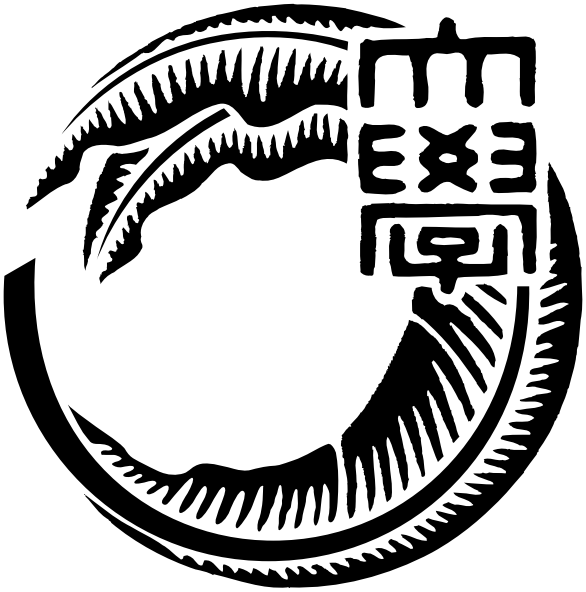
平成31年度 卒業論文

# Christieによるブロックチェーンの実装



琉球大学工学部情報工学科

155753E 赤堀 貴一

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

コンピュータにおいてデータの破損や不整合は深刻な異常を引き起こす原因となる. そのため, 破損, 不整合を検知するためにブロックチェーン技術を用いたい. ブロックチェーンは分散ネットワーク技術であり, データの破損や不整合をハッシュ値によって比較できる. そして, 誤操作や改ざんがあった場合でも, ブロックチェーンを用いることで簡単にデータの追跡が行える.

当研究室では分散フレームワークとして Christie を開発しており, これは GearsOS にファイルシステムとして組み込む予定がある. そのため, Christie にブロックチェーンを実装し, GearsOS に組み込むことにより, GearsOS のファイルシステムにおいてデータの破損, 不整合を検知できる. また, GearsOS 同士による分散ファイルシステムを構成することができ, 非中央的にデータの分散ができるようになる. もし分散システムを構成しない場合でもデータの整合性保持は行え, 上記の目的は達成できる.

本研究では, Christie にブロックチェーンを実装し, 実際に学科の PC クラスタ上の分散環境で動かす.

# 第2章　ブロックチェーンについて

ブロックチェーンとは分散型台帳技術とも呼ばれ, 複数のトランザクションをまとめたブロック, そのブロックをハッシュによって繋げ, 前後関係を表した台帳というものを, システムに参加しているすべてのノードが保持する技術である. ブロックチェーンにはパブリック型とコンソーシアム型の 2 種類がある. パブリック型は不特定多数のノードを対象にしており, コンソーシアム型は管理者が許可したノードが参加している.

## 2.1　P2P

ブロックチェーンのネットワーク間は P2P で動く. つまり, ブロックチェーンネットワークはサーバー, クライアントの区別がなく, すべてのノードが平等である. そのため, 非中央的にデータの管理を行う.

## 2.2　ブロックとその構造

ブロックチェーンにおけるブロックは, 複数のトランザクションをまとめたものである. ブロックの構造は使用するコンセンサスアルゴリズムによって変わるが, 基本的な構造としては次のとおりである.

- BlockHeader
  - previous block hash
  - merkle root hash
  - time
- TransactionList

BlockHeader には, 前のブロックをハッシュ化したもの, トランザクションをまとめた merkle tree の root の hash, このブロックを生成した time となっている.

previous block hash は, 前のブロックのパラメータを並べて, hash 化したものである. それが連なっていることで, 図 2.1 のような hash chain として, ブロックがつながっている.

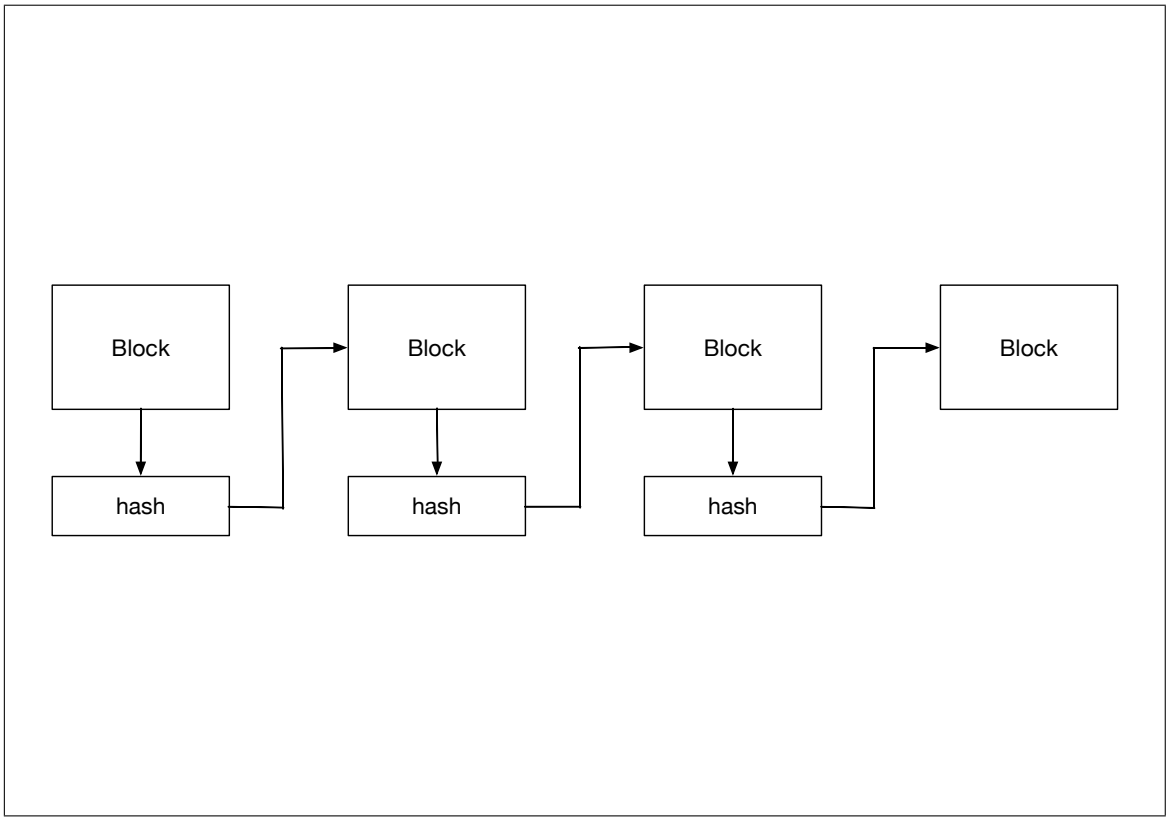


図 2.1: hash chain

そのため, 一つのブロックが変更されれば, その後に連なっているブロックをすべて変更しなければいけなくなる.

ブロックが生成された場合, 知っているノードにそのブロックをブロードキャストする. 実際には通信量を抑えるためにブロック高を送った後にブロックをシリアライズして送る場合もある.

ノードごとにブロックを検証し, 誤りがあればそのブロックを破棄し, 誤りがなければ更にそのノードがブロックをブロードキャストする. . そして, Transaction Pool という Transaction を貯めておく場所から, そのブロックに含まれている Transaction を削除し, 新しいブロックを生成する.

## 2.3 トランザクションとその構造

トランザクションとはデータのやり取りを行った記録の最小単位である. トランザクションの構造は次のとおりである.

**TransactionHash** トランザクションをハッシュ化したもの.

**data** データ.

**sendAddress** 送り元のアカウントのアドレス.

**recieveAddress** 送り先のアカウントのアドレス.

**signature** トランザクションの一部と秘密鍵を SHA256 でハッシュ化したもの. ECDSA で署名している.

トランザクションはノード間で伝搬され, ノードごとに検証される. そして検証を終え, 不正なトランザクションであればそのトランザクションを破棄し, 検証に通った場合は Transaction Pool に取り組まれ, また検証したノードからトランザクションがブロードキャストされる.

## 2.4 fork

ブロックの生成をしたあとにブロードキャストをすると, ブロック高の同じ, もしくは相手のブロック高のほうが高いブロックチェーンにたどり着く場合がある. もちろん, 相手のブロックチェーンはそのブロックを破棄する. しかしこの場合, 異なるブロックを持った 2 つのブロックチェーンができる. この状態を fork という. fork 状態になると, 2 つの異なるブロックチェーンができることになるため, 1 つにまとめなければならない. 1 つにまとめるためにコンセンサスアルゴリズムを使うが, コンセンサスアルゴリズムについては次章で説明する.

# 第3章　コンセンサスアルゴリズム

ブロックチェーンでは, パブリックブロックチェーンの場合とコンソーシアムブロックチェーンによってコンセンサスアルゴリズムが変わる. この章ではパブリックブロックチェーンの Bitcoin, Ethereum に使われている Proof of Work について説明し, コンソーシアムブロックチェーンに使える Paxos を説明する.

## 3.1　Proof of Work を用いたコンセンサス

パブリックブロックチェーンとは, 不特定多数のノードが参加するブロックチェーンシステムのことを指す. よって, 不特定多数のノード間, 全体のノードの参加数が変わる状況でコンセンサスが取れるアルゴリズムを使用しなければならない. Proof of Work は不特定多数のノードを対象としてコンセンサスが取れる. ノードの計算量によってコンセンサスを取るからである. 次のような問題があっても, Proof of Work はコンセンサスを取ることができる.

1. プロセス毎に処理の速度が違う. つまり, メッセージの返信が遅い可能性がある
2. 通信にどれだけの時間がかかるかわからず, その途中でメッセージが失われる可能性がある,
3. プロセスは停止する可能性がある. また, 復旧する可能性もある.
4. 悪意ある情報を他のノードが送信する可能性がある.

Proof of Work に必要なパラメータは次のとおりである.

- nonce
- difficulty

nonce はブロックのパラメータに含まれる. difficulty は Proof of Work の難しさ, 正確に言えば 1 つのブロックを生成する時間を調整している. Proof of Work はこれらのパラメータを使って次のようにブロックを作る.

1. ブロックと nonce を加えたものをハッシュ化する. この際, nonce によって, ブロックのハッシュは全く違うものになる.

2. ハッシュ化したブロックの先頭から数えた 0 ビットの数が difficulty より多ければ, そのブロックに nonce を埋め込み, ブロックを作る.

3. 2 の条件に当てはまらなかった場合は nonce に 1 を足して, 1 からやり直す.

difficulty = 2 で Proof of Work の手順を図にしたものを図 3.1 に示す.

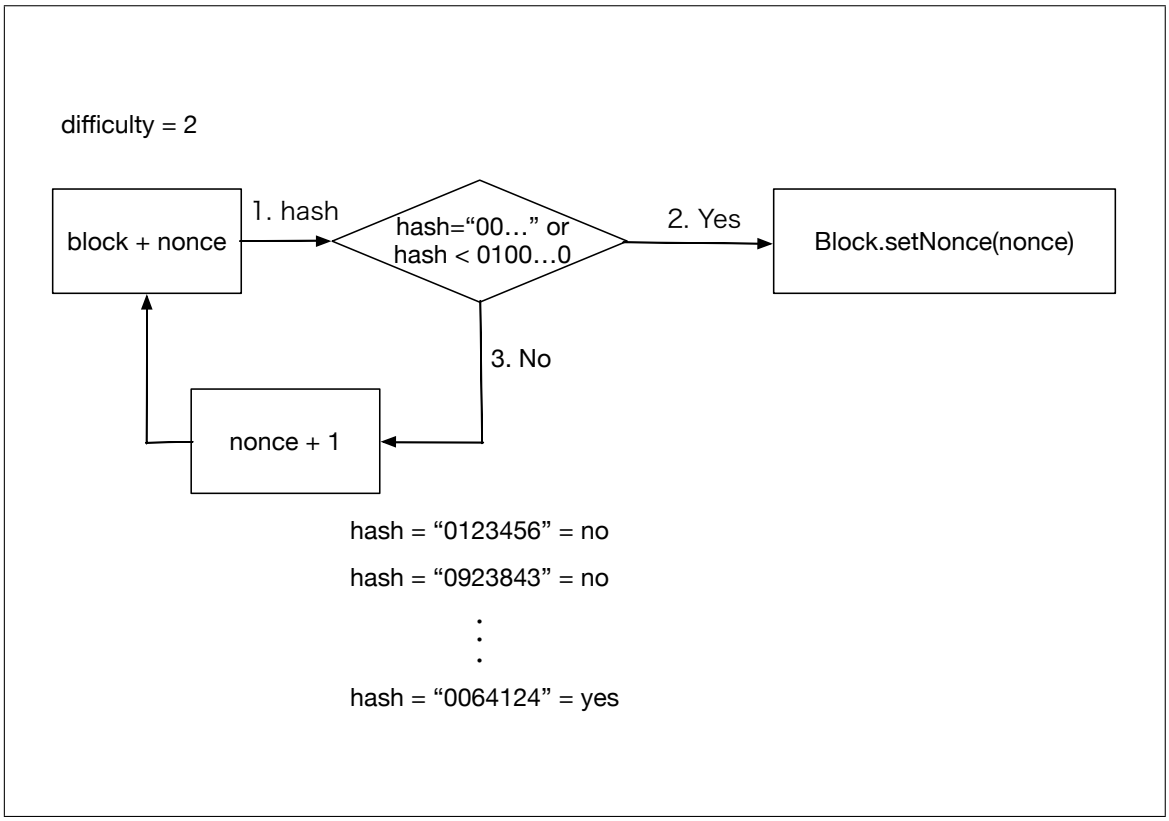



図 3.1: proof-of-work の例

2 の条件については, 単純に (桁数 $- difficulty + 1) \times 10 > hash$ とも置き換えることができる.

nonce を変えていくことで, hash はほぼ乱数のような状態になる. つまり, difficulty を増やすほど, 条件に当てはまる hash が少なくなっていくことがわかり, その hash を探すための計算量も増えることがわかる.

これらが Proof of Work でブロックを生成する手順である. これを用いることによって, ブロックが長くなればなるほど, すでに作られたブロックを変更することは計算量が膨大になるため, 不可能になっていく.

Proof of Work でノード間のコンセンサスを取る方法は単純で, ブロックの長さの差が一定以上になった場合, 最も長かったブロックを正しいものとする. これを図で表すと, 図 3.2 のようになる.

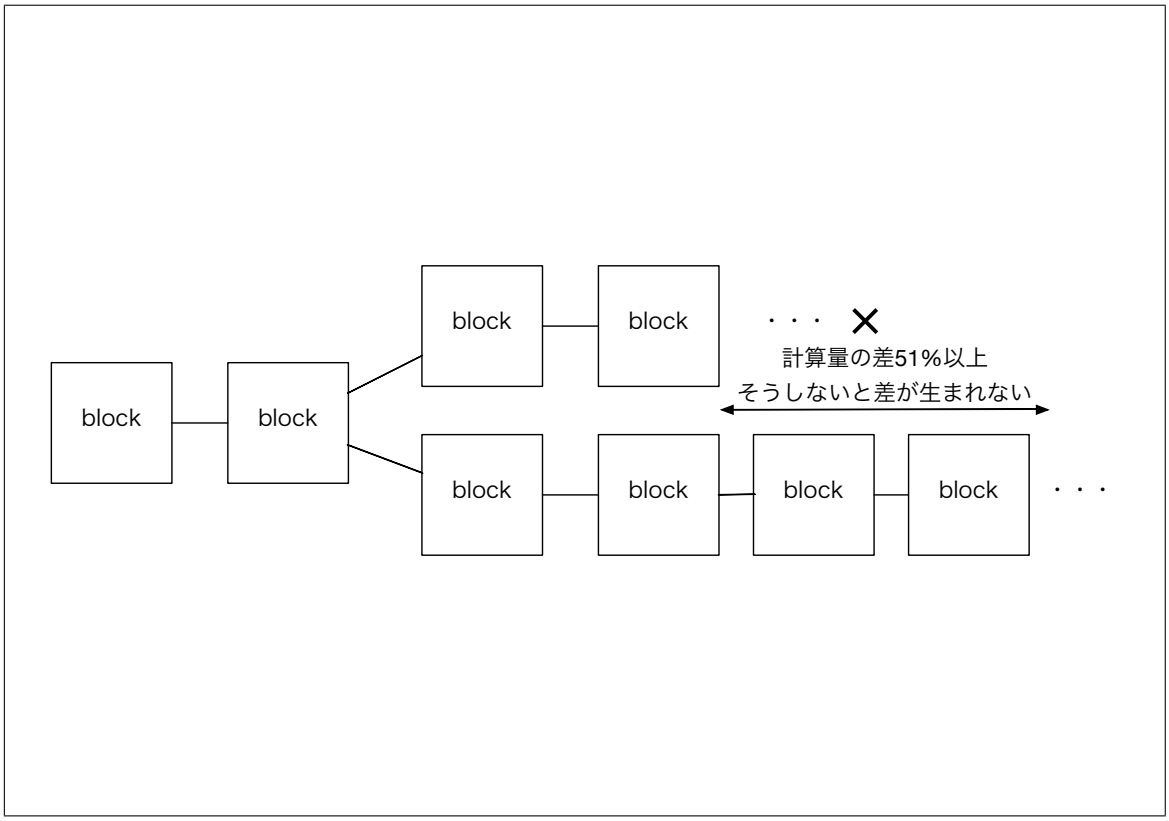


図 3.2: Proof of Work でのコンセンサス

計算量の差が 51%以上になると, fork したブロック同士で差が生まれる. それによって, IP アドレスでのコンセンサスではなく, CPU の性能によるコンセンサスを取ることができる.

コンセンサスでは, ブロックの差が大きければ大きいほど, コンセンサスが正確に取れる. しかし, 正しいチェーンが決まるのに時間がかかる. そのため, コンセンサスに必要なブロックの差はコンセンサスの正確性と時間のトレードオフになっている.

この方法でコンセンサスを取る場合の欠点を挙げる.

- CPU のリソースを使用する.
- Transaction が確定するのに時間がかかる.

## 3.2 Paxos

コンソーシアムブロックチェーンは許可したノードのみが参加できるブロックチェーンである. そのため, ノードの数も把握できるため, Paxos を使うことができる. Paxos はノードの多数決によってコンセンサスを取るアルゴリズムである. ただし, Paxos は次のような問題があっても値を一意に決めることができる.

1. プロセス毎に処理の速度が違う. つまり, メッセージの返信が遅い可能性がある

2. 通信にどれだけの時間がかかるかわからず, その途中でメッセージが失われる可能性がある,

3. プロセスは停止する可能性がある. また, 復旧する可能性もある.

Proof of Work にある特性の 4 がないが, これは許可したノードのみが参加可能だからである. つまり, 悪意あるノードが参加する可能性が少ないためである.

Paxos は 3 つの役割のノードがある.

**proposer** 値を提案するノード.

**acceptor** 値を決めるノード.

**learner** acceptor から値を集計し, 過半数以上の acceptor が持っている値を決める.

Paxos のアルゴリズムに入る前に, 定義された用語を説明する. 以下にその用語の定義を示す.

**提案 (リクエスト)** 提案は, 異なる提案ごとにユニークな提案番号と, 値からなる. 提案番号とは, 異なる提案を見分けるための識別子であり, 単調増加する. 値は一意に決まってほしいデータである.

**値 (提案) が accept される** acceptor によって値 (提案) が決まること.

**値 (提案) が選択される** 過半数以上の acceptor によって, 値 (提案) が accept された場合, それを値 (提案) が選択されたと言う.

Paxos のアルゴリズムは 2 フェーズある.
1 つ目のフェーズ, prepare-promise は次のような手順で動作する.

1. proposer は提案番号 n を設定した提案を過半数以上の acceptor に送る. これを prepare リクエストという.

2. acceptor は prepare リクエストが来たら次の動作をする.

   (a) もし, 以前に送られた prepare リクエストの提案番号より, 今送られてきた prepare リクエストの提案番号のほうが大きければ, それ以下の提案番号の提案を拒否するという約束を返す. この状態を Promise したという.

   (b) もし, 値がすでに accept されていれば, accpet された提案を返す.

2 つ目のフェーズ, accept-accepted は次のような手順で動作する.

1. proposer は過半数の acceptor から返信が来たならば, 次の提案を acceptor に送る. これを accept リクエストという.

(a) もし, 約束のみが返ってきているならば, 任意の値 v を prepare リクエストで送った提案に設定する.

(b) もし, accept された提案が返ってきたら, その中で最大の提案番号を持つ提案の値 v' を prepare リクエストで送った提案の値として設定する.

2. acceptor は accept リクエストが来た場合, Promise した提案よりも accept リクエストで提案された提案番号が低ければ, その提案を拒否する. それ以外の場合は accept する.

このアルゴリズムによって, 各 accptor ごとに値が一意に決まる. 値を集計, 選択するのは Learner の役割である. Learner が値を集計する方法には 2 つの方法がある.

1. Acceptor によって値が accept された時に, 各 Learner に送信される. ただし, Message 通信量が, *Acceptor* の数 × *Learner* の数 になる.

2. 1 つの Learner が各 Learner に選択された値を送信する. 1 の方法に比べて Message 通信量が少なくなる (Acceptor の数の通信量になる) 代わりに, その Learner が故障した場合は各 Learner が Message を受け取れない.

2 つの方法はメッセージ通信量と耐障害性のトレードオフになっていることがわかる.

Paxos でコンセンサスを取ることは, Proof of Work と比較して次のようなメリットがある.

- CPU のリソースを消費しない
- Transaction の確定に時間がかからない.

## 3.3 Paxos によるブロックチェーン

Paxos は Proof of Work に比べ, CPU のリソースを消費せず, Transaction の確定に時間がかからない. そのため, Paxos でブロックのコンセンサスを取るブロックチェーンを実装することにはメリットが有る. また, Paxos 自体がそもそもリーダー選出に向いているアルゴリズムである. そのため, リーダーを決め, そのノードのブロックチェーンの一貫性のみを考えることもできる.

# 第4章　Christieについて

Christie は当研究室で開発している分散フレームワークである．Christie には分散プログラムを簡潔に書くための工夫が複数ある．本章では Christie について述べる．

## 4.1　Christie とは

Christie は Java で書かれた分散フレームワークである．Christie は当研究室で開発している GearsOS に組み込まれる予定がある．そのため, GearsOS を構成する言語 Continuation based C と似た概念がある．Christie に存在する概念として次のようなものがある．

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

CG はクラス, スレッドに相当し, java の継承を用いて記述する．DG は変数データに相当し, CG 内でアノテーションを用いて変数データを取り出せる．CGM はノードであり, DGM, CG, DG を管理する．DGM は DG を管理するものであり, put という操作により変数データ, つまり DG を格納できる．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 と似ているが, 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 と似ているが, 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.

以上が, Christie の概要である.

## 4.2 プログラミングの例

ここでは, Christie で実際にプログラムを記述する例を述べる. CGM を作り, setup(new CodeGear) を動かすことにより, DG を待ち合わせ, DG が揃った場合に CodeGear が実行される. CGM を作る方法は StartCodeGear(以下 SCG) を継承したものから createCGM(port) method を実行することにより, CGM が作られる. SCG のコードの例をソースコード 4.1 に示す.

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.getLocalDGM().put("helloWorld","hello");
16         cgm.getLocalDGM().put("helloWorld","world");
17     }
18 }
```

## 4.3 TopologyManager の実装

Christie は当研究室で開発された Alice を改良した分散フレームワークである. しかし Alice の機能を全て移行したわけではない. TopologyManager は最たる例であり, 分散プ

ログラムを簡潔に書くために必要である. そのため, Christie に TopologyManager を実装した.

ここでは, TopologyManager とはどのようなものかを述べる. そして, TopologyManager を実装する際に, Christie 自身のコードを変更する必要があったため, TopologyManager でどのような問題が起こり, Christie の基本機能をどのような変更したかも述べる.

TopologyManager とは, Topology を形成するため, 参加を表明したノード, TopologyNode に名前を与え, 必要があればノード同士の配線も行うノードである. TopologyManager の Topology 形成方法として, 静的 Topology と動的 Topology がある. 静的 Topology はソースコード 4.2 のような dot ファイルを与えることで, ノードの関係を図 4.1 のようにさせる. 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 が実行される.

ソースコード 4.2: ring.dot

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

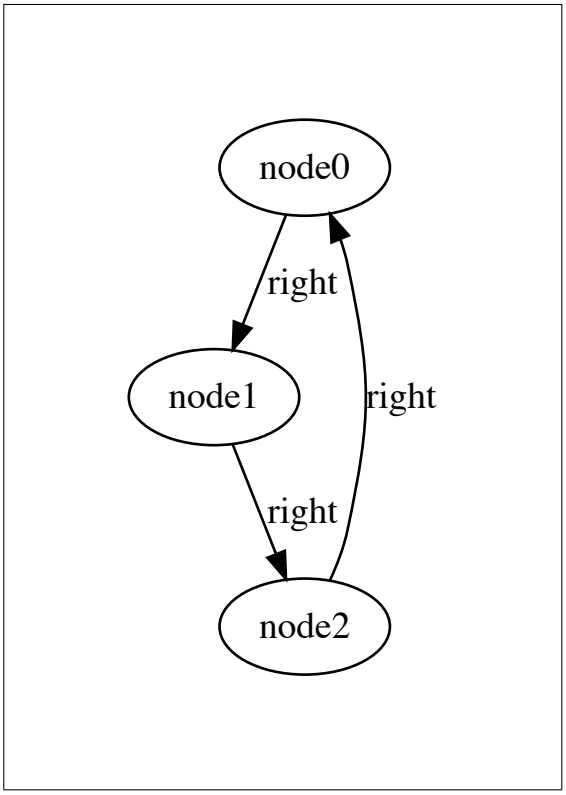



図 4.1: ソースコード 4.2, ring.dot を図にしたもの

動的 Topology は参加を表明したノードに対し, 動的にノード同士の関係を作る. 例え

ばTreeを構成する場合, 参加を表明したノードから順に, rootに近い位置の役割を与える. また, CodeGearはノードが参加し, parentに接続したあとに実行される.

TopologyManagerを実装するに当たって, 以下の2つの問題点が出た.

- Take, Peek操作でSuperClassの型を持ったデータを取り出す際にNullPointerExceptionが表示される.
- ノード間で繋がる前にput操作を行うとデータが送られない.

Take, Peek操作でSuperClassの型を持ったデータを取り出す際にNullPointerExceptionが表示される問題に対しては, DataGearでdataを代入する際にSuperClass, interfacesまで比較するように書き換えた. また, 型の不一致が起こった際は例外を投げるようにした. TopologyNodeにおいて, 実行するCodeGearをputしておき, 参加するノードがすべて揃ったら, そのCodeGearを実行する. しかし, 実際には実行するCodeGearはCodeGearを継承したものである. Christieは, putされたdataのクラスとTakeされるデータのクラスが一致したならば, dataを代入するという処理を行っている. つまり, SuperClass, interfacesの型までは比較をしない. そのため, 型の不一致が起こり, dataの代入をしないため, NullPointerExceptionが表示されていた.

ノード間で繋がる前にput操作を行うとデータが送られない問題に対しては, waitを付け加えた. この問題は, ノードが繋がる前にputを行うため, 相手のDataGearに書き込みが行われないために起きた. そのため, 相手とDataGearがつながるまでputメソッドをwaitしておき, つながってからput操作を行うように書き換えた.

## 4.4 Christieの良い点, 悪い点

Christieの元となった分散フレームワークAliceと比較し, Christieの良い点, 悪い点をそれぞれ述べる.

良い点としては次のようなことが挙げられる.

- ソースコードの可読性が上がった. Aliceでは動的にDataGearのKeyを変更できるため, 実際に使われているクラスと別のところでKeyが変更されている場合も多かった. しかし, Christieでは変数の名前がKeyとなる. そのため, put操作した変数がどこで使われているかがわかりやすくなった.
- データの取り出しが簡単. アノテーションを用いることで, データを簡単に取り出すことができる. また, Aliceでは型をコード内で再定義しなければならなかったが, その操作がなくなった.
- DGMの操作がわかりやすくなった.

悪い点としては次のようなことが挙げられる

- TakeFrom, PeekFrom の使い方が難しい. TakeFrom, PeekFrom は引数で DGM name を指定する. しかし, DGM の名前を静的に与えるよりも, 動的に与えたい場合が多かった.
- デバッグが難しい. cgm.setup で CodeGear が実行されるが, key の待ち合わせで止まり, どこで止まっているかわからないことが多かった. 例えば, put する key のスペルミスなどでコードの待ち合わせが起こり, CodeGear が実行されず, エラーなども表示されずに wait することがあり, どこで止まっているかわからない事があった.

# 第5章　評価

本研究では, 実際にコンセンサスアルゴリズム Paxos を分散環境上で実行した. 分散環境上で動かすため, JobScheduler の一種である Torque Resource Manager(Torque) を使った. ここでは Torque とはなにか, どのような目的で評価をしたかを述べる.

## 5.1　Torque とは

PC クラスタ上でプログラムの実験を行う際には, 他のプログラムとリソースを取り合う懸念がある. それを防ぐために Torque を使用する. Torque は job という単位でプログラムを管理し, リソースを確保できたら実行する. job は qsub というコマンドを使って複数登録することができる. また, 実行中の様子も qstat というコマンドを打つことで監視ができる.

Torque には主に 3 つの Node の種類がある.

**Master Node** pbs_server を実行しているノード. 他のノードの役割とも併用できる.

**Submit/Interactive Nodes** クライアントが job を投入したり監視したりするノード. qsub や qstat のようなクライアントコマンドが実行できる.

**Computer Nodes** 投入された job を実際に実行するノード. pbs_mom が実行されており, それによって job を start, kill, 管理する.

今回は図 5.1 のように, 学科の KVM 上に Master Node, Submit/Interactive Node の役割を持つ VM1 台と, Computer Nodes として 15 台の VM を用意し, job の投入を行った.

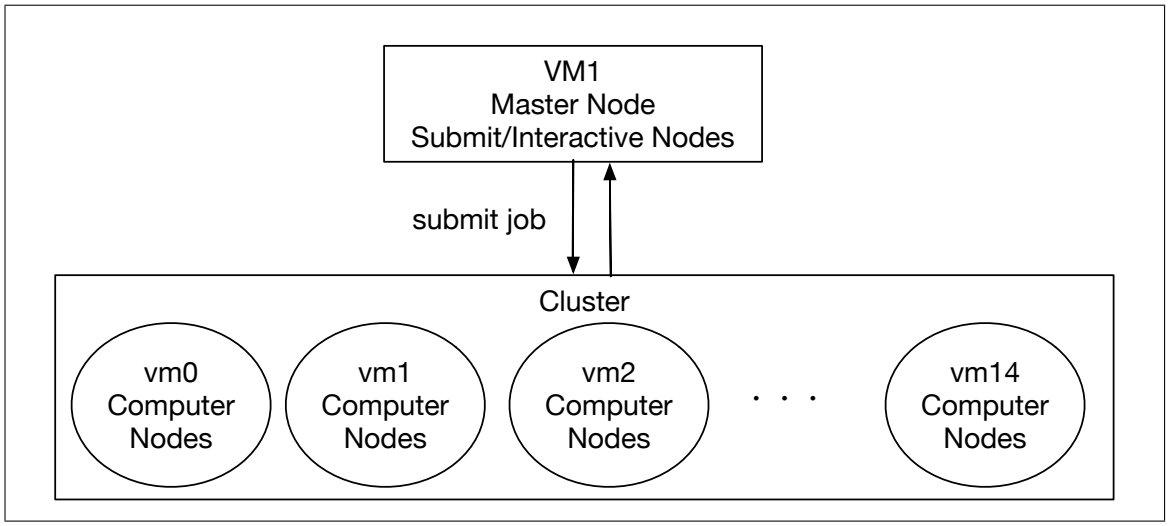



図 5.1: 実験環境

job はシェルスクリプトの形で与えることができる. ソースコード 5.1 を例としてあげる.

ソースコード 5.1: torque-example.sh

```
1 #!/bin/sh
2 #PBS -N ExampleJob
3 #PBS -l nodes=10,walltime=00:01:00
4 for serv in ‘cat $PBS_NODEFILE‘
5 do
6   ssh $serv hostname &
7 done
8 wait
```

「#PBS オプション」とすることにより実行環境を設定できる. 使用できるオプションは参考文献 [3] に書かれてある. 例として, ノード数 10(vm0 から vm9 まで), job の名前を「ExampleJob」という形で実行した. その結果を ExampleJob が

# 第6章　まとめ

# 参考文献


[1] Bitcoin: A Peer-to-Peer Electronic Cash System
https://bitcoin.org/bitcoin.pdf
(2018 年 2 月 15 日 アクセス)

[2] TORQUE Introduction.
http://docs.adaptivecomputing.com/torque/4-2-8/help.htm#topics/0-intro/introduction.htm\%3FTocPath\%3DWelcome\%7C_____1
(2018 年 2 月 15 日 アクセス)

[3] qsub document.
http://docs.adaptivecomputing.com/torque/4-0-2/Content/topics/commands/qsub.htm
(2018 年 2 月 15 日 アクセス)

# 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。