分散フレームワークChristieによるBlock chainの実装

一木 貴裕[†1]　河野 真治[†1]

概要：当研究室で開発した分散フレームワーク Christie を用いて，Etherium を参考に Block chain を実装した．Paxos のリーダ選出アルゴリズムと，Christie の持つ Topology Manager との相性が問題になる．Gears OS のファイルシステムなどに使えるかどうかの調査を行う．

Implementing Block chain using Distributed Computing Framework Christie

1. Block Chain と Gears OS

コンピュータのデータに不整合は起こり得る．不整合は誤操作や，複数人によるデータの同時書き込みによって起こる．特に分散環境下で問題になる．ブロックチェーンはデータを分散でき，不整合の検知が可能な仕組みを提供していう．当研究室で開発中の GearsOS の分散ファイルシステムの技術として，ブロックチェーンが使用できるかどうかを調査中である．そのために当研究室では Java 上で開発された分散フレームワーク Christie にブロックチェーンを実装することにした．

2. Chrsitie

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 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．GearsOS とは同様に当研究室で開発しており，言語 Continuation based C によって OS そのものとアプリケーションを記述する．そのため GearsOS を構成する言語 CbC 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，DGM を管理する．

DS Manager(以下 DSM) には Local DSM と Remote DSM が存在する．Local DSM は各ノード固有のデータベースである．

Remote DSM は他ノードの Local DSM に対応する proxy であり，接続しているノードの数だけ存在する (図 1)．他ノードの Local DSM に書き込みたい場合は Remote DSM に対して書き込めば良い．

†1 現在，琉球大学工学部情報工学科
Presently with Information Engineering, University of the Ryukyus.

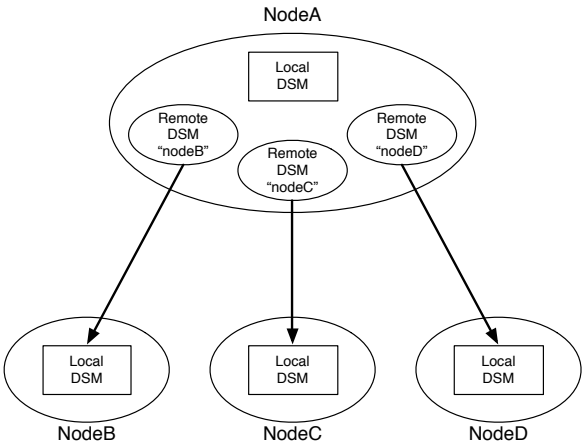

図 1: Remote DSM は他のノードの Local DSM の proxy

Remote DSM を立ち上げるには，DataSegment クラスが提供する connect メソッドを用いる．接続したいノードの ip アドレスと port 番号，そして任意の Manager 名を指定することで立ちあげられる．その後は Manager 名を指定して Data Segment API を用いて DS のやり取りを行うため，プログラマは Manager 名さえ意識すれば Local への操作も Remote への操作も同じ様に扱える．

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，すなわち 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 操作を行った後は，対象の DGM の中に que として補完される．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．

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 を持ち合わせ，DG が揃った場合に CodeGear が実装される．CGM を作る方法は StartCodeGear(以下 SCG) を継承したものから create-CGM(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 1 に示す．

Code 1: StartHelloWorld

```
package christie.example.HelloWorld;

import christie.codegear.CodeGearManager;
import christie.codegear.StartCodeGear;

public class StartHelloWorld extends StartCodeGear {

    public StartHelloWorld(CodeGearManager cgm) {
        super(cgm);
    }

    public static void main(String[] args){
        CodeGearManager cgm = createCGM(10000);
        cgm.setup(new HelloWorldCodeGear());
        cgm.getLocalDGM().put("helloWorld","hello");
        cgm.getLocalDGM().put("helloWorld","world");
    }
}
```

3. Annotation

Christie では Input DG の指定にはアノテーションを使う．アノテーションとは，クラスやメソッド，パッケージに対して付加情報を記述できる Java の Meta Computation である．先頭に@をつけることで記述でき，オリジナルのアノテーションを定義することもできる．Input となる型の変数を直接宣言し，変数名として key を記述する．そして，その宣言の上にアノテーションで Take または Peek を指定する (ソースコード 2).

Code 2: T

```
@Take
public String name;
```

アノテーションで指定した InputDG は，CG を生成した際に CodeGear.class 内で待ち合わせの処理が行われる．これには Java の reflectionAPI を利用しており， アノテーションと同時に変数名も取得できるため，変数名による key 指定が実現した．

Christie のこのインプットアノテーションはフィールドに対してしか記述できないため，key の指定と Take/Peek の指定を必ず一箇所で書くことが明確に決まっている．これにより，外の CS からの key への干渉をされることがなり，key の指定が複数の CG に分散することがない．また，key を変数名にしたことで，動的な key の指定や，key と変数名の不一致による可読性の低下を防ぐことができた．

リモートノードに対して Take/Peek する際は，Take-

From/PeekFrom のアノテーションを用いる (ソースコード 3).

Code 3: TakeFrom の例

```
@TakeFrom("remote")
public String name;
```

Christie は通信の際に DG を圧縮することができる．圧縮の Meta Computation を指定する際に DGM 名の前に compressed をつけることができる．(ソースコード 4).

Code 4: Remote から圧縮して受け取る例

```
@TakeFrom("compressedremote")
public String name;
```

Christie では Input DG は直接変数を宣言されているので，他の場所を辿らなくとも CG を見るだけでインプットされるデータの型が分かるようになっている．ソースコード 5 は InputDG のデータを扱うである．

Code 5: InputDG を扱う例

```
public class GetData extends CodeGear{

    @Take
    public String name;

    @Override
    protected void run(CodeGearManager cgm) {
        System.out.println("this␣name␣is␣:␣" + name);
    }
}
```

InputDG として宣言した変数の型は，reflectionAPI により内部で保存され，リモートノードと通信する際も適切な変換が行われる．このようにプログラマが指定しなくとも正しい型で取得できるため，プログラマの負担を減らし信頼性を保証することができる．

4. TopologyManager

TopologyManager とは，Topology を形成するために，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コードである．TopologyManager の Topology 形成方法として，静的 Topology と動的 Topology がある．静的 Topology はコード 6 のような dot ファイルを与えることで，ノードの関係を図 2 のようにする．静的 Topology は dot がいるのノード数と同等の TopologyNode があって初めて，CodeGear が実行される．

Code 6: ring.dot

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

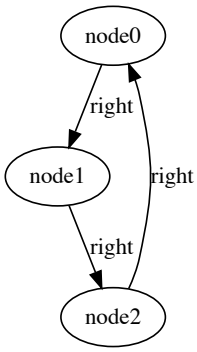

図 2: ring.dot を図式化したもの

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 mparent に接続された後に実行される．

5. ブロックチェーンのトランザクション

ブロックチェーンは P2P にてネットワーク間が動作している，つまり．ブロックチェーンネットワークにはサーバー，クライアントの区別がなく，全てのノードが平等である．そのため，非中央時にデータの管理をおこなう．

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，そ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選べて hash 化したものである．それが連なっていることで図 3 のような hash chain として，ブロックがつながっている．

そのため，一つのブロックが変更されれば，その後につながるブロック全てを更新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

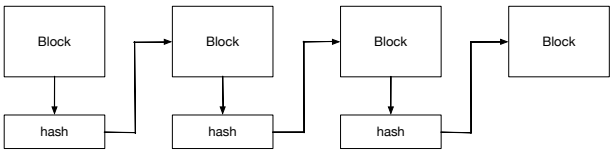

図 3: ブロックチェーンの図

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を送った後，ブロックをシリアライズして送信する場合もある．

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．

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．

6. Proof of Work を用いたコンセンサス

ブロックの生成をした後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高いブロックチェーンにたどり着く場合がある．当然，相手のブロックチェーンはこれを破棄する．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をこの状態を fork と呼ぶ．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，一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用いるが，コンセンサスアルゴリズムについては次章で説明する．

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 と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．

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さす．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．次のような問題が生じても 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，正確にいえば 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

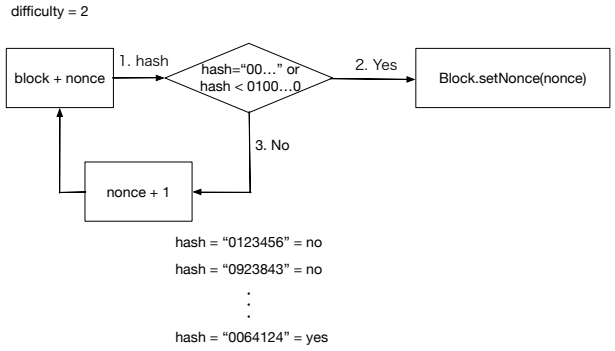

図 4: Proof of Work の手順

4 に示す．

2 の条件については，単純に (桁数 - difficulty + 1) * 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．これが Proof of Work でブロックを生成する手順となる．これを用いることによって，ブロックが長くなるほど， 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に長かったブロックを正しいものとする．これを図で示すと 5 のようになる．

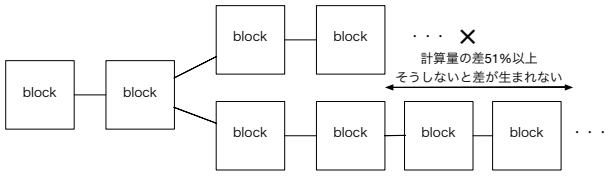

図 5: Proof of Work のコンセンサス

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， ブロックと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．

7. Paxos

コンソーシアムブロックチェーンは許可したの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， コンソーシアムブロックチェーンは 3 つの問題を解決するだけで十分である．何故ならば，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 のアルゴリズムの説明の前に，定義された用語の解説をする．いかにその用語の定義を示す．

提案 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 **(chosen)** 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 フェーズ目を図にしたものを図 6 に示す．

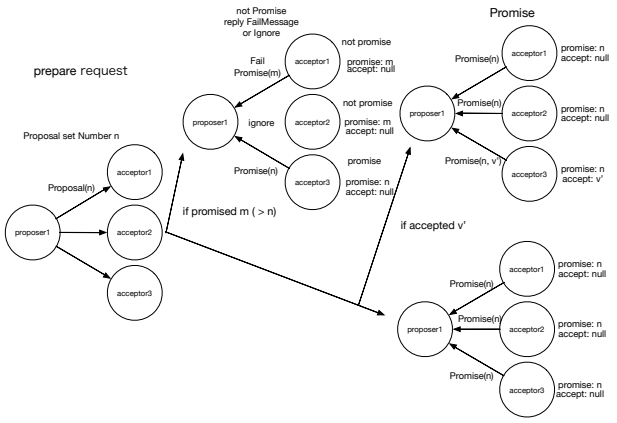

図 6: prepare-promise

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．

2 フェーズ目を図にしたものを図 7 に示す．

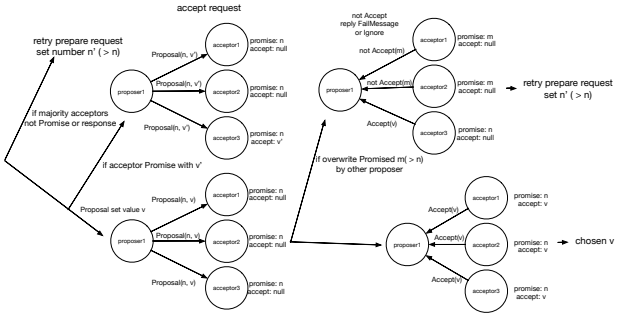

図 7: accept-accepted

このアルゴリズムによって，各 acccepter ごとに値が一意の決まる．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 つの方法である．

(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 times 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 + Learner の数になる) 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．

Paxos は Proof of Work と比べ，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ある．また，Paxos 自体が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．

8. Chrisite におけるブロックチェーンの実装の利点と欠点

Christie においてブロック，トランザクション，Paxos, Proof of Work を実装した．その際，Christie で実装した場合には以下のような利点がある．

- データの取り出しが簡単．Christie は DataGear という単位でデータを保持する．そのため，ブロックやトランザクションは DataGear に包めばいいため，どう送るかという問題を考えなくてすむ．
- TopologyManager でのテストが便利．dot ファイルが有れば，TopologyManager が任意の形で Topology を作れる．そのため，ノードの配置については理想の環境を作れるため，理想のテスト環境を作ることができる．
- 機能ごとにファイルが実装できるため，見通しが良い．Christie は CbC の goto と同じように関数が終わると setup によって別の関数に移動する．そのため自然に機能ごとにファイルを作るため，見通しが良くなる．

一方で, Christie には以下の問題点があることがわかった.

- 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，どこの CG 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 のスペルミスでコードの待ち合わせが起こり，CG が実行されず，エラーなども表示されずに wait することがある．その時に，どこで止まっているか特定するのが難しい．
- 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．
- Take の待ち合わせで CG が実行されない．2 つの CG で同じ変数を Take しようとすると，setup された時点で変数がロックされる．このとき， 片方の CG は DG がすべて揃っているのに，すべての変数が揃っていないもう片方の CG に同名の変数がロックされ， 実行されない場合がある．

9. 実験

本研究室では， 実際にコンセンサスアルゴリズム Paxos を PC 上に分散環境を実装して検証した．分散環境場で動かすため，JobScheduler の一種である Torque Resource Manager(Torque) を使用した.

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 というコマンドを使う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 8 のように，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 の投入を行なった.

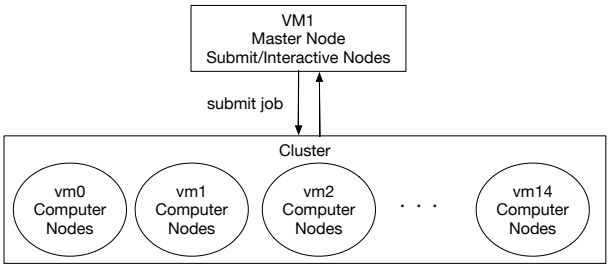

図 8: 実験環境

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 7 を例として挙げる.

Code 7: torque-example.sh

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

このスクリプトでは，ノード数 10(vm0 から vm9 まで), job の名前を「ExampleJob」という形で実行する設定をしている．もし， このコードを投入した場合，Submit/Interactive Nodes が各 vm に ssh し，hostname コマンドを実行する．実行後は stdout，stderror の出力を「job 名.o 数字」， job 名.e 数字」というファイルに書き出す.

PC クラスタ上で実際に Paxos を動かした際の解説をする．今回は単純化し，proposer の数を 2，accepter の数を 3，learner の数を 1 として Paxos を動かし，値が一意に決まる過程を見る．実験の単純化の為に，提案の数値を整数とし，各 processer ごとに異なった値とした．正確には，「proposer + 数字」の部分を値とし，コンセンサスを取るようにした．実際に Paxos を動かし，シーケンス図で結果を示したものが図 9 である.

一意の値を決めることができており，また，Learner が値を選択した後でも，Paxos は常に決めた値を持ち続けるアルゴリズムである．ログの出力では提案番号がより大きいものの提案まで続いていたが，値がこれ以上覆らなかった．今回はわかりやすいよう値を数字で行なった実験であるが，これをタランザクション，ブロックに応用することで，ブロックチェーンにおけるコンセンサス部分を完成させることができる.

10. まとめ

Paxos の動作は確認したが，計測実験を行うには至っていない．トランザクションの速度がノード数にどのように影響されるのかを調べる必要がある.

Christie の Toplogy Manager は実験するノードの設定

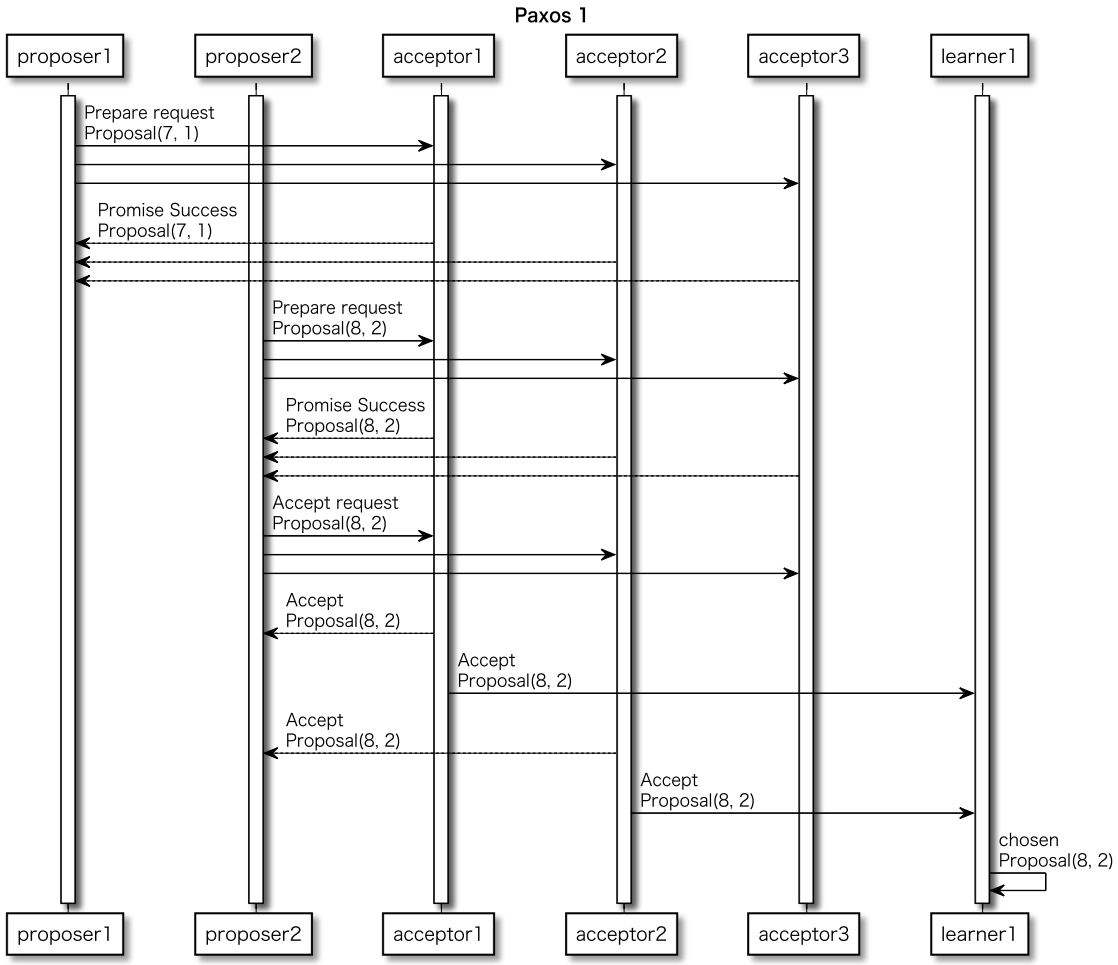

図 9: Paxos 動作を表したシーケンス図

を行う集中制御ノードであり，本来集中制御部分を持たないブロックチェーンとの相性は良くない． しかし，分散ファイルシステム等の用途の場合は Topology Manager のような方法の方がノードの管理が可能な利点がある．

今後は実装を進め，Gears OS でのファイルシステムへの応用などを考えていきたい．

参考文献

[1] 赤堀 貴一，河野真治：**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卒業論文 2019.
[2] 照屋 のぞみ，河野真治：分散フレームワーク **Christie** の設計．琉球大学理工学研究科修士論文 2018.