# 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、Alice を用いた水族館の例題等で確認された。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、通信時に Data Segment を圧縮形式で扱う機能や Data Segment を他ノードへそのまま転送する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 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。

**キーワード**：プログラミング・シンポジウム，冬，予稿集

## 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。

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。

## 2. 分散フレームワーク Alice の概要

### 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 DS は InputDS、CS が計算を行った後に出力

[1] 情報処理学会
[2] プログラミング・シンポジウム幹事団
[a)] dpop@cr.ie.u-ryukyu.ac.jp
[b)] kono@ie.u-ryukyu.ac.jp

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。実際には Alice は Java で実装されており、DS は JavaObject で CS は RannableThread 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。

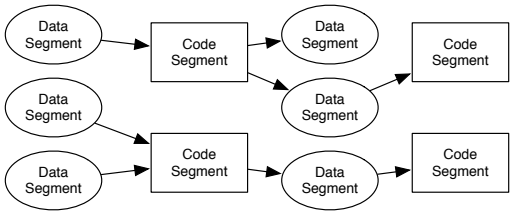


**図 1** CodeSegment の依存関係

### 2.2 DataSegmentManager

DS は Alice が内部にもつデータベースによって管理されており、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。他ノードの Local DSM に書き込みたい場合は Remote DSM に対して書き込む。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 ）。

### 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 を追加する際に用いる。peek と take は Input DS API と呼ばれ、DS を取得する際に使用する。

- `void put(String managerKey, String`

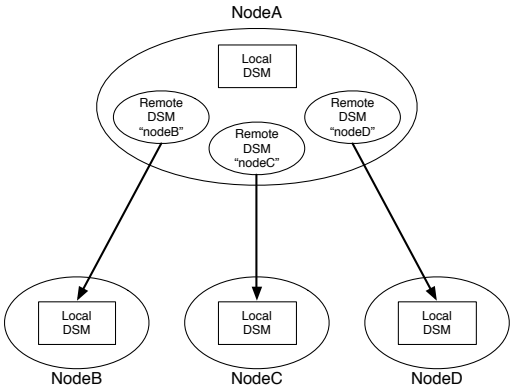


**図 2** Remote DSM は他のノードの Local DSM の proxy

`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 DSM 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 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。

### 2.5 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 Data Segment API を利用する事ができる。

```
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

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。

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 が Meta Computation として分散環境の構築等の機能を提供することで、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。

## 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 ? 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 (図 3 )。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 )。

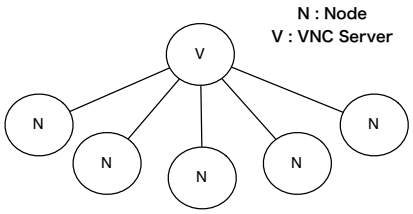


**図 3** VNC の構造

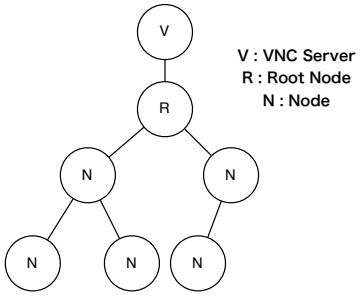


**図 4** TreeVNC, AliceVNC の構造

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。

## 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 3 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 3** 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 4 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 5 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。

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 6 ) 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

```
1  public class RemoteIncrement extends
       CodeSegment {
2
3      public Receiver num = ids.create(
           CommandType.TAKE);
4
5      @Override
6      public void run() {
7          int num = this.num.asInteger();
8          System.out.println("[CodeSegment] "
               + num++);
9          if (num == 10) System.exit(0);
10
11         RemoteIncrement cs = new
               RemoteIncrement();
12         cs.num.setKey("remote", "num");
13
14         ods.put("local", "num", num);
15     }
16
17 }
```

**Code 4** 通常の DS を扱う CS の例

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 7 ) 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。

```
1  public class RemoteIncrement extends
       CodeSegment {
2
3      public Receiver num = ids.create(
           CommandType.TAKE);
4
5      @Override
6      public void run() {
7          int num = this.num.asInteger();
8          System.out.println("[CodeSegment] "
               + num++);
9          if (num == 10) System.exit(0);
10
11         RemoteIncrement cs = new
               RemoteIncrement();
12         cs.num.setKey("compressedremote",
               "num");
13
14         ods.put("compressedlocal", "num",
               num);
15     }
16
17 }
```

**Code 5**　圧縮した DS を扱う CS の例

```
1  public <T> T asClass(Class<T> clazz) {
2      if (val != null) {
3          return (T) val;
4      }
5
6      if (zMessagePack != null &&
           messagePack == null) {
7          messagePack = unzip(zMessagePack,
               dataSize);
8      }
9
10     return packer.read(messagePack, clazz);
11 }
```

**Code 6**　asClass の処理

## 6. 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

**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 |

```
1 public class CommandMessage {
2     public int type;
3     public int seq;
4     public String key;
5     public boolean quickFlag = false;
6     public boolean serialized = false;
7     public boolean compressed = false;
8     public int dataSize = 0;
9 }
```

**Code 7** CommandMessage

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。

今後の課題としては、圧縮機能を AliceVNC で用いることで Meta Computation の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] MIWA OSHIRO, and Shinji KONO:授業やゼミ向け画面配信システム 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).

[2] MessagePack, http://msgpack.org/ tightVNC TightVNC, http://www.tightvnc.com/

[3] 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).

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

[5]