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

複数人がリアルタイムにファイルの操作を行うことができるリモートエディタには, 既存の物として Visual Studio Code 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買いはすることにした. アプリケーションの形に作成することにより, 手軽に同時編集が行えるようにしたい. 先行研究 [1] 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を目指す.

第2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．

2.1　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しする．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 , 要するに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．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.

2.2 プログラムの例

以下のソースコード 2.1 , 2.2 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 setCGM でポート番号を指定した上で CGM を作成しする. 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 を print するコードである. ”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. CGM が起動していると処理が終了しないため, FinishHelloWorld には DG が揃い次第 cgm.getLocalDGM().finish(); というメソッドを実行させ cgm を終了させる処理が書かれている。

ソースコード 2.1: 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 2.2: 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 }
```

2.3 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: 2.3のようなdotファイルを与えることでノードの接続を図 2.1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 2.3: ringを構成するdotファイル

```
1  digraph test {
2      node0 -> node1 [label="right"]
3      node1 -> node2 [label="right"]
4      node2 -> node0 [label="right"]
5  }
```

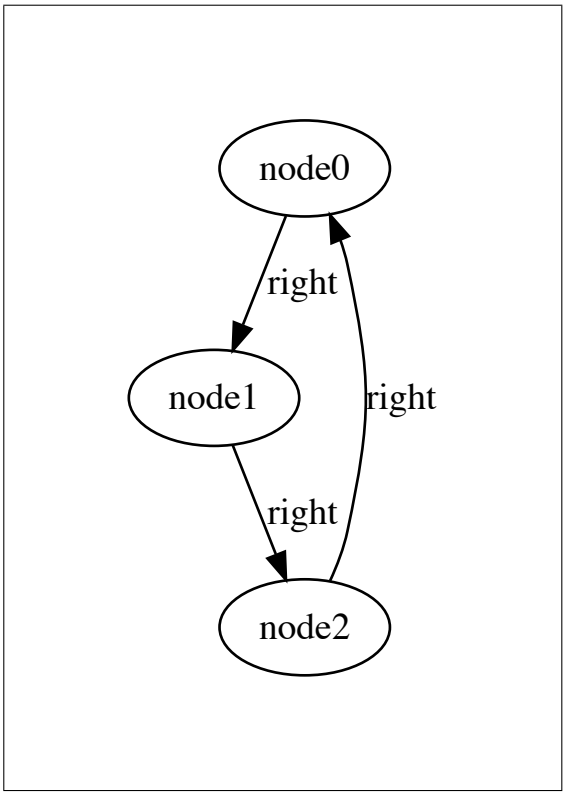

図 2.1: ソースコード 2.3 の dot ファイルを図示化したもの

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:2.4 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.

ソースコード 2.4: 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",
29                     "--localPort", String.valueOf(topologyNodePort + i),
30                     "--totalNodeNum", String.valueOf(nodeNum),
31                     "--i", String.valueOf(i)};
32
33             PrefixNode.main(nodeArg);
34
35
36         }
37     }
38 }
```

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.

第3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. 加えて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。

3.1　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 3.1 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食い違ってしまい最終的に異なった内容となってしまう. 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アルゴリズムを作成しなければならない。

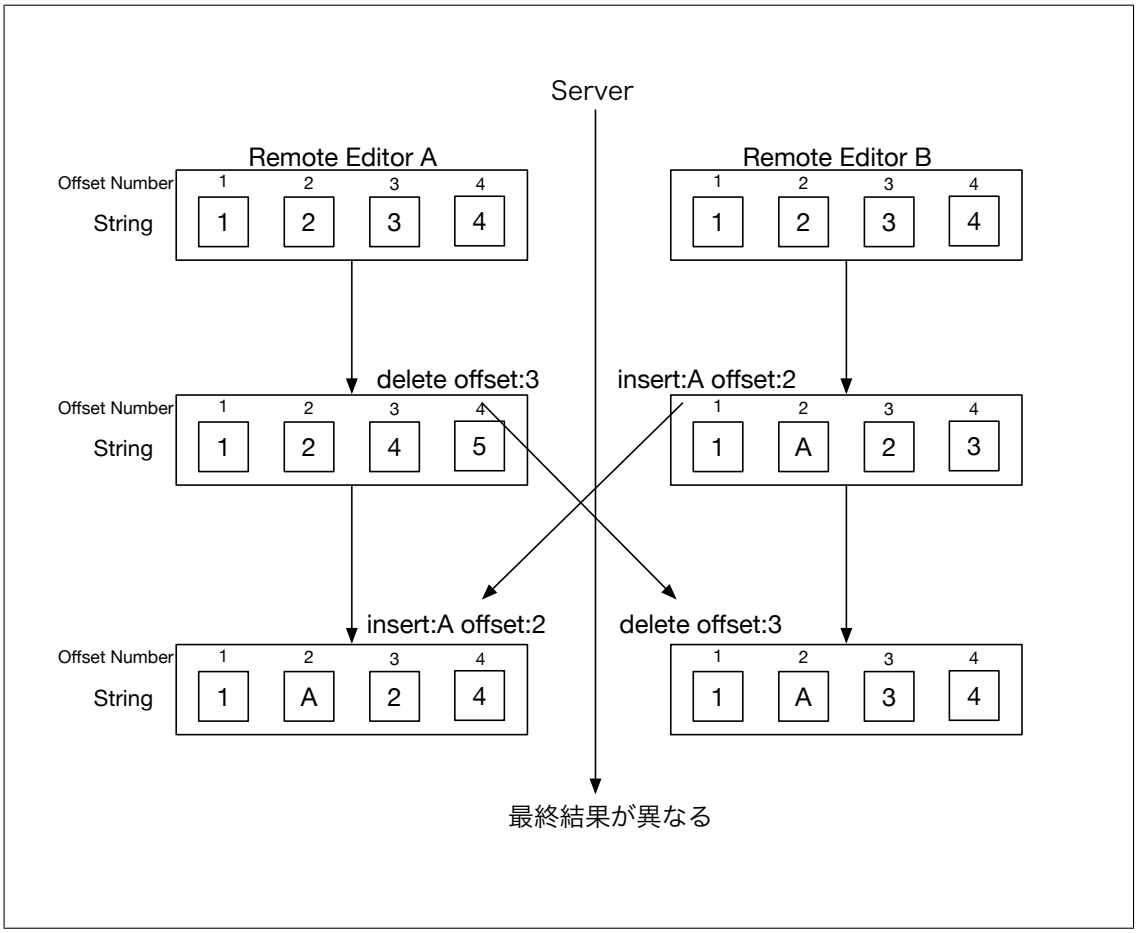

図 3.1: 通信のすれ違いによる編集位置の相違

3.2 編集位置の相違解消方法

3.3 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 と同様の 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, 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.

ソースコード 3.1: 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 }
```

3.4 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対応した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, 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:3.2 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 3.2: 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 //    delete用
17     public Command(int fastOffset, int endOffset, String nodeName){
18         this.fastOffset = fastOffset;
19         this.endOffset = endOffset;
20         this.nodename = nodeName;
21         this.isDeleteCommand = true;
22     }
23
24     public Command(int fastOffset, String string, String nodename) {
25         this.string = string;
26         this.fastOffset = fastOffset;
```

```
27          this.nodename = nodename;
28      }
29  }
```

3.5 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た. 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, 自信で作成する必要が生まれた。

第4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:4.1 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。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. 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することで実装を試みたが,

- 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.

などの問題が見られた. 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.

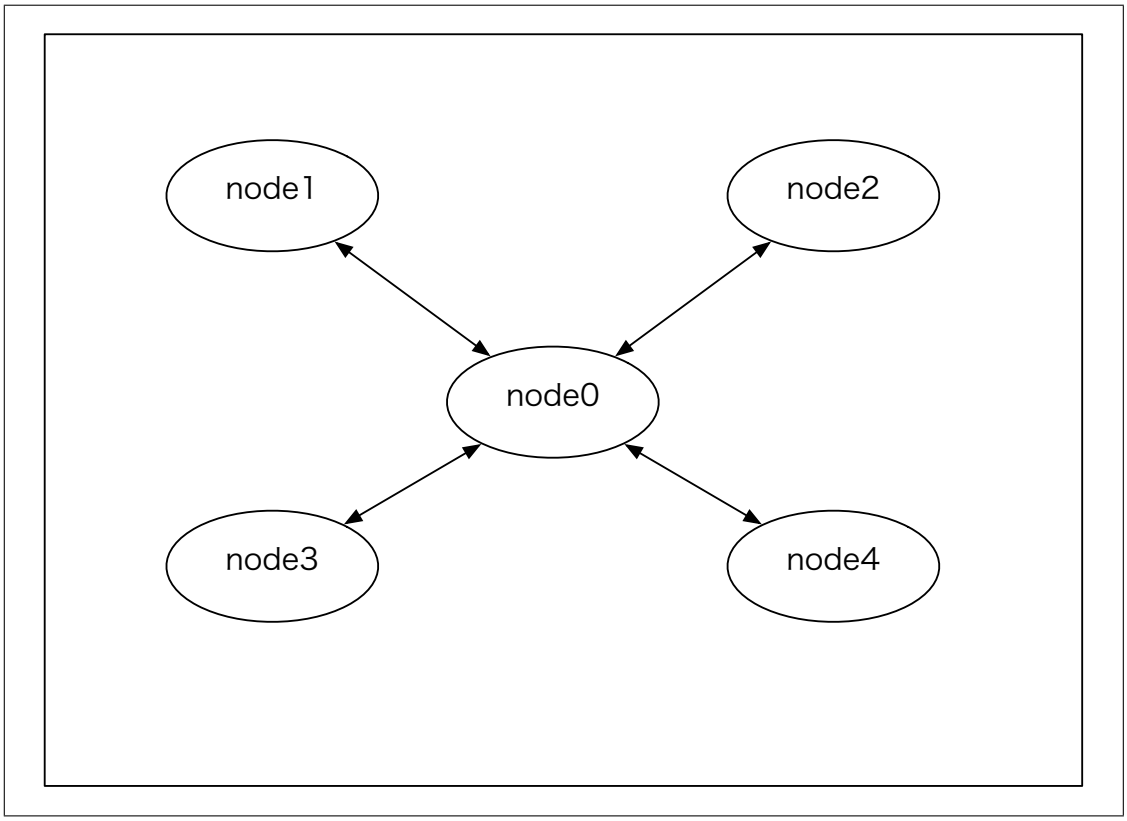

図 4.1: スター型の接続をグラフ化した物

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.

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.

第6章　まとめ

参考文献

[1] 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.

[2] 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.

[3] Bitcoin: A Peer-to-Peer Electronic Cash System
https://bitcoin.org/bitcoin.pdf
(Accessed: 2019/2/15)

[4] Ethereum Homestead Documentation
http://www.ethdocs.org/en/latest/
(Accessed: 2019/2/17)

[5] Paxos made Simple
https://lamport.azurewebsites.net/pubs/paxos-simple.pdf (Accessed: 2019/2/17)

[6] TORQUE Introduction.
http://docs.adaptivecomputing.com/torque/4-2-8/help.htm#topics/0-intro/introduction.htm\%3FTocPath\%3DWelcome\%7C_____1
(Accessed: 2019/2/15)

[7] qsub document.
http://docs.adaptivecomputing.com/torque/4-0-2/Content/topics/commands/qsub.htm
(Accessed: 2019/2/15)

謝辞

本研究を行うにあたり, 日頃より多くの助言, ご指導いただきました河野真治准教授に心より感謝申し上げます。

また, 本研究で使用するツールを作成いただいた照屋のぞみ先輩, 本実験の測定にあたり, torque の環境構築に協力してくださった前城健太郎先輩, 並列信頼研究室の全てのメンバーに深く感謝いたします。最後に、物心両面で支えてくれた両親に深く感謝いたします。