分散フレームワーク Alice の圧縮機能

照屋 のぞみ[†1]　杉　本　　優[†2]　河　野　真　治[†3]

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、Alice を用いた水族館の例題、分散データベース Jungle、木構造画面共有システム AliceVNC によって確認された。しかし、AliceVNC を作成するには、通信時に Data Segment を圧縮形式で扱える機能が必要である。本研究では、Data Segment に Object 型、MessagePack を使った ByteArray 型、圧縮された ByteArray 型の３つの表現を同時に持つメタ計算の設計と実装を行うことで、Data Segment の多態性を用いた圧縮機能を実現した。

Nozomi TERUYA,[†1] Yu SUGIMOTO[†2] and Shinji KONO [†3]

Alice is a framework for distributed programming, which uses Data Segment and Code Segment as programming units. We checked Alice has an ability to write distributed program using aquarium example, distributed database Jungle and screen sharing system AliceVNC.
In this paper, we add Data Segment compression on Alice. These representations are combine with the Data Segment using Meta Data Segment. In this way, Alice Data Segment supports polymorphism of the implementations.

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 という単位で分割して記述する並列分散フレームワーク Alice の開発を行っている。当研究室の先行研究である Federated Linda は、タプルという ID で番号付けられたデータの集合を相互接続された複数のタプルスペース（LindaServer）に出し入れするプログラミングモデルである。並列指向プログラミング言語 Erlang では、プロセスと呼ばれる独立性を備えたスレッドに Pid という識別子が対応しており、Pid を指定してメッセージを送受信する手法を用いて並列分散環境を実現している。これら２つは分散環境の構築等の処理は全てユーザ側のプログラムに記述しなければいけなかった。

一方、Alice では String 型の key に対応する Queue にデータが入っており、タスクは key を指定して必要なデータのみを出し入れするモデルを採用している。そのためタスクとデータの依存関係を分かりやすく記述でき、依存しない部分の並列実行が行える。さらに、Alice では Code Segment を Computation と Meta Computation に分割して考え、分散環境の構築に必要な処理を Meta Computation として提供することで、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している。

先行研究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。だが、実用的な分散プログラムを作成するためには、受け取ったデータをそのまま転送したい場合や圧縮されたデータ形式で通信を行いたい場合がある。

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を Meta Computation として実装した。プログラムに Alice の制御を行うメタプログラムを記述することにより、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る。そして、データの多態性を実現し、扱いたいデータの状態に合わせて DataSegmentManager を切り替えることで、ノード間通信における自由度の向上を図った。

†1 琉球大学工学部情報工学科
Information Engineering, University of the Ryukyus.
†2 琉球大学大学院理工学研究科情報工学専攻
Interdisciplinary Information Engineering, Graduate School of Engineering and Science, University of the Ryukyus.
†3 琉球大学工学部情報工学科
Information Engineering, University of the Ryukyus.

2. 分散フレームワーク Alice の概要

[Data Segment と Code Segment]

Alice はデータを Data Segment、(以下 DS) タスクをと Code Segment (以下 CS) という単位に分割してプログラミングを行う。DS は Alice が内部にもつデータベースによって管理されている。DS に対応する一意の key が設定されており、その key を用いてデータベースを操作する。

CS は実行に必要な DS が揃うと実行されるという性質を持ち、入力された DS に応じた結果が出力される。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ため、並列度を上げるためには CS の処理内容を細かく分割して依存するデータを少なくするのが望ましい。

[Data Segment]

Alice はデータを分割して記述する。その分割されたデータを DS と呼ぶ。Java の実装では MessagePack で特定のオブジェクトにマッピングされ、マッピングされたクラスを通してアクセスされる。

CS の実行において DS は占有されるため、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。

[Data Segment Manager]

DS は実際には queue に保存される。queue には対になる key が存在し、key の数だけ queue が存在する。この key を指定して DS の保存、取得を行う。queue の集合体はデータベースとして捉えられる。このデータベースを Alice では DS Manager (以下 DSM) と呼ぶ。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 は他のノードの Local DSM の proxy であり、接続しているノードの数だけ存在する。(図 1) Remote DSM に対して書き込むと対応するノードの Local DSM に書き込まれる。

[Data Segment API]

以下の Data Segment API を用いてデータベースにアクセスする。put と update は DS を追加する際に、peek と take は DS を取得する際に使用する。

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

DS を queue に追加するための API である。第一引数で指定した DSM の中の、第二引数に対応する queue に対して DS を追加している。

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

update も queue に追加するための API である。put との違いは、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。

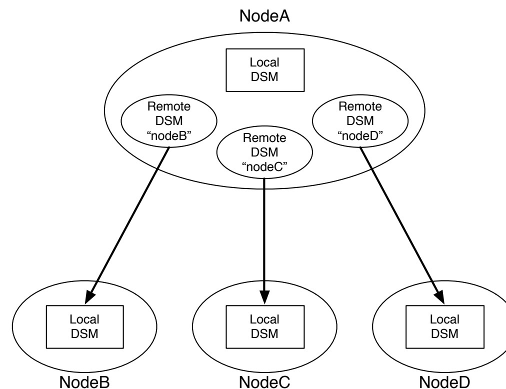

図 1 Remote DSM は他のノードの Local DSM の proxy

- `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。

[Data Segment の表現]

DS の表現には MessagePack for Java 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 enQueue 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 enQueue される。

[Code Segment]

Alice 上で実行されるタスクの単位が CS である。ユーザーは CS を組み合わせることでプログラミングを行う。CS をユーザーが記述する際に、内部で使用する DS の作成を記述する。

Input DS と Output DS は CS に用意されている API を用いて作成する。Input DS は、Local か Remote か、また key を指定する必要がある。CS は、記述した Input DS が全て揃うと Thread pool に送られ、実行される。

Output DS も Local か Remote か、また key を指定する必要がある。Input の場合は setKey を呼ぶ際、Output の場合は put(または update) の際にノードと key の指定を行っている。しかし、どの時点でノー

ドと key の指定を行えばよいか、どのような API を用意するべきかは、議論の余地がある。

[Code Segment の記述方法]

CS をユーザーが記述する際には CS 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 API を利用する事ができる。

```java
public class StartCodeSegment extends
    CodeSegment {

    @Override
    public void run() {
        new TestCodeSegment();

        int count = 0;
        ods.update("local", "cnt", count);
    }

}
```

Code 1　StartCodeSegment の例

```java
public class TestCodeSegment extends
    CodeSegment {
    private Receiver input1 = ids.create(
        CommandType.TAKE);

    public TestCodeSegment() {
        input1.setKey("local", "cnt");
    }

    @Override
    public void run() {
        int count = input1.asInteger();
        System.out.println("data = " + count);

        if (count == 10)
            System.exit(0);

        new TestCodeSegment();
        ods.update("local", "cnt", ++count);
    }
}
```

Code 2　CodeSegment の例

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 DSM を通して Local DSM に対して DS を put している。Output DSM は CS の `ods` というフィールドを用いてアクセスする。Output DSM 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 update が行われると TestCodeSegment は実行される。

ソースコード 2 は、0 から 10 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 DSM がもつ create メソッド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 DSM は CS の `ids` というフィールドを用いてアクセスする。

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） TestCodeSegment が持つフィールド変数 Receiver input1 の定義が行われる。
（2） 次に CS のコンストラクタが呼ばれ、CS が持つフィールド変数の定義と初期化が行われる。
（3） `ids.create(CommandType.TAKE)` が行われ、input1 の初期化が行われる。
（4） 最後に TestCodeSegment のコンストラクタの 5 行目が実行される。

5 行目は Input DSM 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 メソッドにより、どの DSM のある 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 10 行目で取得された DS を Integer 型に変換して count に代入している。16 行目で もう一度 TestCodeSegment の CS が作られる。17 行目で count の値をインクリメントして Local DSM に値を追加する。13 行目が終了条件であり、count の値が 10 になれば終了する。

[Computation と Meta Computation]

Alice の Computation は、key で指し示される DS を待ち合わせて CS を実行させると定義できる。それに対して、Alice の Meta Computation は、Alice の Computation を支えている Computation のプログラミングと定義できる。

例えば、トポロジーを指定する API は Meta Computation である。Alice が動作するためにはトポロジーを決める必要がある。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。トポロジーが決定するとそのトポロジーを構成する計算が行われる。トポロジーを指定する API はその構成の計算をプログラミングして変更するものである。他にも再接続の動作を決める API や切断時の動作を決める API は Meta Computation である。

プログラマーは CS を記述する際にトポロジーや切断、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する。そして、切断や再接続が起こった場合の処理を記述し Meta Computation で指定する。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、シンプルなプログラムを記述できる。

[Meta Data Segment]

DS は、アプリケーションに管理されているデータのことである。アプリケーションを構成する CS によってその値は変更される。それに対して Meta DS は、分散フレームワーク Alice が管理しているデータである。Alice を構成する CS によってのみ、その値は変更される。一部の Meta DS はアプリケーションに利用することができる。

例えば、"start" という key をもつ Meta DS は、ノードが Start CS を実行可能かどうかの状態を表す。他にも"_CLIST" という key では、利用可能な Remote DS の一覧が管理されている。ユーザーはこの一覧にある名前を指定することで、動的に DS の伝搬などを行うことができる。

また、Input DS に付随しているものもある。Input DS は CS 内部で Receiver という入れ物に格納される。ユーザーは、Receiver に対して操作することで DS を入手できる。この Receiver には、from というフィールドがあり、この DS を誰が put したという情報が入っている。この情報をデータの伝搬する際に利用することで、DS を put したノードに送り返すことを防ぐことができる。

Meta DS は DS 同様に DS API を用いて取得できる。

[Meta Code Segment]

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。

3. AliceVNC

当研究室では授業向け画面共有システム TreeVNC の開発を行っている。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 (図 2)。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 3)。

Alice が実用的なアプリケーションを記述する能力をもつことを確認するために、TreeVNC を Alice を用いて実装した AliceVNC の作成を行った。

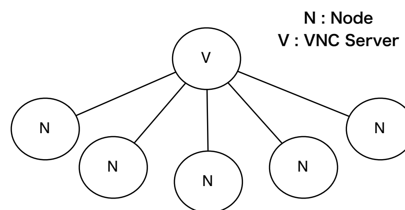

図 2 VNC の構造

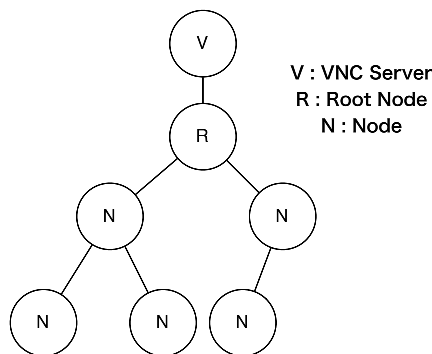

図 3 TreeVNC, AliceVNC の構造

4. 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。

[flip 機能]

Data Segment API の put、update を呼ぶと Output Data Segment が毎回新しく作成され、出力するデータのコピーが行われる。しかし、Input Data Segment として取得したデータをそのまま子ノードに Output Data Segment として出力する場合、コピーを行なうのは無駄である。

そこで、Input Data Segment と Output Data Segment を交換する機能を flip 機能として実装した。ソースコード 4 のように Input Data Segment である Receiver をコピーせずに flip メソッドに引数として渡すことで、コピー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 flip 機能が有用である。

```
public void flip(Receiver receiver) {
    DataSegment.getLocal().put(receiver.key,
            receiver.getReceiveData(), null);
}
```

Code 3　Alice における flip

```
public class SortPhase extends CodeSegment{
        private Receiver info = ids.create(
                CommandType.PEEK);

    public SortPhase(String key){
        info.setKey(key);
    }

    @Override
        public void run() {
                DataList list1 = info.asClass(
                        DataList.class);
        Sort.quickSort(list1);
        ods.flip(info);
    }
}
```

Code 4　flip の使用例

[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 DS Manager 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 5 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。

```
public class ReceiveData {
    private Object val = null;
    private byte[] messagePack = null;
    private byte[] zMessagePack = null;
}
```

Code 5　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 の追加した。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 6 は Remote から DS を take しインクリメントして Local に put することを 10 回繰り返す例題である。これを DS を圧縮形式で行いたい場合、ソースコード 7 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。

```
public class RemoteIncrement extends
      CodeSegment {

    public Receiver num = ids.create(
          CommandType.TAKE);

    @Override
    public void run() {
        int num = this.num.asInteger();
        System.out.println("[CodeSegment]␣" +
              num++);
        if (num == 10) System.exit(0);

        RemoteIncrement cs = new
              RemoteIncrement();
        cs.num.setKey("remote", "num");

        ods.put("local", "num", num);
    }

}
```

Code 6　通常の 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 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 8) 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。

[Alice の通信プロトコルの変更]

2 章 [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、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。

```
public class RemoteIncrement extends
    CodeSegment {

  public Receiver num = ids.create(
      CommandType.TAKE);

  @Override
  public void run() {
    int num = this.num.asInteger();
    System.out.println("[CodeSegment]␣" +
        num++);
    if (num == 10) System.exit(0);

    RemoteIncrement cs = new
        RemoteIncrement();
    cs.num.setKey("compressedremote", "
        num");

    ods.put("compressedlocal", "num", num
        );
  }

}
```
Code 7 圧縮した DS を扱う CS の例

```
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 8 asClass の処理

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 9) 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。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。

```
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 9 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

5. 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。

今後の課題としては、圧縮機能を AliceVNC で用いることで有効性を測る必要がある。また、Alice の Meta Computation に Proxy 機能を実装することで、TreeVNC では実装が困難であった NAT 越えの機能を提供できると期待される。

参 考 文 献

1) Yu SUGIMOTO and Shinji KONO: 分散フレームワーク Alice 上の Meta Computation と応用，琉球大学工学部情報工学科平成 26 年度学位論文 (修士) (2014).
2) Kazuki AKAMINE and Shinji KONO: 分散フレームワーク Alice の提案と実装，琉球大学工学部情報工学科平成 24 年度学位論文 (修士) (2012).
3) Nobuyasu OSHIRO, Yu SUGIMOTO, Shinji KONO and Tatsumi NAGAYAMA: Data Segment の分散データベースへの応用，日本ソフトウェア科学会 (2013).
4) 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 (2013).
5) 柏原正三：プログラミング言語 Erlang 入門，アスキー (2007).