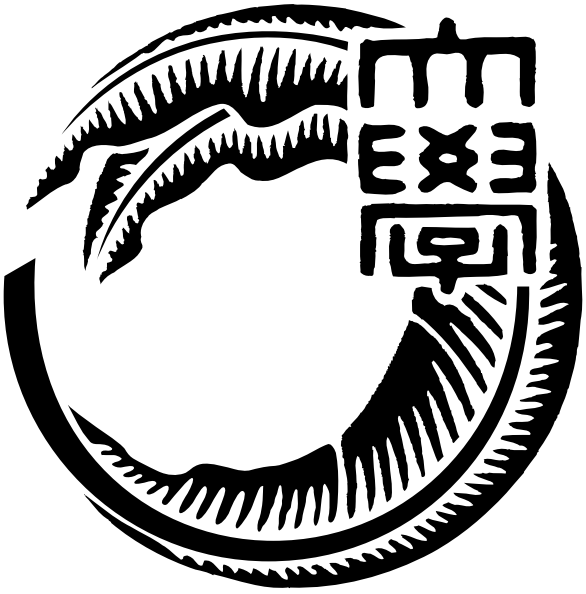
令和元年度 卒業論文

画面配信システム TreeVNC の マルチキャスト導入



琉球大学工学部情報工学科

165729B 安田亮

指導教員 河野 真治

目 次

図 目 次

表 目 次

第1章　はじめに

1.1　背景と目的

1.2　論文の構成

第2章　TreeVNCの基本概念

2.1　Virtual Network Computing

Virtual Network Computing(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。

2.2　Remote Frame Bufferプロトコル

Remote Frame Buffer(以下 RFB) プロトコルとは 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 Framebuffer 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。

2.3 TreeVNCの構造

TreeVNC は java を用いて作成された Tight VNC を元に作成されている。 TreeVNC は VNC を利用して画面配信を行っているが、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 2.1)。

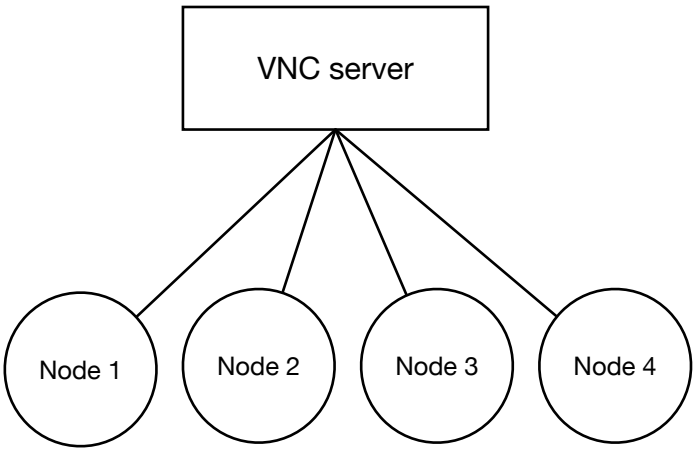


図 2.1: 従来の VNC での接続構造

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 2.2)。

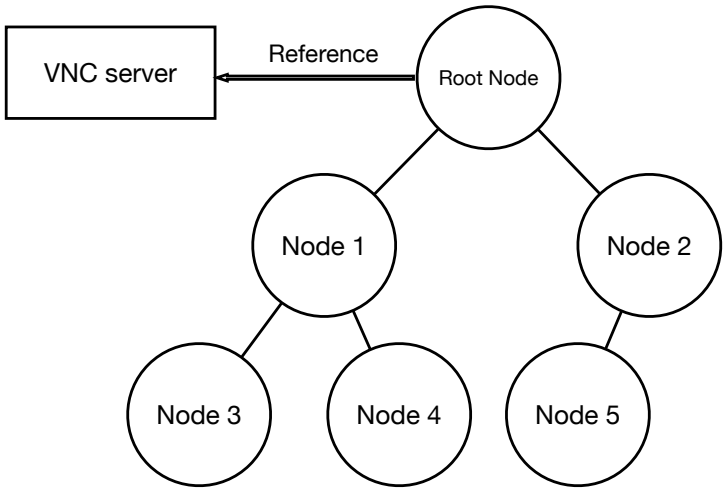


図 2.2: TreeVNC での接続構造

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。

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 回の通信が必要である。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。

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。

2.4 TreeVNC の通信プロトコル

TreeVNC の通信経路としては以下が挙げられる。

- 任意の Node から Root Node に直接通信を行う send direct message (Node to Root)
- Root Node から任意の Node に直接通信を行う send direct message (Root to Node)
- 任意の Node から木構造を上に辿って Root Node まで通信を行う message up tree (Node to Root)
- Root Node から木の末端の Node までの全ての Node に通信を行う message down tree (Root to Node)
- Root Node から配信者の VNC サーバへの通信を行う send message (Root to VNC Server)
- 配信者の VNC サーバから Root Node への通信を行う send message (VNC Server to Root)

Node 間で行われるメッセージ通信には RFB プロトコルで提供されているメッセージに加え、TreeVNC 独自のメッセージを使用している。TreeVNC で使用されるメッセージの一覧を表 2.1 に示す。

表 2.1: 通信経路とメッセージ一覧

通信経路	message	説明
send direct message (Node to Root)	FIND_ROOT	TreeVNC 接続時に Root Node を探す.
	WHERE_TO_CONNECT	接続先を Root Node に聞く.
	LOST_CHILD	子 Node の切断を Root Node に知らせる.
send direct message (Root to Node)	FIND_ROOT_REPLY	FIND_ROOT への返信.
	CONNECT_TO_AS_LEADER	左子 Node として接続する．接続先の Node が含まれている.
	CONNECT_TO	右子 Node として接続する．接続先の Node が含まれている.
message down tree (Root to Node)	FRAMEBUFFER_UPDATE	画像データ．EncodingType を持っている.
	CHECK_DELAY	通信の遅延を測定する.
message up tree (Node to Root)	CHECK_DELAY_REPLY	CHECK_DELAY への返信.
	SERVER_CHANGE_REQUEST	画面切り替え要求.
send message (Root to VNCServer)	FRAMEBUFFER_UPDATE_REPLY	画像データの要求.
	SET_PIXEL_FORMAT	pixel 値の設定.
	SET_ENCODINGS	pixel データの encodeType の設定.
	KEY_EVENT	キーボードからのイベント.
	POINTER_EVENT	ポインタからのイベント.
	CLIENT_CUT_TEXT	テキストのカットバッファを持った際の message.
send message (VNCServer to Root)	FRAMEBUFFER_UPDATE	画像データ．EncodingType を持っている.
	SET_COLOR_MAP_ENTRIES	指定されている pixel 値にマップする RGB 値.
	BELL	ビープ音を鳴らす.
	SERVER_CUT_TEXT	サーバがテキストのカットバッファを持った際の message.

2.5 MulticastQueue

配信側の画面が更新されると、VNC サーバから画面データが FRAME_BUFFER_UPDATE メッセージとして送らる。その際、画像データの更新を同時に複数の Node に伝えるために MulticastQueue というキューにデータを蓄積している。

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。MulticastQueue は java.util.concurrent.CountDownLatch を用いて実装されている。CountDownLatch とは java の並列実行用に用意された API で、他のスレッドで実行中の操作が完了するまで、複数のスレッドを待機させることができるクラスである。スレッドを解放するカウントを設定することができ、カウントが 0 になるまで await メソッドでスレッドをブロックすることができる。

TreeVNC では、親 Node が MulticastQueue を持っており、接続されている子 Node の数だけ画像データにカウントを設定する。子 Node が画像データを取得すると、そのカウントが減る。接続している全ての子 Node が画像データを取得するとカウントが 0 になり、MulticastQueue から画像データが削除される。

2.6 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 の木構造の接続形態は Root Node が持っている nodeList というリストで管理している。nodeList 内の treeNum という値が各 Node に割り当てられており、これによって木構造のどの位置に Node が接続されているかの判別が可能である。よって、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っている。

LOST_CHILD の検出方法には、MulticastQueue を用いている。親 Node が MulticastQueue を持っているが、接続している全ての子 Node が画像データを取得するまで MulticastQueue の中に入っている画像データを削除することができない。子 Node が MulticastQueue から画像データを取得せずに、画像データが溜まり続けると親 Node が MemoryOverFlow を起こしてしまう。この問題を回避するために Timeout スレッドが用意されている。Timeout を検知した際に Node との接続が切れたと判断する。LOST_CHILD の検知と木構造の再構成の手順を以下に示す。

- 子 Node の切断を検知した Node が Root Node へ LOST_CHILD メッセージを送信する (図 2.3 中、1: lostChild())
- LOST_CHILD メッセージを受け取った Root Node は nodeList の更新を行う (図 2.3 中、2: updateNodeList())
- 切断した Node を nodeList から削除し、nodeList の最後尾の Node に切断した Node と treeNum を割り当てる
- Root Node は最後尾の Node に、切断した子 Node が接続していた親 Node に接続するよう、COMMECT_TO メッセージを送信する (図 2.3 中、3: connectTo(1))
- 最後尾の Node が子 Node を失った親 Node へ接続を行う (図 2.3 中、4: connectToParent(1))

LOST_CHILD によって、切断された全ての Node を検知することができるため、nodeList の更新が正しく行われる。よって、新しく接続を行ってきた Node を適切な場所に接続することが可能となる。

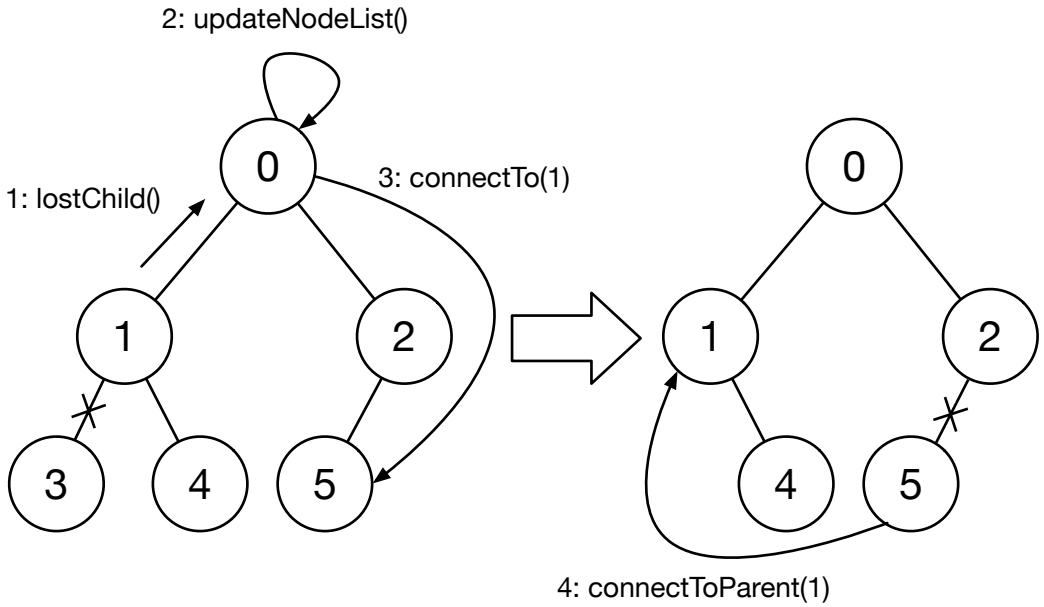


図 2.3: LOST_CHILD の検知・再接続

2.7 データの圧縮形式

TreeVNC は ZRLEE というエンコードタイプでデータのやり取りを行う。ZRLEE は RFB プロトコルで使用できる ZLRE 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 javaùtil.zip.deflater は解凍に必要な辞書を書きだす (flush) ことが出来ない。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出来ない (図 2.4)。

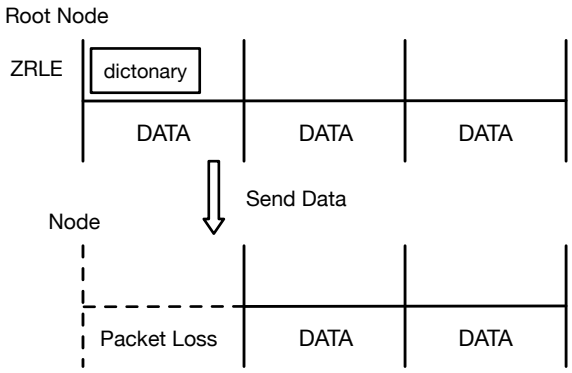


図 2.4: ZRLE でデータを途中から受け取った場合

そこで ZRLEE は一度 Root Node で受け取った ZRLE のデータを unzip し、後述する 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、初めからデータを読み込んでいなくても解凍を出来るようになっている (図 2.5)。

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により adaptive compression を実現していることになり圧縮率が向上する。

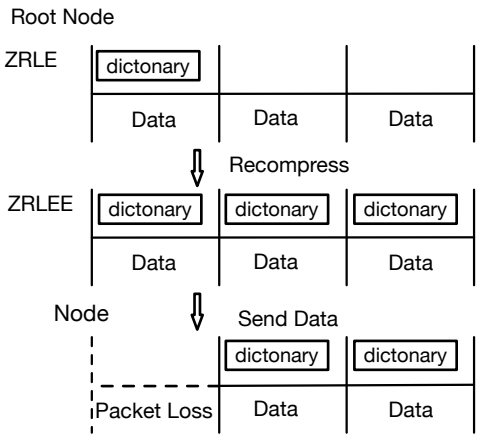


図 2.5: ZRLEE へ再圧縮されたデータを途中から受け取った場合

TreeVNCではRFBプロトコルによって配信側の画面の変更部分はFRAME_BUFFER_UPDATEメッセージとして送られてくる。メッセージの中には変更部分の原点のx,y座標と縦横の幅等が含まれており、長方形として展開される。この長方形をUpdate Rectangleと呼ぶ。

2.8 ShareScreen

ゼミでは発表者が順々に入れ替わる。発表者が入れ替わるたびに共有する画面の切り替えが必要となる。ゼミを円滑に進めるために、画面の切り替えをスムーズに行いたい。

画面の共有にプロジェクタを使用する場合、発表者が変わるたびにケーブルの抜き差しを行う必要がある。その際に、PCとプロジェクタを接続するための変換アダプタが必要になる場合や、接触不良が起こる等の煩わしい問題が生じることがある。

従来のVNCでは、配信者が切り替わるたびに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する必要なしに、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サーバへの際接続を行うことなく、ビューワー側のShare 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。そのためTreeVNCは配信者切り替えのたびにVNCを終了し再接続する必要がない。

2.9 複数のネットワーク接続時の木の構成

TreeVNC は Root Node が複数のネットワークに接続している場合、図 2.6 のようにネットワーク別にを形成する。TreeVNC は Root Node が TreeManager を持っている。TreeManager は TreeVNC の接続部分を管理しており、木構造を管理する nodeList の生成を行う。この nodeList を元に、新しい Node の接続や、切断検出時の接続の切り替え等を行なっている。

TreeManager は Root Node の保持しているネットワーク毎に生成される。新しい Node が接続してきた際、interfaces から Node のネットワークと一致する Tree Manager を取得し、Node 接続の処理を任せる。

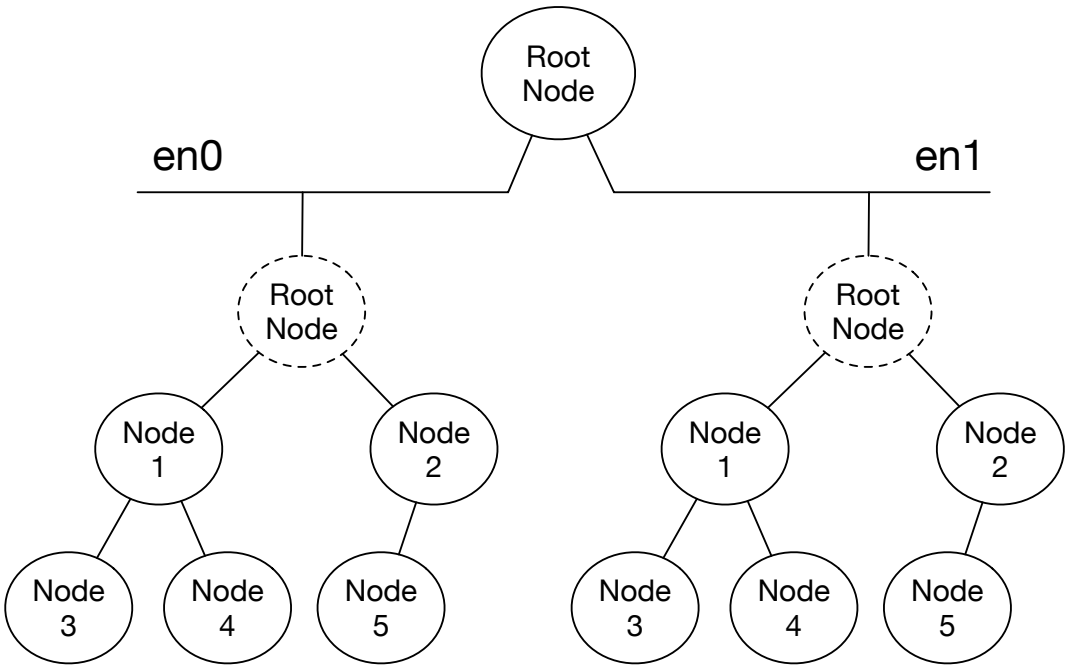


図 2.6: ZRLEE へ再圧縮されたデータを途中から受け取った場合

第3章　Multicastに向けたBlockingの実装

3.1　有線接続と無線LAN接続との違い

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。しかし画像配信のデータ量は膨大なため、現在のTreeVNCでVNCサーバに無線LAN接続を行なった場合、画面配信の遅延が大きくなってしまう。

無線LAN接続時の場合でも画面切り替えの機能は有効であるため、VNCサーバ側が無線LANで接続を行い、クライアント側は有線接続を行うことで画面配信が可能となる。ここで、Wifiの Multicastの機能を用いてクライアント側でもWifiを使用することが可能であると考えられる。Root Nodeは無線LANに対して、変更するUpdate Rectangleを Multicastで一度だけ送信する。

有線接続の場合は従来通り、VNCサーバ、Root Node、Nodeからなるバイナリツリー状に接続されるため、有線接続時と無線LAN接続時でのVNCサーバの接続方法を分割することが可能である (図3.1)。こうすることにより、新しいNodeが無線LAN接続であっても有線接続の木構造には影響を及ぼさない。

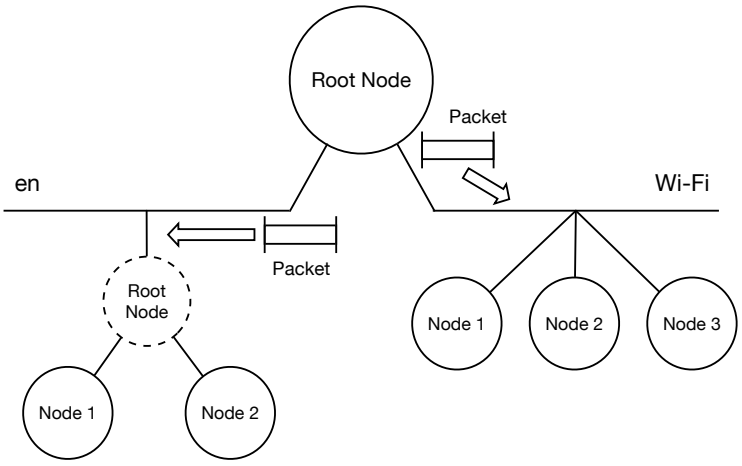


図 3.1: 接続方法の分割

Wifi の Multicast Packet のサイズは 64KByte が最大となっている。4K ディスプレイを例にとると、4K ディスプレイの大きさの画面更新には 8MByte(画素数) 8Byte(色情報)で圧縮前で、64MByte 程度となる。

3.2 Update Rectangle の構成

RFB の Update Rectangle は以下の表 3.1 の構成となっている。

表 3.1: UpdateRectangle 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

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、datelengths 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、さらに 64x64 の tile に分割されている (図 3.2 中 Tile)。

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 Mluticast のためにデータを 64KByte に収まる最大 3 つの Rectangle に再構成する (図 3.2)。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。ZLRE を展開しつつ、Packet を構成する必要がある。

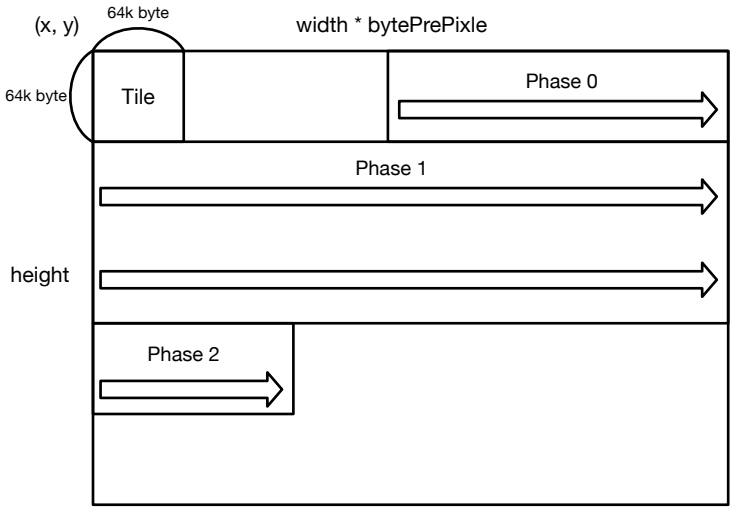


図 3.2: Rectangle の分割

Zlib は丁度良い所で圧縮を flush する必要がある。このためには、Zlib の API を用いて、適当なタイミングで flush を呼ぶ。この時に 1tile ずつ flush してしまうと圧縮率を下げる可能性がある。

64KByte 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 3.2 中 Phase0)
- 行の初めから最後までを構成する Rectangle(図 3.2 中 Phase1)
- 行の初めから、行の途中までを構成する Rectangle(図 3.2 中 Phase2)

3.3 TileLoop

Rectangle の再構成には TileLoop というループ内で行なっている。TileRoop は 64

3.4 Packet Lost

Wift の Multicast Packet は確実に送られることが保証されていない。データに通し番号をつけて、欠落を検出することはできるが、再送処理は複雑であることが予想される。そこで、一定時間ごとに全画面のデータを送信することによって Packet Lost しても画面共有に影響はないと考える。

第4章　TreeVNCのソースコードの修正改善

4.1　Gradle 6.1対応

TreeVNC はソースの build に Gradle を使用している。しかし使用している Gradle のバージョンは 4.8 であった。これを現行の最新バージョンである Gradle 6.1 で build を実行すると、build faild となってしまう。

build faild の原因となっていたのは、MacOS 用の application に書き出すためのプラグイン edu.sc.seis.macAppBundle のバージョンが古かったことが原因だった。これまで使用していた edu.sc.seis.macAppBundle のバージョンは 2.1.7 であったが、これを最新の 2.3.0 に変更することで、Gradle 6.1 でソースコードの build が可能となった。

4.2　java9以降のRetinaAPI対応

現在の Mac には Retina ディスプレイが搭載されている。Retina ディスプレイとはこれまでに使用されていた液晶ディスプレイよりも画素が細かく、画面がより鮮明に描画されるディスプレイである。画素数が異なるため、画面配信のためには接続されているディスプレイが、液晶ディスプレイか、Retina ディスプレイかの判別が必要となる。

java には Retina ディスプレイ判別のための API が提供されている。しかし java9 より大きな仕様変更があり、TreeVNC で使用している Retina の API は非推奨となってしまったため、書き換えを行なった。非推奨となったコードを以下のソースコード 4.1、書き換え後のコードを以下のソースコード 4.2、4.3 に示す。

ソースコード 4.1: java8 以前の Retina ディスプレイ API

```
1 public static int getRetinaScale(int shareScreenNumber) {
2     int scale = 1;
3     GraphicsEnvironment env = GraphicsEnvironment.getLocalGraphicsEnvironment();
4     final GraphicsDevice[] devices = env.getScreenDevices();
5
6     try {
7         Field field = devices[shareScreenNumber].getClass().getDeclaredField("scale")
              ;
8
9         if (field != null) {
10            field.setAccessible(true);
11            Object retinaScale = field.get(devices[shareScreenNumber]);
12
13            if (retinaScale instanceof Integer) {
14                scale = (Integer) retinaScale;
15                return scale;
16            }
17        }
18    } catch (Exception ignore) {}
19    return scale;
20 }
```

ソースコード 4.2: Retina ディスプレイの判別関数

```
1  public boolean getIsRetinaDisplay(int shareScreenNumber) {
2         GraphicsEnvironment env = GraphicsEnvironment.getLocalGraphicsEnvironment();
3         final GraphicsDevice[] devices = env.getScreenDevices();
4         GraphicsConfiguration conf = devices[shareScreenNumber].
              getDefaultConfiguration();
5         return ! conf.getDefaultTransform().isIdentity();
6     }
```

ソースコード 4.3: Retina ディスプレイの表示倍率を取得する関数

```
1 public static int getRetinaScale(int shareScreenNumber) {
2         int scale = 1;
3         GraphicsEnvironment env = GraphicsEnvironment.getLocalGraphicsEnvironment();
4         final GraphicsDevice[] devices = env.getScreenDevices();
5         GraphicsConfiguration conf = devices[shareScreenNumber].
              getDefaultConfiguration();
6         scale = (int)conf.getDefaultTransform().getScaleX();
7        return scale;
8     }
```

java8 以前では表示倍率を取得し、その値より Retina ディスプレイかを判断していた (ソースコード 4.1 中 14 行目)。java9 以降では Retina ディスプレイの判断を行える API が提供された (ソースコード 4.2 中 5 行目)。またソースコード 4.3 の 6 行目にて、接続しているディスプレイの表示倍率を取得する API も提供されため、例外を考える必要がなくなり、コード量が減少した。

4.3 デバッグオプションの修正

TreeVNC のオプションの 1 つに-p オプションがある。通常、TreeVNC のデバッグを行うにはサーバ側とクライアント側の 2 つを同じ PC 上で実行する必要があった。また、ソースコードを変更後には bulid を行わないとデバッグできないという煩わしさがあった。

-p オプションはその煩わしさを解消するために、自らとソケット通信を行うことでデバッグが可能となるオプションとして設計されている。しかし、TreeVNC の仕様変更等の理由により画面データとしての Rectnagle がうまく構成されず使用できないという状態だった。

Blocking の実装中は何度もデバッグを行う必要があったため、前述の煩わしさの解消のために-p オプションの修正を行った。

-p オプションが正常作動しなかった原因として、ディスプレイの選択を行なっていなかったことが挙げられる。設計当初の-p オプションは CUI での動作を想定しており、ディスプレイがない場合もあるという想定だった。そのため、画面配信用の画面が選択されておらず、画面データがうまく生成されない状況が発生していた。

これを、確実に GUI でビューワーを表示するという前提でディスプレイ選択を行うことで、正確に画面データの生成を行わせ-p オプションでのデバッグが可能となった。

第5章　今後の課題

参考文献

[1] hogetestusggdgs

謝辞

本研究の遂行，また本論文の作成にあたり、御多忙にも関わらず終始懇切なる御指導と御教授を賜わりました hoge 助教授に深く感謝したします。

また、本研究の遂行及び本論文の作成にあたり、日頃より終始懇切なる御教授と御指導を賜わりました hoge 教授に心より深く感謝致します。

数々の貴重な御助言と細かな御配慮を戴いた hoge 研究室の hoge 氏に深く感謝致します。

また一年間共に研究を行い、暖かな気遣いと励ましをもって支えてくれた hoge 研究室の hoge 君、hoge 君、hoge さん並びに hoge 研究室の hoge、hoge 君、hoge 君、hoge 君、hoge 君に感謝致します。

最後に、有意義な時間を共に過ごした情報工学科の学友、並びに物心両面で支えてくれた両親に深く感謝致します。

2010 年 3 月

hoge