継続を使用する並列分散フレームワークのUnity実装

安田 亮[1,a)]　河野 真治[2,b)]

概要：FPS や MMORPG などのゲームにおける通信方式には、クライアントサーバ方式と p2p 方式の 2 つが考えられる。しかし、クライアントの負荷軽減やチート対策などを理由にクライアントサーバ方式が主流である。データの同期にはサーバを経由するため低速である。そこで本研究室で開発している分散フレームワーク Christie を用いることで、高速かつ、安全に、データの同期を行いたいと考えた。本研究では Christie をゲームエンジン Unity に対応するため、C#への書き換えを行う。

1. 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さまざま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。通信形態はクラウド上のサーバを中心とした形態が多い。分散プログラムを正しく書くことは難しく、また、ゲームの場合はさまざまな攻撃が行われることが多い。ネットワーク上のパケットを用いた攻撃やウイルスもそうだが、ゲームのルールにそっていても、プレイヤーのデバイス上でプログラムを用いたチートが行われる場合もある。

TCP/IP が現状のインターネットの標準的なプロトコルであり、その上に、信頼性はないがコネクションなしにデータを転送できる Datagram と、信頼性を保証する通信路である TCP の二種類のトランスポート層が用意されている。しかし、より高いレベルの通信ライブラリにより、攻撃に強く、柔軟にゲームの通信プロトコルを拡張できるフレームワークが望まれている。例えば、Windows 上の Direct X などにはそのようなもの搭載されている。Unity ゲームフレームワークでもさまざまな通信ライブラリが存在する。

当研究室では初代 PlayStation 用に作成した Federated Linda を拡張して、CodeGear / DataGear を用いた分散フレームワークを開発中である。これは Java でかかれており、Unity 上で直接動かすことはできない。そこで、C# で再実装することにより Unity 上のゲームの通信ライブラリとして使用できるようにする。

従来の通信ライブラリと異なり、型のある DataGear をタプル空間 (DataGearManger/DGM) に Key を持つストリームとして格納する方式をとっている。DGM は自ノードには local なものがあり、自ノードのスレッド間での通信に用いられる。他のノードは、この DGM の proxy を持ち、proxy に書き込むことで自ノードとの通信を行う。

DGM の構成はトポロジーマネージャーにより自動的に構成される。ゲームノードは、まず、トポロジーマネージャと通信し、自分が接続するべきノードを知らされる。自分が指示されたノードに接続し、DGM を持つ。ゲームプログラム自体は、複数の名前のついた DGM を知っていればよく、IP address などを知る必要はない。

DGM は proxy であり、DataGear のコピーをそこに持っている。なので、ネットワーク接続の切断があっても、ゲームの動きを止めることなく対応が可能となっている。ゲームノードは直接接続される対象が何かを直接知ることはできないので、チートに対する耐性がある。例えば、まず、チートがあるかどうかを調べるノードに接続するなどの工夫が可能となっている。

本論文では Java で書かれた Chirstie と C# で書かれたものの説明し、その機能と実装の差について考察する。

2. Christie の C#への書き換えについて

Christie は Alice というプロジェクトで開発が行われていた。しかし Alice には様々な問題点があった。データを管理している localDataGear をシングルトンで設計してしまい、local で接続を行う際には複数のアプリケーションを立ち上げる必要がある。また、データを受け取る際に Object 型で受け取っている影響で何の方が送信されるか不明瞭である点などがあり、再設計を行う必要性が発生した。それらの問題点を解消するために Alice を再構築したものが Christie である。Christie 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の

[1] 琉球大学大学院理工学研究科情報工学専攻
[2] 琉球大学工学部工学科知能情報コース
[a)] riono210@cr.ie.u-ryukyu.ac.jp
[b)] kono@ie.u-ryukyu.ac.jp

煩雑さなどを解消している。

今回 Chrstie を C# への書き換えを行う。これは、ゲーム制作において多くの開発者に使用されている Unity に対応するためである。Unity は C# でプログラミングが可能であり、C# と java は比較的書き方が似ているため、書き換えが行いやすいと判断した。C# への書き換えの方針は、java 版との互換性を保つために極力同じ動作をする API を用いて実装を行った。

3. Christie の基礎概念

Christie は当研究室で開発している分散通信フレームワークである。同じく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ようなものがある。

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

以下は java 版の Christie について解説を行う。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 annotation を用いて変数データを取得する。CodeGear 内に記述した全ての DataGear の中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 はノードであり、CodeGear、DataGear、DGM を管理する。DGM は DataGear を管理するものであり、put という操作により変数データ、つまり DataGear を格納できる。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 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 を格納できる。

図 1 は、Christie を同一プロセスで複数のインスタンスを立ち上げた際の DGM や CGM の接続の構造を示している。全ての CodeGearManger 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。

put 操作を行った後は、対象の DGM の中に queue として保管される。DataGear を取り出す際には、CodeGearm 内で宣言した変数データに annotation をつける。DataGear

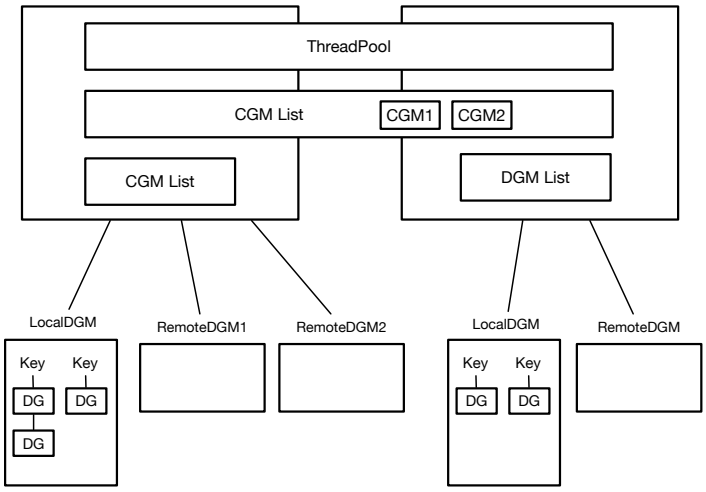

図 1: Christie の複数インスタンスの立ち上げ

の annotation には Take、Peek、TakeFrom、PeekFrom の 4 つがある。

Take 先頭の DataGear を読み込み、その DataGear を削除する。DataGear が複数ある場合、この動作を用いる

Peek 先頭の DataGear を読み込むが、DataGear が削除されない。そのため、特に操作をしない場合は同じデータを参照し続ける。

TakeFrom (Remote DGM name) Take と似ているが、Remote DGM name を指定することで、その接続先 (Remote) の DGM から Take 操作を行える。

PeekFrom (Remote DGM name) Peek と似ているが、Remote DGM name を指定することで、その接続先 (Remote) の DGM から Peek 操作を行える。

4. TopologyManager

TopologyManager とは、Christie 上での Network Topology を形成するために参加を表明したノード、TopologyNode に名前を与え、必要があれば Node 同士の配線も自動で行う機能である。TopologyManager 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 Code 1 のような dot ファイルを与えることで、Node の関係を図 2 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、CodeGear が実行される。

Code 1: ring.dot

```
1 digraph test {
2   node0 -> node1 [label="right"]
3   node1 -> node2 [label="right"]
4   node2 -> node0 [label="right"]
5 }
```

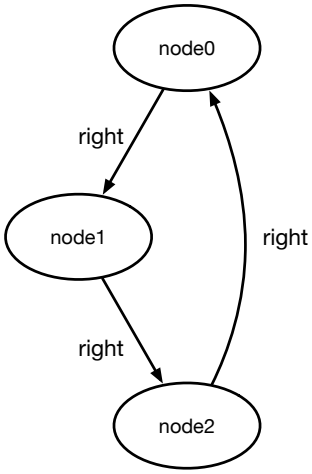

図 2: Code 1 の図示

5. プログラムの例

Code 2 、Code 3 、Code 4 は Christie の機能を使用して hello world を出力する例題である。

Code 2: java StartHelloWorld

```
1  public class StartHelloWorld extends StartCodeGear {
2
3      public StartHelloWorld(CodeGearManager cgm) {
4          super(cgm);
5      }
6
7      public static void main(String[] args){
8          CodeGearManager cgm = createCGM(10000);
9          cgm.setup(new HelloWorldCodeGear());
10         cgm.setup(new FinishHelloWorld());
11         cgm.getLocalDGM().put("helloWorld","hello");
12         cgm.getLocalDGM().put("helloWorld","world");
13     }
14 }
```

Code 3: java HelloWorldCodeGear

```
1  public class HelloWorldCodeGear extends CodeGear {
2
3      @Take String helloWorld;
4
5      @Override
6      protected void run(CodeGearManager cgm) {
7          System.out.print(helloWorld + "␣");
8          cgm.setup(new HelloWorldCodeGear());
9          cgm.getLocalDGM().put(helloWorld,helloWorld);
10     }
11 }
```

Code 4: java FinishHelloWorld

```
1  public class FinishHelloWorld extends CodeGear {
2      @Take String hello;
3      @Take String world;
4
5      @Override
6      protected void run(CodeGearManager cgm) {
7          cgm.getLocalDGM().finish();
8      }
9  }
```

Code 2 では CGM を作り、setup(new CodeGear) を行うことで各 CodeGear に記述された DataGear の待ち合わせを行う。全ての DataGear が揃った場合に CodeGear が実行される。CodeGearaManager の作成方法は StartCodeGear を継承したものから、createCGM(port) を実行することにより、CGM が作成できる。

Code 2 の 11、12 行目は put(key, data) を行うことで DGM の queue にデータを格納することができる。key は string 型のみで格納したい変数名を指定する。11、12 行目の put では Code 3 のフィールド変数 helloWorld を指定し、データは”hello” と”world” を逐次的に格納している。

Code 3 、Code 4 が CodeGear にあたる。それぞれのフィールド変数には@Take annotation が付いており、DGM に格納された key を参照してデータを取得する。その後 DataGearManger に格納されたデータは破棄される。

Code 3 では最初にフィールド変数 helloWorld に string 型の”hellow”を取得、print を行い、再び key hello、data ”hello” を DGM に put している。また 8 行目で自らを setpu しているため、再帰的に HelloWorldCodeGear が再実行される。2 回目の実行ではフィールド変数 helloWorld に”world” が格納と出力がされ、key world、data ”world” が DGM に格納される。Code 3 で put した”hello” と”world” は最終的に、Code 4 の同名のフィールド変数に格納される。2 回目の実行でも setup しているが、DGM には key helloWorld のデータが無いため、3 回目以降は実行されない。

Code 3 の 2 回の実行後、Code 3 のローカル変数 hello と world が全て揃ったことにより Code 3 が実行されプログラムは終了する。

6. Christie

Code 5、Code 6、Code 7、は Code 2 、Code 3 、Code 4 の例題を C# に書き換えたものである。

Code 5: C# StartHelloWorld

```
1  public class StartHelloWorld : StartCodeGear {
2
3      public StartHelloWorld(CodeGearManager cgm) : base
           (cgm) { }
4
5      public static void Main(string[] args) {
6          CodeGearManager cgm = CreateCgm(10000);
7          cgm.Setup(new HelloWorldCodeGear());
8          cgm.Setup(new FinishHelloWorld());
```

```
9         cgm.GetLocalDGM().Put("helloWorld", "hello");
10        cgm.GetLocalDGM().Put("helloWorld", "world");
11    }
12 }
```

Code 6: C# StartHelloWorld

```
1  public class HelloWorldCodeGear : CodeGear {
2      [Take] string helloWorld;
3
4      public override void Run(CodeGearManager cgm) {
5          Console.Write(helloWorld + "␣");
6          cgm.Setup(new HelloWorldCodeGear());
7          cgm.GetLocalDGM().Put(helloWorld, helloWorld);
8      }
9  }
```

Code 7: C# StartHelloWorld

```
1  public class FinishHelloWorld : CodeGear {
2      [Take] private string hello;
3      [Take] private string world;
4
5      public override void Run(CodeGearManager cgm) {
6          cgm.GetLocalDGM().Finish();
7      }
8  }
```

java と C# はクラスや変数などの記述方法が似ているため、書き換えの際の大きな変更は少ない。C# では java の annotation はなく、attribute を利用する。attribute の使用方法は Code 6 の 5 行目のように、attribute を付与したい変数の前に [Take] などつけることで使用可能である。

7. Unity

8. Unity での動作

9. Take annotation の実装

java 版では DataGear を取得する際に、annotation という java の機能を用いて行った。C#には annotation はなく、代わりに attribute を利用して DataGear の取得を行っている。以下の Code 8、Code 9 は java と C# における Take の実装である。

Code 8: java における Take annotation の実装

```
1  @Target(ElementType.FIELD)
2  @Retention(RetentionPolicy.RUNTIME)
3  public @interface Take { }
```

Code 9: C# における Take attribute の実装

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

java で annotation を自作する際には、 @interfacs で宣言する。また、Code 8 の 8 行目では annotation 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できる。9 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適応可能となっている。

C# で attribute を作成する際には、 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、Code 9 の 4 行目でフィールド変数を指定している。

10. MessagePack の相違点

Christie ではデータを送信する際に、MessagePack を使用してデータを圧縮し、送信している。java 版で使用している MessagePack はバージョンが古く現在はサポートされてない。そのため MessagePack の最新版とは記述方法が異なっている。Code 10 は MessagePack の使用方法を示したものである。

Code 10: java における MessagePack の使用方法

```
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 }
```

MessagePack を使用するには圧縮するクラスに対して @Message annotation をつける必要がある。これにより、クラス内で定義した public 変数が圧縮される。Code 10 の 17 - 21 行目は圧縮解凍の例であり、MessagePack のインスタンスを作成後、msgpack.write(data) を行うことで byte[] 型に data を圧縮できる。解凍には msgpack.read を使用し、圧縮された byte[] 型と圧縮対象のクラスを渡すことで解凍できる。

C# の MessagePack は複数存在しており、java と同様な書き方をする MessagePack-CSharp を選択した。

Code 11: C# における MessagePack の使用方法

```
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        byte[] bytes = MessagePackSerializer.Serialize
              (mc);
18        MyClass mc2 = MessagePackSerializer.
              Deserialize<MyClass>(bytes);
19
20        // [99,"hoge","huga"]
21        var json = MessagePackSerializer.ConvertToJson
              (bytes);
22        Console.WriteLine(json);
23    }
24 }
```

MessagePack-CSharp では java 版と同様にクラスに対して圧縮を行うため Code 11 の 1 行目で MessagePackObject attribute を追加している。また、圧縮する変数に対して key を設定することができ、int や string を指定することができる。

データの圧縮には MessagePackSerializer.Serialize (data) を使用し、byte[] 型に圧縮される。解凍には MessagePackSerializer.Deserialize<T>(data) を使用する。Deserialize はジェネリスク関数であるため、<> 内に解凍するデータのクラスを指定する。Code 11 の 21 行目では、変数それぞれに key を設定していることで json に展開することが可能である。

11. CodeGear 実行時の ThreadPool から Task への変更

java 版では CodeGear の実行に ThreadPool を使用していた。C# では書き換えの際に ThreadPool よりも高機能な Task で書き換えを行った。

Code 12: java における CodeGear を処理する ThreadPool の実装の一部

```
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 }
```

Code 12 は java 版における CodeGear を実行する ThreadPool の実装の一部である。java では独自に ThreadPool を作成する際には ThreadPoolExecutor を継承する。また優先度の機構が実装されており、CodeGear 実行時に優先度を決めることが可能になっている。CodeGear の実行には 17 行目の execute を呼び出すことで、実行される。

Code 13: C# における CodeGear を処理する ThreadPool の実装

```
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  }
```

Code 13 は Code 12 を C#に書き換えを行ったものである。CodeGear 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。Task は C# の ThreadPool を拡張したもので、内部に ThreadPool と実行待ち Queue を持っている。スケジューラーを自作することも可能である。実装の優先度が低かったため、今回は CodeGear の priority による実行順変更については実装を行わなかった。

12. Unity で使用されている通信ライブラリとの比較

13. チート対策について

オンラインネットワークゲーム開発にはチート対策が必須になっている。チートの種類には、サーバへ送信するデータをクライアント側で改竄を行うもの、メモリ上の値を書き換えるものなど多岐にわたる。

通常のオンラインゲームでのチート対策としては、クライアントをモニタリングする、ダメージ計算などを全てサーバ側で処理する、ゲームをプレイしている他のプレイヤーからの通報などがある。しかし、チート開発とチート対策はいたちごっこになっているというのが現状である。

Christie では従来の通信ライブラリとは異なり、型がある DataGear をタプル空間である DGM に key を持つストリームとして格納する方式をとっている。他のノードとの通信には DGM の proxy に書き込むことで可能としており、DGM の構成には Topology Manager が自動的に構成される。そのためクライアントはどこに接続を行っているかを直接知ることなく、IP アドレスなどの余計な情報なしに通信が可能となっている (図 3)。

14. 実装の現状

参考文献

[1] RICHARDSON, T., AND LEVINE, J.: The remote framebuffer protocol. RFC 6143 (2011).
[2] TightVNC Software: http://www.tightvnc.com.
[3] RICHARDSON, T., STAFFORD-FRASER, Q., WOOD, K. R., AND HOPPER,: A. Virtual Network Computing (1998).
[4] LOUP GAILLY, J., AND ADLER, M.: zlib: A massively spiffy yet delicately unobtrusive compression library., http://zlib.net.

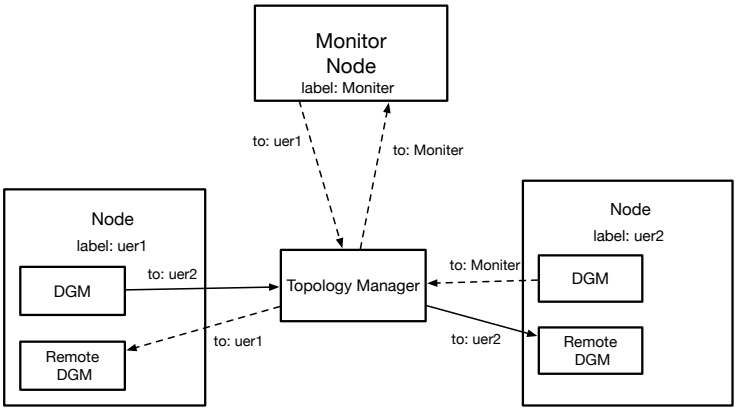

図 3: Topology Manger を介したデータ通信

[5] Yu TANINARI and Nobuyasu OSHIRO and Shinji KONO: VNC を用いた授業用画面共有システムの実装と設計，日本ソフトウェア科学会第 28 回大会論文集 (2011).
[6] Yu TANINARI and Nobuyasu OSHIRO and Shinji KONO: VNC を用いた授業用画面共有システムの設計・開発，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) (2012).