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

継続を使用する並列分散フレームワークの Unity 実装

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)　　名嘉村　盛和

印

(副　査)　　當間　愛晃

印

(副　査)　　河野　真治

要旨

オンラインゲームにおける通信方式は、大多数がクライアントサーバ方式である。分散プログラムを正しく書くことは難しく、増加するユーザに対応する必要がある。

本研究室で開発を行っている Christie は CodeGear と DataGear を用いた並列分散フレームワークである。Java で記述されているため直接ゲームエンジンの Unity で使用することができない。そこで、Christie の C#への再実装を行い、Unity 上で動作する通信ライブラリとして実装を行った。Christie Sharp を利用することで並列処理ライブラリを併用せずに済むため

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

図 目 次

表 目 次

ソースコード目次

第1章　オンラインゲームにおけるデータ通信

オンラインゲームは様々な回線を通じて複数のプレイヤーが関与する分散プログラムである。多くの場合通信形態は Cloud 上の Server を中心としたクライアントサーバ方式である。分散プログラムを正しく書くことは、Debug などを含め難しい。また、スマートフォンたタブレット端末の普及によりインターネット上で提供されているサービスやゲームに参加するユーザ数は増加傾向にある。増加するユーザに対応しつつ、正しく分散プログラムを書くことが求められている。

プログラマが 1 から分散プログラムを書くことは珍しくなっており、ほとんどの場合既存のフレームワークを利用して、分散システムの構築を行う。ゲーム開発においても、Unity や UE4 などで使用できるさまざまな通信ライブラリが存在している。これらの通信ライブラリはゲームエンジン向けに実装されているが、煩雑な記述方法や Debug の行いにくさがある。

当研究室では、初代 PlayStation 向けに作成した Federated Linda を拡張して、CodeGear/DataGear を用いた分散フレームワーク Christie を開発中である。Christie は Java で記述されているため、既存のゲームエンジン上で直接動かすことはできない。

Christie は従来の通信ライブラリとは異なり、型のある DataGear をタプル空間 DataGear-Manager(以下 DGM) に Key をもつストリームとして格納する方式を取っている。DGM は node 内部に local な DGM が用意されており、自身の node の Thread 間での通信に用いられている。他の node の DGM として proxy を持っており、対応した proxy にデータを書き込むことで node 間の通信を行っている。

DGM の構成には Topology Manager により自動的に構成される。node は初めに Topology Manager が動作している host と通信し、自分が接続すべき node が知らされる。node は指示された node に接続を行い、通信が行われる。

本研究では Christie を C#で再実装することにより Unity の通信ライブラリとして使用できるようにする。また、既存の通信ライブラリとの比較を行い、Christie のオンラインゲーム向けの通信ライブラリとしての考察を行う。

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

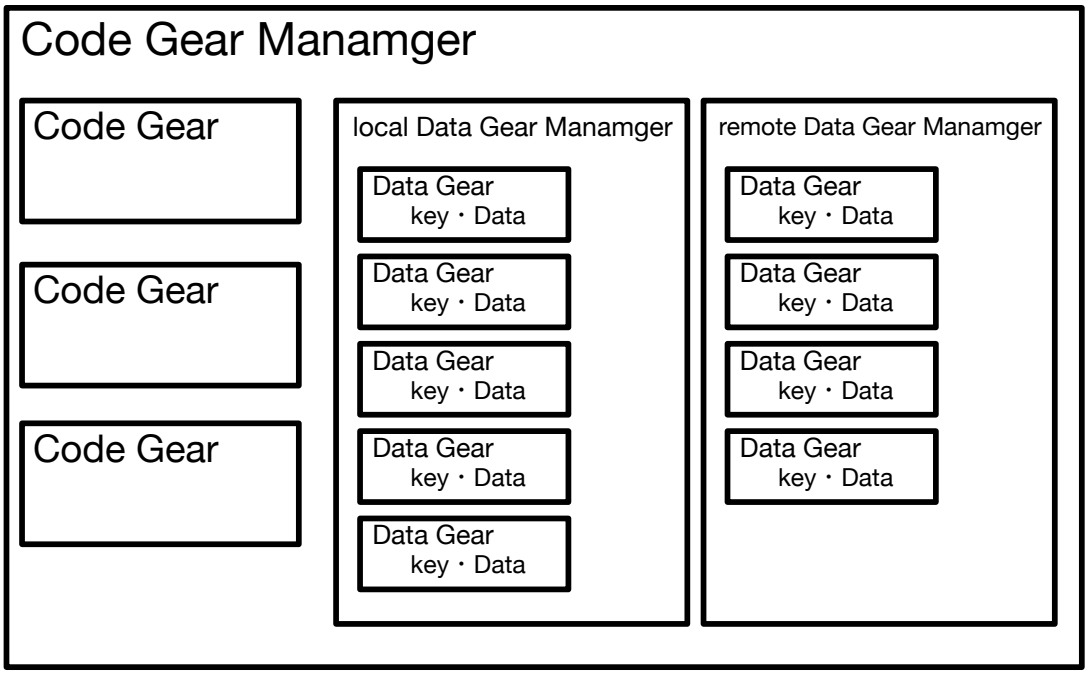

図 2.1: 各 Gear の関係性

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

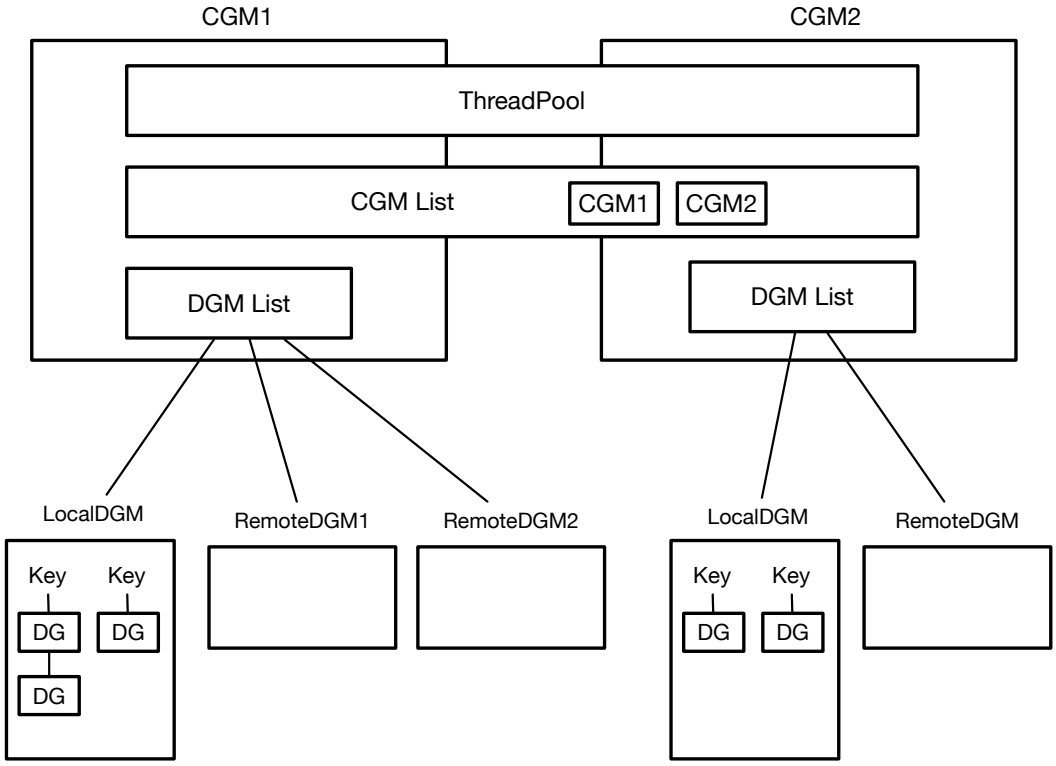

図 2.2: 同一プロセスでの Christie の複数インスタンス立ち上げ

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 Christie における継続

プログラムにおける継続とは、計算の途中を前後に分割し、分割した後の部分を first class object として扱うものである。つまり、計算途中などのどのような状況でもその object を取り出すことが可能であると言うことである。しかし継続を扱うためには Stack を含めた実行環境全体を heap にコピーする必要がある。

継続の一種に軽量継続がある。軽量継続は、Stack や環境のコピーを持っていないが、全ての必要な情報を関数の引数として持っている。必要な情報とは、計算に必要なデータや関数を実行した後に Jump するべき関数である。

Christie では、CodeGear と DataGear でプログラムを行っているが、内部では annotation で key を待ち、DataGearManager に key をつけて DataGear を渡している。key でつながった CodeGear に計算は接続されている。実行された CodeGear は data を put し、次の CodeGear を Setup してプログラムの処理が行われていく。これにより簡易的な継続が行われているといえる。

また、key でデータを渡すデータ構造に TreeMap を使用することがある。TreeMap は Stack のような働きをするが、TreeMap 自体を分散環境かで通信する場合に巨大なデータ構造を渡してしまうことになる。この方法では分散通信のパフォーマンスが低下してしまうと考えられるため、TreeMap の key 使用して Put/Take することで対応可能であると考える。その際、2nd key として接続先の hostname:port を指定する。こうするとで、データは proxy としてアクセスすることが可能となる。これより Christie は、名前付き継続と呼ぶことが可能な継続の一つを使用していると言える。

2.3 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

2.4 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。

2.5 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 StartCountup extends StartCodeGear {
2      public StartCountup(CodeGearManager cgm) { super(cgm); }
3
4      public static void main(String args[]) {
5          StartCountup start = StartCountup(createCGM(10001));
6      }
7
8      @Override
9      protected void run(CodeGearManager cgm) {
10         cgm.setup(new CountUpper());
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 CountUpper 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 CountUpper());
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。

2.6 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。

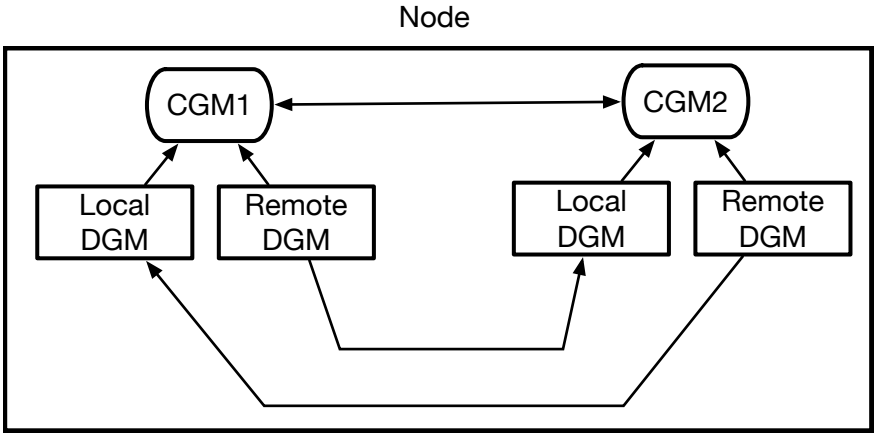

図 2.3: RemoteDGM を介した CGM 間の通信

```
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

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。

2.7 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。

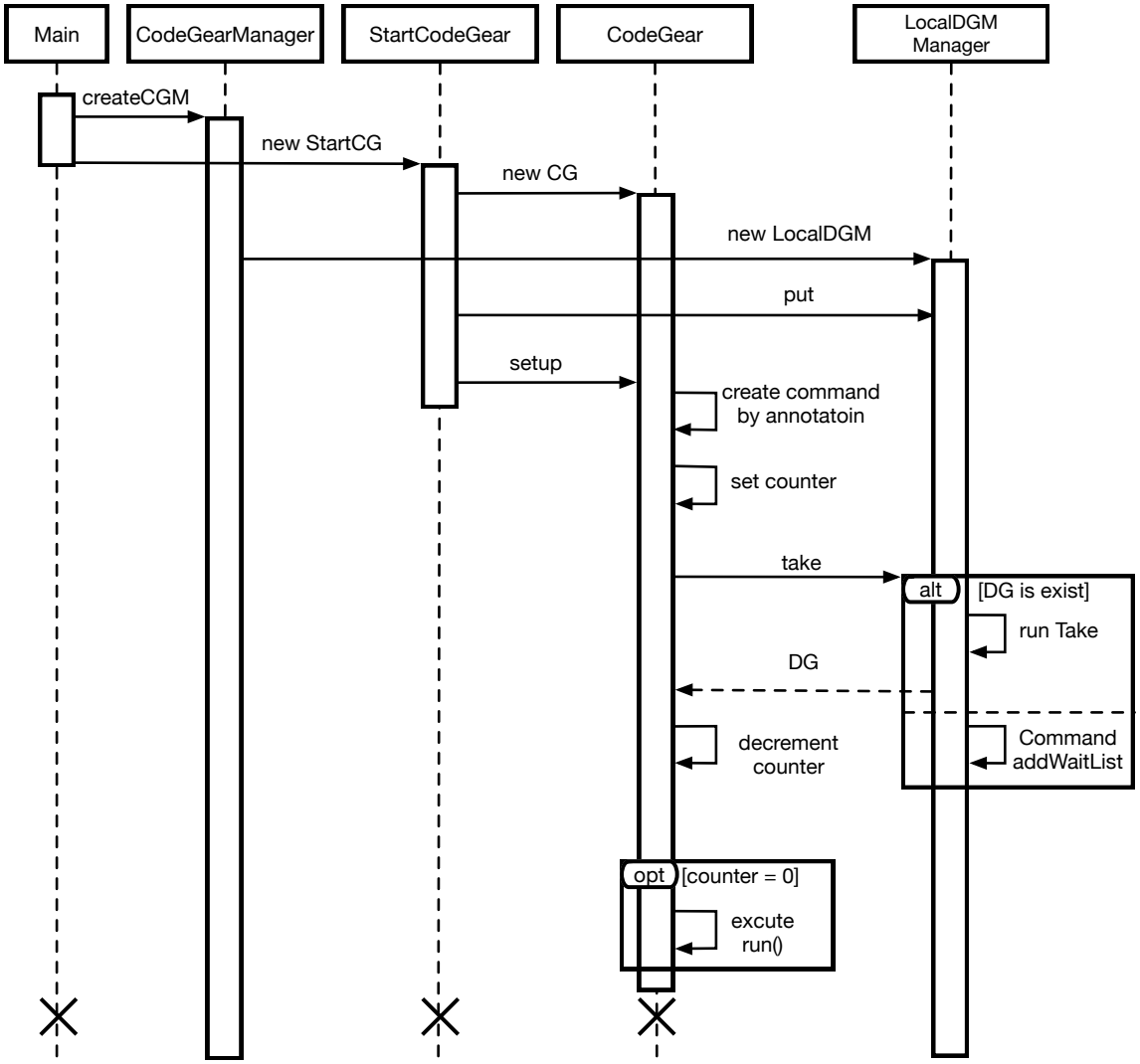

図 2.4: LocalDGM に Take した際のフロー

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。

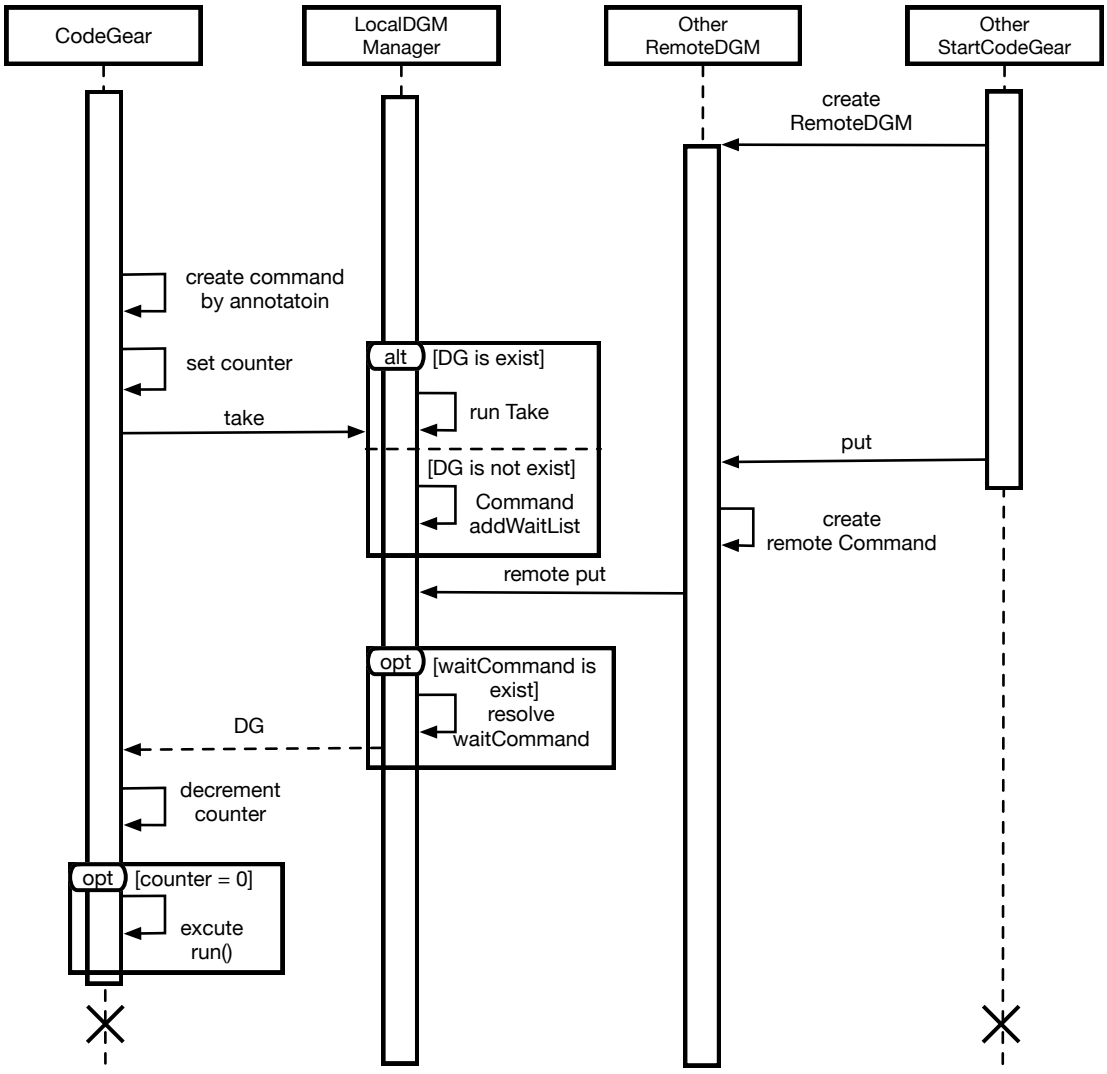

図 2.5: RemoteDGM から Put された際のフロー

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。

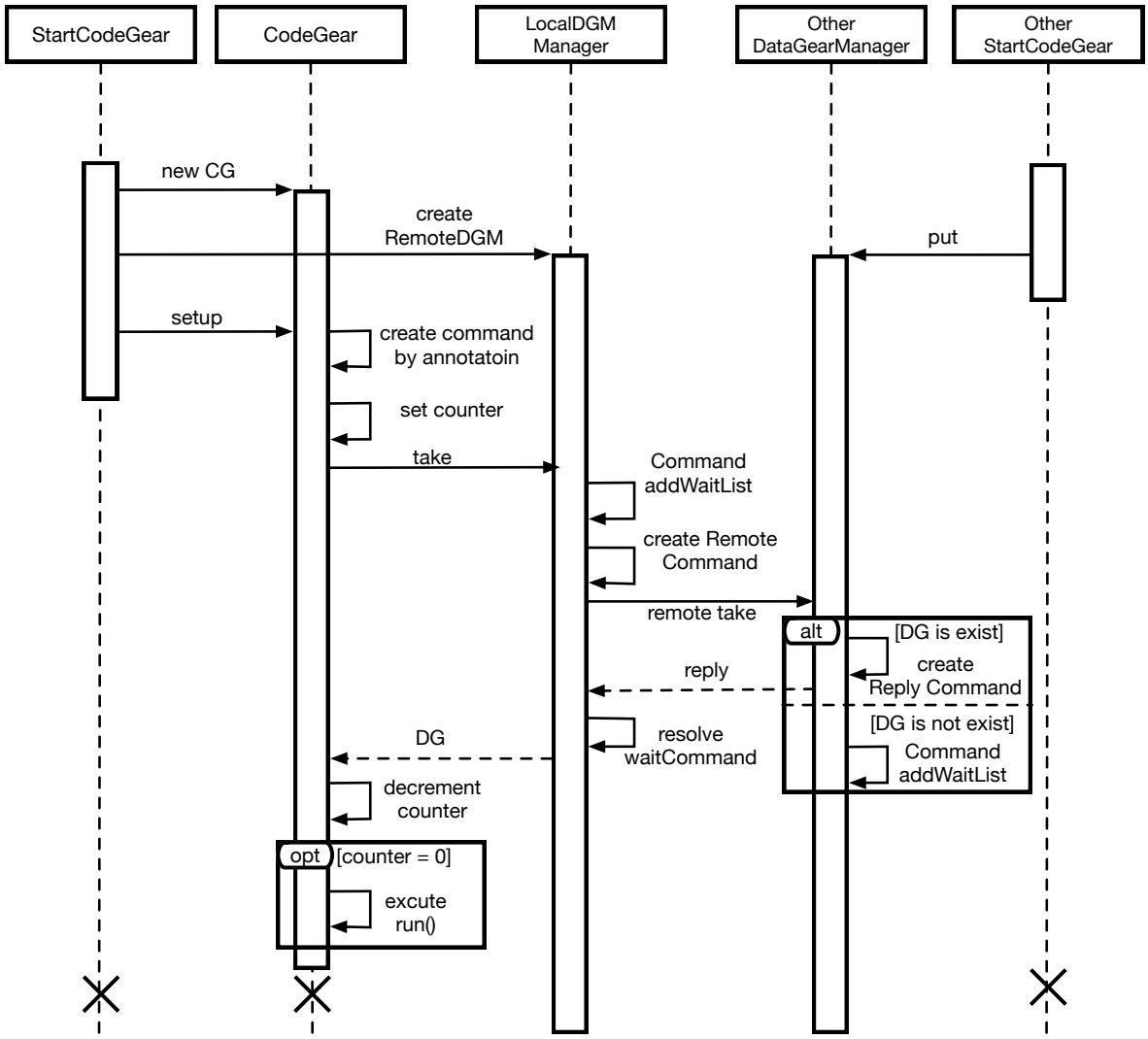

図 2.6: RemoteDGM に Take した際のフロー

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。

2.8 Topology Manager

Topology Manager とは、Christie 上での Network Topology を自動的に形成する機能である。Topology を形成するために参加を表明した node、TopologyNode に名前を与え、必要があれば node 同士の接続も自動で行う。Topology Manager の Topology の形成方法として、静的 Topology と動的 Topology の 2 つがある。

静的 Topology はソースコード 2.8 のような dot 形式のファイルを与えることで node の関係を図 2.7 のように構築できる。静的 Topology は dot ファイルの node 数と同等の TopologyNode があって初めて、CG が実行される。

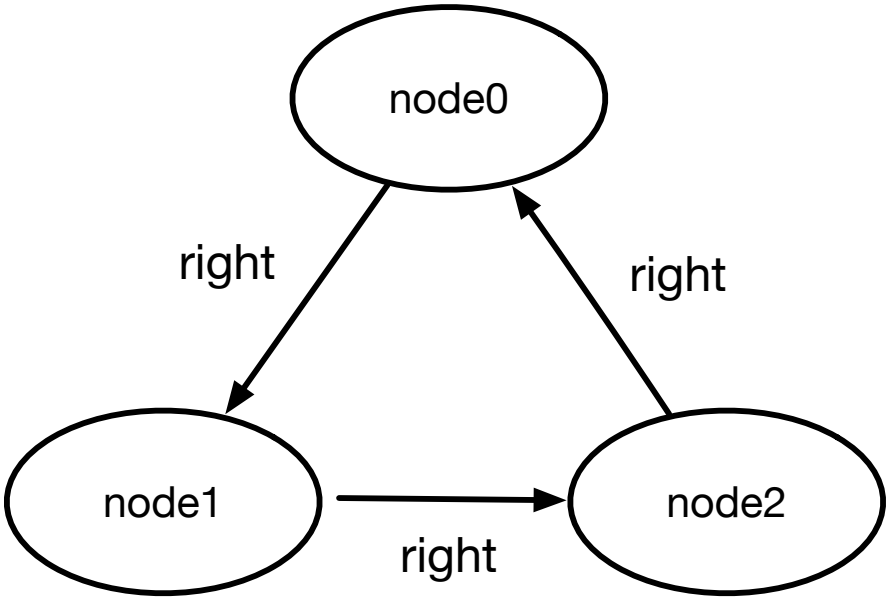

図 2.7: ソースコード **??**による ring 状の接続

```
digraph test {
    node0 -> node1 [label="right"]
    node1 -> node2 [label="right"]
    node2 -> node0 [label="right"]
}
```

ソースコード 2.8: dot 形式による node 間接続の記述

動的 Topology は図 2.8 のような手順で自動的に接続が行われる。node が参加表明を行うと、Topology を管理している Manager から Host message が node へ送信され、node は接続すべき親に接続を行う。親は Host から送信される node info より子 node との接続を行う。

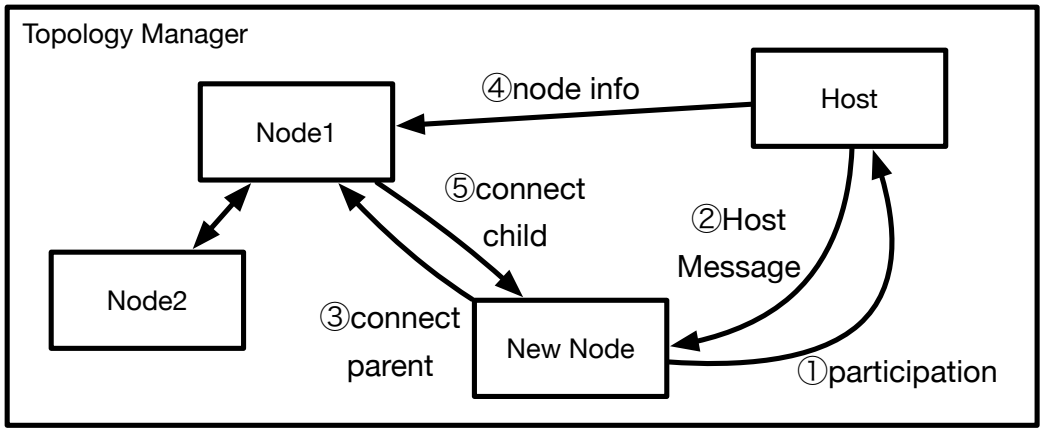

図 2.8: 動的 Topology の接続手順

現在は Tree 型にのみ対応しているが、Star 型への対応も可能である。

第3章　Christieの応用

3.1　Unity

Unity は Unity Technologies が開発、公開しているゲームエンジンである。画像や 3D モデルの表示、物理演算、UI のイベント機能などゲーム制作に必要な機能が標準で備わっており、個人でもゲーム開発が可能となっている。さまざまなプラットフォームに対応可能であり、PC、iOS、Android やその他コンシューマ機器も開発可能である。また、非常に動作が軽いことも特徴であり、スペックが低いノート PC でも十分ゲーム開発が可能である。

プログラミング言語としては C#がサポートされている。最新バージョンである 2021.2.8 では C# 9 がサポートされており、.NET Framework はバージョン 4.6 に対応している。C#向けの既存の API や外部ライブラリ、Unity 用に開発された API なども使用可能である。拡張性が高く、開発に必要な機能を作成し Unity のメニューから実行することも可能である。

本研究では、開発環境を整えるためのハードルの低さ、公開されている通信ライブラリの豊富さなどを考慮し Unity を採用した。

3.2　Photon Unity Networking 2

Photon Unity Networking 2(以下 PUN2) は Unity で利用可能なネットワークライブラリである。自動で他の Client への接続や同期を可能とし、マッチメイキング機能なども備わっている。

図 3.1 は PUN2 での Game Server までの接続の過程を表した図である。PUN2 ではサーバクライアント型の通信を行っており、Photon Cloud という Cloud Server に接続することで通信を可能にしている。Client は Photon Cloud に接続を行うと、始めに Name Server に接続される。Name Server では、その Client が利用可能なリージョンを提供し、最低 Ping の Master Server への接続が自動で行われる。Master Server では、ゲーム全体における接続しているプレイヤーや、作成されているルーム情報などの監視を行っている。マッチメイキングや、新しい Room の作成、参加などが可能である。各リージョンの Master Server は完全に分離しており、マッチメイキングはそれぞれの Master Server でのみ可能

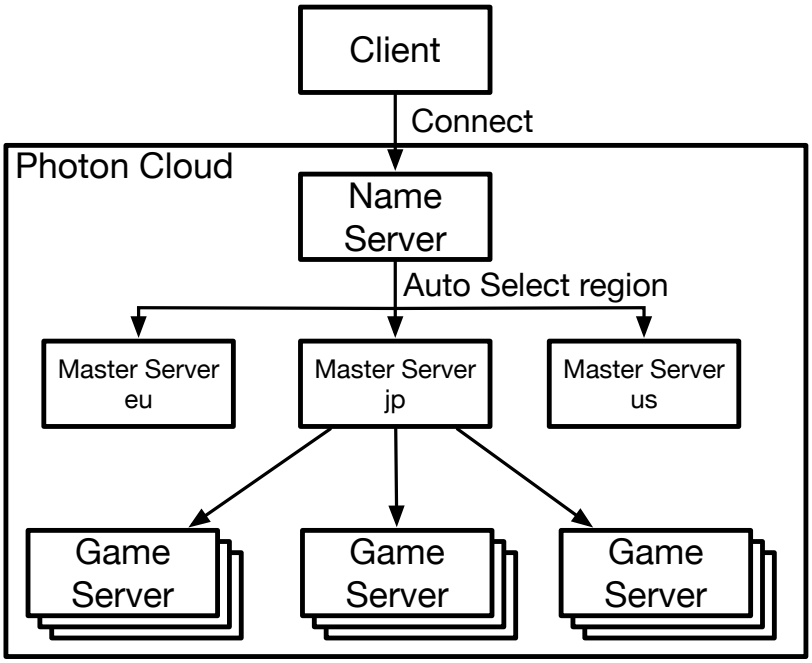

図 3.1: PUN2 の Server 接続

である。Game Server では、Master Server で作成された各ルームの管理が行われおり、実際に Client 同士の通信を行ってゲームプレイを可能としているのは Game Server である。

開発面においては Unity API 対応がされており、座標や回転データなどを持っている Transform、アニメーション、物理演算の Rigidbody の同期にはコードを書かずに用意されている Script を GameObject に Attatch するだけで同期可能となっている。

```
1  public class PhotonController : MonoBehaviourPunCallbacks {
2      void Start() {
3          PhotonNetwork.ConnectUsingSettings();
4      }
5
6      public override void OnConnectedToMaster() {
7          PhotonNetwork.JoinOrCreateRoom("Room", new RoomOptions(),
       TypedLobby.Default);
8      }
9
10     public override void OnJoinedRoom() {
11         PhotonNetwork.Instantiate("Charactor", Vector3.zero, Quaternion.
       identity);
12     }
13 }
```

ソースコード 3.1: PUN2 の使用例

ソースコード 3.1 は、PUN2 で Server に接続を行いゲーム内にプレイヤーを生成する例である。Callback の使用には MonoBehaviourPunCallbacks を継承する必要があり、MonoBehaviourPunCallbacks には Unity の MonoBehaviour を継承しているため、Start メソッドなどが使用可能である。Start メソッドでは、事前に設定した Name Server および Master Server への接続を行っている。OnConnectedToMaster メソッドは Master Server へ接続した際に呼ばれ、Room へ加入を試しなければ作成を行い、OnJoinedRoom メソッドでは GameServer にある Room に加入後に呼ばれ、プレイヤーの生成を行っている。このように PUN2 では各 Server への接続/切断時のコールバックが豊富に用意されており、柔軟に処理を記述することが可能である。

しかし開発時に検証を行う際に Local マシンで通信を行う場合などでも、Photon Cloud への接続が必須となる。そのためネット環境がなければ Local 通信でも PUN2 を利用した通信はできない。

PUN2 の基本料金は無料であるが、使用上の注意としていくつかの制限がある。

- 同時接続人数が 20 人以下
- 1Room の秒間 Message 数が 500 以下
- データの転送量が 60GB 以下
- etc ...

これらの制限を超えて利用する場合には、別途料金や有料プランに加入する必要が発生する。

3.3　Mirror

Mirror は Unity で使用できる OSS のネットワークライブラリである。PUN2 と同様に PC 間の同期を自動で行い、主に MMO 規模のネットワークを想定して開発が行われている。主な機能として、Server と Client を自動接続する Script、GameObject の位置やアニメーションの同期を行う NetworkTransfrom や NetworkAnimator、Room 機能などがある。

図 3.2 は Server と Client の接続を表したものである。Mirror はサーバクライアント型の通信が行われている。Host では Server と同時に Local Client が立ち上がる。他の Clinet は Server に接続を行うことで同期が可能となる。Clinet は Server に対して操作やデータを送信し、Server での処理が Client に同期され反映される仕組みになっている。

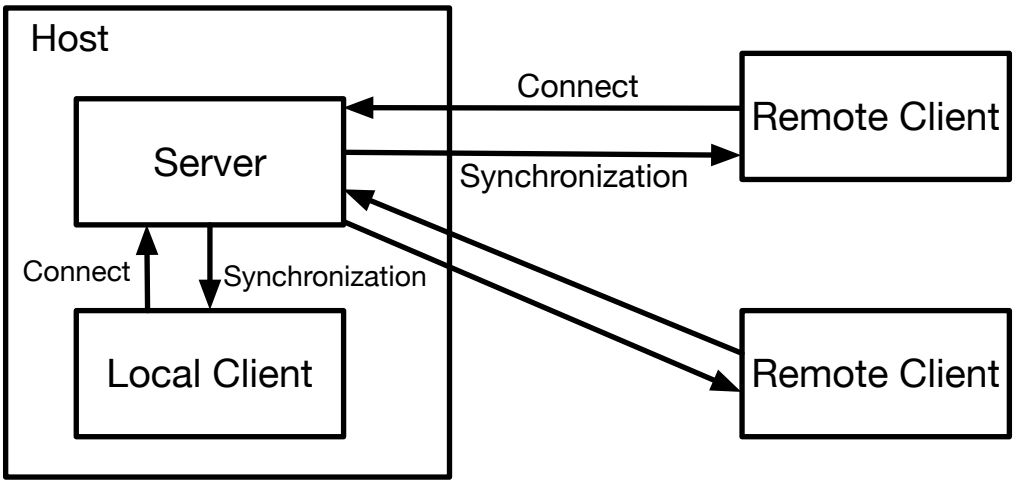

図 3.2: Mirror の接続

```
1  public class PlayerController : NetworkBehaviour {
2      [SyncVar] float speed = 1f;
3
4      public override void OnStartServer() {
5          speed = Random.Range(1f, 10f);
6      }
7
8      void FixedUpdate() {
9          if (isLocalPlayer) {
10             float x = Input.GetAxis("Horizontal");
11             float z = Input.GetAxis("Vertical");
12             CmdMoveSphere(x, z);
13         }
14     }
15
16     [Command]
17     void CmdMoveSphere(float x, float z) {
18         Vector3 v = new Vector3(x + speed, 0, z + speed);
19         GetComponent<Rigidbody>().AddForce(v);
20     }
21 }
```

ソースコード 3.2: Mirror の使用例

ソースコード 3.2 は Mirror を使用して、プレイヤーを表示、操作する例である。NetworkBehaviour は MonoBehavior を継承して作成されており、ネットワーク上で動作させる処理に継承させる必要がある。OnStertServer メソッドは Server でこの GameObject が生成された際に実行される。このような様々な callback が NetworkBehaviour には用意さ

れている。また、フィールド変数に SyncVar attribute を付与することで自動的に変数を同期可能にしている。SyncVar attribute によってサポートされている型は C#のプリミティブ型や string、Unity API から提供されている Vector3 や Quaternion、Mirror から提供されている SyncList や SyncDict などのリストや辞書である。

10 行目の Command attribute を関数に付与することで、Client が Server に対してこの関数を呼び出すことが可能になる。Server 側で処理が行われ、その結果の状態が Client へ同期される。

Mirror には上記のような通信フレームワークとして便利な機能が用意されており、OSS なためプロジェクトに合わせて自由に機能の拡張が可能である。一方で公式 Server などのサービスが用意されていないため、自前で Server を用意する必要がある。そのため、LAN 上で通信を行う場合には特に問題ないが、インターネット上に公開を手軽に行うことができない。

3.4　Christie を C#に書き換える意義

Christie は Java で実装されているが、Unity 開発は基本的に C#が使用される。ここで、Chrsite を Java のまま利用するか、C#に書き換えるかという疑問が生じた。

Unity は android の開発向けに Java で書かれたメソッドを C#で呼び出せる機能がある。Java や Java VM を呼び出すために Java Native Interface(JNI) を利用している。C#側で Java を呼び出すには、Unity API が提供しているヘルパークラスを呼び出す必要があるが、string を使用してリソースディレクトリから検索を行っているため、計算負荷が高い。そのため、Christie を Java から呼び出して使用するには不適合であると考える。

また、C#に書き換えた際にバージョン管理の問題がある。書き換えを行ったとして、Christie には Java と C#の 2 種類あることになり、機能の実装などそれぞれに対応する必要がある。しかし Java と C#それぞれの対応はそこまで難しくないと考える。Java の記法と C#の記法は非常によく似ており、API もほとんど同じ機能を持ったものがそれぞれ実装されている。そのため、Chrisite に機能追加をする際には Java と C#両方に対応することでバージョン間のすり合わせを行えると考える。

以上の理由により、Christie を Unity でサポートされている C#への書き換えを行う意義を見いだすことができた。

3.5　Christie Sharp の書き換えの基本方針

Java で記述された Christie と区別するため、C#で記述する Christie を Chrisite Sharp とする。Chrisite Sharp ではコードの保守性や、Christie 設計時の意図などを守るため、

Chrisite と同じ挙動、同じ動作をする必要がある。初めに C#単体で動作するように、Christie の核となる部分の書き換えを行った。

Christie は Java 9 から開発されていたため、現在では非推奨なコードやバージョンアップが必要な箇所が存在する。そこで書き換えを行う際に、C#に対応しつつ処理動作の向上や最適化を行うために以下の改良を行った。

- MessagePack の変更及びバージョンアップ
- ThreadPool から Task への変更

3.6　attribute の実装

Christie では DG を取得する際に、annotation を用いて Take や Peek などのコマンドを処理していた。Christie Sharp は annotation ではなく、代わりに attribute を利用してコマンドの処理を行っている。

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

ソースコード 3.3: Java における Take annotation の実装

```
1 [AttributeUnsage(AttributeTargets.Field)]
2 public class Take : Attribute {}
```

ソースコード 3.4: C#における Take attribute の実装

ソースコード 3.3 は Java における Take annotation の実装である。Java で annotation を自作する際には、interface で宣言を行う。1 行目では annotation の適用可能箇所を指定しておりフィールド変数に対して付与可能としている。また、2 行目は annotation の情報をどの段階まで保持するかを指定しており、Take の場合 JVM によって保存され、ランタイム環境で使用可能となっている。

ソースコード 3.4 は C#における Take attribute の実装である。C#で attribute を自作する際には、System.Attribute を継承する必要がある。attribute の適用可能箇所については、1 行目にてフィールド変数を指定している。

attribute の使用方法は annotation と同じく変数の宣言の前に、[ ] 内に使用する attribute を宣言する (ソースコード 3.5)。

```
1 [Take] public string data;
```

ソースコード 3.5: Take attribute の例

3.7　Task による CodeGear の処理

Chrisite では CG の実行に ThreadPool を利用していた。しかし ThreadPool は管理や生成が煩雑であり、コストパフォーマンスの低下につながりやすい。C#には Thread をより使いやすく高機能にした Task という機能がある。Task は C#の ThreadPool を拡張しており、内部に ThreadPool と実行待ち Queue を持っている。Task.Run メソッドや Task.Factory.StartNew メソッドで処理を実行でき、処理が渡されると ThreadPool で処理されるため、Christie と同じ動作をすると考え、Christie Sharp では Task で書き換えを行った。

```
1  public class PriorityThreadPoolExecutors {
2
3      public static ThreadPoolExecutor createThreadPool(int nThreads, int
       keepAliveTime) {
4          return new PriorityThreadPoolExecutor(nThreads, nThreads,
       keepAliveTime, TimeUnit.MILLISECONDS);
5      }
6      private static class PriorityThreadPoolExecutor extends
       ThreadPoolExecutor {
7          private static final int DEFAULT_PRIORITY = 0;
8          private static AtomicLong instanceCounter = new AtomicLong();
9
10         public PriorityThreadPoolExecutor(int corePoolSize, int
       maximumPoolSize,
11                                   int keepAliveTime, TimeUnit unit) {
12             super(corePoolSize, maximumPoolSize, keepAliveTime, unit, (
       BlockingQueue) new PriorityBlockingQueue<ComparableTask>(10,
13         ComparableTask.comparatorByPriorityAndSequentialOrder()));
14         }
15
16         @Override
17         public void execute(Runnable command) {
18             // If this is ugly then delegator pattern needed
19             if (command instanceof ComparableTask) //Already wrapped
20                 super.execute(command);
21             else {
22                 super.execute(newComparableRunnableFor(command));
23             }
24         }
25
26         private Runnable newComparableRunnableFor(Runnable runnable) {
27             return new ComparableRunnable((CodeGearExecutor) runnable);
28         }
29
30         @Override
31         protected <T> RunnableFuture<T> newTaskFor(Runnable runnable, T
       value) {
32             return new ComparableFutureTask<>((CodeGearExecutor)runnable,
        value);
33         }
```

```
34      }
35  }
```

ソースコード 3.6: Christie における ThreadPool の実装の一部

ソースコード 3.6 は Christie における CodeGear を処理する ThreadPool の実装の一部である。Java では独自に ThreadPool を実装する際には ThreadPoolExecutor を継承する。また Thread の優先度を変更する機能が実装されており、CodeGear 実行時に処理の優先度を設定することが可能となっている。

```
1  public class ThreadPoolExecutors {
2
3      public ThreadPoolExecutors() {
4          int nWorkerThreads;
5          int nIOThreads;
6          ThreadPool.GetMinThreads(out nWorkerThreads, out nIOThreads);
7          ThreadPool.SetMinThreads(nWorkerThreads, nIOThreads);
8      }
9
10     public ThreadPoolExecutors(int nWorkerThreads, int nIOThreads) {
11         ThreadPool.SetMinThreads(nWorkerThreads, nIOThreads);
12     }
13
14     public void Execute(CodeGearExecutor command) {
15         Task.Factory.StartNew(() => command.Run());
16     }
17 }
```

ソースコード 3.7: Christie Sharp における ThreadPool の実装

ソースコード 3.7 はソースコード 3.6 を C#に書き換えを行ったものである。CG の実行には 14 行目の Execute を呼び出し、Task で実行を行っている。

Thread の優先度による実行順変更については、実装の優先度が低かったため今回は実装を行っていない。Task のスケジューラーは自作可能であり、実行待ち Queue の処理順を変更することができるため実装可能である。

3.8　Socket 通信用の Thread を Task に変更

Christie では、Socket 通信を行っている箇所も Thread を使用した MultiThread で動作している。こちらも可読性や保守性のため Task への書き換えを行った。

```
1  public void run() {
2    while (true) {
3      try {
4        Socket socket = null;
5          socket = ss.accept();
6          socket.setTcpNoDelay(true);
```

```
7           System.out.println("Accept " + socket.getInetAddress().
    getHostName() + ":" + socket.getPort());
8           Connection connection = new Connection(socket, cgm);
9           String key = "accept" + counter;
10          IncomingTcpConnection in = new IncomingTcpConnection(connection);
11          in.setName(connection.getInfoString()+"-IncomingTcp");
12          in.start();
13          cgm.setAccept(key, in);
14          OutboundTcpConnection out = new OutboundTcpConnection(connection)
    ;
15          out.setName(connection.getInfoString()+"-OutboundTcp");
16          out.start();
17          counter++;
18      } catch (IOException e) {
19          e.printStackTrace();
20      }
21    }
22  }
```

ソースコード 3.8: Christie における AcceptThread の実装の一部

ソースコード 3.8 は Christie における Socket 通信を行っている Thread の実行箇所である。無限ループを行い、他 node が Socket で接続される度にデータ送受信専用の Thread を作成する。IncomingTCPConnection ではデータの受け取りを行っており、OutboundTCP-Connection では Socket 通信が終了する際のコマンドの送信をそれぞれ MultiThread で行っている。

```
1  public void Run() {
2    while (true) {
3      try {
4        TcpClient client = null;
5        client = listener.AcceptTcpClient();
6        client.NoDelay = true;
7
8        IPEndPoint endPoint = (IPEndPoint)client.Client.RemoteEndPoint;
9        IPAddress ipAddress = endPoint.Address;
10       IPHostEntry hostEntry = Dns.GetHostEntry(ipAddress);
11       Console.WriteLine("Accept " + hostEntry.HostName + ":" + endPoint.
     Port);
12
13       Connection connection = new Connection(client.Client, cgm);
14       Console.WriteLine("connection:" + connection.GetInfoString());
15       string key = "accept" + counter;
16
17       IncomingTcpConnection incoming = new IncomingTcpConnection(
     connection);
18       Task.Factory.StartNew(() => incoming.Run());
19
20        cgm.SetAccept(key, incoming);
21
22        OutboundTcpConnection outbound = new OutboundTcpConnection(
     connection);
```

```
23          Task.Factory.StartNew(() => outbound.Run());
24          counter++;
25      }
26      catch (Exception e) {
27        Console.WriteLine(e.StackTrace);
28      }
29    }
30  }
```

ソースコード 3.9: Christie Sharp における AcceptThread の実装の一部

ソースコード 3.9 はソースコード 3.8 を C#に書き換えを行ったものである。Christie では、MultiThread で Socket 通信を行っていた。Christie Sharp では CodeGear の処理で使用していた ThreadPool と同様に Task への書き換えを行った。

3.9　MessagePack の変更

Christie ではデータを他 Node に送信する際に、MessagePack を使用してデータを Serialize し、送信を行っている。Chrisite で使用している MessagePack は msgpack java 0.6.12 を使用しており、現在はサポートされていない。そのため Java でサポート対象となっている msgpack java 0.7.x 以上の MEssagePakck とは記述方法が異なっている。ソースコード 3.10 は Christie で使用してる msgpack java 0.6.12 の使用例である。

```
1  public class MessagePackExample {
2      @Message // Annotation
3      public static class MyMessage {
4          // public fields are serialized.
5          public String name;
6          public double version;
7      }
8
9      public static void main(String[] args) throws Exception {
10         MyMessage src = new MyMessage();
11         src.name = "msgpack";
12         src.version = 0.6;
13
14         MessagePack msgpack = new MessagePack();
15         // Serialize
16         byte[] bytes = msgpack.write(src);
17         // Deserialize
18         MyMessage dst = msgpack.read(bytes, MyMessage.class);
19     }
20 }
```

ソースコード 3.10: Java における MessagePack の使用例

MessagePack を使用するには、Serialize を行うクラスに対して明示的に@Message annotation を付ける必要がある。これにより、クラス内で宣言した public 変数がエンコード

の対象となる。ソースコード 3.10 の 14 - 18 行目は Serialize/Deserialize を行う例である。MessagePack インスタンスを作成後、write メソッドを使用することで引数に渡したオブジェクトを byte[] 型に Serialize できる。Deserialize には read メソッドを使用し、引数として Serialize された byte[] 型と Deserialize 対象のクラスを渡すことでデコードできる。

C#の MessagePack は複数存在しており、msgpack java 0.6.12 とほぼ同様の記述方法を採っている MessagePack CSharp 2.3.85 を選択した。

```
1  [MessagePackObject]
2  public class MyClass {
3      [Key(0)]
4      public int Age { get; set; }
5      [Key(1)]
6      public string FirstName { get; set; }
7      [Key(2)]
8      public string LastName { get; set; }
9
10     static void Main(string[] args) {
11         var mc = new MyClass {
12             Age = 99,
13             FirstName = "hoge",
14             LastName = "huga",
15         };
16
17         // Call Serialize/Deserialize, that's all.
18         byte[] bytes = MessagePackSerializer.Serialize(mc);
19         MyClass mc2 = MessagePackSerializer.Deserialize<MyClass>(bytes);
20
21         // [99,"hoge","huga"]
22         var json = MessagePackSerializer.ConvertToJson(bytes);
23         Console.WriteLine(json);
24     }
25 }
```

ソースコード 3.11: C#における MessagePack の使用例

MessagePack CSharp では msgpack java と同様にクラスに対して Serialize を行うため、3.11 の 1 行目で MessagePackObject attribute を追加している。また、Serialize する変数に対して key を設定することができ、indexes としての int や string を key として指定することができる。

データの Serialize には MessagePackSerializer.Serialize メソッドを使用し、引数として渡したオブジェクトを byte[] 型に Serialize する。Deserialize には MessagePackSerializer.Deserialize メソッドを使用する。Deserialize メソッドはジェネリスク関数であるため、<> 内に Deserialize 対象のクラスを指定する。ソースコード 3.11 の 22 行目では json 展開の例であり、変数それぞれに key を指定していることで展開可能となっている。

3.10　送信パケットの修正

MessagePack のバージョンを更新した影響により、Remote node にデータを送信するパケットの形式を変更する必要がある。図 3.3 は Christie と、Chrisite Sharp における送信パケットの構成である。msgpack java では read メソッドの引数に Class<T> を渡すことでデコード可能であり、RemoteMessage の Deserialize にデータ長を指定する必要はない。DG はジェネリスクで記述しており毎回デコードするクラスが異なるため、デコードの際にデータ長を必要としている。

しかし MessagePack CSharp では、Deserialize メソッドに Class<T> を渡してデコードすることができないため、データ長も付属させてデータを送信する。それぞれの Size のデータ長は int で指定しているが、MessagePack で Serialize した際に最大で 5byte になる。

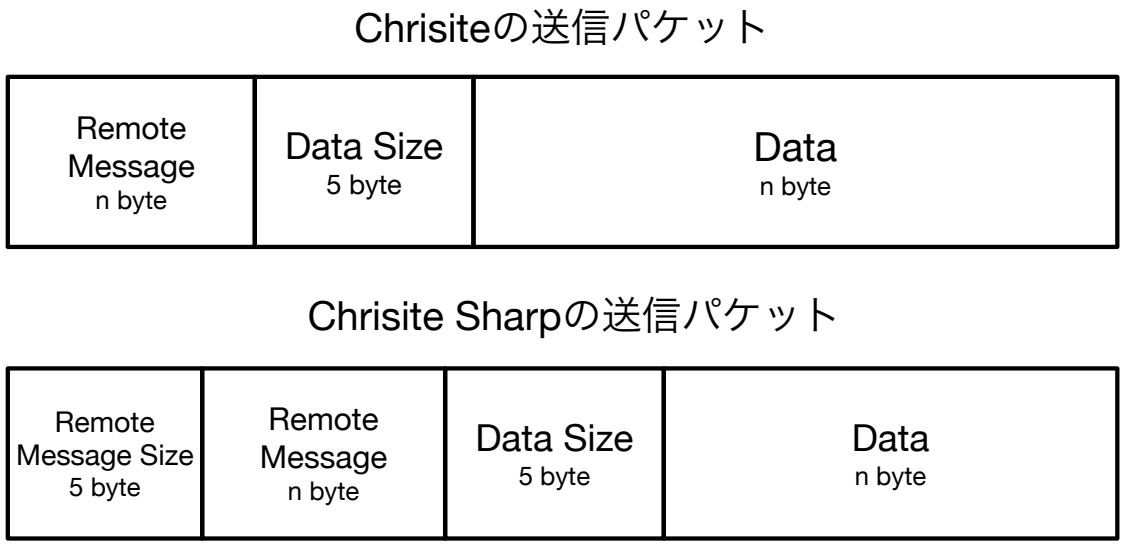

図 3.3: 送信パケットの構成

3.11　Chrisite Sharp の Debug

Christie Sharp の開発には JetBrains が開発・提供している C#向けの IDE、Rider を用いている。Christie Sharp は並列分散プログラミングを行っているため、MultiThread での Debug が必須である。Rider には Debuger の標準的な機能が搭載されているが、Christie Shap を開発する際、MainThread 以外で処理をする箇所に対して張った Break point でプロセスが停止しない状況に陥った。解決方法がないか調査を行ったが原因は不明であり、

MainThread に Break point を張った場合には正しく停止するのに対し、MultiThread に Break point を張った際には停止できないと言うことがわかった。そのため非効率ではあるが、Debug を行いたい処理の箇所に Print 文を挟むことで Debug を行った。

3.12　Christie Sharp の記述方法

ソースコード 3.12、3.13、3.14 はソースコード 2.4、2.5、2.6 を Chrisite Sharp で書き換えたものである。Java と C#は大きく記述方法は変わらず、annotation が attribute になっている点が一番の違いである。そのため Christie と Chrisite Sharpt には互換性があると言える。

```
1  public class StartCountUp : StartCodeGear {
2      public StartCountUp(CodeGearManager cgm) : base(cgm) { }
3
4      public static void Main(string[] args) {
5          StartCountUp start = StartCountUp(createCGM(10001));
6      }
7
8      public override void Run(CodeGearManager cgm) {
9          cgm.Setup(new CountUpper());
10         CountObject count = new CountObject(1);
11         put("count", count);
12     }
13 }
```

ソースコード 3.12: Chrisite Sharp における StartCodeGear の記述例

```
1  public class CountUpper : CodeGear {
2      [Take] public CountObject count;
3
4      public override void Run(CodeGearManager cgm) {
5          Console.WriteLine(count.number);
6
7          if (count.number < 10) {
8              cgm.Setup(new CountUpper());
9              count.number += 1;
10             put("count", count);
11         } else {
12             cgm.GetLocalDGM().Finish();
13         }
14     }
15 }
```

ソースコード 3.13: Chrisite Sharp における CodeGear の記述例

```
1  [MessagePackObject]
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 3.14: Chrisite Sharp における DG として送信されるオブジェクトのクラス

第4章　Christie SharpのUnity上での動作

4.1　Unity API

Unityでは、ゲーム開発を行うためのライブラリとしてUnity APIが公開されている。ソースコード4.1はUnity APIを使用したスクリプトの例である。Spaceキーを押し続けている間、playerObjectは物理演算に従って、毎フレーム$\vec{A} = (10, 5, 10)$だけ力を加えられ移動する例である。4行目では変数が初期化されていないまま使用されているように見えるが、Unityではpublic修飾子を使用するとUnityのエディタ上で事前に変数に代入することが可能になる。

```
1  using UnityEngine;
2
3  public class PlayerMove : MonoBehaviour {
4      public GameObject playerObject;
5      private Rigidbody playerRigidbody;
6      private Vector3 power = new Vector3(10f, 5f, 10f);
7
8      // Start is called before the first frame update
9      void Start() {
10         playerRigidbody = playerObject.GetComponent<Rigidbody>();
11     }
12
13     // Update is called once per frame
14     void Update() {
15         if (Input.GetKey(KeyCode.Space)) {
16             playerRigidbody.AddForce(power);
17         }
18     }
19 }
```

ソースコード 4.1: Unity APIを使用した記述例

Unity APIではC#にはないVector3やGameObject、Rigidbodyなどの特別な型があり、これらはUnityEngineをusingすることで使用可能である。またMonoBehaviourを継承することによって、Startメソッドや Updateメソッドなどの Unity上で決められた、あるタイミングで呼び出される関数を使用することが可能である。

Start メソッドはゲームの開始時、1 フレーム目が始まる前に 1 回だけ実行され、主に初期化に使用されることが多い。今回は playerObject に付与されている物理演算の Component である Rigidbody を取得している。

Update メソッドは Start メソットの実行後に毎フレーム呼び出される関数であり、ゲームのメインロジックで使用されることが多い。15 行目ではキー入力の有無を確認しており、Space キーの入力を監視している。続く 16 行目では、Start メソッドにて取得した Rigidbody に対して、3 次元ベクトルを力として与えている。これを行うことによって、Space キーを押している間 $\vec{A} = (10, 5, 10)$ が playerObject に与えられ、playerObject は物理演算に従って移動する。

Unity では Start メソッドや Update メソッドがある代わりに、C#の一般的な実行メソッドである Main メソッドによる処理の開始ができない。そのため、Unity で処理を行うためには、MonoBehaviour により提供されているメソッドを通して処理を実行する必要がある。

4.2　StartCodeGear を使用しない Christie Sharp の実行

Chrisite Sharp では StartCG に Main メソッドを記述して Chrisite Sharp を実行したが、Unity で動作させるためには Start メソッドや Updete メソッドを使用す必要がある。そのため、Chrisite Sharp の実行に StartCG を使用せずに実行できるよう変更を行った。

```
1 public class FizzBuzzTest : MonoBehaviour {
2     void Start() {
3         CodeGearManager cgm = StartCodeGear.CreateCgm(10001);
4         cgm.Setup(new CountUpper());
5         CountObject count = new CountObject(1);
6         cgm.GetDGM.Put("count", count);
7     }
8 }
```

ソースコード 4.2: Unit 上での Chrisite Sharp の実行 Script

ソースコード 4.2 はソースコード 3.12 を Unity で動作するように書き換えたものであり、StartCG を使用せずに、Unity API である Start メソッドより Chrisite Sharp を実行している。StartCG は、CGM の初期化や処理の開始位置を明確化させるためのクラスであるため、Christie Sharp の実行に必須ではない。また、C#では多重継承が禁止されている。そのため、StartCG を継承しつつ、MonoBehaviour を継承して Start メソッドを使用する、と言うことはできない。そのため可読性が下がってしまう可能性はあるが、CGM の初期化や Setup メソッドを直接 Start メソッド内に記述できる。

4.3　Unity 上での動作確認

Christie Sharpt の動作確認を行うため、以下の 3 点の確認を行った。LocalDGM に関してはソースコード 4.2 で正しく動作することを確認している。

- CodeGear を使用した、GameObject の位置の操作
- 同一プロセスでの、複数 DGM を使用した GameObject の位置の同期
- PC2 台を使用した、LAN 上での GameObject の位置の同期

```
1  public class TransformMoveTest : MonoBehaviour {
2      private CodeGearManager cgm;
3      public Transform otherTransform;
4      private Vector3 pos;
5
6      void Start() {
7          cgm = StartCodeGear.CreateCgm(10000);
8          cgm.Setup(new PositionAssignCodeGear());
9          cgm.GetLocalDGM().Put("transform", transform);
10     }
11
12     public void Update() {
13         pos = otherTransform.position;
14         Vector3 newPos = new Vector3(pos.x + 3, pos.y, pos.z + 3);
15         cgm.GetLocalDGM().Put("pos", newPos);
16     }
17
18     private void LateUpdate() {
19         cgm.Setup(new PositionAssignCodeGear());
20     }
21 }
```

ソースコード 4.3: GameObject の位置の操作の準備を行う Script

```
1  public class PositionAssignCodeGear : CodeGear {
2      [Take] private Vector3 pos;
3      [Peek] private Transform transform;
4
5      public override void Run(CodeGearManager cgm) {
6          MainThreadDispatcher.Post(_ => {
7              transform.position = pos;
8          }, null);
9      }
10 }
```

ソースコード 4.4: GameObject の位置の操作を行う CodeGear

ソースコード 4.3、4.4 は、CodeGear を使用した、GameObject の位置の操作を行う処理である。ソースコード 4.3 を付与した GameObject の位置より、常に x 軸に+3、z 軸に+3 離

れた位置に他の GameObject である otherTransfrom が移動をする。19 行目の LateUpdate メソッドは MonoBehaviour から継承している関数であり、全ての Update メソッドが実行された後、次のフレームに移行する直前に実行される関数である。

Unity で Chrisite Sharp を使用する際に一つ問題が起きた。Unity API でサポートされている関数や型は Main Thread で動作時のみ処理が行われるという仕様がある。これは Unity の根本的な考え方であり、Unity では基本的にシングルスレッドでの動作を想定してる。非同期処理を行うための機能として、Coroutine や Invoce などの機能が提供されており、擬似的に Multi Thread のように処理を行うことが可能である。しかし Chrisite Sharp では CodeGear を Task で処理させている。このため CG に直接 Transform を操作しても処理が実行されない。

この問題を解決するために、UniRx を導入することで使用可能な MainThreadDispatcher.Post メソッドを使用した (ソースコード 4.4 6 行目)。UniRx とは Unity 向けに作成された非同期処理の外部ライブラリである。MainThreadDispatcher.Post を使用することで、Main Thread でない Thread で行っている処理を Main Thread に移譲することが可能となる。Post メソッドは第一引数にデリゲートの一種である Action<T> を渡すため、ラムダ式での記述が可能である。

```
1  public class RemoteMoveTest : MonoBehaviour {
2      private CodeGearManager playerCGM;
3      private CodeGearManager enemyCGM;
4      public Transform otherTransform;
5      private Vector3 playerPos;
6
7      void Start() {
8          playerCGM = StartCodeGear.CreateCgm(10001);
9          enemyCGM = StartCodeGear.CreateCgm(10002);
10         playerCGM.CreateRemoteDGM("enemy", "localhost", 10002);
11
12         enemyCGM.Setup(new ObjectMoveCodeGear());
13         enemyCGM.GetLocalDGM().Put("otherTransform", transform);
14     }
15
16     public void Update() {
17         playerPos = otherTransform.position;
18         Vector3 newPos = new Vector3(playerPos.x + 3, playerPos.y,
       playerPos.z + 3);
19         Vector3Cmd vCmd = new Vector3Cmd(newPos.x, newPos.y, newPos.z);
20
21         playerCGM.GetDGM("enemy").Put("newPos", vCmd);
22     }
23
24     private void LateUpdate() {
25         enemyCGM.Setup(new ObjectMoveCodeGear());
26     }
27 }
```

ソースコード 4.5: RemoteDGM で GameObject の位置の操作の準備を行う Script

```
1  public class ObjectMoveCodeGear : CodeGear {
2      [Peek] private Transform otherTransform;
3      [Take] public Vector3Cmd newPos;
4
5      public override void Run(CodeGearManager cgm) {
6          MainThreadDispatcher.Post(_ => {
7              Vector3 pos = new Vector3(newPos.xAxis, newPos.yAxis, newPos.
       zAxis);
8              otherTransform.position = pos;
9          }, null);
10     }
11 }
```

ソースコード 4.6: RemoteDGM で GameObject の位置の操作を行う CodeGear

```
1  [MessagePackObject]
2  public class Vector3Cmd {
3      [Key("xAxis")]
4      public float xAxis;
5      [Key("yAxis")]
6      public float yAxis;
7      [Key("zAxis")]
8      public float zAxis;
9
10     public Vector3Cmd(float xAxis, float yAxis, float zAxis) {
11         this.xAxis = xAxis;
12         this.yAxis = yAxis;
13         this.zAxis = zAxis;
14     }
15 }
```

ソースコード 4.7: GameObject の位置データとして送信されるオブジェクトのクラス

ソースコード 4.5、4.6、4.7 は同一プロセスにおいて、RemoteDGM を使用し GameObject の位置を操作する Script である。

ソースコード 4.5 の Start メソッドでは、playerCGM と enemyCGM の 2 つの DGM を用意し、playerCGM の方では他方の enemyCGM を RemoteDGM として指定を行っている。また、enemyCGM では Setup を行い処理の待機状態にしている。Update メソッドでは、この Script が付与された GameObject の位置を取得し、x 軸に+3、z 軸に+3 離れた位置に otherTransform を移動させるようにし、送信用クラスに変形を行っている。21 行目では otherTransform を持っている enemyCGM ではなく、playerCGM から Put を行っている。

MessagePack CSharp では Unity にも対応しており、Unity API が提供している値型の Vector3 や Quaternion なども Serialize/Deserialize 可能である。しかし、Christie Sharp では DG でデータを管理する際に全ての型に対応できるよう Object 型に Cast をしている。この Object 型の Cast が原因により Vector3 型を直接送信することができない。そのため、ソースコード 4.7 のようにデータをプリミティブ型に分割してデータを送信している。

ソースコード 4.6 はソースコード 4.4 と同様に MainThreadDispatcher.Post によって、処理は Main Thread に移譲されて実行される。

```
1  public class PlayerManager : MonoBehaviour {
2      public CodeGearManager playerCGM;
3      public string hostName;
4      private Vector3 playerPos;
5
6      void Start() {
7          playerCGM = StartCodeGear.CreateCgm(10001);
8          playerCGM.CreateRemoteDGM("listener", hostName, 10002);
9      }
10     void Update() {
11         playerPos = transform.position;
12         Vector3Cmd vCmd = new Vector3Cmd(playerPos.x, playerPos.y,
       playerPos.z);
13
14         playerCGM.GetDGM("listener").Put("newPos", vCmd);
15     }
16 }
```

ソースコード 4.8: GameObject の位置を RemoteDGM に送信する Script

```
1  public class PlayerListener : MonoBehaviour {
2      private CodeGearManager listenerCGM;
3      public string hostName;
4
5      void Start() {
6          listenerCGM = StartCodeGear.CreateCgm(10002);
7          listenerCGM.Setup(new ObjectMoveCodeGear());
8          listenerCGM.GetLocalDGM().Put("otherTransform", transform);
9      }
10
11     private void LateUpdate() {
12         listenerCGM.Setup(new ObjectMoveCodeGear());
13     }
14 }
```

ソースコード 4.9: GameObject の位置を受信して位置を操作する Script

ソースコード 4.8、4.9 は 2 台の PC を使用し LAN 内で GameObject の位置を通信して操作する Script である。CG や、送信する DG については、ソースコード 4.6、4.7 を流用している。

ソースコード 4.8、4.9 それぞれの 3 行目では hostName を指定しており、通信先の Local IP アドレスを入力することで Socket 通信を行っている。ソースコード 4.8 は位置データの送信元であり、RemoteDGM として指定した listener に向けて、自身の GameObject の位置を Put している。ソースコード 4.9 は何もデータを Put しておらず、LateUpdate メソッドで Setup を行うことでソースコード 4.8 から受け取ったデータを基に GameObject の位置を変更している。

以上のテストにより、Chrisite Shrap の機能が問題なく Unity 上でも機能することが確認できた。

4.4　Unity での Chrisite Sharp の役割

Unity で CodeGear を動作させる場合、特に Unity API を使用して処理を行う際は、MainThreadDispatcher.Post メソッドを利用して Main Thread に処理を以上する必要がある。他方、Unity では Update メソッドや FixedUpdate メソッドなど Unity の実装に従って、フレーム単位や時間単位でのメソッド呼び出しが保証されている。CG 内で MainThread-Dispatcher.Post メソッドを使用し GameObject の移動などの Unity API を利用した処理を行うことで、処理が行われるタイミングが不安定になることがあると考えられる。

そこで、ネットワーク上で通信が必要なものだけを Chrisite Sharp で通信を行い、受信データを MonoBehaviour を継承したクラスで処理・動作させるという方法を考えた。Unity には非同期処理として Coroutine や Task などがあるが、シングルスレッドでの動作を前提として利用されている。この手法を取ることにより、Unity 上の処理を行うタイミングが保証されると共に、並列処理やデータのやり取りを Christie に一任することができ、Multi Thread による並列処理が可能となる。

また Unity 上で操作、処理を行っている GameObject 自体を DG に継続にする、直接的には DG にすることによってより並列処理などを意識することなくゲーム開発が可能となる。GameObject 自体を DG にすることにより、通信に必要なデータを送信するためのコードを書かずに待ち合わせが可能となる。また、待ち合わせを行っていることによりデータが Null のまま処理を開始することがなくなる。位置などの毎フレーム必要なデータや、画面の描画に必要なデータは Peek で取得を行うことで、通信が途中で途切れてしまった場合でもその直前のデータは参照可能であり、接続の復帰も行いやすいと考えられる。

第5章　Christie Sharpの評価

前章までは Christie Sharp の実装や Unity 上での動作について述べた。本章では、Unity で使用可能な通信ライブラリである Photon Unity Networking2、Mirror と Christie Sharp の比較を行い、Unity で使用する場合の Christie Sharp の評価を行う。

5.1　Christie Sharpと既存ライブラリの比較

既存の通信ライブラリとしては前述した、Photon Unity Network2 と Mirror を Christie Sharp との比較を行う。表 5.1 は Chrisite Sharp、PUN2、Mirror のそれぞれの特徴をまとめたものである。

表 5.1: 各通信ライブラリの特徴

	node 間の通信方法	Unity API の対応	拡張可能か
Christie Sharp	p2p	未対応	可能
PUN2	Server Client	対応	一部可能
Mirror	Server Client	対応	可能

PUN2 や Mirror は node 間の通信方法としてクライアントサーバ方式を取っている。背景には、PUN2 は Photon Cloud を前提に Server への接続が行われており、Mirror は MMO での動作を前提に作成されているからである。一方 Christie Sharp は、Topology Manager を利用して各 node の接続を行っているため Server は無く、p2p での接続となる。

Unity API の対応に関しては、PUN2、Mirror が Unity API で提供されている独自の型を簡単に同期できる機能があるが、Christie Sharp にはその機能が存在せず、データの同期には C#のプリミティブ型に変換して送信する必要がある。

拡張が可能かについては、Christie Sharp、Mirror はユーザ自身が拡張可能である。Christie Sharp は GitHub にて公開予定であり、Mirror は OSS なため、プロジェクトの状況に応じた機能の追加や変更などが可能となっている。一方 PUN2 は Unity 上で動作する Client 側のソースコードは公開されているが、Server 側は Photon Cloud を使用する必要があるため、拡張性は他の比較対象と比べ低い。

PUN2 では Photon Cloud があるが、無料での使用には制限がかかり、制限を超えると別途 Server の料金が発生する。Mirror は自前で Server を用意する必要があり、処理の大半は Server で行われるため Server のスペックを考えて開発する必要があある。Chrisite Sharp は Unity API への対応が未対応が目立つ一方、p2p で通信を行うため、強力な Server を必要としない。また p2p の形式を取っているが、Topology Manager の Host を Server で作動させることで、クライアントサーバ方式に変更することも可能である。

5.2　Christie Sharp の利点

他の通信ライブラリと比較して、Chrisite Sharp を利用する利点として、以下が挙げられる。

- 単体でも並列処理が可能
- 通信切断した際にゲームロジックが停止しない

他の通信ライブラリでは、並列処理の外部ライブラリを導入する際に、ライブラリ同士の相性や処理方法について考える必要がある。並列処理や非同期処理の Debug は複雑であり、複数のライブラリで問題が起きていた際に特定と解決は困難である。Chrisite Sharp では、CodeGear/DataGear を使用した待ち合わせ処理を基本として処理が行われる。この処理は Multi Thread で行われているため、ユーザは並列処理や非同期処理を必要以上に意識せずにプログラムが可能である。並列処理のライブラリを導入しなくても済むため、問題の特定は複数のライブラリを組み合わせた場合と比べ容易になる。

node 間の通信において、通信の切断は必ず発生する。通信が切断された際に、ゲームロジックが破綻しゲーム全体が続行不可能になる場合もある。

通常、切断時にゲームロジックが停止しないような例外処理を行う必要があるが、Chrisite Sharp では常に参照すべき値を Peek で取得することによって、通信が切断されても参照データは消えずに参照可能である。また、Topology Manager には Cookie の機能が備わっているため、改良することによって通信が切断された場合にでもゲームに復帰することが可能になると考える。

第6章　まとめ

本研究では、Christie の概要について説明を行い、Unity で使用可能な通信ライブラリの特徴を挙げた上で、Christie を C#に書き換える意義について述べた。C#での再実装時に発生した問題である、attribute の実装、CodeGear の処理を Thread から Task への変更、MessagePack のバージョン変更、MessagePack を使用する際の送信パケットの変更について述べた。また、Christie と Christie Shrp の記述方法の違いと、Unity で Christie Sharp を動作させた際の記述について説明した。

他の通信ライブラリとの比較では、PUN2 と Mirror との比較を行い、Christie Sharp は Unity API は未対応であるが、OSS にする予定のため拡張や改良が可能である。また、p2p で通信を行うため強力な Server を使用する必要はなく、Topology Manager の Host を Server で動作させることで、クライアントサーバ方式に変更が可能である。さらに他のライブラリにはない特徴として、単体でも並列処理が可能であり、外部の並列ライブラリを導入せずに済む。参照すべきデータを Peek で取得することで通信切断時にも、ゲームロジックが停止せずに続行可能であり、Topology Manager の機能を拡張することによって、切断された際にゲームに再接続できるのではないかと考察した。

6.1　今後の課題

Topology Manager の再実装が完了しておらず、機能を十分に使用することができていない。特に Topology Manager が動作している Host と親 node 同士の接続はできているが、子 node からの接続がうまくできていない。また、現状では Tree 型の Topology のみ対応しているため、Star 型など他の Topology も対応する必要があると考える。

DG は C#でサポートされている全ての型を指定可能であるが、TreeMap を使用することがある。しかし TreeMap をデータとして通信を行うと、巨大なデータ構造となって送信してしまい分散通信のバフォーマンスを低下させる要因になる。そこで、TreeMap の key を使用して Put/Take を行うように修正を行う必要があると考える。key の他に接続先の hostname:port を 2nd key として指定し TreeMap 内のデータの通信を行う。

Sockt 通信を行っているのは Task により Thread Pool で実行されているが、これの CodeGear 化を行いたい。Topology Manager の実装はその大半が CodeGear として処理

されているため、Socket 通信部分も CodeGear 化が可能であると考える。必要なデータは Take/Peek を利用して渡すことで実行可能である。

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.