

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¹ 河野 真治^{1,a)}

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 が実用的な分散アプリケーションの記述能力を有することを確認するために、画面共有システム TreeVNC を Alice 上で構築した。TreeVNC を Alice で実装するにあたり必要となった圧縮機能等を Alice が提供する Meta Computation として実装した。そして記述したアプリケーションの性能とコードを比較したことで、Alice が実用的な分散アプリケーションをシンプルな記述で実現でき、仕様の変更を抑えた信頼性の高いプログラミングが可能だということを確認した。

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[1][2] 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。プログラム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。

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 [3] を構築する。TreeVNC は画面変更の差分を木構造にそって配布する分散システムで、差分は数 MByte に達するので圧縮を行う必要がある。そして表示時には伸長したデータを取り扱わなければならない。また、データのノード間の転送では複製を可能な限り避ける必要がある。差分データは Alice の Data Segment に対応するため、Data Segment を扱う Alice の機能として圧縮やゼロコピー転送の機能が必要なる。これらの機能は TreeVNC では ad-hoc に実装されているが、Alice ではこれを Meta Computation として実装する。そして、TreeVNC との比較を行うことで Alice の実用性を示すと共に Alice の Meta Computation の役割と有効性を示す。

¹ 琉球大学工学部情報工学科

^{a)} kono@ie.u-ryukyu.ac.jp

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されるDSのことをInputDS、CSが計算を行った後に出力されるDSのことをOutputDSと呼ぶ。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。

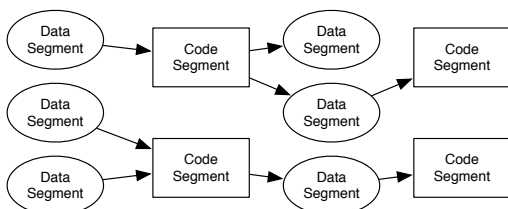


図1 CodeSegmentの依存関係

AliceはJavaで実装されており、DSはJava Objectに相当する。CSはRunnableなObject(void run()を持つObject)に相当する。プログラマがCSを記述する際は、CodeSegmentクラスを継承する。

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と呼ぶ。

CSは複数のDS Managerを持っている。DSには対になるString型のkeyが存在し、それぞれのManagerにkeyを指定してDSにアクセスする。一つのkeyに対して複数のDSをputするとFIFO的に処理される。なのでData Segment Managerは通常のデータベースとは異なる。

2.2 Data Segment Manager

DS Manager (以下DSM)にはLocal DSMとRe-

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上げられる。その後はManager名を指定してDataSegment 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。

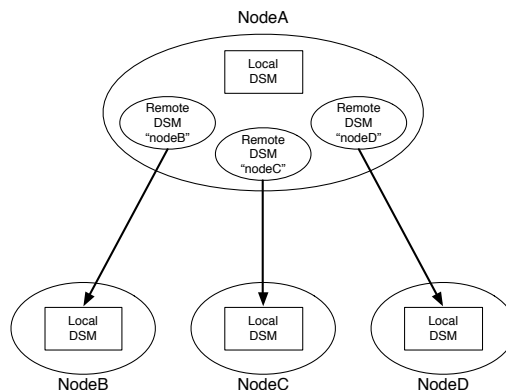


図2 Remote DSMは他のノードのLocal DSMのproxy

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Local DSMかRemote DSM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 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 クラス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 を利用することができる。

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。

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。

CS の Input DS は、CS の作成時に指定する必要がある。指定は CommandType(PEEK か

```

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

TAKE)、DSM 名、そして key よって行われる。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。

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行目では、Input DS API がもつ create メソッドで Input DS を格納する受け皿 (Receiver) を作っている。引数には PEEK または TAKE を指定する。

- Receiver create(CommandType type)

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。

5行目は、2行目の create で作られた Receiver が提供する setKey メソッドを用いて Local DSM から DS を取得している。

- void setKey(String managerKey, String key)を記述できる。

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。

実行される run メソッドの内容は

- (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

Meta Computation は、CS の処理を支える Meta CS と、Meta CS に管理される Meta DS としても考えられる。図3は、Alice の Meta CS/Meta DS の接続関係の例である。プログラマ側は CS と DS の依存関係を記述するが、その裏では Meta CS や Meta DS が間に接続されて処理を行っている。

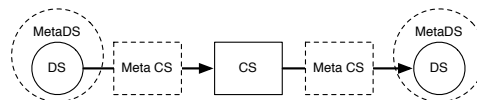


図3 CS/DS の間に MetaCS/MetaDS が接続される

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、分散環境の実現に必要なさまざまな Meta Computation が用意されている。

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 Meta Computation 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[4] 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 コマン

```

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

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 4）。また、トポロジーファイルで label として指定した名前は Remote DSM 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。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

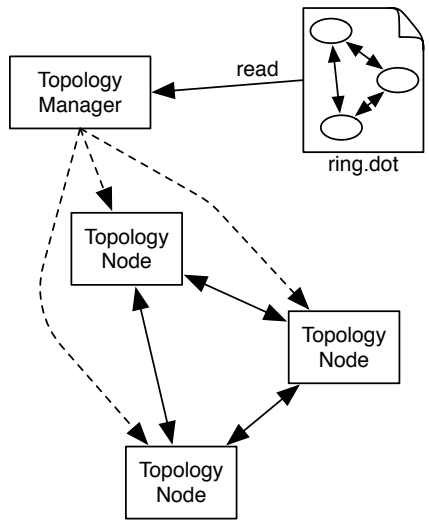


図 4 Topology Manager が記述に従いトポロジーを構成している。

KeepAlive

ノード間通信は Remote DSM に対して put や take を行うことでのみ発生する。アプリケーション次第では長時間通信が行われない可能性があり、その間にノード間接続が切れた場合、次の通信が行われるまで切断を発見することができない。また、接続状態ではあるが応答に時間がかかる場合もある。

これらの問題を検知するために、KeepAlive という定期的に heart beat を送信し生存確認を行う Meta Computation がある。この機能も CS/DS を用いて実装されている。一定時間内にノードからの応答がない場合、KeepAlive により、そのノードの Remote DSM が切断される。

また、トポロジーからノードが切断された際にトポロジーを再構成する機能も Topology Manager に用意した。例えばツリートポロジーでノードが切断された場合、そのノードの子ノードは全体のトポロジーから分断されてしまう。ノードは切断を検知するとただちに Topology Manager に再接続すべきノード情報を要求し、木を構成し直す。

切断・再接続時の処理

MMORPG では、試合の最中にサーバーからユーザーが切断された場合、自動的にユーザーが操作するキャラクターをゲームの開始時の位置に戻すという処理が実行される。同様に、Alice を用いたアプリケーションでもノードの切断時に対する処理を用意したい場合がある。そこで、Alice が切断を検知した際に任意の CS を実行できる機能 (ClosedEventManager) を追加した。プログラマは切断の際に実行したい CS を書き、ClosedEventManager に登録しておけば良い (ソースコード 4)。

```

1 public class StartCodeSegment extends
    CodeSegment {
2
3     @Override
4     public void run() {
5         ClosedEventManager manager =
            ClosedEventManager.
                getInstance();
6         manager.register(ShowClosedNode.
            class);
7
8         new TestCodeSegment();
9         ods.update("local", "key1", "String-
            data");
10    }
11
12 }
    
```

Code 4 切断時に実行される CS の登録方法

また、再接続してきたノードに対し通常の処理とは別の処理を行わせたい場合がある。そのため、切断時と同様に再接続してきたノードに任意の CS を実行できる Meta Computation も用意した。

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 [5] 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 5 は AliceVNC を実現するための構成である。left と right の Remote DSM を用意し子ノードと接続することで木構造を実現する。

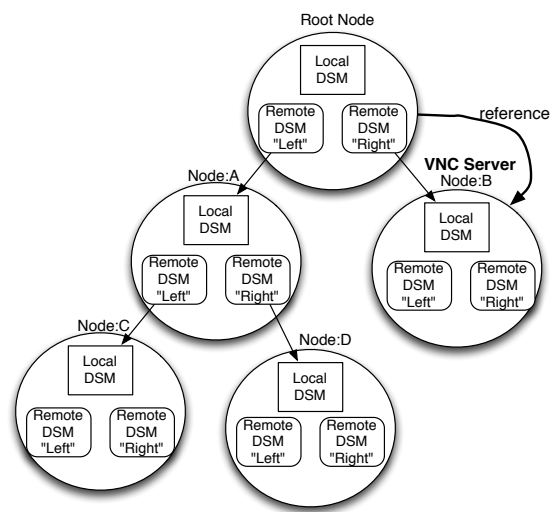


図 5 AliceVNC の構造

TreeVNC は通信処理部分や画面データの処理部分が1つのコード内で記述され、大変複雑になっている。しかし、Alice で記述すれば Meta Computation により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。TreeVNC では3章で述べた動的なトポロジーの構成、切断ノードの発見、再接続・トポロジーの再構成といった通信処理の Meta Computation が活用できる。そのため、TightVNC からの修正の少ない、見通しの良い記述で構成可能と期待される。

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 Meta Computation 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形式のまま子ノードに 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ることができる。

そこで、1つの Data Segment に対し複数の表現を持たせ、必要に応じた形式で DS を扱うことを可能にした。Meta DS に相当する ReceiveData.class に、次の3種類の表現を同時に持つことができるようにしたことで、データの多態性を実現した。

- (1) 一般的な Java のクラスオブジェクト
- (2) MessagePack for Java[6] 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

Local DSM に put された場合は、(1) の一般的な Java クラスオブジェクトとして追加される。Remote DSM に put された場合は、通信時に (2) の byteArray に変換されたバイナリオブジェクトに変換された DS が追加される。この2つの形式は従来の Alice が持っていた表現である。今回、Remote DSM に圧縮形式での通信を行いたいため、(3) の圧縮表現を追加した。

ソースコード 5 は ReceiveData.class が持つ表現である。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。このように DS が複数の表現を同時に保持することで、DS が圧縮表現を持っている場合に再圧縮する必要はない。

```

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

Code 5 データを表現するクラス

また、圧縮表現を持つ DS を扱う DSM として Remote DSM 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なければそれを使用し、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 6 は Remote DSM に対し Int 型のデータを put する記述である。この通信を圧縮形式の DS で行いたい場合、ソースコード 7 のように指定する DSM 名の先頭に”compressed”をつけなければ Compressed DSM 内部の圧縮 Meta Computation が走り圧縮形式に変換された DS となって通信が行われる。

これによりユーザは指定する DSM を変えるだ

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

Code 6 通常の DS を扱う CS の例

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

Code 7 圧縮した DS を扱う CS の例

けで、他の計算部分を変えずに圧縮表現を DS 内で持つことができる。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 メソッドを使うことで適切な形式でデータを取得できる。asClass メソッドは DS を目的の型に cast するためのメソッドである。AliceVNC で圧縮形式を指定して DS を送信すると、それを受け取る DSM は圧縮形式のみを持った DS として保存する。そして asClass メソッドが呼ばれて初めて、メソッド内で解凍して cast が行われ DS が複数の表現を同時に持つようになる。これにより DS の表現を必要になったときに作成できるため、プログラマはどんな形式で DS を受け取っても DS を編集可能な形式として扱うことができる。また、複数表現は必要なときにしか作成されないため、メモリ使用量も必要最低限に抑えることができる。

5.3 Alice の通信プロトコルの変更

5.2 で述べたように、Remote から put されたデータは必ずシリアル化されており byteArray で表現される。しかし、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別がつかなくなった。

そこで、Alice の通信におけるヘッダにあたる CommandMessage.class に圧縮状態を表すフラグを追加した。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。これにより、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。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。

計測用のヘッダは以下の要素で構成されている。

表 1 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

time にはパケットの送信時刻を、dataSize に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 MessagePack 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。

到達時間の計算方法は、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。

実験結果

3 段目の測定結果の散布図を示す (図 6 ~ 8)。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。

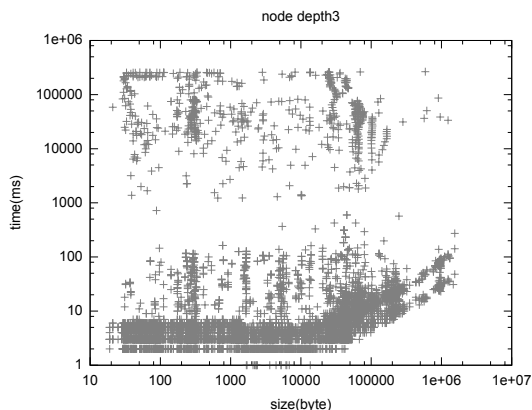


図 6 TreeVNC の測定結果

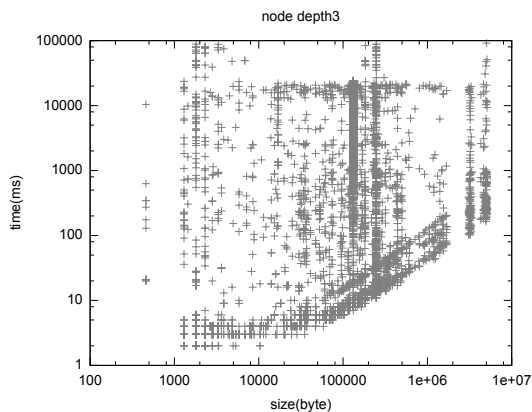


図 7 AliceVNC(圧縮・転送機能なし) の測定結果

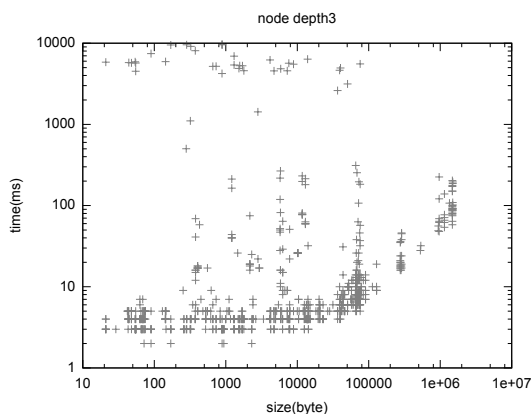


図 8 AliceVNC(圧縮・転送機能あり) の測定結果

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。

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 2 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 2 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 [7] 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 3 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。

TreeVNC で最高値を出した TreeRF-BProto.class は全てプログラマが記述したコードであり、データの待ち合わせのためのタイマー処理や通信処理、画面データの圧縮処理などの複数のスレッド処理が集中した複雑なコードになっている。これを Alice で記述した場合、データの待ち合わせは CS が行うためプログラマがデータの不整合を気にする必要はなく、また通信処理や圧縮処理も Meta Computation が提供するためコードが複雑になることはない。

AliceVNC で複雑度の最高値を出した 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 3 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

Aliceが採用しているCS/DSのプログラミングモデルやMeta Computationの特性を明確にするため、他言語・フレームワークとの類似点・相違点を比較した。

Erlang

並列指向プログラミング言語Erlang[8]は、プロセスと呼ばれるid付きの独立したタスクに対して、データをメッセージでやりとりする。タスクをプロセスという細かい単位に分割して並列に動かす点や、メモリロックの仕組みを必要としない点はAliceと同様である。

しかしErlangでは分散環境の構築等はすべてプログラマ自身が記述しなければならない。Aliceでは分散環境の構築はTopology ManagerなどのMeta Computationが行うためプログラマはトポロジーを指定するだけで良い。また、Erlangでは複数のデータの待ち合わせのための再帰処理も自分で書かないといけない。一方、Aliceのプログラミング手法はCSが必要なデータが全て揃うまで待ち合わせを行うためその必要はない。

Akka

Akka[9]はScala・Java向けの並列分散処理フレームワークである。アクターモデルを採用しており、アクターと呼ばれるアドレスを持ったタスクに、データをメッセージでやりとりする点がErlangと似ている。AkkaもErlangもプロセス/アクターに直接データをやりとりする。データには名前がないため、メッセージを受け取ったあとにその内容を確認した上で次にどう振る舞うかを判断する。一方Aliceでは、DSをCSに直接やりとりはせず、keyを指定してDSMにputする。また、DSをtakeするときもkeyを指定して取り出すためどんなデータが入っているかを確認する必要がなく、扱い易い。

Akkaの特徴として、メッセージを送りたいプロ

セスのアドレスを知っていればアクターがどのマシン上にあるかを意識せずにプログラミングできるという点がある。逆にAliceほどのRemote DSMに対してやり取りをするかを考慮するが、CSがOutputしたDSを次にどのCSに渡すかを意識する必要がない。この点はアクターモデルとCS/DSモデルのパラダイムの違いと言える。

一方AliceとAkkaは提供されるAPIという点で類似している。AkkaのメッセージAPIでは、メッセージを送るtellメソッドと、メッセージを送って返信を待つaskメソッドが用意されている。これはAliceのDataSegmentAPIのput/takeメソッドに対応している。

また、Akkaのもう一つの特徴として、アクターで親子関係を構成できる点がある。分散通信部分の子アクターに分離し、親アクターは子アクターのExceptionが発生した時に再起動や終了といった処理を指定できる。さらにRouterという子アクターへのメッセージの流れを制御するアクターや、Dispatcherというアクターへのスレッドの割当を管理する機能をAkkaが提供している。このように処理を階層化し複雑な処理をフレームワーク側が提供する仕組みはAliceのMeta Computationと共通している。相違点としては、AliceのMeta DSのようにデータの多態性を実現する機能はAkkaにはない。

7. まとめ

並列分散フレームワークAliceでは、スケラブルかつ信頼性の高いプログラムを記述する環境を実現するため、CS/DSの計算モデルとMeta Computationによる実装の階層化を採用している。Aliceが実用的な分散アプリケーションを記述するために必要なMeta Computationとして、多態性を持つデータを扱う機能や無駄なコピーなくデータを転送する機能を実装した。そしてMeta Computationを用いて分散アプリケーションTreeVNCをAlice上で実装し性能評価を行った。その結果、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 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。

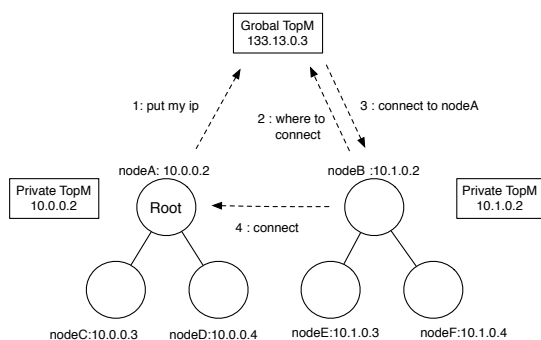


図9 複数の Topology Manager で NAT 超えを実現

各ネットワークごとに Topology 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 Topology Manager は今までどおりネットワーク内に木を構築・管理する。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、AliceVNC の仕様の変更を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。

また、現在の Alice はネットワーク通信におい

てセキュリティをサポートしていない。しかし、圧縮機能と同様に、暗号化形式を扱う Meta Computation を追加すれば、プログラマが記述する 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できる。

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] Yu SUGIMOTO and Shinji KONO: 分散フレームワーク Alice の DataSegment の更新に関する改良, 情報処理学会システムソフトウェアとオペレーティング・システム研究会 (OS) (2013).
- [3] Yu TANINARI, Nobuyasu OSHIRO and Shinji KONO: VNC を用いた授業用画面共有システムの設計・開発, 情報処理学会システムソフトウェアとオペレーティング・システム研究会 (OS) (2012).
- [4] : Dot Language, <http://www.graphviz.org/>.
- [5] : TightVNC Software, <http://www.tightvnc.com>.
- [6] : MessagePack, <http://msgpack.org/>.
- [7] McCABE, T. J.: A Complexity Measure, *IEEE TRANSACTIONS ON SOFTWARE ENGINEERING VOL. SE-2, NO.4* (1976).
- [8] : Erlang, <http://www.erlang.org/>.
- [9] Lockney, T. and Tay, R.: Developing an Akka Edge, *Bleeding Edge Press* (2014).