# 授業やゼミ向けの画面共有システム TreeVNCの設計と実装

# A Screen Share System using Tree Structure for Seminar and Classwork

平成25年度 学位論文(修士)

琉球大学大学院 理工学研究科
情報工学専攻

谷成 雄

# 要 旨

各クライアントをツリー状に接続し、親が配信したデータを木の上から下へとリレーさせることのできる分散版 VNC(TreeVNC) のアプリケーションを実装した。通常の VNC では配信者へ負荷が集中する設計となっている。例えば、大学の講義等で通常の VNC を用いて画面共有を行った時、クライアントの増加に比例して配信者への負荷が増えてしまう。この問題を解決する為に、ツリー構造にクライアントを接続させ、データを上から下へ流していくことでスケーラビリティを持たせた。スケーラビリティを持つサービスとは、利用者が増加してもサービスの質を落とすこなく利用できるサービスのことである。TreeVNC を使用した結果、クライアントの数を増やしてもサーバ側への負荷を抑えスループットを落とすことなく利用する事ができた。更に、発表者をボタン一つで切り替えられるように UserInterface の拡張を行い、マルチディスプレイを使用している場合、一つのディスプレイのデータを受け取ることができるようにマルチディスプレイのサポートも行った。

# 要旨

I implemented the distributed application version VNC that can be connected to a tree-type of each clients, to relay the data parent is delivered (TreeVNC). it is designed the load is concentrated on the distributor in the usual VNC. for example, when performing screen sharing using VNC in lectures, load on the distributor would increase in proportion to the increase of the client. To solve this problem, it was to have a scalability and the child is allowed to connect the client to the tree structure, send data from the server to the client. The service with scalability refers to a service that can be used not come sacrificing quality of the service user may be increased. As a result of using TreeVNC, it could be used without reducing the throughput suppress the load on the server side can also increase the number of clients. Further, if it is to be switched by one button presenter, you have a multi-display, we also extended the UserInterface receiving the data of the display so that one possible.

# 目 次

# 図 目 次

# 表 目 次

# 第1章　序論

普段授業を行う際、プロジェクタなどの投影機を使用して授業を進める風景をよく目にする。しかし、広い部屋だと後ろの席に座っている生徒が見えにくいなどの不便を感じることがよくある。もし、授業を受けている学生の手元にパソコンがあるならば、手元のパソコンに先生の説明しているスライドを表示して授業を進めることでどこの席に座っていても、手元の画面に表示されるので見えづらいという問題は解決される。
みんなの手元に先生の画面を配信するシステムとして、ビデオケーブルを引いて画面を配信する方法があり、このような方法で画面を配信するのが一般的である。しかし、この方法で画面共有をするには、工事を行ってビデオケーブル引かなければならないので、コストがかかってしまう。
ビデオケーブルを引かずに、WEB ページに授業のスライドを載せることで、擬似的に画面を共有することもできる。しかし、この場合は、ページが同期していないのでどのスライドを説明してるのかわからなくなるといった問題がある。プログラミングの授業などでは、先生がどのような作業をしているかがとても重要になってくるが、これは WEB ページを使用しても実現することができない。
これらの問題を解決するために、オープンソースなアプリケーションである VNC を用いることで、無料で画面を共有することができる。しかし、VNC は多人数で同時に接続してしまうと処理性能が落ちて授業の進行に画面がついていかなくなったり、アプリケーションの処理自体が止まってしまったりしてしまう。この問題の原因は一つのコンピュータに多人数が繋がるときに生じる問題である。
そこで、多人数で画面共有ができるようにクライアントをツリー構造に接続させ、上から順番にデータを流していく方法によって、VNC サーバに対する負荷を分散させることができると考えた。更に、ゼミで VNC を使用することを想定する。従来の VNC では、発表者が変わるごとに新しく VNC に接続し直す必要がある。このような手間を省くことで、ゼミをスムーズに進行させることができる。
本研究では、多人数で画面共有ができるようにクライアントをツリー構造に接続させ、上から順番にデータを流していく方法で、VNC サーバに対する負荷を分散させる TreeVNC を作成し、更に、ゼミなどで使いやすいようにユーザインタフェースの提案と実装を行う。

## 1.1　本論文の構成

本論文では、第 2 章でこれまでの画面共有システムについて説明し、その中で使用されているプロトコルについて述べ、VNC を授業で使用する際の問題点を挙げる。
第 3 章では、第 2 章で挙げた問題点に対して考察を行い、Tree 構造を使用した設計やユーザインターフェースの提案を行う。
第 4 章では、第 3 章で設計した木構造を利用した TreeVNC の詳細な設計と実装方法そして、実装時の問題点とその解決方法ついて述べる。
第 5 章では、卒業論文で作成した TreeVNC との違いについて述べる。第 6 章では、画面共有システム TreeVNC の評価を行い、作成したユーザインタフェースの妥当性について述べる。第 7 章では、本研究のまとめとこれからの課題について述べる。

# 第2章　画面共有システム

この章では、従来画面共有で使用されている TightVNC とそれに使われてる RFB プロトコルについて説明する。その上で、多人数で使用する際の問題点を説明する。

## 2.1　RFB プロトコル

RFB(remote frame buffer) プロトコル [1] とは、画面共有システムなどに用いられている、リモートのグラフィカルユーザーインターフェースにアクセスするためのプロトコルである。ユーザが居る側を RFB クライアント側と呼び、フレームバッファ(画像データ)への更新が行われる側は RFB サーバと呼ぶ。
RFB プロトコルでは、最初にサーバ・クライアント間でハンドシェイクが行われる。ハンドシェイクでは、プロトコルバージョンの確認・接続に対しての認証が行われる。
ハンドシェイク後には、クライアントに向けて初期メッセージが送信される。初期メッセージにはフレームバッファ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。
RFB サーバ側はフレームバッファの更新が行われるたびに、RFB クライアントに対してフーレムバッファの更新データを矩形で送信する。更に RFB クライアントのフレームバッファアップートリクエストが来るとそれに答え返信する。
RFB プロトコルは、描画データに使われっるエンコードが多数用意されており、また独自のエンコードを実装することもできるシンプルなプロトコルである。
RFB プロトコルの概要を図 2.1 に示す。

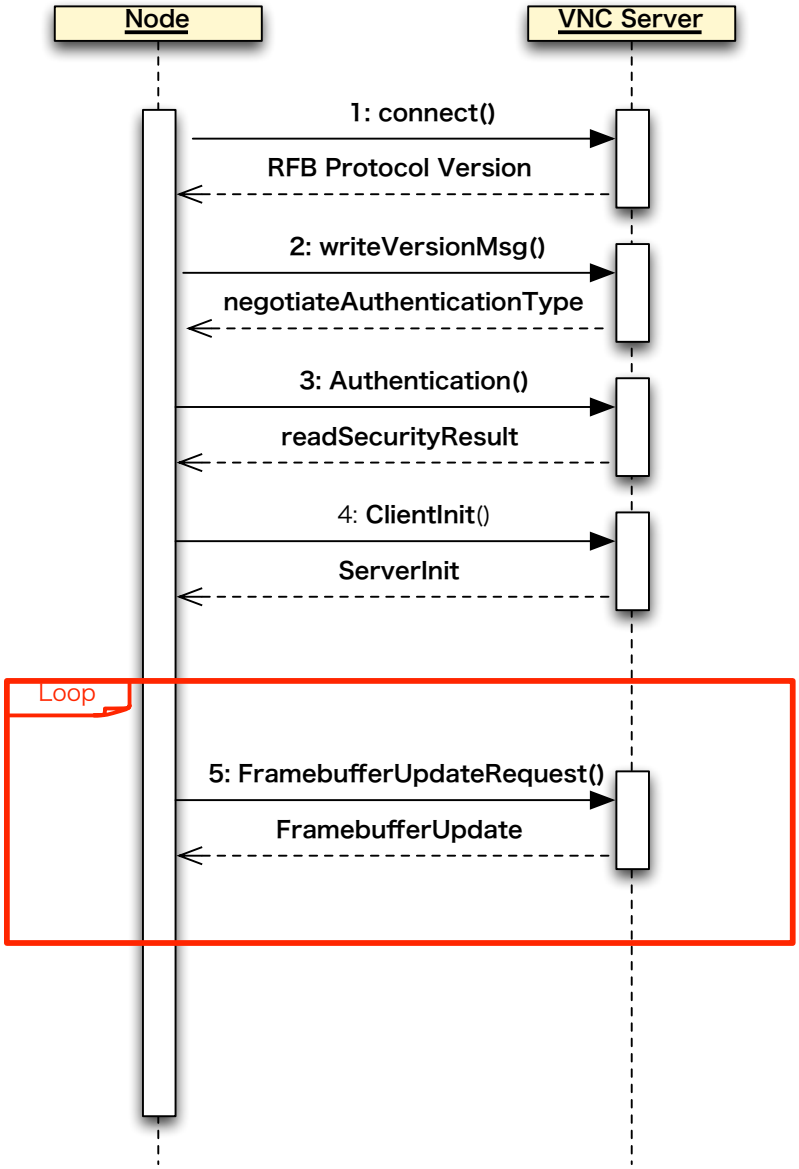


図 2.1: RFB プロトコル

## 2.2 TightVNC

TightVNC(Tight Virtual Network Computing)[2] は RFB プロトコルを用いて作成された、画面共有をするためのオープンソースなアプリケーションである。TightVNC はサーバ側とクライアント (ビューア) 側に分かれていて、サーバを起動し、クライアントがサーバに接続を行い画面共有を可能とする。TightVNC の特徴として、圧縮された画像データを扱うことができることが挙げられる。そのため、低速回線環境でも動作できるようになっている。しかしエンコードとデコードを行う分 CPU のパワーが必要になる。

## 2.3 VNC Reflector

VNCReflector はサーバへの負荷を減らすように設計されている VNC ソフトの一つである。Java を用いて実装されており、フリーで入手することができる。Vnc Reflector は、Vnc サーバとクライアントとの間に入り、VNC サーバとの通信を代わりに行うプログラムである。クライアントは Vnc Reflector へ接続するので、VNC サーバとの接続は Vnc Reflector のみとなり、VNC サーバ側の負荷を減らすことができる。しかし、VNC Reflector も接続は Proxy に一極に集中してしまっているためサーバの負荷は軽減するが、Proxy に対しては負荷がかかる。

## 2.4 授業で VNC を使用するときの問題点

VNC を多人数で使用する際は、サーバに対してクライアントの接続が一極集中してしまうことが問題である。実験としてまず、iMac で複数の PC から VNC をかけ検証してみた。10 台接続すると VNC クライアントでの画面の更新が遅くなり、さらに CPU 使用率も跳ね上がっていた。

表 2.1: スループットと CPU 使用率

| | スループット (Byte) | CPU 使用率 |
|---|---|---|
| １台 | 20M(最大速度) | 55% |
| ４８台 | 5M | 100% |

一本の通信網への負荷が高く、サーバの CPU への負荷が高いというのが問題である。

## 2.5　ゼミで VNC を使用するときの問題点

ゼミで VNC を利用することを想定する。ゼミでは通常発表者が複数人いるので、発表者が切り替わる。VNC を用いて発表を行う場合、発表者が切り替わるごとに接続し直さなければならない。それに伴い接続し直す際の認証が毎回発生する。
また、高解像度のマルチディスプレイを使用している人などがいる場合は、送信される画像データの量が多くなりすぎてしまいメモリを圧迫してしまうことがある。

## 2.6　Broadcast と Multicast

Broadcast は同じネットワークアドレス (同一セグメント) 上のすべての端末に対して、同時にデータを送信することである (図 2.2)。Multicast は同じマルチキャストアドレスを持った端末に同時にデータを送信することである (図 2.3)。画像データを配信する際に Broadcast や Multicast を用いて、画像データを送信すると、木構造を構成する必要がなくなり、画像データ送信も一回送信するだけで良い。そこで、Broadcast を用いた実装について考察してみた。

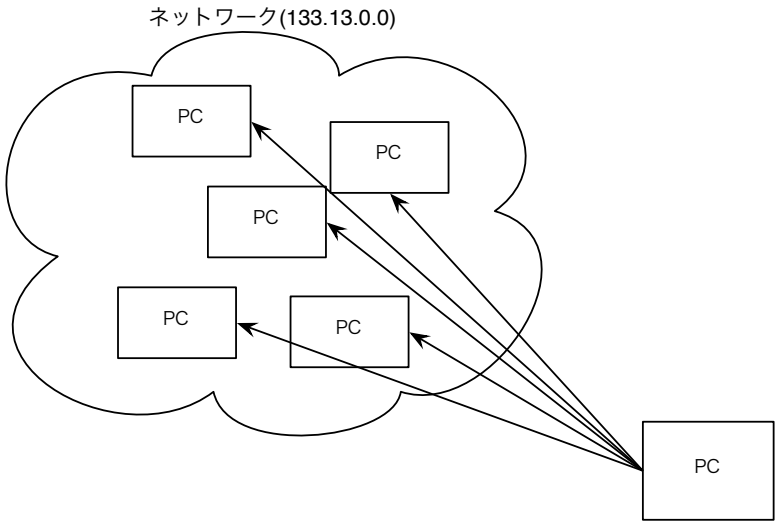


図 2.2: Broadcast

### 2.6.1　Braodcast パケットの性質

Broacdcast パケットの性質として大きすぎるデータの送信ができないという性質がある。どの程度の大きさのパケットまで送れるのかをテストしてみた結果 64000byte までだと送信することができることがわかった。

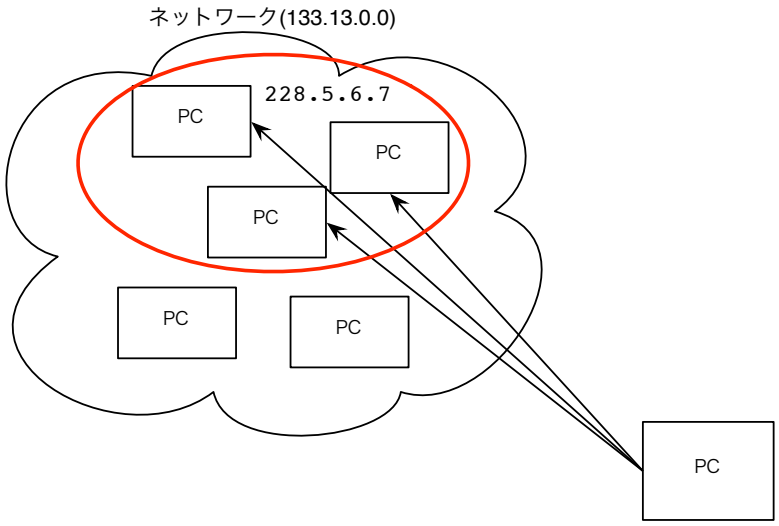


図 2.3: Multicast

もう一つの性質にパケットが消失 (ロスト) しても特定することができないという性質がある。この問題に対しては後述する。Multicast についても Broadcast と同じ性質を持っている。

### 2.6.2　パケットの分割

VNC では画像データが更新される際に矩形単位で更新データを送信する。
画面全体の更新やマルテディスプレイの更新などの大きいデータの更新ではどうしてもデータサイズが大きくなってしまう。例えば 1920*1080 の解像度のディスプレイのデータ量は 6220800byte(1920*1080*3) である。
Broadcast では、このような大きなデータを受け取った際は、64000byte 以下に分割して送らなければならない。私が作成している TreeVNC では Root Node に送られてきた画像データを一旦解凍し、再圧縮して Node に送っているため、解凍した後に分割して、64000byte 以下にして圧縮後送信することで実現できる。

### 2.6.3　消失したパケットの検出

Broadcast の性質で説明したとおり、Broadcast ではデータが消失したことをクライアントが検出することができない。そこでプロトコルを拡張し、データごとにシリアル番号を振り、連続でない値が来た場合、正しくデータが届いていないと判断することができる。

### 2.6.4　Acknowledge の設計

データが消失したのを検出した際、どのような対応をするのかが問題になる。シリアル番号を振っているのでプロキシに消失したデータのシリアル番号を指定することで、消失したデータの再送が可能となる。この際、再送に Broadcast を用いると、データの消失が起こるので、消失したデータを再送する際は TCP コネクションを用いて送信を行うのが良い。

# 第3章　画面共有システムTreeVNCの設計

## 3.1　木構造を用いたTreeVNCの設計

まず、多人数が参加している授業でVNCを使う場合に起こる問題は、一つのコンピュータに多人数が繋がり、処理が集中してしまって、性能が大幅に落ちてしまうところである(図3.1)。多人数の同時接続を可能にするには、一極集中で接続するのではなく、Node同士で負荷を分散させることによって実現できるのではないかと考えた。
負荷分散を行う際、Node同士どのようなトポロジを組むのが適切か検討した結果、上から流れてきたデータを下のNodeへと伝えていくことのできる木構造が良いと考えた。

今回行った設計ではNodeを木構造に接続させデータを流すためにサーバとNodeの間にRoot Node(サーバとNodeの通信を仲介するもの)を設置する方式をとった。Root Nodeは主にNodeの管理とServerから流れてきた画像データの管理を担当する。
木構造で設計したものを(図3.2)に示す。

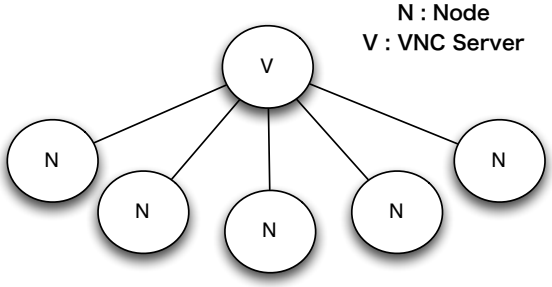


図 3.1: VNCの構造

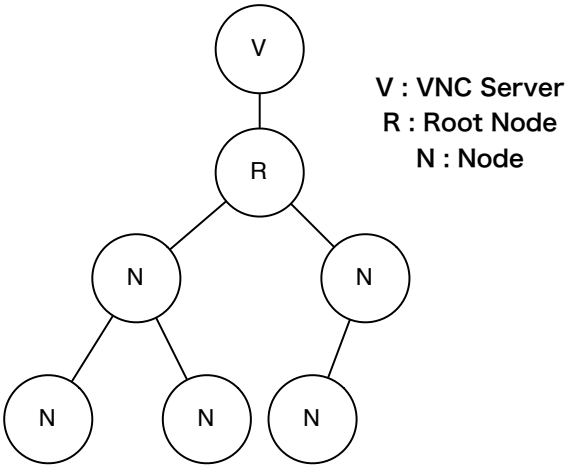


図 3.2: TreeVNCの構造

## 3.2　TreeVNC の原理

TreeVNC がどのような負荷分散を行った結果負荷がどうなっているのかを説明する。通常の VNC では、一極集中型でサーバに接続してしまうのでサーバに負荷がかかって性能を低下させたり停止してしまっている。そこで Node を木構造に配置させることで、負荷がなくなっているように見える。しかし、実は負荷がスイッチにかかっていて、消えているわけではない。通常の VNC と TreeVNC の構造を比較した図を (図 3.3) に示す。

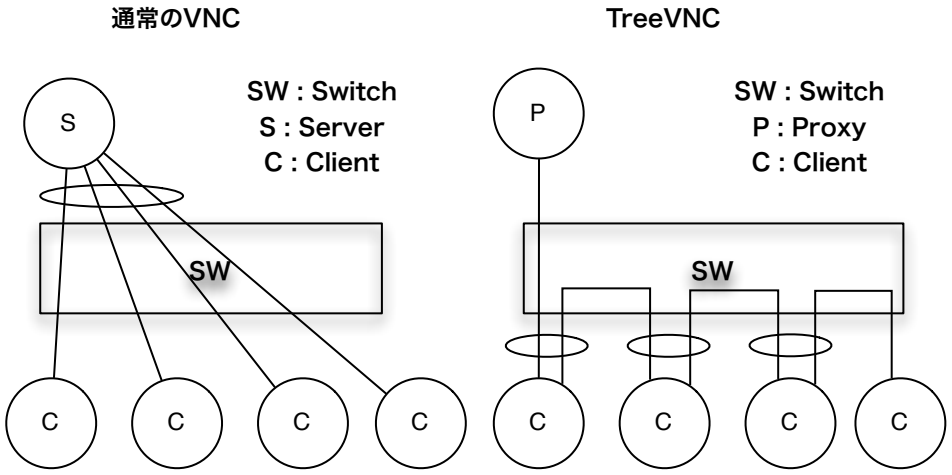


図 3.3: TreeVNC の構造

(表 3.1) はポート一本あたりの通信量である。
表から推察できるように、ポート一本あたりの負荷は通常の VNC の場合は Node 数に比例して増えている。しかし TreeVNC の場合は Tree の子供の数が一定なので、Node 数に関係なく一定である。
送信する量も通常の VNC の場合 Node 数に比例した量のデータ送信しなければならいので、CPU に負荷がかかり性能が低下したり停止したりしている。
対して TreeVNC はが増えても配信するデータは一定なので性能が低下せず使用することができる。

表 3.1: ポート一本あたりの通信量 (N は Node 数、M は Tree の子供の数)

| | 通常の VNC | TreeVNC |
|---|---|---|
| 通信量 | N * データ量 | (M + 1) * データ量 |

### 3.2.1　木の生成

負荷を分散させるために木構造を用いるので、Node をツリー状に接続する仕組みが必要である。TreeVNC では、以下の点順で、木の構成を行う。
図 3.4 は、２分木で木を構成する際の手続きを示したシーケンスダイアグラムである。
1. 初めに Node は Root Node に接続先 IP を尋ねる (Where connect)。
2.Root Node は、Node に接続するべきホストの IP を引き渡す (Answer)。
3.Node は指定されたホストに接続する (Connect)。

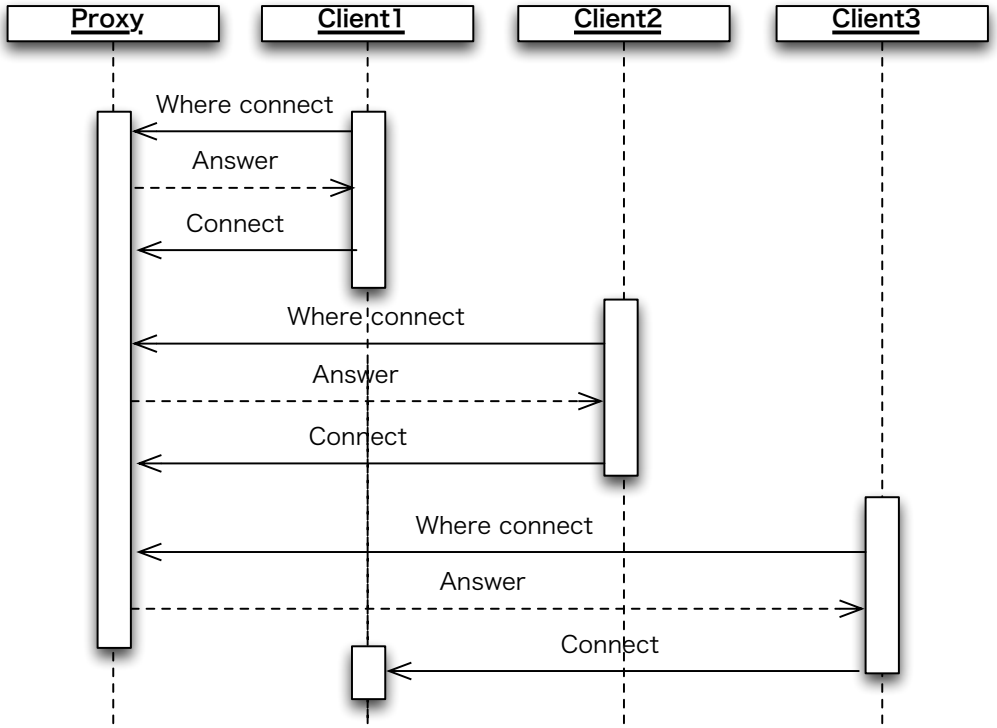


図 3.4: TreeVNC の構造

### 3.2.2　Root Node の仕事

Root Node の大きな仕事は、Node の管理である。Node の IP アドレス情報を Java の LinkedList で保持している。LinkedList の簡単な説明を 3.2 に示す。

表 3.2: LinkedList

| 関数 | 説明 |
|---|---|
| add(E e) | List の最後に e を追加する。 |
| get(int index) | リストの *index* 番目の値を取得する。 |
| remove(int index) | リストの *index* 番目の値を削除する。 |

Listing3.1 は Node を管理している部分のプログラムである。

*line* は接続先を訪ねてきた Node の IP アドレスである。初めに接続してきた、Node のアドレスを自分が保持している list(LinkedList) に登録する。
親の番号は $(counter - 1)/TreeBraanche$ で求めることができるので、list に親の番号を指定し、親の IP アドレスを取得する。
取得した Ip を接続してきた Node(*line*) に送り返すことで、Node はどの IP アドレスに接続すればよいのか知ることができる。

Listing 3.1: クライアント管理のプログラム

```
1  private LinkedList<String> ls = new LinkedList()<String>;
2  private synchronized void replyCreateTree(PrintStream os, String line) throws
       InterruptedException {
3    ls.add(line);
4    parentnum = (nodeCounter − 1) / treebranch;
5    request = ls.get(parentnum);
6    outputStream(os, request, String.valueOf(parentnum), String.valueOf(nodeCounter));
7  }
```

## 3.3　表示画面の切り替え

VNC を使用して画面共有を行う場合、授業などでは講師の画面を共有していれば問題ない。しかし、ゼミなどの発表者が多数いる場合は画面共有の対象をを切り替えなければならない。画面共有対象の切り替えを行う場合、発表者ごとにサーバを立ち上げ直さなければならないという問題が発生する。そこで、ユーザ側から Root Node にリクエストを出して、画面共有の対象を切り替える機能が必要になる。画面を切り替えるときは、Root Node だけが接続を切り替え、Node たちはそのデータを受け取るだけで良い (図 3.5)。

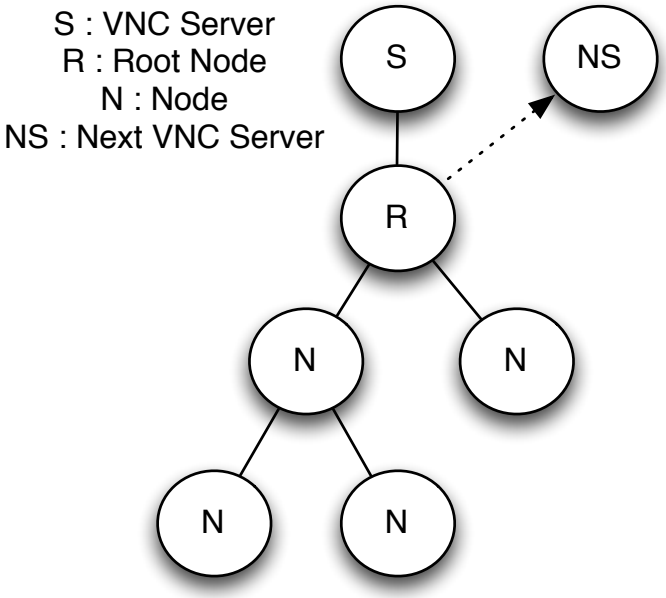


図 3.5: 表示画面の切り替え

画面の切替をどのユーザが行うのかという問題がある。Root Node に対してユーザが毎回 IP アドレスを入力して切り替えるのは手間がかかる。そこで、Node 側に画面切り替えボタンを設置し、ボタンを押すと Root Node に自分の画面へ切り替えるように命令を出し Root Node が了承すると画面が切り替わるように設計した。

## 3.4　マルチディスプレイの対応

マルチディスプレイを用いて VNC を行うと、すべてのディスプレイのデータが繋がって表示されてしまう。通常発表などに使用されるディスプレイは一つである。すべての画面のデータを送ってしまうとその分だけ無駄なデータを送っていることになる。そこで、発表用に使用する画面のデータだけを送ることのできるようにディスプレイが指定できるようになることが必要になる。

## 3.5　再接続

木を構成することはできたが途中の Node が切断してしまった場合に木を再構成しなければならない。木を再構成する手順は以下の用に行う。

1. 子供のリーダー (最初に親につないだ子供) が親が落ちたことを Root Node に対して報告する。
2. Root Node は報告を受けると番号の一番大きい Node(最後の Node) に対して落ちた親の代 わりになるように報告する。
3. Root Node から命令を受けた Node は指定された Node に接続をしなおす。
4. Root Node は Node のリストを更新して、親が落ちた子供たちに新しい親の情報を教える。
5. 親が切断された子供たちは、Root Node からもらった情報を元に新しい親に対して接続を行う。

このようにして木を再構成することができる。

図 3.6 は再接続の様子を記したコラボレーションダイアグラムである。以下に関数の説明をする。

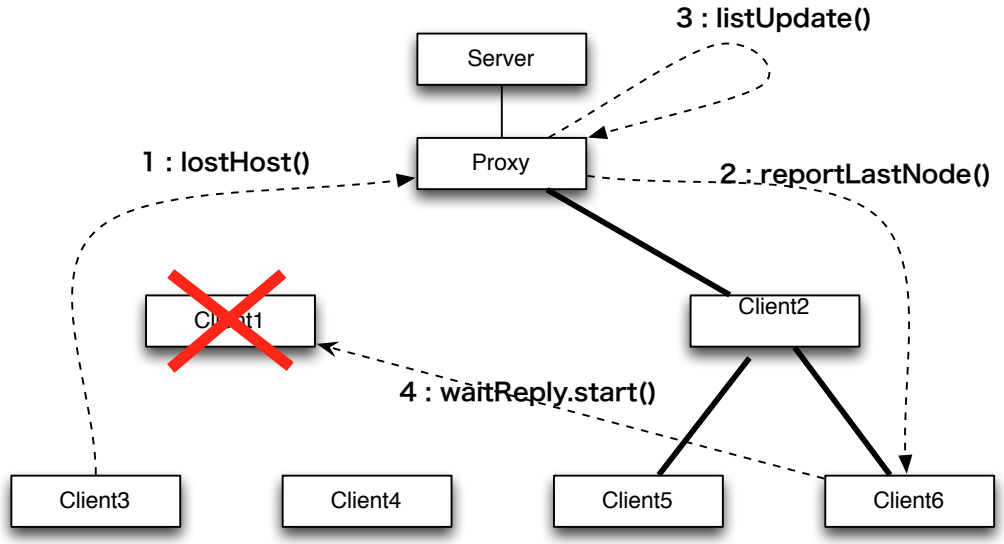


図 3.6: 再接続の手続き

1:lostHost():親に切断されたこと報告する関数である。

2:reportLastNode():番号の一番大きい (最後のノード) に対して親の代わりをするように命令する関数である。

3:listUpdate():Root Node が持つ Node のリスト情報をアップデートする (切断したノードを削除し最後のノードのアドレスをそこに追加する)。

4:waitReply.start():Node は waitReply というクラスを main スレッドとは別にスレッドを作成して走らせている。もし Root Node からの命令が来ると Node は Root Node から指定された場所に接続を行う。

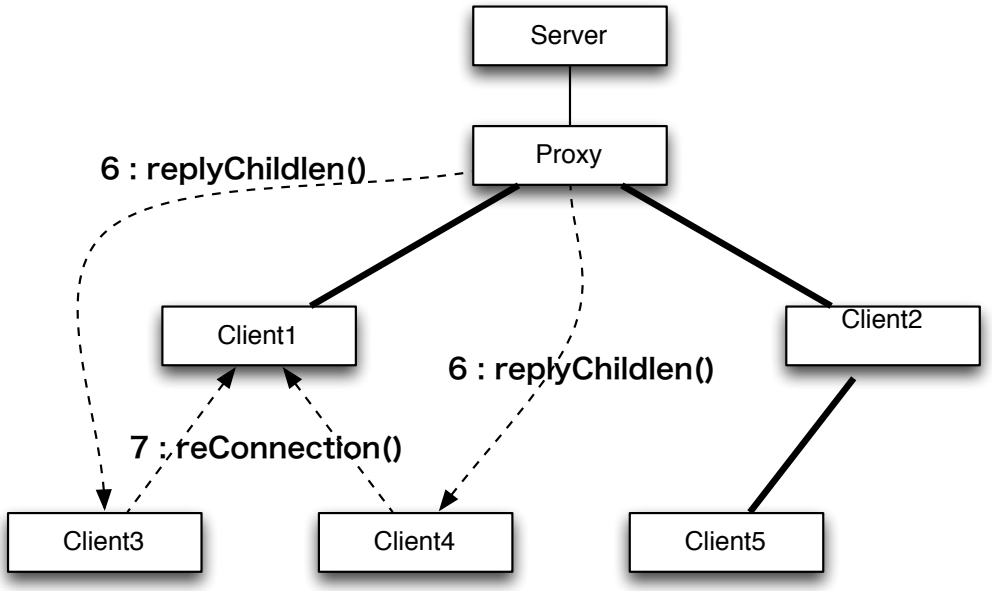


図 3.7: 再接続の手続き 2

6:replyChildlen() は、親が切断した子供たちに対して新しい親の情報を報告する関数である。

7:reConnection() は Root Node から来た情報をもとに VNC 接続を行う関数である。
以上の関数を用いることで Node が落ちても木を再構成することができる。

## 3.6 MulticastQueue

画面が更新された際に画像の更新を Node に伝えなければならない。ノードが多数ある場合、各ノードに順番に更新を知らせるのではなく、同時に画面の更新を知らせたい。同時に更新を知らせるために、CountDownLatch を用いて MultiCastQueue を作成した。

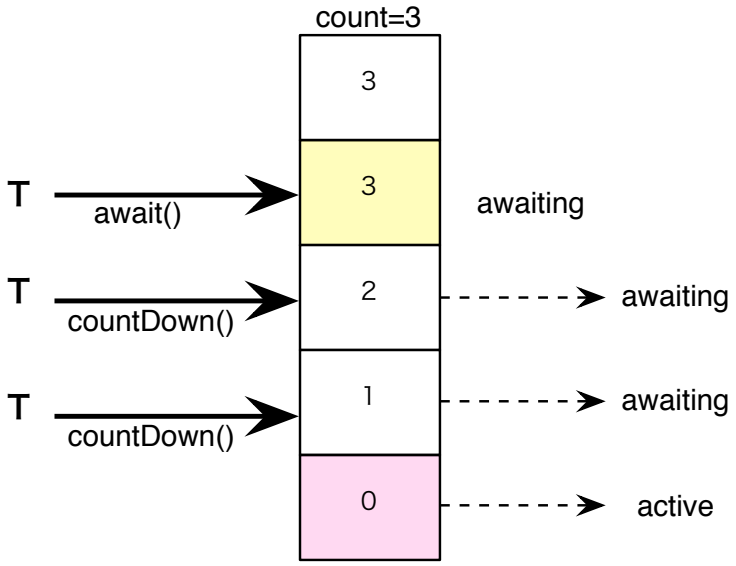


図 3.8: CountDownLatch

java.util.concurrent.CountDownLatch は java の並列用に用意された 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のできるクラスである。使い方は、カウント (何回カウントしたらスレッドを開放するのかの数) を指定して CountDownLatch の初期化を行う。countDown メソッドを使うとカウントダウンを行うことができる。await メソッドは cowntDown メソッドの呼び出しの結果カウントがゼロになるまでの間スレッドをブロックすることができる。countDwon メソッドの結果がゼロになると await メソッドで停止していたスレッドが動き出す (図 3.8)。MulticastQueue は Queue のように使用することができる。put メソッドを使用してデータを queue に追加する。データを put する際に CountDownLatch をカウントダウンする。poll メソッドを用いることで、次のデータを取得することができる。poll メソッドの中で await が使われているので次の put でデータが来るまでスレッドがブロックする。(図 3.9)

新しくデータが put されるとデータの読み込みが再開される。(図 3.10)

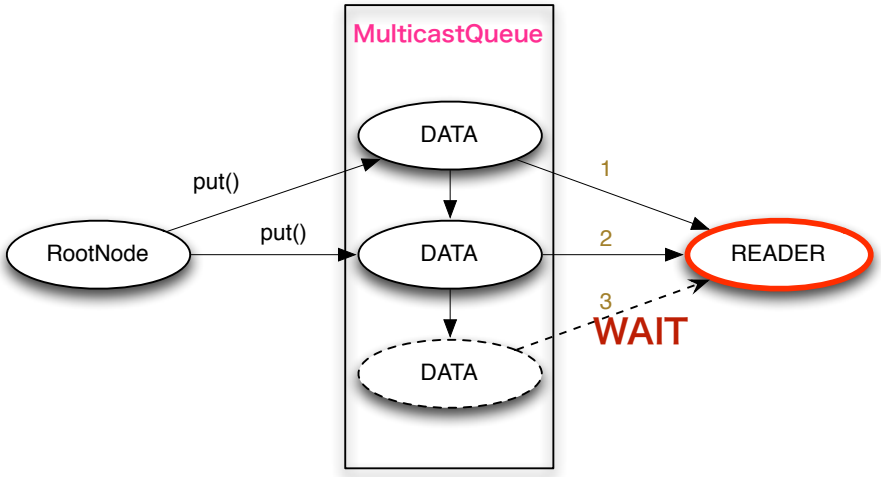


図 3.9: MulticastQueue(データが来るまで待つ)

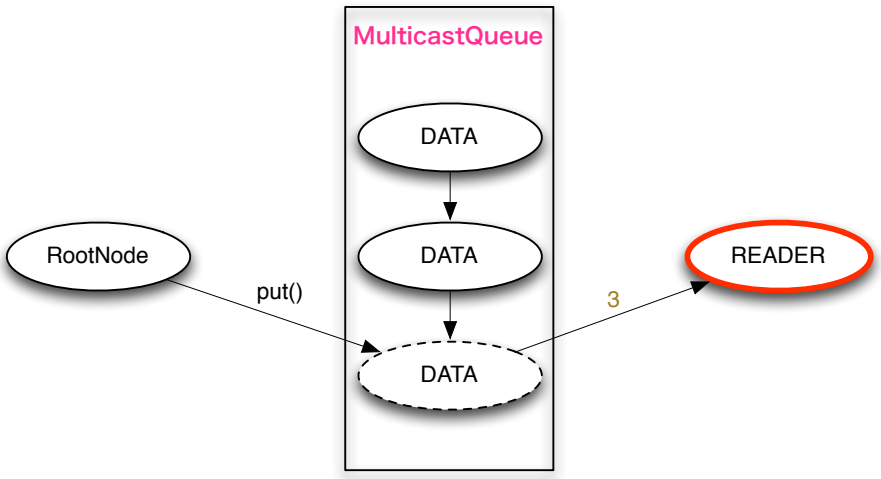


図 3.10: MulticastQueue(新しいデータが来るとデータを読み出す)

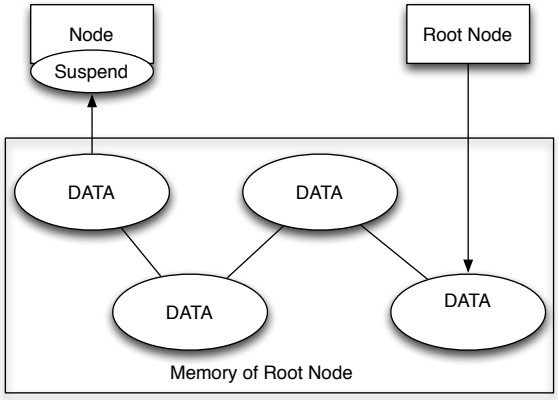


図 3.11: Node サスペンド時の Root Node のメモリの様子。データが残り続けメモリを圧迫してしまう。

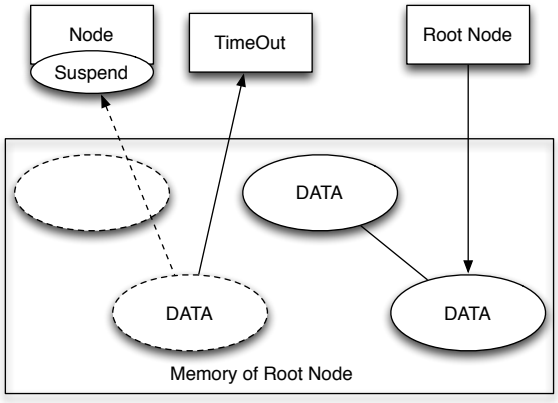


図 3.12: TimeOut が代わりにデータを取得する

### 3.6.1　TimeOut

MultiCastQueue を使ってのデータの取得には問題が発生した。それは、接続してきた Node がデータを取得しない状況、例えばサスペンド状態になったときに Root Node のメモリの中にデータが残り続けるというものである。メモリに残り続けたデータはやがてメモリオーバーフローを引き起こしてしまうのである。その様子を図 3.11 に示す。TimeOut スレッドが Node の代わりにデータを取得することで、MulticastQueue の中からデータが削除され Root Node のメモリを圧迫することがなくなった。(図 3.12)

そこで、ある一定の時間が経過すると代わりにデータを取得してくれる TimeOut 用のスレッドを作成した。TimeOut スレッドはサスペンドしている Node の代わりにデータを取得する。

## 3.7　圧縮の問題

VNC で扱う RFB プロトコルには、使えるエンコーディングのタイプの 1 つとして ZRLE(Zlib Run-Length Encoding) がある。ZRLE 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けられ送られてくる。Zlib はフリーのデータ圧縮及び解凍を行うライブラリである。可逆圧縮アルゴリズムの圧縮と解凍が行える java.util.zip.deflater と java.util.zip.inflater を実装している。

### 3.7.1　java.util.zip.deflater の実装の問題

Zlib 圧縮は辞書を持っていて、その辞書に登録されているデータを元に解凍が行われる (図 3.13)。しかし、java.util.zip.deflater は現在持っている辞書を書き出すこと (flush) ができないことが分かった。辞書を書きだすことができない為、Zlib 圧縮されたデータを途中から受け取ってもデータが正しく解凍を行うことができない (図 3.14)。

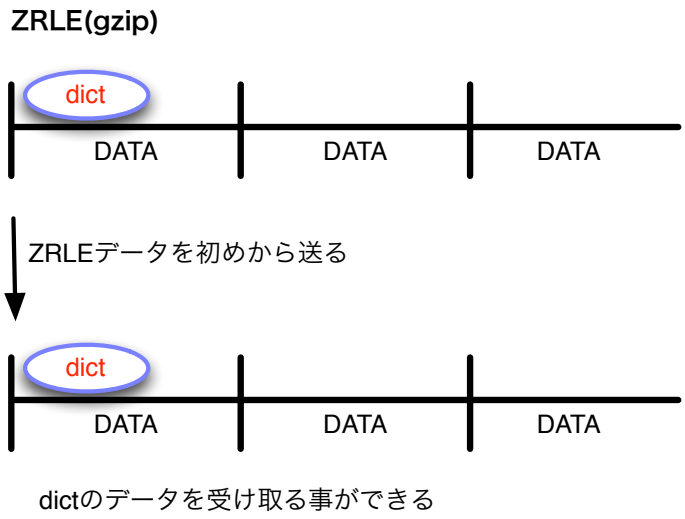


図 3.13: ZRLE

### 3.7.2　ZRLEE

そこで、Root Node が ZRLE で受け取ったデータを unzip し、データを zip し直して最後に finish() をいれることで初めからデータを読んでいなくても解凍を行えるようにした (毎回新しい辞書を使うようにした)。(図 3.15) このエンコードは ZRLEE エンコードと定義した。一度 ZRLEE エンコードに変換してしまえば、そのデータをそのまま流すだけで良い。よって変換は Root Node が行う一回だけですむ。ただし、deflater,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 と Node 側では毎回

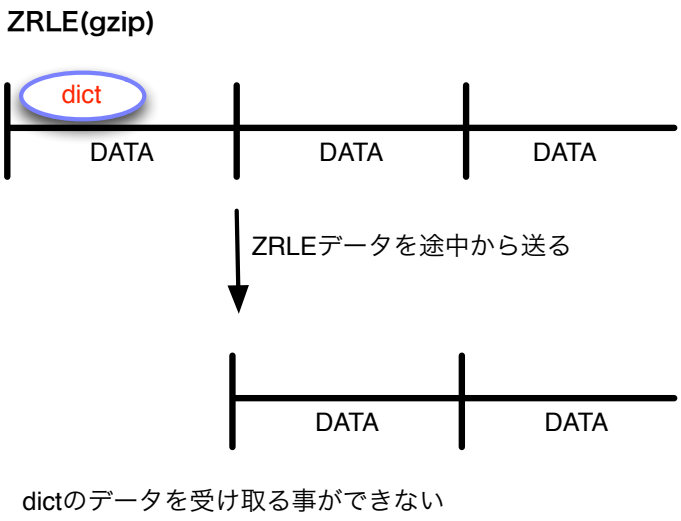


図 3.14: ZRLE2

新しく作る必要がある (Node 側は inflater だけ)。また、ZRLEE は Node 側が対応していなければならないという問題がある。

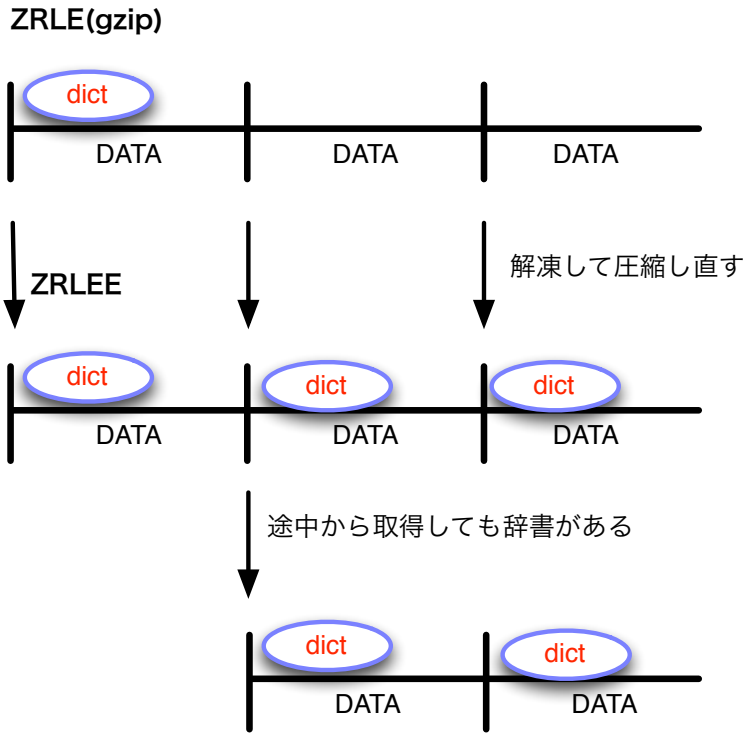


図 3.15: ZRLEE

### 3.7.3　接続先自動検索システム

Node が Root Node に接続する際、Root Node の IP アドレスを指定する必要がある。IP アドレスを毎回入力するのは手間がかかる上、間違うと接続することができない。
そこで、Node が起動した際に、起動している TreeVNC の Root Node を検索し、IP アドレスの情報を取得し、一覧にして選択させることによって直接アドレスを手入力する必要がなくなる。
TreeVNC の Root Node を検索する際には、Broadcast 通信を用いることによって実現することができる。

# 第4章　画面共有システムTreeVNCの実装

## 4.1　TightVNCのアップデートへの対応

TightVNC は現在も開発が続いていて、アップデートされている。このアップデートに対応するために、私が作成している TreeVNC にも対応させる必要がある。
卒業論文後から私は２種類のアップデートを行った。その２種類のアップデートの対応について説明する。
まず、はじめに行ったアップデートは Version が 1.x から 2.x へ変わるメジャーアップデートと呼ばれる大型アップデートである。
このアップデートでは、パッケージ構成が追加され、元のソースコードがほとんど残っていない状態であった。このような大型なアップデートに対応するには、新しい TightVNC を元にして、作成した TreeVNC の機能を一つづつ移行していく必要がある。このソースコードのアップデートに加えソースコードの質を高めるためにリファクタリングを行った。リファクタリングとは、将来の仕様変更に柔軟に対応できるようにソースコードの手直しを行うことである。

## 4.2　UIの実装

授業やゼミなどで使用してみて、必要な機能の提案が出てきたので、実装を行った。

### 4.2.1　マルチディスプレイへの対応

VNC では画面の情報を矩形型にして送信する。もし複数のディスプレイが存在する場合すべてのディスプレイの情報が送られてくる。しかし、発表などに使用するディスプレイは一つである場合が多い。
必要な画像が一つのディスプレイなのに、すべてのディスプレイのデータを送ると無駄なデータが発生する。そこで、ディスプレイを指定して、その画像だけ送信する機能を追加することで、無駄なデータ送信を省くことができる。
RFB プロトコルでは、FramebufferUpdate によって、矩形状の画像データが送信されてくる。
FrameBufferUpdate の概要を (表 4.1, 表 4.2) に示す。

表 4.1: FramebufferUpdate

| バイト数 | 型 | 説明 |
|---|---|---|
| 1 | U8 | message-type |
| 1 | U8 | padding |
| 2 | U16 | number-of-rectangles |

この後に number-of-rectangles の数だけ矩形のピクセルデータが続く。各矩形は (表 4.2) に示す。

表 4.2: FramebufferUpdate

| バイト数 | 型 | 説明 |
|---|---|---|
| 2 | U16 | x-position |
| 2 | U16 | y-position |
| 2 | U16 | width |
| 2 | U16 | height |
| 4 | S32 | encoding-type |

ここまでが header として送信されるデータである。矩形の画像なので x-position、y-position、width、height の４つの値で画像の位置と大きさを決めることができる。
header に続いて、実際の画像データが送信されてくる。
画像データは ZRLE エンコーディングで送信される。最初の 4 バイトはデータの大きさを表現して、次にその大きさ分の zlibData が送信される (表 4.3)。

表 4.3: ZRLE データ

| バイト数 | 型 | 説明 |
|---|---|---|
| 4 | U32 | length |
| length | U8 array | zlibData |

送られてきた zlibData は展開されると左から右、上から下へ並んだ、64*64 ピクセルのタイル群画像データとなる。

ここで、画像データがどのように送られてくるのかを調べてみたところ、２つディスプレイがあるとすると、両ディスプレイにまたがった画像更新が来ることがないことがわかった。
図 4.3 の黒い部分が画像データだとすると、図 4.3 のような FramebufferUpdate は送られてくることはない。

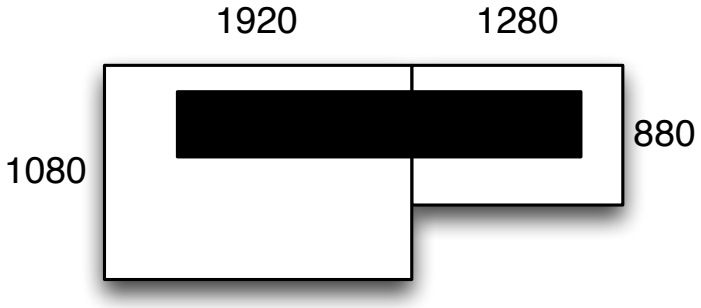


図 4.1: 画面更新時に来る可能性のない UpdateRectangle

以上のことを踏まえ、FramebufferUpdate で送信されてきた header を確認し、x-position を確認することで、どの画面の画像データを送信するかを選択することができる。
例えば、図 4.3 では、左側の画面を送信したいときは、x-position が 1920 より小さい場合送信し、右側を送信したい場合は 1920 以上のデータを送信するようにフィルタリングすることで実現できる。

### 4.2.2　表示画面の切り替え

ゼミなど発表者が多数いる状況で VNC を使用すると、発表者が切り替わるごとにサーバを立ち上げなおさなければならない。
画面の切替手順を図 4.2 に示す。

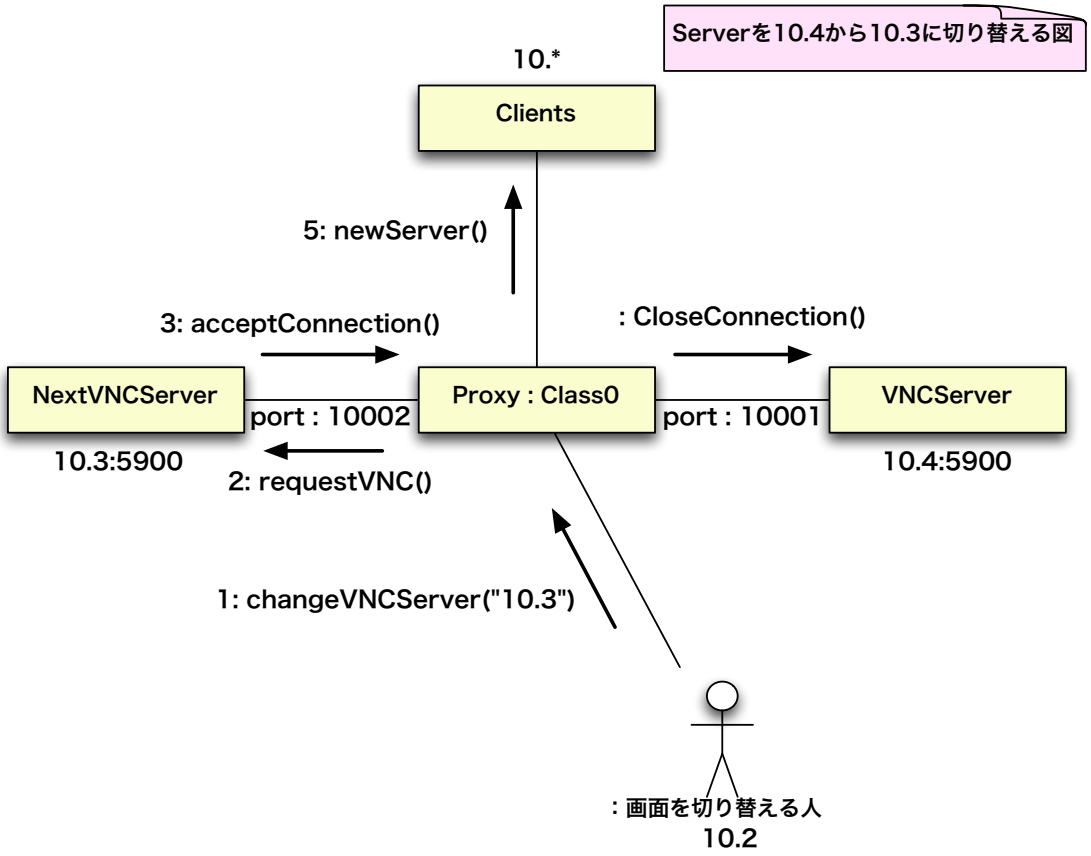


図 4.2: 画面切り替えの流れ

図 4.2 で使用されている関数の説明を表 4.4 に示す。

表 4.4: 画面切り替えの関数

| 名称 | 説明 |
|---|---|
| changeVNCServer(”10.3”) | 受け取った引数のアドレスへ切り替えるための命令を出す関数。 |
| requestVNC() | changeVNCServer() で指定されたホストに対して接続要求を出す関数。 |
| acceptConnection() | Root Node から接続要求が来たコンピュータが、要求に対して許可したことを Root Node へ報告する関数。 |
| CloseConnection() | 今まで使用していた画面共有のストリームを閉じるための関数 |
| newServer() | Node に画面が切り替わったことを報告する関数。この命令で Node は新しいストリームを受け取るようになる。 |

newServer() の内部処理をを Listing4.1 に示す。これは、Root Node が子供に対して、画面の切り替えが起こったことを知らせるソースコードである。
clientList は、現在接続されているクライアント情報が入っている。クライアントにそれぞれ TCP 接続を行い、サーバが変わったので接続し直させる命令を送信する。

Listing 4.1: 画面が切り替わったことを知らせるプログラム

```
1  for (String client : clientList) {
2      Socket echoSocket = new Socket(client, 10001);
3      DataOutputStream os = new DataOutputStream(echoSocket.getOutputStream());
4      os.writeBytes("reconnection\n");
5      os.close();
6  }
```

## 4.3 Authentication

Root Node がサーバに対して VNC 接続を行う際、ハンドシェイクが必要となる。ハンドシェイクの手順として、まず始めに Root Node がサーバに接続を行うと、サーバがサポートする最新のプロトコルバージョンが送られてくる。Root Node はサーバから送られてきたプロトコルバージョン以下の使用できるバージョンをサーバに対し送る。現時点で公開されているプロトコルバージョンは 3.3、3.7、3.8 だけである。今回 TreeVNC は 3.855 というバージョンを用意して 3.855 が来ると TreeVNC を使用するようにした。

プロトコルバージョンが決定すると、サーバ及び Node は、その接続で使用されるセキュリティに合意しなければならない。バージョン 3.7 以降ではサーバは自身のサポートするセキュリティタイプの一覧を提示する。Node のサポートする有効なセキュリティタイプを少なくとも一つサーバが提示した場合、Node はその接続上で使用されるセキュリティタイプを表す単一バイトを送り返す。

登録されているセキュリティタイプの一例として (表 4.5) のようなものがある。

表 4.5: AuthType

| **値** | 名称 |
|---|---|
| **0** | Invalid |
| **2** | None |
| **5** | RA2 |
| **18** | TLS |
| **21** | MD5 ハッシュ認証 |

MAC OS X SnowLeopard で起動している VNC サーバに接続するときには MAC 専用の認証の値 35 がありこれでパスワード認証を行うことができていた。

しかし MAC OS X Lion でパスワード認証を行おうとすると、MAC OS X Lion にしてパスワード認証ができなくなったので、別の認証方法で認証を行うことにした。

調べてみると MAC OS X が返してくる認証番号は [30, 31, 32, 2, 35] がある。32 はサーバに対して画面要求の認証を求めるタイプの認証であることがわかった。この認証を用いるとサーバに対して Root Node が接続する際にサーバ側に確認画面が出るようになる。サーバ側がこれを容認すると認証が成立する。

## 4.4 BroadcastとMulticast

４章で述べたとおり、Broadcast を使用する場合は一回に送信するパケットのサイズを 64000byte 以下にしなければならない。もし、1920*1080 のサイズの画像データを送信する際、約 600 万 byte(1920*1080*3) となってしまう。これでは送信することができないのでデータを分割し 64000byte 以下にして送信しなければならない。
作成した TreeVNC では、サーバから受け取った画像データを Root Node が一旦解凍して、再圧縮し、Node たちに送信するという方式を取っている。
一旦解凍した後は RawData となるのでデータを分割することができる。TreeVNC では ZRLE エンコーディングを使用して、これを解凍すると左から右へ上から下への順の 64*64 のタイル郡の RawData となる。
解凍された RawData を図 4.3 に示す。

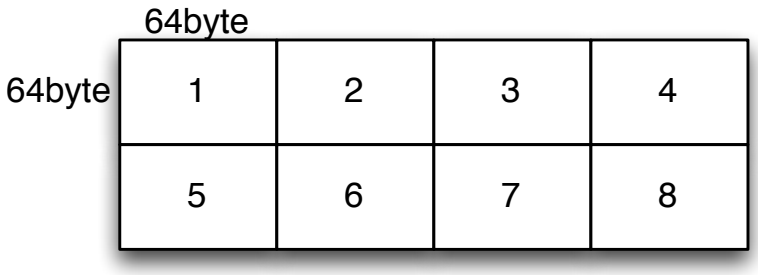


図 4.3: RawData の構造

ひとまずテストで、3.3 のように 2 列づつに分割して送信するプログラムを書いた (Listing4.2)。
毎回バッファをコピーして送っているため画面共有に支障をきたすほど処理が遅くなってしまった。

Listing 4.2: 画像データを分割するプログラム

```
1  LinkedList<ByteBuffer> output = new LinkedList<ByteBuffer>();
2  int width = header.getShort(8);
3  int height = header.getShort(10);
4  int y = header.getShort(6);
5  int dataLen = width * 64 * 3 * 2;
6  int temp = 0;
7  int count = height / 128;
8
9  if (width \% 64 == 0)
10   dataLen += (width / 64) * 2;
11 else
12   dataLen += (((width / 64) + 1) * 2);
13
14 for (int i = 0; i < count; i++) {
15   int tempDataLen = dataLen - temp;
16
17   while (tempDataLen > INFLATE_BUFSIZE) {
18     output.addLast(input.poll());
19     tempDataLen -= INFLATE_BUFSIZE;
20   }
21   if (tempDataLen == INFLATE_BUFSIZE) {
22     output.addLast(input.poll());
23     createHeader(header, i, height, y);
24     zipSplitData(header, output);
25     output.clear();
26     temp = INFLATE_BUFSIZE;
27   } else {
28     ByteBuffer tempBuf = input.poll();
29
30     ByteBuffer buf1 = ByteBuffer.allocate(INFLATE_BUFSIZE);
31     if(tempBuf.remaining()>tempDataLen)
32     tempBuf.get(buf1.array(), 0, tempDataLen);
33     buf1.limit(tempDataLen);
34     output.addLast(buf1);
35     createHeader(header, i, height, y);
36     zipSplitData(header, output);
37     output.clear();
38
39     buf1 = ByteBuffer.allocate(INFLATE_BUFSIZE);
40     tempBuf.get(buf1.array(), 0, tempBuf.remaining());
41
42     buf1.limit(INFLATE_BUFSIZE - tempDataLen);
43     output.addLast(buf1);
44     temp = INFLATE_BUFSIZE - tempDataLen;
45   }
```

Broadcast と Multicast でどのくらいパケットロスするのかもテストを行った。
Broadcast と Multicast を用いてそれぞれの byte 数を 100packet ずつ送信しパケットロス率を表したのが表 4.6 である。
表 4.6 にあるように Multicast を用いても 4000byte を 100packet 送信すると 37%もパケットロスしてしまう。

結果として、データ分割の処理が重い、且つ予想以上のパケットロス率という結果をになったので、Braodcast や Multicast を用いた実装を行うことはもう少し工夫が必要になることがわかった。

表 4.6: Broadcast と Multicast のテスト

| | 256byte | 4000byte |
|---|---|---|
| Broadcasst | 47% | 87% |
| Multicast | 0% | 37% |

## 4.5　接続先自動検索システムの実装

Broadcast を用いて TreeVNC の Root Node を検索するシステムを実装した。
Listing4.3 は Broadcast を使用して、データを送信するプログラムである。

Listing 4.3: Broadcast を用いてサーバを探すプログラム

```
1  public GetHostClient(String _str) {
2    str = _str;
3  }
4
5  public void sendData() {
6    buf = str.getBytes();
7    DatagramPacket sendPacket = new DatagramPacket(buf, str.length(), mAddr, PORT);
8    try {
9      soc.send(sendPacket);
10   } catch (IOException e) {
11     e.printStackTrace();
12   }
13 }
```

Listing4.3 は、Broadcast Packet を受け取ると、受け取った IP アドレスに対し、Reply-BroadCast() の内部で TCP コネクションを張り、現在起動している VNCServer の一覧を送る。

Listing 4.4: Broadcast を受け取るプログラム

```
1  byte[] buf = new byte[BufSize];
2  byte[] resorve = new byte[BufSize];
3  try {
4    InetAddress mAddr = InetAddress.getByName(McastAddr);
5    MulticastSocket soc = new MulticastSocket(Port);
6    DatagramPacket recvPacket = new DatagramPacket(buf, BufSize);
7    soc.joinGroup(mAddr);
8    while (!stopFlag) {
9      soc.receive(recvPacket);
10     address = getAddress(recvPacket.getSocketAddress());
11     inputStream = new ByteArrayInputStream(recvPacket.getData());
12     inputStream.read(resorve);
13     if(str.equals(castString(resorve)))
14       replyBroadCast();
15     if(stopFlag) break;
16   } catch (IOException e) {
17     e.printStackTrace();
18   }
19 }
```

Listing4.3 は、Root Node から受け取った VNCServer 一覧を表示する部分である。こ

こで使用されている text は、java の GUI コンポーネントである JFrame を継承したクラスのインスタンスである。

Listing 4.5: 起動サーバ一覧を表示するプログラム

```
1  Socket socket = server.accept();
2  is = new BufferedReader(new InputStreamReader(
3  socket.getInputStream()));
4  proxyAddr = is.readLine();
5  if(proxyAddr!=null)
6  text.checkBox(proxyAddr);
7  text.setButton();
8  text.visible();
```

# 第5章　TreeVNCの新機能

この章では、卒業論文の TreeVNC からの大きな変更点について述べる。

## 5.1　UIの拡張

TreeVNC をゼミで使用し、必要な機能や欲しい機能を洗い出し、実装を行った。

### 5.1.1　画面の切り替え

今までは、ゼミで使用する際、発表者が変わる毎にサーバを立ち上げなおさなければならなかったが、現在はボタンひとつで、発表者の画面へと切り替えることが可能になった。

### 5.1.2　画面の拡大縮小

今までは、もし自分の画面より大きい画面が来ていたら、フレームを縦や横にスクロールしなければ画面を見ることができない状態であった。しかし、画面の縮小機能を追加したので、自分の画面にサイズを合わせることでスクロールをしなくても見ることができるようになった。拡大機能も追加したので画面が小さくて見えづらい時などは、この機能を利用して、見やすくすることができる。

## 5.2　マルチディスプレイへの対応

マルチディスプレイを使用すると、すべてのディスプレイ情報が送信されていたが、現在は一つのディスプレイの情報だけを送ることが可能となったので、無駄なデータを省くことができる。これにより、Queue に保存している画像の大きさも小さくなるので、メモリの節約の役割も果たしている。

## 5.3　リファクタリングの容易化

TreeVNC は今までサーバ側と Node 側に同じようなプログラムが 2 つあり、コードをリファクタリングが困難であった。例えば、サーバ側を手直しすると同じ用に Node 側も

手直しが必要な状態であった。更に、パッケージも使用されておらず、どのプログラムがどの役割をしているかわかりづらい状態であった。しかし、現在は機能毎にパッケージが別れており、サーバ側と Node 側も同じ処理をしている部分は一つのコードとなっているので、リファクタリングが前より容易にできるようになった。

# 第6章　画面共有システムTreeVNCの評価

## 6.1　実験環境

TreeVNC は多人数の同時接続を可能にするソフトウェアである。よって、評価を行うためには多数のコンピュータが必要となる。
今回は、学科の並列計算環境と OS の授業の時間を使用させてもらって実験を行った。
使用した並列計算環境を表 6.1 に示す。

表 6.1: 検証に利用する VMWare クラスタの仕様

| 名前 | 概要 |
|---|---|
| CPU | Intel(R) Xeon(R) CPU X5650@2.67GHz |
| 物理コア数 | 2 |
| 論理コア数 | 4 |
| CPU キャッシュ | 12MB |
| Memory | 8GB |
| OS | CentOS 5.8 |
| HyperVisor | VMWare ESXi |

### 6.1.1　CUI Version の作成

TreeVNC は GUI(Graphical User Interface) を使用したアプリケーションである。並列計算環境は CUI(Character User Interface) なので、そのままだと実行することができない。
そこで、TreeVNC から GUI を使用している部分を取り除いて CUI 環境で実行できるようにする必要があったので作成することにした。
Listing3.1 は TreeVNC の Main 文の一部である。ここで継承されている JApplet は、GUI のコンポーネントなので、このクラスを使用すると CUI 環境で実行することができない。

CUI 環境で実行するには JApplet を継承していないクラスを作成する必要があるが、Listing6.2 のような Viewer クラスを受け渡す場所をすべてで CUI と GUI のクラスに対応しなければならない。
この問題に対して、CUI と GUI の共通 Interface(Listing6.3) を作成し、この Interface を利用することで解決した。

Listing 6.1: TreeVNC の MainClass

```
1  public class Viewer extends JApplet implements Runnable, WindowListener
2  {
3    final ConnectionView connectionView = new ConnectionView(Viewer.this,
4                                                    connectionPresenter, hasJsch);
5  }
```

Listing 6.2: Viewer の受け取り

```
1  public SwingViewerWindowFactory(boolean isSeparateFrame, boolean isApplet, Viewer viewer)
2  {
3
4  }
```

Listing 6.3: ViewerImpl

```
1  public interface ViewerImpl
2  {
3    public boolean getCuiVersion();
4    public MyRfbProto getRfb();
5    public void closeApp();
6    public void run();
7  }
```

### 6.1.2　capistrano

今回の実験では、48 台のサーバ上で CUI 版の TreeVNC を立ち上げる必要がある。実験する度に、各サーバに一つ一つにログインしてアプリケーションを立ち上げるのは手間がかかりすぎてしまう。capistrano を使用することで、この問題を解決することができる。capistrano は複数のサーバ上で同時に処理を実行するためのオープンソースなソフトウェアであり、Ruby を用いて作成されている。

capistrano を実行する際に使用するスクリプトを Listing6.4 に示す。
スクリプトは Listing6.5 として実行することができる。

Listing 6.4: cap.rb

```
1 set :user, "mass" 実行するユーザ//
2
3 role :pall, "133.13.62.1" ここに命令を送りたいマシンのアドレス//
4
5 task :ls do // 名 task
6   run "ls␣-la" // 実行したい命令
7 end
```

Listing 6.5: capistrano の実行

```
% cap -f cap.rb ls
```

## 6.2　木の深さによる遅延

TreeVNC は、クライアントを木構造に配置し画像を配信している。
木の深さが深くなってしまうと、データが下に届くまでに遅延が発生してしまう可能性がある。そこで、木の深さによる遅延がどの程度発生するのかを測定してみた。

### 6.2.1　遅延の測定方法

Rfb プロトコルでは、送られてくるデータの先頭にどのような処理をするかの命令番号が入っている。
表 6.2 は送られてくるメッセージの一覧である。
命令番号 11(CheckDelay) はプロトコルを拡張して作成した命令である。

表 6.2: Rfb プロトコルと追加したメッセージ一覧

| 命令番号 | 名前 | 説明 |
|---|---|---|
| 0 | FrameBufferUpdate | 画像の更新情報 |
| 1 | SetColourMapEntries | ピクセルフォーマットで Colour Map を使用 |
| 2 | Bell | ビープ音機能 |
| 11 | CheckDelay | 画像が届くまでの Delay を測定 |

Listing6.6、Listing6.7 は遅延を測るためのプログラムである。Root Node は System.currentTimeMill を用いて時間を取得し、Node へ送信する。
System.currentTimeMillis() は、システムの現在時刻をミリ秒（long 型の数値）で取得する関数である。
Node はこの値を受け取ると、そのままサーバへ受け取った値を返す。
Root Node はクライアントからの返信を受け取ると System.currentTimeMillis() を取り、差分を出して Delay を求める。遅延を測る頻度は、画像を 50 回送信するごとに 1 回である。

Listing 6.6: 遅延を測るプログラム

```
1 BufferedReader is = new BufferedReader(new InputStreamReader(echoSocket.getInputStream()))
      ;
2 DataOutputStream os = new DataOutputStream(echoSocket.getOutputStream());
3 os.writeBytes("checkdelay\n");
4 os.writeBytes(String.valueOf(buf.getLong(16))+"\n");
```

Listing 6.7: 遅延を測るプログラム

```
1  Long delay = System.currentTimeMillis()−Long.parseLong(is.readLine());
```

### 6.2.2　遅延の測定結果

2 分木で木を構成した場合、Node 数が 48 台だと深さが 6 となる。
Root Node を起動し、並列計算環境 48 台を起動し、Root Node から一番下の Node までどのくらいの時間がかるのかを測定した。
6.3 はデータを 20 回ほど測定し平均値を取った遅延の表である。

表 6.3: データ送信の遅延

| 段数 | 遅延 |
|---|---|
| 2 | 約 1 ミリ秒 |
| 4 | 約 58 ミリ秒 |
| 6 | 約 139 ミリ秒 |

並列計算環境の VM は高速なネットワークでつながっているので、遅延が少ない可能性がある。そこで、実際に OS の授業で、どの程度の遅延があるのかを測ってみた。
OS の際は 25Node であったので、段数にすると 5 段である。結果は、5 段目で約 48 ミリ秒と並列計算環境より良い結果が出た。

## 6.3　画面のフリーズ

データが TimeOut によってどの程度損失しているのかを調べてみた。
今回、測定するために画像データのヘッダーの前にシリアルナンバーを付加した (Listing6.8)。これにより Node 側は、順番通りに画像が来なかった場合、データが損失したことを知ることができる。Node 側の確認用コードを Listing6.9 に示す。このコードは Root Node から流れてきたデータを受け取り自分の持っている checkCounter と比較して、違う値が出ていればデータが損失していることになる。

Listing 6.8: データの確認プログラム (Root Node 側)

```
1  ByteBuffer serialNum = ByteBuffer.allocate(8);
2  serialNum.putLong(counter++);
3  serialNum.flip();
4  bufs.addFirst(serialNum);
5  multicastqueue.put(bufs);
```

Listing 6.9: データの確認プログラム (Node 側)

```
1  private void getLost(Reader reader)
2  {
3    try
4    {
5      long num = reader.readInt64();
6      if(num != ++checkCounter) {
7        System.out.println("LostData:"+(num − checkCounter));
8        checkCounter = num;
9      }
10   } catch (TransportException e) {
11     e.printStackTrace();
12   }
13 }
```

現在の実装では、0.625 秒データの読み込みがなければ、データを TimeOut Thread が読み込み、Node 側には順番通りのデータが行かなくなるので画面がフリーズしたように見える。
実験の結果、6 段目の Node でデータを受け取って表示してみた結果、データが損失するのを見ることができなかった。
データがロストしなのは良いことであるが、その分 Root Node のメモリ上にデータがあるので、Root Node が MemoryOverFlow を起こす可能性がある。私のディスプレイ環境 (1920*1080) では、MemoryOverFlow が起こることはなかったが、Retina ディスプレイなどの高解像度ディスプレイを使用している場合は、MemoryOverFlow を起こす可能性がある。TimeOut の時間は今後調整が必要である。

## 6.4　分木の最適化

木の分木数が少ないほうが一つ一つのコンピュータにかかる CPU 負荷が少なくなる。しかし、分木数を少なくしてしまうと、Node が増えたとき、木の深さが深くなり、データの伝搬に遅延が起こる可能性がある。上記の実験の結果、木の深さが 6 のときのデータ伝搬に平均 139 ミリ秒 (0.139 秒) の遅延しか起こっていなく、データの欠損も見られなかった。分木数を変更してもコネクションの数はかわらないし、スイッチに対する負荷も変わらない。よって、100 人程度で使用する場合は 2 分木が最適であるということがわかった。

## 6.5　ZRLE と ZRLEE のデータ圧縮率の比較

作成した TreeVNC では、従来の VNC で使用されているエンコードを使用しておらず、独自で作成している ZRLEE エンコードを使用している。
一見 ZRLE は辞書が一つで ZRLEE は辞書が一つ一つの画像データに付加されていて、データ量は ZRLEE のほうが多くなってしまっている可能性があるので、ZRLEE と ZRLE のデータ量にどの程度の差が出るのかを調べてみた。全く圧縮されていない RAW データ,Zlib 圧縮を使用している ZRLEE、ZRLE のデータ量の比較を行った。図 6 は 1920 * 1080 の画面の全描画にかかるデータ量を測った結果を示した図である。ZRLEE の方がデータ量が少なくですんでいる (図 6.1)。これは、ZRLE(Zlib) が初めに送られた辞書を用いての解凍が余り有効的に働いていない場合があるからだと思われる。つまり VNC の場合は ZRLEE の様に毎回辞書のデータを付加させて送ってもデータ量に差がでない可能性があることが分かった。

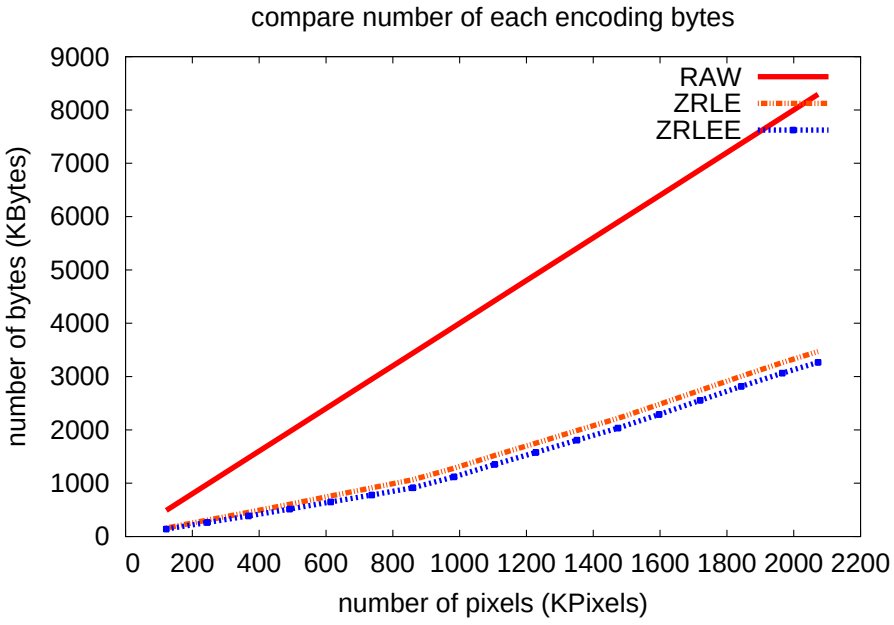


図 6.1: RAW,ZRLE,ZRLEE による 1 画面 (1920*1080) 描画にかかるデータ量。x 軸はピクセル数、y 軸はバイト数を表している。

## 6.6　VNC Reflector との比較

TreeVNC を用いて、Vnc Reflector との比較を行った。並列計算環境の VM48 台を使用し、Blade サーバの外に TreeVNC と Vnc Reflector を起動させた PC を置き、VM48 台にアクセスさせ実験を行った。一極集中型の Vnc Reflector は、48 台繋がった時にスループットが 2MBps から 5KBps へ下がっていた、一方、TreeVNC のスループットは 48 台繋がっている状態でも 2MBps を保つことができていた。一極集中型で繋がっている VNC Reflector と違い、TreeVNC は 48 台のアクセスを複数の通信網へ分散し CPU 負荷を Node 側の分散したいるので、人数が増えてもスループットを落とすことなくアプリケーションを実行することができる。

# 第7章　結論

## 7.1　まとめ

今回の研究では、初めに、プロジェクタを使用している授業の問題点をあげ、解決するために画面共有を使用することを提案した。そこで、画面共有を多人数で使用した際に発生する問題点について考察した。考察から得られた知見を元に、VNC Server への接続者を木構造に接続させ負荷を分散するソフトウェア TreeVNC の設計方法について論じた。実装を行っていく中で必要になる、新しい Encoding タイプ ZRLLE の開発、木の再構成の機能、MulticastQueue 等の機能が必要だとわかったので、これらの設計を行い実装を行った。更に、TreeVNC をゼミで使用した際、画面の切り替え機能、ディスプレイの指定機能などが必要だと感じたので、それらの設計、実装を行った。検証では、学科で用意されている並列計算環境を使用して、木の Root Node から一番下の Node までの画像送信の遅延や TimeOutThread による画像のロスト率の検証を行い、木の分岐数の最適化について考察を行った。それから、実装した画面切り替え UI の妥当性について考察を行った。

## 7.2　今後の課題

### 7.2.1　iPad・無線への対応

Java6 までは、圧縮を行うライブラリ Deflater に flush の機能がなく、TreeVNC では独自の Encoding、ZRLEE を作成する必要があった。しかし、ZRLEE を使用すると、通常の VNC では、データを decode することができないので、Java を実行できない IPad などでは TreeVNC を使用することができない。そこで、Java7 から実装された Deflater の flush 機能を使用して、ZRLEE を ZRLE に変換することで、通常の VNC で decode することができるようになる。

### 7.2.2　Multicast 対応

Broadcast はパケットロス率が高すぎるので使用することは難しいが、Multicast のロス率だと、うまくいく見込がある。Multicast を使用する際は、一回に送るデータ量は 64000Byte である必要がある。私が行ったデータの Blocking は 64000byte 毎に ByteBuffer のコピーを行っていたので、処理が重くなっていた。しかし、コピーせず Buffer を wrap することで、処理が軽くなる。

### 7.2.3　画面範囲の指定

ディスプレイの指定はできるようになったが、現在は画面の範囲指定ができていない。画像データは一旦 Root Node で解凍されるのでその際に画像を加工することで、画面の範囲を指定できるようになる。

# 謝辞

本研究を行うにあたり, ご多忙にも関わらず日頃より多くの助言, ご指導をいただきました河野真治助教授に心より感謝いたします.
研究を行うにあたり, 並列計算環境の調整, 意見, 実装に協力いただいた大城信康さん並列信頼研究室の全てのメンバーに感謝いたします.
最後に, 大学の修士まで支えてくれた家族の方々に深く感謝します.

# 参考文献


[1] Tristan Richardson. The rfb protocol. http://www.realvnc.com/docs/rfbproto.pdf.

[2] TightVNC Software. http://www.tightvnc.com.

[3] 谷成雄, 大城信康, 河野真治. 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, may 2012.

[4] 谷成雄, 大城信康, 河野真治. Java による vnc を用いた授業用画面共有システムの設計と開発. 日本ソフトウェア科学界, sep 2011.

# 発表履歴

- Java による授業向け画面共有システムの設計と実装, 谷成雄, 大城信康 , 河野真治 (琉球大学), オープンソースカンファレンス 2011 Okinawa, Sep, 2011
- Java による授業向け画面共有システムの設計と実装, 谷成雄, 大城信康 , 河野真治 (琉球大学), 日本ソフトウェア科学会第 28 回大会, Sep, 2011
- VNC を用いた授業用画面共有システムの設計・開発,
  谷成雄, 大城信康 , 河野真治 (琉球大学),
  第 121 回 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, May, 2012