# 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

# 目 次

# 図 目 次

# 表 目 次

# 第1章　序論

## 1.1　研究背景と目的

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、授業を受けている生徒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。

VNC を使用すれば、先生のスライドを生徒の手元にあるパソコンに表示することができる。しかし、多人数の生徒が先生のパをコンに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てしまう。この問題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。更に当研究室では、VNC を使用してゼミを進めている。従来の VNC を使用すると発表者が変わるごとに新しく VNC を立ち上げ直す必要がある。このような手間がなくなるとスムーズにゼミを進めることができる。

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。

## 1.2　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

## 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し、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に対して返信を行う。
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。

## 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、画面共有をする為のオープンソースである。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。圧縮された画像データを扱うことができるので、低速回線環境でも動作できるように設計されている。しかしその分 CPU パワーが必要になる。

## 2.3　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がカクカクになり、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。

## 2.4　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。
また、発表者が変わるごとに接続要求認証を毎回行わなければならない。

o

# 第3章　画面共有システムTreeVNCの設計

## 3.1　木構造を用いた設計

序論で述べたように従来のVNCソフトを利用して多人数が同時に接続してしまうと、接続がサーバに集中してしまい (図3.1) ネットワークに負荷がかかり、加えてCPUへの負荷が増加するので、アプリケーションの処理が遅くなったり停止したりしてしまうことが問題である。
そこで、クライアント同士で負荷を分散させることによって、多人数の同時接続を可能にすることができないかと考えた。
負荷分散を行う際、クライアント同士どのようなトポロジを組むのか検討した結果、上から流れてきたデータを下のクライアント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クライアントを木構造に接続させデータを流すためにサーバとクライアントの間にプロキシ (サーバとクライアントの通信を仲介するもの) を設置する方式をとった。プロキシは主にクライアントの管理とサーバ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。

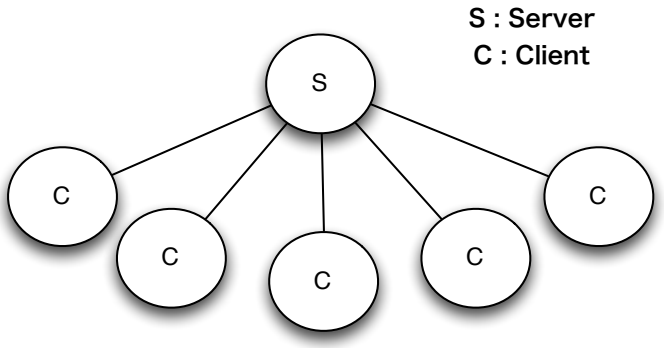


図 3.1: VNCの構造

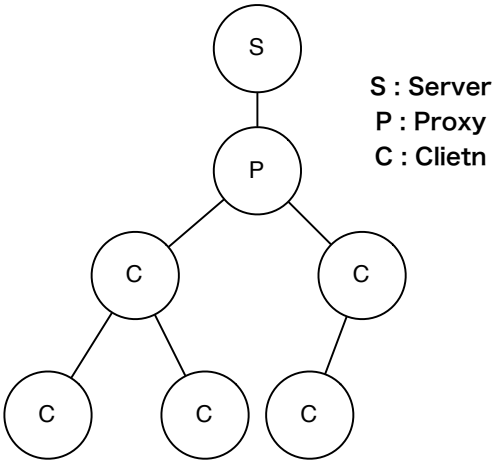


図 3.2: TreeVNC の構造

### 3.1.1　木の生成

負荷を分散させるために木構造を用いるので、クライアントをツリー上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 3.3 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. 初めにクライアントはプロキシに接続先 IP を尋ねる (Where connect)。
2. プロキシは、クライアントに接続するべきホストの IP を引き渡す (Answare)。
3. クライアントは指定されたホストに接続する (Connect)。

### 3.1.2　プロキシの仕事

プロキシの大きな仕事は、クライアントの管理である。クライアントの IP アドレス情報を LinkedList(Java.util.LinkedList) で保持している。
LinkedList の簡単な説明を 3.1 に示す。

表 3.1: 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 はクライアント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クライアントの IP アドレスであり、$(nodeCounter - 1)/treebranch$ で接続する

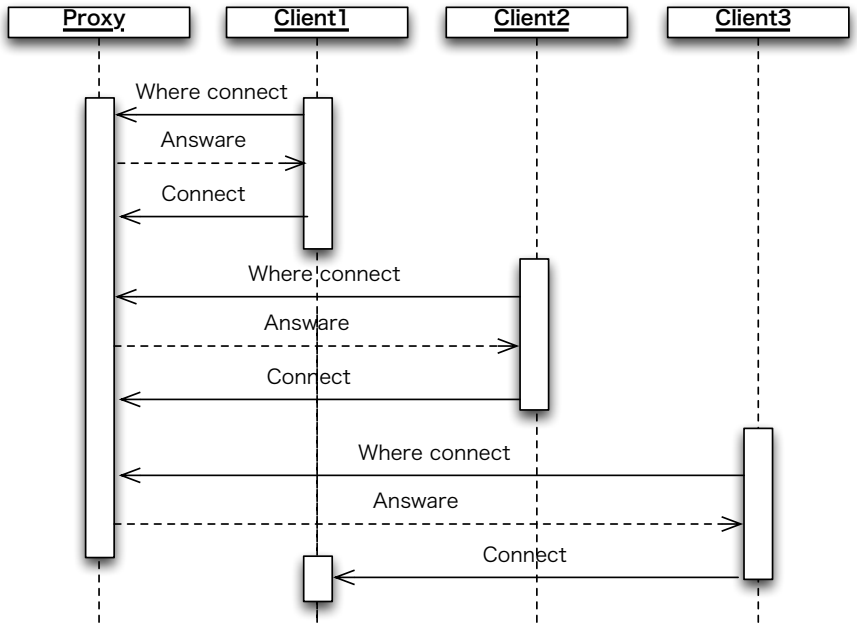


図 3.3: TreeVNC の構造

べき親を求め結果を返している。
treebranch は木の分岐数である。
　親の番号は $(counter - 1)/TreeBraanche$ で求めることができる。

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  }
```

## 3.2　表示画面の切替

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、ユーザ側からプロキシ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。

## 3.3　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。

## 3.4　Broadcast と Multicast

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くなる。そこで、Broadcast を用いた実装について考察してみた。

### 3.4.1　Braodcast パケットの性質

Broacdcast パケットの性質で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。
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。

### 3.4.2　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。
Broadcast では、このような大きなデータを受け取った際は、分割して送らなければならない。私が作成している TreeVNC ではプロキシに送られてきた画像データを一旦解凍し、再圧縮してクライアントに送っているので、解凍した後に分割して、64000byte 以下にして圧縮後送信すると良い。

### 3.4.3　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出来ない。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。

### 3.4.4　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。シリアル番号を降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 コネクションを張り送信を行うのが良い。

## 3.5　再接続

木を構成することはできたが途中のクライアントが落ち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 Top に対して報告する。
2. Top は報告を受けると番号の一番大きいノード (最後のノード) に対して落ちた親の代 わりになるように報告する。
3. Top から命令を受けたクライアントは指定された場所に接続をしなおす。
4. Top はクライアント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. 親が落ちた子供たちは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 3.4 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。

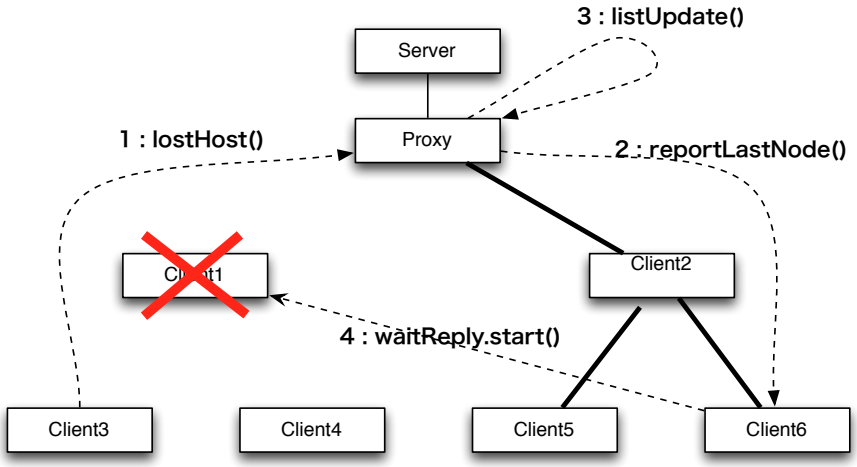


図 3.4: 再接続

1:lostHost() は親が落ちたことを報告する関数である。

2:reportLastNode() は番号の一番大きい (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() はプロキシが持つクライアントのリスト情報をアップデートする (落ち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() はクライアントは 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。もしプロキシからの命令が来るとクライアントはプロキシから指定された場所に接続を行う。

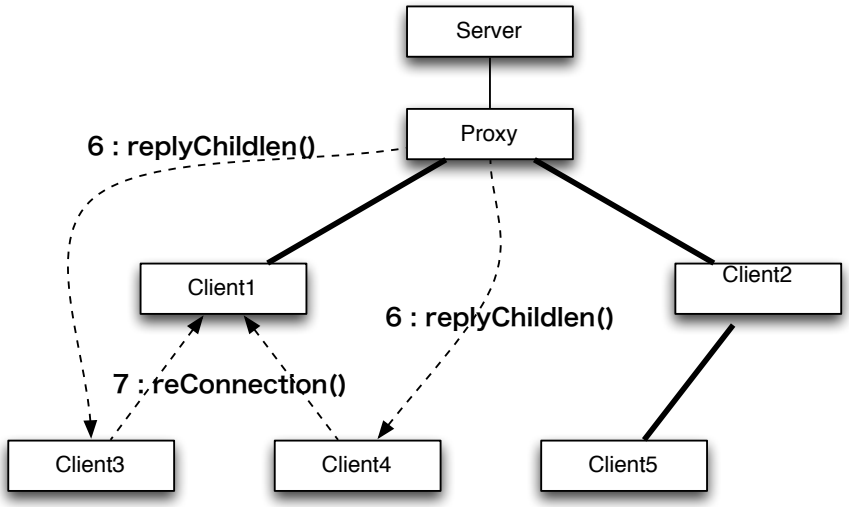


図 3.5: 再接続

6:replyChildlen() は、親が落ち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() はプロキシ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クライアントが落ちても木を再構成することができる。

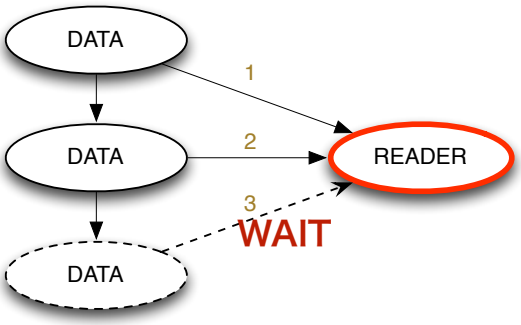


図 3.6: データがなければ wait する

## 3.6 MulticastQueue

画面が更新された際に更新をクライアントに伝えなければならない。ノードが多数ある場合、一人一人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。

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。

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 メソッドを持ちいることで、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 3.6)

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 3.7)

### 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、接続してきたクライアント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 Top の

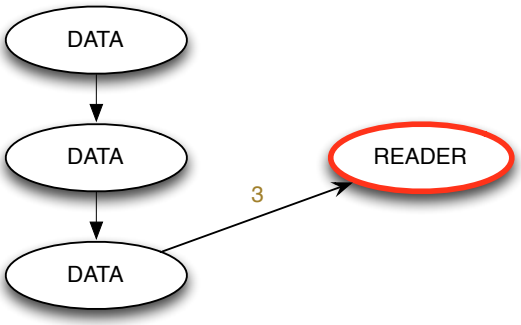


図 3.7: 新しいデータが来るとデータを読み出す

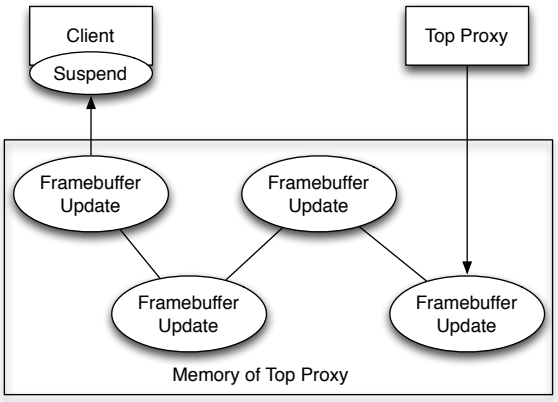


図 3.8: クライアントサスペンド時の Top のメモリの様子。データが残り続けメモリを圧迫してしまう。

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 3.8 に示す。

そこで、ある一定の時間がたつ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クライアントの代わりにデータを取得する。

TimeOut スレッドがクライアント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 Top のメモリを圧迫することがなくなった。(図 3.9)

## 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

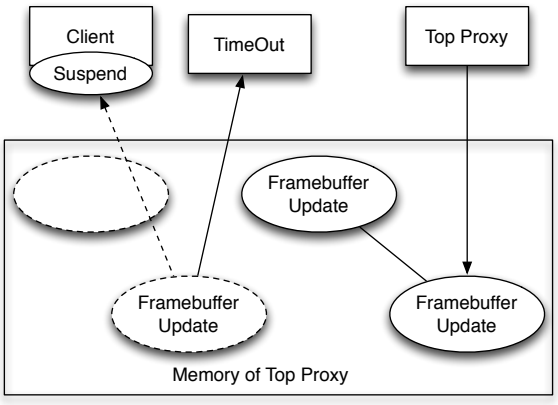


図 3.9: TimeOut が代わりにデータを取得する

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.infla ter 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 3.10)。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 3.11)。

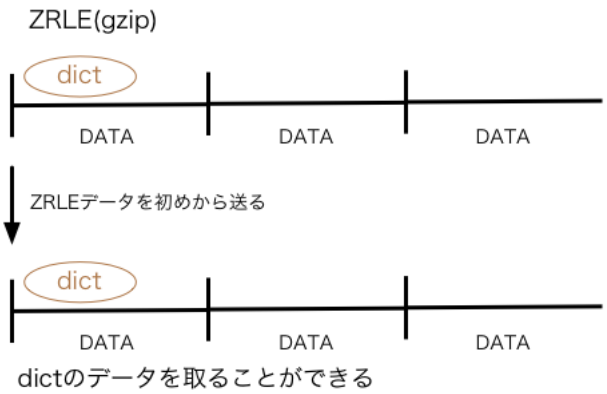


図 3.10: ZRLE

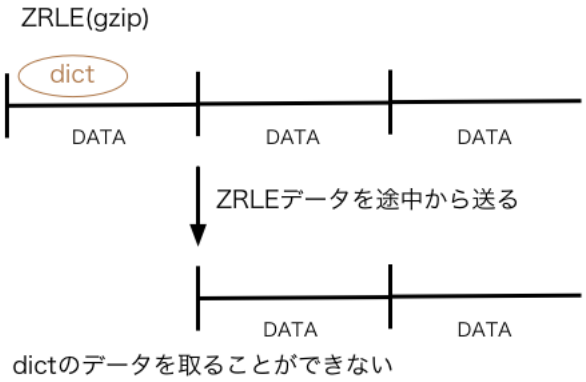


図 3.11: ZRLE2

### 3.7.2　ZRLEE

そこで、Top 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 3.12、図 3.13) 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 Top 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、Top とクライアント側では毎回新しく作る必要がある (クライアント側は inflater だけ)。また、ZRLEE はクライアント側が対応していなければならないという問題がある。

#### ZRLE と ZRLEE のデータ圧縮率の比較

RAW,ZRLE,ZRLEE 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 3.14)。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。

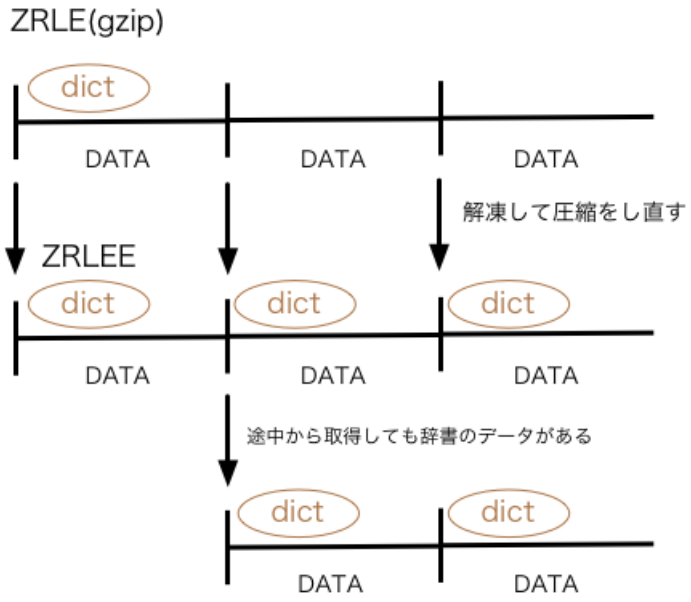


図 3.12: ZRLEE

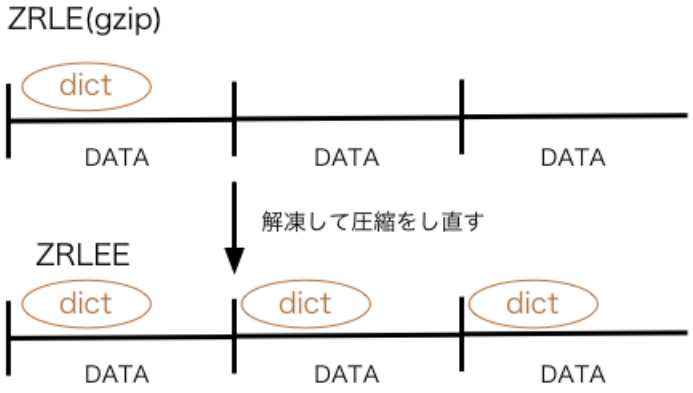


図 3.13: ZRLEE2

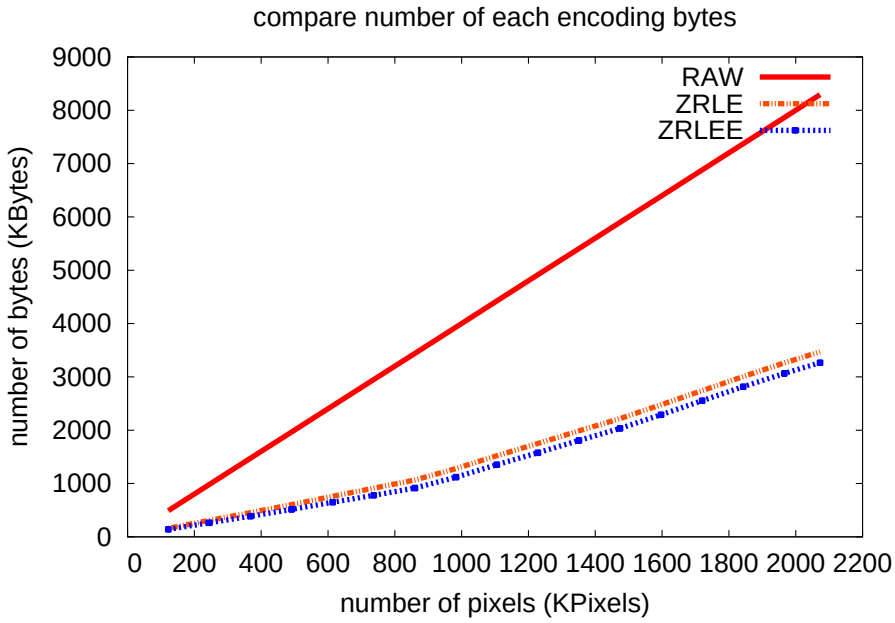


図 3.14: RAW,ZRLE,ZRLEE による 1 画面 (1920*1080) 描画にかかるデータ量。x 軸はピクセル数、y 軸はバイト数を表している。

### 接続先自動検索システム

VNCServer に接続する際、接続先のアドレスを指定する必要がある。アドレスを毎回入力するのは手間がかかるし、間違うと接続ができない。
そこで、クライアントが起動した際に、起動している TreeVNC のサーバを検索し、サーバのアドレス情報を取得す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 サーバを検索する際には 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　TreeVNCの原理

TreeVNC がどのような負荷分散を行って負荷がどこにかかっているのか説明する。
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。そこでクライアント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 4.1) に示す。

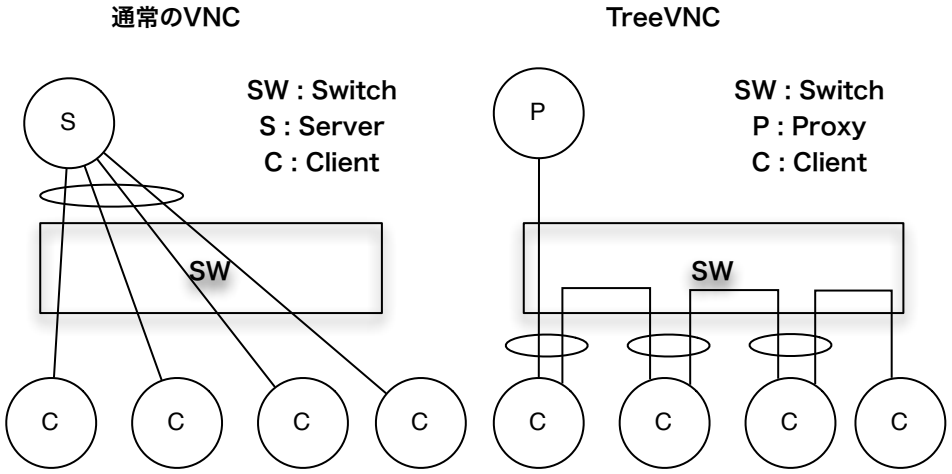


図 4.1: TreeVNC の構造

(表 4.1) 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 の場合はクライアント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、クライアント数に関係なく一定である。
送信する量も通常の VNC の場合クライアント数に比例した量のデータ送信しなければならない。そこで 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 はクライアント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 4.1: ポート一本あたりの通信量 (N はクライアント数、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 |

## 4.2　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。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。このようなアップデートに対応するには、Version2.x 系の TightVNC を手元に持ってきて、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 AbstractFactory パターンを使用して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。

## 4.3　UI の実装

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

### 4.3.1　画面の拡大縮小

### 4.3.2　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 4.2, 表 4.3) に示す。

表 4.2: 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 4.3) に示す。

表 4.3: 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。x-position、y-position、width、height の４つの値で画像の位置と大きさを決めている。
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 4.4)。

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。

表 4.4: ZRLE データ

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

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アップデートの画像更新が来ることがないことがわかった。
図 4.4 の黒い部分が画像データだとすると、図 4.4 のような FramebufferUpdate は送られてくることはない。

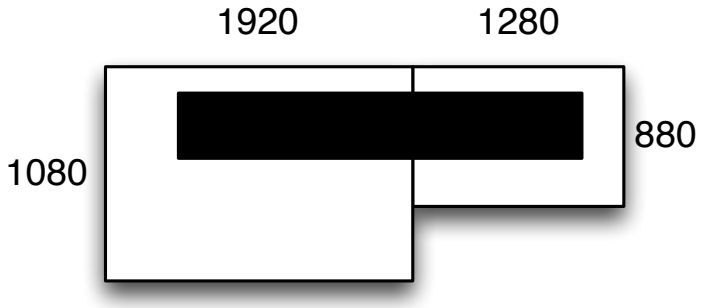


図 4.2: 画面更新時に来る可能性のない UpdateRectangle

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 4.4 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。

### 4.3.3　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 4.3 に示す。

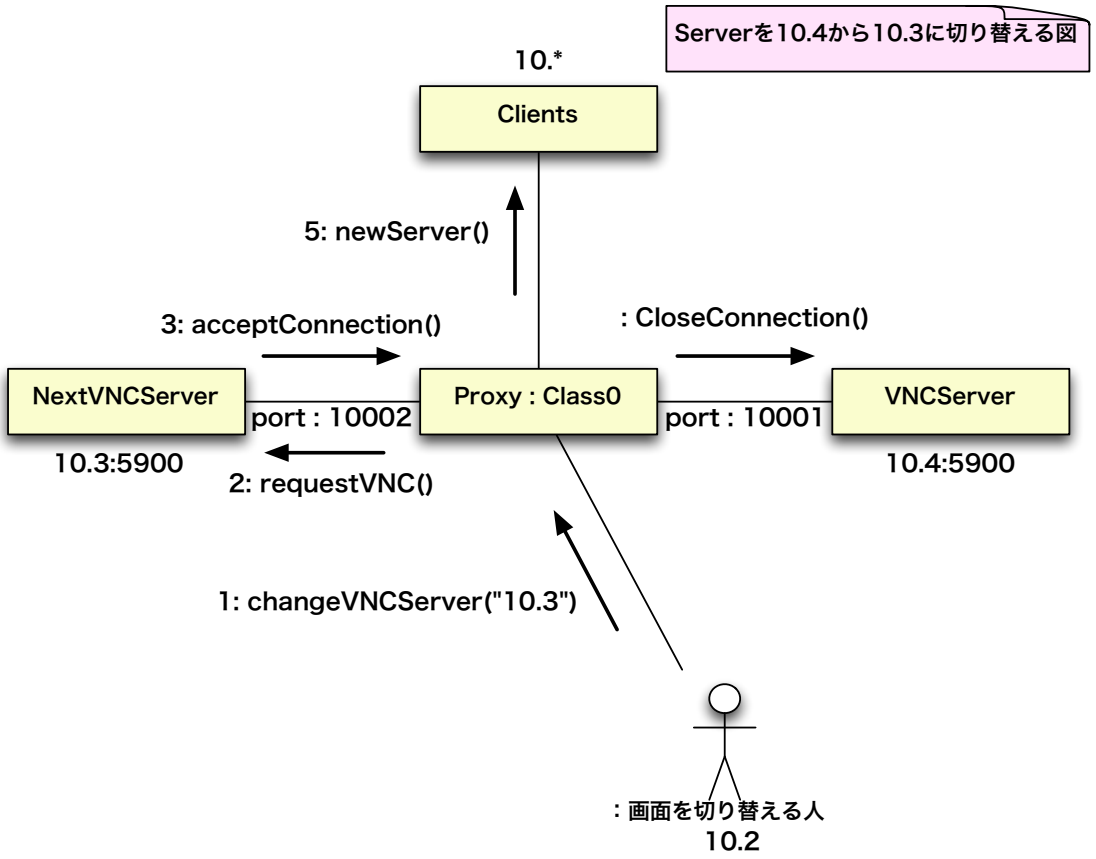


図 4.3: 画面切り替えの CommunicationDiagram

図 4.3 で使用されている関数の説明を表 4.5 に示す。

表 4.5: 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() | プロキシ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プロキシへ報告する関数。 |
| 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() | クライアントに画面が切り替わったことを報告する関数。この命令でクライアント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、プロキシが子供に対して、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。
使用されている clientList は java.util.LinkedList¡String¿であり、中には子供の IP 情報が入っている。

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  }
```

## 4.4 Authentication

プロキシがサーバに対して 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、まず始めにプロキシ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。プロキシ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、サーバ及びクライアントは、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。クライアント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、クライアント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 4.6) のようなものがある。

表 4.6: AuthType

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

MACOSX 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 MACOSX Lion でパスワード認証を行おうとすると、MACOSX 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 MACOSX は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プロキシ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。

## 4.5 Broadcast と Multicast

４章で述べたとおり、Broadcast を使用する場合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プロキシが一旦解凍して、再圧縮し、クライアント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 4.4 に示す。

ひとまずテストで、4.1 のように 2 列づつに分割して送信する書いたてみたが、毎回バッファをコピーして送っているため画面共有に問題が出るくらい処理が遅くなってしまった。その他にも、Broadcast と Multicast でどのくらいパケットロスするのかもテストしてみた。

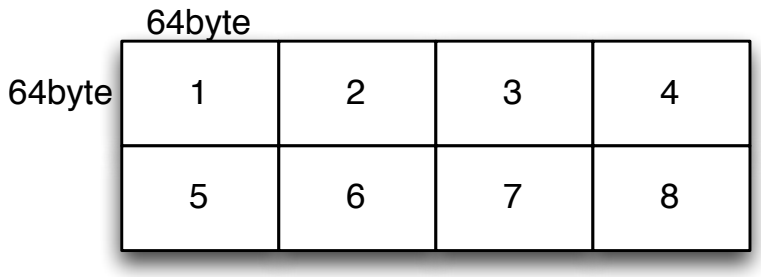


図 4.4: RawData の構造

Broadcast と Multicast を用いてそれぞれの byte 数を 100packet づつ送信しパケットロス率を表したのが表 4.7 である。
表 4.7 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。

表 4.7: Broadcast と Multicast のテスト

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

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。

### 接続先自動検索システム

# 第5章　新しいTreeVNC

## 5.1　リファクタリングの容易化

## 5.2　UIの拡張

### 5.2.1　画面の切り替え

### 5.2.2　画面の拡大縮小

## 5.3　マルチディスプレイへの対応

# 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した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。
使用した計算システムの環境を表 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出来ない。
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    final ConnectionView connectionView = new ConnectionView(Viewer.this, connectionPresenter
         , hasJsch);
3  }
```

Listing 6.2: Viewer の受け取り

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

Listing 6.3: ViewerImpl

```
1 public interface ViewerImpl {
2    public boolean getCuiVersion();
3    public MyRfbProto getRfb();
4    public void closeApp();
5    public void run();
6
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、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であり、Ruby 言語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。
送られてくるメッセージの一覧は表 6.2 がある。
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。サーバは System.currentTimeMillis() を用いて時間を取得し、クライアントへ送信する。
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。
クライアント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。
サーバ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。

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

木の深さはクライアント数を $n$ とすると $\log_2 n$ で求めることができる。

## 6.3　フリーズ

## 6.4　授業とゼミ

## 6.5　分木の最適化

## 6.6　画面切り替え UI の妥当性

## 6.7　再接続の妥当性

# 第7章　結論

## 7.1　まとめ

## 7.2　今後の課題

### 7.2.1　iPad・無線への対応

### 7.2.2　画面範囲の指定

# 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, 谷成雄さん, 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.

[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] Fay Changand Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable : A distributed storege system for structured data.

# 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