 3 から，データの送信量が増加し，接続台数が増加するにつれて 1 台あたりに使う帯域幅が少なくなっていることがわかる．また，これはフレームレートの低下とも関連しており，画像サイズ 640*360 以上，接続台数 4 台以上の場合はフレームレートの大幅な低下や送信エラーが生じている．

このことから，WebRTC の実装において，接続しているセッション全体で使用できる帯域幅が予め制限されているのではないかと推測することが出来る．

また, WebRTC の実装において送信エラーが多発した原因も，使用できる帯域幅に対して送信量が大きかったためと考えられる．

今回の環境で 1280*720 の画面サイズで画面共有を行う場合，フレーム欠けすることなく受信側が画像を表示するために必要な帯域は，図 14 のピーク時のビットレートから，およそ 32Mbps 必要であり，表 2 にあるどの帯域幅よりも大きい値であり, 帯域の制限を超過している可能性が高い．

また，エラーの原因としては，Windows マシン側の問題[11]の可能性も考えられる．これは，UDP で 1024 バイト以上のパケットを送信した場合にデータロックやバッファ処理が発生するという問題である. 図 4 で示した UDP の帯域幅の計測において, データグラムのサイズが 1470 バイトの際に，内部でフラグメント処理が行われていないにも関わらず回線の速度が低下した. また, 図には示していないが，データグラムのサイズを 1025 バイト以上にした場合も速度の低下が見られた．

この問題が OS に起因するものであるかを調査するため, Windows マシンと MacOS X マシンをそれぞれ用意し, iperf による UDP 帯域幅計測を行った．その結果を下図 15 に示す．なお，データグラムのサイズは 1470 バイトとする．

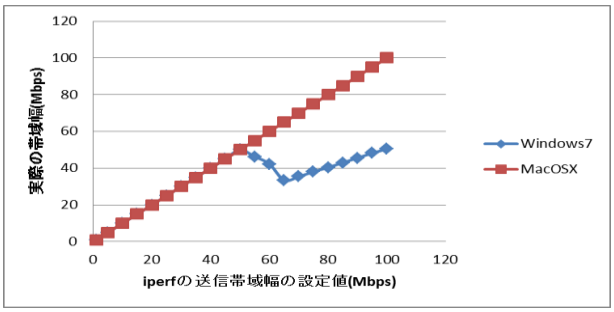

図 15　UDP の実測帯域幅(OS ごとの比較)

図 15 からわかるように, MacOS X のマシンにおいては，iperf での設定値通りの帯域幅になった．以上から，UDP における通信速度の低下は Windows マシン固有の問題であると考えられる．

このことから，WebRTC でデータを送信する際に，Windows マシン側でデータロックが発生し，それがボトルネックとなった可能性が高いと考えられる．1280*720 のサイズの画像データを送信した際に送信エラーが多発したのも，一度に大量のデータを送信し，データロックが多発したためと推測できる．

この問題を回避するためには，Windows マシンのレジストリを書き換える方法がある．レジストリを書き換えることなく問題を回避するためには WebRTC でデータを送信する際に, 送信するデータを 1024 バイトずつに分割する必要があると考えられる．

以上から，実験の結果 WebRTC を使って動画転送を行なった場合の性能が WebSocket を使った場合よりも低かった理由は，WebRTC で使用できる帯域幅が WebSocket に比べて小さく，また Windows マシンのデータロックが発生したことが原因だと考えられる．

逆に言えば，接続したマシン間との帯域幅を確保し，データロックを回避できれば，パフォーマンスを向上させることができる可能性は十分にあると考えられる．

また，WebSocket を利用する場合， サーバは Web クラ

イアントが接続されているネットワークとは異なる外部のネットワークに存在することが多い．この場合，同じネットワーク間のみで通信を行なう場合と比べて通信遅延が大きくなる場合がある．WebSocket は通信遅延に反比例して帯域が狭くなる性質がある TCP を利用しているため，サーバが外部のネットワークにある場合は性能が低下することがある．これに対して WebRTC は UDP を利用するため，遅延による性能低下は緩和される可能性もある．なお，WebSocket を使った場合の性能低下を緩和するため，我々は文献[12] [13]のような研究も行なっている．

6. 今後の計画

今回は JPEG 形式の静止画像を短時間に繰り返して送信側端末から受信側端末に送信する画面共有システムにおける WebSocket と WebRTC の比較を行ったが，帯域幅の問題や，データサイズの増大，Windows マシンの問題が見つかった．

今後は双方の問題点を解決するため，WebRTC による実装における帯域幅の確保や，動画のストリーミングによる実時間画面共有システムの性能調査を視野に入れ，改良を続ける予定である．

WebRTC においては，Session Description Protocol(SDP)をデータのやり取りを開始する前に送信するが，この SDP の中身には，送信するデータの内容や帯域幅などの設定が入っている．これを上手く書き換えることで，接続数あたりの帯域幅を増やすことができる可能性がある．

また，WebRTC を利用すると動画のストリーミングによる実時間画面送受信を容易に実現できる．しかし，WebSocket 側における動画ストリーミング送受信の実装が間に合わなかったため今回の比較実験には用いなかった．

動画形式のデータを利用すれば，従来の静止画像データを繰り返し送信する方法よりも大幅に送信データサイズを小さくすることができると考えらえる．ただし，エンコード処理に時間がかかることによる遅延の発生や，使用する端末の性能への依存度も高くなる可能性がある．

そこで，今後は，動画ストリーミングによる実時間画面共有システムに関しては，

① WebRTC の既存のストリーミング(Stream API と PeerConnection API)を用いたもの．
② 動画のエンコードを他のアプリケーションやプログラムに任せて，データ送信だけ WebSocket，あるいは WebRTC の RTCDataChannel を用いるもの．
③ 今回の実験に用いたものを改良したもの．

以上 3 つのシステムを開発し，その性能を比較して，実時間画面共有システムに最適な構造を追及していく予定である．また，動画以外のデータ，たとえば音楽のストリーミング配信や，ファイルの共有といったものに関しても同様に調査を行っていく予定である．

7. まとめ

従来 WebSocket を通信手段に用いていた画面共有システムについて，性能の向上を期待して，WebRTC の RTCDatachannel を通信手段に用いたものを試作したが，帯域幅の制限，送信するデータサイズの大きさ，Windows マシンの設定等が障害となり，性能が向上しないことがわかった．逆に言えば，その問題を解決すれば，従来の実時間画面共有システムよりも性能向上が図れる可能性があると考えられる．

謝辞 実時間画面共有システムの開発，及びその評価実験にご協力頂いた皆様に，謹んで感謝の意を表する．

参考文献

1) The WebSocket Protocol I. Fette, A. Melnikov (TXT = 162067) (Status: PROPOSED STANDARD) (Stream: IETF, Area: app, WG: hybi) (2011).
http://tools.ietf.org/html/rfc6455
2) WebRTC Google Chrome team(2011).
https://sites.google.com/site/webrtc/
3) 山之上 卓,樋高 想士,小林 幸司,小荒田 裕理,片桐 太樹,小田 謙太郎,下園 幸一 : ポータブルクラウドの試作，情報処理学会研究報告,Vol. 2013-IOT-22, No. 12, pp. 1-5(2013)
4) 山之上 卓,小荒田 裕理,Katagiri Taiki ,小田 謙太郎,下園 幸一 : HTML5 技術を利用した授業や会議向けデスクトップ画面実時間配信システムとその管理システムの試作，情報処理学会研究報告,Vol. 2014-IOT-26, No. 11, pp. 1-8(2014)
5) 樋高 想士,山之上 卓,小田 謙太郎 他 ：携帯端末利用者のための会議・授業の支援システム ：ポータブルクラウド (インターネットアーキテクチャ)電子情報通信学会技術研究報告,Vol.113, No. 443, pp.13-17(2014).
6) WebRTC 1.0: Real-time Communication Between Browsers:W3C Working Draft 10 ,5.2-5.3(2013).
http://www.w3.org/TR/webrtc/#rtcdatachannel
7) 鈴木 新一,水越 一貴,深澤 昌志 他 : 学校間ネットワーク上に構築した遠隔教育支援システムの接続手法の提案とその評価，情報処理学会論文誌,Vol. 54, No. 3, pp. 1050-1060 (2013)
8) GUNAY MERT KARADOGAN: Evaluating WebSocket and WebRTC in the Context of a Mobile Internet of Things Gateway(2013).
9) chrome.desktopCapture (2014).
https://developer.chrome.com/extensions/desktopCapture
10) Iperf - The TCP/UDP Bandwidth Measurement Tool(2011)
https://iperf.fr/
11) Microsoft サポート : [netshow]サーバーが利用可能な帯域幅を利用しない
http://support.microsoft.com/kb/235257
12) Yamanoue, T., Koarata, Y., Oda, K., Shimozono, K. 2014. A Technique to Assign an Appropriate Server to a Client, for a CDN Consists of Servers at the Global Internet and Hierarchical Private Networks, In *proceedings of the 2014 IEEE 38th Annual Computer Software and Applications Conference Workshops (COMPSACW),* (Västerås, Sweden, July 21-25) , IEEE, 90-95
13) 杉田 裕次郎,白澤 竜馬,亀澤 健太 他 : P2P を利用した画面配信システムの性能改善に関する研究,情報処理学会研究報告, Vol.2011-IOT-12 , No. 14, pp. 1-5(2011)