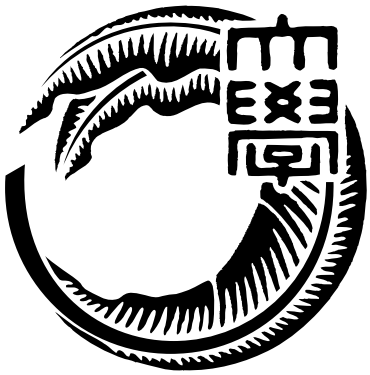
平成27年度 卒業論文

# 分散フレームワークAliceのMeta Data Segment



琉球大学工学部情報工学科

e125769 照屋 のぞみ

指導教員 河野 真治

# 目 次

# 第1章　研究背景と目的

近年、スマートフォンやタブレット端末の普及率が増加している。それに伴いインターネット利用者数も増加しており、ネットワーク上のサービスの利用者の増加は必至である。従って、サービスには、信頼性とスケーラビリティーが要求される。ここでいう信頼性とは、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さす。またスケーラビリティーとは、サービスの利用者が増大した場合、メモリ等のリソースを追加するだけでサービスを維持できる性能をさす。しかし、これらをもつ分散プログラムをユーザーが一から記述することは容易ではない。

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。

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。プログラマは仕様を大きく変更することなくプログラムの挙動が変えられるため、変更前の信頼性を保ったまま拡張ができる。

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。差分データは Alice の Data Segement に対応するため、Alice の Meta Data Segment として圧縮機能が必要なる。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。

# 第2章　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 Output DS 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 2.1 ）。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。

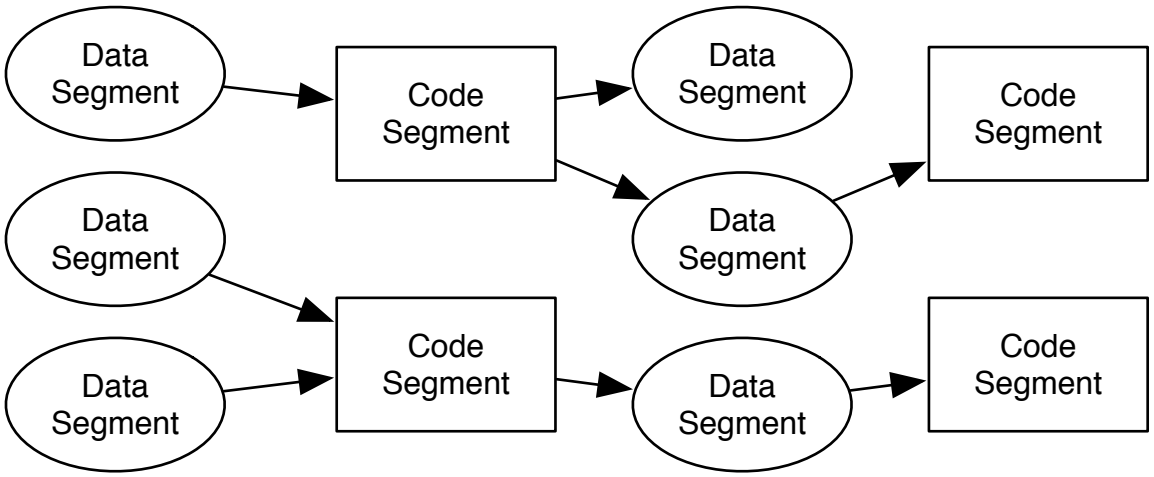



図 2.1: CodeSegment の依存関係

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 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 2.2 ）。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。

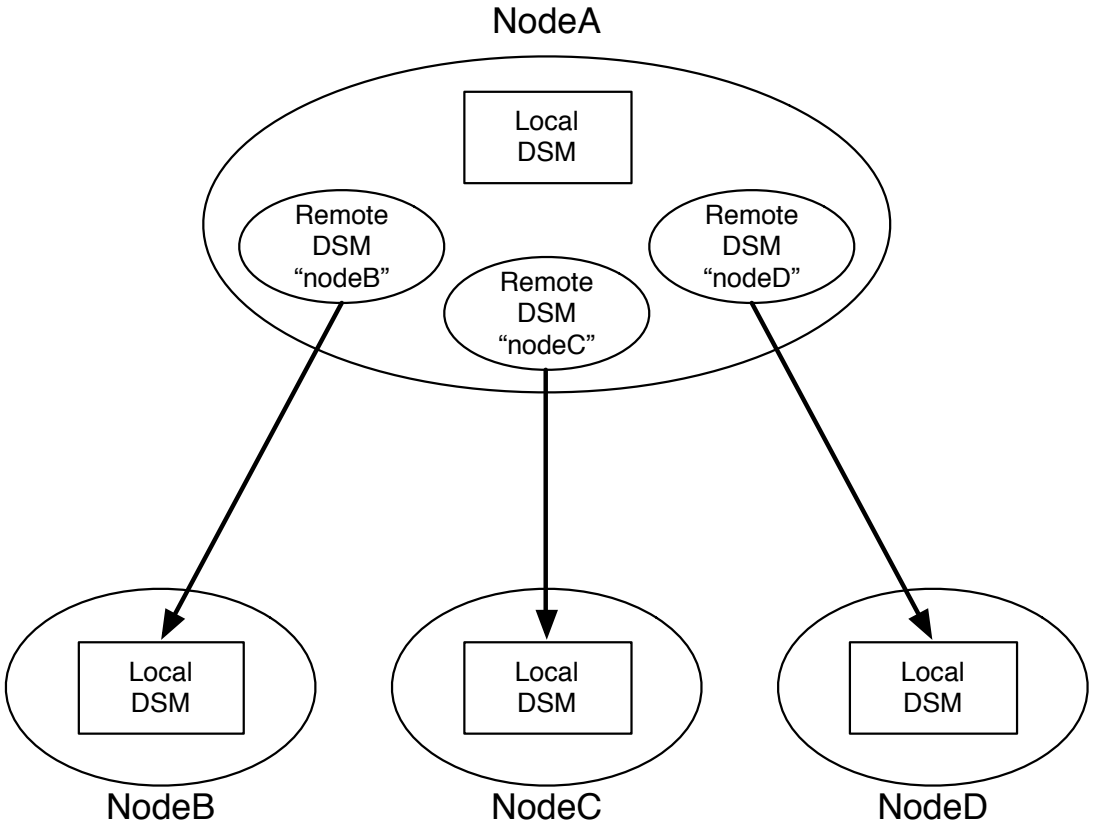



図 2.2: Remote DSM は他のノードの Local DSM の proxy

## 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、flip 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 の中にある DS の個数は変わらない。

- `void flip(String managerKey, String key, Receiver val)`

flip は DS の転送用の API である。取得した DS に対して何もせずに別の Key に対し保存を行いたい場合、一旦値を取り出すのは無駄である。flip は DS を受け取った形式のまま転送するため無駄なコピーなく DS の保存ができる。

- `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 2.1 , 2.2)。

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 2.1 ) 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 2.1: 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 2.2: CodeSegment の例

ソースコード 2.1 は、5 行目で次に実行させたい CS（ソースコード 2.2 ）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` と `flip` 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 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/flip 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 2.2 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)`

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。

# 第3章　AliceのMeta Computation

## 3.1　Computetion と 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を記述できる。

現在 Alice には、トポロジーの構成・管理機能、ノードの生存確認機能、ノードの切断・再接続時の処理管理機能などの Meta Computation が用意されている。

## 3.2 Meta Code Segment と Meta Data Segment

Alice 提供する Meta Computation も CS/DS により実現される。CS の処理を支える処理を Meta CS、Meta CS に管理される Meta DS として考える。

図 3.1 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。

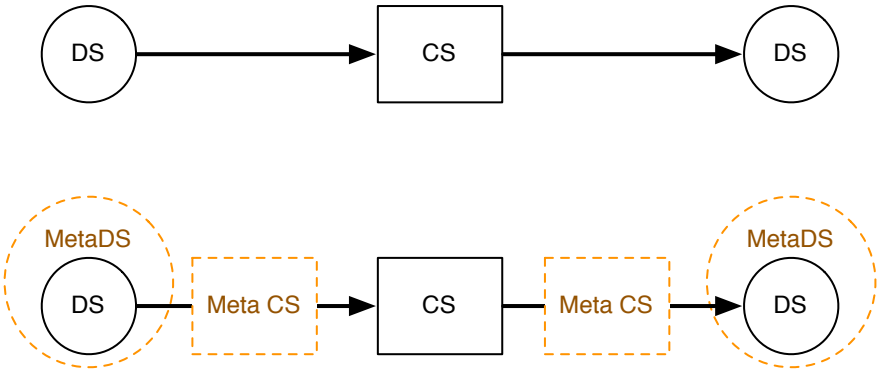



図 3.1: CS/DS の間に MetaCS/MetaDS が接続される

Meta DS は基本的に Alice を構成する CS によってのみ管理され、プログラマは認識できない。しかし一部の Meta DS はプログラマがアプリケーションに利用することもできる。例えば、トポロジー管理の Meta Computation などで使われる”_CLIST” という Meta DS には、利用可能な Remote DSM の情報が管理されている。プログラマはこの Meta DS を取得し Remote DSM 名を指定することで、動的に DS の伝搬などを行うことができる。

## 3.3 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。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 3.1 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。

```
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 3.1: トポロジーファイルの例

DOT Language ファイルは dot 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3.2)。また、トポロジーファイルでlavel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。

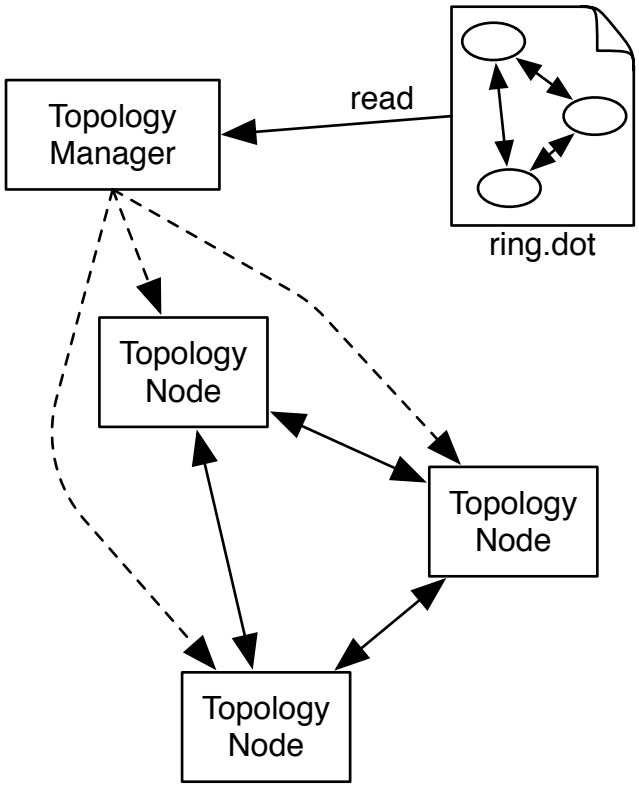



図 3.2: Topology Managerが記述に従いトポロジーを構成

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している。

## 3.4 Keep Alive

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。

## 3.5 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。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。

# 第4章　AliceのTreeVNCへの応用

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 [5]。

## 4.1　TreeVNC

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 [6] 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 4.1)。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 4.2)。

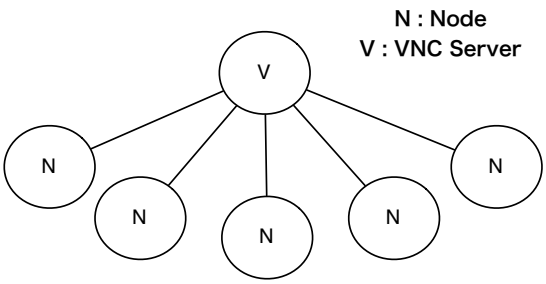



図 4.1: 通常の VNC の構造

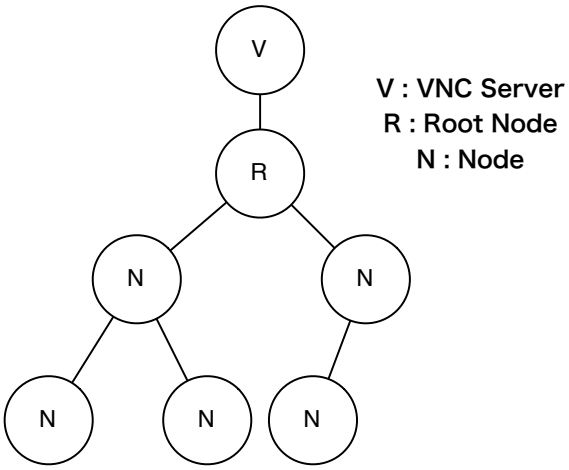



図 4.2: TreeVNC の構造

## 4.2 AliceVNC

図 4.3 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。

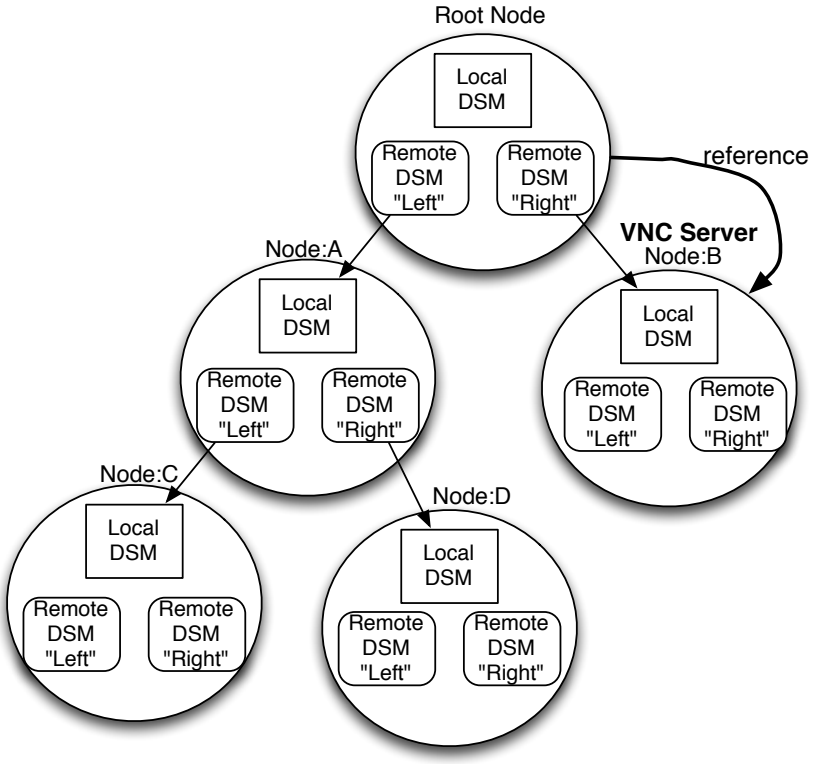



図 4.3: AliceVNC の構造

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。

# 第5章　圧縮のMeta Computationの追加

## 5.1　圧縮のMeta Data Segment

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、DSを複数作るのではなく1つのDS内で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[7]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 5.1 は変更前の ReceiveData.class である。変更前は DS の表現は 1 つでフラグによって、Local DSM に put する (1) の形式と Remote DSM に put する (2) の形式を判別していた。しかしこの実装では圧縮形式と非圧縮形式を同時に持つことができないため、AliceVNC では解凍・再圧縮が必要になってしまう。

変更後の実装ではソースコード 5.1 のようになっている。`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がなくなる。プログラマ側がから見れば 1 つの DS であり直接これら 3 つの表現を操作することはないため、これらは Alice の Meta Data Segment と言える。

```
1 public class ReceiveData {
2     private Object val;
3     private boolean serialized = false;
4     private boolean byteArray = false;
5
6     ...
7 }
```

Code 5.1: 変更前のデータ表現

```
1 public class ReceiveData {
2     private Object val = null;
3     private byte[] messagePack = null;
4     private byte[] zMessagePack = null;
5
6     ...
7 }
```

Code 5.2: 変更後のデータ表現

## 5.2 圧縮の Meta Code Segment

圧縮表現を持つ 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。Local Compressed DSM は表現の判別や変換を行うだけで、操作する対象は Local DSM と同じ DS を指すため、DS の管理が別々になるわけではない。

ソースコード 5.3 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 5.4 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 が走り圧縮形式に変換された DS となって通信が行われる。

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

Code 5.3: 通常の DS を扱う CS の例

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

Code 5.4: 圧縮した DS を扱う CS の例

この機能は先述の Meta Data Segment を扱う Meta Code Segment と言える。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 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

4.2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(ソースコード5.5表5.1)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。

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

Code 5.5: CommandMessage

表 5.1: 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 |
| 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 |

# 第6章　DSとMetaDSのKeyの領域分け

DS と Meta DS は同じ Data Segment Manager で管理されており、同じ API を利用してアクセスされる。そのため誰でも Meta DS の変更が可能になってしまっている。プログラマが定義しようとした Key が偶然 Alice の Meta DS の Key と衝突してしまった場合、ユーザーが意図した動作にならずエラーとなる状況は充分にありえる。しかも Meta DS はプログラマ側からは認識できないため衝突の認識やエラーの解決がしづらくなっている。このような事態を避けるためにも、DS の領域分けが必要である。

いままでは LocalDataSegmentManager.class は 1 つだけだったが、もう 1 つ追加して Meta Local DSM として登録した (ソースコード 6.1)。これで managerKey=”local” ならば Local DSM に、managerKey=”metaLocal” ならば Meta Local DSM に対して操作できる。

```
1  public class DataSegment {
2
3      private LocalDataSegmentManager local = new LocalDataSegmentManager();
4      private LocalDataSegmentManager metaLocal = new LocalDataSegmentManager
           ();
5
6      private DataSegment() {
7          dataSegmentManagers.put(”local”, local);
8          dataSegmentManagers.put(”metalocal”, metaLocal);
9      }
10
11     ...
12 }
```

Code 6.1: LocalDSM の追加

CodeSegment.class を継承した MetaCodeSegment.class を追加した (ソースコード 6.2)。MetaCodeSegment は、ids と ods に対して CS か Meta CS かのフラグをセットする。これを継承したクラスは DS Manager を指定せずに API を呼び出すと自動的に MetaDSM に対して操作したことになる。

```
1  public class MetaCodeSegment extends CodeSegment {
2
3      public MetaCodeSegment(){
4          ids.setMeta();
5          ods.setMeta();
6      }
7
8      @Override
9      public void run() {
10
11     }
12 }
```

Code 6.2: MetaCodeSegment.class

Alice の Meta CS は今まで CS 同様 CodeSegment.class を継承して作られていたが、継承元を MetaCodeSegment.class に変更する (ソースコード 6.3 1 行目) だけでそれ以外は変更せずに DS の領域分けができる。

```
1  public class StartTopologyManager extends MetaCodeSegment {
2      TopologyManagerConfig conf;
3
4      public StartTopologyManager(TopologyManagerConfig conf) {
5          this.conf = conf;
6      }
7
8      @Override
9      public void run() {
10         new CheckComingHost();
11         ods.put("absCookieTable", new HashMap<String, String>());
12
13         ...
14
15     }
16 }
```

Code 6.3: MetaCodeSegment を継承

managerKey="metaLocal" を指定すればプログラマ側の CS からも Meta DS の操作ができるが、Meta DS に take をした場合、その DS を InputDS として指定している Meta CS が想定どおりに動作しないことが考えられる。そのため、take を行っても内部で peek しか行わないようにした。これによりプログラマが間違えて take で DS を取得しても DS が削除されることはないため、Meta Computation に干渉することはない。

# 第7章　評価と考察

## 7.1　圧縮の Meta Computation の評価

圧縮の Meta Computation が正しく機能しているかを確認するために通信時間の計測を行った。2 台の PC 間でお互いにデータを 100 回転送し合う。圧縮を指定したコードとしていないコードで 5 回計測した。

結果が表 7.1 である。データサイズから見ても圧縮に成功していることがわかる。通信においても、所要時間が 1/3 以下に抑えられていることから圧縮が効果的に作用している。TreeVNC と同等の性能をだすために有用な Meta Computation が実装できたと言える。

表 7.1: 圧縮機能の測定結果

| | 圧縮なし | 圧縮あり |
|---|---|---|
| データサイズ | 300KB | 91KB |
| 平均通信時間 | 6014ms | 1764ms |

## 7.2　TreeVNC と AliceVNC のメッセージ伝達速度の比較

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 [8]。

まず、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。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 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 7.2: 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 |

time にはパケットの送信時刻を、dataSize には圧縮された画面データのサイズを付けて送信する。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 7.1 〜 7.2）。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 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。

図から同様の傾向があり、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 Alice は TreeVNC と同等の処理性能を持つアプリケーションを実装するに十分な能力があることがわかる。

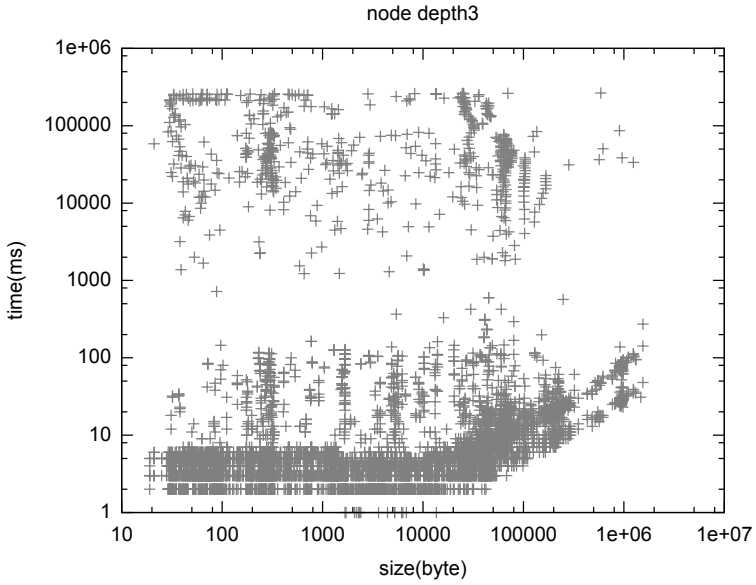



図 7.1: TreeVNC の測定結果

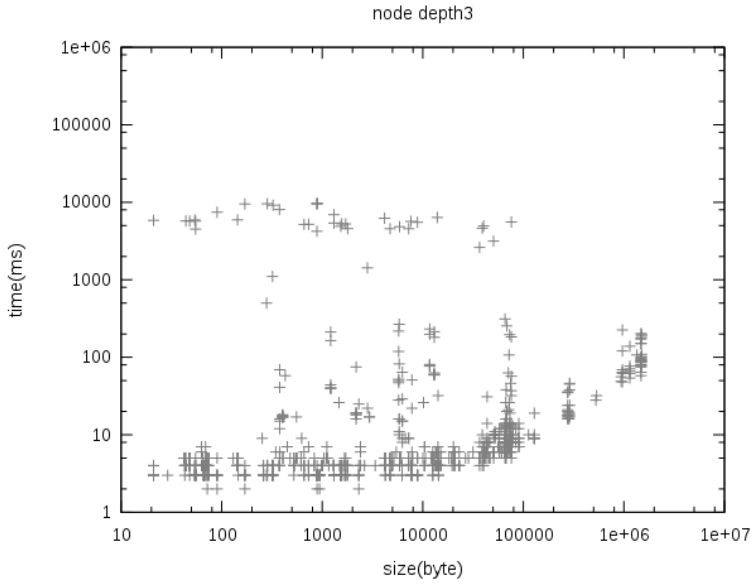



図 7.2: AliceVNC(圧縮・転送機能あり ) の測定結果

## 7.3 TreeVNC と AliceVNC のコード量比較

TreeVNC と AliceVNC のコード量を比較した表が表 7.3 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。

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

## 7.4 TreeVNC と AliceVNC のコードの複雑度比較

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 [9] 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 7.3 は TightVNC、TreeVNC、AliceVNC における循環的複雑度の比較である。

表 7.3: 複雑度の比較

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

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 TreeRFBProto.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。

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。

# 第8章　他言語等との比較

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。

## 8.1　MPICH

メッセージパッシング方式の標準規格である MPI の C++/FORTRAN の実装系が MPICH である。メッセージパッシングとは、並列分散処理におけるプロセス間通信の一形態で、分散メモリ同士の書き込みをメールのようにメッセージの送受信で行う方式である。

プロセスごとにランクという id が付与され、プロセスはコミュニケーターという通信単位でグループ化できる。コミュニケーターとランクをそれを指定してメッセージを送受信する。API は Sender/Receiver メソッドが用意されており、宛先情報、データの種類・サイズ、そしてプログラマがメッセージに対し任意に命名できるタグを指定して送受信を行う。Sender/Receiver は Alice の put/take に対応しており、コミュニケーターは DSM 名、タグは Key に対応している点が類似している。

しかし、MPICH は Alice の Meta Computation に対応するものはない。そのため分散環境の構築やデータの圧縮は全てプログラマ側が記述しなければならない。一方、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。

## 8.2　Erlang

アクターモデルの並列指向プログラミング言語 Erlang[11] 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、id を指定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。

Erlang もプロセス/アクターに直接データをやりとりする。MPICH にはメッセージにタグが付けられたが、Erlang にはデータには名前がない。そのため、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記述が必要となる。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。

そして、Erlang では静的な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ければならない。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。

また、MPICH 同様 Erlang にも Meta Computation に対応する部分がないため、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。

## 8.3 Akka

Akka[12] 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 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 はどの 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。

また、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 DataSegment API の put/take メソッドに対応している。

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。分散通信部分を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 のようにデータを分離し多態性を実現する機能は Akka にはない。例えば、データを圧縮して通信した場合は、用意されている compress/uncompress メソッドを使い圧縮・伸長のコードをプログラマが挟まなければならない。そのためコードを圧縮通信に変更したいときはメッセージの送信側と受信側を両方書き換えが必要になり、どちらか一方を書き忘れるとエラーとなる。一方 Alice では送信側が DSM 名に”compress”をつけるだけで他を変更しなくとも圧縮通信に切り替えられるので、コードの変更量が抑えられ、変更前の信頼性が保存される。

# 第9章　まとめ

並列分散フレームワーク Alice では、スケー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。Meta Computation は Meta CS / Meta DS に分けられ、それらが通常の処理の間に挟まれることでプログラマ側の記述する Computation の変更を抑えた挙動変更を可能にする。

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、Meta DS に複数の形式を同時に持たせ、DSM を切り替えることで多態性を持つデータを扱う機能を実装した。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。

また、Data Segment の Key の管理領域を分けたことで、DS と Meta DS の Key の衝突を避け、CS / DS のプログラミングスタイルにおける信頼性向上を図った。

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に成功した。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できていた。このことから Alice の Meta Computation が信頼性・拡張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。

# 第10章　今後の課題

## 10.1　AliceVNCのNAT超え通信の実装

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 10.1 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。

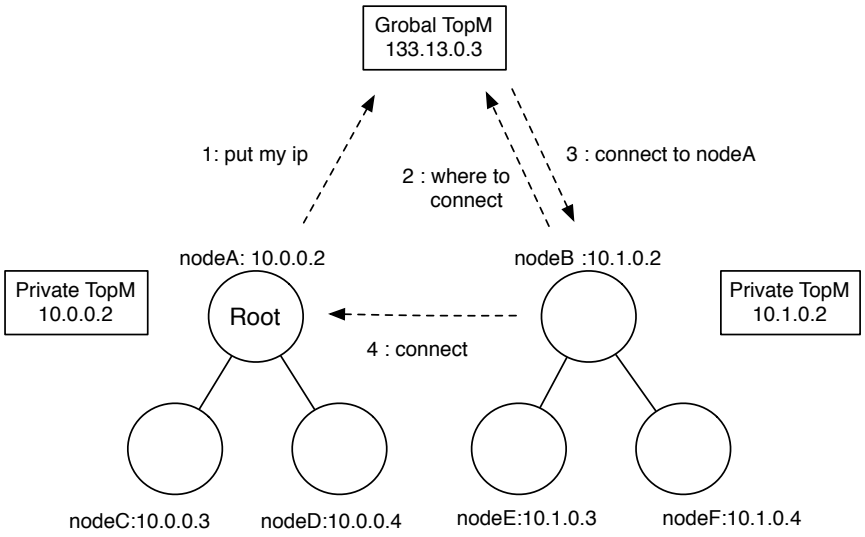



図 10.1: 複数の Topology Manager で NAT 超えを実現

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。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。

## 10.2 APIの再設計

2.4で示したように、DSを取得するときのAPIはpeek/takeが直接扱えず、create/setKeyを組み合わせてプログラミングしなければならない。この設計だとプログラマにとってわかりづらく、コンストラクタ内でtakeを行いたい場合はsetKeyを必ず最後に呼ばなければならない等の注意点がある。put/update/flipと同様に、peek/takeをそのまま呼べるように再設計する必要がある。

また、動的に複数のDSを取得する場合は、プログラマが末尾再帰処理を書かなければならない。MPICHのReceiverメソッドは引数でタグと個数を指定することで複数のメッセージの待ち合わせができる。複数DSを待ち合わせしたい場面は多いため、個数指定のできるAPIの追加も望ましい。

## 10.3 DSの型情報のマネジメント

Aliceでは型情報がないので、peek/takeする際にどんな型のデータが入っているのかがわからない。takeしたDSの型を確認したい場合には、そのDSをputしている部分を確認しなくてはならない。そのため、型情報をサポートする機能が必要である。

## 10.4 データの永続性の確保

現在のAliceは、On memoryであるためプロセスの終了とともにData Segmentは全て失われてしまう。

この問題を解決するためには、Data Segmentを他のKey Value Store等のシステムに保存し、永続性を確保する昼用がある。また、当研究室で開発しているJungle Database[13]のようにLogファイルとして出力することでも解決ができる。

## 10.5 Java以外での実装

Alice に Garbage Collection は必要ない。Alice では、すべての Data Segment は Key Value に格納され、実行時の Data Segment は Code Segment が active な時のみにメモリ上にある。この最大値を見積ることは、Active Task の量を見積もれば良い。したがって、Alice には Garbage Collection は必要ない。一方で、Key Value Store 上のデータは決して Garbage Collection の対象にならない。しかし、それは Garbage Collector には負荷をかけてしまう。つまり、Alice 自体は Java で実装するのには向いていない。

当研究室ではCode Segment/Data Segmentのプログラミング形式で記述する言語CbC(Continuation based C)[14]と、CbCを用いて記述されるGears OS[15]の開発を行っている。そのため、CbCを用いてGears OSの分散機構の一部としてAliceを再設計することが望ましい。

# 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] Yu SUGIMOTO, Nozomi TERUYA and Shinji KONO :分散フレームワーク Alice の圧縮機能, 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) (2015).

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

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

[8] Nozomi TERUYA and Shinji KONO :分散フレームワーク Alice の PC 画面配信システムへの応用, 第 57 回プログラミング・シンポジウム (2016).

[9] McCABE, T. J.: A Complexity Measure, *IEEE TRANSACTIONS ON SOFTWARE ENGINEERING VOL. SE-2, NO.4* (1976).

[10] : MPICH, `https://www.mpich.org/`.

[11] : Erlang, `http://www.erlang.org/`.

[12] Lockney, T. and Tay, R.: Developing an Akka Edge, *Bleeding Edge Press* (2014).

[13] Tatsuki KANAGAWA and Shinji KONO :非破壊的木構造データベース Jungle とその評価, 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)(2015).

[14] Kaito TOKUMORI and Shinji KONO :Implementing Continuation based language in LLVM and Clang, LOLA 2015 Kyoto (2015).

[15] Shohey KOKUBO, Tatsuki IHA and Shinji Kono :Monad に基づくメタ計算を基本とする Gears OS の設計, 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)(2015).

# 第11章　謝辞

本研究の遂行、また本論文の作成にあたり、ご多忙にも関わらず終始懇切なる御指導と御教授を賜わりました河野真治助教授に深く感謝したします。

そして、数々の貴重な御助言と技術的指導を戴いた杉本優さん、比嘉健太さん、伊波立樹さん、並びに並列信頼研究室の皆様に感謝いたします。

また、東京大学の横山大作教授をはじめ、OS 研究会、プログラミング・シンポジウムにおいて多くのフィードバックを頂いた先生方に感謝いたします。

本研究を遂行するにあたり参考にさせていただいた先行研究の Federated Linda, Cerium, TreeVNC の設計・実装に関わった全ての先輩方に感謝いたします。

最後に、日々の研究生活を支えてくださった米須智子さん、情報工学科の方々、そして家族に心より感謝いたします。

2016 年 2 月
照屋のぞみ