分散フレームワークAliceのPC画面配信システムへの応用

照屋 のぞみ,[a)]　河野 真治,[b)]

概要：当研究室ではデータを Data Segment、タスクを Code Segment という単位で分割して記述する手法を提唱しており、それに基づく並列分散フレームワーク Alice を開発している。Alice が分散プログラムを記述する能力を有することは水族館の例題等において確認された。しかし、実用的な分散アプリケーションを構築するには圧縮形式で通信を行う機能等が必要であることが分かった。本研究では、 実用的なアプリケーションである画面共有システム TreeVNC を Alice で実装するにあたり必要となった圧縮機能等を Meta Computation として実装した。データの多態性の実現により、扱うデータの形式を元のコードを大きく変更することなく指定することができ、ノード間通信における自由度の向上を図った。

キーワード：並列分散フレームワーク

1. 研究背景と目的

当研究室ではデータを Data Segment、タスクを Code Segment という単位で記述する分散フレームワーク Alice の開発を行っている。Alice ではスケーラブルな分散プログラムを信頼性高く記述できる環境を実現する。ここで言う信頼性とは、定められた環境下で安定して仕様に従った動作を行うことを指す。

Alice では、処理を Computation と Meta Computation に階層化し、コアな仕様と複雑な例外処理に分離する。そして分散環境の構築に必要な処理を Meta Computation として提供する。プログラマはコアな仕様の変更を抑えつつプログラムの挙動変更ができるため、信頼性の高い分散アプリケーションの記述が可能となる。

先行研究 [1] の水族館の例題等において、Alice が分散プログラムを記述する能力を有することは確認された。しかし、実用的な分散アプリケーションを作成するためには、動的トポロジーを管理・構成する機能や通信時に Data Segment を圧縮形式で扱う機能が必要な場合がある。本研究では、Alice 上に実用的な分散アプリケーションの例題である画面共有システム TreeVNC [2] を構築する。構築するにあたり必要となった圧縮などの機能を、Alice の Meta Computation として実装する。そして Alice を使用していない TreeVNC との比較を行うことで Meta Computation の役割と有効性を示す。

2. 分散フレームワーク Alice

2.1 Code Segment と Data Segment

Alice では Code Segment（以下 CS）と Data Segment（以下 DS）の依存関係を記述することでプロ

a) dpop@cr.ie.u-ryukyu.ac.jp
b) kono@ie.u-ryukyu.ac.jp

グラミングを行う。CS は実行に必要な DS が全て揃うと実行される。CS を実行するために必要な入力 DS は InputDS、CS が計算を行った後に出力される DS は Output DS と呼ばれる。データの依存関係にない CS は並列実行が可能である（図 1 ）。CS の実行において DS が他の CS から変更を受けることはない。そのため Alice ではデータが他から変更され整合性がとれなくなることはない。

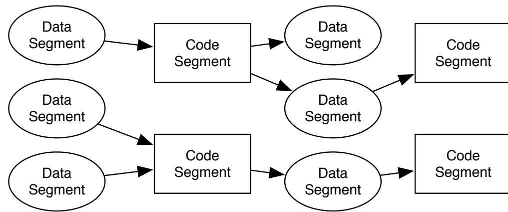

図 1　CodeSegment の依存関係

実際には Alice は Java で実装されており、DS は JavaObject で CS は RunnableThread である。プログラマーが CS を記述する際は、CodeSegment クラスを継承し、DS を操作する API を使用する。

2.2　DataSegmentManager

DS は数値や文字列などの基本的なデータの集まりを指し、Alice が内部にもつデータベースによって管理されている。このデータベースを Alice では DS Manager（以下 DSM）と呼ぶ。

DS には対になる String 型の key が存在し、この key を指定して DS の保存・取得を行う。一つの key に対して複数の DS を登録することもでき、その場合 DS は queue に保存され FIFO で取り出される。

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 2 ）。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。Remote DSM を立ち上げるには、DataSegment.class 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。

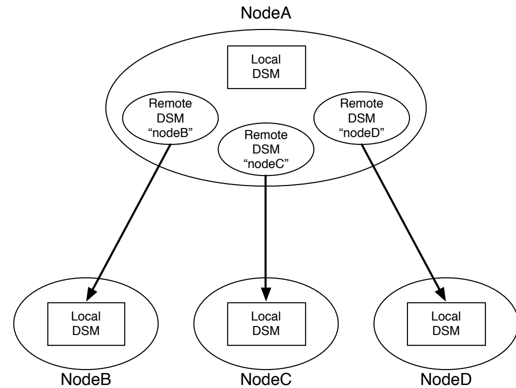

図 2　Remote DSM は他のノードの Local DSM の proxy

2.3 Data Segment API

DS の保存・取得には Alice が提供する API を用いる。put と update は Output DS API と呼ばれ、DS を DSM に保存する際に用いる。peek と take は Input DS API と呼ばれ、DS を DSM から取得する際に使用する。

- `void put(String managerKey, String key, Object val)`

DS を DSM に追加するための API である。第一引数は LocalDSM か RemoteDSM かといった Manager 名を指定する。そして第二引数で指定された key に対応する DS として第三引数の値を追加する。

- `void update(String managerKey, String key, Object val)`

update も DS を DSM に追加するための API である。put との違いは、queue の先頭の DS を削除してから DS を追加することである。そのため API 実行前後で queue の中にある DS の個数は変わらない。

- `void take(String managerKey, String key)`

take は DS を読み込むための API である。読み込まれた DS は削除される。要求した DS が存在しなければ、CS の待ち合わせ（Blocking）が起こる。put や update により DS に更新があった場合、take が直ちに実行される。

- `void peek(String managerKey, String key)`

peek も DS を読み込む API である。take との違いは読み込まれた DS が削除されないことである。

2.4 Code Segment の記述方法

CS をユーザーが記述する際には CodeSegment.class を継承して記述する (ソースコード 1，2)。継承することにより Code Segment で使用する Data Segment API を利用する事ができる。

Alice には、Start CS (ソースコード 1) という C の main に相当するような最初に実行される CS がある。Start CS はどの DS にも依存しない。つまり Input DS を持たない。この CS を main メソッド内で new し、execute メソッドを呼ぶことで実行を開始させることができる。

```
1  public class StartCodeSegment extends
       CodeSegment {
2
3      @Override
4      public void run() {
5          new TestCodeSegment();
6
7          int count = 0;
8          ods.put("local", "cnt", count);
9      }
10
11 }
```

Code 1 StartCodeSegment の例

```
1  public class TestCodeSegment extends
       CodeSegment {
2      private Receiver input1 = ids.create(
           CommandType.TAKE);
3
4      public TestCodeSegment() {
5          input1.setKey("local", "cnt");
6      }
7
8      @Override
9      public void run() {
10         int count = input1.asInteger();
11         System.out.println("data = " +
               count);
12         count++;
13         if (count == 10)
14             System.exit(0);
15
16         new TestCodeSegment();
17         ods.put("local", "cnt", count);
18     }
19 }
```

Code 2 CodeSegment の例

ソースコード 1 は、5 行目で次に実行させたい CS（ソースコード 2 ）を作成している。8 行目で Output DS API を通して Local DSM に対して DS を put している。Output DS API は CS の `ods` というフィールドを用いてアクセスする。`ods` は `put` と `update` を実行することができる。

TestCodeSegment はこの"cnt"という key に対して依存関係があり、8 行目で put が行われると TestCodeSegment は実行される。

ソースコード 2 は、0 から 9 までインクリメントする例題である。2 行目で取得された DS が格納される受け皿を作る。Input DS API がもつ create メソッドを使うことで作成できる。

- `Receiver create(CommandType type)`

引数には CommandType が取られ、指定できる CommandType は `PEEK` または `TAKE` である。Input DS API は CS の `ids` というフィールドを用いてアクセスする。Output DS は、`ods` が提供する put/update メソッドをそのまま呼べばよかったが、Input DS の場合 `ids` に peek/take メソッドはなく、create/setKey メソッド内で CommandType を指定して実行する。

4 行目から 6 行目はコンストラクタである。コンストラクタはオブジェクト指向のプログラミング言語で新たなオブジェクトを生成する際に呼び出されて内容の初期化を行う関数である。

TestCodeSegment のコンストラクタが呼ばれた際には、

(1) CS が持つフィールド変数 `Receiver input` に `ids.create(CommandType.TAKE)` が行われ、`input` が初期化される。
(2) 5 行目にある TestCodeSegment のコンストラクタの TAKE が実行される。

5 行目は Input DS API がもつ setKey メソッドにより Local DSM から DS を取得している。

- `void setKey(String managerKey, String key)`

setKey メソッドは peek/take の実行を行う。どの DSM のどの key に対して peek または take コマンドを実行させるかを指定できる。コマンドの結果がレスポンスとして届き次第 CS は実行される。

run メソッドの内容としては

(1) 10 行目で取得された DS を Integer 型に変換して count に代入する。
(2) 12 行目で count をインクリメントする。
(3) 16 行目で次に実行される CS が作られる。(この時点で次の CS は Input DS の待ち状態に入る)
(4) 17 行目で count を Local DSM に put する。Input DS が揃い待ち状態が解決されたため、次の CS が実行される。
(5) 13 行目が終了条件であり、count の値が 10 になれば終了する。

となっている。

3. Meta Computation

Alice では、計算の本質的な処理を Computation、Computation とは直接関係ないが別のレベルでそれを支える処理を Meta Computation として分けて考える。Alice の Computation は、key により DS を待ち合わせ、DS が揃った CS を並列に実行する処理と捉えられる。それに対して、Alice の Meta Computation は、Remote ノードとの通信時のトポロジーの構成や切断・再接続の処理と言える。つまりトポロジーの構成は Alice の Computation を支えている Computation とみなすことができる。

Alice の機能を追加するということはプログラマ側が使う Meta Computation を追加すると言い換えられる。Alice では Meta Computation として分散環境の構築等の機能を提供するため、プログラマーは CS を記述する際にトポロジー構成や切断、再接続という状況を予め想定した処理にする必要はない。プログラマーは目的の処理だけ記述し、切断や再接続が起こった場合の処理を Meta Computation として指定する。このようにプログラムすることで、通常処理と例外処理を分離することができるため、仕様の変更を抑えたシンプルなプログラムを記述できる。現在 Alice には、動的・静的トポロジーの管理構成機能、ノードとの接続状態確認機能、切断・再接続時の処理を指定できる機能などの Meta Computation が用意されている。

3.1 DS の Meta Computation

CS はアプリケーションを動作させるために必要なタスクであり、ユーザーによって定義される。それに対して Meta CS は Alice を構成するタスクである。つまり Meta CS の群は Alice の Computation と言い換えることができる。一部のみユー

ザーが定義をすることができ、Alice の挙動を変更することができる。

Topology Manager

Alice では、ノード間の接続管理やトポロジーの構成管理を、Topology Manager という MetaComputation が提供している。この Topology Manager も CS/DS を用いて実装されている。プログラマーはトポロジーファイルを用意し、Topology Manager に読み込ませるだけでトポロジーを構成することができる。トポロジーファイルは DOT Language?という言語で記述される。DOT Language とは、プレーンテキストを用いてデータ構造としてのグラフを表現するためのデータ記述言語の一つである。ソースコード 3 は 3 台のノードでリングトポロジーを組むときのトポロジーファイルの例である。また、DOT Language ファイルは dot

```
1 digraph test{
2     node0 -> node1[label="right"]
3     node0 -> node2[label="left"]
4     node1 -> node2[label="right"]
5     node1 -> node0[label="left"]
6     node2 -> node0[label="right"]
7     node2 -> node1[label="left"]
8 }
```

Code 3 トポロジーファイルの例

コマンドを用いてグラフの画像ファイルを生成することができる。そのため、記述したトポロジーが正しいか可視化することが可能である。

Topology Manager はトポロジーファイルを読み込み、参加を表明したクライアント（以下、Topology Node）に接続するべきクライアントの IP アドレスやポート番号、接続名を送る（図 3）。また、トポロジーファイルで lavel として指定した名前は RemoteDSM の名前として Topology Node に渡される。そのため、Topology Node は Topology Manager の IP アドレスさえ知っていれば自分の接続すべきノードのデータを受け取り、ノード間での正しい接続を実現できる。

また、実際の分散アプリケーションでは参加するノードの数が予め決まっているとは限らない。

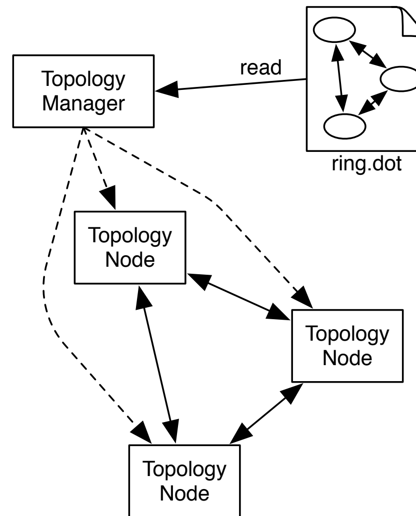

図 3 TopologyManager が記述に従いトポロジーを構成

そのため Topology Manager は動的トポロジーにも対応している。トポロジーの種類を選択して Topology Manager を立ち上げれば、あとは新しい Topology Node が参加表明するたびに、Topology Manager から Topology Node に対して接続すべき Topology Node の情報が put され接続処理が順次行われる。そして Topology Manager が持つトポロジー情報が更新される。現在 Topology Manager では動的なトポロジータイプとして二分木に対応している。

Data Segment の表現

DS の表現には MessagePack for Java [3] を利用している。

- `DS` は一般的な `Java` のクラスオブジェクト
- `MessagePack` を用いて変換した `byte[]` で表現されたバイナリオブジェクト

の 2 種類があり、LocalDSM に put された場合は一般的な Java のクラスオブジェクトとして追加される。RemoteDSM に put された場合は通信時に byteArray に変換されたバイナリオブジェクトが追加される。

3.2 CS の Meta Computation

KeepAlive

切断・再接続時の処理

4. AliceVNC

Alice の Meta Computation が実用的なアプリケーションの記述において有用であることを確認する。そのために、TreeVNC を Alice を用いて実装した AliceVNC の作成を行った。

TreeVNC とは、当研究室で開発を行っている授業向け画面共有システムである。オープンソースの VNC である TightVNC [4] をもとに作られている。授業で VNC を使う場合、1つのコンピュータに多人数が同時につながるため、性能が大幅に落ちてしまう。この問題をノード同士を接続させ、木構造を構成することで負荷分散を行い解決したものが TreeVNC である。図 4 は AliceVNC を実現するための構成である。left と right の RemoteDSM を用意し子ノードと接続することで木構造を実現する。

TreeVNC は通信処理部分の記述が大変複雑になっている。しかし、Alice で記述すれば本質的な処理とそれを支える通信処理部分で分離できる。そのため、TightVNC からの修正も少なく、見通しの良い記述で構成可能と期待される。

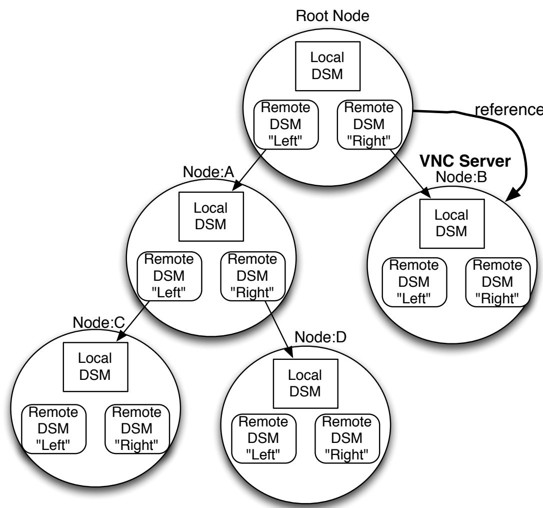

図 4 AliceVNC の構造

5. Alice の新機能

実用的なアプリケーションである TreeVNC を Alice 上で実装することで、Alice に必要な機能を洗い出した。

5.1 転送機能

Input DS を Reciever に取得したあと、プログラマーは Reciever から値を任意の型で取り出し、値を操作した後 put メソッドで再度別クラスに変換され Output DS として出力する。しかし、Input DS として取得したデータをそのまま子ノードに Output DS として出力する場合、一度 Reciever から取り出し再変換する操作は無駄である。

そこで、Input DS として受け取った DS をそのまま Output DS として転送する機能を put/update とは別に flip メソッドを Data Segment API に実装した。Input DS である Receiver を展開せずに flip メソッドに引数として渡すことで、展開のオーバーヘッドをなくしている。TreeVNC では親ノードから受け取った画面データをそのまま子ノードに配信するため、Meta Computation として転送機能が有用である。

5.2 Data Segment の表現の追加（圧縮機能）

TreeVNC では画面配信の際、データを圧縮してノード間通信を行っている。そのため、AliceVNC にも圧縮されたデータ形式を扱える機能が必要だと考えた。しかし、ただデータを圧縮する機構を追加すればいいわけではない。

AliceVNC では、ノードは受け取った画面データを描画すると同時に、子ノードの Remote DSM に送信する。ノードは DS を受信するとそれを一度解凍して画面を表示し、再圧縮して子ノードに送信する。しかし、受け取ったデータを自分の子ノードに対して送信する際には、解凍する必要はない。圧縮状態のまま子ノードに送信ができれば、解凍・再圧縮するオーバーヘッドを無くすことができる。

そこで、1つの Data Segment に対し複数の表現を持たせることで、必要に応じた形式で DS を扱う

ことを可能にした。DS を扱う ReceiveData.class に、次の 3 種類の表現を同時に持つことができる。

(1) 一般的な Java のクラスオブジェクト
(2) MessagePack for Java でシリアライズ化されたバイナリオブジェクト
(3) 2 を圧縮したバイナリオブジェクト

ソースコード 4 は ReceiveData.class が持つ表現であり、`val` に (1) 一般的な Java のクラスオブジェクト の表現でデータ本体が保存される。`messagePack` には (2) シリアライズ化されたバイナリオブジェクトが保存され、通常の RemoteDSM への通信にこの表現が扱われる。そして、`zMessagePack` には (3) 圧縮されたバイナリオブジェクトが保存される。

```
1 public class ReceiveData {
2     private Object val = null;
3     private byte[] messagePack = null;
4     private byte[] zMessagePack = null;
5 }
```

Code 4 データを表現するクラス

また、圧縮状態を持つ DS を扱う DSM として Local と Remote それぞれに Compressed Data Segment Manager を追加した。Compressed DSM の内部では、put/update が呼ばれた際に ReceiveData.class が圧縮表現を持っていればそれを使用し、持っていなければその時点で圧縮表現を作って put/update を行う。

ソースコード 5 は RemoteDSM に対し Int 型のデータを put する記述である。これを DS を圧縮形式で行いたい場合、ソースコード 6 のように指定する DSM 名の先頭に"compressed"をつければ Compressed DSM 内部の圧縮 Meta Computation が走り DS を圧縮状態で扱うようになる。

```
1 ods.put("remote", "num", 0);
```

Code 5 通常の DS を扱う CS の例

```
1 ods.put("compressedremote", "num", 0);
```

Code 6 圧縮した DS を扱う CS の例

これによりユーザは指定する DSM を変えるだけで、他の計算部分を変えずに圧縮表現を持つ DS を扱うことができる。ノードは圧縮された DS を受け取った後、そのまま子ノードに flip メソッドで転送すれば圧縮状態のまま送信されるので、送信の際の再圧縮がなくなる。画面表示の際は ReceiveData.class 内の `asClass()`(ソースコード 7) を使うことで適切な形式でデータを取得できる。`asClass()` は DS を目的の型に cast するメソッドであり、ReceiveData.class が圧縮表現だけを持っている場合はこのメソッド内で解凍して cast を行っている。これにより DS の表現を必要になったときに作成できる。

5.3 Alice の通信プロトコルの変更

2.4 Data Segment の表現 で述べたように、Remote から put されたデータは必ずシリアライズ化されており byteArray で表現される。しかし、TreeVNC のようにもとから byteArray の画像デー

```
public <T> T asClass(Class<T> clazz) {
    if (val != null) {
        return (T) val;
    }

    if (zMessagePack != null &&
         messagePack == null) {
        messagePack = unzip(zMessagePack,
              dataSize);
    }

    return packer.read(messagePack, clazz);
}
```

Code 7 asClass の処理

タを put する場合、MessagePack でシリアライズされたものかの判別が付かない。また、データの表現に圧縮した byteArray を追加したため、Remote から put された byteArray が圧縮されているのかそうでないのかを判断する必要がある。

そこで、Alice の通信におけるヘッダにあたる CommandMessage.class(ソースコード 8) にシリアライズ状態表すフラグと、圧縮状態を表すフラグを追加した。

```
public class CommandMessage {
    public int type;
    public int seq;
    public String key;
    public boolean quickFlag = false;
    public boolean serialized = false;
    public boolean compressed = false;
    public int dataSize = 0;
}
```

Code 8 CommandMessage

表 1 CommandMessage の変数名の説明

変数名	説明
type	CommandType PEEK，PUT などを表す
seq	Data Segment の待ち合わせを行っている Code Segment を表す unique number
key	どの Key に対して操作を行うか指定する
quickFlag	SEDA を挟まず Command を処理を行うかを示す
serialized	データ本体のシリアライズ状態を示す
compressed	データ本体の圧縮状態を示す
dataSize	圧縮前のデータサイズを表す

これによって put された DSM はフラグに応じた適切な形式で ReceiveData.class 内に DS を格納できる。また、CommandMessage.class に圧縮前のデータサイズも追加したことで、適切な解凍が可能になった。

6. 評価と考察

TreeVNC を Alice 上で構築するために必要な機能を Alice の Meta Computation として実装した。それにより、AliceVNC が簡潔な記述で TreeVNC と同等の性能を出せれば、実用的な分散アプリケーションの実装において Alice の Meta Computation は有用であるといえる。そこで、TreeVNC と AliceVNC の性能評価としてメッセージ伝達速度の比較を、コードの評価としてコード量とその複雑度の比較を行った。

また、Alice の Meta Computation の価値を明確にするため、他言語・フレームワークとの比較を行った。

6.1 メッセージ伝達速度の比較

TreeVNC/AliceVNC において、配信する画像データは構成した木を伝ってノードに伝搬され、接続する人数が増える毎に木の段数は増えていく。そこで、木の段数ごとにメッセージの到達にどれぐらい時間がかかっているかを計測した。

実験環境

講義内で学生に協力してもらい、最大 17 名の接続がある中で TreeVNC、AliceVNC(圧縮・転送機能あり)、AliceVNC(圧縮・転送機能なし) の木の段数 1 3 の測定を行った。

実験内容

ルートノードから画面データを子ノードに伝搬する際に、計測用のヘッダをつけたパケットを子ノードに送信する。各子ノードはパケットを受け取り自身の Viewer に画面データを表示すると同時に、計測用ヘッダ部分のみの DS を作成し、親ノードに送り返す（図 5）。計測用 DS は木を伝ってルートノードまで送り返され、ルートノードは受け取った計測用 DS から到達時間を計算する。

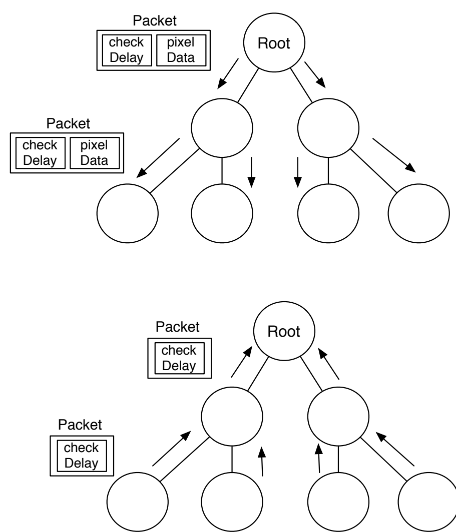

図 5 各ノードごとに到達時間を測定

計測用のヘッダは以下の要素で構成されている。time にはパケットの送信時刻を、dataSize に

表 2 計測用ヘッダの変数名の説明

変数名	説明
time	ルートノードがパケットを送信した時刻
depth	木の段数。初期値=1。
dataSize	送信時の形式に変換済みの画面データのサイズ

は画面データのサイズを付けて送信する。今回、TreeVNC と AliceVNC(圧縮・転送機能あり) では圧縮形式の画面データのサイズを、AliceVNC(圧縮・転送機能なし) では MessegePack 形式でのサイズを dataSize にセットする。depth は各ノードに到達するごとにインクリメントされる。

計算方法をソースコード ??に示す。到達時間は、計測用 DS を受け取った時刻と DS の time（送信した時刻）の差をとる。この到達時間は画面データがノードまで到達した時間と計測 DS をルートまで送り返す時間を含めているが、送り返す時間は誤差として考える。また、depth は各ノードに到達するごとにインクリメントされるため、送り返す際もインクリメントされる。そのため、木の段数を計算するには depth を 1/2 した値となる。

```
1 time = System.currentTimeMillis() −
        receiveDS.getTime();
2 depth = receiveDS.getDepth() / 2;
```

Code 9 到達時間・木の段数の計測方法

実験結果

3 段目の測定結果の散布図を示す（図 6 〜 7)。X 軸が画面データのサイズ (byte)、Y 軸が計算した到達時間 (ms) である。実験時間の都合上、AliceVNC（圧縮・転送機能あり）の計測時間が他より短くなってしまったためプロットされた点の数が少なくなっている。また、それぞれの図で処理に 10000ms 以上かかっている点の集合が見られるが、これは今回の実験において 3 段目に PC のスペック上処理が遅いノードが 1 台あったためである。そのため比較においてこの点集合は無視する。

どの図も同様の傾向があり、画面データのサイズが小さいうちは処理時間も 5ms 程度だが、50000byte 以上から比例して処理時間も遅くなっている。このことから AliceVNC は TreeVNC と同等の処理性能があることがわかる。

また、AliceVNC を圧縮機能の有無でデータサイズ比較すると、圧縮機能のない AliceVNC はデータサイズがほとんど 1000byte 以上なのに対し、圧縮機能のある AliceVNC では TreeVNC 同様 10byte 程度のサイズに抑えるので圧縮も成功している。

さらに転送機能の有無で比較した場合、転送機能がない AliceVNC では木の段数に関係なく 1000ms 近く到達に時間がかかってしまっているが、転送機能のある AliceVNC ではデータサイズが大きくなっても 100ms 程度に抑えられている。これは

転送機能が余計なコピーを防いでいるためだと考えられる。このことから、圧縮・転送の Meta Computation は分散通信において有用であると言える。

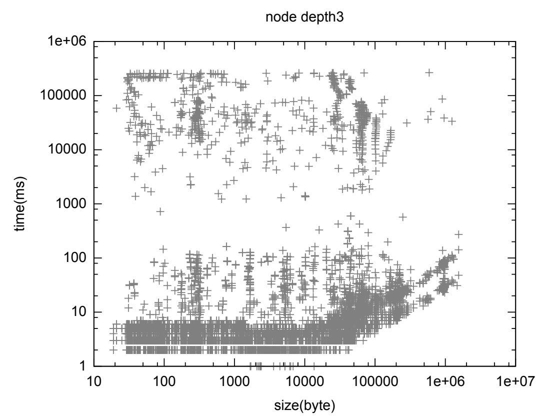

図 6 TreeVNC の測定結果

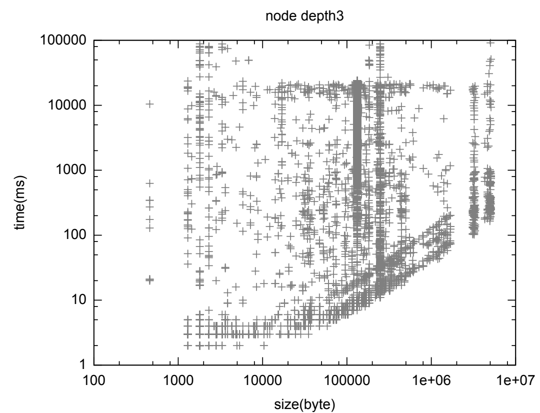

図 7 AliceVNC(圧縮・転送機能なし) の測定結果

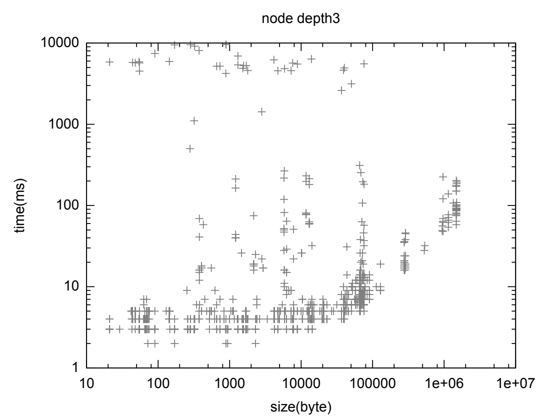

図 8 AliceVNC(圧縮・転送機能あり) の測定結果

6.2 コードの比較

コード量

TreeVNC と AliceVNC のコード量を比較した表が表 3 である。TightVNC を含むコード全体に wc を行い、行数と単語数を比較した。また、hg diff で TightVNC からの変更行数を調べ変更量を比較した。

表からわかるように、Alice を用いればコードの行数が 25%削減できる。また、TreeVNC では TightVNC に大幅に修正を加えながら作成したため仕様の変更が多かった。しかし、AliceVNC では TightVNC にほとんど修正を加えることなくトポロジー構成等の Alice の Meta Computation を使うために新しいクラスを作成したのみであった。そのため TreeVNC に比べ 75%も仕様の変更が抑えられている。

表 3 コード量の比較

	行数	単語数	変更行数
TreeVNC	19502	73646	7351
AliceVNC	14647	59217	1129
減少率 (%)	25	20	75

コードの複雑度

コード量の比較で述べたように、TreeVNC は TightVNC からの変更が多い。その理由の一つがトポロジーの構成や通信処理がコアな仕様と分離できておらず、そのため TreeVNC は大変複雑な記述になってしまっている。

そこで TreeVNC と AliceVNC においてコードの複雑度を比較した。今回、複雑度の指標として Thomas McCabe が提案した循環的複雑度?を用いた。循環的複雑度とはコード内の線形独立な経路の数であり、if 文や for 文が多ければ複雑度も高くなりバグ混入率も高まる。一般的に、循環的複雑度が 10 以下であればバグ混入率の少ない非常に良いコードとされる。計測には IntelliJ の CodeMetrics 計測プラグインである MetricsReloaded を使用した。

表 4 は TightVNC、TreeVNC、AliceVNC における循環的複雑度の比較である。プロジェクト全体でのクラスの複雑度の平均値と最高値をとった。平均値・最高値ともに AliceVNC のほうが複雑度が低いことから、Alice ではシンプルな記述が可能だということがわかる。また、TreeVNC で最高値を出した TreeRFBProto.class は全てプログラマが記述したコードであり、通信処理や画面データの読み込みなどの複数のスレッド処理が集中している。しかし、AliceVNC で最高値を出した SwingViewerWindow.class は TightVNC で最高値を出したクラスと同じであり、コード量の比較でも示したように AliceVNC で変更を加えた点がほとんどない。つまりこの複雑度は元来 TightVNC が持っている複雑度と言える。

表 4 複雑度の比較

	平均値	最高値
TightVNC	13.63	97
TreeVNC	15.33	141
AliceVNC	10.95	99

AliceVNC と TreeVNC の性能比較・コード比較から、AliceVNC は TreeVNC と同等の性能を持つ分散アプリケーションの記述ができ、かつコードの修正量・複雑度共に低く抑える能力を有することがわかった。

6.3 他言語・フレームワークとの比較

Erlang

Akka

?

7. まとめ

並列分散フレームワーク Alice の計算モデルと実装について説明を行い、Alice におけるプログラミング手法を述べた。

Alice が実用的なアプリケーションを記述するために必要な Meta Computation として、データの多態性を実現し、指定する DSM の切り替えで扱うデータ表現を変えるようにした。これにより、必要に応じた形式を扱うことができ、ユーザが記述する Computation 部分を大きく変えずに自由度の高い通信を行うことが可能になった。同様の手法を用いれば、圧縮形式以外にも暗号形式・JSON

形式などの複数のデータ表現をユーザに扱いやすい形で拡張することができる。Alice に圧縮等の Meta Computation を追加したことで、AliceVNC ではシンプルな記述で TreeVNC と同等の性能を提供できると期待される。

TreeVNC を Alice 上で実装し性能評価を行ったことで、TreeVNC で使用される基本機能は Alice でも実現でき、同等の性能を出すことができるということが分かった。またコードの観点から TreeVNC と AliceVNC を比較した結果、Alice の Meta Computation が仕様の変更を抑えたシンプルな記述を実現し、信頼性の高い実用的な分散アプリケーションを構築するに有用であることが確認された。

今後の課題としては、TreeVNC で実装が困難であった NAT を超えたノード間通信を AliceVNC で実現し、その性能とコード修正量を比較することが挙げられる。図 9 は 2 つの違うプライベートネットワークを超えた接続の設計例である。

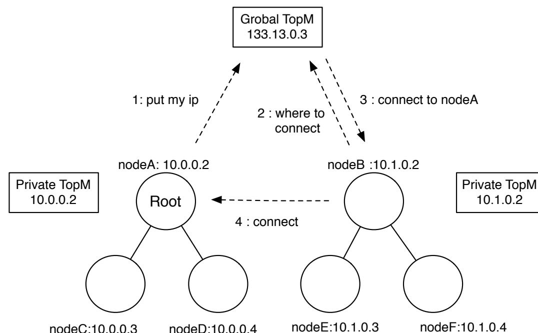

図 9 複数の TopologyManager で NAT 超えを実現

各ネットワークごとに Topoogy Manager を立ち上げることでネットワークを超えたノード間接続を実現する。プライベートネットワークの Topoogy Manager は今までどおりネットワーク内に木を構築・管理する（図 9 左）。他のネットワークにあるノード B がノード A に接続したい場合は、グローバルアドレスを持った Topology Manager に参加表明をすればノード A の情報が提供され、ノード A の子ノードとして接続される。つまり、Topology Manager を複数用意するだけで、Topology Manager 自体の「参加表明のあったノードで木を構成する」という仕様は全く変更しないで良い。TreeVNC では 500 行以上の変更が必要とされたが、Alice では複数の Topology Manager に接続するための config ファイルを変更するだけなので、変更は 100 行以下に抑えられると期待される。この機能も実現できれば、Alice の Meta Computation が拡張性の高い環境を提供できると言える。

参考文献

[1] Yu SUGIMOTO and Shinji KONO: Code Segment と Data Segment によるプログラミング手法, 第 54 回プログラミング・シンポジウム (2013).

[2] Miwa OSHIRO and Shinji KONO: 授業やゼミ向けの画面配信システム TreeVNC の拡張機能，琉球大学工学部情報工学科平成 26 年度学位論文 (学士) (2014).

[3] : MessagePack, `http://msgpack.org/`.

[4] : TightVNC Software, `http://www.tightvnc.com`.

[5] 柏原正三：プログラミング言語 Erlang 入門，アスキー (2007).