Multicast Wifi VNCの実装と評価

安田 亮[1,a)]　河野 真治[2,b)]

概要：講義やゼミではPC画面で用意した資料を見ながら進行することが多い。PCごとにアダプターや解像度が異なっており、正常にPC画面を表示できない場合がある。当研究室で開発しているTreeVNCは、発表者のPC画面を参加者のPCに表示する画面配信システムである。TreeVNCの画像共有は、送信するデータ量が多いために有線LANでの接続に限られている。本稿では無線LANでもTreeVNCを利用可能にするため、Wifi上にシステム制御用の従来の木構造と、画像データ送信用のMulticastの両方を構築を行う。Multicastでは、サーバから送信された画像データUpdateRectangleを小さいパケットに分割し送信を行うよう実装した。

1. 画面配信ソフトウェアTreeVNCの活用

最近のコロナ禍などの社会情勢によりリモートワークの重要性が高まっている。リモートワークではビデオ通話を行いながら自宅で仕事をすることになるが、PCの画面共有を利用して情報を共有することも多い。

ビデオ通話ソフトウェアの1つにZoomがある。Zoomはカメラを利用したビデオ通話に重点を置いて開発されているため、送信される画像に対してある程度のロスを許容して圧縮が行われている。そのためPC画面を共有した際に書類の文字がはっきり見えないということが発生する。またビデオ通話には、Zoomが管理するデータセンターにあるサーバを経由して通信を行っている。

当研究室で開発している画面配信システムTreeVNC[1]は、発表者の画面を参加者のPC画面に表示するソフトウェアである。画面共有に特化しており、共有するPC画面をロスなく圧縮しデータを送信することが可能である。TreeVNCは木構造型のオーバーレイネットワークを動的にLANまたはWAN上に構成し、画面共有に参加しているPC同士でのP2P型の通信を行う。これにより、データセンター上の強力なサーバやネットワークを要求せずに配信を可能にしている。

しかし、オーバーレイネットワークは無線LAN接続では共有された通信帯域を消費してしまう。有線の場合はネットワークスイッチの容量が十分に大きければ問題にならないが、LAN/WAN/Wifiで Multicast通信がサポートされていれば、それを使うことが望ましい。本研究では、TreeVNC の Multicast通信の実装を行い実際の動作確認を行う。

[1] 琉球大学大学院理工学研究科情報工学専攻
[2] 琉球大学工学部工学科知能情報コース
[a)] riono210@cr.ie.u-ryukyu.ac.jp
[b)] kono@ie.u-ryukyu.ac.jp

2. TreeVNCの基本概念

Virtual Network Computing[2](以下 VNC) は、サーバ側とクライアント (ビューワー) 側からなるリモートデスクトップソフトウェアである。遠隔操作にはサーバを起動し、クライアント側がサーバに接続することで可能としている。また、動作にはRFBプロトコルを用いている。

Remote Frame Buffer プロトコル [3](以下 RFB) とはVNC上で使用される、自身のPC画面をネットワーク上に送信し、他人のPC画面に表示を行うプロトコルである。画面が表示されるユーザ側をRFBクライアントと呼び、画面送信を行うためにFrameBufferの更新が行われる側をRFBサーバと呼ぶ。

Framebufferとは、メモリ上に置かれた画像データのことである。RFBプロトコルでは、最初にプロトコルのバージョンの確認や認証が行われる。その後、RFBクライアントへ向けてFramebufffer の大きさやデスクトップに付けられた名前などが含まれている初期メッセージを送信する。

RFBサーバ側はFramebufferの更新が行われるたびに、RFBクライアントに対してFramebufferの変更部分を送信する。さらに、RFBクライアントからFramebuffer - UpdateRequestが来るとそれに答え返信する。変更部分のみを送信する理由は、更新があるたびに全画面を送信すると、送信するデータ面と更新にかかる時間面において効率が悪くなるからである。

TreeVNCはjavaを用いて作成されたTight VNC[4]を

元に作成されている。TreeVNC は VNC を利用して画面配信を行なっているが、従来の VNC では配信 (サーバ) 側の PC に全ての参加者 (クライアント) が接続するため負荷が大きくなってしまう (図 1)。

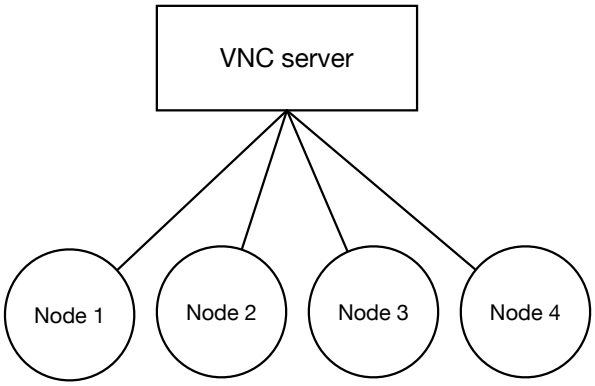

図 1: 従来の VNC での接続構造

そこで TreeVNC ではサーバに接続を行なってきたクライアントをバイナリツリー状 (木構造) に接続する。接続してきたクライアントをノードとし、その下に新たなノードを最大 2 つ接続していく。これにより人数分のデータのコピーと送信の手間を分散することができる (図 2)。

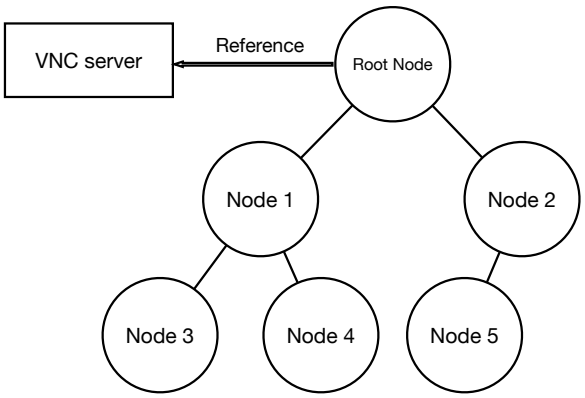

図 2: TreeVNC での接続構造

通信の数は、送信されるデータは従来の方法だと N 個のノードに対して N-1 回必要である。これはバイナリツリー状の構造を持っている TreeVNC でも通信の数は変わらない。

バイナリツリー状に接続することで、N 台のクライアントが接続を行なってきた場合、従来の VNC ではサーバ側が N 回のコピーを行なって画面配信する必要があるが、TreeVNC では各ノードが最大 2 回ずつコピーするだけで画面配信が可能となる。

木構造のルートのノードを Root Node と呼び、そこに接続されるノードを Node と呼ぶ。Root Node は子 Node にデータを渡す機能、各 Node の管理、VNC サーバから送られてきたデータの管理を行なっている。各 Node は、親 Node から送られてきたデータを自身の子 Node に渡す機能、子 Node から送られてきたデータを親 Node に渡す機能がある。

3. MulticastQueue

配信側の画面が更新されると、VNC サーバから画像データが FRAME_BUFFER_UPDATE メッセージとして送られる。その際、親 Node が受け取った画像データを同時に複数の子 Node に伝えるために MulticastQueue というキューに画像データを格納する。

各 Node は MulticastQueue からデータを取得するスレッドを持つ。MulticastQueue は複数のスレッドから使用される。

4. 木の再構成

TreeVNC はバイナリツリー状での接続のため、Node が切断されたことを検知できずにいると構成した木構造が崩れてしまい、新しい Node を適切な場所に接続できなくなってしまう。そこで木構造を崩さないよう、Node 同士の接続の再構成を行う必要がある。

TreeVNC の木構造のネットワークトポロジーは Root Node が持っている nodeList で管理している。Node の接続が切れた場合、Root Node に切断を知らせなければならない。

TreeVNC は LOST_CHILD というメッセージ通信で、Node 切断の検知および木構造の再構成を行なっている。LOST_CHILD の検出方法には MulticastQueue を使用しており、ある一定時間 MulticastQueue から画像データが取得されない場合、MemoryOverFlow を回避するために Timeout スレッドが用意されている。そして、Timeout を検知した際に Node との接続が切れたと判断する。

5. データの圧縮形式

TreeVNC では、ZRLEE[5] というエンコード方法でデータの圧縮を行う。ZRLEE は RFB プロトコルで使用できる ZRLE というエンコードタイプを元に生成される。

ZLRE(Zlib Run-Length Encoding) とは可逆圧縮可能な Zlib 形式 [6] と Run-Length Encoding 方式を組み合わせたエンコードタイプである。

ZLRE は Zlib で圧縮されたデータとそのデータのバイト数がヘッダーとして付与され送信される。Zlib は java.util.zip.deflater と java.util.zip.inflater で圧縮と解凍が行える。しかし java.util.zip.deflater は解凍に必要な辞書を書き出す (flush) ことができない。従って、圧縮されたデータを途中から受け取ってもデータを正しく解凍することができない。

そこで ZRLEE は一度 Root Node で受け取った ZRLE のデータを unzip し、データを update rectangle と呼ばれる画面ごとのデータに辞書を付与して zip し直すことで、始めからデータを読み込んでいなくても解凍をできるようになっている (図 3)。

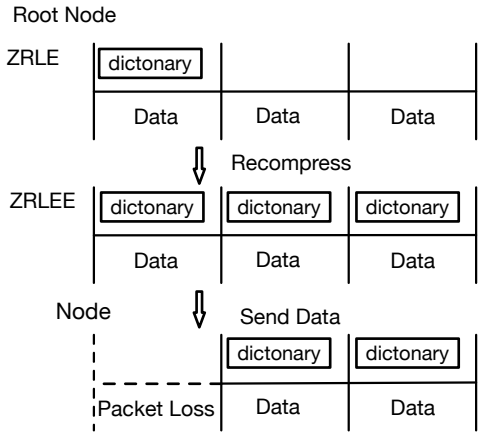

図 3: ZRLEE へ再圧縮されたデータを途中から受け取った場合

一度 ZRLEE に変換してしまえば、子 Node はそのデータをそのまま流すだけでよい。ただし、deflater と inflater では前回までの通信で得た辞書をクリアしないといけないため、Root Node 側と Node 側では毎回新しく作る必要がある。辞書をクリアすることで短時間で解凍され画面描画されるという、適応圧縮を実現していることになり圧縮率は向上する。

6. ShareMyScreen

従来の VNC では、配信者が交代するたびに VNC の再起動、サーバ・クライアント間の再接続を行う必要がある。TreeVNC では配信者の切り替えのたびに生じる問題を解決している。

TreeVNC を立ち上げることでケーブルを使用せずに、各参加者の手元の PC に発表者の画面を共有することができる。画面の切り替えについてはユーザが VNC サーバへの再接続を行うことなく、ビューワー側の Shere My Screen ボタンを押すことで配信者の切り替えが可能となっている。

TreeVNC の ROot Node は配信者の VNC サーバと通信を行なっている。VNC サーバから画面データを受信し、そのデータを子 Node へと送信している。配信者切り替え時に Share Screen を実行すると、Root Node に対し SERVER_CHANGE_REQUEST というメッセージが送信される。このメッセージには Share Screen ボタンを押した Node の番号やディスプレイ情報が付与されている。メッセージを受け取った Root Node は配信を希望している Node の VNC サーバと通信を始める。

7. Multicast の利用

現在の TreeVNC では有線接続と無線 LAN 接続のどちらでも、VNC サーバから画面配信の提供を受けることが可能である。しかし無線 LAN の帯域は接続 PC 全部で共有されるためにオーバーレイネットワークを用いる場合は少数の台数の場合でしか実用的には動作しない。Wifi の Multicast の機能を用いればこの欠点を克服することができると考えられる。Root Node は無線 LAN に対して変更する Update Rectangle を Multicast で一度だけ送信すればよい。Tree の構築には従来通りのオーバーレイネットワークを用いる。Root Node は複数のネットワーク毎に木構造と Multicast を管理することになる。(図 4)。この Multicast はネットワークがサポートしていれば LAN/WAN/Wifi のそれぞれで有効であると思われる。

制御構造を木構造オーバーレイネットワークによって行うのは従来のコードを再利用できるのが主な理由で Root に直接接続する方法でも良い。

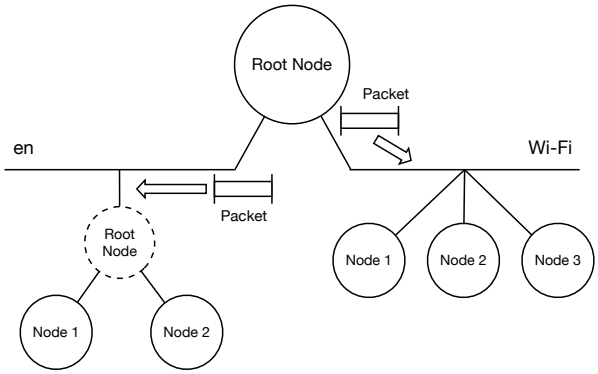

図 4: 接続方法の分割

Wifi の Multicast Packet のサイズは 64KB が最大となっている。4K ディスプレイと例にとると、画面更新には 8MB の画素数 * 8B の色情報となり、圧縮前で 64MB 程度となる。

8. RFB の UpdateRectangle の構成

RFB の Update Rectangle によって送られてくる Packet は表 1 のような構成となっている。

1 つの Update Rectangle には複数の Rectangle が入っており、さらに 1 つ 1 つの Rectangle には x,y 座標や縦横幅、encoding type が含まれている Rectangle Header を持っている。ここでは ZRLE で圧縮された Rectangle が 1 つ、VNC サーバから送られてくる。Rectangle には、Zlib 圧縮されたデータが detalengths と呼ばれる指定された長さだけ付いてくる。このデータは、画面を 64x64 の Tile で構成できるように分割されている (図 5 中 Tile)。

表 1: UpdateRectangle による Packet の構成

1 byte	messageID
1 byte	padding
2 byte	n of rectangles
2 byte	U16 - x-position
2 byte	U16 - y-position
2 byte	U16 - width
2 byte	U16 - height
4 byte	S32 - encoding-type
4 byte	U32 datalengths
1 byte	subencoding of tile
n byte	Run Length Encoded Tile

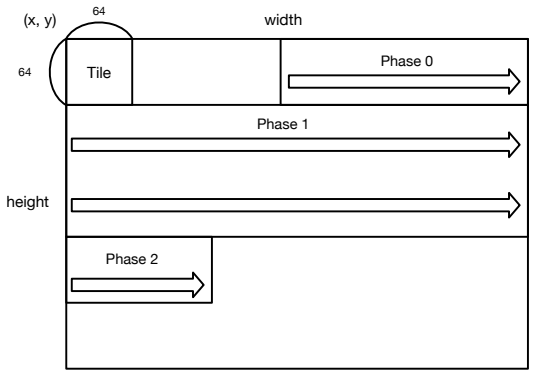

図 5: Rectangle の分割

Tile 内はパレットなどがある場合があるが、通常は Run Length encode された RGB データである。これまでの TreeVNC では VNC サーバから受け取った Rectangle を分割せずに ZRLEE へ再構成を行なっていた。これを Multicast のためにデータを 64KB に収まる最大 3 つの Rectangle に再構成する。(図 5)。この時に Tile 内部は変更する必要はないが、Rectangle の構成は変わる。ZRLE を展開しつつ、Packet を構成する必要がある。

64KB の Packet の中には複数の Tile が存在するが、連続して Rectangle を構成する必要がある。3 つの Rectangle の構成を下記に示す。

- 行の途中から始まり、行の最後までを構成する Rectangle(図 5 中 Phase0)
- 行の初めから最後までを構成する Rectangle(図 5 中 Phase1)
- 行の初めから、行の途中までを構成する Rectangle(図 5 中 Phase2)

9. TileLoop の圧縮とブロッキング

TileLoop は VNC サーバから受け取った ZRLE を図 5 のように Rectangle を分割し、ZRLEE に再構成を行った Packet を生成する。通常では 1 画面全部を一つの Update Rectangle で送ることがあり、数 Mbyte のパケットが生成されしまう。これを再圧縮を行いながら 64kbyte 以内の Update Rectangle に分割する。個々のパケットは長方形の集合なので、分割されたパケットは 1-3 個の長方形を含むことになる。

以下の図 6 に TileLoop で生成される Packet 全体と、分割される各 Phase の Rectangle を示した。

Packet Header には表 1 に示した messageID、padding、n of rectangle が核にのうされている。また、分割された Rectangle にはそれぞれ表 2 に示した Rectangle Header を持っている。

次に TileLoop の処理について説明する。Code 1 は TileLoop の処理を大まかに抜粋したものである。

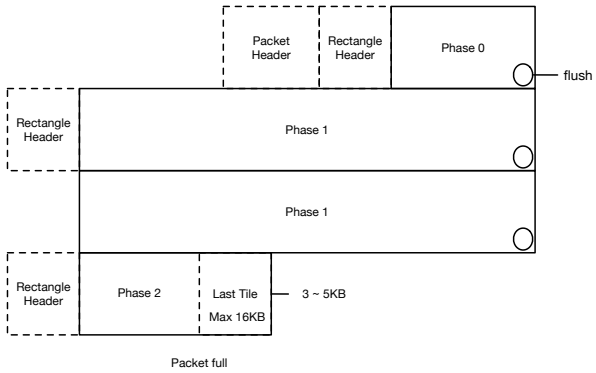

図 6: ZRLEE の Packet の構成と分割された Rectangle

表 2: Rectangle Header の構成

2 byte	U16 - x-position
2 byte	U16 - y-position
2 byte	U16 - width
2 byte	U16 - height
4 byte	S32 - encoding-type
4 byte	U32 datalengths
1 byte	subencoding of tile
n byte	Run Length Encoded Tile

Code 1: TileLoop の処理関数

```
1  public void decode(Renderer renderer, ByteBuffer
       header, FramebufferUpdateRectangle rect,
       ByteBuffer buf, int zippedLength, TreeRFBProto
       rfbProto) throws TransportException {
2    int offset = zippedLength;
3    int maxX = rect.x + rect.width;
4    int maxY = rect.y + rect.height;
5    byte[] bytes = buf.array();
6    WifiMulticast = rfbProto;
7
8    TileLoop tileloop = new TileLoop(rfbProto,
         zippedLength);
9    if (WifiMulticast) {
10     tileloop.zrleeBlocking(rfbProto, header, rect,
           bytes);
11   }
12   for (int tileY = rect.y; tileY < maxY; tileY +=
         MAX_TILE_SIZE) {
13     int tileHeight = Math.min(maxY - tileY,
           MAX_TILE_SIZE);
14     if (tileloop.blocking) {
15       tileloop.c1rect.height += tileHeight;
16     }
17     for (int tileX = rect.x; tileX < maxX; tileX +=
           MAX_TILE_SIZE) {
18       int tileWidth = Math.min(maxX - tileX,
             MAX_TILE_SIZE);
19       offset += decodePacked(bytes, offset, renderer,
             paletteSize, tileX, tileY, tileWidth,
             tileHeight);
20       if (WifiMulticast) {
21         tileloop.multicastPut(rfbProto, false, bytes,
               offset, tileWidth, tileHeight, tileX,
               tileY);
22       }
23     }
24   }
25 }
```

Code 1: 8 - 11 行目は TileLoop の初期化で Blocking と構築する Packet の準備を行っている。Code 1: 12 行目からの Loop では ZRLE で受け取った Rectangle を 1Tile 64x64 に分割し、1Tile ずつ処理を行う。

TileLoop には c1Rect と呼ばれる Rectangle を持ってい

る。これは読み込んだ Tile 分だけ縦横を拡張していくことによって Rectangle の再構成を行なっている (Code 1: 15 行目)。

Code 1: 21 行目では処理対象の Tile のデータ圧縮とフェーズ確認を行う関数である。Code 2 はその関数の処理を大まかに抜粋したものである。

Code 2: TileLoop の処理関数

```
1  public void multicastPut(TreeRFBProto rfb, boolean
       last, byte[] bytes, int offset, int tileW, int
       tileH, int tileX, int tileY) throws
       TransportException {
2    int span = offset - prevoffset;
3    if (span==0) return;
4    deflater.setInput(bytes, prevoffset, span);
5    width += tileW;
6    c1rect.width += tileW;
7    do {
8      deflater.deflate(c1, Deflater.SYNC_FLUSH);
9      if (!deflater.needsInput()) {
10       flushDeflator(true,"␣full␣");
11       int bytesRead = (int)deflater.getBytesRead();
12       prevoffset = flushOffset+bytesRead;
13       if (c0rect!=null) { // finish pahse 1
14         flushRectangle(c0rect,prevC1LineOffset,"␣full␣
               c0rect");
15         c0rect = null;
16        }
17       flushRectangle(c1rect,c1.position(),"␣full␣
             c1rect"); // phase 2
18       flushMuticast(rfb, bytes);
19     }
20   } while (! deflater.needsInput());
21   prevoffset = offset;
22   if (last) {
23     flushDeflator(false,"␣last");
24     if (c0rect!=null) {
25       flushRectangle(c0rect,prevC1LineOffset,"␣last␣
             c0rect");
26       makeHeaderSpace();
27       c0rect = null;
28     }
29     flushRectangle(c1rect,c1.position(),"␣last␣c1rect"
           );
30     flushMuticast(rfb, bytes);
31     return;
32   }
33   if (c1rect.x > rect.x) { // phase 0
34     assert(c0rect==null);
35     if (width >= rect.width) { // end of phase 0
36       boolean end = flushDeflator(false,"␣end␣of␣phase
             ␣0␣");
37       width = 0;
38       flushRectangle(c1rect,c1.position(),"␣end␣of␣
             phase␣0");
39       c1rect = new FramebufferUpdateRectangle(rect.x,
             c1rect.y+tileH,0,0);
40       if (end || c1rect.y >= rect.y+rect.height) {
41         flushMuticast(rfb,bytes);
42         return;
43       }
44       c1.position(c1.position()+ RECT_HEADER_SIZE); //
             header space
45       prevC1Offset = c1.position();
46     }
47   } else { // phase 1
48     if (width >= rect.width) { // next line
49       boolean end = flushDeflator(false, "␣phoase␣1␣
             next␣line␣");
50       width = 0;
51       prevC1LineOffset = c1.position();
52       if (c0rect!=null) { // extend phase 1
53         c0rect.height += tileH;
54       } else { // first phase 1 case
55         c0rect = c1rect;
56       }
57       c1rect = new FramebufferUpdateRectangle(rect.x,
             c0rect.y+c0rect.height, 0, 0);
58       if (end || c1rect.y >= rect.y+rect.height) {
59         c0rect = null; // next will be first phase 1
               case
60         flushMuticast(rfb,bytes);
61         return;
62       }
63     }
64   }
65 }
```

Code 2: 2 - 19 行目では、読み込んだ Tile のデータを圧縮用の Stream に格納し、java.util.zip.deflater を利用して圧縮を行う。

java.util.zip.deflater には下記の 3 種類の圧縮方法がある。

- NO_FLUSH : Stream に格納されたデータを最効率で圧縮を行う。Stream にある入力データが規定量に満たない場合は圧縮されない
- SYNC_FLUSH : これまでに Stream に圧縮されたデータの圧縮を行う。ただし圧縮率が低下する可能性がある
- FULL_FLUSH : SYNC_FLUSH 同様、これまでに Stream に格納されたデータ圧縮を行う。異なる点はこれまでの辞書情報がリセットされるため、圧縮率が極端に低くなる可能性がある

Packet のサイズは 62KB としているが、一旦の制限として 37KB までを格納可能としている。これには理由があり、ZRLE と java.util.zip.deflater を使用した圧縮では、圧縮後のデータ長を予測することができない。Packet が満杯になってしまうと、圧縮書き込みの途中であっても圧縮書き込みが中断する。そのため、Packet サイズを余分に確保する必要がある。したがって最初から最大の 62KB ではなく、37KB に制限を行なっている。TileLoop ではデータの圧縮に NO_FLUSH を利用していたが、圧縮後のデータが Packet の上限である 62KB を超えてしまうことが多発した。

これは圧縮されるための入力データの規定量が想定以上に多く、圧縮後のデータ長が Multicast Packet の上限を超えてしまったためである。そこで圧縮率は悪くなるが、

確実に Packet に書き込まれる SYNC_FLUSH を利用し、ZRLEE の生成を行う。

また圧縮は別スレッドで行われているため、圧縮が途中の段階で圧縮用 Stream に新しい入力があった場合、先に格納されている入力は消えてしまう。そのために、Code 2: 20 行目で needInput() が True になるまで待機する。

Code 2: 21 行目以降の処理は Tile のフェーズ確認である。

Code 2: 22 - 27 行目は行の最後の Tile だった場合の条件分岐である。図 6 中の flush に該当する Tile であり、そこで圧縮されたデータとループ内で再構成を行っている c1Rect の情報を Packet へ書き込みを行う。ここで FULL_FLUSH を行う理由は、次の行に移る際圧縮用の Stream にデータを残さないためである。また、この箇所で Phase2 かどうかの判断も行っている。

Code 2: 33、47 行目では Phase0、Phase1 の確認と処理を行っている。どちらの処理も圧縮されたデータと c1Rect の情報を Packet に書き込み、次の Phase のための初期化などを行っている。Packet へ書き込みが完了すると、子 Node への送信のために Packet が MulticastQueue へ格納される。

10. Multicast 用のシステム構成

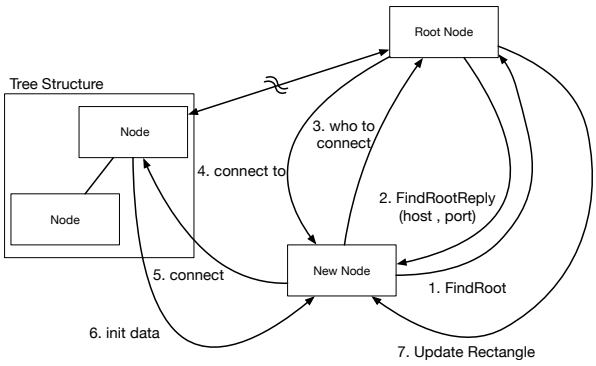

図 7: Multicast のネットワークシステム

画面配信を Multicast に対応できるように図 7 のようなシステムを構築した。新しく接続しに来たクライアントは Root Node に対して Find Root を送信する。Root Node が見つかれば、Root Node はこれに対して Find Root Reply を送り、host と port についての情報をクライアントに知らせる (図 7 中 1,2)。次のメッセージ通信で木構造のどの Node を親とすれば良いかが決まる (図 7 中 3,4)。木構造に接続する理由は、画面配信の初期メッセージを受け取るために行っている (図 7 中 5,6)。初期メッセージを受け取った後は、Root Node より Update Rectangle を受け取ることで Multicast における画面配信を可能としている (図 7 中 7)。

11. 実装の現状

理論上は Wifi 接続で画面配信が可能であったが、いくつかの問題が発生していることがわかった。まず IPv4 と IPv6 の両方で Multicast を行うことはできない。Java の library のレベルで失敗してしまう。IPv4 の Multicast ではクライアントが 1 台のみしか接続できない状況となった。IPv6 では Multicast 実装は必須なので状況が改善する可能性がある。また、これは Wifi Station の制限である可能性もある。接続している PC 上では有線と同等の性能を発揮することができ、Blocking による遅延も数秒程度であった。また、パケット落ちもほとんど見られなかった。

Multicast 実装は有線上でも有効であると考えられるが、コロナ禍下で大学での実験に制約があり、十分な測定ができていない。

12. まとめ

今回の実装では、Wifi にデータを乗せるためのパケット分割を行う Blocking と、Multicast を行うためのオーバーレイネットワークの実装を行なった。

実装により Multicast で実用的に画面配信が可能であると推測されるが、まだ、接続が限られるなどの問題があり解決されていない。

現在 TreeVNC は PC 画面のみ配信をすることが可能であるが、音声を送信することも可能である。その場合、画面よりも音声の方がデータ量少ないため、Multicast でも送信することは可能である。

参考文献

[1] Yu TANINARI and Nobuyasu OSHIRO and Shinji KONO: VNC を用いた授業用画面共有システムの実装と設計，日本ソフトウェア科学会第 28 回大会論文集 (2011).
[2] RICHARDSON, T., STAFFORD-FRASER, Q., WOOD, K. R., AND HOPPER,: A. Virtual Network Computing (1998).
[3] RICHARDSON, T., AND LEVINE, J.: The remote framebuffer protocol. RFC 6143 (2011).
[4] TightVNC Software: `http://www.tightvnc.com`.
[5] Yu TANINARI and Nobuyasu OSHIRO and Shinji KONO: VNC を用いた授業用画面共有システムの設計・開発，情報処理学会システムソフトウェアとオペレーティング・システム研究会 (OS) (2012).
[6] LOUP GAILLY, J., AND ADLER, M.: zlib: A massively spiffy yet delicately unobtrusive compression library., `http://zlib.net`.
[7] Surendar Chandra, Jacob T. Biehl, John Boreczky, Scott Carter, Lawrence A. Rowe: Understanding Screen Contents for Building a High Performance, Real Time Screen Sharing System, *ACM Multimedia* (2012).
[8] 立樹伊波，真治河野：有線 LAN 上の PC 画面配信システム TreeVNC の改良，第 57 回プログラミングシンポジウム予稿集，Vol. 2016, pp. 29–37 (2016).