修士 (工学) 学位論文

Master's Thesis of Engineering

分散フレームワークChristieの設計

Design of Distributed framework Christie

2018年3月

March 2018

照屋 のぞみ

NOZOMI TERUYA

琉球大学

大学院理工学研究科

情報工学専攻

Information Engineering Course

Graduate School of Engineering and Science

University of the Ryukyus

本論文は、修士 (工学) の学位論文として適切であると認める。

論　文　審　査　会

印

(主　査)　　和田　知久

印

(副　査)　　長田　智和

印

(副　査)　　玉城　史朗

印

(副　査)　　河野　真治

要 旨

インターネットが広がり、IoT(Internet of Things) で通信するサービスが広く使われるようになり、分散計算の重要性は増している。分散計算では参加する台数に寄らずサービスの質を維持するスケーラビリティ、要求された仕様を満たす信頼性とともに、拡張性が要求される。特にハードウェアは日進月歩であり、同じサービスであっても、使用するハードウェアに適用するための拡張が必須となっている。

これまでに開発してきた並列分散フレームワーク Alice 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している。Alice での通信はノード毎の Data Segment のプールにキーを用いてお互いに書き込むという手法を用いている。Code Segment は必要とする Data Segment のキーを指定して取得 (take) または参照 (peek) し、それらがそろった時点で実行を並列に行う。Alice では通常の処理の間に Meta Computation という処理を挟むことで、コードを大きく変更せずに挙動変更を可能にしている。分散型画面共有システム TreeVNC などの実装によりその有効性が確認されている。特に Node の Topology を静的または動的にメタ計算として指定できる Topology Manager が有効であった。

しかし、Java で実装された Alice ではキーと取得モードの設定を Java のオブジェクトの作成される側 (constructor) で行ったり、作成側で行なったりする自由度があり、キーによる通信の見通しが良くないということが判明した。また、Alice は Data Segment や Code Segment の管理を大域変数的な Singleton で行なっており、複数の Alice のインスタンスを同一プロセス上で起動できない。これにより、単一プロセス内での分散計算のテストや、複数の Topology manager に接続する必要がある NAT 越えなどの Meta Computation を実装することができなかった。大域 Singleton は、その性質上、どこからでもアクセスできるために除去は困難であると判明した。さらに、Alice の直感的でない API を改善し、型の整合性を保証することで信頼性を向上させるべきだと考えた。

本研究では Alice で得られた知見を元に分散フレームワーク Christie の設計を行った。Christie では Segment の代わりに Gear とう名称を使用する。 Christie では Java の Annotation を用いてキーと取得モードの記述を行う。これによりキーで指定された Data Gear の型を明示することができ、Alice で必要だった実行時の型変換を隠すことができるようになった。また、複数の Code/Data Gear Manager を使用することで単一プロセスでのテストや NAT 越えなどの拡張が可能になった。本論文では、他の並列分散フレームワークである Akka や Hazelcast との比較も行う。

Abstract

The importance of distributed computers is increased as widely used Internet or IoT (Internet of Things). Distributed computations include scalability which keeps quality of service on larger number of nodes, reliability which assures the specification of the service, and expand-ability. If if the service itself is not changed, the hardware of platform shall be changed, which is necessary to adapt by the service.

We developed a distributed computation framework called Alice, which uses Data Segments and Code Segments as units of computation. Alice uses pools of Data Segments, whose Data Segments are accessed by keys from computation nodes. A Code Segments takes or peeks Data Segments by keys, and when all Data Segments are ready, the Code Segment is executed in a concurrent way.

In Alice besides Code Segment and Data Segments, Meta Code Segments and Meta Data Segments can be inserted. Using Meta Code Segments and Meta Data Segment, we can change the behavior without changing the normal level code. In the example of distributed screen sharing system called TreeVNC, usefulness of Meta computation of Alice is confirmed such as Data Segment Compression and failure node managements. A topology manger which controls topology of distributed node in both static and dynamic way is also useful meta computation in Alice.

Keys and access modes in Alice are described in Java implementations, which are sometimes put in the separated places, such as Java object constructor or other. This makes the communication protocol using keys untraceable. In Alice implementations, Code and Data Segment manager is implemented as a global singleton, as a result, we cannot use multiple instance of Alice in a process. So it is impossible to simulate distributed computation in a process. It is also difficult to implement communication over NAT (Network Address Translations), which have to handle two different networks at a node. Type correctness is not so clear in Alice, since dynamic cast is used. It better to define more intuitive APIs.

In this research, we designed a new distributed framework Christie from Alice experiences. Christie use name Gears instead of Segment which is sometimes confusing. A simple key and access mode description using Java Annotation is introduces. Also, multiple Data or Code Gear Managers are supported which makes possible to simulating distributed computation in a node or NAT handling.

Comparisons with other framework such as Akka or Hazelecast are also shown.

目 次

図 目 次

ソースコード目次

第1章　分散フレームワークへの要求事項

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、ネットワークサービスにはそれに対する処理能力が求められる。サーバの処理能力を増強するアプローチとして、スケールアップとスケールアウトがある。スケールアップはサーバそのものを増強することで処理能力を向上させる手法であり、スケールアウトは複数のサーバを接続することで処理能力を上げる手法である。スケールアウトは安価なサーバで実現でき、サーバが故障しても別のサーバでカバーできるため、システムを安定運用させやすく、現代のネットワークサービスの要求に合っていると言える。

スケールアウトのように複数のサーバで処理をまたぐ場合、サーバには分散プログラムが必要になる。分散プログラムとは、プログラムの個々の部分が複数のノード上で並列に実行され、各ノードがネットワークを介して互いに通信を行いながら全体の処理を進行する計算手法のことである。安定したネットワークサービスを提供するためには、分散プログラムに信頼性とスケーラビリティが要求される。

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指す。これには仕様を記述しやすさも含まれ、可読性が高いほどバグを抑えた信頼性が高いと言える。またスケーラビリティーとは、分散ソフトウェアに対して単純にノードを追加するだけで性能を線形的に上昇させることができる性質である。

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。なぜなら、並列で動く分散した資源を意識しながら記述するのは容易ではなく、また、どのように分散したノードの選択を行えば良いのか明確ではないからである。

分散プログラムには以下の 3 つの要素がある [1]。

- ノード内の計算
- ノード間通信
- 地理的に分散したノード

ノード内の計算には通信プロトコルやデータベースが含まれる。またノード間通信には、トポロジーの構成や通信の信頼性や速度、データの転送やデータの圧縮などの要素がある。

これらの要素をサポートするのが分散フレームワークである。すなわち分散フレームワークには、プロトコルの定義、それによってアクセスされるデータベース、トポロジーの決定いった分散アルゴリズムや、信頼性と拡張性の高い通信の提供が求められる。

本章では、これらの項目別に分けて、分散フレームワークである Akka [2]、Hazelcast [3] と当研究室で開発した Alice [4] [5] を比較し、本論文で設計する Christie の設計目標を述べる。

Akka は Scala および Java 向けのオープンソースの並列分散処理フレームワークであり、Hazelcast は Hazelcast 社が開発した Java 向けのオープンソースインメモリデータグリッドである。

1.1　従来の分散フレームワーク

Akka ではアクターモデルという、アクターと呼ばれるオブジェクト同士が並列で非同期メッセージを送受信するモデルを採用している。アクターは固有のアドレス持っており、ローカルのアクターにもリモートのアクターにも同じようにアドレスを指定することでメッセージを送りあえるというプロトコルになっている。アクターはそれぞれメールボックスというキューを持っており、メールボックスに受け取ったメッセージをパターンマッチで順次処理していく。このパターンマッチには Scala の case class を用いられる。case class とは、データ構造とデータ型名を同時に持つことができ、その両方を一括してパターンマッチさせることができるクラスである。メッセージを case class で記述することにより、異なるメッセージをメールボックス上で使用することがきる。

Hazelcast は、キーと値の 1 対 1 でデータを管理するインメモリ・データグリッドである。インメモリ・データグリッドとは、複数のノードに分散させたデータを、アプリケーション側からは仮想的な 1 つのメモリ空間に置かれているように見せるモデルである。そのため、プログラマがサーバを意識せずに共有のタプルスペースに対してデータを get/put できるプロトコルとなっている。共有のタプルスペースに書き込むとそれに接続しているノードにデータを行き渡らせる、マルチキャストベースの通信を採用している。

Alice は、タスクを Code Segment、データを Data Segment という単位で記述し、Code Segment はインプットとなる Data Segment が全て揃うと並列に実行される。Data Segment は対になる key が存在し、Data Segment Manager というノードごとに存在する独自のデータベースによって管理されている。各ノードにはラベル付きのプロキシである Remote Data Segment Manager を立て、ラベルと key を指定してデータを take/put するプロトコルとなっている。

記述の面において、Akka では基本的にメッセージを FIFO 的に処理するため、複数のインプットを待ち合わせる場合は、待ち合わせの処理をプログラマが各必要があった。しかし Alice は複数のインプットを 1 箇所に記述するるため、そのような煩雑さがない。

プロトコルの設計方針において、Akka や Hazelcast は分散通信の複雑さを抽象度を高めることで隠す方針であるため、ロケーション透過性が高く、プログラマからは処理の流れを把握しにくくなっていた。一方で Alice では明示的に分散性を意識する代わりに、計算を通常計算とそれを支えるメタ計算として分離することで記述の複雑さを回避している。これにより処理の流れを明確にし、分散計算のチューニングをしやすくしている。

また、Alice の分散ノード間の通信はラベルを用いてリモートノードを選択することによって指定する。これは Akka で送り先をドメインで指定しているのと同様である。Alice ではラベルの命名を Toplogy Manager によって自動的に指定することもできる。

1.2　トポロジーの構成

Akka では Akka Stream という機能で処理の流れが記述できる。N 入力 1 出力、1 入力 N 出力、出力のみ、などが用意された Junctions と呼ばれる要素をコード上で矢印でつなぎ合わせることでトポロジーを記述する。

Hazelcast には Map や Queue といったメモリ空間内のデータ構造は指定できるが、具体的なノード間トポロジーを記述する機構がない。

Alice では TopologyManager という機構が分散ノードを管理しており、静的・動的なトポロジーを自動構成する。静的トポロジーではプログラマがトポロジーを図として記述できるため、より分かりやすく詳細な設定ができる。

1.3　信頼性・拡張性の高い通信の提供

ここでは信頼性・拡張性の高い通信の指標として、障害耐性、圧縮・転送通信、NAT 越えについてを比較する。

障害耐性

分散プログラムでは、1 つのノードがダウンしてもシステム全体は動き続けなければならないため、フォールト・トレラントであることが重要である。

Akka では親子関係を構成でき、親アクターは子アクターを監視し障害が起こった際に再起動や終了といった処理を指定できる。

また、Hazelcast は、1 つのサーバで障害が起きても他のサーバがデータを共有しているため、データを失うことなく素早く復旧ができる。

Alice では、TopologyManager 内に KeepAlive という機能があり、常にノードが生きているか Heartbeat を送信して監視しており、どこかのノードに障害が起こればトポロジーを再構成するといった対応ができる。

圧縮・転送通信

ノード間通信でサービスに沿った柔軟な通信をするためには、送信するデータを圧縮する機能や、受け取ったデータをそのままほかノードへ転送する機能が求められることがある。

データの圧縮を指定したい場合、Akka、Hazelcast はシリアライザが用意されているため、そのメソッドを呼び出すことで圧縮伸長を行う。また、転送を指定したい場合、Akka には forward メソッドがあるためそれを呼び出すことで受け取ったデータの転送が可能だが、Hazelcast は一つの Map へのアクセスに見立てているため、転送にも put を用いる。

一方で Alice は圧縮の伸長と転送を同時に行うことを想定した圧縮・転送機能を持っている。Data Segment 内に圧縮と非圧縮の両形式を同時に持てるため、受け取った圧縮データを伸長をしながら圧縮したまま別ノードに転送することができる。また、圧縮するには送信する宛先ラベルに”compressed”とつけるだけでよく、データ取得時に自動で伸長もされるため、プログラマがメソッドの呼び出しを追加する必要がなく圧縮・非圧縮を簡単に切り替えられる。

NAT 越え

ネットワーク間通信の大きな問題の一つに、NAT がある。NAT とは、WAN と LAN の間にある IP アドレスの変換機構である。NAT を隔てたプライベートネットワーク内では、LAN 内だけでユニークなプライベート IP アドレスを持っており、WAN 側からはその IP アドレスを直接指定してアクセスできないためアドレス変換を行う必要がある。そのため NAT を越えたノード間通信は容易ではなく、分散フレームワークでそれをサポートできることが望ましい。

しかし Hazelcast では NAT 越えをサポートする機能がなく、プログラマが自前で書かなければならない。Akka ではノードの設定にグローバルアドレスとプライベートアドレスを両方登録することで NAT を越えた通信を可能にする。Alice に NAT 越えの機能はないが、TopologyManager が各ノードの Data Segment Manager と通信してトポロジー管理をしており、TopologyManager/Data Segment Manager を複数立ち上げることによりプライベートトポロジーとグローバルトポロジーの同時構成が可能だと考えた。しかし、Alice が複数の Data Segment Manager を持てない実装だったため、Alice のままで NAT 越えを実装することは困難であると判明した。よりスケーラブルな分散環境を提供するためにも、Alice を再設計する必要がある。

第2章　分散フレームワーク Aliceの概要

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。

2.1　CodeSegment と DataSegment

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。

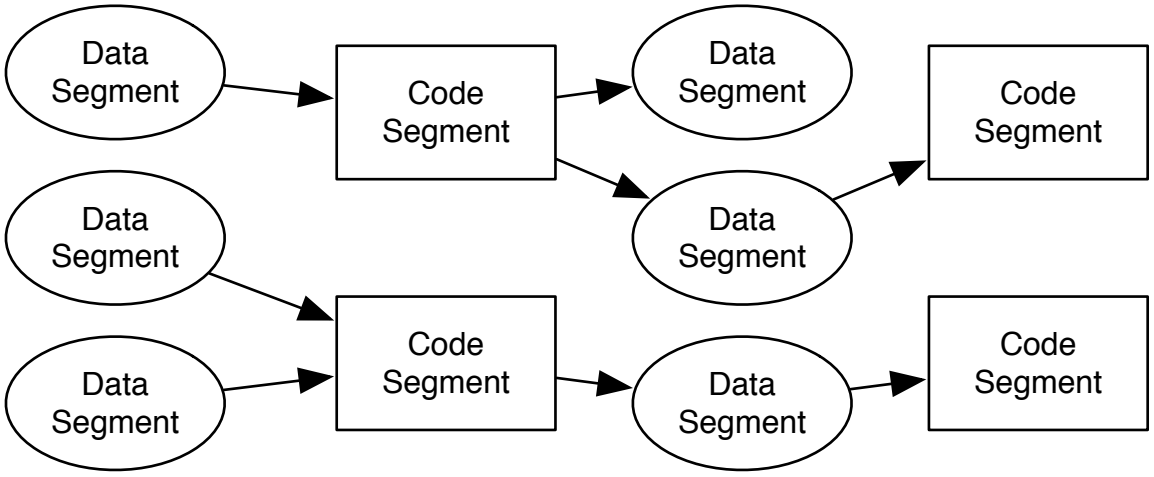

図 2.1: CodeSegment の依存関係

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 DataSegmentManager

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。

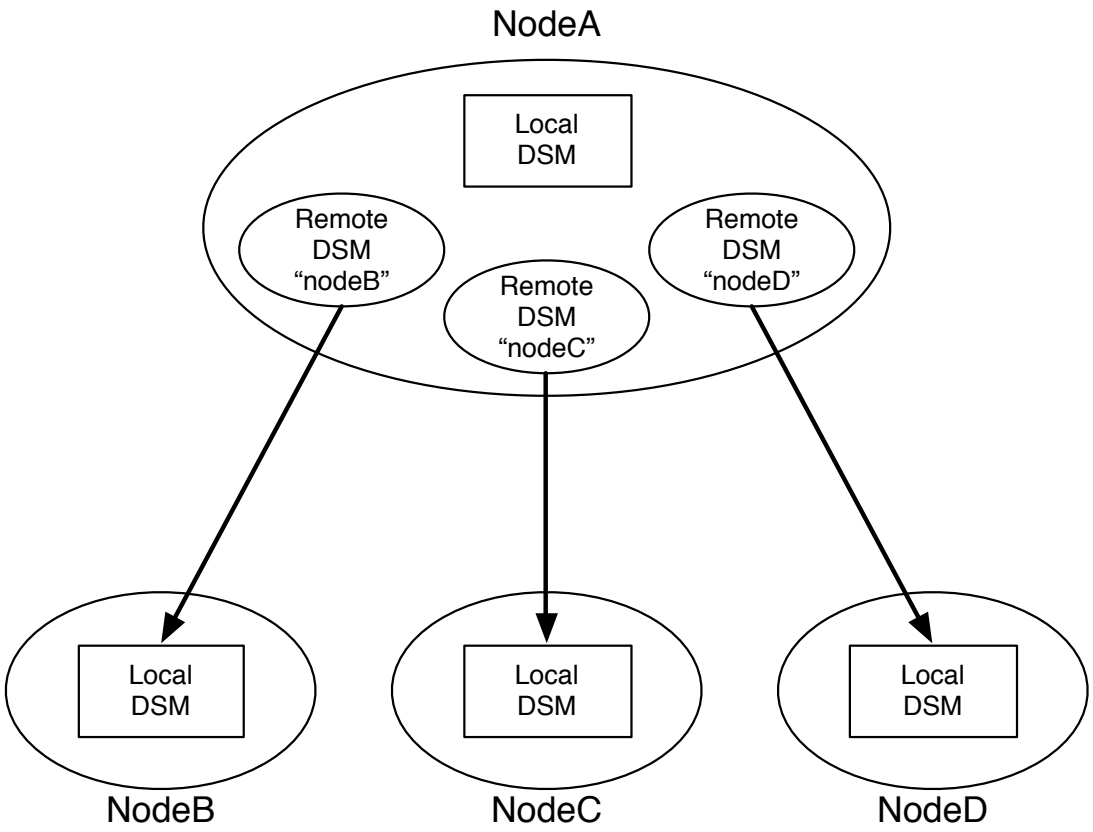

図 2.2: Remote DSM は他のノードの Local DSM の proxy

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。

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 CodeSegment 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 }
```

ソースコード 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 }
```

ソースコード 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。

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、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 を作る。run 内の処理を終えたら CS は破棄されるため、処理を繰り返したい場合はこのように新しいく CS を作る必要がある。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。

1. で用いられている asInteger() は asClass メソッドの一部であり、asClass は take/peek で取得した DS を Object 型から任意の型で取得するための API である。

- `<T> T asClass(Class<T> clazz)`

CS 内で DS のデータを扱うには、正しい型を意識しながらこの asClass メソッドを使わなければならない。

2.5 Alice の Meta Computation

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。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 ノードとの通信トポロジーの構成や、通信するデータ形式の変換と言える。

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 として指定するだけでよい。

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。仕様の変更を抑えてプログラムの拡張ができるということは、コードを破壊しないため変更以前の信頼性を保てるということである。

Meta Computation も CS/DS で作られており、プログラマ側から見えないこれらの CS/DS は Meta CS/Meta DS と呼ばれる。

現在 Alice には、データの圧縮機能、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。これらの有用性は水族館の例題 [6] や TreeVNC の例題 [7] [8] によって示された。

2.5.1　Alice の圧縮機能

リモートノードに大きなデータを送るために、データを圧縮したい場合がある。そこで、Alice は圧縮をサポートしている。しかし、単に圧縮のメソッドを用意したわけではない。圧縮データの伸長と、圧縮したまま別ノードへの転送を同時に実現したい場合があるため、Meta CS を介すことで DS に圧縮と非圧縮のデータを同時に持てるようにしている (図 2.3)。

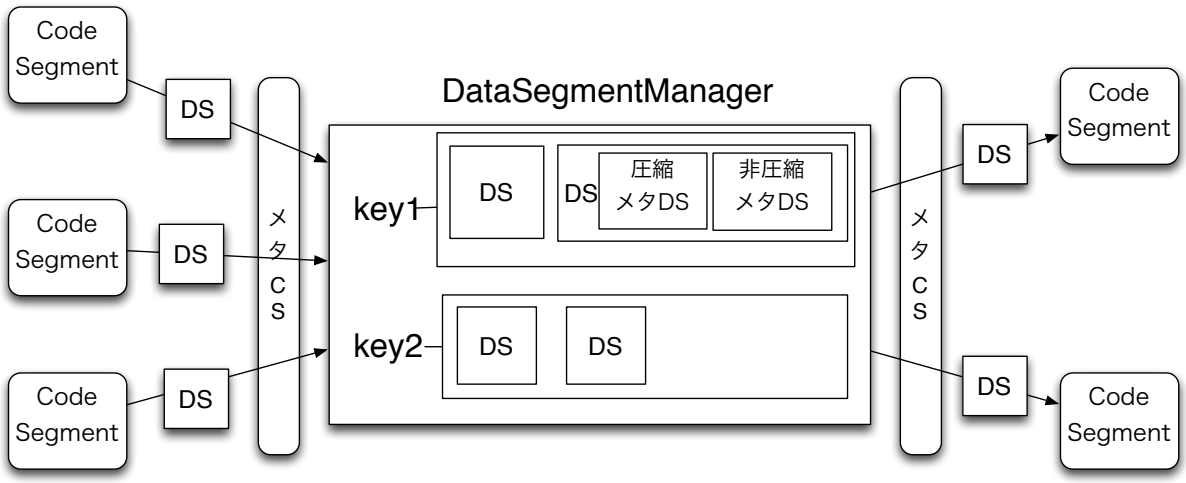

図 2.3: DS が圧縮と非圧縮の両方を持つ

1 つの DS 内に Meta DS として以下の 3 つの表現を持たせることでデータに多態性を持たせ、必要に応じた形式で DS を扱う。

1. 一般的な Java のクラスオブジェクト
2. MessagePack for Java [9] 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。Local/Remote DSM に DS を圧縮して保存したい場合は (3) の圧縮形式を用いる。

データの圧縮を指定するには、put する DSM の名前の前に”compressed”をつけるだけでよい。ソースコード 2.3,2.4 は通常の DS と圧縮の DS を扱う際の記述の例である。

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

ソースコード 2.3: 通常の DS を扱う CS の例

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

ソースコード 2.4: 圧縮した DS を扱う CS の例

このようにコードの変更を抑えて圧縮できるため、他の計算部分を変えずにデータ形式が指定できる。また、DS を取り出す際も asClass() 内部で自動で伸長が行われるため、コードの変更がなく、プログラマがデータの伸長を考える必要がない。

2.5.2 TopologyManager

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 Topology Node 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 [10] 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 2.5 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 }
```

ソースコード 2.5: 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 2.4)。

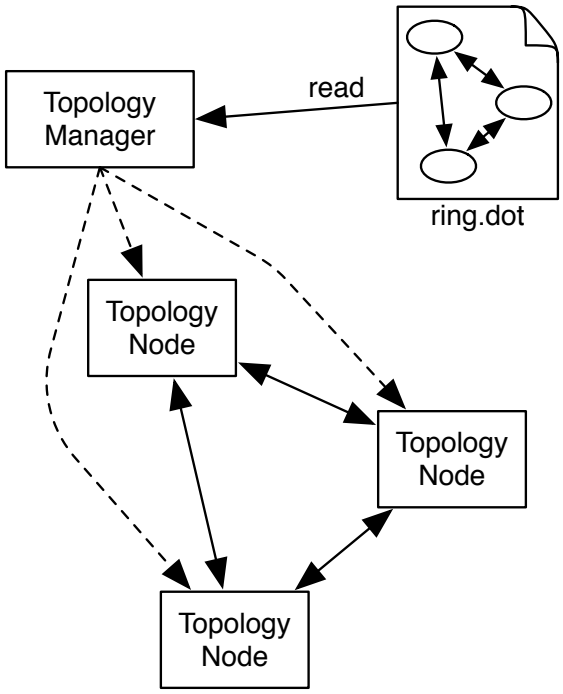

図 2.4: Topology Manager が記述に従いトポロジーを構成

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。

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。

第3章　Aliceの問題点

Alice を拡張していく中でいくつかの問題点が明らかになり、これらを解決するには Alice 自体を再設計する必要があるとわかった。

3.1　APIの記述の分離

2.4 で示したように、InputDS を記述するには、一度フィールドで Receiver を create して、その後 Reveiver に対して setKey で待ち合わせる key を指定しなければならない。このようにインプットの処理が分離されてしまっていては、記述が煩雑な上にコードを読んだ際にどの key に対して待ち合わせを行っているのか直感的に分からない。

さらに、setKey は明確な記述場所が決まっていないため、その DS を待ち合わせている CS 以外からも呼び出せてしまう (ソースコード 3.1)。

```
1 public class StartCodeSegment extends CodeSegment {
2     @Override
3     public void run() {
4         TestCodeSegment cs = new TestCodeSegment();
5         cs.input.setKey(”data”);
6         ods.put(”local”, ”data”, 1);
7     }
8 }
```

ソースコード 3.1: setKey を外部から呼び出す例

```
1 public class TestCodeSegment extends CodeSegment {
2     private Receiver input = ids.create(CommandType.TAKE);
3
4     @Override
5     public void run(){
6         System.out.println(”data␣=␣” + input.asInteger());
7     }
8 }
```

ソースコード 3.2: 外部 setKey によりどの key を待っているかがわからない CS の例

このような書き方をされると、CS だけを見てどの key に対して待ち合わせを行っているのかわからないため、setKey を呼び出しているコードを辿る必要がある。これでは見通しが悪いため、どこで key を指定するのか明確にすべきである。

可読性の低いコードはプログラマの負担となるため、CS が何を待ち合わせているのかその CS を見ただけで理解できるように記述の分離問題を改善しなくてはならない。

3.2 setKey は最後に呼ばなければならない

setKey メソッドをコンストラクタで呼ぶ際、setKey メソッドを必ず最後に呼ばなければならない。

CS は内部で実行に必要な DS を数えている。DS の取得に成功するとこの値が、デクリメントされ、0 になると必要な DS が全て揃ったことと判断され Thread pool へ送られる。

setKey 移行に処理を記述した場合、その処理が行われない可能性があり Thread pool へと送られ NullPointerException を引き起こす。

```
1  public class ShowData extends CodeSegment{
2      private Receiver[] info;
3
4      public ShowData(int cnt) {
5          info = new Receiver[cnt];
6          for (int i= 0;i < cnt; i++) {
7              info[i] = ids.create(CommandType.TAKE);
8              info[i].setKey(SetInfo.array[i]);
9          }
10     }
11
12     @Override
13     public void run() {
14         int size = 0;
15         for (Receiver anInfo : info) {
16             DataList dlist = anInfo.asClass(DataList.class);
17             dlist.showData();
18         }
19     }
20 }
```

ソースコード 3.3: NullPointerException になる可能性がある

ソースコード 3.3 は、for 文で setKey と ids.create を cnt の回数呼び、動的に DS の取得数を決めようとしている。しかし、setKey が最初に呼ばれた際に、DS の取得に成功すると実行可能と判断されてしまう。run の中で info の配列の要素だけ中身を表示させようとしてるが、2 回目の asClass で NullPointException を引き起こす。この問題もインプット API の記述の分離により引き起こされるエラーである。

今回の場合、コンストラクタ内をソースコード 3.4 のように記述する必要がある。

```
1  public ShowData(int cnt) {
2      info = new Receiver[cnt];
3      for (int i= 0;i < cnt; i++) {
4          info[i] = ids.create(CommandType.TAKE);
5      }
6
7      for (int i= 0;i < cnt; i++) {
8          info[i].setKey(SetInfo.array[i]);
9      }
10 }
```

ソースコード 3.4: NullPointerException にならない記述

このように記述の順序を考えながらプログラミングしなければならない設計では、バグを引き起こし信頼性を損なうことに繋がる。より自然に扱える API 設計にするべきだと考える。

3.3 動的な setKey

setKey は CS のコンストラクタで指定することが多い。このとき、指定する key は引数などから動的に受け取り、セットすることができる。しかし、それでは実際にどんな処理が行われているのかわかりづらく、また、put する部分などの該当する key を扱う全てコードを変更しなければならない。このように、Alice では CS を使いまわすことを考慮して動的な setKey を可能にしてしまったせいで、慎重に書かなければプログラムの信頼性が保てないようになってしまっている。そのため、動的な setKey はできないように制限し、コードの見通しを良くする必要がある。CS に対してインプットとなる key が静的に決まれば、待ち合わせている key に対しての put のし忘れなどの問題をコンパイル時のモデル検査などで発見することができると考えられる。

3.4 型が推測できない

inputDS を受け取る Receiver はデータを Object 型で持っており、そのデータを CS 内で扱うには正しい型にキャストする必要がある。しかし、inputDS で指定するのは key のみであり、そのデータの型までは分からない。そのため、DS の型を知るには put している部分まで辿る必要がある。辿っても flip されている可能性もあるため、最初にその DS を put している部分を見つけるのは困難である。従って、待ち合わせている key にどのような型のデータが対応しているのかをその CS を見ただけで分かるようにするべきと考える。

3.5 key 名と変数名の不一致

2.4 の CodeSegment の例題である通り、key 名とその key で待ち合わせた DS を受け取る Receiver 名は異なることがある。もしプログラマが適当に命名してしまえば後々混乱を招くため、待ち合わせる key 名と input DS の変数名一致を強制させたい。

3.6 DataSegment の型の明瞭性

2.5.1 で示したように、Alice に圧縮の Meta Computation を実装した際、DS 内に複数の型を同時に持たせるようにした。

しかしこれでは、DS が今どの形式を持っているのか、どの状態にあるのかがわかりづらい。また、DS が byteArray 型を受け取った場合、データである Object 型として渡されたものなのか、MessagePack や圧縮で変換されたものなのかを判別する処理を入れなければならなかった。今後 DS により多様な形式を同時に持たせることになれば、さらにその判別の処理が増えることになる。

Alice 自体の拡張・デバッグをしやすくするためにも、DS がどの型を持っているのかをひと目で分かるようにしたい。

3.7 LocalDataSegmentManager を複数持てない

Alice では 1 つのノードにつき 1 つしか LocalDSM を立ち上げられない作りになっている。そのために以下のような問題が発生した。

3.7.1 1 つのノードで複数台 DSM 同士のテストが行えない

当研究室では分散データベース Jungle [11] を開発しており、その分散通信部分には Alice が用いられている。Jungle のような分散アプリケーションの開発では、1 つのマシン上で複数の疑似ノードを立ててテストを行いたい場合があった。しかし、Alice では一つのアプリケーション内に LocalDSM は一つと決まっていたため、テストに必要なノード数分だけアプリケーションを別で立ち上げなければならないという手間があった。このためのシェルスクリプトをプログラマが書かなければならないのは本質的な作業ではない。より気軽にテストができるよう、同一プログラム内で LocalDSM を複数立ち上げられるようにすべきだと考えた。

3.7.2　TopologyManager の拡張が困難

Alice ではより自由度の高い通信を行うために、TopologyManager に幾つかの機能を追加すること考えていた [12]。

その一つが NAT 越えの機能である。NAT 越えは分散アプリケーション構築における課題の 1 つでもあるが、プログラマにとってその実装は容易ではない。Topology Manager に NAT を越えたノード間通信機能をつけることにより、ネットワークを気にせずに通信が行えるようにしたい。

図 3.1 は TopologyManager を用いて NAT 越えをするための設計である。

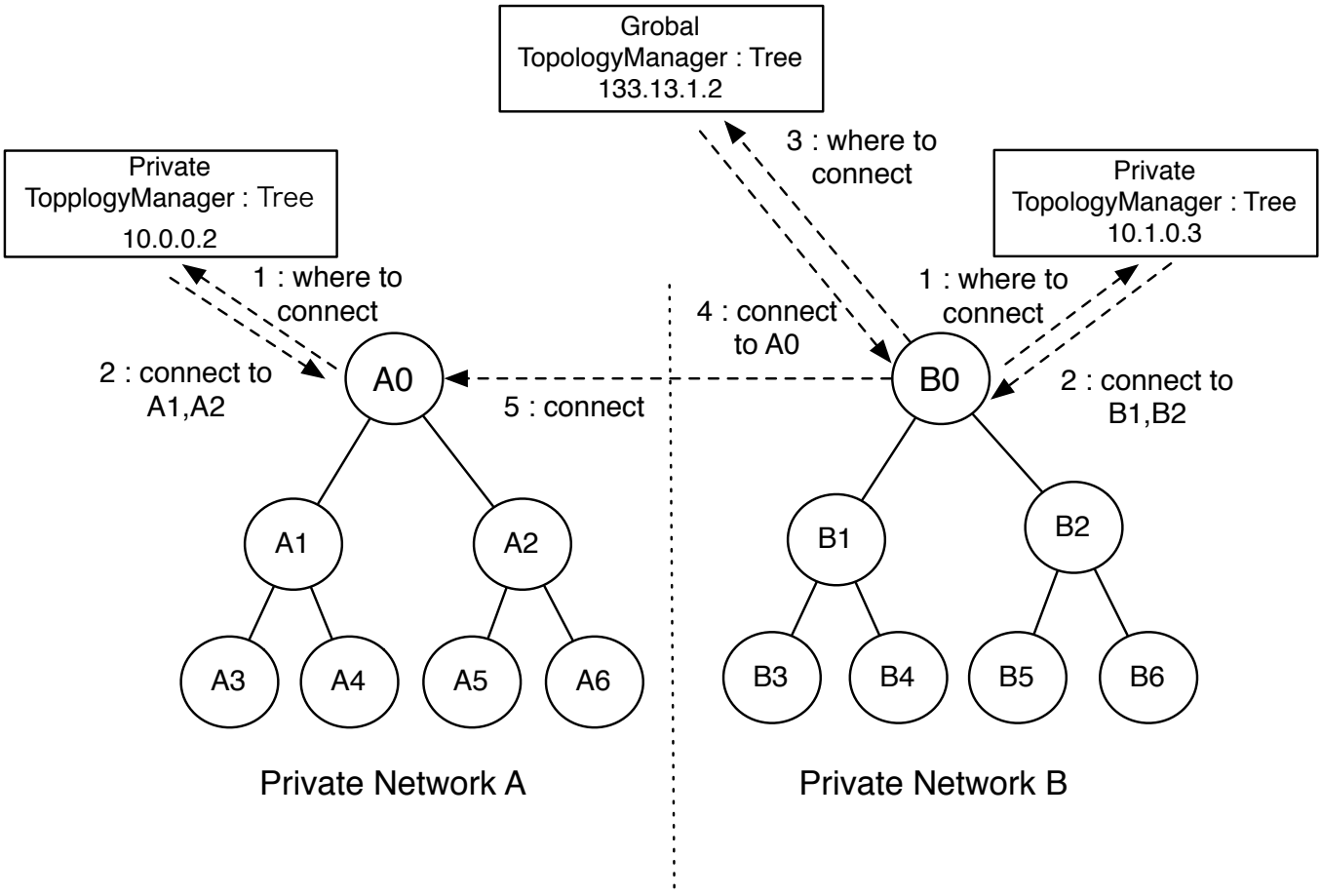

図 3.1: 複数の TopologyManager による NAT 越えの実現

また、別トポロジーで立ち上げたアプリケーション同士を接続する機能も追加したいと考えていた。TreeTopology の VNC アプリと StarTopology のチャットアプリを連携したいという要望が生まれたためである。別トポロジーのアプリケーションが接続可能になれば、VNC 画面のスナップショットを Chat 上に載せたり、VNC 上に Chat の内容をコメントとして流すといった拡張が容易になる (図 3.2)。

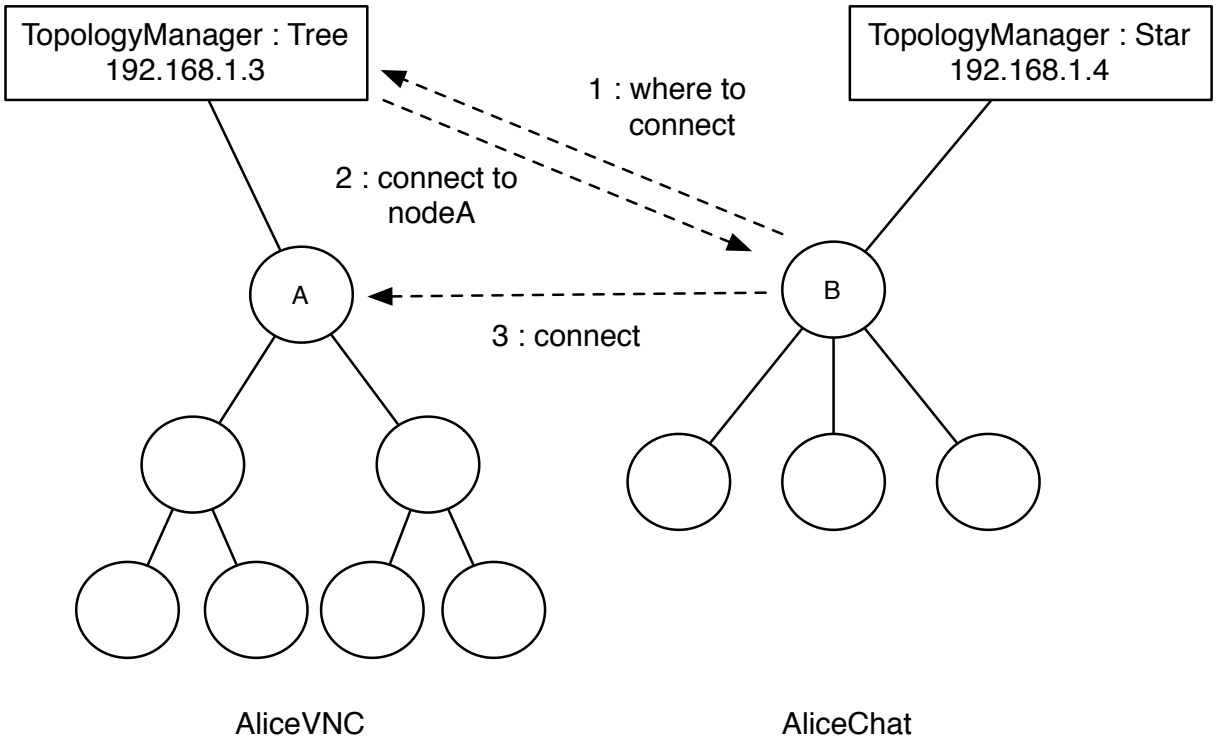

図 3.2: 別トポロジーのアプリケーションの接続

TopologyManager はネットワークごと、トポロジーごとに存在するため、いずれの機能も複数の TopologyManager を立ち上げ、連携させることで実現可能となる。

今までの Alice では、1 つのノードに対して Topology Manager は 1 つと決められていた。Topology Manager と各ノードのやり取りをするのは、ノードごとに実行される Topology Node という Meta Computation である。Topology Manager は接続された node の情報 (nodeName と IP アドレスの HashMap) を”nodeTable” という Key に対応する DS として保

存している。そして Topology Node は Topology Manager から割り当てられた nodeName を”hostname”という Key に保存する。つまり、接続する Topology Manager が増えれば TopoloyNode に割り当てられる nodeName も増えるため、今までのように”hostname”という 1 つの Key だけでは対応できない。1 つのノードに複数の TopologyManager を対応させるには、TopologyNode が複数の nodeName を持つ必要がある。TopologyNode が複数の TopologyManager に対応できるようにしなければならない。

そこで、Meta Computation として、通常の Local DSM とは別に Topology Manager ごとの Meta Local DSM を立ち上げる方法が考えられる (図 3.3)。

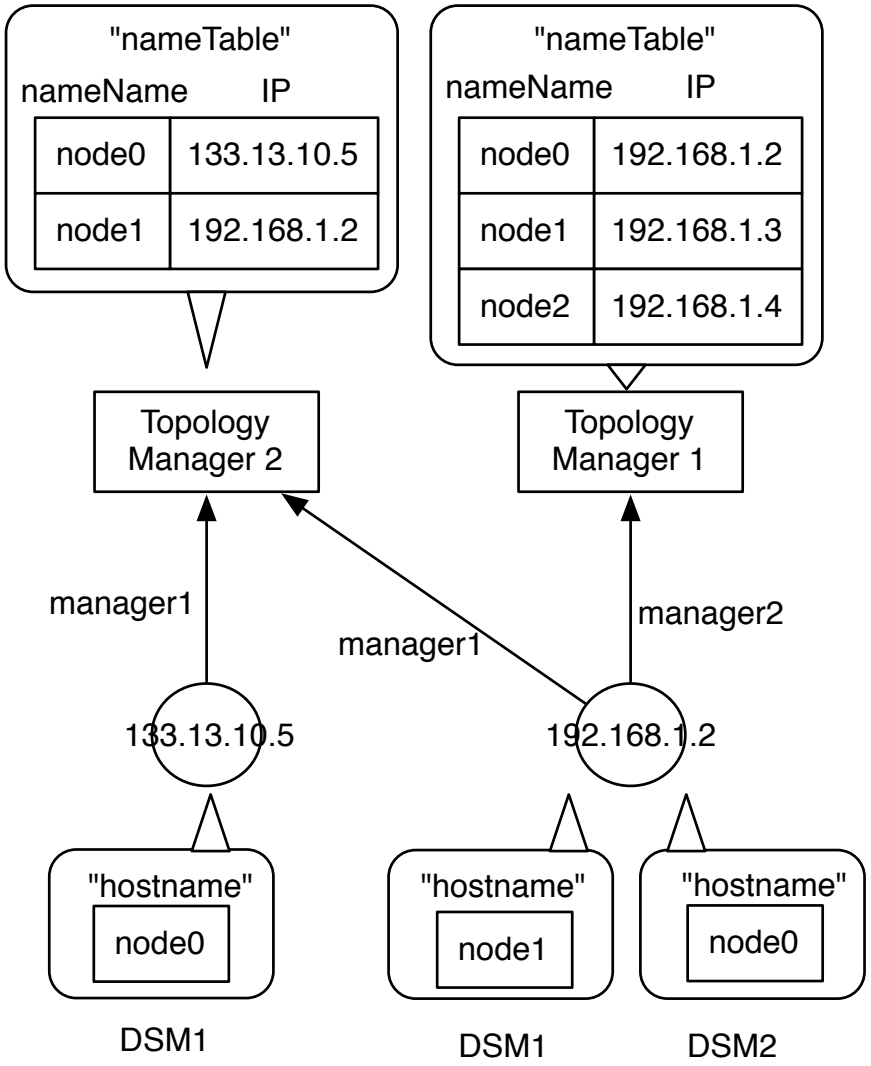

図 3.3: 複数の TopologyManager に複数の LocalDSM が対応

それぞれの Topology Manager に対応する LocalDSM を作り、それぞれに対応した nodeName を格納することで、DSM を切り替えるだけで TopologyNode の仕様は変えずに複数の Topology Manager に対応できるという設計である。

しかし、現在の Alice のコードでは DSM を管理する class が static class であったため、複数の Local DSM を持つことはできなかった。static を取り除こうとしたところ、Alice の大部分のコードを修正する必要があることがわかった。よって、再設計の際には static class のない実装を行い、DSM 切り替えによる方式を実現したい。

第4章　分散フレームワークChristieの設計

4.1　Christieの必要条件

3章でのAliceの問題点を踏まえ、新たにフレームワークを作り直すべきだと考えた。本章では、新たに作った分散フレームワークChristieの設計を説明する。Christieに必要な要件は以下のように考える。

- `create/setKey`のような煩雑な`API`をシンプルにし可読性を向上させる
- プログラマが型を推測しなくとも整合性がとれるように型を解決し、信頼性を向上させる
- `static`な`LocalDSM`をなくし、複数のインスタンスを同時に立ち上げられるようにすることでスケーラビリティを向上させる

4.2　Christieの基本設計

基本的にはAliceと同じ、タスクとデータを細かい単位に分割して依存関係を記述し、入力が揃った順から並列実行するというプログラミング手法を用いる。

ChristieはAliceと同じくJavaで書かれている。しかし将来的に当研究室が開発するGearsOS [13]に取り入れたいため、GearsOSを構成する言語であるContinuation based C(CbC)に互換可能な設計を目指す。

GearsOSではCodeSegment/DataSegmentと同様の概念としてCodeGear/DataGearという名称を用いているため、Christieでもそれに倣いCodeGear/DataGear(以下、CG/DG)と呼ぶこととする。

DG は Alice と同様に DataGearManager(以下 DGM) が管理する。DGM は Local と Remote があり、全ての DGM は CodeGearManager(以下 CGM) で管理される。GearsOS では Context という全ての CG/DG を一括管理するプロセスがあり、Alice の CGM もこの Context に相当する。全ての CGM は ThreadPool と他の CGM 全てのリストを共有しているため、全ての CG/DG にアクセス可能である (図 4.1)。

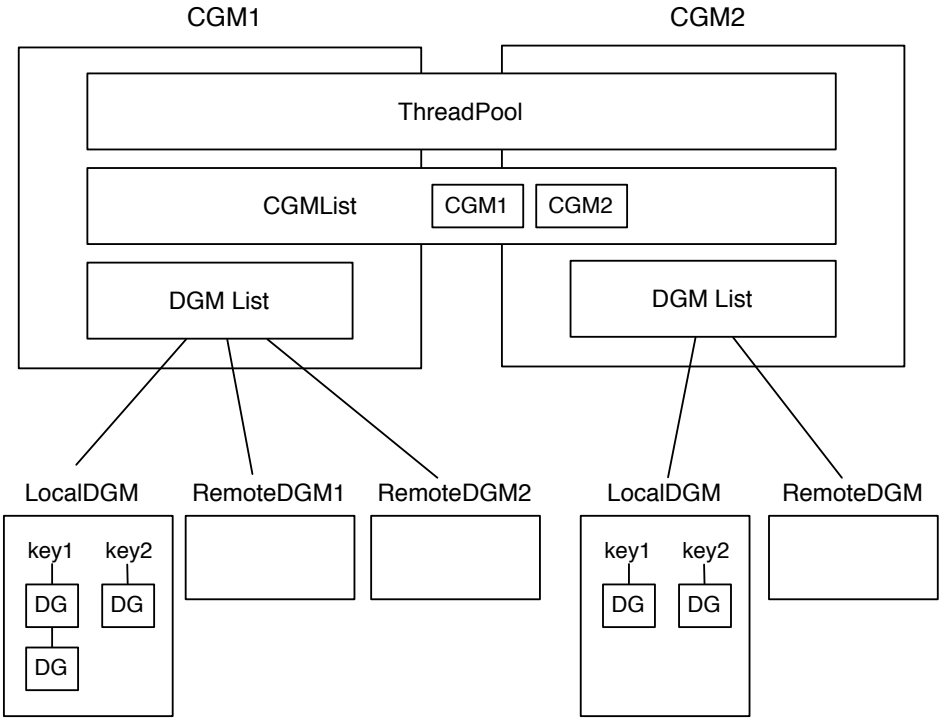

図 4.1: CGM は CGM と DGM を管理する

CG を記述する際は Alice 同様 CodeGear.class を継承する。CodeGear は

void run(CodeGearManager cgm) を持つ class であり、プログラマは run メソッド内に処理を記述する。インプットで指定した key に対応した DG が全て揃ったとき、run に書かれた処理が実行される。Christie の API には run の引数で受け取った CGM を経由してアクセスする。GearsOS では CG 間で Context を受け渡すことによって CG は DG にアクセスするため、Christie でもその記述方法を採用した。

通常の Runnable クラスではこのように引数をうけとることができないが、CodeGearExecutor という Runnable の Meta Computation を挟んだことでこのように CGM を受け渡しながらの記述を可能にした。

詳しい CodeGear の記述方法については、4.4 章で説明する。

4.3　API の改善

ここでは Alice の API の問題を踏まえて設計した Christie の API について、インプット、アウトプット、データの取り出しに分けて説明する。

アノテーションの導入によるインプットの記述

InputAPI には Alice と同じく Take と Peek を用意した。Christie では Input DG の指定にはアノテーションを使う。アノテーションとは、クラスやメソッド、パッケージに対して付加情報を記述できる Java の Meta Computation である。先頭に@をつけることで記述でき、オリジナルのアノテーションを定義することもできる。

Alice では Input の受け皿である Receiver を作り後から key をセットしていたが、Christie では Input のための DG を作り、その上にアノテーションで Key を指定する (ソースコード 6.1)。

```
1 @Take(”count”)
2 public DataGear<Integer> count = new DataGear<>();
```

ソースコード 4.1: Take の例

アノテーションで指定した InputDG は、CG を生成した際に CodeGear.class 内で待ち合わせの処理が行われる。これには Java の reflectionAPI を利用している。

Christie のこのインプットアノテーションはフィールドに対してしか記述できないため、InputDG の生成と Take/Peek の指定と key の指定を必ず一箇所で書くことが明確に決まっている。そのため Alice のように外の CS からの key への干渉をされることがない。また、アノテーションの指定は RUNTIME ではできないため、動的な key の指定も防ぐことができる。このように、アノテーションを用いたことで、Alice の記述の分離問題が解決された。

ソースコード 6.1 の 2 行目にあるように、InputDG を宣言する際には必ず型の指定が必要となる。DataGear は様々な型のデータを扱うために Java の総称型で受け取るようにしており、<>内に指定した型でデータの型を限定できる。このように記述することで、Christie では他の部分を辿らなくても CG を見るだけでインプットされるデータの型が分かるように可読性を向上させた。また、取得してきた DG が指定と違う型であった場合はエラーとなるため、型の整合性を保ちながら信頼性の高いプログラミングが可能となった。

また、Alice では key と変数名の不一致から可読性が低くなっていた。しかし Christie では key と変数名が一致しないとエラーとなるため、自然と読みやすいコードが書けるようになっている。この部分に関しては、Java のメタプログラミング API である Javassist [14] を用いてアノテーションから変数の自動生成も試みたが、Javassist では変数生成の前に

他のどのクラスも生成してはならないという制限があったため、Christie では実現できなかった。

リモートノードに対して Take/Peek する際は、RemoteTake/RemotePeek のアノテーションを用いる (ソースコード 6.2)。そのため待ち合わせ先が Local か Remote かはアノテーションの違いからひと目でわかるようになった。

```
1 @RemoteTake(dgmName = "remote", key = "count")
2 public DataGear<Integer> count = new DataGear<>();
```

ソースコード 4.2: RemoteTake の例

なお、圧縮の Meta Computation は Alice と同様で、指定する際に DGM 名の前に compressed をつける (ソースコード 4.3)。

```
1 @RemoteTake(dsmName = "compressedlocal", key = "count")
2 public DataGear<Integer> count = new DataGear<>();
```

ソースコード 4.3: Local への圧縮の指定の例

Local からの TAKE では DGM 名の指定がないが、それは Local での圧縮は基本想定していないためである。しかし、Local での圧縮をしようと思えば RemoteTake を用いて間接的にすることは可能である。

DGM を指定してのアウトプットの記述

OutputAPI には put/flip を用意した。put/flip のメソッドは DGM に用意されている。cal.java CodeGear.class には DGM を取得するメソッドがあり、それを用いて書き込みたい DGM を指定して直接 put する。そのため Local/Remote の切り替えは指定する DGM の切り替えによって行う。ソースコード 4.4、4.5 は Local と Remote に put する記述の例である。

```
1 getLocalDGM().put("count", 1);
```

ソースコード 4.4: Local へ put する例

```
1 getDGM("remote").put("count", 1);
```

ソースコード 4.5: Remote へ put する例

flip も同様に DGM に直接 DG を渡す (ソースコード 4.6)。

```
1  public class Flip extends CodeGear {
2
3      @RemoteTake(dgmName = "remote1", key = "name")
4      public DataGear<String> name = new DataGear<>();
5
6      @Override
7      protected void run(CodeGearManager cgm) {
8          getDGM("remote2").put("name", name);
9      }
10 }
```

ソースコード 4.6: Remote へ flip する例

Christie では DGM に対して直接 put するため、Alice の ODS にあたる部分はない。ODS を経由するより直接 DGM に書き込むような記述のほうが直感的であると考えたためである。

型を指定しないデータの取り出し

Alice の asClass に相当するのが getData である。ソースコード 4.7 は getData を用いて InputDG からデータを取得する例である。

```
1  public class GetData extends CodeGear{
2
3      @Take("name")
4      public DataGear<String> name = new DataGear<>();
5
6      @Override
7      protected void run(CodeGearManager cgm) {
8          System.out.println("this name is : " + name.getData());
9      }
10 }
```

ソースコード 4.7: getData の例

Alice と違う点は、プログラマが型を指定しなくて良い点である。4.3 で示したように、InputDG を生成する際には型を指定する。この型は内部で保存され、リモートノードと通信する際も保たれる。このように getData するだけでプログラマが指定しなくとも正しい型で取得できるため、プログラマの負担を減らし信頼性を保証することができる。

4.4　CodeGear の記述方法

以下のコードは LocalDSM に put した DG を取り出して表示するのを 10 回繰り返す例題である。

```
1  public class StartTest extends StartCodeGear{
2
3      public StartTest(CodeGearManager cgm) {
4          super(cgm);
5      }
6
7      public static void main(String args[]){
8          StartTest start = new StartTest(createCGM(10000));
9      }
10
11     @Override
12     protected void run(CodeGearManager cgm) {
13         cgm.setup(new TestCodeGear());
14         getLocalDGM().put("count", 1);
15     }
16 }
```

ソースコード 4.8: StartCodeGear の例

```
1  public class TestCodeGear extends CodeGear {
2
3      @Take("count")
4      public DataGear<Integer> count = new DataGear<>();
5
6      public void run(CodeGearManager cgm){
7          System.out.println(hoge.getData());
8
9          if (count.getData()!= 10){
10             cgm.setup(new TestCodeGear());
11             getLocalDGM().put("count", count.getData() + 1);
12         }
13     }
14 }
```

ソースコード 4.9: CodeGear の例

Alice 同様、Christie でも InputDG を持たない StartCG から処理を開始する。StartCG は StartCodeGear.class を継承することで記述できる。Alice では StartCS も CodeSegment.class を継承して書かれていたため、どれが StartCS なのか判別しづらかったが、Christie ではその心配はない。

StartCG を記述する際には createCGM メソッドで CGM を生成してコンストラクタに渡す必要がある。ソースコード 4.8 の 8 行目でそれが行われている。createCGM の引数にはリモートノードとソケット通信する際使うポート番号を指定する。CGM を生成した際に LocalDGM やリモートと通信を行うための Daemon も作られる。

CG に対してアノテーションから待ち合わせを実行する処理は setup メソッドが行う。そのためソースコード 4.8 の 13 行目、ソースコード 4.9 の 10 行目のように、new した CG を CGM の setup メソッドに渡す必要がある。Alice では new すれば CG が待ちに入ったが、Christie では一度 CG を new しないとアノテーションから待ち合わせを行う処理ができないため、new の後に setup を行う。そのため、CG の生成には必ず CGM が必要になる。run で CGM を受け渡すのはこのためである。なお、StartCG はインプットを持たないため、setup を行う必要がなく、new された時点で run が実行される。

4.5　DataGearManager の複数立ち上げ

Alice では LocalDGM が static で書かれていたため複数の LocalDGM を立ち上げることができなかった。しかし Christie では CGM を 2 つ生成すれば LocalDGM も 2 つ作られる。複数の LocalDGM 同士のやりとりも、Remote への接続と同じように RemoteDGM を proxy として立ち上げアクセスする (図 4.2)。

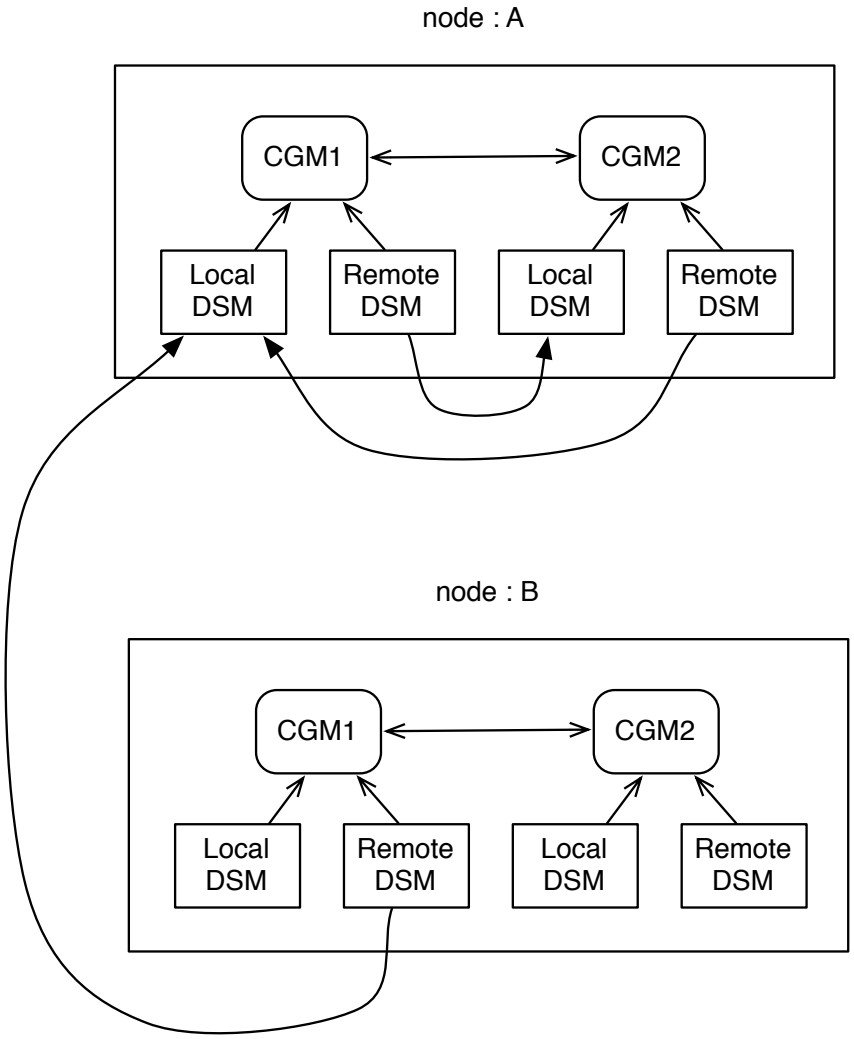

図 4.2: RemoteDGM を介して他の LocalDGM を参照

ソースコード 4.10 は、LocalDSM を 2 つ立ち上げ、お互いをリモートに見立てて通信する例である。11 行目にあるように、RemoteDGM を立ち上げるには CGM が持つ createRemoteDGM メソッドを用いる。引数には RemoteDGM 名と接続するリモートノードの IP アドレス、ポート番号を渡している。

```
1  public class StartRemoteTakeTest extends StartCodeGear{
2
3      public StartRemoteTake(CodeGearManager cgm) {
4          super(cgm);
5      }
6
7      public static void main(String args[]){
8          CodeGearManager cgm = createCGM(10000);
9          new StartRemoteTake(cgm);
10
11         cgm.createRemoteDGM("remote", "localhost", 10001);
12         cgm.setup(new RemoteTakeTest());
13
14         CodeGearManager cgm2 = createCGM(10001);
15         cgm2.createRemoteDGM("remote", "localhost", 10000);
16         cgm2.setup(new RemoteTakeTest());
17     }
18 }
```

ソースコード 4.10: LocalDGM を 2 つ作る例

リモートの場合の同じようにアクセスできることで、コードの変更をせずに、同一マシン上の 1 つのアプリケーション内で分散アプリケーションのテストができるようになった。

また、CGM は内部に CGM のリストを static でもっており、複数生成した CGM を全て管理している。つまり、メタレベルでは RemoteDGM を介さずに各 LocalDGM に相互アクセス可能である。そのため、Christie では容易に NAT 越えが実装できることが期待できる。

4.6 DataGear の拡張

Alice ではデータの多態性を実現するために DS 内に複数のデータ形式を保持していた。しかし Christie ではデータ形式ごとに別の class に分けている。DataGear を継承した MessagePackDataGear と、それを更に継承した CompressedDataGear を用意した。そのため子クラスは親クラスのデータ形式を保持しながら新しいデータ形式を持つ形になっている。クラスを見るだけで今どの形式を保持しているかわかるようになったため、デバッグがしやすくなった。

4.7　通信フロー

本章で説明した Christie の設計をいくつか例をあげて Christie の通信のフローをシーケンス図を用いて解説する。図 4.3 は LocalDGM に Take を行い、LocalDGM 内に DG があったときの処理の流れである。

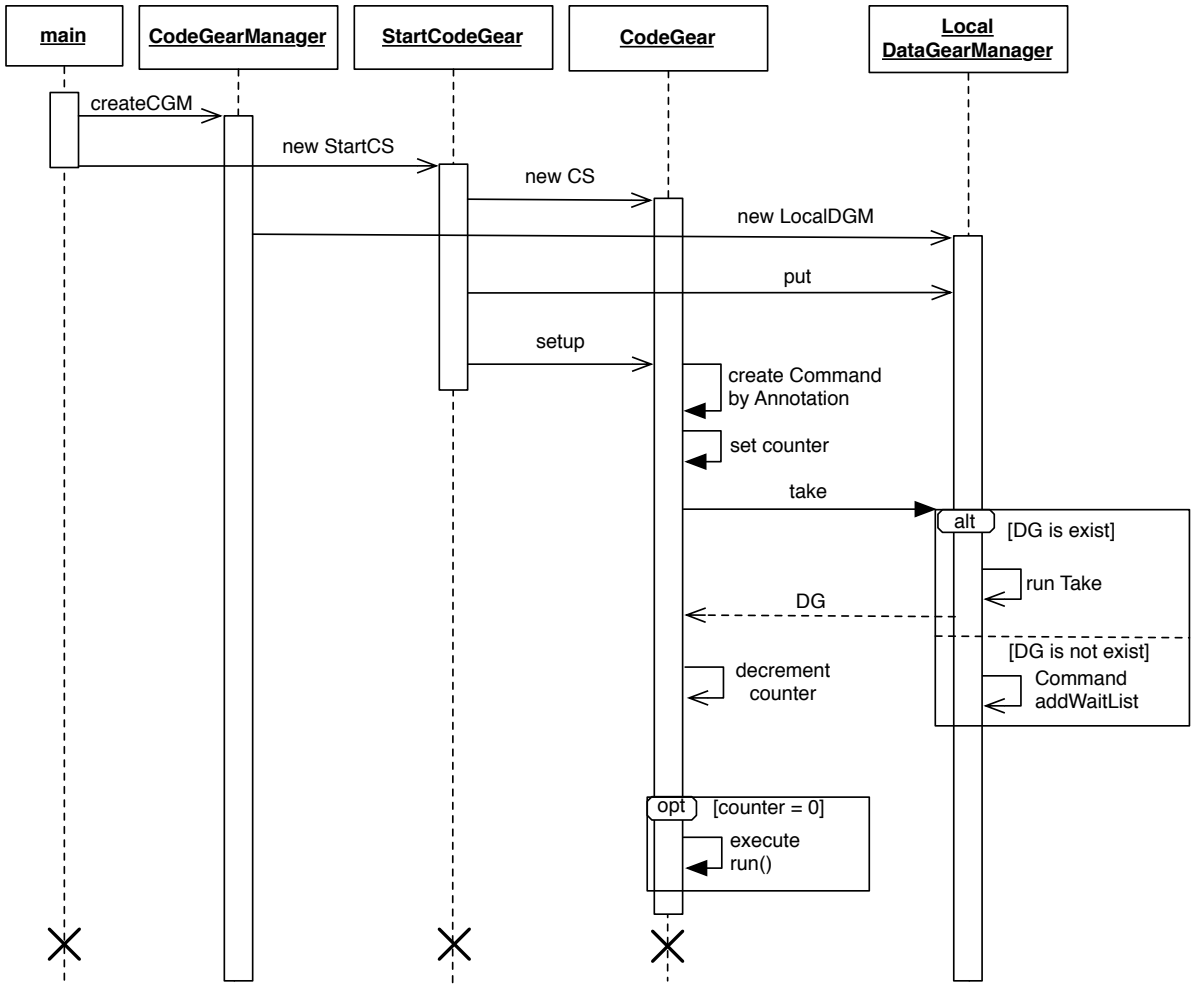

図 4.3: LocalDGM に Take したときのフロー

プログラマは main で CGM と StartCG を生成する。CGM と同時に LocalDGM は作られる。CG が生成され、setup メソッドが呼ばれるとアノテーションから TAKE コマンドが作られ実行される。CG は生成したインプットコマンドの総数を初期値としたカウンタを持っており、コマンドが解決される (InputDG が揃う) たびにカウンタは減っていき、0 になると run 内の処理が ThreadPool へ送られる。

図 4.4 は、LocalDGM に Take を行うが、LocalDGM 内に DG がなかったために Put の待ち合わせをするときの処理の流れである。main などの最初の処理は図 4.3 と同様のため省略する。

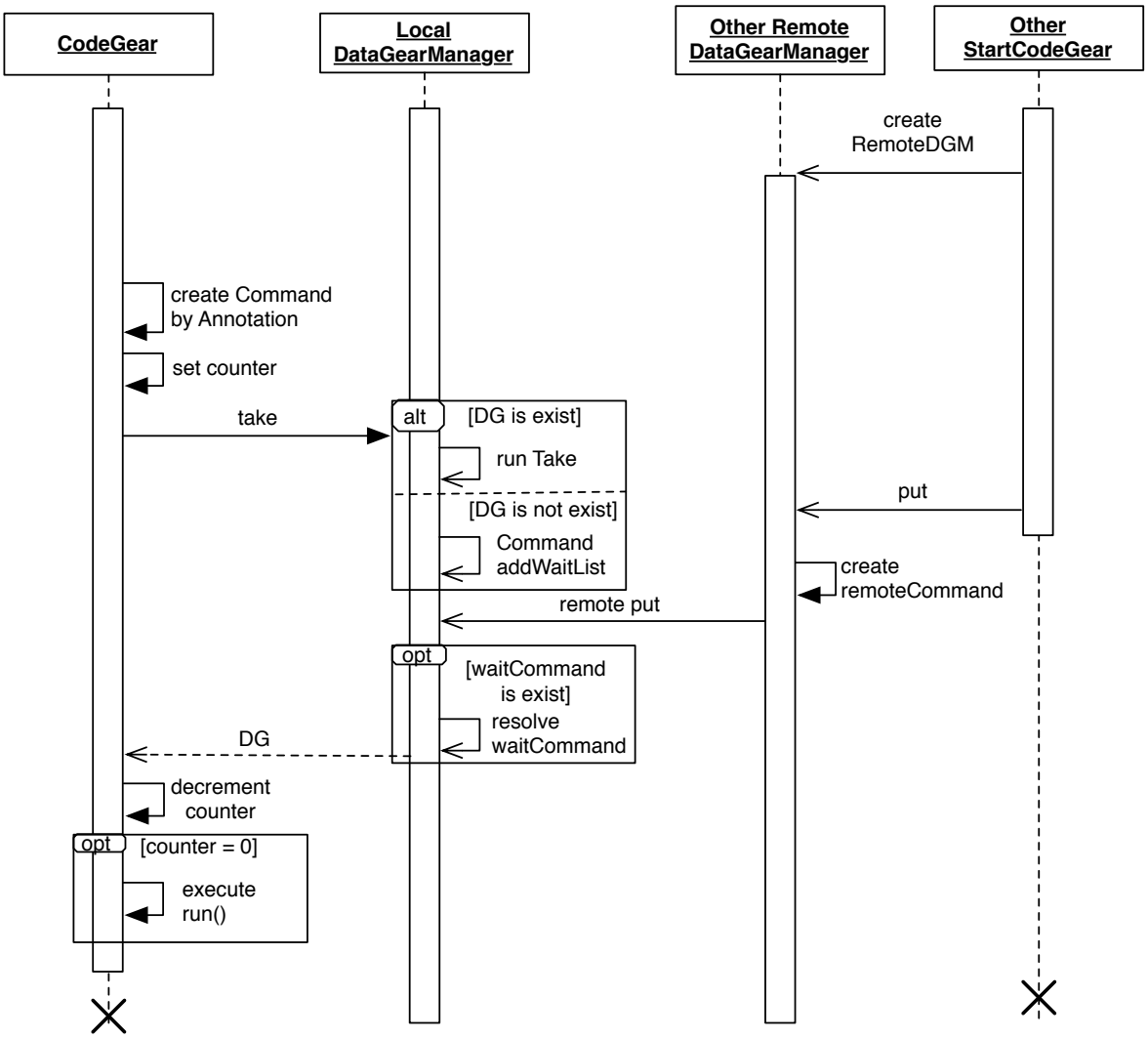

図 4.4: RemoteDGM に Put したときのフロー

Local またはリモードノードから PUT コマンドが実行された際、もし waitList に Put した DG を待っているコマンドがあれば実行される。

図 4.5 は、RemoteDGM に Take を行ったときの処理の流れである。

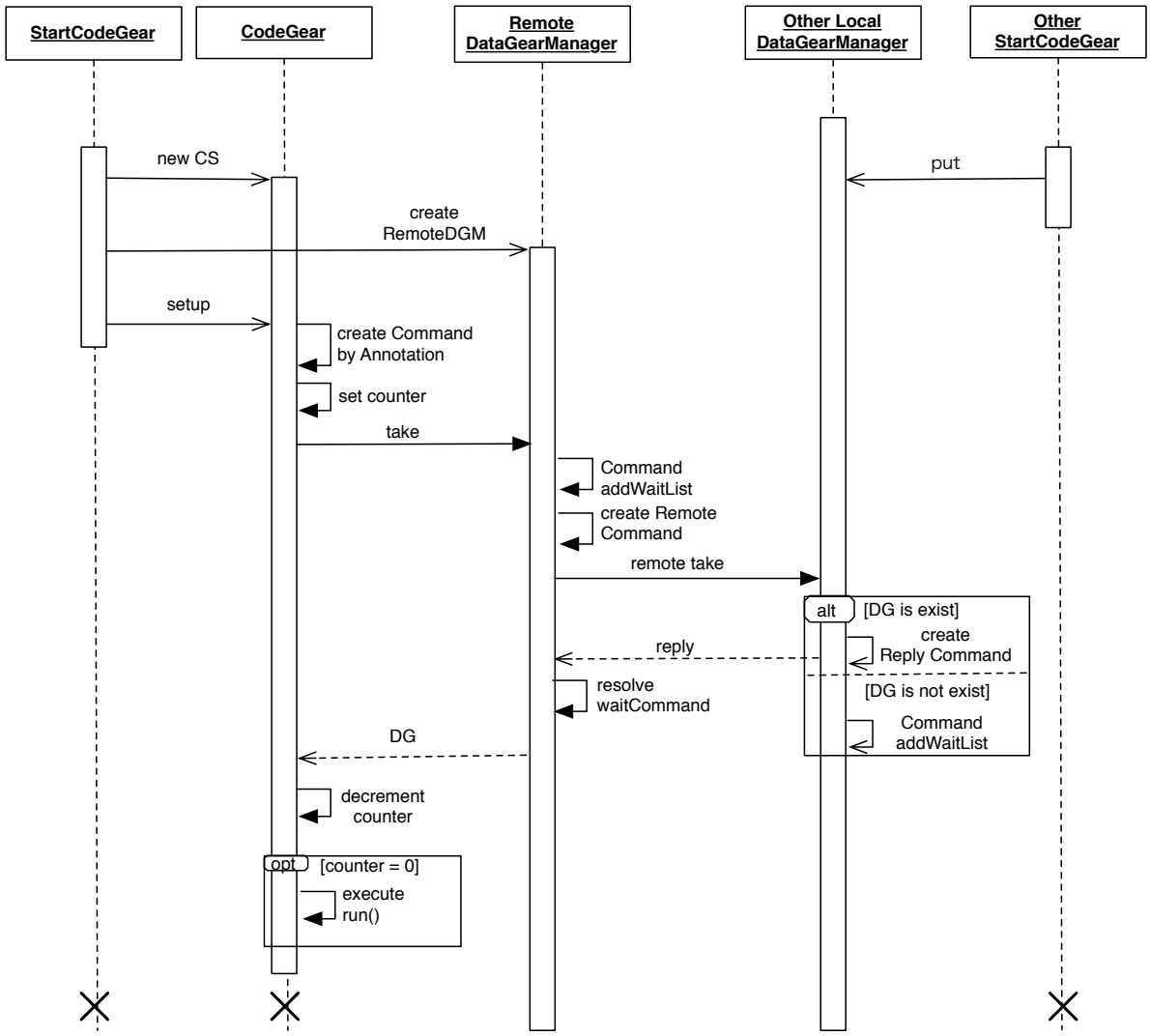

図 4.5: RemoteDGM に Take したときのフロー

StartCG で事前に RemoteDGM を生成しておく。RemoteTake アノテーションから RemoteDGM に対する Take コマンドを生成し実行する。RemoteTake のようにリモートからの応答を待つコマンドは RemoteDGM の waitList に入る。そして、MessagePack 形式に変換した RemoteCommand を作成し、それを RemoteDGM が参照している別ノードの LocalDGM に送る。

それを受け取った側の LocalDGM は、DG があれば REPLY コマンドを生成して送り返

す。もし DG がなければ、リモートから来たコマンドもローカルの場合と同様に LocalDGM の waitList に入る。

REPLY を受け取ると RemoteDGM は waitList に入っていたコマンドを解決する。

第5章　再設計への考察

Christie ではアノテーションを用いることで分離問題を解決することができた。このようにアノテーションを用いた API は Akka や Hazelcast にはないため、より記述性が高いフレームワークとなったと言える。

また、設計をし直したことで Alice より幅広い Meta Computation の実装が容易になった。これにより細かな分散プログラムの実装が可能になった。ロケーション透過性の高い Akka や Hazelcast ではこのようなプログラミングは困難である。

InputDG の指定において、CG に DG を宣言するというのは、DG をそのまま flip できるようにするためであった。逆に言えばそれ以外で DataGear 型でプログラマが利用することは少ない。そのため、DG を宣言せずにアノテーションから生成し完全にメタレイヤーに移すことで、より分かりやすい記述が可能である。flip する場合は、key を指定するだけで行えるようにすべきである。

また、put/flip する際に DGM 名を直接指定する書き方も、まだひと目でアウトプットしている部分が分かるようなシンタックスではないため、改善の余地がある。

第6章　結論

6.1　まとめ

本研究では、まず分散フレームワークに必要な要件を洗い出し、Akka、Hazelcast と比較しながら分散フレームワーク Alice が分散性を意識して記述できる特徴をもつことを示した。また、Alice の持つ Code Segment/Data Segment の計算モデルや記述方法、Meta Computation についてを説明し、Alice で NAT 越えを実現するための手法を示した。さらに現状の Alice の問題点として、NAT 越えをするために必要な Local Data Gear Manager の複数立ち上げができないこと、分散プログラムのテストがしづらいこと、API シンタックスの分離により信頼性が損なわれていること、型の整合性がとれていないことなどを示し、再設計の必要性を述べた。そして、分散フレームワーク Christie の設計を示し、Code Gear Manager という Code Gear/Data Gear の管理機構を挟むことで Data Gear Manager を複数立ち上げられるようにした。これによりテストが容易になり、提案した NAT 越えの手法に対応できる。また、アノテーションを用いることでシンタックスの分離問題を解決し、さらに型整合のとれたより信頼性の高い記述が可能になったことを示した。そして、これら実装した Christie に対しても更に改善すべき点を考察した。

6.2　今後の課題

TopologyManager の実装

Alice と同じく、静的・動的なトポロジー管理のできる TopologyManager の実装が必要である。Christie では複数の LocalDSM が立ち上げ可能なため、TopologyManager での NAT 超えも実装し実用性があるかを検証するまた、通信の信頼性を保証するために、TopologyManager がダウンした際に新たな TopologyManager を立ち上げる Meta Computation も必要だと考える。

実用性の検証

本論文では Christie の設計と基本実装までを行ったが、それがどれほどの分散性能を持っているのかはまだ計測していない。CG/DG のプログラミングモデルなどの基本的には Alice と同じであるが、アノテーションの処理がどれほどのオーバーヘッドに繋がっているか現時点では不明である。そのため、Alice と同等の速度性能を持っているか、コードの量や複雑度は抑えられているかなどを分散処理の例題を用いて測定する必要がある。

Jungle との統合

DGM は一種のデータベースであると述べたが、現状の DGM はデータベースに必要なトランザクションを持っていない。当研究室で開発している Jungle データベースはトランザクションを持っており、更にマージ可能な差分管理システムを持っている。そのため Jungle データベースとの統合することで、DGM への操作を信頼性高くすることが望ましい。

GearsOS への移行

GearsOS はまだ開発途中であったため、本論文の作成時点では Christie のような分散機能を実装することが叶わなかった。GearsOS ではモデル検査機構 akasya [15] があるため、待ちに入っている key の put し忘れなどをコンパイルの段階で見つけることができる。GearsOS 上で分散プログラミングができればより信頼性の高いプログラミングが期待できるため、将来的には Christie を GearsOS の分散機構として取り込みたい。

GearsOS に Christie を移行するには、GearsOS に Java のアノテーションに相当する Meta Computation を実装する必要がある。そして Christie では実現できなかったアノテーションからの変数の自動生成が行えれば更にプログラミングしやすい API になると考えられる。

付録

独自のアノテーション定義

Christie のアノテーションの実装方法と、そのアノテーションから take を実行する部分を解説する。

ソースコード 6.1、6.2 が Christie 独自のアノテーションの定義である。

```
1 @Target(ElementType.FIELD)
2 @Retention(RetentionPolicy.RUNTIME)
3 public @interface Take {
4     String value();
5 }
```

ソースコード 6.1: Take の実装

```
1 @Target(ElementType.FIELD)
2 @Retention(RetentionPolicy.RUNTIME)
3 public @interface RemoteTake {
4     String dgmName();
5     String key();
6 }
```

ソースコード 6.2: RemoteTake の実装

@Target や@Retention はアノテーション定義のためのアノテーション、メタアノテーションである。@Target には、フィールドやメソッド、コンストラクタなど、このアノテーションの付加対象となる構文要素が何かを記述する。@Retention には、SOURCE・CLASS・RUNTIME が選択でき、アノテーションで付加された情報がどの段階まで保持されるかを定義する。reflectionAPI を利用するには RUNTIME でなければならないため、Christie のアノテーションの@Retention は全て RUNTIME である。

定義したアノテーションの仕様例がソースコード 6.3、6.4 である。

```
1 @Take(”count”)
2 public DataGear<Integer> count = new DataGear<>();
```

ソースコード 6.3: Take アノテーションの使用例

```
1 @RemoteTake(dgmName = ”remote”, key = ”count”)
2 public DataGear<Integer> count = new DataGear<>();
```

ソースコード 6.4: RemoteTake アノテーションの使用例

アノテーションを使う際、() 内に記述する値がソースコード 6.1 の value やソースコード 6.2 の dsmName といったキーに保存される。通常キーに対して値を入れる場合は、ソースコード 6.4 のように key=の形で記述しなければならないが、Take のようにキーが 1 つの場合、キー名を value にすることでその記述を省略することができる。

setup メソッド内では生成されたフィールドに対してアノテーションを含めた情報を処理している。これには Java の reflectionAPI が使用されている。reflectionAPI では対象となるクラスのフィールドやメソッド、それに対するアノテーションやアノテーションが保持するキーにアクセスすることができる。ソースコード 6.5 は setup メソッド内で reflectionAPI を用いてアノテーションから Take コマンドを作成する部分である。

```
1 for (Field field : this.getClass().getDeclaredFields()) {
2     if (field.isAnnotationPresent(Take.class)) {
3         Take ano = field.getAnnotation(Take.class);
4         setTakeCommand(”local”, ano.value(), initDataGear(field, ano.value()));
5     } else if (field.isAnnotationPresent(RemoteTake.class)) {
6         RemoteTake ano = field.getAnnotation(RemoteTake.class);
7         setTakeCommand(ano.dgmName(), ano.key(), initDataGear(field, ano.key()));
8     }
9 }
```

ソースコード 6.5: reflectionAPI でフィールドの情報を取得

フィールドから取得した DG とアノテーションから取得した key からインプットコマンド (TAKE/PEEK) を生成し、DGM へ送って実行する。

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准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伊波立樹さん、他フレームワークの調査に協力してくださった清水隆博さん、赤堀貴一さん、浜瀬裕暉さん、大城由也さん、並びに信頼研究室の皆様に感謝いたします。先行研究である Alice, Federated Linda, Jungle, TreeVNC がなければ本研究はありませんでした。 これら先行研究の設計や実装に関わった全ての先輩方に感謝いたします。

また、本フレームワークの名前の由来となったクリスティー式戦車の生みの親、ジョン・W・クリスティーに敬意を評します。

最後に、日々の研究生活を支えてくださった新里幸恵さん、大嶺志歩さん、阿波連知恵さん、米須智子さん、菱田正和さん、情報工学科の方々、そして家族に心より感謝いたします。

参考文献

[1] 安村恭一, 河野真治. 動的ルーティングによりタプル配信を行なう分散タプルスペース federated linda. 日本ソフトウェア科学会第 22 回大会論文集, September 2005.

[2] Akka documentation. `https://akka.io/docs/`. Accessed: 2018/02/3(Sat).

[3] Hazelcast documentation. `https://hazelcast.org/documentation/`. Accessed: 2018/02/3(Sat).

[4] 赤嶺一樹, 河野真治. Data segment api を用いた分散フレームワークの設計. 日本ソフトウェア科学会第 28 回大会, September 2011.

[5] 杉本優, 河野真治. 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), May 2013.

[6] 杉本優. 分散フレームワーク alice 上の meta computation と応用. Master's thesis, 琉球大学 大学院理工学研究科 情報工学専攻, 2015.

[7] 谷成雄. 授業やゼミ向けの画面共有システム treevnc の設計と実装. Master's thesis, 琉球大学 大学院理工学研究科 情報工学専攻, 2014.

[8] 照屋のぞみ, 河野真治. 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, January 2016.

[9] Messagepack documentation. `http://msgpack.org/`. Accessed: 2018/02/3(Sat).

[10] Dot language documentation. `http://www.graphviz.org/`. Accessed: 2018/02/3(Sat).

[11] 大城信康, 杉本優, 河野真治, 永山辰巳. 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第 30 回大会論文集, September 2013.

[12] 照屋のぞみ, 河野真治. 分散システム向けの topology manager の改良. 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), May 2016.

[13] 小久保翔平. Code segment と data segment を持つ gears os の 設計. Master's thesis, 琉球大学 大学院理工学研究科 情報工学専攻, 2016.

[14] Javassist documentation. `http://jboss-javassist.github.io/javassist/`. Accessed: 2018/02/3(Sat).

[15] 比嘉健太. メタ計算を用いた continuation based c の検証手法. Master's thesis, 琉球大学 大学院理工学研究科 情報工学専攻, 2017.