平成31年度 卒業論文

分散フレームワークChristieによるリモートエディタ

琉球大学工学部情報工学科

165713F 一木 貴裕

指導教員 河野 真治

目 次

図 目 次

ソースコード目次

第1章　はじめに

ペアプログラミングを行う際に有効的手法の一つとして, 同じファイルを複数人が場所を問わずに同時編集することができるリモートエディタをあげられる.

複数人の編集がリアルタイムに同期されるリモートエディタには, 既存の物として Visual Studio Code(以下 VScode) の remote session がある. しかし, 編集のセッションに参加するユーザ全員が VSCode を使うことになり, また VSCode の環境を導入する必要がある.

そこで, セッションに参加するユーザー全員が各々好きなエディタを使用することができるリモートエディタアプリケーションを作成したい. アプリケーションの形に作成することにより, 開発のための環境に手間を使うことなく, ユーザーが慣れ親しんだエディタを利用できるようしたい. これによりリモートワークやペアプログラミングを, 手軽に行える機能を作る. また, 最終的には VScode のリモートセッションとの接続も目指して制作したい.

先行研究ではネットワークをリング型で構成しトークンを巡回させていたが, ノードごとの整合性の確立が難しい, ネットワーク全体の障害に対する脆弱性の弱さといった問題点が見られた. これらの反省点を踏まえ本研究では スター型ネットワークを用いることで remote editor の障害耐性を高める.

また新しく, 本研究室で開発している分散フレームワーク Christie を使用することにより簡潔な実装と, Christie 自体の性能と信頼性の向上も目指す.

第2章　リモートエディタ

リモートエディタとは他のマシン上に存在するファイルのバッファを別デバイスから開いて編集, 保存することができる機能である.

本研究ではこのリモートエディタを複数人が同時に同じファイルを編集し, その上変更がリアルタイムに反映されるように設計する. この章では同期リモートエディタの実装の上で踏んだプロセスや, 開発の上で問題となる点と解決策について説明する。

2.1　document listener による編集オフセット番号の読み取り

エディタ同士の基本通信環境の構成のため, Chrisitie と同様に java 言語で作成したエディタのインスタンスを使い, 異なるマシン同士の同期の実現を目指した. 自作エディタは java. swing の機能で構成されており, コードをオフセット番号で取り扱っている.

追記または削除されたオフセット位置とその内容の取得は DocumentListenr を使用した. DocumentListener のクラスは swing で実装したエディタ部分の入力と削除を検知し, 動作するメソッドであり, DocumentEvent 内に入力されたオフセットとその長さや文字列が入力されるため, それを Chrisitie 側で検知し処理を行った.

insertUpdate メソッドではバッファに入力が行われた際に自動的に実行され, removeUpdate メソッドは同様にバッファ内の文字のいずれかが削除が行われた際に実行される. 他ノードから送信されてきた命令によるバッファの変更によっても実行され, 意図しないループが発生したため, 受信した命令では実行されないように記述をおこなった.

コード 2.1 は insertUpdate, removeUpdate の記述部分である.

ソースコード 2.1: DocumentListener のコード部分

```
1  public class MyDocumentListener implements DocumentListener {
2      public void insertUpdate(DocumentEvent e) {
3          if(canWrite == true) {
4              Document doc = e.getDocument();
5              loc = e.getOffset();
6              sendLoc = loc;
7              try {
8                  inserted_string = doc.getText(loc, 1);
9                  System.out.println("string = " + doc.getText(loc, 1));
10             } catch (BadLocationException e1) {
11                 e1.printStackTrace();
12             }
13             send = true;
14         }
15         canWrite = true;
16
17     }
18
19     @Override
20     public void removeUpdate(DocumentEvent e) {
21         if(canWrite == true) {
22             Document doc = e.getDocument();
23             sendLoc = e.getOffset();
24             int e_length = e.getLength();
25             endLoc = sendLoc + e_length;
26                 if (e_length == 1) {
27                     System.out.println("delete " + sendLoc);
28                 } else {
29                 System.out.println("delete " + sendLoc + " to " + endLo
       c);
30             }
31             dFrag = true;
32         }
33         canWrite = true;
34     }
35
36     @Override
37     public void changedUpdate(DocumentEvent e) {
38     }
39 }
```

2.2 Command パターンによる命令オブジェクトの作成

リモートエディタを実装する上において, 各エディタは自身に起きたバッファの変更を他ノードに送信する必要がある. この変更の送り合いを Command パターンとして実装した. Command パターンとは, 命令を一つのオブジェクトとして表現する方法である. コマンドパターンの利点として,

- インスタンスを利用して命令を作成するため, 後述の Christie の Gear の概念と相性が良い.
- 命令に必要な内容をまとめて送信するため, 相違の発生を防ぐことができる.
- 命令の管理が行いやすい, 行列に並ばせ命令の順番を管理したり, 命令の際, 実行, 取り消しが容易になる.

といった点が挙げられる. ソースコード:2.2 は書き込み, 送信を行う際の命令をクラスとして作成したものである. このクラスのインスタンスを命令オブジェクトとして送信し合う.

ソースコード 2.2: Command パターンとして実装した命令

```
1  package christie.remoteTextEditor;
2
3  import christie.textEditor.NewTextEditor;
4  import org.msgpack.annotation.Message;
5
6  @Message
7  class Command {
8      public String string;
9      public int fastOffset;
10     public int endOffset;
11     public String nodename;
12     public  boolean isDeleteCommand = false;
13
14     public Command() {}
15
16     public Command(int fastOffset, int endOffset, String nodeName){
17         this.fastOffset = fastOffset;
18         this.endOffset = endOffset;
19         this.nodename = nodeName;
20         this.isDeleteCommand = true;
21     }
22
23     public Command(int fastOffset, String string, String nodename) {
```

```
24          this.string = string;
25          this.fastOffset = fastOffset;
26          this.nodename = nodename;
27      }
28  }
```

2.3 命令オブジェクトを実装する際に起きた問題

インスタンス化した命令を他ノードに送信する際にエラーが発生し, 送信に失敗してしまうという問題が発生した. クラスの送信の際のシリアライズは msgpack クラスを利用している. msgpack クラスは, シリアライズしたいクラスに Message アノテーションをつけることにより, シリアライズ化を行う. 原因を調査した結果, 以下の原因が見つかった.

- Christie の java バージョンは 11 を使用していたが, msgpack バージョン 0.6.12 は java11 に対して対応していなかった.
- msgpack の最新版 0.8.20 はシリアライズ機能が含まれなくなった.

以上の原因に対処するために以下のことを行った.

- Christie の java バージョンを 8 まで下げ, msgpack バージョン 0.6.12 を動作できるようにした.
- シリアライズする命令クラスに対し, フィールドを public にした.
- javassist のバージョンを最新版へ変更した.

java のバージョンを下げたのは応急的な処置となってしまったが, これらの処置により問題なく Command パターンでの命令実装を行うことができた. java のバージョンに左右されずリモートエディタを実装するには, シリアライズの機能について他のパッケージを使うか, 自身で作成する必要が生まれた。

2.4 編集位置の相違

セッション中のエディタ間の通信で生じうる, 編集結果の相違について説明する. エディタ同士のコマンドの送信はそれぞれが独立して行うため, 編集対象の領域にエディタ間で相違が生じる場合がある.

例としてエディタが一対一の接続となっている時に発生しうる相違を図 2.1 を使用して解説する. 編集対象は各オフセット番号に同じ値の数字が入っているものとする. EditorA ではオフセット番号 3 の 3 という 要素を削除 (テキストエディタ上のため削除されたオフセットにはその後ろの要素が繰り上げられる.), EditorB では オフセット番号 2 に A という要素を挿入するという編集をしたとする.

この編集を共通プロトコルとして互いに送信しあった際, 本来編集する予定だったオフセットの中身が異なってしまい編集結果に違いが生じてしまう. これらの問題を解決することのできるエディタ同士の通信手法を作成しなければならない.

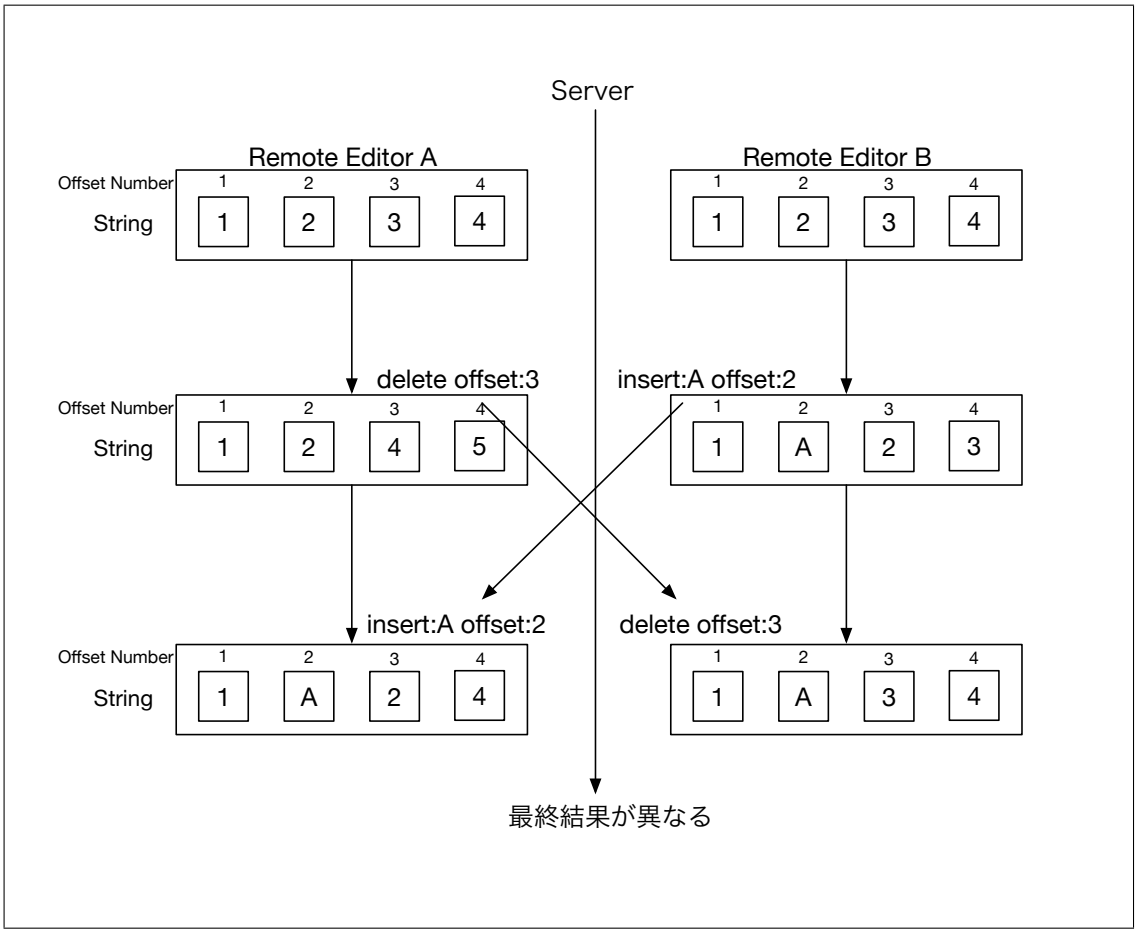

図 2.1: 通信のすれ違いによる編集位置の相違

2.5 編集位置の相違解消方法

編集するオフセットに相違が発生する条件として, サーバーとノードがお互いにコマンドを送り合った際, その命令コマンドが相手に到着する前に相手が自身のバッファに変更を加えてしまった場合に起きる. したがって, 相違の解消に必要なことは

- サーバーとノード間のコマンド送信のすれ違いが発生したということを検知する方法
- すれ違いが発生した際に編集したオフセットのズレを修正する方法

が挙げられる.

そこで, すれ違いが発生したか否かを検知するために、ノード同士が送信し合うコマンドにそれぞれ番号を割り振る方法を考案した. 図 2.2 を用いて解説する.

以下の図はコマンド番号を実装し, すれ違いを検知する際の動作の想定図である. また, 本説明のコマンドは文字列の書き込み命令であり, バッファの指定されたオフセットに insert されるため, それ以降のオフセット内の文字列は, 一つずつ後方オフセットへ移動する.

命令コマンドはクラスのインスタンスを用いて作られており, 中には 1 オフセット分 (文字数分) の文字列, 入力するオフセット, コマンド番号, コマンドを発行したノード名が記録されている.

二つのノード server と node は同じファイルのバッファを開いており, そのバッファには何も記述されていないとする. また, server が送信するコマンドは他のノードから送られてきたコマンドであるとする. 加えて各サーバー, ノードは実行したコマンドをスタック領域に保持しておいている.

コマンドとコマンド番号について以下の特性が存在する.

- 各ノードは最後に実行した数値を変数 (図では server が cNum, node が nodeCNum) に保持している.
- いずれかのノードがコマンドを発行したら, そのコマンドのコマンド番号は前に実行したコマンドの番号+1 となる. そしてそれは送信されてきたコマンドか自分が発行したコマンドであるかは問わない.
- 送信されてきたコマンドのコマンド番号が自身が保持しているコマンド番号+1 でなければ、自身が先に発進したコマンドとすれ違いが発生していることが判明できる. (保持コマンド番号より同値以下の場合)

つまり, 送信されたコマンド No.102 を実行したらそれぞれのノードは実行済みコマンド番号を 102 へ書き換えなければならない. そして続いて自らがバッファに変更を加えたとき, その変更をコマンド No103 として作成し, 送信と実行済みコマンド番号を 103 に書き換える処理が必要となる.

以上の処理でコマンド送信のすれ違いを検知する.

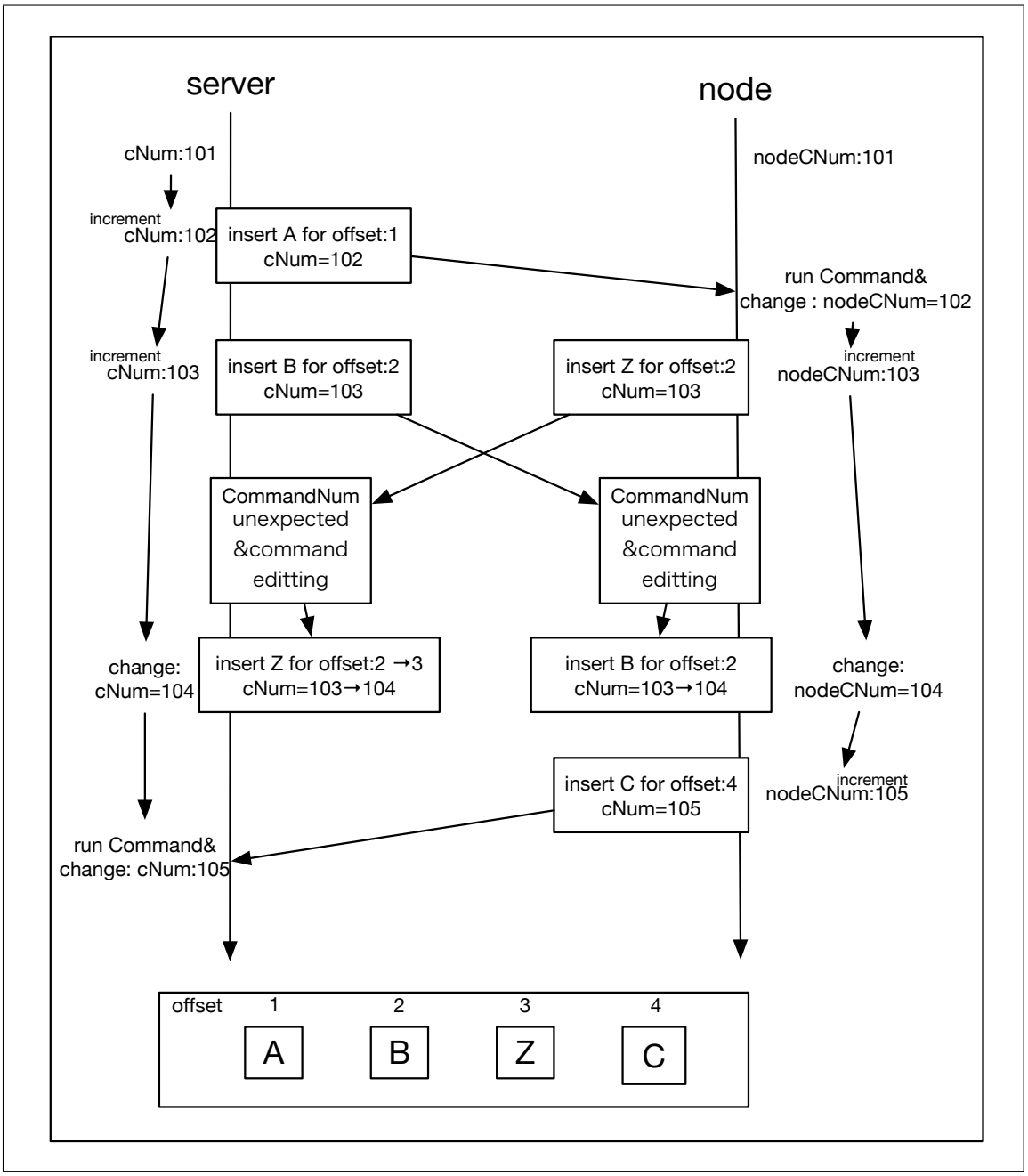

図 2.2: コマンド番号を利用した相違の解消

コマンドのすれ違いを検知した際の処理は以下のように行う.

- 前提としてノードよりサーバーの整合性を維持したいため優先度は常にサーバーが高い. 例えば同じオフセットにそれぞれのノードが別の文字列に変更した場合, サーバーノード共にサーバー側が発進したコマンドの文字列が書き込まれる.
- すれ違いが発生したら, 各ノードはコマンドを記録しているスタックを参照し, 受け取ったコマンドのオフセットのズレを集計しそのコマンドを修正する.

図上では server からの insert B for offset:2 と node からの insert Z for offset:2 の命令がすれ違いを起こしている. この場合, server のバッファ状態が優先されるため, server 側が node からの命令の offset 位置を調整してやらなければならない. この処理は server と全ての接続しているノード間で独立して行われる.

コマンドのオフセットの修正は

- 受け取ったコマンドのオフセット $>$ 受信コマンドとすれ違ったコマンドのオフセット のとき, 受診したコマンドのオフセットに+1 する.
- 受け取ったコマンドのオフセット $<$ 受信コマンドとすれ違ったコマンドのオフセット のとき, 受診したコマンドのオフセットを変えずにそのまま実行できる.
- 受け取ったコマンドのオフセット $=$ 受信コマンドとすれ違ったコマンドのオフセット のとき, サーバー側の文字列が優先される.

と言った方法で行うことができる. これを削除, 文字列の置き換えにもそれぞれ対応する方法を作ることによりズレの修正を行う.

また, 上記の修正方法のテストコードを試みた. テストコードを書き上げる中でこの方法でも網羅できていない問題点や実装上の問題がみられた.

- 既存のエディタ (emacs や vim) のバッファ処理はオフセット単位で行われていない.
- コマンドの put(送信) 失敗についての想定ができていない.
- 現在の Christie では リスト型や Stack 型が put できない.
- コマンドを保持し続けるとメモリ容量に無駄が発生する.

と言った問題点が露見した. これらの問題についての対応方法を模索する必要がある. また実装した際に初めて判明する通信速度や複数の独立した処理による影響も対処しなければならない可能性がある.

第3章　スター型接続によるネットワーク通信

リモートエディタのセッションに参加するノード (ユーザ) はスター型で接続を行い, リモートエディタの通信部分の障害に対する耐性を保障する.

スター型とは中心となるノードから放射状に他のノードにそれぞれ一対一の接続を行う接続であり, 図 3.1 はスター型接続をグラフ化した物である.

図で説明すると, node0 がハブノード (サーバーの役割) として他の node1, 2, 3, 4 と接続する. 例えばここに新しく node5 が接続に加わると仮定すると, 他のノードと同様に node0 と接続するのみでセッションに参加ができる..

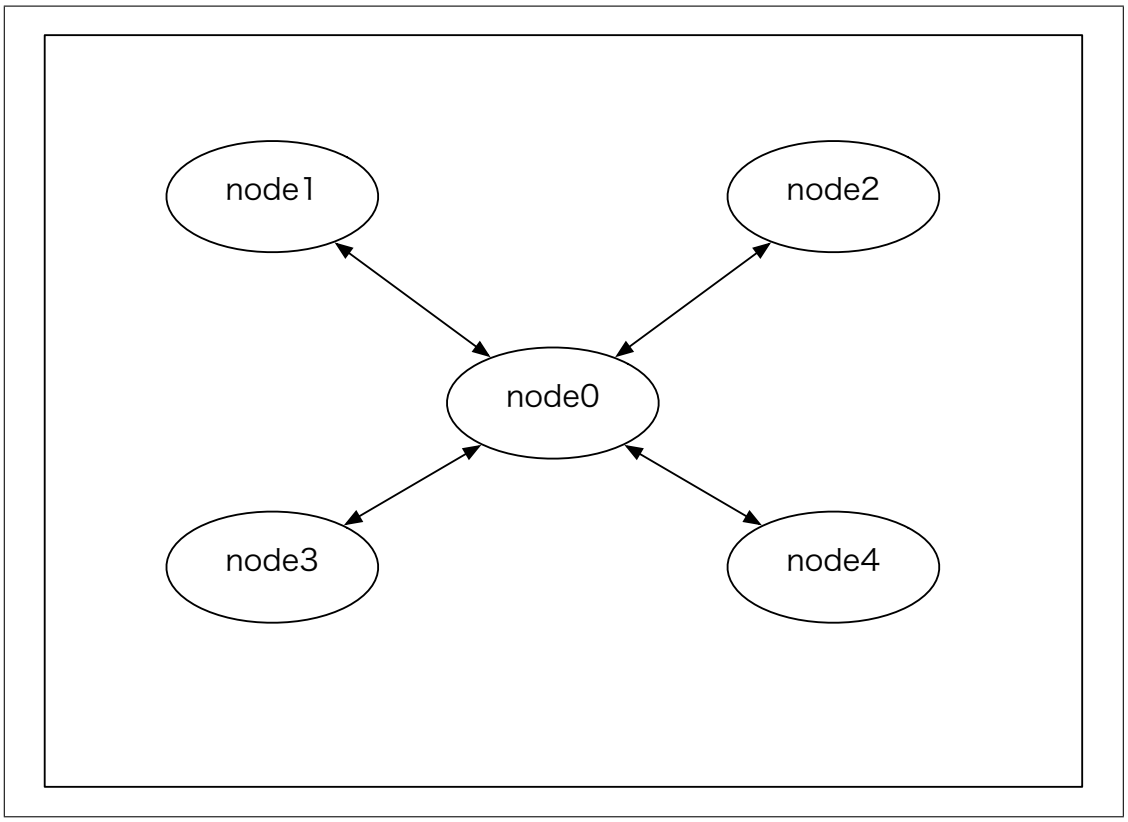

図 3.1: スター型の接続をグラフ化した物

3.1 スター型の利点と比較

先行研究においてはノードの通信をリング型, つまりノード同士を円となる形で接続し, そこに巡回トークンを巡らせコマンドを回収することで実装を試みていた. しかし, リング型には以下の欠点が見られた.

- ノードごとのもつファイルの整合性の維持が難しい.
- どこかのノード同士の通信が切断された際の再接続が難しく, また障害が全体に影響してしまう.
- 障害からの復帰が難しい.

リング型と比較した際のスター型の利点として,

- ノードの中心 (サーバー) が正しいファイル状況を保持するため, 整合性を保つことが容易である.
- どこかのノードの接続が切断されても, 障害の範囲をそのノードのみに抑えることができる.
- 新しいノードが参加した, もしくはノードの再接続の際にはサーバーのファイル状況を参照するのみで参加, 復帰ができる.

と言ったことが挙げられる.

TopologyManager の接続相手にラベルをつける機能により, サーバーでは各 node すべてをまとめて一つの名前で処理をすることができる. 反対に各ノードもラベルを利用することで, CG 内に大きな工夫をつけることなくサーバーとの通信を行うことができる.

懸念点として

- 通信がサーバーのみに集中するため, それを原因に遅延が発生する可能性がある.
- サーバーと他ノードとの一対複数という通信形式から発生する, 予期せぬ編集誤差の危険性.

と言った点が挙げられる. これらの発生を防ぐため,

- 送信するデータ量や頻度を減らす工夫などを凝らし, 通信の負荷がなるべく少ない設計を構築する.
- サーバーを中心とした整合性維持のための設計をする.

と言った対策が考えられる.

第4章　分散フレームワークChrisiteについて

ここでは本研究室で開発している分散フレームワークChrisiteについて解説する．Chrisiteは複雑な分散プログラムを簡潔に書くことのできる構成となっている．

4.1　Chrisiteとは

Christieは Java 言語で構成された本研究室独自の分散フレームワークである．同じく本研究室で開発を行っているGearsOSのファイルシステムに組み込む予定があるため，GearsOSを構成している本研究室の独自の言語Continuation based C (以下CbC言語) とにた，Gearというプログラム概念が存在する．

- CodeGear(以下 CG)
- DataGear(以下 DG)
- CodeGearManager(以下 CGM)
- DataGearManager(以下 DGM)

CodeGearはクラスやスレッドに相当する．DataGear は変数データに相当し，javaのアノテーション機能を用いて記述する．CG内に記述したKeyに全てのDGが揃った際に初めてそのCGが動作するという仕組みになっている．CodeGearManager はいわゆるノードに相当し，CG, DG ,DGM を管理する．DataGearManager は DG を管理するもので，put という操作により DG , つまり変数データを格納することができる．

DGM の put 操作を行う際には Local と Remote と 2 つのどちらかを選び，変数の key とデータを引数に書く．Local であれば，Local の CGM が管理している DGM に対し，DG を格納していく．Remote であれば接続した Remote 先の CGM の DGM に DG を格納できる．put 操作を行ったあとは，対象の DGM の中に queue として保管される．

DG を取り出す際には，CG 内で宣言した変数データにアノテーションをつける必要がある．javaのアノテーションとは注釈，注記を意味し，java.lang.Annotation インターフェースを継承して独自のアノテーションを作成できる．DG のアノテーションには Take, Peek, TakeFrom, PeekFrom の 4 つがある．

Take 先頭の DG を読み込み, その DG を削除する. DG が複数ある場合, この動作を用いる.

Peek 先頭の DG を読み込むが, DG が削除されない. そのため, 特に操作をしない場合は同じデータを参照し続ける.

TakeFrom(Remote DGM name) Take と同様に読み込んだ後, DG を削除する. Remote DGM name を指定することで, その接続先 (Remote) の DGM から Take 操作を行える.

PeekFrom(Remote DGM name) Peek と同様に読み込み後も DG が削除されないが, Remote DGM name を指定することで, その接続先 (Remote) の DGM から Peek 操作を行える.

4.2 プログラムの例

以下のソースコード 4.1 , 4.2 のプログラムは Chrisitie の基本動作となる DGM による put 操作を用いた hello world の出力プログラムである.

メソッド sreateCGM でポート番号を指定した上で CGM を作成しする. CGM に CG (クラスファイル) を指定した上で setup することで CGM が CG を動作させることができる. HelloWorldCodeGear() と FinishHelloWorld() がここでは CG に当たる. HelloWorldCodeGear() クラスには String 型の”helloWorld” という key が用意され, ”helloWorld”に入力された DG(String 型の変数データ) を print するコードである. 当然 key:helloWorld には String 型しか当てはめられない.

”helloWorld” に hello と world という DG を put することで出力結果が hello world となる. また helloWorld の key はアノテーションが Take である. 従って, 一度目に hello を put し, hello を print した後, helloWorld の key の中身はなくなるため, 二度目の CG:HelloWorldCodeGear を setup した際は world を問題なく key に put できる.

もし helloWorld の key が Peek アノテーションがついていた場合, 一度目の put にて入力された hello が key:helloWorld に残り続けるため, CG:HelloWorldCodeGear を setup するたびに CG が動作し, 以下のコード 4.2 では hello を print し続ける無限ループが起こってしまう.

ソースコード 4.1: StartHelloWorld

```
1  package christie.example.HelloWorld;
2
3  import christie.codegear.CodeGearManager;
4  import christie.codegear.StartCodeGear;
5
6  public class StartHelloWorld extends StartCodeGear {
7
8      public StartHelloWorld(CodeGearManager cgm) {
9          super(cgm);
10     }
11
12     public static void main(String[] args){
13         CodeGearManager cgm = createCGM(10000);
14         cgm.setup(new HelloWorldCodeGear());
15         cgm.setup(new FinishHelloWorld());
16         cgm.getLocalDGM().put("helloWorld","hello");
17         cgm.getLocalDGM().put("helloWorld","world");
18     }
19 }
```

ソースコード 4.2: HelloWorldCodeGear

```
1  package christie.example.HelloWorld;
2
3  import christie.annotation.Peek;
4  import christie.annotation.Take;
5  import christie.codegear.CodeGear;
6  import christie.codegear.CodeGearManager;
7
8  public class HelloWorldCodeGear extends CodeGear {
9
10     @Take
11     String helloWorld;
12     @Override
13     protected void run(CodeGearManager cgm) {
14         System.out.print(helloWorld + " ");
15         cgm.setup(new HelloWorldCodeGear());
16         cgm.getLocalDGM().put(helloWorld,helloWorld);
17     }
18 }
```

CGM は起動し続けていると処理が自動的に終了しないという問題点がある. そこで役目がなくなった CGM を終了させるための処理を行わなければならない.

CGM を終了させるためのプログラムはソースコード 4.3 であり, 二つの key が揃ったら cgm.getLocalDGM().finish() の処理で cgm を終了させるよう記述されている.

ソースコード 4.3: FinishHelloWorld

```
1  package christie.example.HelloWorld;
2
3  import christie.annotation.Take;
4  import christie.codegear.CodeGear;
5  import christie.codegear.CodeGearManager;
6
7  public class FinishHelloWorld extends CodeGear {
8      @Take String hello;
9      @Take String world;
10
11     @Override
12     protected void run(CodeGearManager cgm) {
13         cgm.getLocalDGM().finish();
14     }
15 }
```

4.3 TopologyManager について

ここでは Chrstie 上でノード同士の接続をより簡潔にするために使われる TopologyManager という機能について説明する.

TopologyManager とは Topology を形成するために, 参加を表明したノード, TopologyNode に label を与え, 必要があればノード同士の配線も自動で行う機能である. TopologyManager の Topology の形成方法として静的 Topology と動的 Topology の二つの方法がある. 静的 Topology はソースコード: 4.4 のような dot ファイルを与えることでノードの接続を図 4.1 のように接続することができる. 例えば node0 からは node1 は right という名前で参照することができ, それぞれのノードが同じ CG を実行しても label の与え方次第で想定した DG の差し合いを実現することができる.

静的 Topology は dot ファイルのノード数と同等の TopologyNode があって初めて, CodeGear が実行され, ノード数が合わないとエラーが表示される.

ソースコード 4.4: ring を構成する dot ファイル

```
digraph test {
    node0 -> node1 [label="right"]
    node1 -> node2 [label="right"]
    node2 -> node0 [label="right"]
}
```

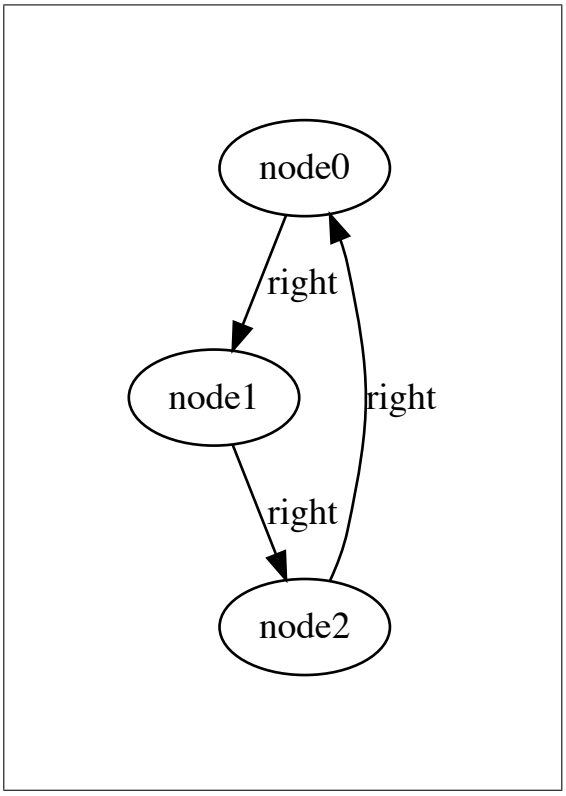

図 4.1: ソースコード 4.4 の dot ファイルを図示化したもの

動的 Topology は参加を表明したノードを順番に Topology の構成要素として接続していく物である. 現在時点で実装済みの Tree の構成を例とすると

1. 参加したノードを順に root(根) に近い要素として接続する.
2. Topology の要素に構成されたノードはそれぞれ親, 子のノードを特定の名前 (parent, child[n]) で参照できる.
3. 途中参加したノードは, 木の末端要素として接続する.

以上の形で Topology が形成される.

コード:4.5 は TopologyManager を使用して Topology を構成するコードである. String 型のリスト (今回は managerArg) に構成したい Topology の形状を dot ファイル, もしくは実装済みの動的 Topology の構成型を設定し, TopologyManagerCondfig を起動することで TopologyManager が起動できる.

ソースコードでは Tree を構成しており, for 文で nodeNum 個分のノードを生成し, それぞれ managerPort を記憶させている. これによりノードすべてが TopologyManager により Tree の構成要素として接続される.

現状では通信アルゴリズムの構成のため, dot ファイルにより接続を行なっているが, 最終的には Star 型の動的 Topology 機能を作成し, 途中で参加してきたノードを接続が行えるようにする必要がある.

ソースコード 4.5: TopologyManager による Tree 型 Topology を構成するコード

```
1  package christie.example.PrefixTree;
2
3  import christie.codegear.CodeGearManager;
4  import christie.codegear.StartCodeGear;
5  import christie.topology.manager.StartTopologyManager;
6  import christie.topology.manager.TopologyManagerConfig;
7  import christie.topology.node.StartTopologyNode;
8  import christie.topology.node.TopologyNodeConfig;
9
10 public class StartPrefixTree extends StartCodeGear {
11
12
13     public StartPrefixTree(CodeGearManager cgm) {
14         super(cgm);
15     }
16
17     public static void main(String[] args) {
18         int topologyManagerPort = 10000;
19         int topologyNodePort = 10001;
20         int nodeNum = 8;
21         String[] managerArg = {"--localPort", String.valueOf(topologyMana
       gerPort), "--Topology", "tree"};
22         TopologyManagerConfig topologyManagerConfig = new TopologyManager
       Config(managerArg);
23         new StartTopologyManager(topologyManagerConfig);
24
25         for (int i = 0; i < nodeNum ; i++){
26             String[] nodeArg = {
27                     "--managerPort", String.valueOf(topologyManagerPort),
28                     "--managerHost", "localhost",
```

```
                    "--localPort", String.valueOf(topologyNodePort + i),
                    "--totalNodeNum", String.valueOf(nodeNum),
                    "--i", String.valueOf(i)};

            PrefixNode.main(nodeArg);


        }
    }
}
```

第5章　今後の課題

ここではリモートエディタの実装において今後開発, 修正しなければならないことについて解説する.

5.1　既存エディターに対する編集方法

ユーザーが自身の好みなエディタを選択し、リモートセッションが行えるためには各種類のエディタのプロトコルをリモートエディタに対応させなければならない. まずは emacs 続いては vim の実装を予定している.

ただし, emacs や vim はバッファの構成が java による自作エディタとは異なり, オフセットによる管理を行なっていないため, 対応させる方法を模索する必要がある. 加えて, emacs にリモートエディタを対応させる際には emacs-lisp を用いる必要があることが予測される. java 言語で構成された Christie から emacs の操作をするまでの処置の方法も模索しなければならない.

5.2　編集するファイルの共有方法

現段階では編集位置とその文字列, もしくは削除されたかどうかという情報の送り合いしか実装しておらず, 編集対象のファイルの共有が行えていない.

ファイルの共有方法としてファイルの中身をそのまま送信すると言った方法が考えられるが, ファイル要領や通信への負担といった要因を考えると最適な手段とは言えない. そのためユーザが編集するファイルの一部部分のみ送信するといった方法を考案する必要がある.

5.3 動的なStar型Topologyの構成機能

現開発段階では, 編集位置の相違の解消方法の設計のため, Star 型の接続を dot ファイルを用いて静的に行っている. 先述したが静的 Topology の構成では参加ノードの数が想定と一致しなければ動作しないという問題点がある.

作成するリモートエディタは不特定数のユーザの参加を前提としているため, 動的に Star 型の Topology を構成する機能を作成する.

また, リモートエディタのセッションでは, セッション開始者とは別にサーバーを立て, そのサーバーに開始者を含めた他のユーザを接続する予定である.

5.4 複数のマシンがセッションに参加した際の動作

現在は一つのマシン上にポートを複数立て, 実際の動作を確認している. これを実際に複数のマシンからセッションを参加した際の通信上でどのような問題や利便性の低下が起きるかが確認できていない.

また, セッション参加者にも上限が存在することが予測される.

第6章　まとめ

本研究ではリモートエディタを制作する上で必要となる構造と, 土台となる分散フレームワーク Christie について述べた. 現時点ではリモートエディタのコマンドを送り合う上での基盤となるコマンドパターンでの送受信を実装し, 命令コマンドのやり取りにて問題となってくる編集の相違の発生をテスト作成することまでができた.

しかしまだ, 第 5 章で述べたとおりリモートエディタを動かすまでの最低限の構造制作までは終わっていない. 現時点から最低限のセッションができるようになるためには, 複数人が参加できる Star 型 Topology を構成すると編集ファイルの共有方法を作成しなければならない. 制作の途上で判明してくるであろう問題点についても検討しなければならない. 加えて, 本研究の制作物が実用的になるには既存のエディタを動作できるようになることは絶対条件である.

これらを踏まえて, 今後も制作を続け, 本研究で述べた構成が実際に実用性に応えられるか, ユーザに苦痛なく使ってもらえる処理速度を得られるか検証していきたい.

参考文献

[1] 赤堀 貴一. Christie によるブロックチェーンの実装, Merch 2019

[2] 河野 真治. 分散フレームワーク Christie と分散木構造データベース Jungle, IPSJ SIG Technical Report, May 2018.

[3] 照屋のぞみ. 分散フレームワーク Christie の設計, Master ' s thesis, 琉球大学 大学院理工学研究科, 2018.

[4] 安村恭一. 巡回トークンを用いた複数人テキスト編集とセッション管理, Merch 2004

[5] Kaito TOKKMORI and Shinji KONO. Implementing continuation based language in llvm and clang. LOLA 2015, July 2015.

[6] 宮城健太. Remote editing protocol の実装, Merch 2008.

[7] 結城浩. デザインパターン入門, 2004.

謝辞

本研究を行う上で, 多くの教鞭と助言を頂きました河野真治准教授, 並びに琉球大学工学部知能情報コース教員, 職員の方々に心より感謝申し上げます.

また, 研究進行の上で快くお手伝いや助言を行なってくださった並列信頼研配属の全てのメンバー, 忙しい中研究ツールについての伝授を行ってくれた赤堀 貴一先輩, Christie を作成していただいた照屋のぞみ先輩に感謝いたします.

最後に地元静岡より支援と見守り続けてくれた両親に感謝いたします.