分散ネットワークフレームワーク Alice の提案と実装

Design and Implementation Distributed network framework Alice

平成23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

赤嶺 一樹

要旨

ブロードバンド環境やモバイル端末の普及により、ネットワーク上におけるサービスに参加するユーザー数の巨大化は必至である。しかし、フレームワークやデバッグ等を含めてスケーラビリティーに優れた分散プログラムを記述する事は極めて難しい。ここでいうスケーラビリティーとは、サービスの大きさが増えても、リソースの追加のみでサービスの質を維持できることを示す性能基準のことである。そのスケーラビリティーを備えた分散プログラムを作成するためには、いろいろなプロトコルを提案し、実装し、検証していく必要がある。

そこで本研究室では、シンプルにスケーラブルな分散プログラムを書くことができるフレームワークとして、”分散ネットワークアプリケーションフレームワーク Alice” を提案し、実装した。

Alice には本研究の先行研究である Federated Linda の開発を通して得られた知見が大いに生かされている。また、それと同時に本研究室で開発している並列タスク管理フレームワーク Cerium の新設計から得られたアイディアも生きている。

Alice は Data Segment と Code Segment という単位でデータと処理を細かく分割し、それぞれの依存関係をシンプルに記述して分散プログラムを作成することができる。また、他のマシンとの接続トポロジーの構築等の機能も盛り込まれており、ユーザーはトポロジー構築後の処理を記述することに集中できる。

最後に、先行研究である Federated Linda との性能比較を行う。

The spread of the environments of broadband internet and mobile data terminal would be cause the huge increase in active users of internet services. But, it is so difficult to write a distributed program having scalability. The scalability is the performance standard. It can keep quality by only additional resource. We have to propose more protocol, implement it and verify it to develop the distributed program having scalability.

Then we design and implement a distributed network application framework Alice. You can simply write a scalable distributed program with using it.

Alice made by an experiment of developping Federated Linda, and an idea proposing new design of concurrent task manager Cerium we developped.

You can simply develop a distributed program with splitting data as Data Segment and processing as Code Segment, and you set dependency each other. And Alice has a feature building topology with distributed computers. You can concentrate writing source code for distributed program.

At last Alice are compared with Federated Linda.

目 次

図 目 次

表 目 次

第1章　序論

1.1　研究背景と目的

ブロード環境やモバイル端末の普及により、ネットワーク上におけるサービスに参加するユーザー数の巨大化は必至である。しかし、フレームワークやデバッグ等を含めてスケーラビリティーに優れた分散プログラムを記述する事は極めて難しい。ここでいうスケーラビリティーとは、サービスの大きさが増えても、リソースの追加のみでサービスの質を直線的に維持できることを示す性能基準のことである。そのスケーラビリティーを備えた分散プログラムを作成するためには、いろいろなプロトコルを提案し、実装し、検証していく必要がある。

分散プログラムの開発は単純に離れたホスト間で通信して動作するだけならば、プログラム自体は難しくない。ただ、逐次プログラムに通信処理を導入すればよいだけである。分散プログラム開発において困難とされるのは、ネットワークの動的な変化、故障、性能の多様性を吸収し、その上でスケーラビリティーを確保することである。

スケーラビリティ等を確保したまま、分散プログラムを正しく記述することやデバッグを行うために、これまで数多くの分散プログラミングモデルが開発されてきたが、それらは逐次プログラムのプログラミングモデルの延長であったり、アプリケーション開発者が複雑で理解しにくい記述を熟知しなければならなかった。それらは真の分散プログラミングモデルとは言えない。

そこで本研究室では、シンプルにスケーラブルな分散プログラムを書くことができるフレームワークとして、”分散ネットワークアプリケーションフレームワーク Alice” を提案し、実装した。本研究の目標は、スケーラブルな分散プログラムを自然に記述できる道標をフレームワークという形で分散プログラマーに提供することである。

1.2　論文の構成

第 2 章では、先行研究である Federated Linda の開発を通して得られた知見について説明する。分散プログラムに重要な事は何であるのかについてまとめ、新しいフレームワークの設計に活かす。

第 3 章では、前章で得られた知見と、同じく本研究室で開発している並列タスク管理フレームワーク Cerium の新設計から得られたアイディアを生かした、新しいフレームワークの設計の概観を示す。

第 4 章では、前章で示した概観を、 Java の便利なライブラリの紹介と、具体的に Java でどのように実装したかについてまとめている。

第 5 章では、新設計のフレームワークである Alice の実験方法を交えながら、 Federated Linda の性能比較、考察を行う。

第 6 章においては、まとめと今後の課題となっている。

第2章　Federated Linda の実装により得られた知見

2.1　Federated Linda

本研究室では、自然に分散プログラミングが書けるようなプログラミングモデルとして、グローバルな ID を持たない連邦型タプルスペース (以下 Federated Linda [1, 2, 3, 4, 5, 6, 7, 8] と記す) を提案してきた。

2.1.1　Linda

Linda は、タプルという ID で番号づけられたデータの集合を、以下の API (表 2.1) を用いて共有されたタプルスペースに出し入れすることにより、外部との通信を行うプログラミングモデルである。(図 2.1)

表 2.1: Linda API

API	**概要**
in(id)	タプルスペースからタプルを取り出す。 タプルスペースにタプルは残らない。
read(id)	タプルスペースからタプルを取り出す。 タプルスペースにタプルが残る。
out(id,data)	タプルスペースへタプルを書きこむ。

2.1.2　Federated Linda

Federated Linda は、複数のタプルスペース (Linda Server) を相互に接続することにより分散プログラミングを実現する。一つのタプルスペースには少数の接続があることが期待されており、多数のタプルスペースが接続することにより分散アプリケーションを実現する (図 2.2)。 smtp/nntp デーモンが行単位でプロトコルを作るのと似た形で、タプルスペースへの in/out でプロトコルを作ることになる。

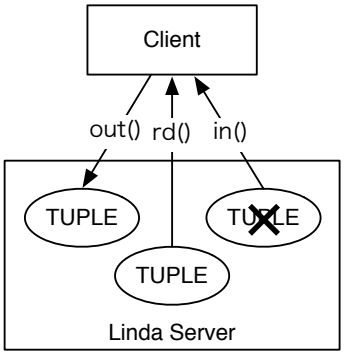

図 2.1: Linda Server とそれを API を用いて操作するクライアント

通信モデルは、タプルの出し入れによるリレー転送のようになる。インターネットのパケット転送のように、タプルスペースからタプルスペースへとタプルを転送していく。

クライアントのアクセス数が増えても、タプルスペース等の数を増やし、ネットワーク処理を分散することにより、スケーラビリティを保つ。

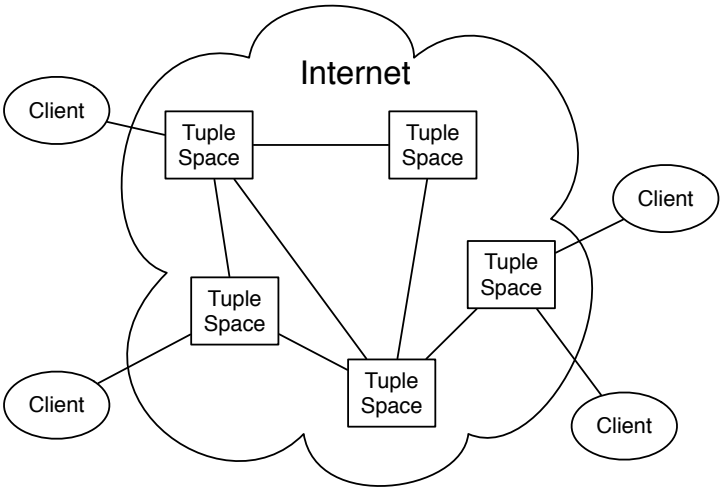

図 2.2: タプルスペースの相互接続

2.2 Federated Linda の分散プログラミング

Federated Linda の分散プログラムには ”Tuple Space”, ”Protocol Engine”, ”Link Configuration” の 3 つの要素がある。Federated Linda は、この 3 つの要素に基づいてプログラミングモデルを提供する。

2.2.1 Tuple Space

プロトコルへのアクセスは Linda の API を用いる。つまり、タプルスペースへの ”in”, ”read”, ”out” 等である。これらのコマンドは単純で、理解しやすいものである。タプルの出し入れというモデルで通信を行うことができる。

また、 Protocol Engine との通信は非同期に行われる。

2.2.2 Protocol Engine

プロトコルを規定する Protocol Engine は、分散アルゴリズムを内包し、他のプロトコルへのアクセスもタプルスペース経由で行う。通信はタプルをタプルスペースからタプルスペースへと伝搬させるように転送する。クライアントはタプルスペースを介した通信を行うので、クライアントからはプロトコルの細かい挙動は見えない。しかし、クライアントプログラムのプロトコルへの依存を低く抑えることが可能である。

タプルスペースとのタプルのやり取りは非同期で行われる u が、これらのライブラリはシングルスレッドで実装されている。そのため、”in”, ”read” を実行した際にリクエストは送信されず、また返ってくるであろうタプルのレスポンスは返ってこない。”sync” を行った時にキューに入っているリクエストを Linda Server へ送信する。この時、サーバー側からクライアントへのレスポンスが準備できていればそれらを受け取る。

サーバー側からレスポンスデータを受信した際、ユーザーはそれらを確認する以下の 2 つの方法がある。

poll 方式

poll 方式とは、 ”sync” を行った後に、その ”in”, ”out” のレスポンスが ready 状態になっているか確認する方法である。この方式を用いた場合、レスポンスが ready 状態かどうかをユーザーが好きなタイミングで記述するため、行いたい処理を順次書くだけでよくなるため、ソースコードの可読性が上がる。

しかし、レスポンスが返って来ない間、ループを行い、状態チェックを行う等の処理を書かなくてはいけなくなるため、処理効率は悪化する。(図 2.3)

callback function 方式

callback function 方式とは、 ”in”, ”out” を行う際に、そのレスポンスがサーバー側から返ってきた時の処理を予め記述しておく方法である。

レスポンスがサーバー側から返って来た時に指定したコールバック関数が自動で実行されるため、ユーザーは受信したかどうかチェックする処理を書かなくてもよい。また、それらの処理がまるごと消えるため、処理効率は向上する。(図 2.4)

しかし、実行されるタイミングをユーザーが把握することが難しくなる。そのため、ソースコードの難読化につながる。コールバック関数間のデータの共有も難しくなる。

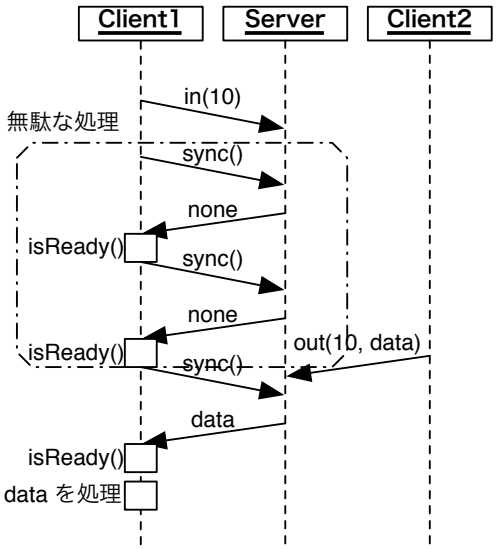

図 2.3: poll 方式の例

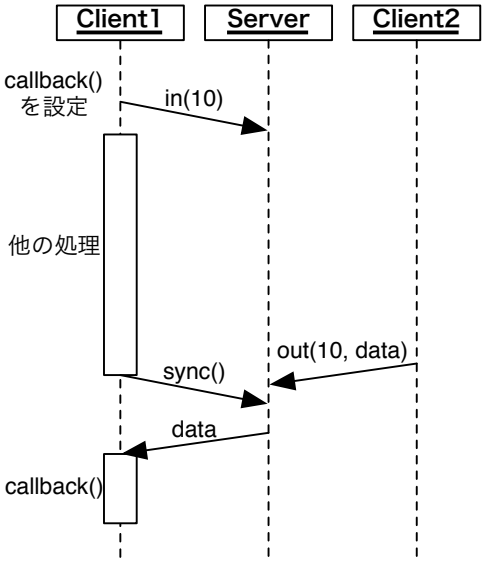

図 2.4: callback function 方式の例

2.2.3 Link Configuration

タプルスペースや Protocol Engine の接続を規定する。接続の状態を XML として表し、各ノードがそれにしたがって IP の上位レイヤーでオーバーレイネットワークを構築する。オーバーレイネットワークを構築するために、接続等を扱うモジュールを提供する。

2.3 Federated Linda の改良

Federated Linda はいくつかの段階を経て実装されてきた。それは、プロトタイプをプログラミングすることによって、設計を詳細なものへと落としこんでいく必要があったからである。

2.3.1 Meta Protocol Engine

従来の Protocol Engine は、タプルスペース (Linda Server) とは独立したプロセスであった。(図 2.5) それでは、その 2 つのプロセスがローカルの同じマシン上にあっても、タプルをやり取りするためにソケット通信を行う必要があった。

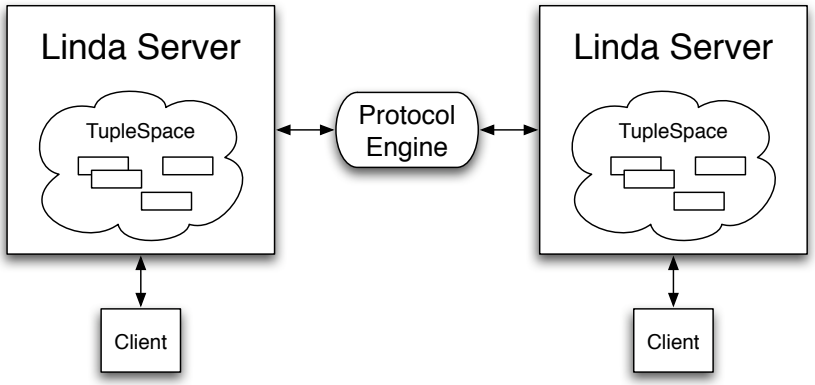

図 2.5: タプルスペースと Protocol Engine が別プロセス

そこで、 Linda Server と一体型の Protocol Engine として、 Meta Protocol Engine を提案し、実装した。(図 2.6) Protocol Engine のプロセスがタプルスペースと同じであるため、ローカルのタプルスペースへアクセスする際にメモリ空間に直接アクセスすることができるようになった。

2.3.2 update API の追加

Federated Linda の例題を書いているときに、 ”in” の直後に ”out” を実行することが多いことに気がついた。これは即ち、タプルを消して、新しいタプルを書き込むということである。そこで、新しく ”update” API を追加することにした。(表 2.2)

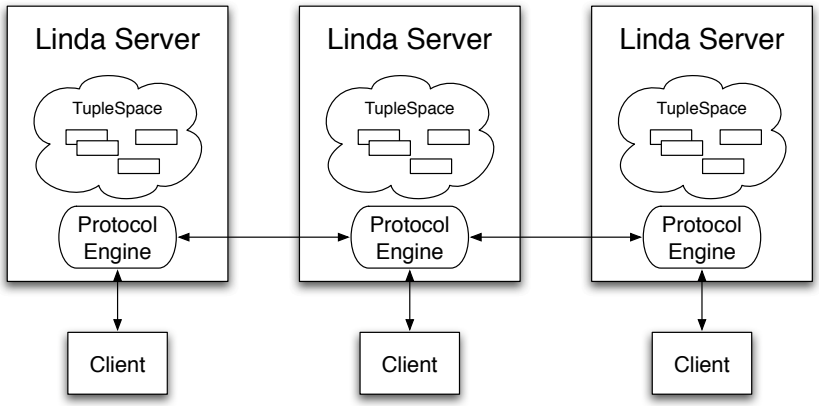

図 2.6: タプルスペースと Protocol Engine が同じプロセス

表 2.2: 追加 Linda API

API	概要
update(id,data)	タプルスペースからタプルを取り除く。 タプルスペースへデータを書き込む。

2.4　Federated Linda の問題点

Federated Linda の設計と実装、それらを使った例題の作成を通して、 Federated Linda の抱える問題点が浮き彫りとなった。これらの問題点を整理し、次章に記す新しい分散アプリケーションフレームワークの設計に活かすことができた。その問題点を以下に記す。

2.4.1　シングルスレッドを用いた設計

Federated Linda の設計上の問題点としてシングルスレッドで設計されていることが挙げられる。シングルスレッドで設計された理由として、 Federated Linda の実装を行った当時は、シングルコア CPU が主流であったことが挙げられる。さらに、Java 2 SE, v1.4 がリリースされ、 java.nio がノンブロッキング I/O をサポートたことも理由として挙げられる。java.nio.channels.Selector を使用し、シングルスレッドで複数の入出力を取り扱うという方法が当時のサーバーの主流であった。

この方法では、大きな容量のデータを送受信した際に、 CPU がそのデータの送受信にかかりっきりになってしまい、その処理が終わるまで他の小さな処理が全て待ち状態になってしまうといった問題が発生していた。

しかし、近年では、シングルコア CPU の性能向上が発熱量等の問題により頭打ちとなり、マルチコアのマシンが主流となっている。将来的にはメニーコアのマシンが主流になっていくと考えられる。そのような背景を踏まえて、 Apache の Cassandra プロジェ

クト [9] ではデーモンの実装に SEDA アーキテクチャ [10] を採用している。これは、マルチ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る。これを用いることによってマルチコア性能を生かし、パイプライン的に効率よく処理することが可能となっている。

分散アプリケーションを作成する上で、マルチスレッドをどのように活用するかといったことが、重要な課題となっている。

2.4.2　タプルの表現方法

次の問題点として、タプルの表現方法が挙げられる。Federated Linda におけるタプルはバイトの配列として表現されている。例えば、 Java を用いた実装では、 ByteBuffer を利用している。バイトの配列を用いることによる問題点としては、データを送受信する際に、ユーザーが Protocol Engine で利用しているデータをバイトの配列へ変換、またはその逆を行う必要があるということが挙げられる。

また、異なるアーキテクチャ間におけるデータフォーマットの取り扱いも問題となる。異なるアーキテクチャとは、CPU の違い (バイトオーダー、ビット数等)、使用するプログラミング言語の違い、 OS の違い等である。例えば、 PlayStation 3 (Cell) 上の Linux で動作するアプリケーションと、 Mac OS X (Intel) 上で動作するアプリケーション間である。そのアプリケーション間で、とある共通の構造体のデータをやり取りするといった事案でデータフォーマットに関する問題が発生する。通信するたびに、 XDR (External Data Representation) などのデータフォーマットを用いて、ユーザーがデータを変換する必要がある。

2.4.3　タプルの ID

更なる問題点として、タプルの ID が整数値であることが挙げられる。これは、整数値であるため、送受信する際や、タプルスペースのデータを探すときには効率がよいというメリットがある。一方で、ユーザーが整数 ID を管理するためには、 enum やハッシュ、即ち文字列から整数値への射影を管理する必要が出てくるのである。新しいタプルをタプルスペースで使用する際に新しい整数 ID と文字列の対応をユーザーが管理する手間がかかる。この問題はデバッグ時に顕著となる。それら無数の整数値をユーザーがログを見ながらどのデータか把握しなければならないからである。

Cassandra 等の KVS (Key-Value Store) では、キーに直接文字列を使用しているため、キーの管理が行いやすくなっている。

2.4.4　Protocol Engine の記述方法

次に Protocol Engine の記述方法である。

Protocol Engine は処理効率の観点から基本的に callback function 方式 (図 2.4) を用いて行うが、これらの取り扱いが問題となる。

”in” や ”read” を行う時にコールバック関数を準備し、タプルに設定する。更に、そのコールバック関数内で、新しいコールバック関数を準備し、 ”in” や ”read” を行ってタプルに設定する。このように、コールバック関数が ”in” や ”read” を介してツリー状に接続されているという構図が生まれる。(図 2.7) このように、ユーザーはタプルの ID を元にコールバック関数の繋がりを把握する必要がある。分散アプリケーションであるため、ローカルマシンのタプル ID だけでなく、リモートマシン上のコールバック関数の影響も受ける。そのため、スタック上に関数を呼び出して読んでいくことが難しい。

これらのタプルの接続をユーザーが ”in” や ”read” を直接用いて表現するのではなく、フレームワーク側で接続関係をサポートする必要がある。

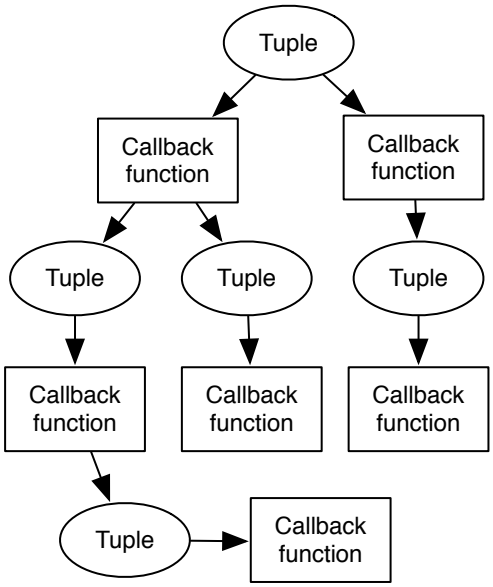

図 2.7: コールバック関数間のタプルを介した接続 (ツリー状)

2.4.5　接続しているタプルスペースリソースの管理

Federated Linda にはタプルスペースへのコネクションを管理する機構がないため、ユーザーがそれらのコネクションを管理する必要があった。分散アプリケーションにおいて、どのようなグラフ構造でマシンが接続されているかという情報は重要である。接続時に、コネクションに把握しやすい名前をつけることで、ユーザーは接続を管理しやすくなる。例えば、リングトポロジーの場合、1 つのマシンに着目してみると、右への接続を”right”、左への接続を”left”といったキーで呼び出すことができると嬉しい。

また、受け取ったデータがどこから来たか、受け取るときに把握できると望ましい。なぜならば、ルーティングを行うときに役立つからである。ツリートポロジーの場合、親から来たデータを子に伝搬するといった処理を記述することがよくある。その時に、どこからデータが来たかという情報は重要である。

これらの問題点を踏まえて設計した、新しい分散アプリケーションフレームワークの設計を次章では示す。

第3章　分散フレームワーク Alice の設計

前章までは、先行研究である Federated Linda の設計と実装について示した。しかし、その設計が古いものであり、時代にそぐわなくなってきていたため、新しく分散向けのフレームワークを開発することにした。

また、本研究室で開発している並列フレームワークの開発から得られた知見も参考にし、それらを本章で示す新設計へと活かすことにした。

本章では、新設計である、分散ネットワークアプリケーションフレームワーク Alice (以下、 Alice) の設計を示す。

3.1　分散プログラミングにおいて重要なこと

まず、 Federated Linda の設計と実装を通して得られた、分散プログラミングにおいて重要な三点を元に基本設計を行った。(図 3.1) その三点とは以下の通りである。

- Database
- Protocol Engine
- Link Configuration

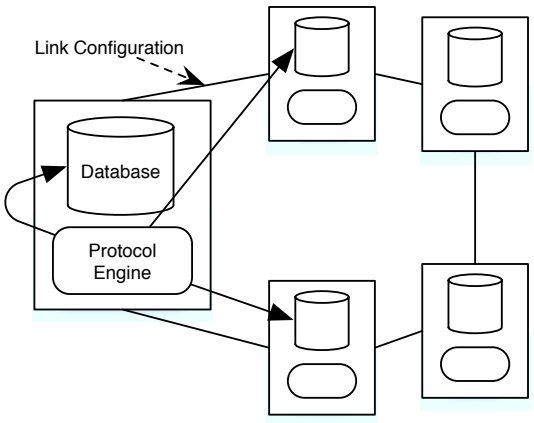

図 3.1: 分散プログラミングにおいて重要な三点

分散プログラミングを行うにあたり、どのようなデータベースでデータを表現、管理するかということが重要になる。 Federated Linda では、 Linda のタプルスペースを用いた。データベースの設計は、ユーザーが意図したとおりにデータを扱えるために、重要なものとなる。

次に、どのようにマシンを接続するか、そのデータベースをどのように操作するかといった処理を記述できる Protocol Engine が挙げられる。この部分はユーザーが直接コードを記述できる。ユーザーが望んだマシンへの接続や必要なデータへ簡単にアクセスでき、簡潔に処理を記述できることが求められる。

そして、どのようにマシンを接続するか、すなわち接続トポロジーをどうするかが重要である。多様なアプリケーション間の通信を分散させるために自由度の高いトポロジーを作成できる必要がある。

3.2 Data Segment と Code Segment

本研究室では、並列タスク管理フレームワーク Cerium[11, 12, 13] (以下、 Cerium) の開発を行なっているが、その設計と実装を行うにあたり、並列プログラミングと分散プログラミングは本質的には同じことを行なっていることが分かった。特に、ヘテロジニアスマルチコアであり、ローカルストアをそれぞれのコアが持っているという Cell の特異な環境は、分散プログラミングの環境と酷似している。それらを踏まえて、 Cerium のタスク管理手法を分散に応用できないかと考えた。

Cerium においても、タスクとそれの入出力データの取り扱いは難しい課題である。データを操作する API を考えて、タスクを自然に記述し、効率よく実行する方法を考えてきた。

そこで考えられた手法が、データを Data Segment、タスクを Code Segment という単位に分割して記述するという手法である。

まず、 Code Segment (タスク) は入力データと出力データ (それぞれ、 Input Data Segment、 Output Data Segment とする) を持つ。 (図 3.2) Input Data Segment はそのタスクを実行するにあたり必要なデータ群であり、 Output Data Segment はそのタスクが計算を行った結果を格納するデータ群である。

Code Segment と依存するデータ群を記述することにより、明示的にそれらが実行される順番が決定される。(図 3.3)

3.3 Data Segment (Database)

Alice では、Data Segment をデータベースとして利用する。

3.3.1 Data Segment API

まず、分散プログラム向けに Data Segment を操作する Data Segment API を定義した。(表 3.1) この Data Segment API を用いてデータの送受信を行う。

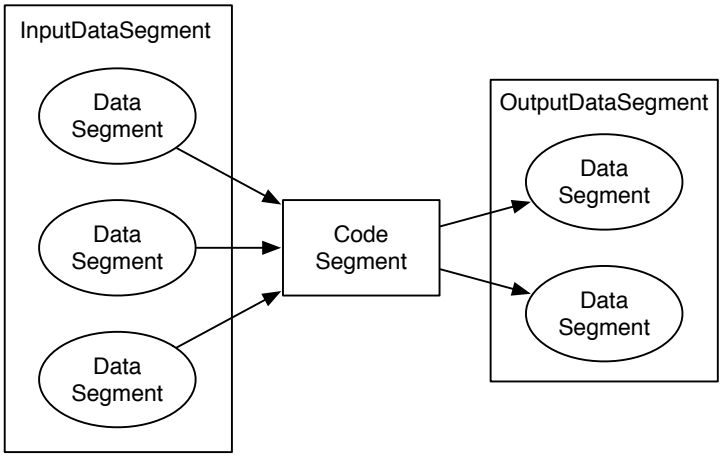

図 3.2: Code Segment と入出力 Data Segment

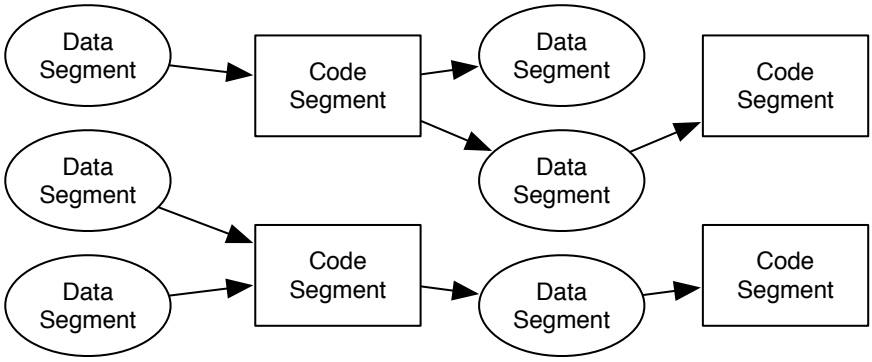

図 3.3: Data Segment と Code Segment の依存関係で一意に決まる実行順序

表 3.1: Data Segment API

API	**概要**
take(key,index)	key に対応するデータセグメントリストからデータを取り出す。 データセグメントリストにデータは残らない。
peek(key,index)	key に対応するデータセグメントリストからデータを読みだす。 データセグメントリストにデータセグメントは残る。
put(key,data)	key に対応するデータセグメントリストに data を追加する。
update(key,data)	key に対応するデータセグメントリストから一つデータを取り除く。 data を追加する。

Data Segment はキーごとにリストを持っている。 Data Segment API を用いれば、これらのリストにデータを追加、削除を適宜行うことができる。

なお、 ”take” と ”peek” には index という引数が存在する。これは、一つの Data Segment に対し、データが出力される度にインクリメントされる値で、前回の ”peek” や ”take” で読み込んだ値からデータの更新があれば読み込むといったデータの入力待ちを行うことができる。なお、 index は省略できる。 index を指定しない場合は、 0 が指定されたものとして扱う。

3.3.2　MessagePack を利用したデータ表現

また、データの表現には MessagePack (`http://msgpack.org/`) を利用する。MessagePack とは、バイナリをベースにした効率重視のシリアライズライブラリーである。MessagePack は、シリアライズされたデータにオブジェクトの型情報も埋め込むため、自己記述的なデータ形式となっている。そのため、 IDL (Interface Definition Language) によるデータ構造の定義は必要なくなる。データの表現方法は JSON と似ており ProtocolBuffers (`http://code.google.com/p/protobuf/`) や Thrift (`http://thrift.apache.org/`) 等の他のシリアライズライブラリーと比較しても高速で利便性がある。

MessagePack で取り扱える型は大きく次の通りである。

- Boolean
- Integer
- Long
- Float
- Double
- Raw (String)
- Array
- Map

Array や Map を用いることで、基本的なデータ型を組み合わせて複雑なデータ構造を作ることが可能である。これにより、データの末尾に付加情報を付けたい場合などにも対応できる。

IDL のように、自作したクラスを構造体と見立てて、シリアライズ、デシリアライズすることも可能である。

最近では、多言語対応が進み、利用できるプログラミング言語の幅が広がっている。そのため、異なるアーキテクチャ、オペレーティングシステム、プログラミング言語の壁を超えて、ネットワーク上で繋がることが可能になった。

3.4 Code Segment (Protocol Engine)

Alice では、 Protocol Engine の代わりとして、 Code Segment と呼ばれる単位でタスクを作成する。

Code Segment は Input Data Segment と Output Code Segment それぞれのデータリストを持っている。これにより、データの依存関係が自明になり、他の Code Segment と依存関係の全くない Code Segment を並列実行することが可能となる。並列度を上げるためには、 Code Segment の処理内容は細かく分割して記述することが望ましい。Code Segment の並列実行を行えるように、コア数分だけスレッドプールを準備する。

3.5 Topology Manager (Link Configuration)

Alice では、 Topology Manager と呼ばれるマシンがマシン間の接続を管理する。

分散アプリケーションを開発する際に直面する問題として、接続とルーティングに関する記述が複雑になってしまうといったことが挙げられる。Alice では、トポロジーを作成するロジックをユーザーがスクリプト言語などで気軽に組むことができる。そのスクリプトが出力したトポロジー設定ファイルを Topology Manager が読み込む。設定を読み込んだ Topology Manager は、 IP アドレスやポート番号、接続名などの接続情報を、トポロジーへの参加を表明したマシンに送信する。

第4章　分散フレームワーク Alice の実装

前章では、分散ネットワークアプリケーションフレームワーク Alice の設計について示した。本章では、それらの設計を踏まえ、 Java でどのように実装を行うのがよいかということから始め、具体的にどのような実装を行ったかについて、詳細に記すことにする。

4.1　Java

Federated Linda は Java を用いて実装が行われていたが、シングルスレッドを用いた設計であったがために、並列処理が含まれていなかった。今回の実装を行う前に並列処理の観点から、 Java について改めて見直し、フレームワークの実装方法について考えることにした。

4.1.1　Java の選定

Alice の実装に Java を選んだ理由は、充実したライブラリ群の存在である。java.io、java.nio 等の入出力ライブラリを始め、 java.net 等のネットワークライブラリも揃っている。また、並列処理ライブラリである java.util.concurrent が非常に強力である。ここでは java.util.concurrent の非常に便利な部品を挙げる。

java.util.concurrent.BlockingQueue

BlockingQueue は、並列で使用できるキュー構造である。複数のスレッドがデータの追加、取り出しを行なっても、問題なく動作する。また、キュー内にデータがない場合、取り出しを行うときにブロッキングが入る。(ソースコード 4.1) そのため、データが来るまで待つという処理を記述することができる。

ソースコード 4.1: キューにデータがない場合、処理がブロックされる

```
1 // 別スレッドで blockingQueue.put(value); されるまでブロック
2 blockingQueue.take();
```

そういった機能を持つ故に BlockingQueue は異なるスレッド間で通信を行うのにしばしば用いられる。例えば、 SEDA アーキテクチャで作られた Cassandra の実装においても、ステージスレッド間でパイプライン処理をする際、データの受け渡しに使用されている。

java.util.concurrent.ConcurrentMap

ConcurrentMap は並列で使用できるハッシュ構造である。複数のスレッドがデータを追加、読み込み、削除を行なっても問題なく動作する。

キーを指定してデータの存在を確認し、データが無ければ追加するといった場合を考える。(ソースコード 4.2) 他のスレッドが同時に同じキーで異なるデータを書きこみを行おうとした場合、通常は排他処理ができていないため、不整合が発生する。しかし、ConcurrentMap の putIfAbsent メソッドを用いれば、原子的に確認とデータ操作を行うため、その問題を解消することができる。(ソースコード 4.3)

ソースコード 4.2: キーの確認とデータ操作が排他処理されていない

```
1 if (!map.containsKey(key))
2     return map.put(key, value);
3 else
4     return map.get(key);
```

ソースコード 4.3: キーの確認とデータ操作が内部で原子的に実行される

```
1 return map.putIfAbsent(key, value);
```

java.util.concurrent.Executor

Executor は受け取った Runnable タスクを実行するオブジェクトである。(ソースコード 4.4) Executor で様々な種類のスレッドプールを作成することができる。

ソースコード 4.4: スレッドプールに Runnable を投入

```
1 executor.execute(runnable);
```

java.util.concurrent.atomic.AtomicInteger

AtomicInteger は、原子的なデータ更新が可能な int 値である。

例えば、データを取得してその後にインクリメントを行うという処理を考える。(ソースコード 4.5) synchronized ブロックで囲み、排他制御を行わないと、データの不整合を起こす。そこで、 AtomicInteger の getAndIncrement() を用いることで、取得と変更の処理を原子的に行うことが可能になる。(ソースコード 4.6)

ソースコード 4.5: synchronized を用いて排他制御を行う必要がある

```
1 synchronized (i) {
2     num = i++;
3 }
```

ソースコード 4.6: getAndIncrement() を用いると原子的に実行される。

```
1 num = i.getAndIncrement();
```

この機能は、複数のスレッドで共有された ID を重複なく連番発行する処理等で用いられる。

このような並列処理において利便性が高く性能のよいライブラリが揃っているため、Java を実装に用いるプログラミング言語に選定した。

4.2 Data Segment の詳細な設計と実装

まず、 Alice の実装を行うにあたって、データベース機能はもちろんのこと、タスク間通信やネットワーク間通信の要となる Data Segment の設計が重要である。本節では、Data Segment の詳細な設計と実装を記す。

4.2.1 Data Segment Manager

大量の Data Segment を管理するのが Data Segment Manager である。Data Segment Manager は、文字列のキーで Data Segment を整理する。キーごとにデータを出し入れすることになる。各キーごとに、キュー構造を持っている。(図 4.1) それらを Data Segment API を用いて操作する。

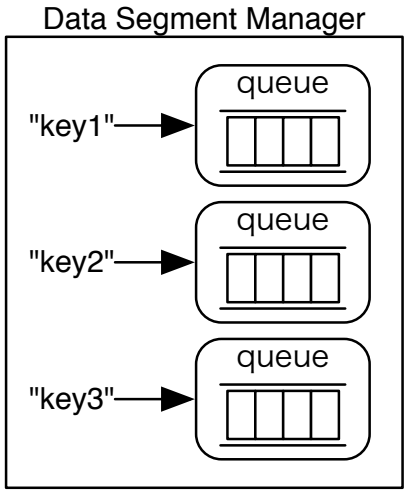

図 4.1: キーごとにデータがキュー構造で管理される

また、データの読み出し (”peek” または ”take”) 時に、希望のデータがなかった場合、ブロッキングを行う機能を持つ。しかし、ブロッキングといってもそこで同期するわけではない。非同期でデータを通信する。そのため、 ”peek” と ”take” は他の API とは違い、レスポンスが発生する。(図 4.2)

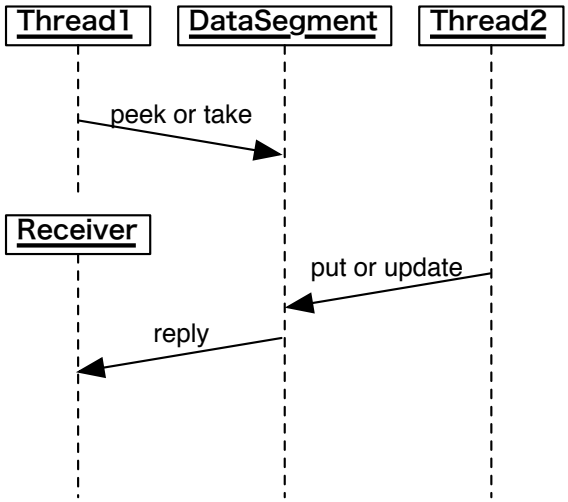

図 4.2: ”peek” や ”take” に対して、レスポンスが発生する

4.2.2 Data Segment API

Data Segment Manager をユーザーが操作できるインタフェース、それが Data Segment API である。まずは以下のとおり Data Segment API を詳細に定義した。

- `void put(String key, Value val)`
- `void update(String key, Value val)`
- `void peek(Receiver receiver, String key, int id)`
- `void take(Receiver receiver, String key, int id)`

”put”

”put” はデータを追加するための API である。

”put” は受け取ったデータ val を Data Segment 内のキューに対してエンキューする。この時、キーごとに重複しない連番の ID を受け取った順に振る。(図 4.3)

”update”

”update” はデータを置き換えるための API である。

”update” はキューの先頭にあるデータをひとつだけ削除する。その後は ”put” と同じく、 受け取ったデータ val を Data Segment 内のキューに対してエンキューする。この時、キーごとに重複しない連番の ID を受け取った順に振る。(図 4.4)

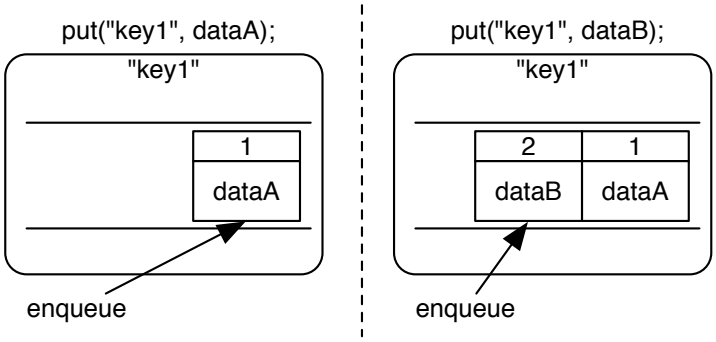

図 4.3: ”put” は重複しない ID を振りながらデータを追加する

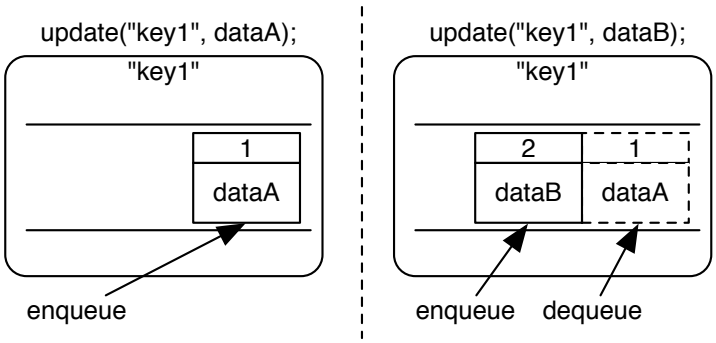

図 4.4: ”update” は先頭データを取り除き、重複しない ID を振りながらデータを追加する

”peek”

”peek” はデータを読み込むための API である。

”peek” は前回読み込んだデータの id を引数で指定する。省略した場合は、 0 が id として渡される。id よりも値の大きい id のデータがキューに含まれていれば、そのデータを receiver に返す。もし id 以下のデータしか無いならば、データの更新が前回の ”peek” 発行時から更新が無いものと考え、リストに格納されて保留される。(図 4.5)

”take” や ”update” によりデータの更新があれば、 ”peek” が直ちに実行される。

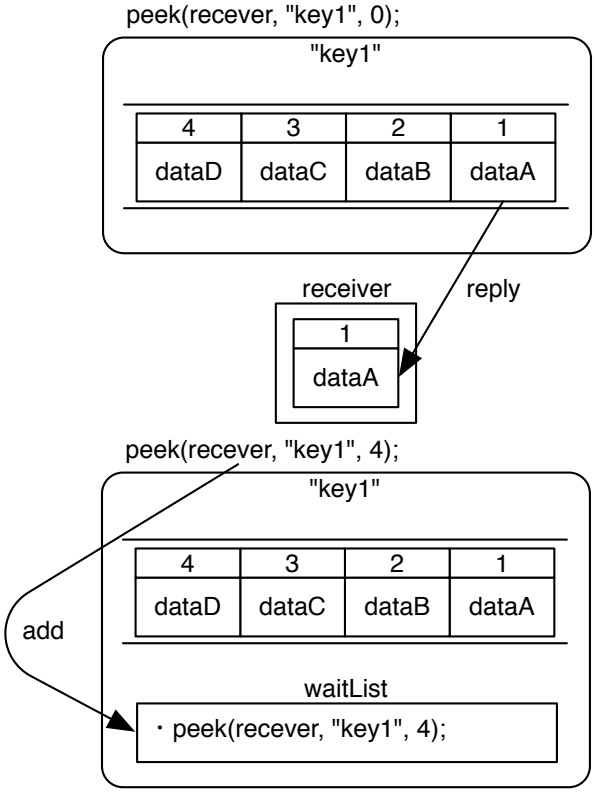

図 4.5: ”peek” はデータを receiver に読み込む。希望のデータがない場合は保留する

”take”

”take” もデータを読み込むための API である。基本的な id に関する部分は ”peek” と同じである。

”peek” との決定的な違いは、読み込まれたデータは Data Segment 内のキューから取り除かれるということである。(図 4.6)

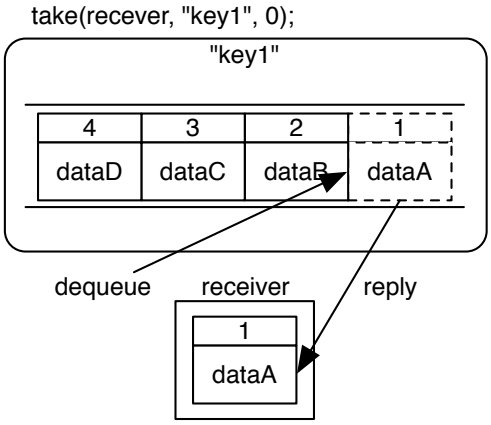

図 4.6: ”take” はデータを receiver に読み込む。その際、読み込んだデータは削除される

4.2.3　コマンドを処理する流れ

これらの API から発行されたコマンドを Data Segment Manager は複数のスレッドから受け取る。その後、 ConcurrentHashMap で文字列であるキーから Data Segment を解決する。

キーが異なればデータセグメント間に依存関係は全く無いので、別スレッドでこれらの API を処理する事ができる。(図 4.7)

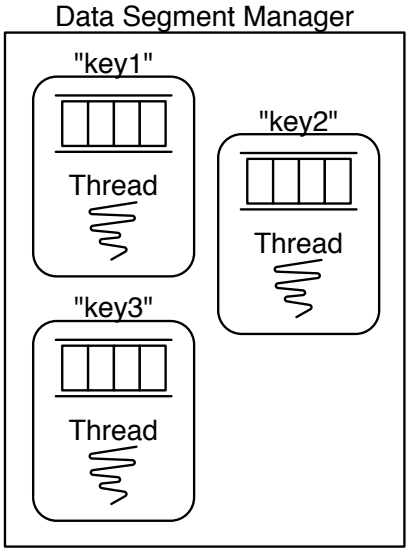

図 4.7: キーごとに Data Segment を処理する Thread を持つ

Data Segment Manager から各キーの Thread へのコマンドの受け渡しには、

java.util.concurrent.LinkedBlockingQueue が使われる。これをコマンドキューと呼ぶことにする。各キーの Thread では、コマンドキューが空になるまでコマンドを繰り返し取り出す。その取り出したコマンドに従って処理が行われる。キューが空になったときは、次のコマンドが挿入されるまでブロックされる。

Data Segment Manager はユーザーが API を使うと、コマンドを作成し、キーから Data Segment を探し、そのコマンドキューに挿入する。(図 4.8)

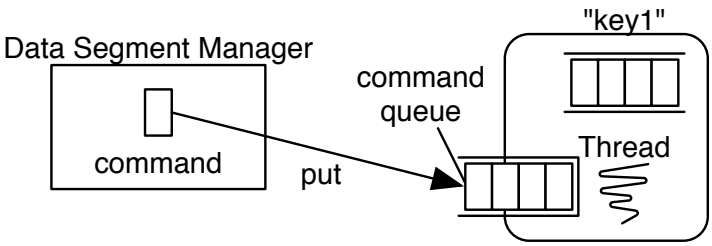

図 4.8: Data Segment Manager はコマンドを作成し、コマンドキューに渡す

4.2.4　Data Segment のデータ表現

Data Segment のデータ表現には MessagePack を利用する。

Java 版の MessagePack の実装 MessagePack for Java を利用して実装を行う。

MessagePack に関して Java におけるデータ表現は以下の 3 段階ある。これらのデータ表現は、型の種類等の制限を伴うが互いに変換可能である。

1. 一般的な Java のクラスオブジェクト
2. MessagePack for Java の `Value` オブジェクト
3. `byte[]` で表現されたバイナリ

Data Segment API では、この MessagePack for Java の Value オブジェクトを用いてデータを表現する。MessagePack は Java のように静的に型付けされたオブジェクトではなく、自己記述的なデータ形式である。MessagePack for Java の Value オブジェクトは MessagePack のバイナリにシリアライズできる型のみで構成された Java のオブジェクトである。そのため、 Value も自己記述式のデータ形式となっている。例えば、 ArrayValue を用いれば、ユーザーがデータを後付けで繋げたりすることも可能になる。

この Value オブジェクトは通信に関わる時は、シリアライズ・デシリアライズを高速に行うことができ、ユーザーは便利なメソッドを用いてオブジェクト内部のデータを閲覧、編集できる。例えば、 Value の toString メソッドは、 JSON 形式で出力してくれる。

(ソースコード 4.8) このように MessagePack Value は Java の静的な型付けを脱却しようとした形式である。

また、ユーザーは一般的なクラスを IDL (Interface Definition Language) のように用いてデータを表現してもよい。そのように使う場合には、クラス宣言時に `@Message` というアノテーションを付けるだけでよい。(ソースコード 4.7) もちろん、 MessagePack で扱うことができるデータのみをフィールドに入れなくてはならない。

ソースコード 4.7: 一般的なクラスを IDL のように使用

```
1 import org.msgpack.annotation.Message
2
3 @Message
4 public class MessagePackTest {
5     public String key;
6     public int val;
7 }
```

ソースコード 4.8: 一般クラスオブジェクトから Value に逆変換して JSON 形式で出力

```
1 msgpackTest = new MessagePackTest();
2 msgpackTest.key = "Test";
3 msgpackTest.val = 1000;
4
5 MessagePack msgpack = new MessagePack();
6 Value value = msgpack.unconvert(msgpackTest);
7
8 System.out.println(value); // ["Test",1000]
```

MessagePack for Java は内部で `@Message` アノテーションが付けられたクラスを変換する時に、 Javassist `http://www.csg.is.titech.ac.jp/~chiba/javassist/` を用いて動的にテンプレートを生成してコンパイルしている。そのため高速に Value オブジェクトと一般クラスオブジェクトの変換、逆変換を行うことができる。

また、 MessagePack は Packer と Unpacker を用いることにより、次から次へとストリームからシーケンシャルにバイナリをシリアライズ・デシリアライズすることもできる。そのため、通信を行うときの入出力部分のコード記述もシンプルになる。(ソースコード 4.9) 従来のプロトコルでは、通信を行う際、パケットサイズ等を含んだ固定長のヘッダーを作成しなくてはいけなかったが、 MessagePack は自己記述的なデータ形式なので、先頭にデータの長さが含まれているため、プログラマーが管理する固定長ヘッダーは必要無くなる。

ソースコード 4.9: Unpacker を用いると通信入力部分の記述がシンプルになる

```
1 MessagePack msgpack = new MessagePack();
2 Unpacker unpacker = msgpack.createUnpacker(socket.getInputStream());
3
4 while (true) {
5     CommandMessage msg = unpacker.read(CommandMessage.class); // block
6     // メッセージを受け取った後の処理を記述するだけでよい。
7 }
```

4.2.5　Remote Data Segment Manager

これまで説明してきた Data Segment Manager はローカルで動作する Data Segment Manager である。これからリモート接続版の Data Segment Manager へと拡張するにあたって区別のためにローカルで動作する Data Segment Manager を Local Data Segment Manager とする。これに対し、リモート接続したホスト上の Local Data Segment Manager の操作を行う機構を Remote Data Segment Manager とする。(図 4.9)

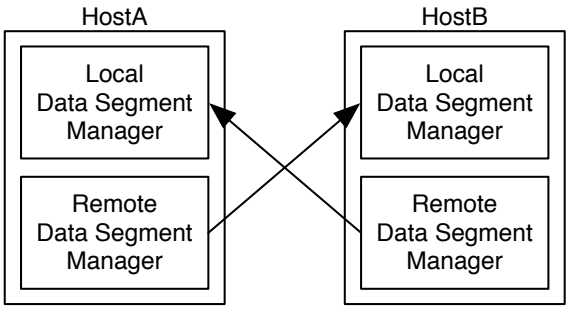

図 4.9: Remote Data Segment Manager は、他のマシン上の Local Data Segment Manager を操作できる

ローカルでもリモートでも、 Data Segment Manager の API は共通なので、継承して実装する。(図 4.10) そのことにより、 Remote Data Segment Manager にアクセスするときもローカルと同じようにアクセスすることができる。

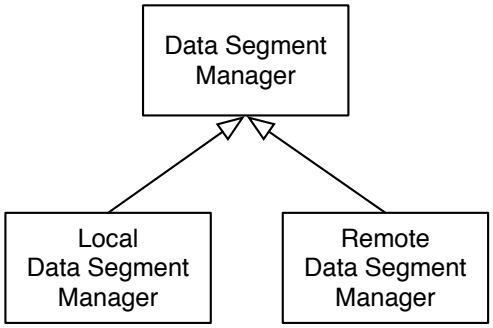

図 4.10: Data Segment Manager を継承して、 Local DSM と Remote DSM を作成する

4.2.6　ネットワーク周りの実装

Alice において、処理のボトルネックになる部分はネットワーク通信周りである。そのため、ネットワーク周りの処理をスレッド分けし、パイプライン処理を施している。これ

は、 SEDA アーキテクチャのステージ処理と似ている。
さらに通信に関する部分を、送信と受信の二つのスレッドに分割している。(図 4.11)

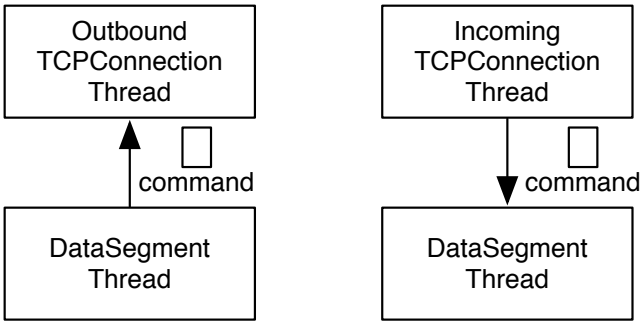

図 4.11: Data Segment の処理と TCP の送受信処理をスレッド分割してパイプライン化

互いのスレッド間の通信には、他のパターンと同じく、 LinkedBlockingQueue を用いてコマンドを投げている。

4.3 Code Segment

前節では、 Data Segment の詳細な設計と実装を示した。本節では、 それら Data Segment API を用いて記述する Code Segment の使い方やそれらの実装方法について示す。
Code Segment は、タスクのことである。 Code Segment をユーザーが記述するときに、記述している Code Segment 内で使用する Data Segment を記述する。すなわち、 Code Segment と Data Segment の依存関係を記述する。
具体的に使用する Data Segment は、 Code Segment の入出力に相当する。それらを、それぞれ Input Data Segment、 Output Data Segment とする。 Input Data Segment、Output Data Segment はそれぞれ複数記述できる。

4.3.1 基本的な Code Segment の実行方法

Code Segment の記述方法であるが、 Start Code Segment と呼ばれる、開始 Code Segment を記述するところから始まる。 Start Code Segment は、最初の Code Segment なので、どの Data Segment にも依存しない。すなわち、 Input Data Segment はない。
この Start Code Segment を main メソッド内で new することで、 Code Segment の実行を開始させることができる。 Input Data Segment がない Code Segment を実行するためには、 execute メソッドを実行する。(ソースコード 4.10)

ソースコード 4.10: 最初の Code Segment を実行させる方法

```
1  public class TestLocalAlice {
2
3      public static void main(String args[]) {
4          new StartCodeSegment().execute();
5      }
6
7  }
```

Start Code Segment は CodeSegment を継承して作成する。 StartCodeSegment の例 (ソースコード 4.11) では、次の Code Segment である、 TestCodeSegment (ソースコード 4.12) を生成している。この TestCodeSegment は input1 という Input Data Segment を ”peek” している。

ソースコード 4.11: Start Code Segment の例

```
1  public class StartCodeSegment extends CodeSegment {
2
3      @Override
4      public void run() {
5          System.out.println("run␣StartCodeSegment");
6
7          TestCodeSegment cs = new TestCodeSegment();
8          cs.input1.setKey("local", "key1");
9
10         System.out.println("create␣TestCodeSegment");
11
12         ods.update("local", "key1", "String␣data");
13     }
14
15 }
```

ソースコード 4.12: Code Segment の例

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

この input1 に対して、具体的な Data Segment のキーを割り当てているのが、 setKey メソッドである。この例では、 TestCodeSegment の input1 が local.key1 に依存するよ

うに設定している。

TestCodeSegment の内部では、情報を標準出力に出力した後、再度、自身と同じである TestCodeSegment を生成し、先ほどと同じようにキーを割り当てている。 input1.index の値は、 Data Segment 内部で ”put” や ”update” が実行されたときに、インクリメントしているシーケンシャルでユニークな ID 値である。

このように、今回の例題では、10 回ほど TestCodeSegment をループしていることがわかる。(図 4.12)

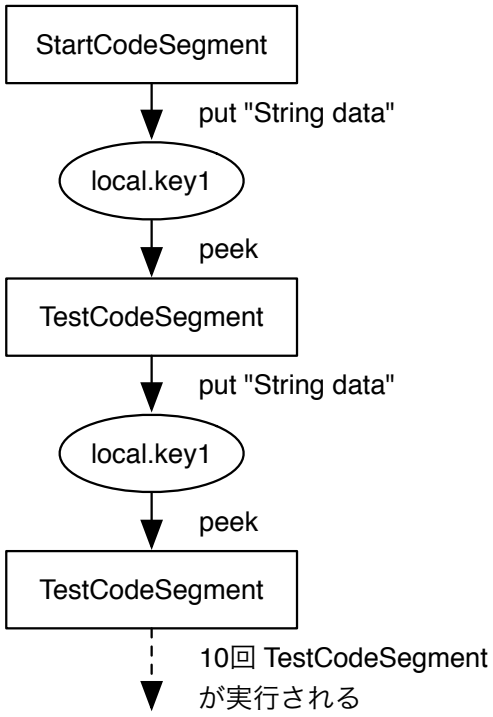

図 4.12: StartCodeSegment 以降の実行過程

ここでは、 Code Segment の大まかな利用方法を示した。次は、ここで利用している API の詳細である。

4.3.2　Code Segment の詳細な書き方

新たな Code Segment をユーザーが記述するときは、 CodeSegment を継承して記述する。その CodeSegment は、 InputDataSegmentManager と OutputDataSegmentManager を利用する事ができる。

InputDataSegmentManager

InputDataSegmentManager は、 ids という、 CodeSegment のフィールドを使うことでアクセスすることができる。

- `Receiver create(CommandType type)`

InputDataSegmentManager の基本機能は、 create である。create で新しい Data Segment の Receiver を生成することができる。create の引数は、 CommandType である。ここで指定できる CommandType は PEEK か TAKE である。ここで定義しているのは、 ”peek” や ”take” を実行した結果を格納する受け皿であるので、実際に ”peek” や ”take” を実行しているわけではない。実際にコマンドが実行されるのは、 Receiver 内にある setKey メソッドである。

- `void setKey(String managerKey, String key, int id)`

setKey メソッドにより、どこの Data Segment のなんというキーを実行するかということを指定する。それにより、 ”peek” や ”take” のすべての引数が揃い、実行される。そして、そのコマンドの結果がレスポンスとして Receiver に届き次第、 Code Segment の実行が行われる。(図 4.13)

OutputDataSegmentManager

OutputDataSegmentManager は、 ods という、 CodeSegment のフィールドを使うことでアクセスすることができる。

OutputDataSegmentManager は、 ”put” または ”update” を実行することができる。

- `void put(String managerKey, String key, Value val)`
- `void update(String managerKey, String key, Value val)`

4.3.3 Code Segment Manager

Code Segment Manager は、 Code Segment の並列実行を司る部分である。 各 Code Segment は一つずつ InputDataSegmentManager を持っているが、 Receiver を作成し、キーを解決し、 Data Segment API を実行し、 Receiver にレスポンスが返ってきたあと、依存関係が解決された Input Data Segment Manager は、Code Segment の実行を Code Segment Manager に依頼する。

Code Segment Manager はスレッドプールを持っている。Input Data Segment Manager から Code Segment を受け取った Code Segment Manager は、それをスレッドプールへと投入する。

スレッドプールには java.util.concurrent.ThreadPoolExecutor を利用する。(ソースコード 4.13)

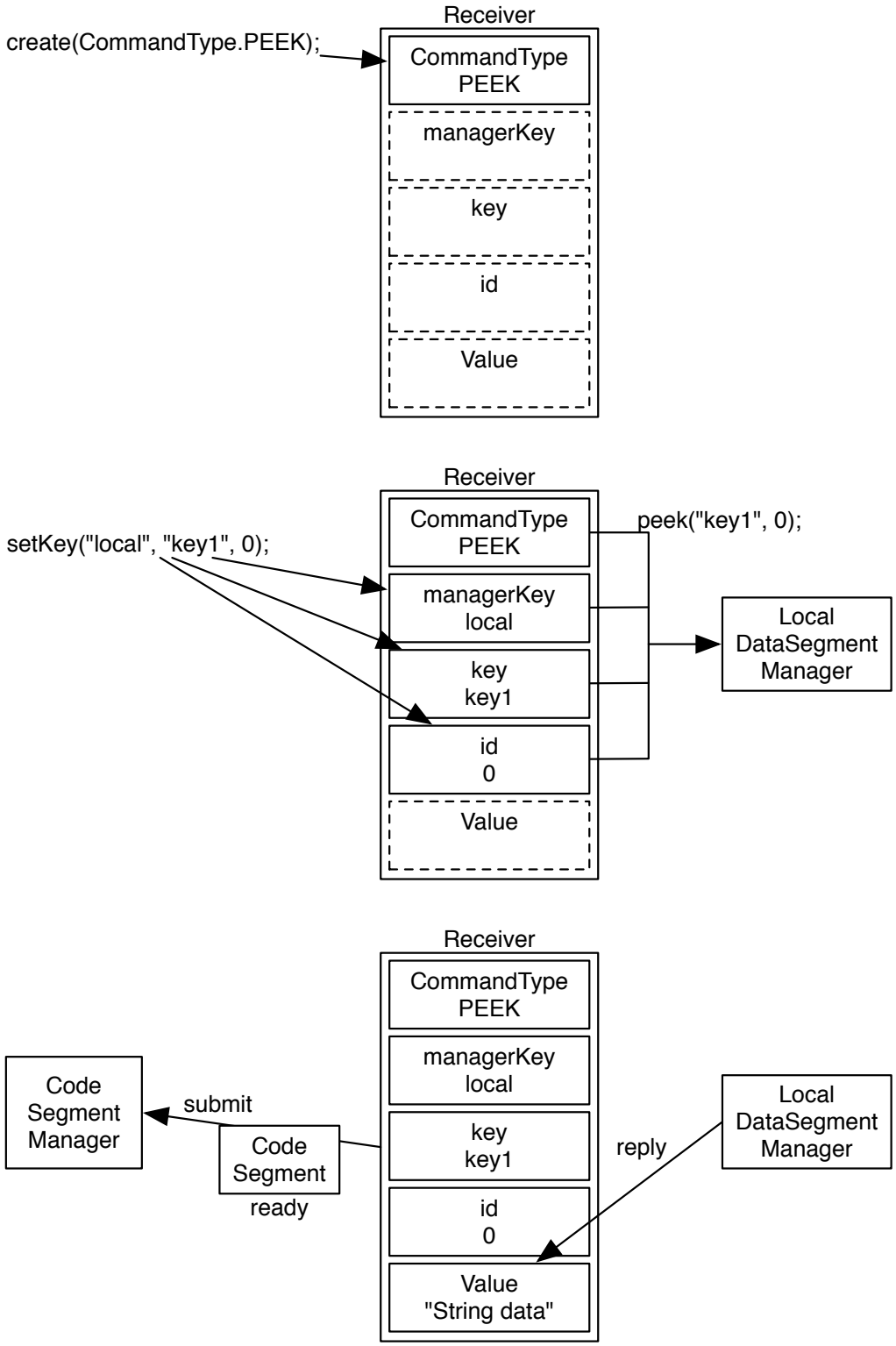

図 4.13: Data Segment の依存関係の設定と解決と Code Segment が実行されるまで

ソースコード 4.13: java.util.concurrent.ThreadPoolExecutor の利用

```
codeSegmentExecutor = new ThreadPoolExecutor(1,
                          Runtime.getRuntime().availableProcessors(),
                          Integer.MAX_VALUE,
                          TimeUnit.SECONDS,
                          new LinkedBlockingQueue<Runnable>());}
```

第一引数は、スレッドプールが最低保持するスレッドの数である。最低、1 だけ保持するようにした。

第二引数は、スレッドプールが最大保持するスレッドの数である。マシンに応じたコアの数だけスレッドを持っておくと、効率がよいので、
`Runtime.getRuntime().availableProcessors()` でマシンのコア数を入力している。

第三、第四引数は、スレッドを保持する期間である。

第五引数は、投入される Runnable のキュー待ちを行うブロッキングキューである。

ThreadPoolExecutor は、 Runnable がキューへ投入される度、必要に応じてスレッドの最大保持数まで上げて処理を行う。

Code Segment Manager はシングルトンパターンで作成されている。

Code Segment Manager は個別スレッドを持っており、 Executor に投入する部分もパイプライン化を行なっている。(ソースコード 4.14)

ソースコード 4.14: ジョブ投入関連処理

```
private CodeSegmentManager() {
    Runnable prepareThread = new Runnable() {

        @Override
        public void run() {
            while (true) {
                try {
                    CodeSegment cs = readyQueue.take();
                    codeSegmentExecutor.execute(cs);
                } catch (InterruptedException e) {
                    // 例外処理
                }
            }
        }

    };
    new Thread(prepareThread, "CodeSegmentManager").start();
}

public static void submit(CodeSegment cs) {
    try {
        CodeSegmentManager.get().readyQueue.put(cs);
    } catch (InterruptedException e) {
        // 例外処理
    }
}
```

4.4 Topology Manager

前節では、 Code Segment の利用方法とその実装を示した。本節では、 Alice 同士の接続トポロジーを管理する、 Topology Manager について示す。

Topology Manager 関連の通信処理は全て、前節で実装した Code Segment を用いて記述している。

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 4.14)

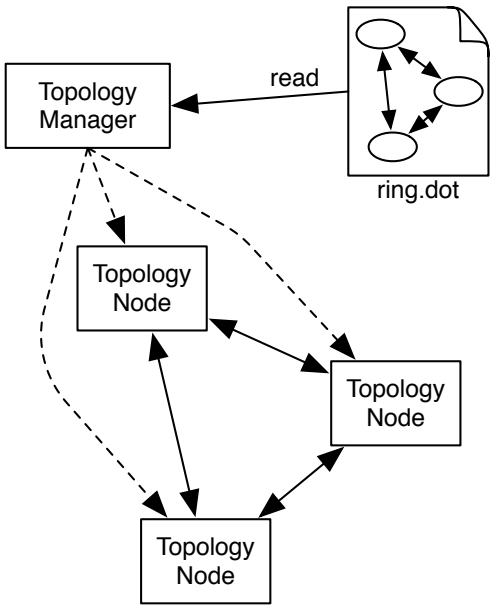

図 4.14: Topology Manager はトポロジーファイルの通りに Topology Node を接続させる

4.4.1 Topology Manager の設定ファイル

まず、 Topology Manager は、トポロジーファイルを読み込む。トポロジーファイルは、DOT Language (`http://www.graphviz.org/`) 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種である。この DOT Language のグラフ構造を利用して、 Topology Node 間の接続を表現する。(ソースコード 4.15)

ソースコード 4.15: 3 台でリングを組んだ時の例

```
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 }
```

また、 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。(図 4.15) 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。

- `dot -T png ring.dot -o ring.png`

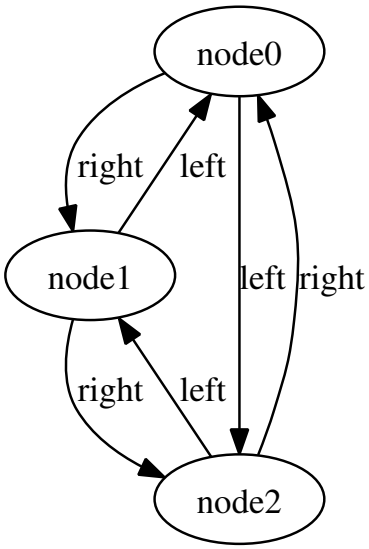

図 4.15: dot コマンドで生成された 3 台で構築されたリングのグラフ

ノード間の接続には label を用いて名前が振られている。この接続名を用いて、ユーザーは接続先の Data Segment Manager にアクセスすることができる。 Receiver に setKey していた部分の、 managerKey がこれに相当する。

例えば、リングトポロジーの場合、 ”right” と ”left” という二つの接続が提供される。

また、各ノードには Topology Manager への先着順に ”node0”、 ”node1”、 ”node2” のように、トポロジーファイルに記述されている抽象名が割り当てられる。

4.4.2 Topology Manager の初期化

Topology Manager は起動したら、コマンドライン引数で渡されたトポロジーファイルを読み込む。読み込んだファイルをパースし、ノードの抽象名同士の関わりをオブジェクトに起こす。

その後、 Topology Node からの参加表明を待つ。

4.4.3 Topology Node の参加表明処理

Topology Node は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。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。

Topology Manager は、自分自身のキー ”hosts” を take しているので、 Topology Node からの参加表明をシーケンシャルに読み込むことが可能となる。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、参加表明した Topology Node に依存している Toology Manager 上の Topology Node 名のキーに、参加表明した Topology Node の接続情報 (IP アドレス、ポート番号等) を put する。このため、準備ができていない Topology Node がいたとしても、準備ができ次第、 Topology Manager へ接続し、自分自身の接続名を take することにより、シーケンシャルに取り出し、データに合わせて接続していけばよい。 (図 4.16)

4.4.4 接続終了後の Topology Node

Topology Node は、ライブラリーのように用いる事ができる。

Topology Node を利用するには、TopologyNode を new するだけでよい。TopologyNode の第一引数は Alice デーモンの設定オブジェクトであり、第二引数は開始 Code Segment である。ここで指定する開始 Code Segment がトポロジーに必要な接続完了後に呼ばれる。

ソースコード 4.16: Topology Node の使い方

```
1 public class RingTopology {
2
3     public static void main(String[] args) {
4         RingTopologyConfig conf = new RingTopologyConfig(args);
5         new TopologyNode(conf, new StartRing(conf));
6     }
7
8 }
```

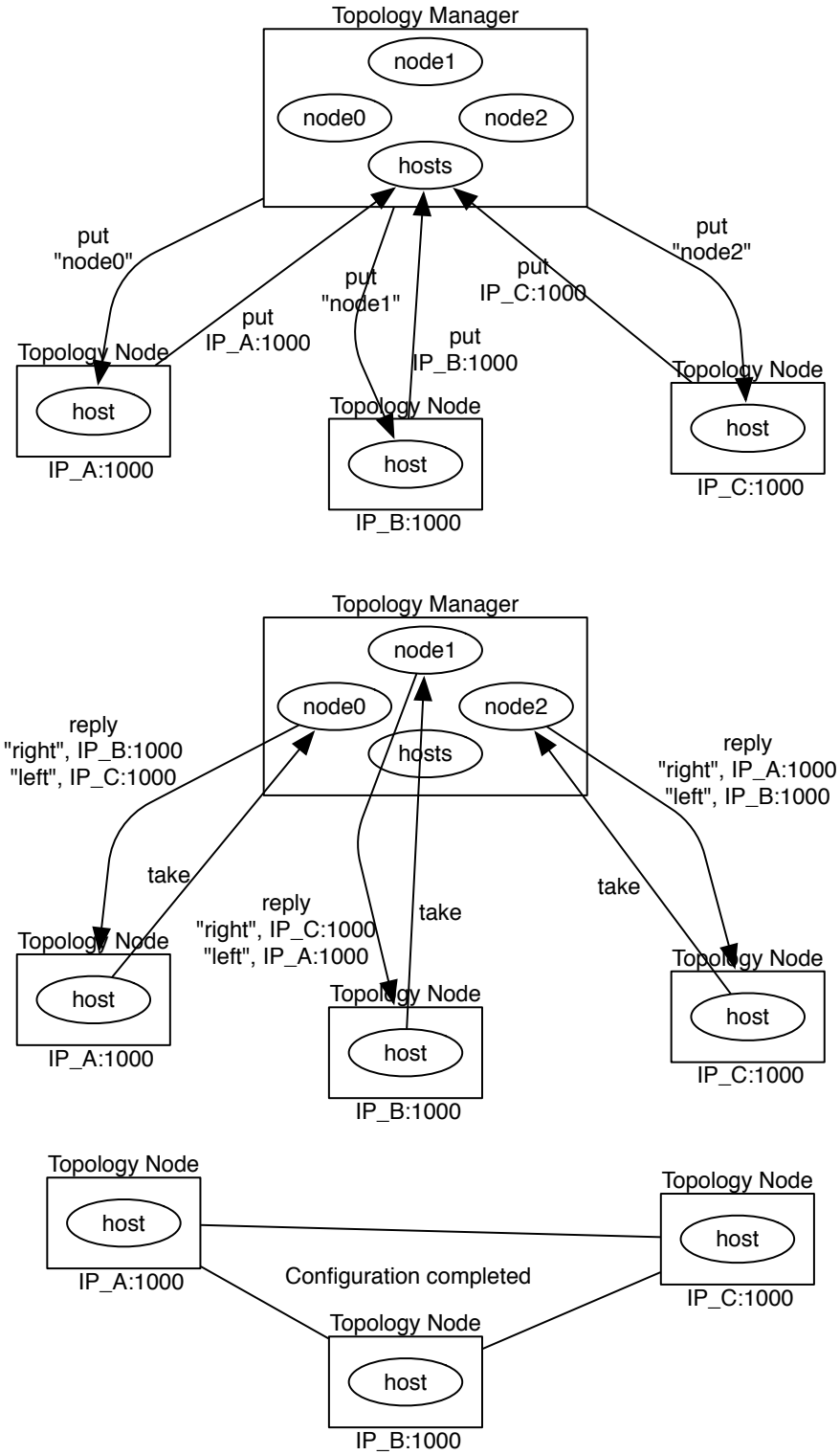

図 4.16: Topology Manager と Topology Node 間の通信

第5章　分散フレームワーク Alice の評価

前章までは、 Alice の設計と実装について示した。本章では、 Alice を用いた実験方法等についてまとめ、先行研究である Federated Linda との性能比較を行い、 Alice を評価する。

5.1　TORQUE Resource Manager を用いた実験方法

Alice の性能を実験する際に、学科にある共用のブレードサーバーを用いた。共用のブレードサーバーを利用するため、実験を行う際は、自身の実験の正確さと、他の利用者の実験の邪魔をしないためにも、他の利用者とリソースが競合しないようにする必要がある。つまり、ジョブスケジューラーを利用しなくてはならない。

今回の実験では、 TORQUE Resource Manager (`http://www.adaptivecomputing.com/products/torque.php`) を利用した。

5.1.1　TORQUE Resource Manager

TORQUE は、クラスター等の分散コンピューティング環境を複数名で共有する場合に有効である。

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 5.1) 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。

ユーザーは、マスターにリモートログインして TORQUE を利用する。

ユーザーは、ジョブをシェルスクリプトを用いて記述し、ジョブスケジューラーへと投入する。その時、利用したいマシン数、 CPU コア数等を詳細にオプションで指定する事ができる。

ユーザーからのジョブを受け取った TORQUE は、利用可能な状態にあるマシンが揃い次第、受け取ったジョブを実行する。

5.1.2　TORQUE のジョブの書き方

TORQUE のジョブは、シェルスクリプトを用いて記述する。

TORQUE のジョブは実行されたときに、いくつかの実験に関する環境変数を得る。

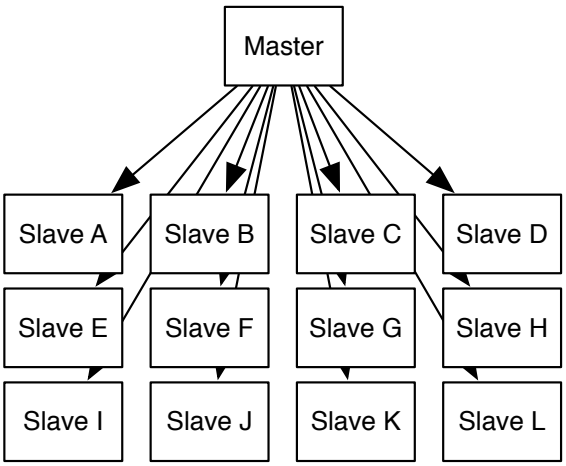

図 5.1: TORQUE の構成

- PBS_NODEFILE
  すべての参加マシンが列挙されているファイルへのパス
- PBS_NUM_NODES
  全ての参加マシン数
- PBS_NUM_PPN
  参加マシン一台あたりのコア数

PBS_NODEFILE に記述されているマシン名に順番に、それぞれ走らせたいコマンドを ssh で記述していく。(ソースコード 5.1)

ソースコード 5.1: 割り当てられたマシンにログインして hostname コマンドを走らせる例

```
1 function run() {
2     while read node
3     do
4         ssh $node hostname < /dev/null
5     done
6     wait
7 }
8 run < $PBS_NODEFILE
```

ジョブを投入するには、 qsub コマンドを用いる。(ソースコード 5.2)

ソースコード 5.2: 10 台 (1 台あたり 1 コア) で走らせる例

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

しかし、複数のコアを持つマシン上で、1 マシン 1 プロセスで分散プログラムを動かしたい場合、問題が発生する。例えば、4 コアのマシンは 4 つリソースがあると計算されるので重複して割り当てられるのである。

そこで、正確に 1 マシン 1 プロセスの実験を行うには、1 プロセスに 4 コア分だけリソースを割り当てないといけない。(ソースコード 5.3)

ソースコード 5.3: 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
```

しかしこの場合、PBS_NODEFILE の中に含まれるマシンは 4 つずつ重複して登録されている。(PBS_NUM_NODES は重複していない)

そのため、 uniq コマンド等でマシンリストの重複を除いてから ssh コマンドを実行していかなくてはならない。(ソースコード 5.4)

ソースコード 5.4: ノードファイルからマシンの重複を除く

```
1 uniq $PBS\_NODEFILE /tmp/nodes
```

ソースコード 5.5: Alice でリングトポロジーの実験に使ったジョブ

```
1  #!/bin/bash
2  #
3  # Alice Ring Topology
4  #
5  #PBS -q dque
6  #PBS -N AliceRingTopology
7  #PBS -l walltime=00:05:00
8
9  alicepath=/home/mass/student/k108572/test
10 node_num=‘expr $PBS_NUM_NODES - 1‘ # TopologyManager の分を 1引く
11 port=10000 # 利用するポート番号
12 count=100 # リングを回る回数
13 size=4096 # リングを回すメッセージのサイズ
14
15 function run() {
16     read serv
17     ssh $serv "ruby␣$alicepath/ring.rb␣$node_num␣>␣/tmp/ring.dot" < /dev/
           null
18     ssh $serv killall java < /dev/null
19     ssh $serv java -cp $alicepath/Alice.jar alice.topology.manager.
           TopologyManager -p $port -conf /tmp/ring.dot -log $alicepath/log/
           manager.log -level info < /dev/null &
20     cnt=0
21     while read node
22     do
23       ssh $node killall java < /dev/null
24       ssh $node java -cp $alicepath/Alice.jar alice.test.topology.ring.
             RingTopology -host $serv -port $port -p $port -log $alicepath/log
             /ring${cnt}.log -log -level info -count $count -size $size -
             nodeNu
25 m $node_num < /dev/null &
26       cnt=‘expr $cnt + 1‘
27     done
28     wait
29 }
30
31 uniq $PBS_NODEFILE /tmp/nodes
32 run < /tmp/nodes
```

ソースコード 5.6: 5 台から 45 台までを 1 刻みで実行するシェルスクリプト

```
1 #!/bin/sh
2 count=6
3 while [ $count -le 46 ];
4 do
5     qsub -l nodes=$count:ppn=4 ring.sh
6     count=‘expr $count + 1‘
7 done
```

5.2 Federated Linda との性能比較

続いて、 Alice と Federated Linda の性能比較を行う。実験内容は、 Federated Linda がかつて実験したリングトポロジーによる実験を、 Alice 上で再現した。

5.2.1 実験環境

今回はブレードサーバー (表 5.1) 上の仮想マシン (表 5.2) による仮想クラスタ環境を用いて実験した。

表 5.1: 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 5.2: 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

5.2.2　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 5.2)

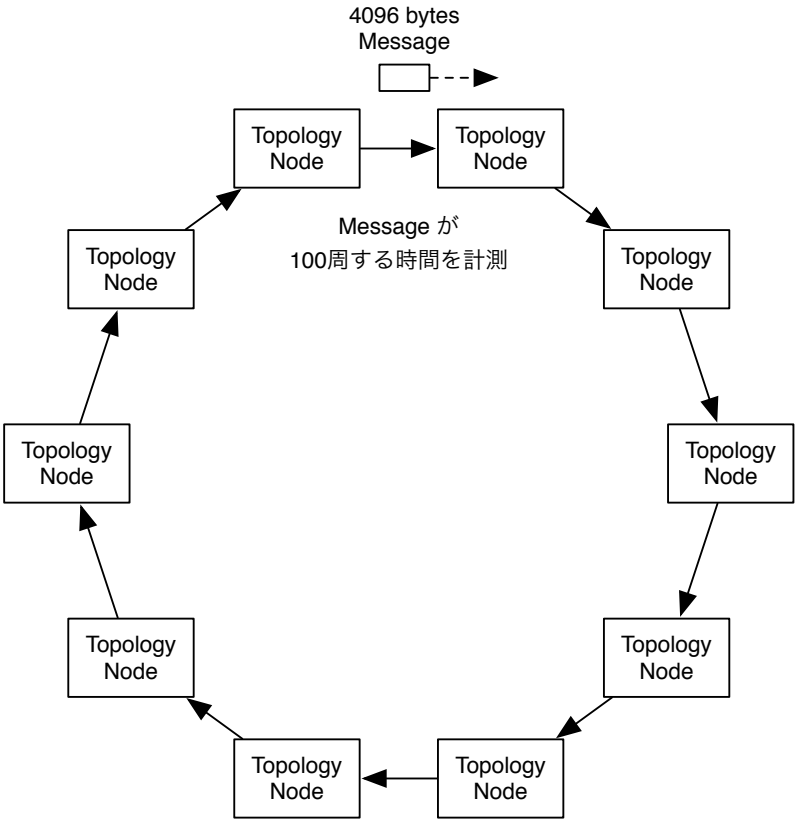

図 5.2: 100 周にかかる時間を計測し、1 周あたりの平均時間を求める

パケットのサイズは、 10 bytes の比較的軽いデータから、 4096 bytes の比較的重いファイルを用いて実験した。

今回の実験では、トポロジーの構築時間は実験に含めてはいない。純粋な API の実行速度のみで比較している。

5.2.3　実験結果

実験結果は以下のようになった。(図 5.3)(図 5.4)

これらに更に追加して、 10KB のデータを送信した時と 100KB のデータを送信した時の時間も測った。(図 5.5)(図 5.6)

このように、 45 台で実験した場合、 12 ms 程、Alice が遅いことが分かる。

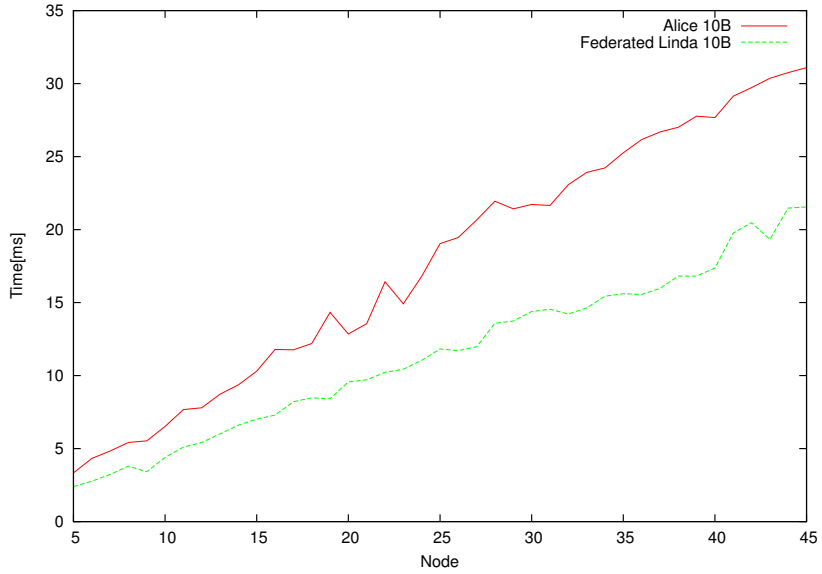

図 5.3: 10 bytes のデータを 100 周させたときの 1 周にかかる平均時間

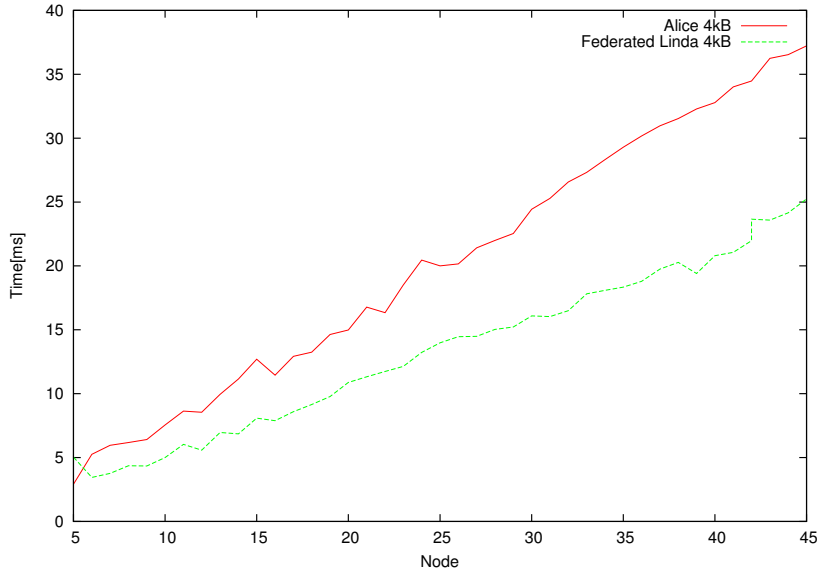

図 5.4: 4096 bytes のデータを 100 周させたときの 1 周にかかる平均時間

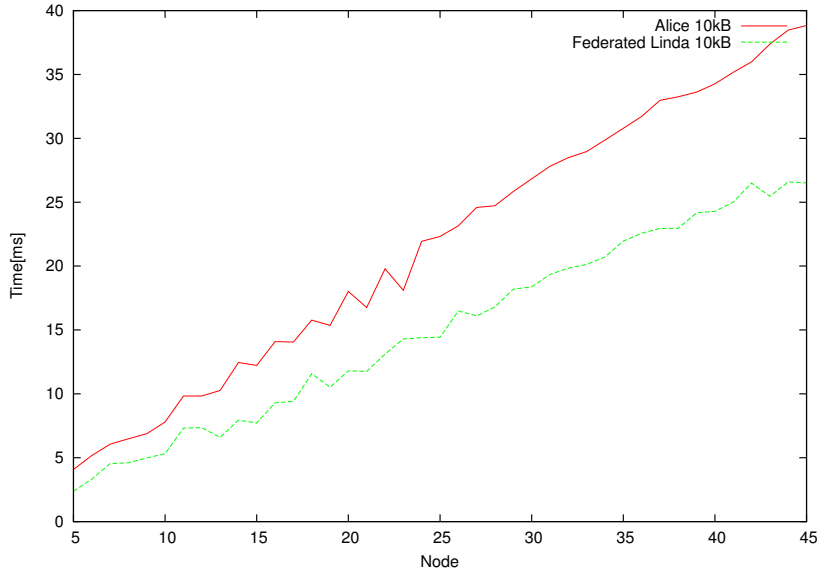

図 5.5: 10 Kbytes のデータを 100 周させたときの 1 周にかかる平均時間

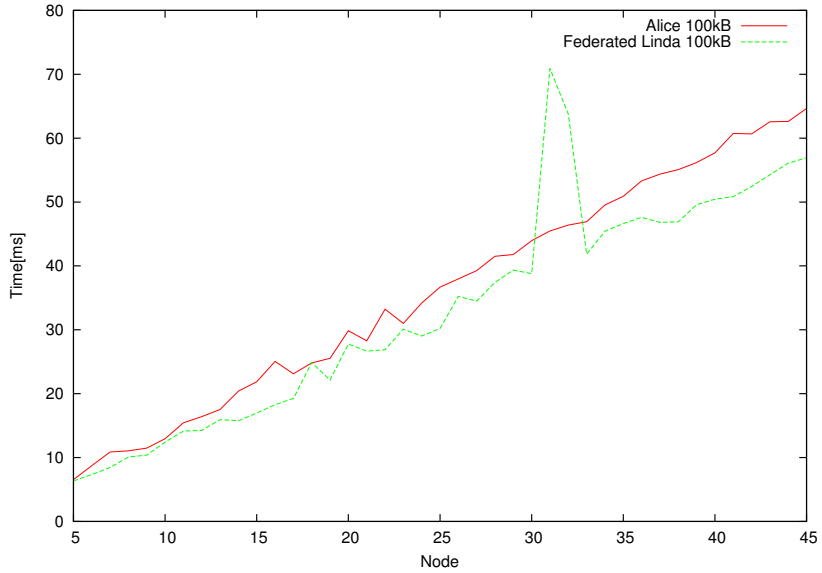

図 5.6: 100 Kbytes のデータを 100 周させたときの 1 周にかかる平均時間

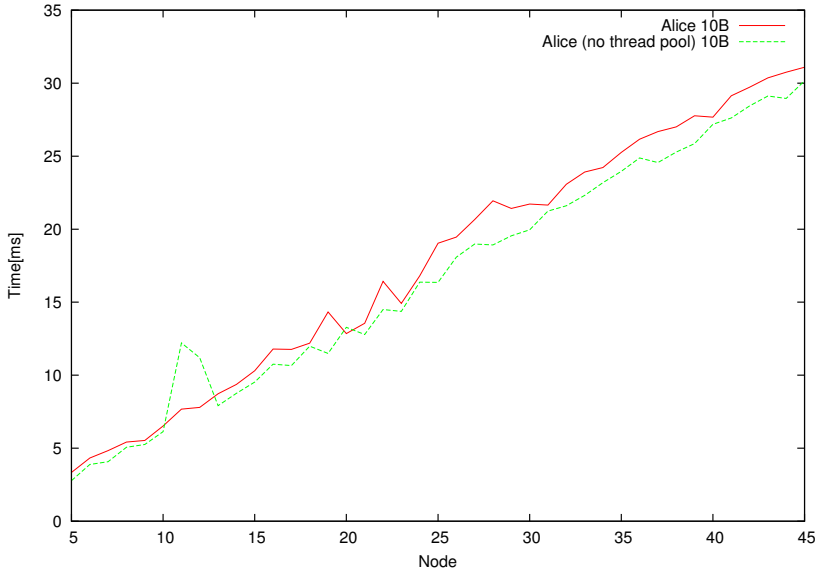

図 5.7: Code Segment のスレッドプールを使用せずに、 10 bytes のデータを回した時の実験結果

また、 Code Segment のスレッドプールを使用せずに、そのまま実行した場合の実験も行った。(図 5.7)

この場合、スレッドプールを使用しないほうが速いことが分かる。

5.2.4　考察

今回の結果により、 Alice は本研究室で長年チューニングされてきた Federated Linda にはまだ勝てていないことが分かった。

しかし、図 fig:ringnothread の結果を見ると、 Code Segment の並列化を断ったほうが、結果が良くなっている。今回の実験は、リングを 1 周させるという、同時に複数のタスクが走らない実験であった。これは全てシングルスレッドで作成されている Federated Linda が得意なシーケンシャルな実験だったので、こうなったことが分かる。

最後に、今回の実験において Alice が実行する処理の段階をまとめてみた。(図 5.8) 赤い部分がスレッドで走る部分である。また、太い矢印が、 LinkedBlockingQueue による、データの中継地点である。

このように、複数のスレッド分けを行なっているが、それらが直列的に実行されている事がわかる。なので、この実験では Alice のスレッド分けした実装が逆に悪影響を与えている事がわかる。

Federated Linda に勝てるような実験は、もっと多数のマシンからアクセスが同時に行われるような実験をしたほうがよい。

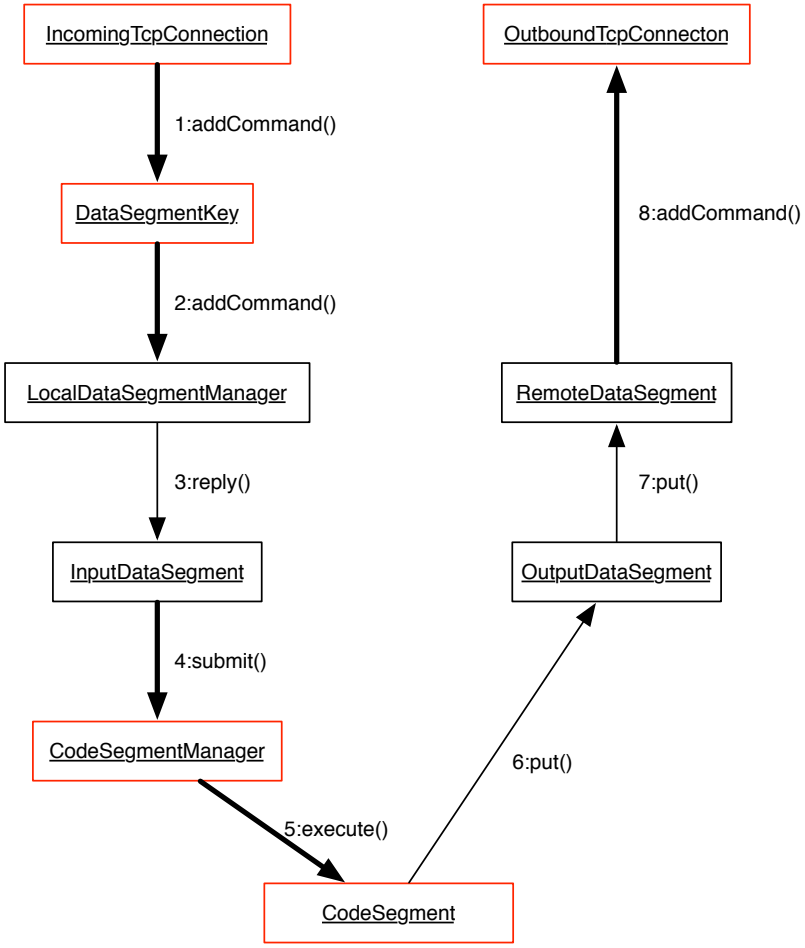

図 5.8: リングの実験において 1 台の Alice が実行する処理の段階

第6章　結論

6.1　まとめ

今回の研究では、まず始めに、先行研究である Federated Linda の開発から得られた知見をまとめ、分散プログラミングに重要な事とは何かについて論じた。

その次に、本研究室で開発している Cerium の新設計のアイディアを踏襲し、 Data Segment と Code Segment という単位に分けてプログラミングを行うというスタイルを取り入れ、新しい分散フレームワークの概観を設計した。

さらに、それらの設計をもとに更に詳細な設計を重ね、 Java を用いて Alice の実装を行った。

最後に Alice の動作方法や、クラスター上での実験方法についてまとめ、 Federated Linda との性能比較等を行った。

実際に実装、実験してみることで、初めて得られた知見がたくさんあり、新たな問題点もたくさん見えてきた。

今後、このフレームワークを用いて、たくさんのプロトコルを提案し、実装し、検証していく過程で、分散プログラミングの分野に有意義な影響を与えることができれば幸いである。

6.2　今後の課題

6.2.1　データの永続性の確保

現在の Alice のデータベースである Data Segment は、プロセスが終了した場合、メモリ解放と共に失われる。

これらを解決するには、 Data Segment を他の Key-Value Store 等のシステムに格納し、永続性を確保することが必要である。

または、自前でファイルに出力することを考えても良い。 Alice では Data Segment の表現に MessagePack を利用しているため、 Data Segment のシリアライズ化が可能である。そのため、比較的容易に Data Segment の永続性をサポートできるのではないかと考えられる。

6.2.2 Data Segment のキーごとにスレッドを持つ事による問題点

現在の Alice では、 Data Segment のキーごとに Data Segment を処理するスレッドを持っている。このため、 Data Segment のキーが異なる Data Segment にアクセスする API は並列実行することが可能となっている。

しかし、これからキーを大量に利用するアプリケーションの開発を行うことになった際、それらのキー全てがスレッドを持つことになってしまう。そのため、スレッドの切り替えが頻繁に起こり、処理効率の低下が懸念される。

そこで、大量にあるキーごとのスレッドを 1 本にまとめる方法と、スレッドプールを利用して、ある一定の数までのキーだけ並列処理できる方法が挙げられる。

それらを検証するための例題を作成し、どちらの方法を用いたほうが効率が良くなるのか実験をしていかねばならない。

6.2.3 Federated Linda の実行速度を目指したチューニング

今回の実験結果では、シンプルな実装であった Federated Linda のリングトポロジーの実装速度に勝ることは出来なかった。

マルチスレッドによる多段パイプラインを備えた Alice は、リングを 1 周させるようなシンプルなプロトコルではなく、複雑な通信が頻繁に飛び交うプロトコル向きだとは言えども、シンプルなプロトコルでも Federated Linda に勝てるように、プロファイルを重ねてチューニングを行なっていきたい。これからボトルネックの炙り出しを行っていく必要がある。

6.2.4 HashMap の多用による処理速度の低下

先行研究である Federated Linda に比べ、 Data Segment Manager のマネージャーキーによる探索や、 Data Segment のキーによる探索など、 HashMap を多用している。

しかし、 Code Segment のコーディングを行なっているときに、キーが固定であるケースも多いため、探索数を減らすことができると考える。

また、 Javassist などのリフレクションを行えるツールを利用して、元になったソースから動的にコンパイルを行うという方法を模索していきたい。

6.2.5 Code Segment のユーザビリティ

Federated Linda に比べ、分散プログラミングをシンプルに行えるようになったとはいえ、まだまだユーザーの Code Segment への記述量は多い。

この記述量を減らすためには、コード生成系を開発すればよい。

また、 Javassist などのリフレクションを行えるツールを利用して、動的にコンパイルを行い、依存関係を解決できるようにするとよいかもしれない。

6.2.6　デバッグ方法の提案と実装

現在、 Alice はログファイルによるデバッグしか行なっておらず、実行時にリアルタイムに Data Segment の内容を調べたり、現在の依存関係を調べる方法がない。もちろん、全てのマシンに接続を張るというモデルでは、確実にスケールしない。そのため、効率の良いトポロジーとルーティングを使って、デバッグデータの収集を行う方法を考えていかなくてはならない。

謝辞

本研究を行うにあたって、ご多忙にも関わらず日頃より多くのご助言、ご指導を頂きました河野真治助教授に心より感謝いたします。

また、本研究に大きく役立つ技術的指導を賜りました、与儀健人さん、小野雅俊さん、並びに情報工学科の先生方に感謝いたします。

先行研究である Federated Linda や Cerium がなければ、本研究はありませんでした。Federated Linda や Cerium 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。

研究を行うにあたり貴重なご意見と日々の強力を頂いた、金城裕さん、玉城将士さん、並びに並列信頼研究室の皆様に深く感謝いたします。

最後に、長年にわたり理解を示し、援助してくれた家族に感謝します。

参考文献

[1] 河野真治, 仲宗根雅臣. 同期型タプ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 を持たない連邦型タプルスペース 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, 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] 渕田良彦, 河野真治. 連邦型タプ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 会大会, 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 会大会, Sep 2011.

[9] Avinash Lakshman and Prashant Malik. Cassandra - a decentralized structured storage system. *LADIS*, Mar 2003.

[10] Matt Welsh, David Culler, and Eric Brewer. Seda: An architecture for well-conditioned, scalable internet services. *Computer Science Division*, Mar 2003.

[11] 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 会大会, Sep 2010.

[12] 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.

[13] 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 会大会, Sep 2011.

[14] 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.

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