修士 (工学) 学位論文
Master's Thesis of Engineering

修士論文のテンプレート

2022 年 3 月
March 2022

安田 亮
Ryo Yasuda

琉球大学
大学院理工学研究科
情報工学専攻

Information Engineering Course
Graduate School of Engineering and Science
University of the Ryukyus

指導教員：教授 和田 知久
Supervisor: Prof. Tomohisa Wada

本論文は、修士 (工学) の学位論文として適切であると認める。

論　文　審　査　会

印

(主　査)　　和田　知久

印

(副　査)　　山田　孝治

印

(副　査)　　當間　愛晃

印

(副　査)　　河野　真治

要旨

tsetd

Abstract

hogefuga

発表履歴

- 安田 亮, 河野 真治. Multicast Wifi VNC の実装と評価. 情報処理学会 システムソフトウェアとオペレーティング・システム研究会 (OS), May, 2020
- 安田 亮, 河野 真治. 継続を使用する並列分散フレームワークの Unity 実装. 情報処理会 システムソフトウェアとオペレーティング・システム研究会（OS）, May, 2021

目 次

研究関連論文業績 **iv**

第 1 章　序論 **5**
1.1　はじめに 5
1.2　hogehoge 5
1.2.1　foofoo 5
1.2.2　研究目的 5
1.3　論文の構成 5

第 2 章　Christie **6**
2.1　Christie の基礎概念 6
2.2　annotation を使用したデータの記述 9
2.3　データの型整合性 10
2.4　CodeGear の記述方法 11
2.5　DataGearManager の複数立ち上げ 12
2.6　通信フロー 14

第 3 章　まとめ **18**
3.1　総括 18
3.2　今後の課題 18
3.2.1　hogehoge 18

謝辞 **19**

参考文献 **20**

付録 **20**

図 目 次

2.1 各 Gear の関係性 7
2.2 同一プロセスでの Christie の複数インスタンス立ち上げ 8
2.3 RemoteDGM を介した CGM 間の通信 13
2.4 LocalDGM に Take した際のフロー 14
2.5 RemoteDGM から Put された際のフロー 15
2.6 RemoteDGM に Take した際のフロー 16

表 目 次

ソースコード目次

2.1 Take の例 9
2.2 TakeFrom の例 9
2.3 DG のデータを扱う例 10
2.4 StartCodeGear の記述例 11
2.5 CodeGear の記述例 11
2.6 DG として送信されるオブジェクトのクラス 11
2.7 LocalDGM を 2 つ作る例 12

第1章　序論

1.1　はじめに

1.2　hogehoge

1.2.1　foofoo

1.2.2　研究目的

ほげほげほげ

1.3　論文の構成

fugafuga

1章

2章

3章

4章

5章

6章

第2章　Christie

Christie とは Java で記述された並列分散通信フレームワークである。Alice という前身のプロジェクトが開発されていたが、以下のような問題があった。データを送受信する API はインスタンスを生成して関数を呼び出す様に設計されているが、外部 Class からも関数が実行できてしまうため、受信する際どの key に紐づいたデータを受け取るのか、直感的にわからない。データを管理している localDataSegment がシングルトンで設計されており、Local で接続を行う際に複数のアプリケーションで立ち上げる必要がある。データを受け取る際に Object 型で受け取っているため、送信元を辿らない限り何の型が送信されているか不明瞭である。

それらの問題点を解消するために Alice を再設計し、作成されたものが Christie である。Christie では Alice の機能や概念を維持しつつ、Alice で発生していた問題点やプログラムを煩雑さなどを解消している。

2.1　Christie の基礎概念

Chrisite はタスクとデータを細かい単位に分割してそれぞれの依存関係を記述し、タスク実行に必要な入力が揃った順から並列実行するというプログラミング手法を用いている。

将来的に当研究室で開発している GearsOS のファイルシステムに組み込まれる予定があるため、GearsOS を構成する言語 Continuation based C と似た概念を持っている。Christie に存在する概念として以下の様なものがある。

- CodeGear
- DataGear
- CodeGearManager (以下 CGM)
- DataGearManager (以下 DGM)

図 2.1 はそれぞれの Gear の関係性を図示したものである。CodeGear はクラスやスレッドに相当する。DataGear は変数データに相当し、CodeGear 内で Java の annotation の

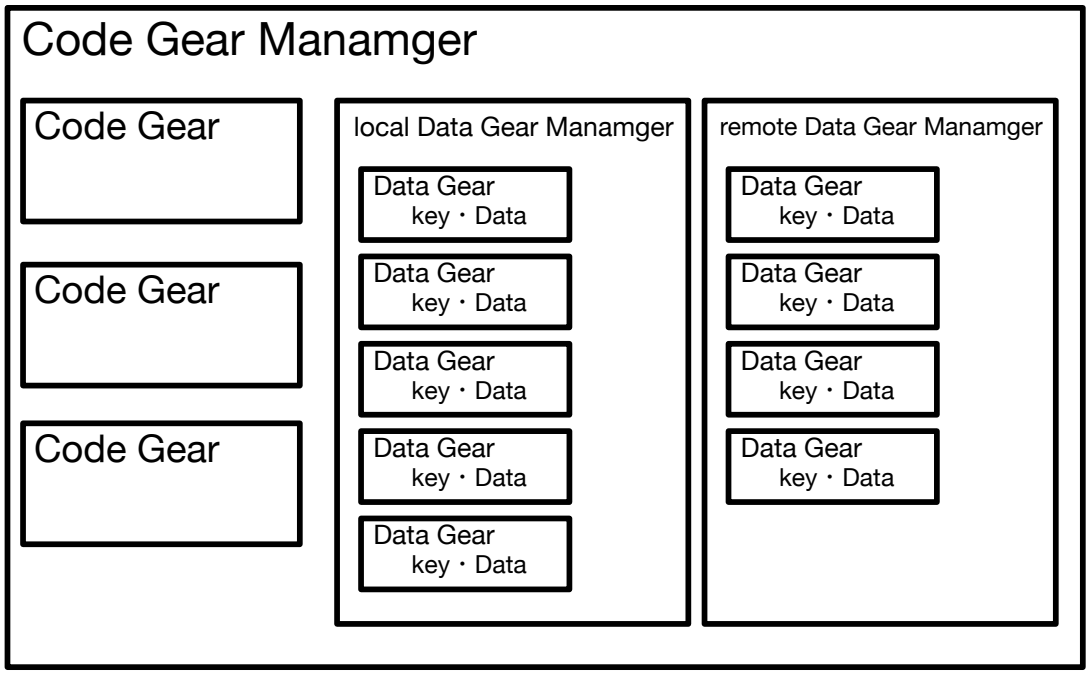

図 2.1: 各 Gear の関係性

機能を用いて key に紐づいた変数データを取得できる。CodeGear 内に記述した全ての DataGear にデータが格納された際に、初めてその CodeGear が実行されるという仕組みになっている。

CGM はノードに相当し、CodeGear、DataGear、DGM を管理している。DGM は DataGear を管理しており、put という操作によって変数データ、つまり DataGear を DGM に格納できる。DGM の put 操作を行う際には Local と Remote のどちらかを選び、変数の key とデータを引数として渡す。Local であれば Local の CGM が管理している DGM に対して DataGear を格納していく。Remote であれば、接続した Remote 先の CGM が管理している DGM に DataGear を格納する。

図 2.2 は、Christie を同一プロセスで複数のインスタンスを生成した際の DGM や CGM の接続構造を示している。全ての CGM は ThreadPool と他の CGM を List として共有している。ThreadPool とは CPU に合わせた並列度で queue に入った Thread を順次実行していく実行機構である。ThreadPool が増えると、CPU コア数に合わない量の Thread を管理することになり並列度が下がるため、1 つの ThreadPool で全ての CGM を管理している。また CGM の List を共有することでメタレベルで全ての CodeGear/DataGear にアクセス可能となっている。CG を記述する際は CodeGear.class を継承する。CodeGear は Runnable インターフェースを持っており、run メソッド内に処理を記述することでマルチ

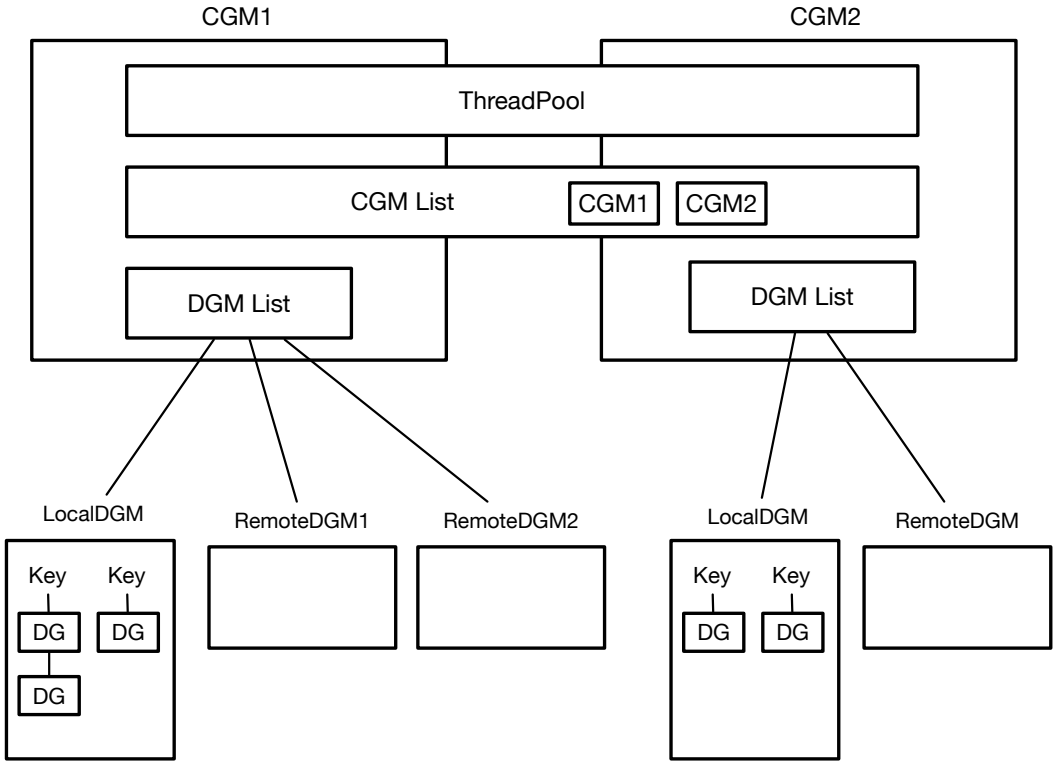

図 2.2: 同一プロセスでの Christie の複数インスタンス立ち上げ

スレッドで処理が行われる。CG に記述した key と一致する DG が全て揃った時、run に記述された処理が実行される。run メソッドの引数に CGM を指定しており、その CGM を経由して Christie の API にアクセスする。GearsOS では CG 間で Context を受け渡すことによって CG は DG にアクセスを行なっているため、Christie でもそのアクセス方法が採用されている。

通常の Runnable クラスでは引数を受け取ることができないが、CodeGearExecutor という Runnable の Meta Computation を挟んだことで CGM を受け渡しながらの記述を可能にしている。

DGM に対して put 操作を行うことで DGM 内の queue に DG を保管できる。DG を取り出す際には、CG 内で宣言した変数データに annotation を付ける。

2.2 annotation を使用したデータの記述

Christie では DG の指定に annotation を使用する。 annotation とはクラスやメソッド、パッケージに対して付加情報を記述できる Java の Meta Computation である。先頭に@を付けることで記述でき、オリジナルの annotation を定義することもできる。Christie の annotation には以下の 4 種類がある。

Take 先頭の DG を読み込み、その DG を削除する。

Peek 先頭の DG を読み込むが、DG は削除されない。そのため、特に操作をしない場合には同じ DG を参照し続ける。

TakeFrom(Remote DGM name) Take と処理動作は同じであるが、Remote DGM name を指定することで、その接続先 (Remote) の DGM から Take 操作を行うことができる。

PeekFrom(Remote DGM name) Peek と処理動作は同じであるが、 Remote DGM name を指定することで、その接続先 (Remote) の DGM から Peek 操作を行うことができる。

DG の宣言には型と変数を直接宣言し、変数名として key を記述する。 そして、その宣言の上に annotation で Take または Peek を指定する (ソースコード 2.1)。

```
1 @Take
2 public String data;
```

ソースコード 2.1: Take の例

annotation で指定した DG は CG を生成した際に CodeGear.class 内で待ち合わせの処理が行われる。これには Java の reflectionAPI を利用しており、annotation と同時に変数名も取得できるため、面数名による key の指定が可能になっている。

Christie の annotation はフィールドに対してのみ記述可能であるため、key の指定と Take/Peek の指定を必ず一箇所で書くことが明確に決まっている。これより、Alice の様に外の CG から key やデータへの干渉をされることがない。さらに key を変数名にしたことで、動的な key の指定や key と変数名の不一致による可読性の低下を防ぐことが可能となった。

リモートノードに対して Take/Peek をする際には、TakeFrom/PeekFrom annotation を用いる (ソースコード 2.2)。

```
1 @TakeFrom("remote_name")
2 public String data;
```

ソースコード 2.2: TakeFrom の例

2.3 データの型整合性

Alice では送受信するデータは Receive 型という専用の型を使用していた。内部のデータ自体は object 型だったため、元データの型を知るためには送信元を確認する必要があった。Christie では annotation で DG の宣言を行っているため、直接変数の型を宣言することが可能となっている。ソースコード 2.3 は DG のデータを扱う例である。

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

ソースコード 2.3: DG のデータを扱う例

DG として宣言した変数の型は、Java の reflectionAPI を用いて CGM で保存され、Local や Remote ノードと通信する際に適切な変換が行われれる。この様にプログラマが指定せずとも正しい型でデータを取得できるため、プログラマの負担を減らし信頼性を保証することができる。

2.4 CodeGear の記述方法

以下のソースコード 2.4、2.5、2.6 は LocalDGM に put した DG を取り出して表示し、インクリメントして 10 回繰り返す例題である。

```
1  public class StartCGExample extends StartCodeGear {
2      public StartCGExample(CodeGearManager cgm) { super(cgm); }
3
4      public static void main(String args[]) {
5          StartCGExample start = StartCGExample(createCGM(10001));
6      }
7
8      @Override
9      protected void run(CodeGearManager cgm) {
10         cgm.setup(new CodeGearExample());
11         CountObject count = new CountObject(1);
12         put("count", count);
13     }
14 }
```

ソースコード 2.4: StartCodeGear の記述例

```
1  public class CodeGearExample extends CodeGear {
2      @Take
3      public CountObject count;
4
5      @Override
6      protected void run(CodeGearManager cgm) {
7          System.out.println(count);
8
9          if (count < 10) {
10             cgm.setup(new CodeGearExample());
11             count.number += 1;
12             put("count", count);
13         } else {
14             cgm.getLocalDGM().finish();
15         }
16     }
17 }
```

ソースコード 2.5: CodeGear の記述例

```
1  @Message
2  public class CountObject {
3      public int number;
4
5      public CountObject(int number) {
6          this.number = number;
7      }
8  }
```

ソースコード 2.6: DG として送信されるオブジェクトのクラス

Chrisite では、StartCodeGear(以下 StartCG) から処理を開始する。StartCG は StartCodeGear.java を継承することで記述可能である。

StartCG を記述する際には、createCGM メソッドにより CGM を生成する必要がある (ソースコード 2.4 5 行目)。createCGM の引数には remote ノードとソケット通信をする際に使用するポート番号を指定する。CGM を生成した際に LocalDGM や remote と通信を行うための Daemon も生成される。

CG に対して annotation から待ち合わせを実行する処理は setup メソッドが行う。そのためソースコード 2.4 の 6 行目、ソースコード 2.5 の 10 行目のように、CG のインスタンスを CGM の setup メソッドに渡す必要がある。そのためどこでも CG の待ち合わせを行うことができず、必ず CGM の生成を行う必要がある。その制約により、複雑になりがちな分散プログラミングのコードの可読性を高めている。実行された CG を再度実行する場合にも、CG のインスタンスを生成して setup メソッドに渡す。

2.5 DataGearManager の複数立ち上げ

Alice では LocalDGM と同じ機能の LocalDataSegmentManagaer(以下 LocalDSM) が static で実装されていたため、複数の LocalDSM を立ち上げることができなかった。しかし Christie では CGM の生成に伴い LocalDGM も生成されるため、複数作成が可能である。複数の LocalDGM 同士のやり取りも、Remote への接続と同じ様に相手を RemoteDGM として proxy として立ち上げることでアクセス可能である (図 2.3)。

ソースコード 2.7 は LocalDGM を 2 つ立ち上げ、お互いを remote に見立てて通信する例である。6 行目にあるように、RemoteDGM を立ち上げるには CGM が持つ createRemoteDGM メソッドを用いる。引数には RemoteDGM 名と接続する remote ノード host 名、ポート番号を指定している。

```
1  public class StartRemoteTake {
2      public StartRemoteTake(CodeGearManager cgm) { super(cgm); }
3
4      public static void main(String args[]) {
5          CodeGearManager cgm1 = createCGM(10001);
6          cgm1.createRemoteDGM("cgm2", "localhost", 10002);
7          cgm1.setup(new RemoteTakeTest());
8
9          CodeGearManager cgm2 = createCGM(10002);
10         cgm2.createRemoteDGM("cgm1", "localhost", 10001);
11         cgm2.setpu(new RemoteTakeTest());
12     }
13 }
```

ソースコード 2.7: LocalDGM を 2 つ作る例

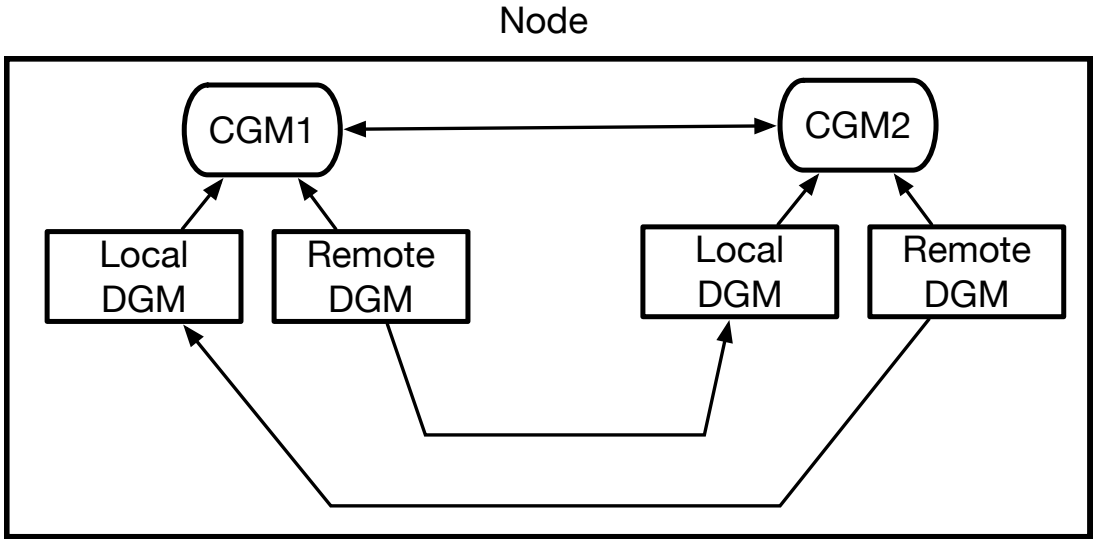

図 2.3: RemoteDGM を介した CGM 間の通信

remote への接続と同じ様にアクセスが可能になっており、コードを変更せずに同一マシン上の 1 つのアプリケーション内で分散アプリケーションのテストが可能となっている。

また、CGM は内部に同一プロセス全ての CGM リストを static で持っており、複数生成した CGM を全て管理している。つまり、メタレベルでは RemoteDGM を介さずに各 LocalDGM に相互アクセス可能である。

2.6 通信フロー

本章で説明した Christie の設計をいくつか例を挙げて Christie の通信のフローをシーケンス図を用いて解説する。図 2.4 は LocalDGM に Take を行い、LocalDGM 内に DG があったときの処理の流れである。

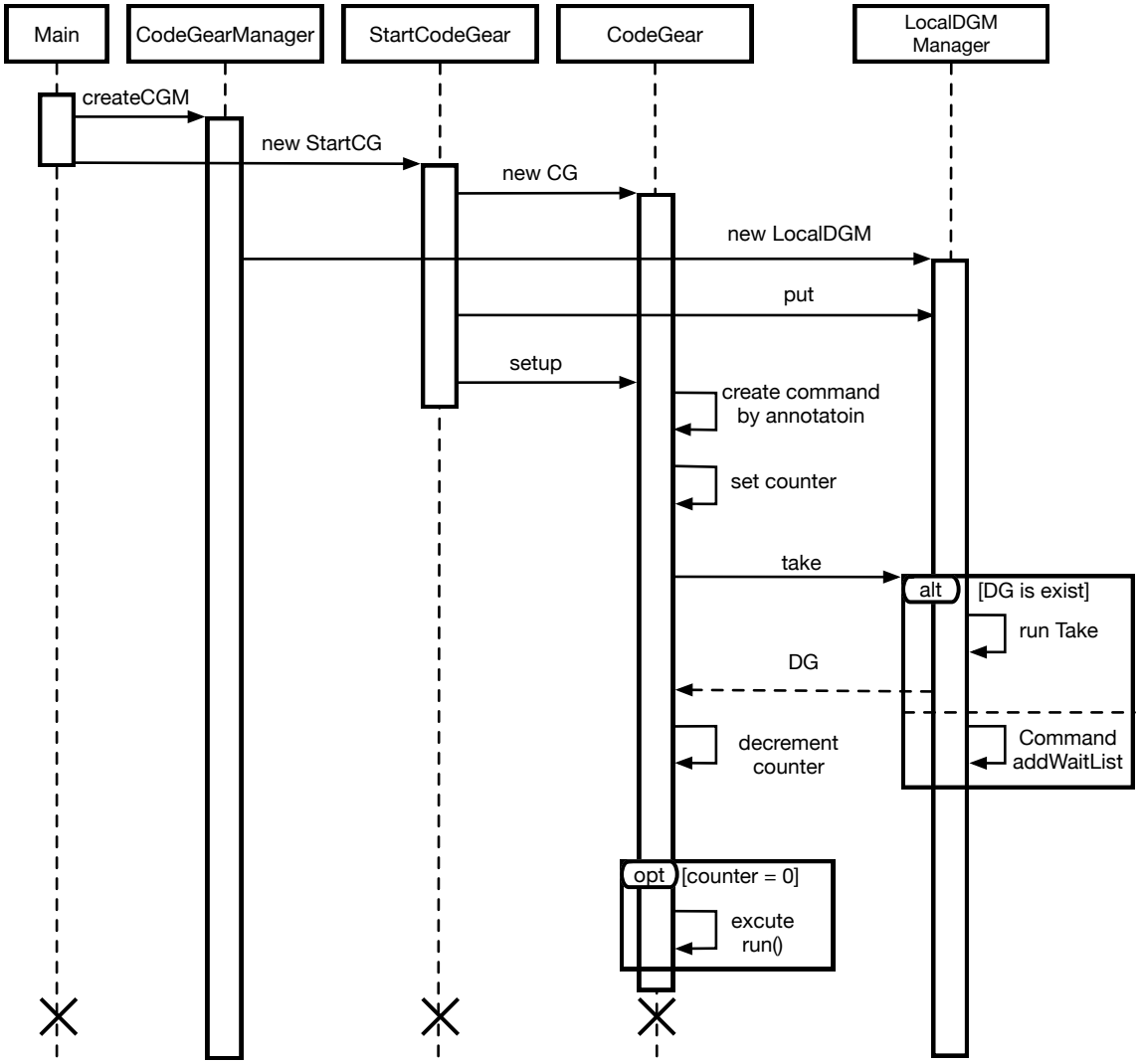

図 2.4: LocalDGM に Take した際のフロー

プログラマは main で CGM を生成する。CGM と同時に LocalDGM が作成される。続いて StartCG 内で CG のインスタンスを作成し、setup メソッドが呼ばれると、DG に付

与された annotation から Take コマンドが作られ実行される。CG は生成したコマンドの総数を初期値としたカウンタを持っており、コマンドが解決される (DG が揃う) 度にカウンタは減少する。カウンタが 0 になると待ち合わせが完了したとなり、run 内の処理が ThreadPool へ送られる。

図 2.5 は、LocalDGM に Take を行うが、LocalDGM 内に DG がなかったために Put の待ち合わせを行うときの処理の流れである。main などの最初の処理は図 2.4 と同様のため省略する。

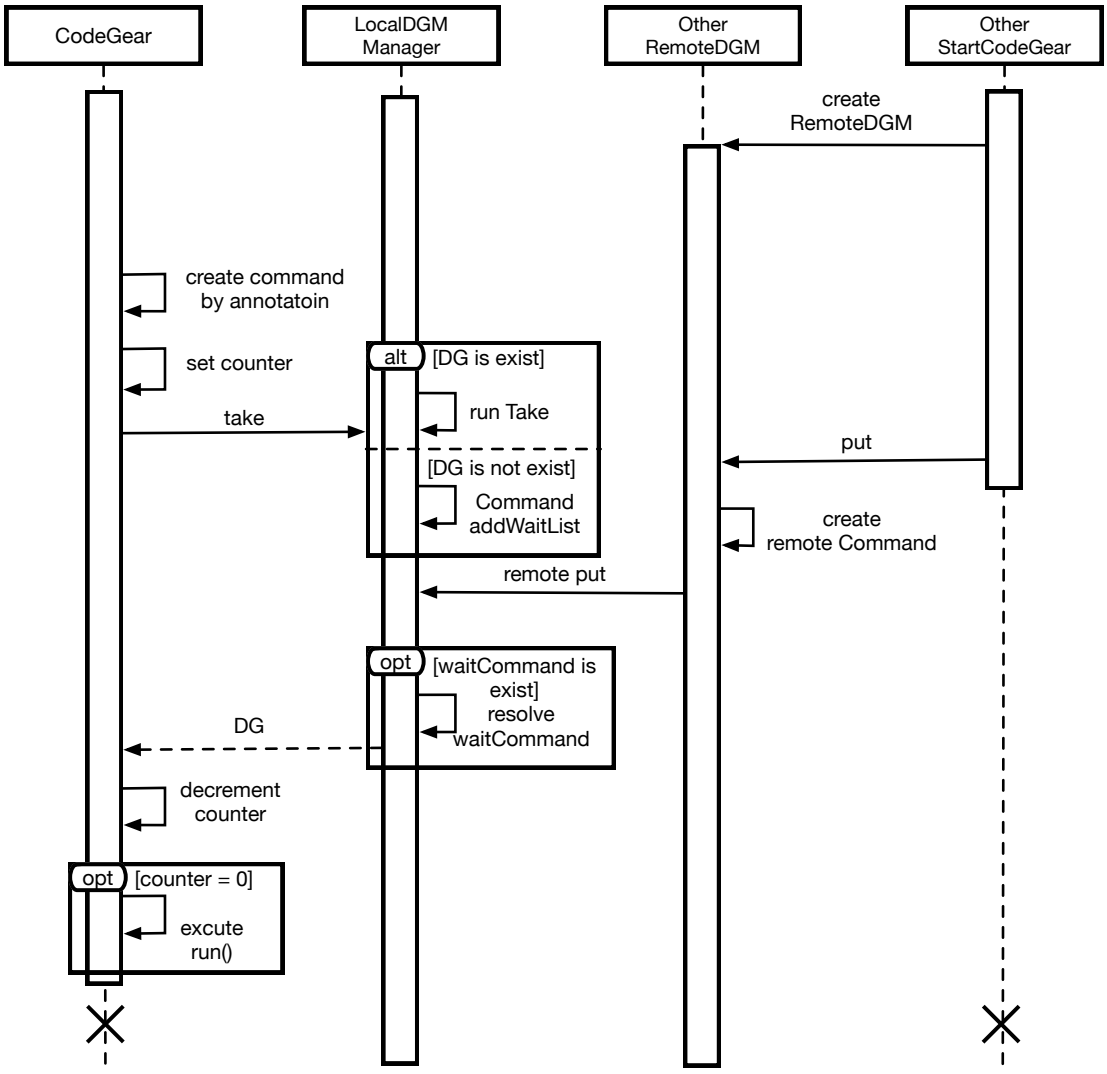

図 2.5: RemoteDGM から Put された際のフロー

Local または Remote ノードから Put コマンドが実行された際に waitList を確認し、Put された DG を待っているコマンドが存在すれば、そのコマンドは実行される。

図 2.6 は RemoteDGM に Take を行った際の処理の流れである。

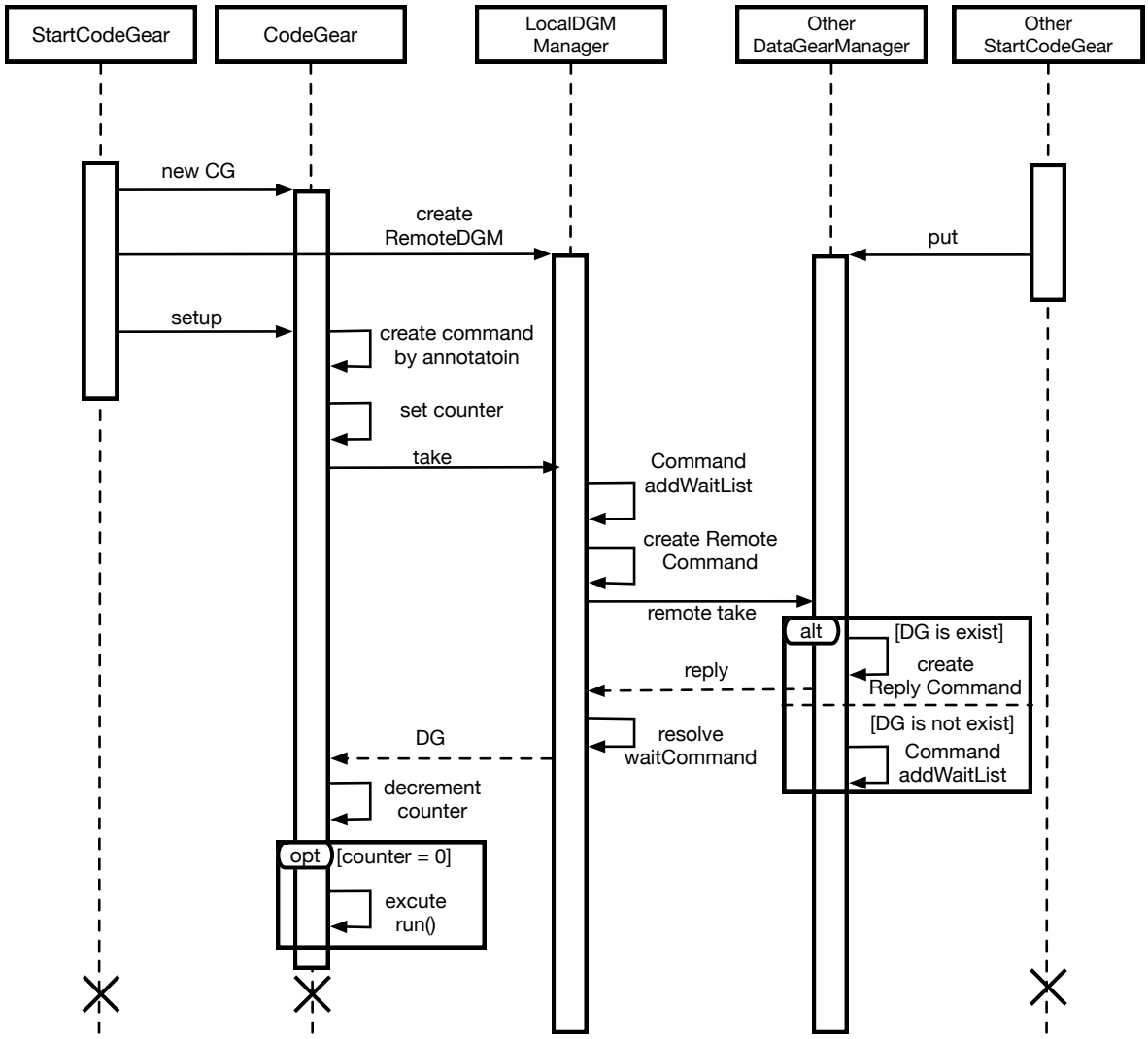

図 2.6: RemoteDGM に Take した際のフロー

プログラマは StartCG で事前に RemoteDGM を生成しておく。続いて、TakeFrom annotation から RemoteDGM に対する Take コマンドが生成され実行される。TakeFrom の様に Remote からの応答を待つコマンドは LocalDGM ではなく、RemoteDGM の waitList に格納される。RemoteDGM に対する Take コマンドは MessagePack 形式に変換さ

れ、RemoteDGM が参照している別ノードの LocalDGM に送信される。

送信された Take コマンドを受け取った LocalDGM は、要求された DG があれば Reply コマンドを生成して送り返す。もし DG がなければ、Remote から来たコマンドも Local の場合と同様に LocalDGM の waitList に格納される。

Reply コマンドを受け取ると RemoteDGMh は waitList に入っていたコマンドを解決し、待ち合わせが完了する。

第3章　まとめ

3.1　総括

3.2　今後の課題

3.2.1　hogehoge

謝辞

ホゲ様，フガ様ありがとうございます

参考文献

[1] Kaito TOKKMORI and Shinji KONO. Implementing continuation based language in llvm and clang. *LOLA 2015*, July 2015.

[2] Eugenio Moggi. Notions of computation and monads. *Inf. Comput.*, Vol. 93, No. 1, pp. 55–92, July 1991.

[3] Jean Yang and Chris Hawblitzel. Safe to the last instruction: Automated verification of a type-safe operating system. In *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '10, pp. 99–110, New York, NY, USA, 2010. ACM.

[4] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. sel4: Formal verification of an os kernel. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, SOSP '09, pp. 207–220, New York, NY, USA, 2009. ACM.

[5] Helgi Sigurbjarnarson, James Bornholt, Emina Torlak, and Xi Wang. Push-button verification of file systems via crash refinement. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'16, pp. 1–16, Berkeley, CA, USA, 2016. USENIX Association.