

分散フレームワーク Alice 上の Meta
Computation と応用

Meta Computation on Distributed
framework Alice and apply

平成26年度 学位論文(修士)



琉球大学大学院 理工学研究科
情報工学専攻

杉本 優

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) Mohammad Reza Asharif

印

(副査) 名嘉村 盛和

印

(副査) 和田 知久

要旨

スマートフォンやタブレット端末の普及率が年々増加している。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が要求される。ここでいう信頼性は定められた環境下で安定して期待された動作を行うことをさす。スケーラビリティ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。

また、CPUが発熱の問題からメニコアが主流になっているため、プログラムには並列性も求められる。しかし、これら全てをもつ分散プログラムをユーザーがーから記述することは容易ではない。

そこで本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、分散フレームワーク Alice はそのプロトタイプとして開発されたものである。プロトタイプであるため、実際に Alice を用いてさまざまな例題を記述すると、Alice に足りない機能や問題点が浮かび上がってきた。

本研究では Alice に Alice の Computation の制御を行う Meta Computation を追加した。そして実用的なアプリケーションを Alice 上で実装することで Alice の有効性を確認した。

Abstract

It increases in rate of diffusion of the smart phone and tablet device every year. And people who use service on Internet increase. It would be cause the huge increase in active users of Internet services. So the service which on Internet need scalability and reliability. The program should keep quality of service while increasing number of users by additional resources and perform same action under same environment with same code .The scalability and reliability are the performance standard. Moreover the program need concurrency because computer has many cpu. But it is difficult to create program which has all of them.

We develop an distributed programming framework Alice which uses Data Segment and Code Segment as programming units. But we find many problems and need functions for programming.

We add Meta Computation which control Alice Computation. And We show usability, creating application with Alice.

目次

第 1 章	序論	1
1.1	研究背景と目的	1
1.2	論文の構成	1
第 2 章	分散フレームワーク Alice の概要	2
2.1	Data Segment と Code Segment	2
2.2	Computation と Meta Computation	3
2.3	Alice の実装	3
2.3.1	Data Segment	3
2.3.2	Code Segment	7
2.3.3	Code Segment の実行方法	7
2.3.4	Code Segment の記述方法	8
2.3.5	Meta Data Segment	9
2.3.6	Meta Code Segment	9
2.3.7	Topology Manager	10
2.4	Alice によるプログラミング手法	11
第 3 章	Alice を使った例題	13
3.1	水族館ゲーム	13
3.1.1	処理の流れ	13
3.2	bitonic sort	14
3.2.1	処理の流れ	14
3.3	分散データベース Jungle	16
3.4	AliceVNC	17
3.4.1	AliceVNC の原理	18
第 4 章	Alice の新機能	20
4.1	Dynamic Topology への対応	20
4.2	Keep Alive	21
4.3	切断時の処理	23
4.4	Topology の再構成	24
4.5	再接続の処理	26
4.6	Multicast Data Segment	27

4.6.1	Multicast Data Segment の制限	27
4.6.2	Multicast Data Segment Manager	28
第 5 章	改善点	30
5.1	並列環境における改善	30
5.1.1	SEDA Architecture	30
5.1.2	Data Segment の再構成 (flip 機能の追加)	31
5.1.3	Data Segment のデータ表現の追加	32
5.2	分散環境における改善	32
5.2.1	Data Segment のデータ表現の変更	32
5.2.2	パケットの再設計	34
第 6 章	分散フレームワーク Alice の評価	38
6.1	並列環境の改善効果の測定	38
6.2	分散環境の改善効果の測定	39
6.2.1	TORQUE Resource Manager を用いた実験方法	39
6.2.2	実験概要	40
6.3	考察	43
6.4	TreeVNC との Code の比較	44
第 7 章	結論	45
7.1	まとめ	45
7.2	今後の課題	45
7.2.1	データの永続性の確保	45
7.2.2	DataSegmentKey の領域分け	45
7.2.3	記述に関する注意点	46
	謝辞	49
	参考文献	50
	発表文献	52

目 次

2.1	Code Segment は Input Data Segment と Output Data Segment が存在する	2
2.2	Input Data Segment と Output Data Segment が Code Segment 間の依存関係を自動的に記述する	3
2.3	Remote DSM は他のノードの Local DSM の proxy	4
2.4	”put” は Queue に Data Segment を追加する	5
2.5	”update” は先頭の Data Segment を取り除き、Data Segment を追加する	6
2.6	”take” は Queue の先頭にある Data Segment を読み込む。	6
2.7	”peek” は Queue の先頭にある Data Segment を読み込む。読み込まれた Data Segment は残る。	7
2.8	Meta Code Segment と Code Segment は無関係ではない	10
2.9	Topology Manager はトポロジーファイルの記述に従ってトポロジーを生成する	11
2.10	Topology Manager と Topology Node 間の通信	12
3.1	JavaFx 水族館ゲーム	13
3.2	データの伝搬の様子	15
3.3	width=100mm	15
3.4	Alice における bitonic sort の動き	16
3.5	非破壊的なデータの編集	17
3.6	VNC の構造	18
3.7	TreeVNC, AliceVNC の構造	18
3.8	AliceVNC の構造	19
4.1	Dynamic Topology Manager と Topology Node 間の通信	22
4.2	keepAlive の仕組み	23
4.3	切断ノードの検知	25
4.4	接続すべきノード情報の送信	26
4.5	再構成の完了	26
4.6	Multicast Data Segment	27
6.1	TORQUE の構成	39
6.2	100 周にかかる時間を計測し、1 周あたりの平均時間を求める	40
6.3	10 bytes のデータを 100 周させたときの 1 周にかかる平均時間	42

6.4	100 Kbytes のデータを 100 周させたときの 1 周にかかる平均時間	42
6.5	TCP_NODELAY の有無の比較	43

表 目 次

3.1	ポート一本あたりの通信量 (N は Node 数、M は Tree の子供の数)	18
5.1	CommandMessage の変数名の説明	35
6.1	実行環境の詳細	38
6.2	bitonic sort の結果	38
6.3	ブレードサーバーの詳細	41
6.4	仮想クラスタの詳細	41
6.5	コードの増加量	44

第1章 序論

1.1 研究背景と目的

スマートフォンやタブレット端末の普及率が年々増加している。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が要求される。ここでいう信頼性は定められた環境下で安定して期待しされた動作を行うことをさす。スケーラビリティ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。また、CPUが発熱の問題からマルチコアが主流になっているため、プログラムには並列性も求められる。しかし、これら全てをもつ分散プログラムをユーザーから記述することは容易ではない。

そこで本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。Alice は Java で実装され、ノード間の通信のための API が提供されている。また、オーバーレイネットワークを自動的に構成する Topology Manager という機能を持つため、分散プログラムのシミュレーションを行うことができる。

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動的なトポロジー、つまりノードの切断や再接続に対応しなければならない。また、プログラムによっては切断や再接続等の状態に対応した処理を行いたい場合がある。

本研究では、Alice に動的なトポロジーを管理構成する機能と Alice の Computation の制御を行う Meta Computation を実装した。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切断や再接続の状況に応じた処理を元のコードを変更することなく再接続、切断時の処理を指定することができる。また、Alice の実行速度の改善を並列処理と分散処理の両方の観点から行い、その改効果を計測した。さらに TreeVNC と Alice を用いて実装した AliceVNC の比較をコードの観点から評価を行った。

1.2 論文の構成

第2章 分散フレームワーク Alice の概要

この章では、Alice の計算モデルを説明した後、実際にどのように実装されているかを説明する。

2.1 Data Segment と Code Segment

Alice はデータを Data Segment、タスクを Code Segment という単位に分割してプログラミングを行なう。Code Segment は Input Data Segment と Output Data Segment を持つ (図 2.1)。Input Data Segment はその Code Segment を実行するために必要なデータ群であり、Output Data Segment は Code Segment が計算を行った結果を格納するデータ群である。

Data Segment 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。Input Data Segment を取得する際と Output Data Segment を保存する際にデータベースに通信を行う。取得する Data Segment は Data Segment に対して一意である KEY を用いて指定する。Input Data Segment の取得に失敗した場合、Code Segment は Database で Data Segment の待ち合わせを行う。Input Data Segment が揃った場合に Code Segment は実行される。

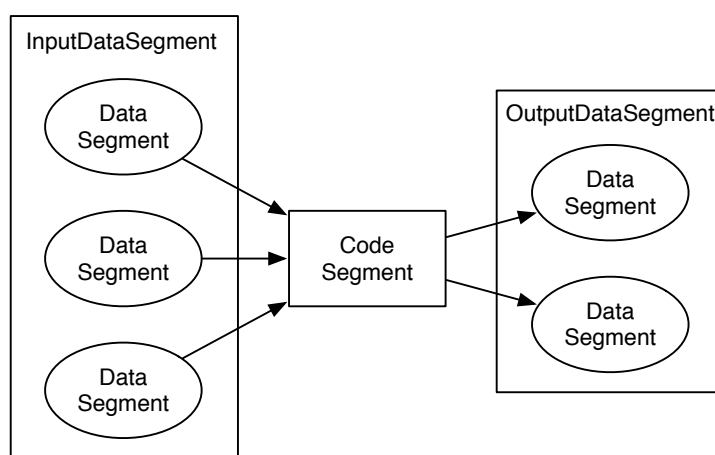


図 2.1: Code Segment は Input Data Segment と Output Data Segment が存在する

Code Segment に依存するデータ群であると出力されるデータ群を記述することにより、Code Segment が実行される順番が決定される (図 2.2)。データの依存関係にない Code

Segment は並列実行が可能であるため、並列度を上げるためには Code Segment 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。

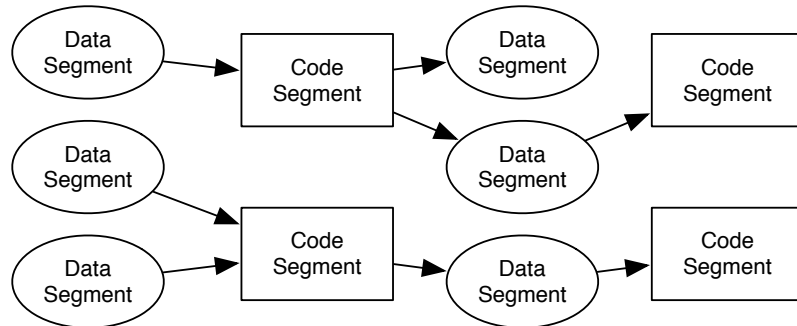


図 2.2: Input Data Segment と Output Data Segment が Code Segment 間の依存関係を自動的に記述する

2.2 Computation と Meta Computation

Alice の Computation は 2.1 で示したように、key で指し示される Data Segment を待ち合わせて Code Segment を実行させるというものである。アプリケーションを作成するためにはアルゴリズムを Alice の Computation で表現する必要がある。

また、アプリケーションで Alice の Computation を設定する Computation を利用することができる。この Computation を Meta Computation と呼ぶ。

例えば、Alice のトポロジーと再接続の管理は Meta Computation とみなすことができる。Code Segment 単位ではトポロジーや切断を想定せずにアプリケーションを記述することができる。トポロジーの Meta Computation を利用することで、トポロジーが 2 分木から n 分木に変わってもアプリケーションは動く。再接続の Meta Computation を利用すれば、アプリケーションはノードが再接続してきた場合でも、切断していないノードと同様に動かすことができる。

2.3 Alice の実装

2.3.1 Data Segment

Data Segment は数値や文字列などのデータを構造的に保持する。Alice は Data Segment をデータベースとして扱っている。Data Segment には必ず対になる Key が存在する。つまり Key Value Store として考える事ができる。

しかし、Alice のデータベースは通常の KVS とは異なっている点がある。通常の KVS はプログラミング言語の連想配列や Map と同様に「Key (キー)」と「Value (値)」が

ペアとなっている。そのため 1 つの Key に対して値は 1 つである。しかし、Alice の場合は「Key」と「Queue」がペアとなっているため、Key に対して複数回 put できる。従って取得できる Value も複数存在する。つまり Key Value Queue として捉える事ができる。

Data Segment は Data Segment Manager (以下 DSM) によって管理されている。ノード毎に Local DSM と Remote DSM が存在する。Local DSM は各ノード固有の KVS となっている。従って Remote DSM を指定する Key はノード内部で unique なものである。Remote DSM は他のノードの Local DSM の proxy と考えられる。つまり Remote DSM は複数存在し、それぞれに対応するノードは異なる。

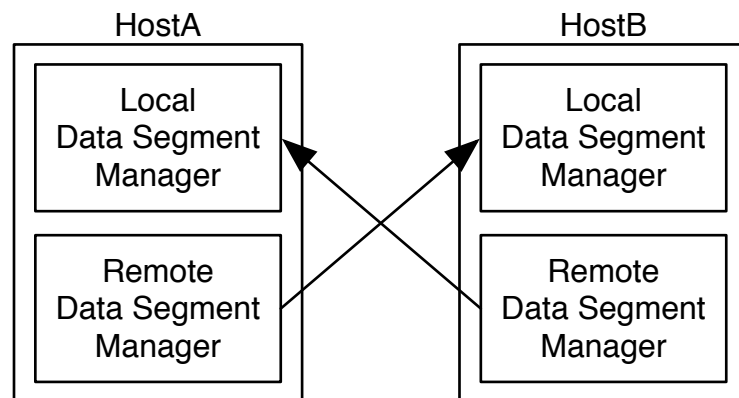


図 2.3: Remote DSM は他のノードの Local DSM の proxy

KVQ へのアクセスはノード内部で逐次化される。それ以外は、すべて Java の Thread Pool により並列実行される。

Data Segment の表現

Data Segment の表現には Message Pack を利用している。Message Pack に関して Java におけるデータ表現は以下の 3 種類があり、制限を伴うが互いに変換可能である。

- 一般的な Java のクラスオブジェクト
- MessagePack for Java の Value オブジェクト
- byte[] で表現された binary

Data Segment API の内部においてデータは、一般的な Java のクラスオブジェクトまたは byteArray で表現された binary で表現されている。Local からデータが put された場合は一般的な Java のクラスオブジェクトの状態 で enqueue される。Remote からデータが put されると byteArray で表現された binary の状態 で enqueue される。

ユーザーが一般的なクラスを IDL (Interface Definition Language) のように用いてデータを表現することができる。この場合、クラス宣言時に `@Message` というアノテーションをつける必要がある。もちろん、`MessagePack` で扱うことのできるデータのみをフィールドに入れなければならない。

`Remote` に対して `put` できるデータは、`@Message` をもつクラスオブジェクトか `MessagePack` で扱える型に限られる。

Data Segment API

以下が用意されている Data Segment API である。これらを用いてデータの送受信を行う。

- `void put(String key, Object val)`
- `void update(String key, Object val)`
- `void peek(Receiver receiver, String key)`
- `void take(Receiver receiver, String key)`

put

`put` は Data Segment を Queue に追加するための API である。図 2.4 では `key1` に対応する Queue に対して Data Segment を追加している。

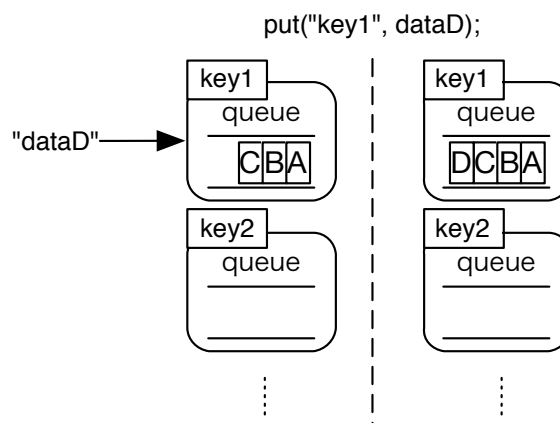


図 2.4: "put" は Queue に Data Segment を追加する

update

update も Queue を追加するための API である。put との違いは、先頭の Data Segment を deQueue してから Data Segment を enqueue することである。(図 2.5)

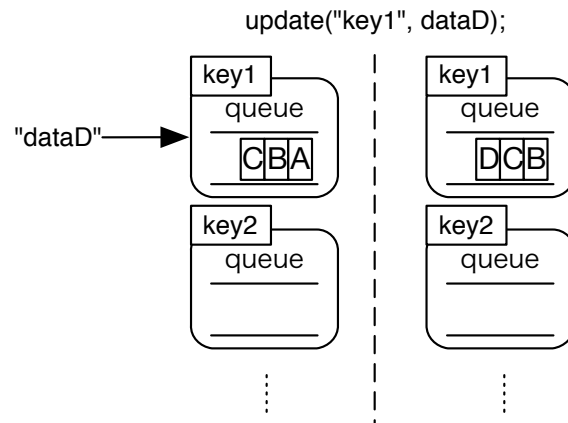


図 2.5: "update" は先頭の Data Segment を取り除き、Data Segment を追加する

take

take は Data Segment を読み込むための API である。読み込まれた Data Segment は削除される。要求した Data Segment が存在しなければ、Code Segment の待ち合わせ (Blocking) が起こる。put や update により Data Segment に更新があった場合、take が直ちに実行される。(図 2.6)

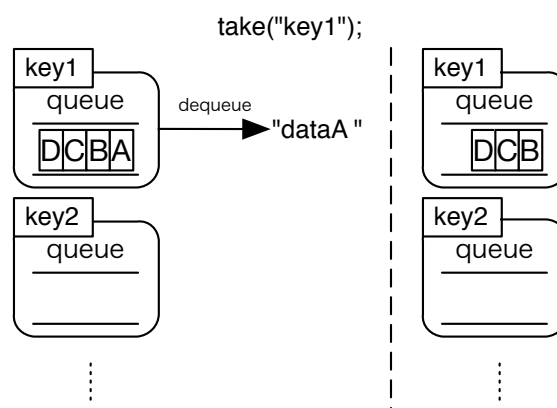


図 2.6: "take" は Queue の先頭にある Data Segment 読み込む。

peek

peek も Data Segment を読み込む API である。Data Segment は読み込まれても削除されない。(図 2.7)

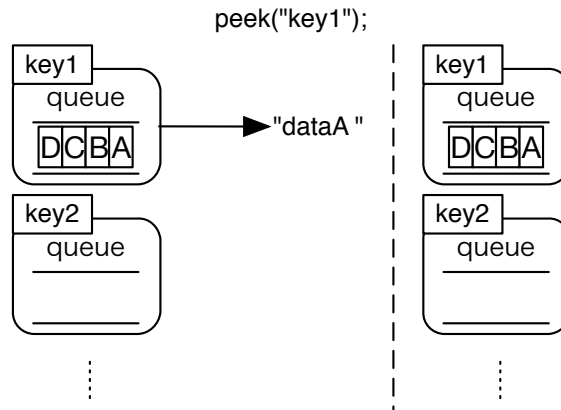


図 2.7: "peek" は Queue の先頭にある Data Segment を読み込む。読み込まれた Data Segment は残る。

2.3.2 Code Segment

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

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

Output Data Segment 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 を用意すべきかは、議論の余地がある。

2.3.3 Code Segment の実行方法

Alice には、Start Code Segment (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 Code Segment がある。

ソースコード 2.1: StartCodeSegment の例

```
1 public class StartCodeSegment extends CodeSegment {
2
3     @Override
4     public void run() {
5         System.out.println("run_StartCodeSegment");
6
7         new TestCodeSegment();
8         System.out.println("create_TestCodeSegment");
9
10        ods.update("local", "key1", "String_data");
11    }
12
13 }
```

Start Code Segment はどの Data Segment にも依存しない。つまり Input Data Segment を持たない。この Code Segment 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。

2.3.4 Code Segment の記述方法

Code Segment をユーザーが記述する際には Code Segment を継承して記述する (ソースコード 2.2)。Code Segment は Input/Output Data Segment Manager を利用することができる。

Input DSM は Code Segment の ids というフィールドを用いてアクセスする。

ソースコード 2.2: CodeSegment の例

```
1 public class TestCodeSegment extends CodeSegment {
2     private Receiver input1 = ids.create(CommandType.TAKE);
3
4     public TestCodeSegment() {
5         input1.setKey("local", "key1");
6     }
7
8     @Override
9     public void run() {
10        int num = input1.asInteger();
11        System.out.println("data_=_ " + num);
12
13        if (num == 10)
14            System.exit(0);
15
16        new TestCodeSegment();
17        ods.update("local", "key1", ++num);
18    }
19 }
```

- Receiver create(CommandType type)

create でコマンドが実行された際に取得される Data Segment が格納される受け皿を作る。引数には CommandType が取られ、指定できる CommandType は PEEK または TAKE である。

- void setKey(String managerKey, String key)

setKey メソッドにより、どこの Data Segment 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することができる。コマンドの結果がレスポンスとして届き次第 Code Segment は実行される。

Output DSM は Code Segment の ods というフィールドを用いてアクセスする。Output DSM は put または update を実行することができる。

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

2.3.5 Meta Data Segment

Meta Data Segment は Data Segment の一種である。Data Segment 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 Code Segment によってその値は変更される。それに対して Meta Data Segment 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 Code Segment によってのみ、その値は変更される。一部の Meta Data Segment 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 では、ノードが Start Code Segment 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 Data Segment 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 Data Segment の伝搬などを行うことができる。

また、Input Data Segment に付随しているものもある。Input Data Segment は Code Segment 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 Data Segment 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 Data Segment 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、Data Segment を put したノードに送り返すことを防ぐことができる。

Meta Data Segment は Data Segment 同様に Data Segment API を用いて取得できる。

2.3.6 Meta Code Segment

Meta Code Segment は Code Segment の一種である。Code Segment 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 Code Segment は Alice を構成するタスクである。つまり Meta Code Segment 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。

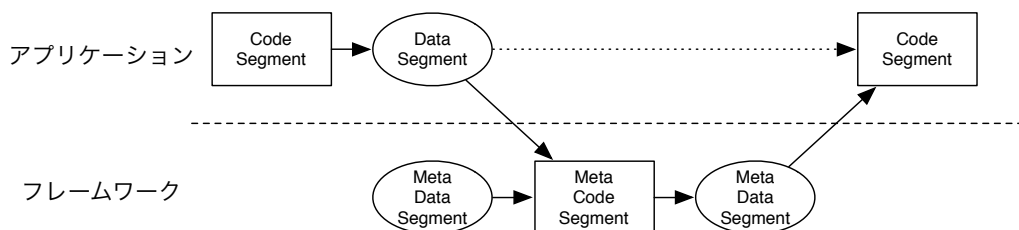


図 2.8: Meta Code Segment と Code Segment は無関係ではない

Meta Code Segment と Code Segment は完全に独立して動作しているわけではない。図 2.8 のように Code Segment と Code Segment の間に Meta Code Segment が実行されている場合も存在する。しかし、間に Meta Code Segment が入った場合でも Code Segment の実行順序には影響を与えることはない。

2.3.7 Topology Manager

Alice は複数のノードで構成され、相互に接続される。通信するノードは URL により直接指定するのではなく Topology Manager で管理する。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 Topology Node の IP アドレス、ポート番号、接続名を送りトポロジーファイルに記述されたとおりにトポロジーを作成する。(図 2.9)

Code Segment 内部で Remote DSM にアクセスする場合は TopologyManager によって指定されたノード内部だけで有効な label(文字列) を使う。これにより特定の URL が Code Segment 内部に記述されることを防いでいる。

トポロジーファイルはグラフ構造を表現するデータ記述する言語の一種である DOT Language と呼ばれる言語で記述する。また、dot コマンドを用いてトポロジーファイルを可視化することができる。

Topology Manager の参加表明処理

Topology Manager への参加表明は、Topology Node 起動時にコマンドライン引数から Topology Manager の IP アドレスとポート番号を指定すればよい。指定された Topology Manager に接続を行うと、Topology Manager 側のキー”hosts”に、自分自身の IP アドレスとポート番号を put する。

参加表明を受け取った Topology Manager は、抽象名を参加表明した Topology Node のキー”host”に put する。その後、Topology Manager 上の Topology Node 名のキーに、接

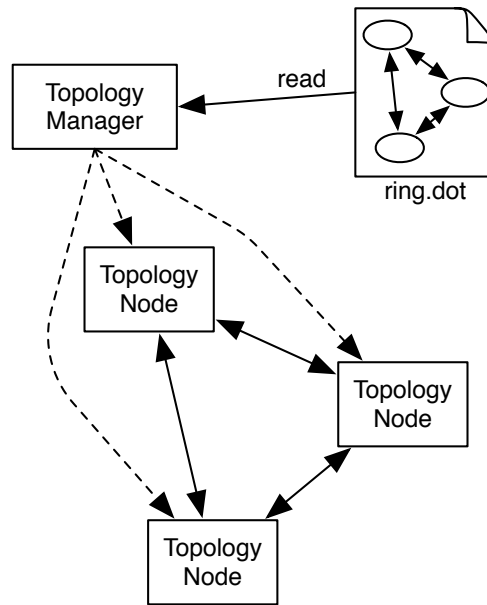


図 2.9: Topology Manager はトポロジーファイルの記述に従ってトポロジーを生成する

続すべき Topology Node の情報 (IP アドレス、ポート番号等) を全て put する。Topology Node は、その情報を 1 つずつ Take し接続処理を行う。全ての接続処理が終わると Topology Manager から Topology Node に対して Start Code Segment の実行命令が出され、アプリケーションが開始される。

2.4 Alice によるプログラミング手法

計算モデルで説明したように、Alice は Code Segment と Data Segment によってプログラミングを行う。Code Segment から別の Code Segment へ Data Segment を引き渡す際、コンストラクタは使わない。Code Segment がデータベースに対して put を行い、別の Code Segment がデータベースに対して peek を行うことで引き渡される。つまり、Code Segment は実行前後にデータベースへ通信が行われるのである。この通信の順序が Code Segment の実行順序を決定している。すなわち、Alice によるプログラミングとはデータベースへの通信管理を行うことであり、プロトコルを設計することと捉える事ができる。

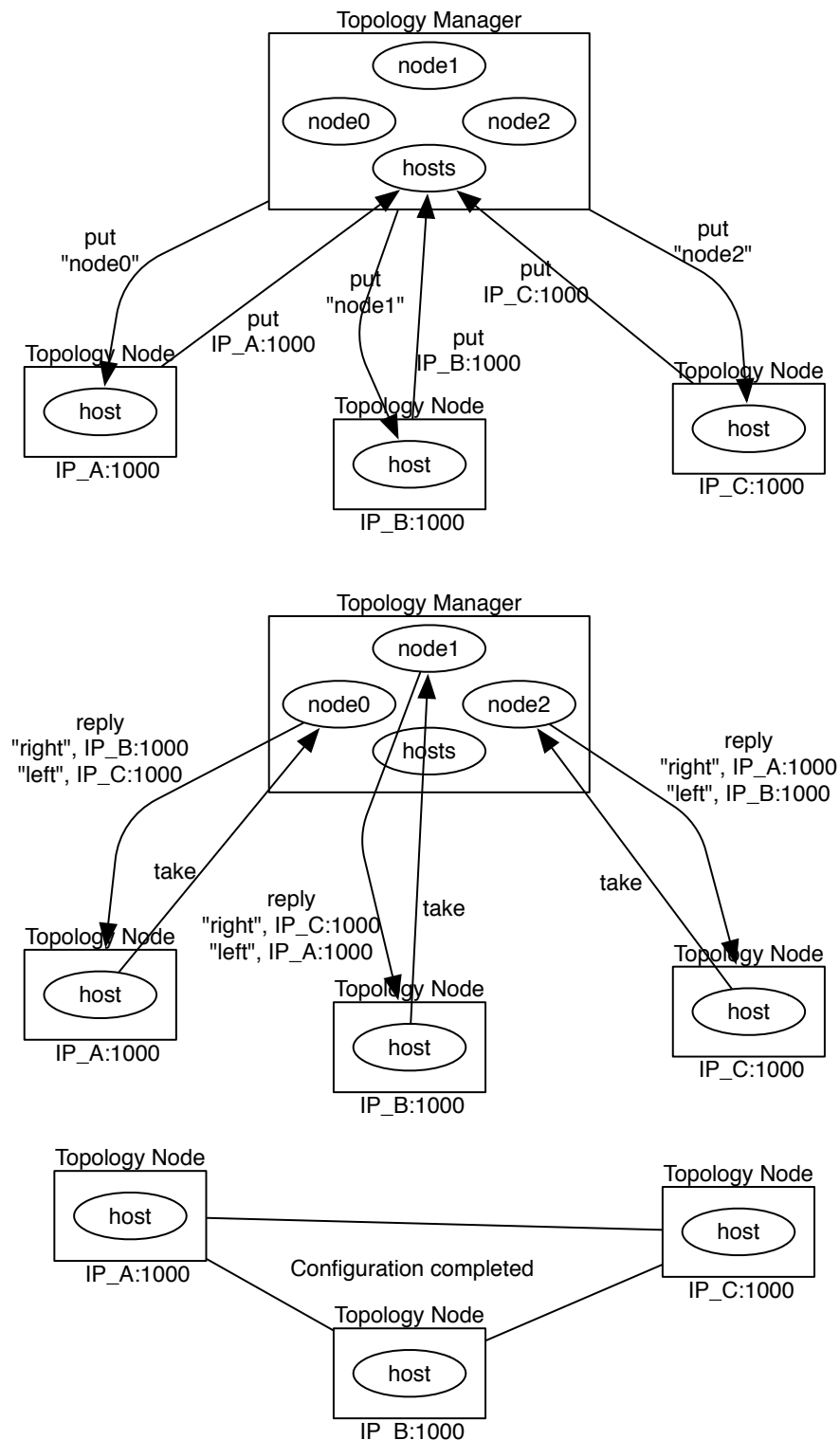


図 2.10: Topology Manager と Topology Node 間の通信

第3章 Aliceを使った例題

この章では Alice を用いて作成されたアプリケーションを紹介する。これらのアプリケーションで Alice の性能テスト、必要な機能の洗い出しを行っている。

3.1 水族館ゲーム

Alice で作成された初めての分散アプリケーションである。Alice に分散アプリケーションを記述する能力があることを確かめるために作成された。過去に Federated Linda でも作成されている。UI として Java7 から組み込まれた JavaFx が使用されている。

アプリケーションを起動すると参加したノード 1 台ごとに 1 つウィンドウが表示される。表示されたウィンドウの中にユーザが操作可能な魚が 1 匹表示されている。魚は画面端まで移動すると自分の画面上からは消え、隣のプレイヤーの画面端に表示される。



図 3.1: JavaFx 水族館ゲーム

3.1.1 処理の流れ

図 3.2 はデータの伝搬の様子をコラボレーションダイアグラムで示したものである。

1. ユーザーが魚を操作することで魚の座標の Data Segment である fishData が更新される。

2. fishData が魚のオブジェクトに座標をセットするための Code Segment である SetLocation に reply される。
3. SetLocation が実行され魚が移動する。
4. 他のノードに更新された fishData を送信するための Code Segment である SendData に fishData が reply される。
5. SendData に自分と接続されているノード一覧の Data Segment である list が reply される。
6. SendData は list を参照して fishData を送信する。
7. 各 client で 2 から 6 が実行され、fishPosition が全体で共有される。

6 では list を参照して、利用可能な Remote Data Segment に Data Segment を put しているが、この利用可能な Remote Data Segment の中には Data Segment を送信してきたものが含まれている。全ての Remote Data Segment に送信してしまうと同じ Data Segment を永遠にやりとりすることになる。しかし、Data Segment は送信元のメタ情報が付加されており、このメタ情報を利用して送信元の Remote Data Segment に対して fishData を送り返すことを防いでいる。

今回の研究で再接続の際の挙動を Meta Computation により変更することができる。水族館の例題ではノードが再接続してきた場合に前回の位置から始められるように挙動が変更されている。

3.2 bitonic sort

bitonic sort は並列ソートであり、Alice がマルチコアに対応していることを確認するため実装した。

3.2.1 処理の流れ

指定された数の乱数を生成し、Sort を行う例題である。また、図 3.4 は Sort されるまでの流れをコラボレーションダイアグラムで示したものである。

1. SetTask (Code Segment) が乱数列を分割して array1 と array2 に put する。
2. reply された Data Segment を Sort (Code Segment) で昇順に整列させる。
3. 整列された配列を分割する。上半分を array1-F、下半分を array1-B に put する。
4. 分割した各数列 (array2) に対しても同様に 2 と 3 を行う。
5. reply された 2 つの Data Segment (array1-B、array2-F) を合体させ、整列させる。

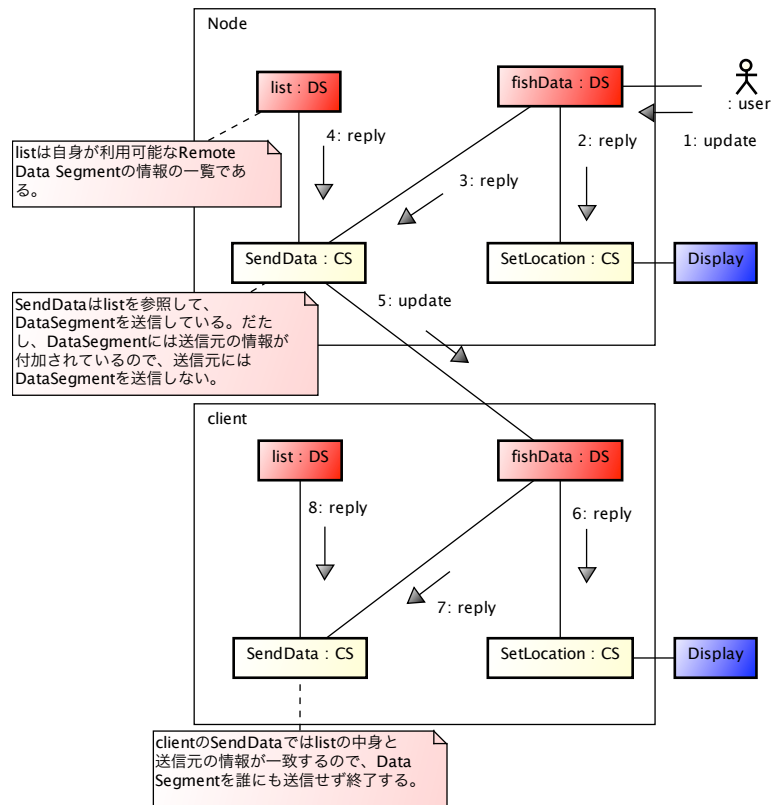


図 3.2: データの伝搬の様子

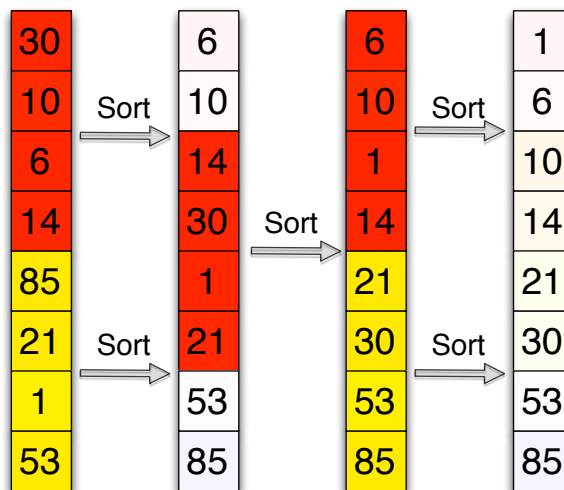


図 3.3: sort flow

6. 整列された配列の上半分を array1-B、下半分を array2-F に put する。
7. reply された 2 つの Data Segment(array1-F、array1-B) を合体させ、整列させる。
8. 整列された配列の上半分を array1-F、下半分を array1-B に put する。
9. array2 に対しても操作 7 と 8 を行う。
10. 5 - 9 を繰り返し行うことで全体が Sort される。

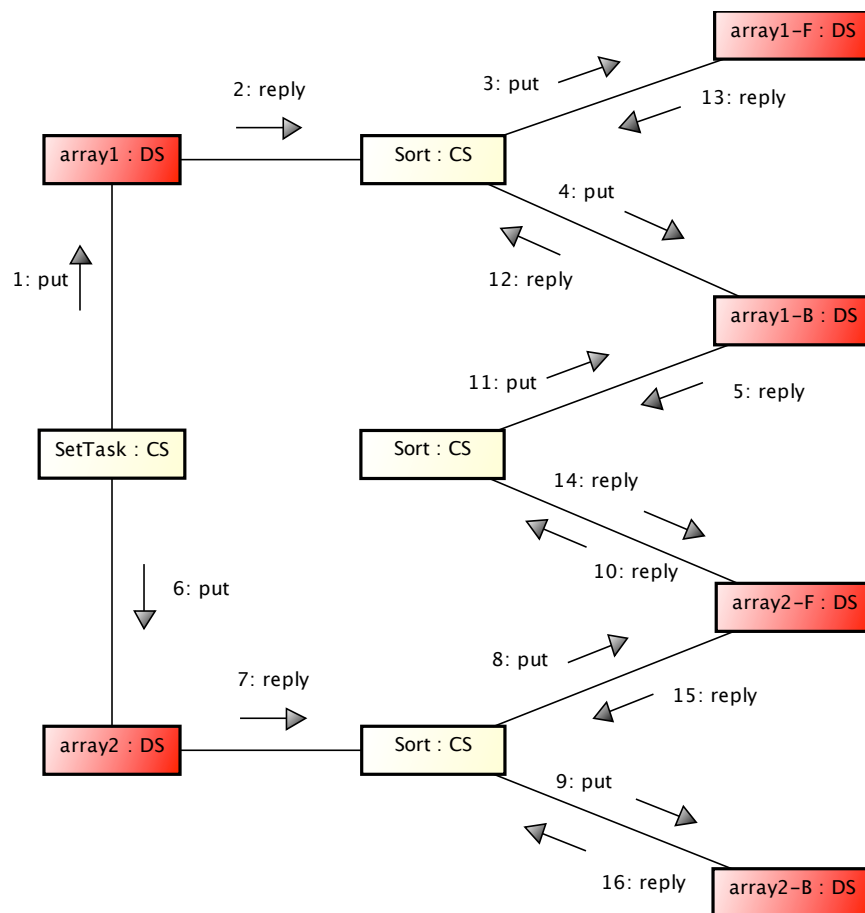


図 3.4: Alice における bitonic sort の動き

3.3 分散データベース Jungle

Jungle はスケーラビリティのある CMS の開発を目指して当研究室で開発されている非破壊的木構造データベースである。非破壊的にデータを編集を行なうため 3.5、過去の全

てのデータを参照することができる。また、編集の際にロックが不要であるため、破壊的木構造に比べスケールアウトがし易い。Jungle はデータの編集の際に log が残すため、障害が発生しても log を読み込ませることにより前回の状態を再現することができる。この log をハードディスクに書き出すことにより永続性を持たせることも可能である。

この木構造データベースを複数接続することにより、可用性と分断耐性をもつ分散データベース Jungle となる。Alice はトポロジーの形成とデータアクセスへの機構を提供している。

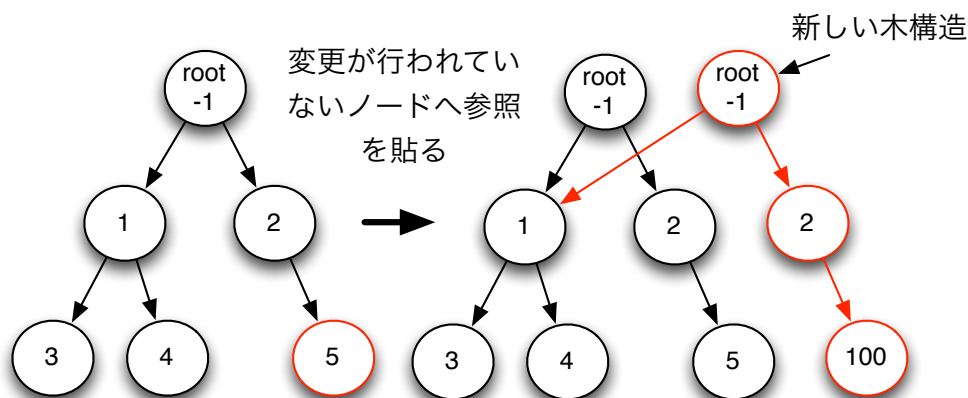


図 3.5: 非破壊的なデータの編集

3.4 AliceVNC

AliceVNC は、当研究室で開発を行っている TreeVNC を Alice を用いて実装された、授業向け画面共有システムである。

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、性能が大幅に落ちるという問題がある (図 3.6)。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 3.7)。TreeVNC は、TightVNC のソースコードを利用して開発されている。

しかし、TreeVNC にも問題が存在する。ノードを木構造にするためアプリケーション内で管理する必要があるが、不特定多数のノードを管理することは容易ではない。また、通信プロトコルは、TightVNC のプロトコルを拡張して利用している。アプリケーション層でプロトコルを拡張する場合、サーバとクライアントの両方のプログラムコードを変更する必要があり、プロトコル処理部の保守性を維持していくことが難しい。

実際に、プロトコルを拡張したことにより予想外のエラーで TreeVNC が強制終了することがある。

さらに、TighVNC のアップデートに対応する必要がある。だが、アップデートによってはパッケージ構成が変更され、元のコードが残っていないことも考えられる。この場

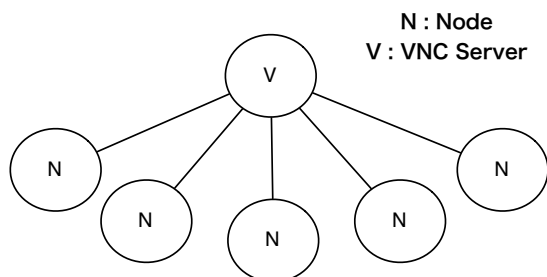


図 3.6: VNC の構造

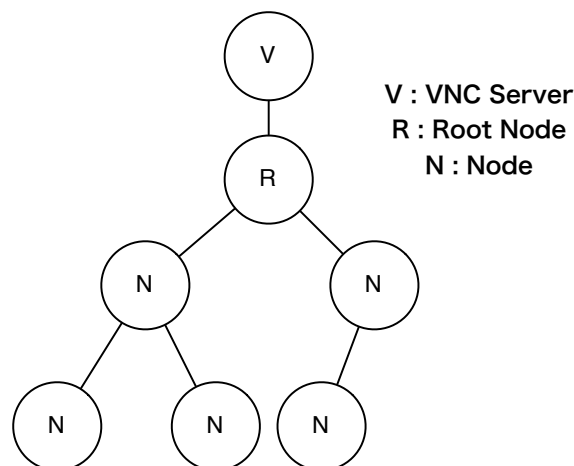


図 3.7: TreeVNC, AliceVNC の構造

合、新しい TightVNC に作成した機能を 1 つずつ移行するしなければならないためコストが高い。

これらの問題は、分散フレームワーク Alice を使うことで解決することができる。ノードの管理は全て Alice が行なうため、プログラマーは画面共有の処理のみ記述すれば良い。プロトコルは Alice 上で構築することで TightVNC のプロトコルと干渉することがないため、エラーが起こった場合は Alice で記述したコードのみを確認すればよい。アップデートの問題に関しては、TightVNC に必要なデータを put する処理さえ追加すれば良いため、問題なく対応できるはずである。

Alice が以上の問題に対応できることを証明するため、また実用的なアプリケーションの記述が可能であるかを検証するため AliceVNC の開発を行った。

3.4.1 AliceVNC の原理

従来の VNC と TreeVNC の構造を比較した図を (図 3.8) に示す。

表 3.1 はポート一本あたりの通信量である。従来の VNC の場合、ポート一本あたりの負荷はノード数に比例して増える。しかし、AliceVNC の場合は Tree の子供の数が一定なので、Node 数に関係なく一定である。通信量が増えるほど、CPU に負荷がかかり性能が低下する。AliceVNC の場合、通信量は一定なので性能が低下せず使用することができる。

表 3.1: ポート一本あたりの通信量 (N は Node 数、M は Tree の子供の数)

	従来の VNC	AliceVNC
通信量	$N * \text{データ量}$	$(M + 1) * \text{データ量}$

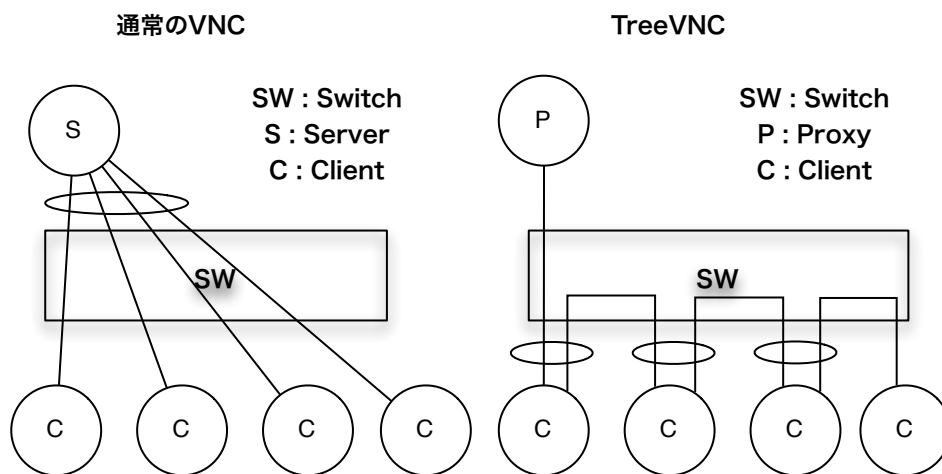


図 3.8: AliceVNC の構造

第4章 Aliceの新機能

水族館の例題によって、Alice を用いて分散プログラムを記述可能であることが証明された。しかし、実用的なアプリケーションを記述するには、まだ機能が足りないと予測される。そこで、当研究室で開発を行っている 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。

4.1 Dynamic Topology への対応

第2章で示したように分散フレームワーク Alice は Topology File を読み込むことで Topology を構成する。つまり、予め参加するノードの台数が決まっている必要がある。また、Topology に全ノードが参加するまでアプリケーションが起動しない。

しかし、実際のアプリケーションでは、参加するノードの数は決まっていないため、Topology を動的に変化させる必要がある。そこで、Topology File の代わりに Topology を指定して起動させた場合 (ソースコード 4.1)、ノードが Topology に従い、順次追加されるように Topology Manager に機能を追加した。

ソースコード 4.1: Dynamic Topology Manager の起動方法

```
1 java -cp Alice.jar alice.topology.manager.TopologyManager -p 10000 --  
   Topology tree --Child NUM
```

現在は、TreeVNC の Topology である木構造のみ指定可能である。引数 Child の後に子供の数を指定することで2分木だけではなく3分木などの Topology を作成することができる。デフォルトでは2分木になっている。

Dynamic Topology Manager の参加表明処理

トポロジーファイルを指定して起動した場合は、図 2.10 で示したように、アプリケーションが起動する前に全ての接続すべき Topology Node の情報が Topology Node に put され、接続処理が行われる。

Dynamic Topology の場合は、新しく Topology Node がアプリケーションに参加するたびに Topology Manager から Topology Node に対して、接続すべき Topology Node の情報が put され、接続処理が行われる。(図 4.1)。

Dynamic Topology への参加表明の方法は、従来の方法と同じく、Topology Node の起動時にコマンドライン引数から Topology Manager の IP アドレスとポート番号を指定するだけでよい。

4.2 Keep Alive

ノード間の通信は、Remote Data Segment に対して put や peek を行うことでのみ発生する。従って、Code Segment の実装次第では、長時間通信が行われない可能性がある。通信が行われない間に Remote Data Segment との接続が切れた場合、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問題が発生し、応答に時間がかかる場合も考えられる。

以上の問題を検知するためにアプリケーションでは Keep Alive という、定期的に heart beat を送り生存確認を行う機能を持つことが望ましい。そこで、Alice 自体に Keep Alive の機能を実装した。

一定時間内にノードから応答がない場合、Keep Alive により、そのノードの Remote Data Segment が切断される。

図 4.2 は、keepAlive の処理をコミュニケーションダイアグラムで示したものである。keepAlive は、タスクとタスクを実行する TaskExecutor と実行順序を管理する Scheduler によって実装されている。タスクの種類には、PING、CLOSE、CREATE があり、TaskExecutor は、タスクの種類に従って処理を行なう。PING は、指定された Remote Data Segment に対して heartbeat を送信する。CLOSE は、指定された Remote Data Segment を削除する。CREATE は PING のタスクを作成し、Scheduler に登録する。

タスクを作成する際に実行時間の指定をする。Scheduler にタスクを登録すると、Scheduler はタスクの実行時間に従って、タスクを実行順に並び替える。

処理の流れ

1. CreateTask (Code Segment) により PING タスクが投入される。
2. TaskExecutor は Scheduler から次に実行すべき PING タスクを受け取る。
3. 指定された時間が訪れると heartbeat が Remote Data Segment (Node B) に送信される。
4. 3 と同時に heartbeat を送信した Remote Data Segment との接続を切断する CLOSE タスクが投入される。
5. 4 で投入された CLOSE タスクが実行されるまでに、Remote Data Segment (Node B) からのレスポンスがあった場合、RemoveTask (Code Segment) により CLOSE タスクが削除される。

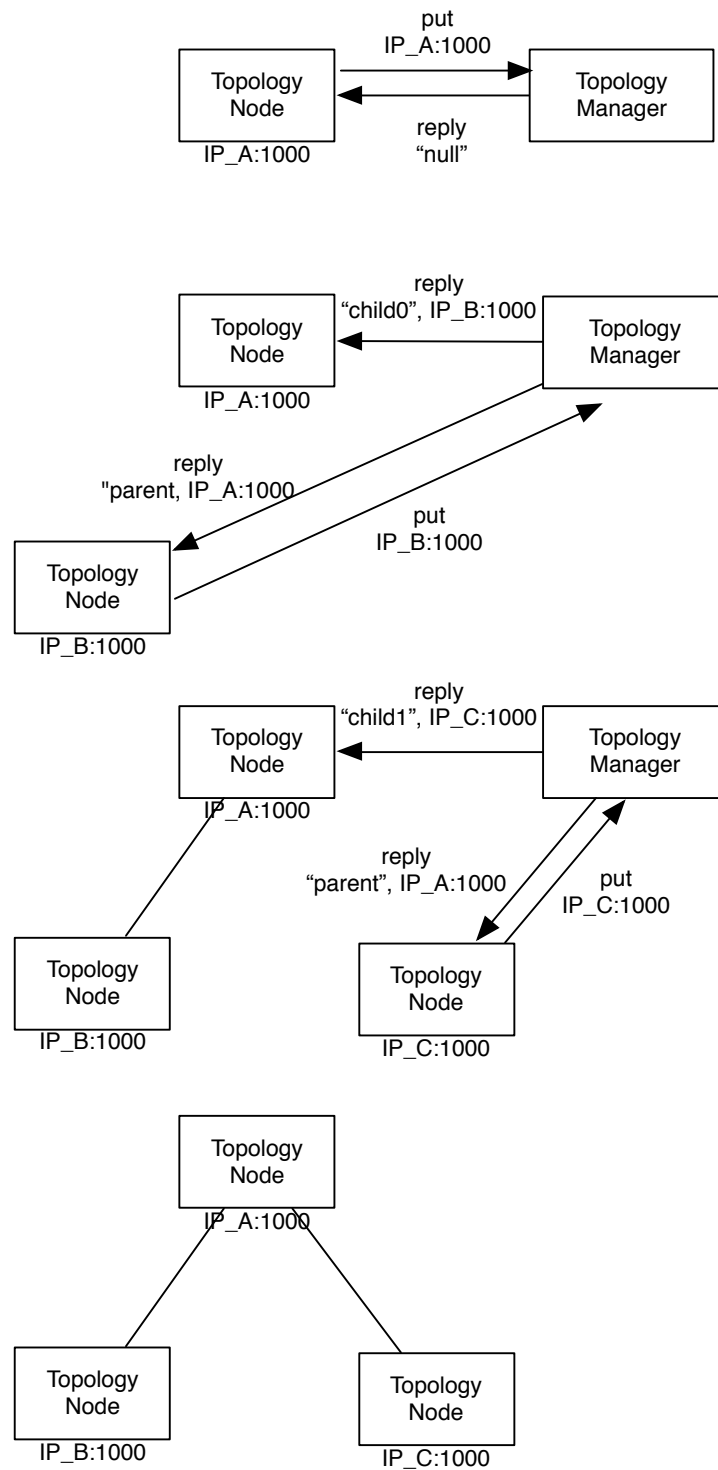


図 4.1: Dynamic Topology Manager と Topology Node 間の通信

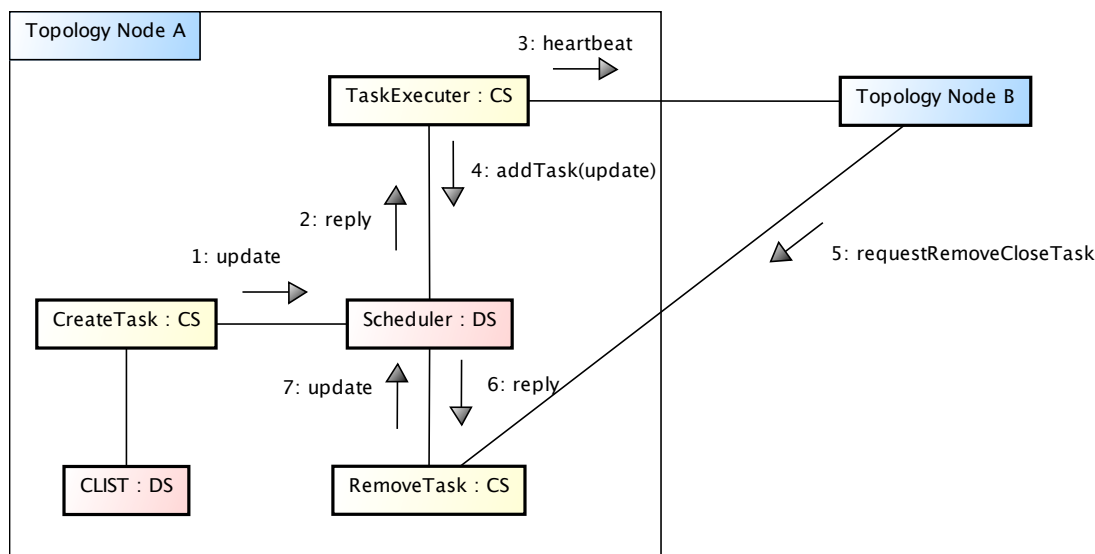


図 4.2: keepAlive の仕組み

1 で作られる PING タスクは、接続している Remote Data Segment の数だけ投入される。”_CLSIT” という key には、アクセス可能な Remote Data Segment の key の一覧が保存されている。CreateTask は、この一欄により動的にタスクを作成している。

以上で説明した処理の流れは、接続状態に問題がない場合である。接続状態に問題があり、CLOSE タスクの実行されるまでにレスポンスがない場合は、CLOSE タスクにより Remote Data Segment が削除される。

heartbeat は、Node A から Node B に一方向に送られる訳ではなく、Node B から Node A にも送られている。

4.3 切断時の処理

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開始時の位置に戻すという処理が実行される。同様に TreeVNC では切断を検知した場合、LostParent というメッセージがトップノードに対して送信される。

以上の例のように、アプリケーションはノードの切断に対する処理を用意したい場合がある。しかし、Alice を用いたアプリケーションの場合、アプリケーション側で検知するのは難しい。切断自体は、Remote Data Segment に対して write または read を行った際に出る Exception により判断することができる。だが、I/O の処理は Code Segment を実行する Thread で行われたい。専用の I/O Thread によって行われるため、Code Segment 内で Exception を捕まえず、例外処理を行なうことができない。

そこで、Alice が切断を検知した際に、任意の Code Segment を実行できる機能 (ClosedEventManager) を追加した。ユーザは ClosedEventManager に Code Segment を登録することで、切断時の処理として実行する Code Segment を指定できる。

ソースコード 4.2: 切断時に実行される Code Segment の登録方法

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

また、切断した Remote Data Segment の情報を利用したい場合は、Code Segment を extends する代わりに Closed Event Code Segment を extends し、用意されている Method を使うことで取得可能である (ソースコード 4.3)。

ソースコード 4.3: CloseEventCodeSegment を継承した CodeSegment

```
1 public class ShowClosedNode extends CloseEventCodeSegment{
2
3     @Override
4     public void run() {
5         ConnectionInfo e = getConnectionInfo();
6         System.out.println("IPAddr□"+ e.addr);
7         System.out.println("port□"+ e.port);
8     }
9 }
```

ClosedEventCodeSegment を継承した Code Segment に、Input Data Segment を追加記述する事ができる。その際は、もちろん Input Data Segment が全て揃うまで Code Segment は実行されない。

4.4 Topology の再構成

ノードは永続的にアプリケーションに参加し続ける訳ではない。目的を果たすとアプリケーションから離脱する。Topology 次第では、アプリケーションに支障をきたす。例えば、AliceVNC は木構造であるため、子ノードを持つノードがアプリケーションから離脱した場合、その子ノードに対してデータを送信することができなくなる。

この問題を解決するには、アプリケーションからノードが切断するたびに Topology の再構成を行なう必要がある。そこで、Dynamic Topology Manager に、Topology の再構成を行う機能を追加した。

図 4.3、4.4、4.5 は Topology の再構成をコラボレーションダイアグラムで表したものである。

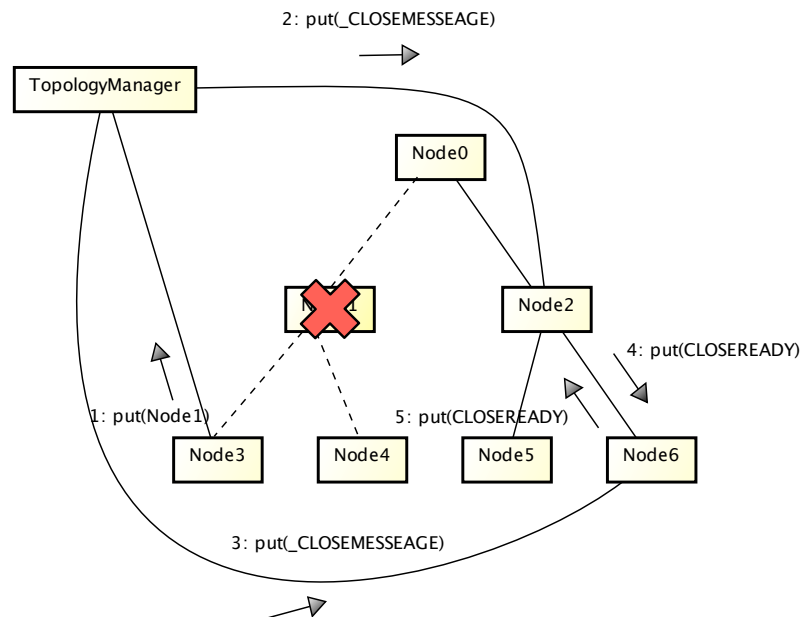


図 4.3: 切断ノードの検知

1. Keep Alive が Node1 の切断を検知すると、Node3 から Topology Manager に対して切断したノードの情報が put される。keep Alive は各ノードで動いているため、実際には Node0 と Node4 も Topology Manager に対して Node1 の情報を put する。
2. Topology Manager は最後にアプリケーションに参加した Node6 とその親 Node2 に対して互いを切断するための準備を行わせる、CLOSEMESSAGE を put する。
3. 切断する準備ができるとお互いに準備完了を知らせる CLOSEREADY を put する。
4. CLOSEREADY を受け取ると切断を行い、Remote Data Segment が使用不可能になる。
5. Node1、Node3、Node4 に対して Node6 の情報を送り、接続を行わせる。

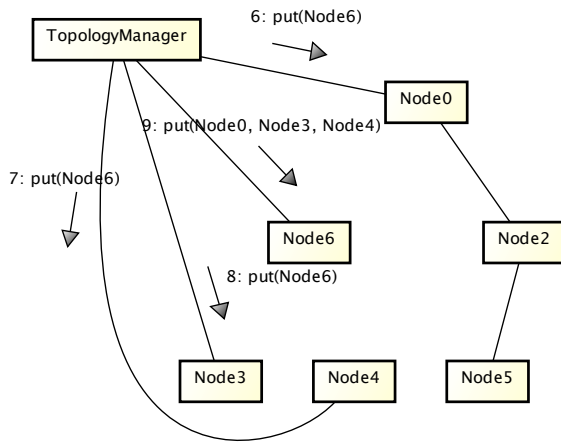


図 4.4: 接続すべきノード情報の送信

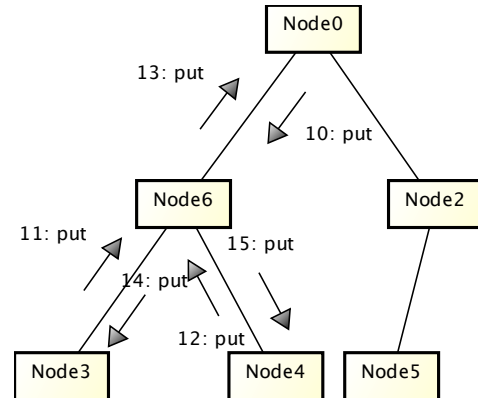


図 4.5: 再構成の完了

- 6. Node6 に対して、Node1、Node3、Node4 の情報を送り、接続を行わせる。
- 7. お互いに Remote Data Segment の名前を贈り合い、Remote Data Segment が利用可能になる。

4.5 再接続の処理

MMORPG では、試合の最中に障害などによりサーバーから離脱したユーザーが、試合が終わるまでに再びサーバーに接続してきた場合に、再びその試合に参加できるような再接続の処理が用意されている。

分散にアプリケーションでは、MMORPG の例のように再接続してきたノードに対して通常の処理とは別の処理を行わせたい場合がある。そこで、Alice に再接続してきたノードに、任意の Code Segment を実行できる機能を追加した。ユーザーは config に Code Segment を継承した Class を登録することで、再接続時に実行される Code Segment を指定することができる。

ソースコード 4.4: 再接続に実行する Code Segment の登録方法

```

1 public class StartAquariumFX {
2     public static void main(String args[]){
3         AquariumConfig conf = new AquariumConfig(args);
4         conf.register(MoveBeforePosition.class);
5         StartCodeSegment cs = new StartCodeSegment();
6         new TopologyNode(conf, cs);
7     }
8 }
    
```

4.6 Multicast Data Segment

TreeVNC には、Multicast を利用して起動している TreeVNC の Root Node の情報の一覧にして表示する接続先自動検索システムという機能がある。この機能により TreeVNC の起動の際に IP アドレスを入力する手間を省くことができる。

現在の Alice は起動時に Topology Manager の IP アドレスを入力する必要があり、TreeVNC の接続先自動検索機能が必要と考えられる。その機能を実現するためには Multicast に対応する必要がある。そこで、同じマルチキャストアドレスを持つ端末を 1 つの Data Segment として扱う Multicast Data Segment を追加した。Multicast Data Segment も他の Data Segment 同様、Data Segment API を用いて扱う。

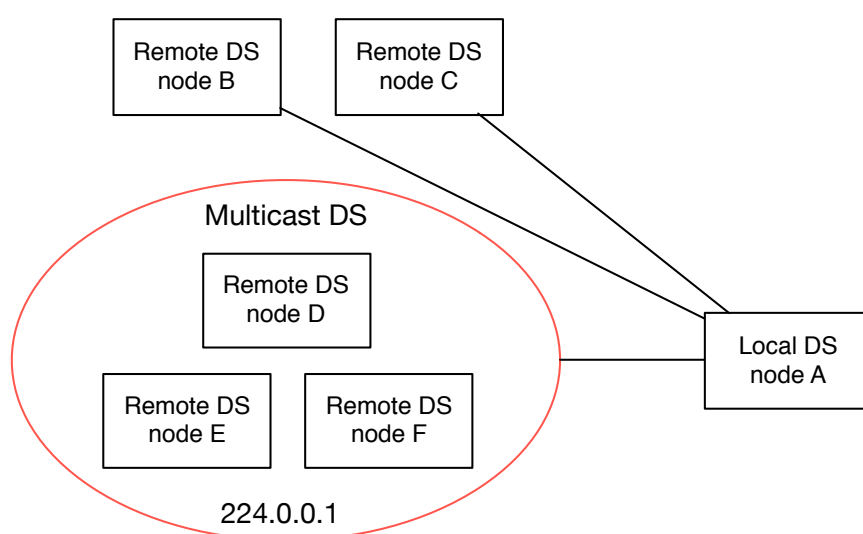


図 4.6: Multicast Data Segment

図 4.6 は Multicast Data Segment を図で表したものである。Multicast Data Segment に対して put を行うと node D、node E、node F の 3 つの node に対してデータが put される。同様に Take を行うと 3 つの node から reply が来る。

4.6.1 Multicast Data Segment の制限

Multicast Data Segment は UDP を用いて実装されている。1 つの UDP のパケットで運ぶことのできるデータが、65507bytes (65535bytes から IP ヘッダの最低サイズ 20bytes と UDP のヘッダのサイズ 8bytes を引いた大きさ) である。現状、分割して送る処理を Multicast Data Segment が持たない。従って、Multicast Data Segment を利用する際には、データのサイズを 65507bytes 以下にしなければならない。

4.6.2 Multicast Data Segment Manager

Multicast Data Segment も Remote Data Segment 同様 Manager を経由して操作を行う。Multicast DSM を作成すると Multicast Data Segment に対する送受信のスレッドが作成される。

- `public static MulticastDataSegmentManager
connectMulticast(String connectionKey ,String MCSTADDR, int port, String
nis, SocketType type)`

DataSegment class の static メソッドである、connectMulticast を呼ぶことで Multicast DSM が作成される。第1引数は Multicast DSM へのアクセスするための key を指定できる。第2引数はマルチキャストアドレスを、第3引数はポート番号を、第4引数はネットワークインターフェイスを指定する。第5引数は SocketType を指定する。SocketType には、Sender と Receiver と Both が存在する。Sender を指定した場合は、第2引数で指定したマルチキャストアドレスに対して送信処理を行うスレッドを作成する。Receiver を指定した場合は、第2引数で指定したマルチキャストアドレスに対して送信されたデータを受信するスレッドを作成する。Both は、送受信両方のスレッドを作成する。

現在、Multicast Data Segment は自動では作成されないため、ユーザー自身で作成する必要がある。

ソースコード 4.5 と 4.6 は実際に Multicast Data Segment を利用した例題である。例題の内容は multicast Data Segment に対して take を行い、取得したデータをインクリメントして再び put を行うものである。

Multicast DSM を作成する際の SocketType の引数は Both であるため、Receiver と Sender 両方作成される。そのため、"multicast" という key に対して put を行うと、自分自身に対しても Data Segment が put される。take もまた同様に自分自身に対しても take が要求され、要求に対して reply を返すことになる。

ソースコード 4.5: multicast DSM の作成例

```

1 public class MulticastStartCodeSegment extends CodeSegment {
2
3     @Override
4     public void run() {
5         DataSegment.connectMulticast("multicast",
6                                     "224.0.0.1",
7                                     10000,
8                                     "en1",
9                                     SokcetType.Both);
10
11         new MulticastIncrement();
12         ods.put("multicast", "num", 0);
13     }
14 }
15 }
```

ソースコード 4.6: multicast Data Segment の使用例

```
1 public class MulticastIncrement extends CodeSegment {
2
3     private Receiver num = ids.create(CommandType.TAKE);
4
5     public MutlicastIncrement() {
6         num.setKey("multicast", "num");
7     }
8
9     @Override
10    public void run() {
11        int num = this.num.asInteger();
12        System.out.println("[CodeSegment] " + num);
13        if (num == 10) System.exit(0);
14
15        new MulticastIncrement();
16
17        ods.put("multicast", "num", ++num);
18    }
19
20 }
```

第5章 改善点

この章では、分散フレームワーク Alice に対して行った改善点を示す。

5.1 並列環境における改善

分散フレームワーク Alice は、並列環境にも対応したフレームワークである。しかし、並列環境に対応していることを確認するために bitonic sort を作成、計測したところ、Data Segment の更新のオーバーヘッドにより、期待した効果を得ることができなかった。その際に、行った改善点を示す。

5.1.1 SEDA Architecture

SEDA Architecture とはマルチコアスレッドを用いて大量の接続を管理し、受け取ったデータを処理ごとに分けられたステージと呼ばれるスレッドに投げ、処理が終わると次のステージにデータを伝搬させていく処理方式である。スループット重視なのでありレスポンスは多段のパイプラインのせいで遅れてしまう。Alice に置いて SEDA を実装するにあたり、データを次のステージにへ伝搬する際、LinkedBlockingQueue を使用している。LinkedBlockingQueue は片方向の連結リストをベースとした Queue 実装である。enqueue / dequeue の操作時の排他制御にはそれぞれ別々のロックオブジェクトが使用されている。そのため、enqueue と dequeue が重なってもロック解除待ちは発生しないが、そのかわりに連結リストの Node オブジェクトの生成操作などが発生してしまうため、enqueue 操作の処理コストが高い。さらに、非力なマシンでは SEDA の効果を得られず、スレッドを切り替えが頻繁に起こりオーバーヘッドになってしまう。

以上の理由から Local Data Segment に対して操作をする際は SEDA を使用せず処理を行なうように変更した。変更前は、Local Data Segment に対して操作する場合、put や peek に沿った Command を作成するステージ (Code Segment が実行されているスレッド)、受け取った Command を処理するステージ、Code Segment に Data Segment をセットするステージ (peek と take の場合) の 2 段または 3 段のパイプラインで構成されていた。これらを 1 つのステージにまとめて処理することで、並列環境における性能を向上させた。

5.1.2 Data Segment の再構成 (flip 機能の追加)

Data Segment API の put、update を呼ぶと Output Data Segment が毎回新しく作成される。そして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に変更を行い、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、このコピーを減らすことで速度改善を行った。

このコピーを無くし、Data Segment の更新におけるオーバーヘッドを減らす方法として Cerium でも良好な結果を得ている flip を用いた。Cerium における flip は、Input Data Segment と Output Data Segment を swap させる API である。(ソースコード 5.1)

ソースコード 5.1: Cerium における flip

```

1 void swap() {
2     void * tmp = readbuf;
3     readbuf = writebuf;
4     writebuf = tmp;
5 }
    
```

ソースコード 5.2: Alice における flip

```

1 public void flip(Receiver receiver) {
2     DataSegment.getLocal().put(receiver.key, receiver.getReceiveData(),
3         null);
3 }
    
```

ソースコード 5.3: flip の使用例

```

1 public class SortPhase extends CodeSegment{
2     private Receiver info = ids.create(CommandType.PEEK);
3
4     public SortPhase(String key){
5         info.setKey(key);
6     }
7
8     @Override
9     public void run() {
10        DataList list1 = info.asClass(DataList.class);
11        Sort.quickSort(list1);
12        ods.flip(info);
13    }
14 }
    
```

Cerium の場合、Output Data Segment は Task が実行された段階ですでに用意されている。そのためデータを Output Data Segment に書き込む前に flip を呼ぶ。Alice の場合、put または update を呼んだ段階で Output Data Segment が作られるため、ソースコード 5.3 のように Input Data Segment である Receiver を flip メソッドに引数として渡すことで、無駄なコピーを減らす。

5.1.3 Data Segment のデータ表現の追加

変更前は Data Segment のデータ表現は Message Pack for Java の Value オブジェクトのみを用いて表現していた。Value オブジェクトとは、Message Pack のバイナリにシリアル化できる型のみで構成された Java のオブジェクトであり、自己記述形式のデータ形式となっている。そのため、ArrayValue を用いることにより、ユーザーはデータを後からつなげたりすることも可能である。

この Value オブジェクトの特徴の 1 つに、通信に関わる際のシリアル化・デシリアル化を高速に行えることがある。この特徴を用いて、Remote Data Segment に対する通信の高速化を狙っていた。

しかし、Local Data Segment に対する通信においては逆効果である。データを Local Data Segment に対して put するたびに Value 型に変換するコストがかかる。データを peek する際にも Value 型から元の型に変換するというコストがかかる。

この問題を解決するために、一般的な Java のクラスオブジェクトでもデータ表現を可能にした。Local Data Segment に対して put する場合は、Value オブジェクトに変換せず一般的な Java のクラスオブジェクトのまま、Remote Data Segment に対して put する場合にのみ Value に変換する。これにより、無駄な変換コストを抑えられるようになった。

5.2 分散環境における改善

AliceVNC を実装するにあたり、Alice の送受信部分に問題が発見された。ここでは発見された問題とその解決方法を示す。

5.2.1 Data Segment のデータ表現の変更

AliceVNC は、3.4 で説明したように、当研究室で開発している TreeVNC を分散フレームワーク Alice を用いて実装した画面共有システムである。

Topology Node は受け取った画面データを描画すると同時に、Remote Data Segment に送信する。Remote Data Segment に送信する際には Message Pack により Value 型に変換し、その後シリアル化 (byteArray で表現されたバイナリに変換) される。Topology Node は受信するとデシリアル化し Value 型に変換した後 put される。

この Value 型への変換が問題である。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、デシリアル化し Value 型に変換する必要はない。シリアル化状態のまま子ノードに送信すれば、Value 型に変換するオーバーヘッドと Value 型をシリアル化するオーバーヘッドを無くすることができる。そこで、Remote から put されたデータ表現を Value 型から byteArray で表現された binary に変更した。また、Remote に put する際にも Value 型に変換せずに直接 byteArray に変換するように変換した。

しかし、この変更で新しい問題が発生した。Remote から put されたデータは必ず byteArray で表現される。しかし、put された byteArray が全てシリアル化された状態であ

るとはいえない。一般的な Java のクラスオブジェクトとして `byteArray` が使用されている場合が存在する。例えば、AliceVNC で使われる画像データは `byteArray` で表現されているが、これは Local から put されている。Input Data Segment が格納される Receiver クラスには `asClass()` というメソッドがある。

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

このメソッドは取得したデータが Remote から put された場合、Value 型でなっているため Message Pack を使い適切な型に変換するものである。しかし、`byteArray` 型に変更したため、変換可否を判断することができなくなってしまった。

ここからわかることは、データを表現するにはデータ単体をやりとりするだけでは不十分ということである。変更以前は Value 型であるということが状態を表していた。しかし、一般的な Java のクラスオブジェクトと `byteArray` で表現された binary が混在する現在では、データと一緒にデータの状態を表すメタデータもやりとりする必要がある。そこで、データとデータの状態を 1 つのオブジェクトにまとめ扱うように変更した。(ソースコード 5.4)

ソースコード 5.4: データを表現するクラス

```

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

```

`val` がデータ本体が保存格納される。`serialized` と `byteArray` がデータの状態を表すメタデータである。`serialized` は、データ本体がシリアライズ化されているかを示す。`byteArray` がデータ本体が `byteArray` であるかを示す。この 2 つの状態があることで `asClass()` を使い、適切に変換することができる。(ソースコード 5.5)

ソースコード 5.5: `asClass` の処理

```

1 public <T> T asClass(Class<T> clazz) {
2     if (!byteArray) {
3         return (T) val;
4     }
5     byte[] b = (byte[]) val;
6
7     if (serialized) {
8         return SingletonMessage.getInstance().read(b, clazz);
9     } else {
10        return (T) b;
11    }
12 }
13 }

```

asClass が行う処理は、Local から put されたデータ (serialized と byteArray が false の場合または byteArray のみ true の場合) は、目的の Class に cast する。Remote から put されたデータ (serialized が true の場合) は Message Pack を使い変換する。

Message Pack の機能追加

通信入力部は Message Pack の Unpacker を用いる事により、ストリームを次から次へとデシリアライズすることができる。しかし、提供されている API は全てデシリアライズを行うものであり、シリアライズ状態のオブジェクトを取得することができない。そこで Unpacker にシリアライズ状態のオブジェクトを取得するメソッドを追加した。

ソースコード 5.6: ByteBuffer 作成部分

```

1 while (true) {
2     Command cmd = null;
3     ReceiveData rData = null;
4     CommandMessage msg = unpacker.read(CommandMessage.class);
5     CommandType type = CommandType.getCommandTypeFromId(msg.type);
6     switch (type) {
7         case UPDATE:
8         case PUT:
9             int dataSize = unpacker.readInt();
10            rData = new ReceiveData(unpacker.getSerializedByteArray(dataSize),
11                                   msg.compressed, msg.serialized);
12            cmd = new Command(type, null, null, rData, 0, 0, null, null,
13                               reverseKey);
14            lmanager.getDataSegmentKey(msg.key).runCommand(cmd);
15            break;

```

ソースコード 5.6 は受け取ったデータを Local Data Segment に追加する処理である。getSerializedByteArray メソッドでシリアライズ状態のオブジェクトを取得している。

このメソッドの実装をもって、受け取ったデータをデシリアライズせずに、子ノードに渡すことが可能となった。

5.2.2 パケットの再設計

Alice の通信の際には、CommandMessage.class のインスタンスを Message Pack によりシリアライズ化したものが送信される。つまり、CommandMessage.class がパケットの構造を表すものといえる。

ソースコード 5.7 が変更前の CommandMessage の内容である。表 5.1 は CommandMessage の各変数が何を表しているかを示したものである。

このパケット構造に問題が存在する。データ本体は CommandMessage がシリアライズ化されるときにはすでに、シリアライズされている。つまり、このまま CommandMessage をシリアライズ化を行うと、データ本体をもう 1 度シリアライズ化を行ってしまう。配列

ソースコード 5.7: 変更前の CommandMessage

```

1 @Message
2 public class CommandMessage {
3     public int type;
4     public int seq;
5     public String key;
6     public byte[] val;
7     public boolean quickFlag;
8     public boolean serialized;
9 }
    
```

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
val	データ本体
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

をシリアライズ化する場合、配列のサイズによってはオーバーヘッドが大きいいため、2度シリアライズするのを防がなければならない。

ソースコード 5.8: 変更後の CommandMessage

```

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

そこで、CommandMessage をソースコード 5.8 のように変更した。データ本体を CommandMessage のフィールドから外し、後から ByteBuffer にまとめることにより 2 度のシリアライズを防ぐ。(ソースコード 5.9)

この実装では CommandMessage 部をヘッダーとして扱っている。データ部は CommandType が UPDATE、PUT、REPLY の時のみ後から付加される。以前の実装では byte[] の値として null を示す NilValue があるものとしてシリアライズ化されており、これもオーバーヘッドである。現在の実装には、CommandType が UPDATE、PUT、REPLY 以外はの時は、

データ部をシリアライズ化しないため、null をシリアライズ化するオーバーヘッドはなくなっている。

ソースコード 5.9: ByteBuffer 作成部分

```
1 public ByteBuffer convert() {
2     ByteBuffer buf = null;
3     MessagePack msg = SingletonMessage.getInstance();
4     try {
5         byte[] header = null;
6         byte[] data = null;
7         byte[] dataSize = null;
8         boolean serialized = false;
9
10        switch (type) {
11            case UPDATE:
12            case PUT:
13            case REPLY:
14                data = msg.write(rData.getObj());
15                CommandMessage cm = new CommandMessage(type.id, seq, key, false,
16                    serialized);
17
18                header = msg.write(cm);
19                dataSize = msg.write(data.length);
20                buf = ByteBuffer.allocate(header.length+dataSize.length+data.
21                    length);
22                buf.put(header);
23                buf.put(dataSize);
24                buf.put(data);
25                break;
26            default:
27                header = msg.write(new CommandMessage(type.id, seq, key, quickFlag
28                    , false));
29                buf = ByteBuffer.allocate(header.length);
30                buf.put(header);
31                break;
32        }
33
34        buf.flip();
35    } catch (IOException e) {
36        e.printStackTrace();
37    }
38    return buf;
39 }
```

第6章 分散フレームワーク Alice の評価

この章では、Alice を用いた実験方法等についてまとめ、第5章で行った効果の測定、先行研究である Federated Linda との性能比較を行い、評価を行なう。また、TreeVNC と AliceVNC の比較をコードの観点からも評価を行なう。

6.1 並列環境の改善効果の測定

第5.1章の分散環境における改善効果を bitonic sort による実験によって測定を行なう。

実験環境

コア数が少ないマシンでは、同時に走る Code Segment が少ないことから、メニコア環境で実験を行った。

表 6.1: 実行環境の詳細

CPU	Intel Xeon E5-1650 v2 @3.50GHz
物理コア数	6
CPU キャッシュ	12MB
Memory	16GB

実験結果

100万の要素をもつ配列の Sort にかかる時間を計測する。同時に走る Code Segment が物理コア数と同じになるように、分割数は4個で行った。

表 6.2: bitonic sort の結果

	改善前	改善後
実行時間 (ms)	164.8	112.1

6.2 分散環境の改善効果の測定

第 5.2 章 の分散環境における改善効果をリングトポロジーによる実験によって測定を行なう。また、先行研究である Federated Linda との比較も行なう。

6.2.1 TORQUE Resource Manager を用いた実験方法

分散環境の実験する際に、学科にある共用のブレードサーバーを用いた。TORQUE Resource Manager (<http://www.adaptivecomputing.com/products/torque.php>) というジョブスケジューラーによって、他の利用者とのリソースが競合しないように管理されている。

TORQUE Resource Manager

TORQUE は、1 台のマスターと複数台のスレーブで構成される。(図 6.1) スレーブは、マスターへ現在の自身のリソースの利用状況を報告する。

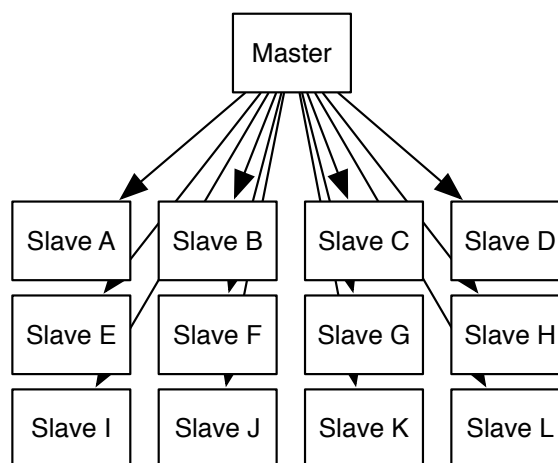


図 6.1: TORQUE の構成

ユーザーはマスターを用いて TORQUE を利用する。ジョブを記述したシェルスクリプトを用意し、スケジューラーに投入する。投入するタイミングで、利用したいマシン台数、CPU コア数を指定することができる。(ソースコード 6.1)

TORQUE は、ジョブに必要なマシンが揃い次第、受け取ったジョブを実行する。

ソースコード 6.1: 10 台 (1 台あたり 4 コア) で走らせる例

```
1 qsub -l nodes=10:ppn=4 job.sh
```


6.2.2 実験概要

リングのトポロジーを構成し、メッセージが 100 周する時間を計り、1 周あたりの平均時間を求める実験である。(図 6.2)

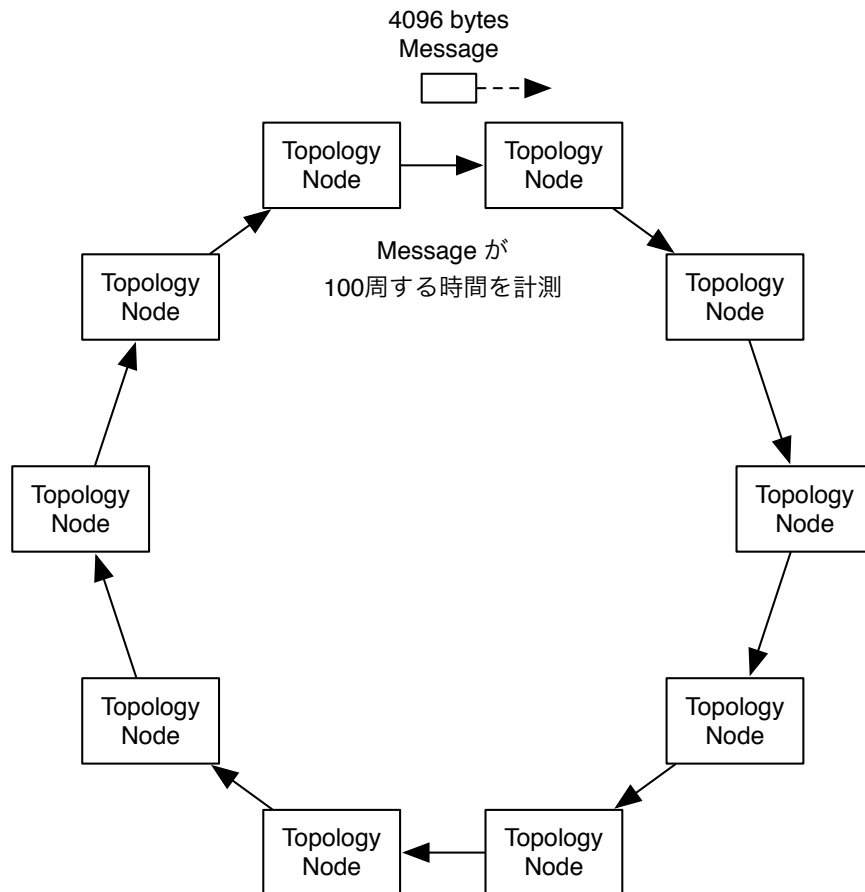


図 6.2: 100 周にかかる時間を計測し、1 周あたりの平均時間を求める

実験では、トポロジーの構築時間は実験に含めてはいない。

実験環境

ブレードサーバー (表 6.3) 上の仮想マシン (表 6.4) による仮想クラスタ環境を用いて実験を行った。

表 6.3: ブレードサーバーの詳細

マシン台数	8 台
CPU	Intel(R) Xeon(R) X5650 @ 2.67GHz
物理コア数	12
論理コア数	24
CPU キャッシュ	12MB
Memory	132GB

表 6.4: 仮想クラスタの詳細

マシン台数	48 台
CPU	Intel(R) Xeon(R) X5650 @ 2.67GHz
物理コア数	2
仮想コア数	4
CPU キャッシュ	12MB
Memory	8GB

実験結果

改善効果と Federated Linda との比較

データのサイズは 10B と 100KB で実験を行った。10B の結果は図 6.3、100KB の結果は図 6.4 である。

10B と 100KB の両方の結果で Alice に行った改善の効果を確認することができる。45 台を使用した実験では 10B の小さいパットの場合では 17%、100KB の大きいパケットの場合では 12% 程度高速化することができた。Federated Linda と改善後の比較では、10B の場合で Alice のほうが 20% 程遅い。しかし、100KB の場合ほとんど差がないことがわかる。

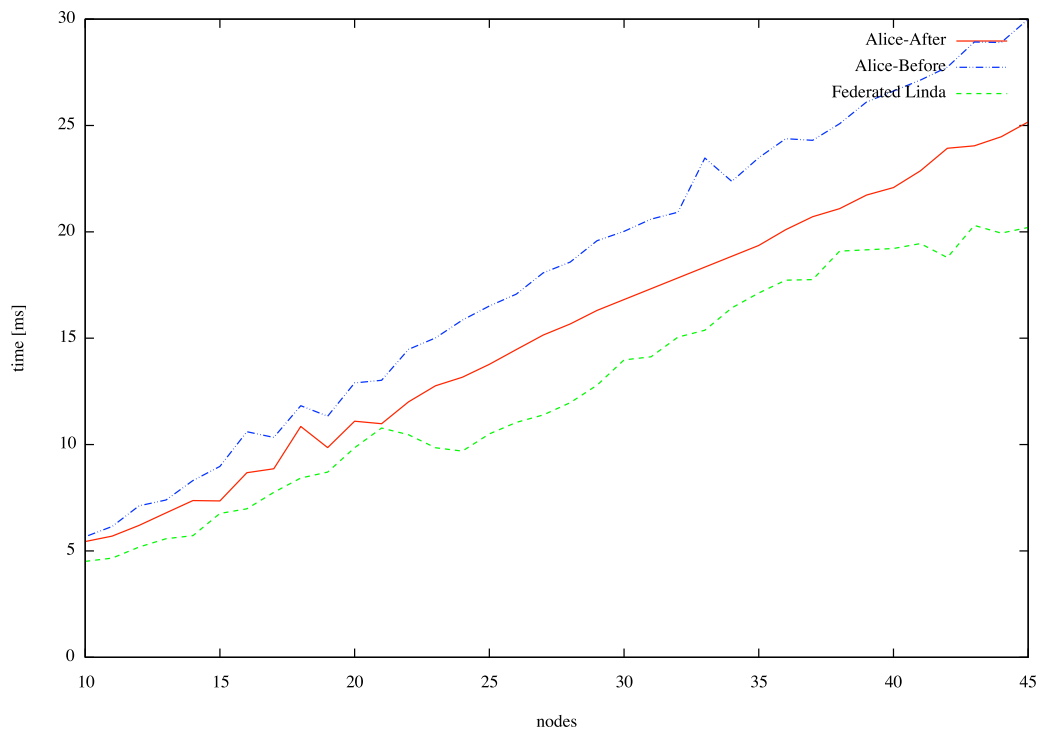


図 6.3: 10 bytes のデータを 100 周させたときの 1 周にかかる平均時間

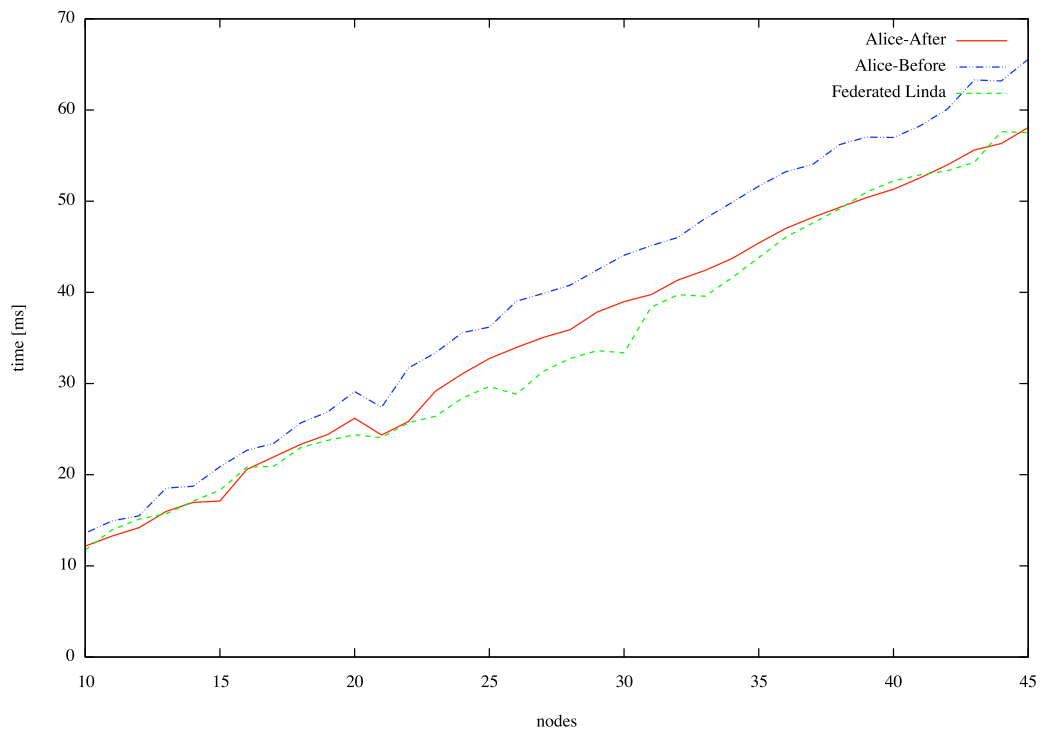


図 6.4: 100 Kbytes のデータを 100 周させたときの 1 周にかかる平均時間

no-tcp-delay 有無の比較

TCP はデフォルトで、Nagle アルゴリズムを使用している。Nagle アルゴリズムは、小さいパケットを集めてまとめて送信することで、送信するパケット数を減らし効率性をあげるアルゴリズムである。このアルゴリズムにより、実験結果に影響があるか調査した。

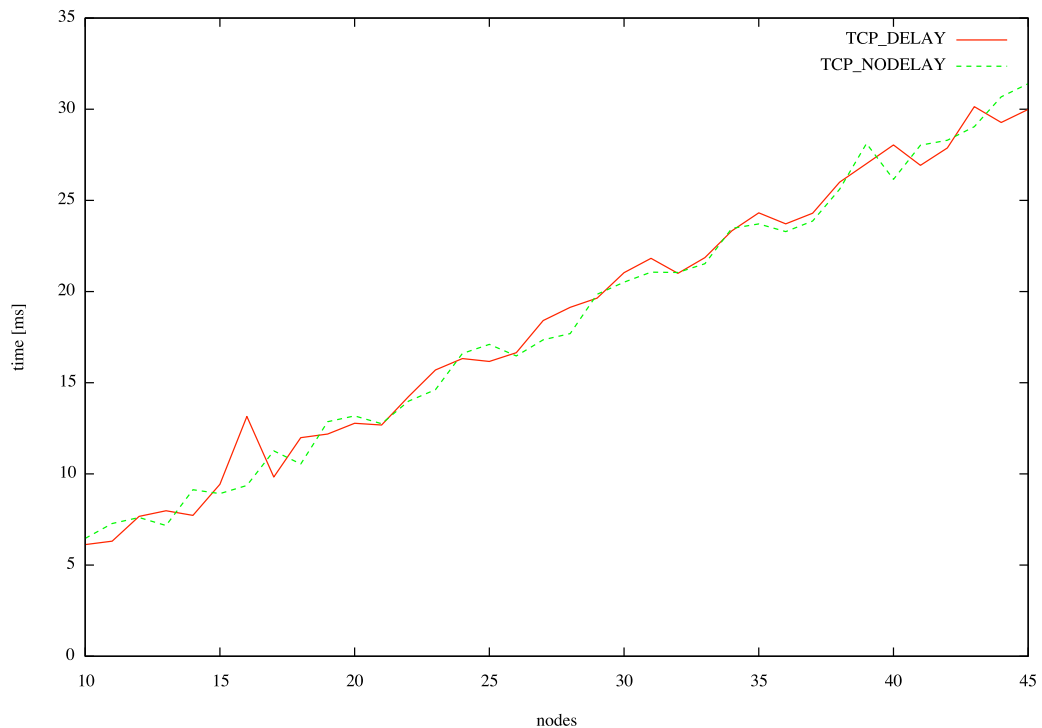


図 6.5: TCP_NODELAY の有無の比較

図 6.5 から TCP_NODELAY における影響はないことがわかる。

6.3 考察

今回の結果から、Alice は先行研究である Federated Linda と同等の性能を持つことが確認できた。また、並列性能の改善と分散性能の改善の両方に効果があることを確認できた。両方に共通して行った改善として、複数の SEDA のステージをまとめて 1 つのステージにしたことがあげられる。SEDA が実行結果に大きく影響を与えていることが分かる。

10B の実験で Federated Linda に及ばない理由としても SEDA が原因と考えられる。リングの実験は並列処理を行なう部分がないシーケンシャルな実験であるため、全ての処理は直列的に実行される。SEDA による Thread の切り替えが発生する分 Alice の実行速度は遅くなる。100KB の実験では Data Segment の送受信にかかる時間に比べ、Thread の切り替えの時間が無視できる程度小さいため、Federated Linda と同じグラフとなる。

Alice が Federated Linda に対して優位な点は、マルチコアによる並列実行である。従って、複数の Code Segment が同時に走る実験では、小さなパケットの場合でも Federated Linda に勝つことができると予想される。

6.4 TreeVNC との Code の比較

TreeVNC と AliceVNC のソースコードに対して wc を行い、TightVNC からどの程度コードが増加しているかを調べた。(表 6.5)

	行数	単語数
TreeVNC	5049	14191
AliceVNC	989	2355

表 6.5: コードの増加量

AliceVNC は TreeVNC の 20% の行数で記述できることがわかる。コード量が少なければ管理する手間が少ないためプログラマー負担を減らすことができる。つまり、Alice を使うことでプログラマーの負担を 20% 減らせる。

第7章 結論

7.1 まとめ

本研究では、まずはじめに 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。次に、今回新たに追加した動的なトポロジーの作成の機能と Alice の Computation を変更する Meta Computation について説明を行った。そして、追加した機能を用いて実際に例題を記述することで、その有効性を示した。改善点では例題を記述する際に発見された問題点について並列と分散両方の観点から説明し、その改善を示した。

最後に、今回行った改善点の効果を測るためリングトポロジー上での実験を行った。また、先行研究である Federated Linda との性能比較を行い、同程度の性能を持つことを確認した。また、TreeVNC と AliceVNC をコードの量という観点から評価を行った。

7.2 今後の課題

7.2.1 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、Jungle 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。

7.2.2 DataSegmentKey の領域分け

現在 Meta Data Segment と Data Segment は、同じ Key Value Queue で管理されている。つまり、Data Segment API を用いれば、誰でも Meta Data Segment を変更することができてしまう。Meta Data Segment に Alice の状態を表すものがあるため、ユーザーによる変更は望ましくない。また、ユーザーが意図せず Meta Data Segment Key に対して、put することも考えられる。そして、Meta Code Segment が active になりエラーを起こす。Meta Code Segment でエラーが起こった場合、ソースコードを見ることができないため解決しづらい。

このようなエラーを防ぐために Meta Data Segment と Data Segment の領域を分ける必要がある。Data Segment を分けることにより Key の重複によるエラーをアプリケーションレベルにすることができる。しかし、領域を分けるだけでは足りない。領域をわけることで解決することができるのは put または update の場合だけである。Meta Data Segment を取得する際に take を使用した場合 Meta Data Segment が Queue から取り除かれてしまう。この問題に対処するためには権限を Code Segment に対して設定する必要がある。その権限により、take と記述しても実際には peek を行うようにすることができる。

7.2.3 記述に関する注意点

setKey のシンタックス問題

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。

Code Segment は内部で実行に必要な Data Segment を数えている。Data Segment 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 Data Segment が全て揃ったことと判断される。全て揃った際には 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。

ソースコード 7.1: NullPointerException になる可能性がある

```

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 7.1 は、for 文で setKey と ids.create を cnt の回数呼び、動的に Data Segment 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、Data Segment 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いるが、2 回目の `asClass` で `NullPointerException` を引き起こす。今回の場合、コンストラクタ内をソースコード 7.2 のように記述する必要がある。

ソースコード 7.2: `NullPointerException` にならない記述

```
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 }
```

この問題は現状のコンストラクタ内で `setKey` を行う方法では `Code Segment` のインスタンスを扱うことができないことを意味する。インスタンス化ができないため `CloseEventManager` に登録する `Code Segment` は `Class` で指定しなければならない。`setKey` をどのタイミングで呼ぶべきか考えなおす必要がある。

singleton Code Segment

Java には、クラスのインスタンスを 1 つに限定する Singleton パターンがある。`Code Segment` に Singleton パターンを使用したい場合があり得る。その場合、`setKey` を行なう前に `ids.init()` を行なう必要がある。先ほどの `setKey` のシンタックス問題でも述べたが、`Code Segment` は内部で実行に必要な `Data Segment` を示す値がある。一度実行された状態の `Code Segment` と新しくインスタンスを作成した場合の `Code Segment` では、値が異なるため `ids.init()` を呼ばずに `setKey` を行なうと `Data Segment` が揃ったにもかかわらず `Code Segment` が実行されない。そのため `setKey` を呼ぶ前に `ids.init()` を呼び内部の値を初期化する必要がある。(ソースコード 7.3)

ソースコード 7.3: setKey を呼ぶ前に init を呼ぶ必要がある。

```
1 public class TaskExecutor extends CodeSegment {
2     private Receiver info = ids.create(CommandType.TAKE);
3     private static TaskExecutor instance = new TaskExecutor();
4
5     private TaskExecutor() {}
6     public static TaskExecutor getInstance() {
7         return instance;
8     }
9
10    public void setKey() {
11        ids.init();
12        info.setKey("_SCHEDULER");
13    }
14 }
```

謝辞

参考文献

- [1] 河野真治, 仲宗根雅臣. 同期型ダブル通信を用いたマルチユーザ playstation ゲームシステム, March 1998.
- [2] 安村恭一, 河野真治. 大域 id を持たない連邦型ダブルスペース federated linda. 第 99 回 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・システム研究発表会, May 2005.
- [3] 安村恭一, 河野真治. 動的ルーティングによりダブル配信を行なう分散ダブルスペース federated linda. 日本ソフトウェア科学会第 22 回大会, Sep 2005.
- [4] 安村恭一, 河野真治. 分散プログラミングモデル federatedlinda. Master's thesis, 琉球大学理工学研究科情報工学専攻, Feb 2006.
- [5] 淵田良彦, 河野真治. 連邦型ダブルスペースを使ったコンパクトルーティングの実験. 情報処理学会プログラミング研究会, Feb 2007.
- [6] 赤嶺悠太, 小野雅俊, 河野真治. 連邦型 linda による分散アルゴリズムをデバッグするためのメタプロトコル. 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, Apr 2009.
- [7] 赤嶺一樹, 河野真治. Meta engine を用いた federated linda の実験. 日本ソフトウェア科学会第 27 回大会, Sep 2010.
- [8] 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, Sep 2011.
- [9] 上里献一, 河野真治. Suci ライブラリのスナップショット api を利用した並列デバッグツールの設計. 日本ソフトウェア科学会第 20 回大会, Sep 2003.
- [10] 金城裕, 河野真治. Fine grain task manager cerium のチューニング. 日本ソフトウェア科学会第 27 回大会, Sep 2010.
- [11] 金城裕, 河野真治. ゲームフレームワーク cerium taskmanager 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, Apr 2011.
- [12] 金城裕, 河野真治. Cerium における datasegment api の設計. 日本ソフトウェア科学会第 28 回大会, Sep 2011.

- [13] Matt Welsh, David Culler, and Eric Brewer. Seda: An architecture for well-conditioned, scalable internet services. *Computer Science Division*, Mar 2003.
- [14] Avinash Lakshman and Prashant Malik. Cassandra - a decentralized structured storage system. *LADIS*, Mar 2003.

発表履歴

- Meta Engine を用いた Federated Linda の実験,
赤嶺 一樹, 河野 真治 (琉球大学),
日本ソフトウェア科学会第 27 回大会 (2010 年度) 講演論文集, Sep, 2010
- Data Segment API を用いた分散フレームワークの設計,
赤嶺 一樹, 河野 真治 (琉球大学),
日本ソフトウェア科学会第 28 回大会 (2011 年度) 講演論文集, Sep, 2011