CB-006

自律型並列分散処理システム AgentSphere における JSON を用いた強マイグレーション方式の提案と改良

Proposition and Improvement of a Strong Migration Method using JSON in Autonomous Parallel/Distributed Processing System “AgentSphere”

ディダ ベサリ† 甲斐 宗徳†
Besar Dida Munenori Kai

1. はじめに

近年では並列処理や分散処理に対するニーズが上昇している。しかしながら、すべてのユーザが並列処理や分散処理を行うための専門的な知識およびそれらの環境を有しているわけではない。筆者らはそのようなユーザでも手軽に並列処理や分散処理を行えるプラットフォーム、AgentSphere を開発している。これはモバイルエージェントシステムであり、AgentSphere を動かした複数のマシンが接続されたネットワーク上の他の AgentSphere 間でモバイルエージェントを移動させ、それによって並列分散処理を可能とする。AgentSphere は並列に動作する様々なモバイルエージェントを活用することを想定しており、エージェントのモビリティには強マイグレーションを可能にすることが重要な技術であると考えている。

この強マイグレーションでは、マシン間で実行時のコード、スタック領域、ヒープ領域の移動が必要であり、実行中のモバイルエージェントの処理の中断、移動、移動先での処理の再開が出来る限りスムーズに行くことが必要である。すなわち高性能な並列分散処理のためには、強マイグレーションに伴うオーバヘッドは極力小さくしなければならない。モバイルエージェントのコードが移動先で再コンパイルを必要とするのでは、その分オーバヘッドがかかることになるため、移動先でそのまま実行できるコードとなる Java を開発言語にした。ただし、Java が持つシリアライズ機能は移動できるデータ構造に制約があるとともに、弱マイグレーションしかサポートされないといった問題を持っている。

AgentSphere における強マイグレーションの実現方法として、筆者らはいくつかの手法を試みてきた。初期は、Java のシリアライズ機能を利用するが、ソースコード変換を行って擬似的に強マイグレーションを実現した。他にも Javaflow を用いた方式も実現したが、いずれも移動可能なデータ構造に制約が残り、またマイグレーションに伴うオーバヘッドも大きく、モバイルエージェントのレスポンス時間が大きくなる問題点が明らかとなった。

そこで本論文では Java のシリアライズ方式ではなく、JSON 形式でのシリアライズを可能とする変換方式を用いた新たな強マイグレーション方式を提案する。この方式により改善されるオーバヘッドの評価を示し、他の実装方式と比べてその有用性を示す。

2. AgentSphere とは

AgentSphere は著者らが開発しているモバイルエージェントシステムであり、並列分散処理のプラットフォームを提供するものである。開発目的は単一のマシンではなく複数のマシン上にモバイルエージェントを分散させ並列処理することにより、処理の高速性と信頼性を向上すること、それに加えて、専門的な知識と高性能な並列処理環境を持たなくても、ネットワーク接続された複数のコンピュータを利用でき、並列分散処理以外のプログラミングが可能なユーザであれば、手軽に並列処理や分散処理を実現できるようにすることである。これはモバイルエージェントが自律性を持って行動できるようにすることで実現している。また、モバイルエージェントのモビリティとしては、弱・強マイグレーション両方に対応して幅広い応用に対応できるようにしている。AgentSphere の開発は Java ベースで行われ、JavaVM が起動する環境のすべてに対応できる。

ネットワーク上のあるコンピュータで AgentSphere を動かすということは、モバイルエージェントの出入りする空間を提供することを意味するものとし、AgentSphere 間に展開されるモバイルエージェントを本論文では単に “エージェント” と呼ぶことにする。AgentSphere は将来の機能の拡張性を保つために、複数のモジュールおよびサブモジュールから構成されており、必要に応じてこれらのモジュールやサブモジュールを追加することにより拡張できる。また AgentSphere で開発されるアプリケーションにおいては、必ずしも高機能なエージェントを動かすということではなく、比較的分かりやすい小さな単位の専門機能を持つエージェント群を色々と組み合わせて目的に合った処理を実現することを可能にするものである。

2.1 AgentSphere の構成

エージェントが自律的な動作を行うために AgentSphere はいくつかの主要モジュールから構成される。主要モジュールはすべてのモジュールを総括し内包するシェルモジュールの上に階層型クラスローディングモジュール、ランチャモジュール、ネットワークモジュール、メッセージモジュールが読み込まれる。上記のモジュールを支援するためにモニタモジュール、ファイルシステムモジュール、ウォッチャモジュールなどもある。

AgentSphere のモビリティを実現しているモジュールの一つは階層型クラスローディングモジュール [2] である。クラスローディングモジュールの一般的な設計では検索委譲先として親クラスローダへのリンクをもっている。一つの AgentSphere からマイグレーションの命令を受けて他の AgentSphere に移動したエージェントは未知クラス、すなわちクラスパス上に存在しないクラスとなるので、AgentSphere で使用されているクラスローダはこれらクラスパス上にないクラスの処理に特化している。実行中のエージェントの実行状態を保存し別マシンに移動させてもこ

†成蹊大学理工学研究科理工学専攻 Graduate School of Science and Technology, Seikei University

Copyright © 2018 by
The Institute of Electronics, Information and Communication Engineers and
Information Processing Society of Japan All rights reserved.

のエージェントが未知クラスの場合には Java 標準のクラスローダでは例外が発生する。AgentSphere の階層型クラスローダは、移動してきた未知エージェントの処理を再開させる時に、そのエージェントが使用しているクラスを順にロードしていきそこでの JVM 内のインスタンスとする。

ユーザが快適に AgentSphere を使用するためには環境である AgentSphere を立ち上げると、AgentSphere 自体を操作するためのシェルも起動する。AgentSphere ではこれをシェルモジュールとして搭載している。このシェルは AgentSphere の実行環境のすべてのモジュールを内包し、操作を行うための最低限のシェルとしていくつかのコマンドの機能を提供する。他モジュールを内包するのはもちろんのこと、ユーザにも使いやすくするために OS を問わずに操作できるシェルとしても使用される。シェルはノンブロックであり、ユーザは同じマシンで複数のローカルエージェントの操作を行うことができる。このシェルの上にはエージェントを起動させるためのランチャモジュールも搭載され、エージェント同士の通信を可能とするためのリスナとしてメッセージモジュールも含まれている。

次に、AgentSphere のシェル内で重要なモジュールがネットワークモジュールである。このモジュールは一つの AgentSphere のネットワークに関するマイグレーションやメッセージングをすべて総括する。これがなければすべてのモジュールとエージェントのために独自のネットワーク機構をその都度生成しなくてはならないので余計なオーバヘッドとなる。ネットワーク上のすべてのエージェントは DHT [1] でお互いの位置を把握している。

2.2 Migration

マイグレーションの技術は AgentSphere の核となる技術である。AgentSphere が並列分散処理を行うためにはエージェントが同一ネットワーク内のほかの AgentSphere に移動する必要がある。ただし、コードのみを移動させると再開をするときに常に先頭から開始しなければならない。これは処理の種類によっては問題ないが、移動前の処理を無駄にしていることになる。一方、エージェントが現在の実行状況を保存し、移動先で今までの処理の続きから再開するためにはコードだけではなく、スタック領域、ヒープ領域も移動し復元する必要がある。前者のコードのみの移動を弱マイグレーション、後者のスタック領域、ヒープ領域も併せて移動させるのを強マイグレーションと呼ぶ。

AgentSphere に採用されているのは強マイグレーションである。しかし、この強マイグレーションは、現在最良と考え提案する JSON を利用した実装の他に、次節で説明する 2 つの方式を試みた。

3. Migration の今までの状況

3.1 初期の強マイグレーション

初期の AgentSphere の強マイグレーション [3] はソースコード変換器を使用し、Java 標準のシリアライズ機能を用いた弱マイグレーションの上で疑似的な強マイグレーションを実装していた。

このソースコード変換器は強マイグレーションを行うメソッドを記述したエージェントを変換対象とし、実行中のエージェントのコードとヒープ領域、そしてスタック領域とプログラムの再開位置を一時的にヒープ領域に待避して、弱マイグレーションにより移動させ、移動先で再開位置までに構築されていたスタック領域を復元して実行再開できるようにコード変換を行う。この方式を疑似強マイグレーションと呼ぶことにする。しかし、これは疑似的ではあるものの AgentSphere のその後のマイグレーション方式をを開発するための土台となった。

具体的な変換内容を図 3.2-1.に示す。図 3.2-1.の左側の図は変換前のエージェントコードを表している。コードの記述者は，エージェントが実際に行う振る舞いに加え，移動をさせたい任意の位置に強マイグレーション命令である migrate()を記述する。

ソースコード変換器は，変換対象となるソースコード中の強マイグレーション命令の位置を基にしてソースコード変換を行う。図 3.2-1.の左図のコードを変換した結果は 3.2-1.の右図のようになる。

このように強マイグレーション命令の前後を if 文で分割することで移動後のエージェントが移動前の状態から処理を再開することが可能となり，JVM に変更を加えずに強マイグレーションな振る舞いを実現する事が可能となる。

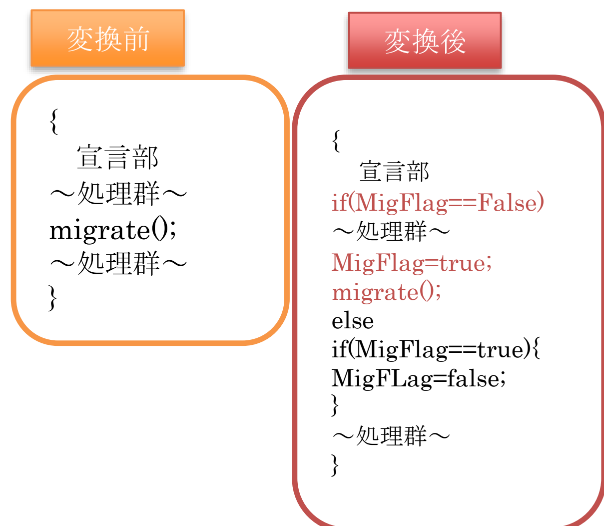


図 3.2-1.ソースコード変換前後の様子

3.2 JavaFlow マイグレーション

次のマイグレーション方式として導入されたのが Javaflow である。Javaflow とは、JVM 上に実行されているオブジェクトの現在の実行状況を保存するための API である。Javaflow を使用するようにビルドされた AgentSphere 上では Continuation.suspend()メソッドにより現在のプログラムの実行をすべて中断し Continuation データが作成される。この中には現在のスタック領域、ヒープ領域および JVM のプログラムカウンタが含まれることとなる。これをシリアライズしたのち、同じ JVM においてデシリアライズされると Continuation.startWith()メソッドを使用して Continuation データから中断されたプログラムを再開することができる。ただし、再開は同一 JVM 上に限定されており、シリアライズ後の直列化データを他の JVM に移動した場合には、そこ

Copyright © 2018 by
The Institute of Electronics, Information and Communication Engineers and
Information Processing Society of Japan All rights reserved.

では未知クラスとなり、実行を再開することができない。しかし AgentSphere との組み合わせでは、前述の階層型クラスローダの働きにより、移動先でもこのクラスを実行することが可能となる。従って、3.1 で述べた疑似強マイグレーションよりも実装は容易になるが、Javaflow の Continuation データに入るデータ構造の中にはシリアライズできないものもあり、移動可能なエージェントの制約になってしまう問題が生じる.

Javaflow を利用した強マイグレーションの実現にはいくつかの問題点があり、AgentSphere を運用する上では変更しなければならないところがあった。強マイグレーションを可能とするシリアライザは AgentSphere で運用するために想定されていない問題が存在している。JavaFlow 強マイグレーションを用いた既存の直列化には AgentSphere にとって不必要などうさ。（JDK1.8 現在）

オーバヘッドの要因の一つとなるのが Java 既存のシリアライザがクラスを直列化・復元するときにすべてのクラス階層をたどりながら複数の読み書き込みのシステムコールを行っていることである。これらのシステムコールはシリアライズを行う際の Read/Write であり、マイグレーションの移動先にもすでに存在しているすべての AgentSphere が有しているところも参照している。

次に、直列化を行うときにリフレクション［4］を用いてコード内の互換性を確認している。しかしながら Javaflow を用いた強マイグレーションの AgentSphere は JRE1.6 までしか動作を保証しないためこれ以前のバージョンの互換性を確認すること自体は不要となる。

Java の直列化は JDK の変更などにも弱い。Java の既存の直列化を使用した場合、同じ JDK 内で直列化されていないとシリアルバージョンの食い違いが発生し、復元が不可能となる。

従来の直列化を用いることによって発生した上記の手法を AgentSphere 用に改良すればよりオーバヘッドの少ない AgentSphere となる。

3.3 改良案

AgentSphere のシリアライザおよび現状のビルドに含まれている問題点を解決するためにはいくつかの案を考慮した。

Java の既存シリアライザにおける上記 3.2 での問題点を解決するには Java 既存シリアライザの独自実装が求められる。AgentSphere に不必要なメソッドやシステムコールを削除すればよりレスポンスタイムの早いマイグレーションが実装できる。しかし、これには Java の既存シリアライザに付随するすべてのライブラリもすべて変更させる必要があり、本来の想定されている並列分散処理を行う専門的なマシンや知識を持たないユーザにとって望ましくないものである。加えて、これらのライブラリの変更はすべての JVM 上において均一で行われる必要があるため、実装方法としては AgentSphere を複雑化させるものである。

次に考慮された方式は、Java の既存シリアライザではなくカスタム性の高い、公開されているものを AgentSphere に見合うように組み込むことである。高いカスタム性のシリアライザを使用すれば AgentSphere 専用の設定を行うだけで JVM における大掛かりな変更をせずに済む。これにより、レスポンスの向上だけではなくより一貫とした AgentSphere が実現できるのではないかという点に着目した。

これらの実装方式を踏まえたうえで筆者らは高いカスタム性を保ちながらオープンソースで今でも開発が続けられている Google-Gson を用いることにした。他の類似したシリアライザもあるが、AgentSphere が必要とするシリアライズを行うにはこれがより適任だった。

既存の JavaFlow を用いた直列化されたデータは基本的に AgentSphere および Java 環境でしか読み込みを行えない。これを、Google-Gson のシリアライザで JSON データベースの形にすることで将来の拡張性を確保するとともに後述の先読みを用いたセキュリティの実装を容易にすることができる。

Google-GsonJSON 形式にしたことで本来直列化できないフィールドやクラスが直列化可能となる。Google-Gson の直列化では専用のインスタンスクリエイタをシリアライザ内部で記述することが可能となり、例外に対する対応が柔軟である、さらに、未知のクラスや Private フィールドを受け入れることが可能である。Private フィールドの場合はデータが保存されないが、これらは外部でそれぞれのクラスに対応する AgentSphere 内部の専用クラスが保管する。これにより直列化の方式が改良されていない現状では直列化不可能だったクラスの直列化が行えるようになった。こういったクラスらの主な問題点としては自身の実行状態を記述するフィールドがプライベートであるため直列化できない。本研究では Thread クラスが直列化できないクラスの類であったため、AgentSphere に追加実装されている AgentThread クラスが直列化寸前の時にそれらの情報を保存することにより JSON 形式で直列化した時にアペンドする。受け取り側の AgentSphere はこの方法で Private フィールドを再構築することができる。現状では Thread クラスのみを考慮したが同じ手法であれば他の直列化不可能な記述およびクラスも直列化可能となる。

4. Google-Gson

Google-Gson [5] とは、オープンソースの Java ライブラリであり、JAVA オブジェクトを JSON 形式に直列化・復元するために使用される。Gson はユーザが動作を記述できるリフレクションを使用しているため基本的にどのようなクラスオブジェクトでも直列化・復元が行える。

例として、事前に Car や Person などのクラスを定義していたならば、コード 4-1.のように変換される。

Copyright © 2018 by
The Institute of Electronics, Information and Communication Engineers and
Information Processing Society of Japan All rights reserved.

```
Gson gson = new Gson();
Car audi = new Car("Audi", "A4", 1.8, false);
Car skoda = new Car("Škoda", "Octavia", 2.0, true);
Car[] cars = {audi, skoda};
Person johnDoe = new Person("John", "Doe", 245987453, 35,
cars);
System.out.println(gson.toJSON(johnDoe));
```

```
   "name": "John",
   "surname": "Doe",
   "cars": [
       {
           "manufacturer": "Audi",
           "model": "A4",
           "capacity": 1.8,
           "accident": false
       },
       {
           "manufacturer": "Škoda",
           "model": "Octavia",
           "capacity": 2,
           "accident": true
       }
```

コード 4-1. ソースコード変換前後の様子

Gson の主な特徴として、復元を行っているとき、Gson は復元されているオブジェクトのタイプツリーのみをたどっているため、不必要なフィールドが含まれていることがなくなる。さらに、独自のシリアライザ・デシリアライザを記述することで、ソースコードのないクラスのインスタンスもシリアライズ・デシリアライズ可能となる。

インスタンスクリエイタを記述することで本来シリアライズ可能ではなかったクラスおよび未知のクラスを復元できる。本研究では独自のインスタンスクリエイタおよびデシリアライザを記述することで階層型クラスローダ [2] の内部で行われていた未知クラスのパス取得およびインスタンスのキャスティング部分を省略することで高速化と安定性が増している。

4.1 JSON

JSON とは、JavaScript Object Notation の略であり、オープン標準なファイル形式である。キーバリューペアおよびシリアライズ可能な配列を人が読める形のテキストで取り扱う。もともとは JavaScript から派生したものだが、今はどのプログラミング言語でも標準に扱えるようになっている。JSON の最も好まれる使い方はサーバとクライアントの情報のやり取りである。AgentSphere も他の AgentSphere と通信をしているときは一種のサーバ・クライアント状態にあるため本研究では JSON に着目している。
Json は以下のコード 4.1-1.ようにデータの構造体を出力する。

```
{
  “firstName”: “Besar”,
  “lastName”: “Dida”,
  “loginCount”: 2,
  “isProgrammer”: true,
  “worksWith”: [“Kailabo”],
  “pets”: [
    {
      “name”: “Tama”,
      “type”: “Cat”
    }
  ]
}
```

コード 4.1-1. JSON 出力の例

本研究では上記の JSON 出力をベースに、Java のインスタンスオブジェクトに適用させることで、本来の Java のバイトコード変換ではなくより柔軟に使用できる JSON 形式のマイグレーションを実装した。

この上記の Gson は Java オブジェクトを JSON 形式でシリアライズする。そして、Javaflow 強マイグレーション方式で実装されている Agent 自身もオブジェクトとして存在しているため、それらを Java の既存シリアライズ・デシリアライズを行うかわりに、Gson でシリアライズすることが可能となる。

そうした場合、コードは以下の 4.1-2.の通りに JSON オブジェクトとして変換される。

```
   "public class Hoge":[
      {
          "public static void main":[
                int:[
                    {
                      name: "piyo1",
                      value: "11"
                    },
                    {
                      name: "piyo2",
                      value: "22"
                    }
                ]
          }
      ]
```

コード 4.1-2. JSON オブジェクトとしての出力

コード内に存在するそれぞれのメソッドやメンバはすべて JSON 形式でその型、中身、値、引数などがすべて渡される。JSON 形式は様々な研究や分野で使用されている [6] [7] [8] で利用されているため本研究における AgentSphere の柔軟性を向上させるために適任の変換方式だと着目した。拡張されたそれらの利用方法については本論文の範囲外であるが、将来のさらなる研究のための布石となった。

4.2 JSON を用いたマイグレーション

Gson ではエージェントをシリアライズ・デシリアライズし、専用の階層型クラスローダへ組み込む手順は既存のシリアライズと大まかな流れは同じだが、シリアライズとデシリアライズに関するところだけが変更されている。流れ

Copyright © 2018 by
The Institute of Electronics, Information and Communication Engineers and
Information Processing Society of Japan All rights reserved.

としては、新たな Gson のためのインスタンスを作成し、現在のエージェントとしての記述をインプリメントした型である AbstractAgent が何なのかを学習させるために、すなわち Json としてどのように出力させるのかを定めるために Type トークンとして AbstractAgent を与える。

これにより Gson はエージェントをどのように記述すればよいのかを学習し Gson.toJson でそれを実際に Json 形式に変換させる。本来ここで AgentSphere のモジュールの一つであるネットワークモジュールを用いたメッセージパッシング機能で Network Sender に直列化されたバイトデータが対象のマイグレーション先に運ばれるのだが、代わりに Json 形式のデータが運ばれる。

InputStream から Json 形式を受け取るのだが、この時点ではまだ Java のバイトデータと同等に扱う、つまり、マイグレーションが終了した後デシリアライズが発生し、階層型クラスローダによってクラスパス外のクラスとして扱われる。続くステップでは Gson.toJson の逆である Gson.fromJson を用いて復元し新たな実行可能なインスタンスを JVM 内で作成し、実行させる。しかし、最後ではインスタンスを作るときにマイグレーションしてきた Json データをどのように処理すればよいのかがわからずインスタンスの作成がうまくいかない。そのため、Gson はユーザに独自のインスタンスクリエイタの記述を可能としている、コード 4.2-1.のように、このクリエイターを用いれば Json の中身をどのようにオブジェクトとして生成するのかを学習させる。

```
final class AgentDeserializer
      implements JsonDeserializer<AbstractAgent> {

   private static final JsonDeserializer<AbstractAgent> AgentDesirializer = new
AgentDesirializer();

   private AgentDesirializer() {
   }

   static JsonDeserializer< AbstractAgent > getAgentDesirializer() {
      return AgentDesirializer;
   }
   @Override
   public AbstractAgent deserialize(final JsonElement jsonElement, final Type
AbstractAgent, final JsonDeserializationContext obj) {
      final JsonObject root = jsonElement.getAsJsonObject();
      final AbstractAgent aa = (AbstractAgent) obj;
      aa.key = root.get("key").getAsJsonPrimitive().getAsInt();
      aa.obj = GsonAgentCreator(aa.key).fromJson(root.get("obj"), obj.class);
      return aa;
   }
   private static Gson GsonAgentCreator(final int key) {
      return new GsonBuilder().registerTypeAdapter(obj.class,
(InstanceCreator<(AbstractAgent) obj>) type -> new obj(key)).create();
   }}
```

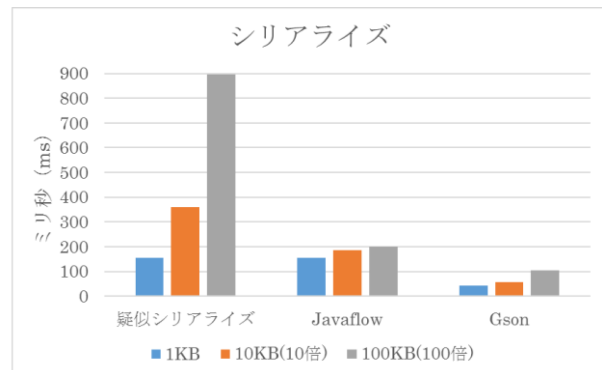
コード 4.2-1. GSON を用いた特殊インスタンスクリエイタ

4.3 比較実行結果

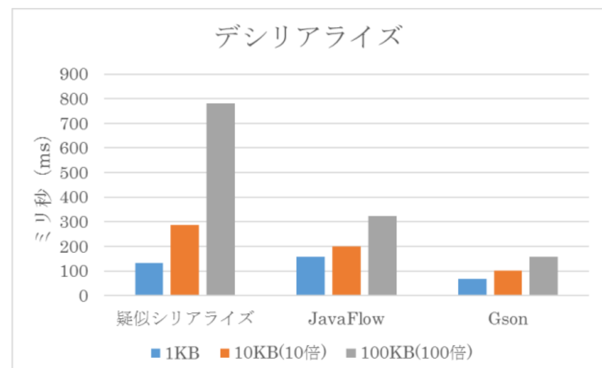
ここでは今までの AgentSphere の強マイグレーションの実装方式を同じデータ構造体（強マイグレーション記述されたエージェント）で測ってみた。ファイルサイズもそれぞれ初期のソースコードからそれぞれ 1KB、10KB および 100KB となるように記述を書き足しているが同じ構造体が繰り返されるだけである。この構造体の中ではエージェントの記述の中で最も使用されるであろう Java の基本構文および再帰的な呼び出しを行うメソッドを含んでいる。

初期に疑似強マイグレーションおよびソースコード変換器を用いて構築された AgentSphere を「疑似シリアライズ」、後期の Javaflow を用いて強マイグレーションを実装した AgentSphere を「Javaflow」そして Gson を用いて強マイグレーションを実装した AgentSphere を「Gson」として、性能の比較をした。

下記のグラフ 4.3-1.および 4.3-2.ではシリアライズ時とデシリアライズ時の実際にその処理だけが完了するまでの時間差を計測した。疑似シリアライズではシリアライズ以前にソースコードの変換というオーバヘッドが発生しているのが確認できる。厳密には Java のオブジェクトのシリアライズが可能という特性を用いて完成されたた技術であり、上記の他の強マイグレーション比べると性能は落ちてしまう。しかし、これが AgentSphere の強マイグレーションをより進めるために必要な過程である。次に、Javaflow を用いたシリアライズ、つまり Java の既存のシリアライザを使用している AgentSphere では 4.3-1.で発生しているオーバヘッドにより、Gson を用いた AgentSphere より遅くなっている。これらの問題点については 3.2 で述べている通りである。



グラフ 4.3-1　シリアライズの性能差



グラフ 4.3-2　デシリアライズの性能差

Copyright © 2018 by
The Institute of Electronics, Information and Communication Engineers and
Information Processing Society of Japan All rights reserved.

この Gson を用いた新たな方式により、AgentSphere は既存より軽量化され、拡張性を増すことができた。しかし、それでもいくつかの問題が残っている。

Java のコードより JSON 形式へ変換する際、転送される実質のソースコード、すなわち Json 記述されたエージェントのオブジェクトと各領域のデータ量がやく 30%増加する。これは JSON 形式になったことによる追加の記述が発生したのが原因である。通常の AgentSphere の運用上ではこの増加によるオーバヘッドはシリアライズ速度を向上させた効果を上回ることは無く、全体的な速度の向上が実現された。

今後の課題としてはより大容量のエージェントについての実験とそれらの影響ではあるが、運用されるエージェントが現実的に大容量となることは今のところは想定されていない。

5. AgentSphere のその他の機能

5.1 セキュリティ

AgentSphere のベースとなっている Java の言語には今もなお解決されていないセキュリティの脆弱性が存在する。これらの脆弱性の中で危険なものは新しい JDK と Java のバージョンアップデートで対応されるが、中では根本的に解決が難しいものもまだ存在している。

AgentSphere に残る一つの問題シリアライズおよびデシリアライズを用いた構造である。複数のマシンはお互いの未知のデータ構造を受け入れなければならず、それらの中に悪いのあるコードが含まれている可能性が大いに存在している。これらの未知のデータ構造のデシリアライズが階層型クラスローダおよび Gson の専用インスタンスクリエイタの機能により可能となっているため、AgentSphere の強マイグレーションの強みでもある。

5.2 先読みの必要性

未知のエージェントをデシリアライズすると現状の AgentSphere においていくつかの問題が発生してしまう。AgentSphere が機能するためにはエージェントが動く空間を作り出さなければならず、この空間こそが AgentSphere である。この空間、すなわちマシン上で起動された JVM には様々な OS やサードパーティの依存関係にあるライブラリと AgentSphere 専用のライブラリが組み込まれている。ユーザ権限でもこのようなライブラリやミドルウェアへのアクセスがデシリアライズ中に図 5.2-1 のクラス記述子を受け取っている時点で一時的ではあるが発生する。[9] このライブラリを使用して、データの盗難や AgentSphere に常駐する悪意のあるエージェントの作成などが行われる恐れがある。未知のクラスをすべてシャットアウトすれば解決するのだが、未知のクラスを受け入れられる AgentSphere にとってはモビリティを損なうためそのようなことは現状ではできない。

本研究ではブラックリストおよびホワイトリストを使用したデシリアライズ中の Java コードのパターン認識を可能とする SerialKiller [10] のデシリアライズライブラリを導入した。

SerialKiller は直列化で普段使われている ObjectInputStream の代わりに独自の入力ストリームを使用することによりオブジェクトのデシリアライズが行われていると同時にコード内にあるパターンをホワイトリストやブラックリストと比べている。

```
ObjectInputStream ois = new ObjectInputStream(is);
String msg = (String) ois.readObject();
```

↓

```
ObjectInputStream ois = new SerialKiller(is, "/primula/serialkiller.conf");
String msg = (String) ois.readObject();
```

コード 5.2-1 シリアルキラーの導入

SerialKiller は Java Regex（通常表現）ファイルを内蔵し、設定ファイルをユーザが独自に記述できるブラックリストがその一つである。すでにデフォルトの設定ファイルが存在し大まかな脆弱性をつくと判明している表現が含まれている。このデフォルト設定にはすでに有名なデシリアライズの脆弱性を活用する Ysoserial のペイロードが含まれている。Ysoserial とは Java の標準ライブラリの中で、特定の方法で使用された際にユーザが予期せぬ働きを行うものをまとめたガジェットのペイロードである。

SerialKiller の導入はオプションとして実装されている。なぜなら、階層型クラスローダを通ってきたオブジェクトが復元される直前に SerialKiller で余分に中身を確認するという手順が追加されたことによって若干のオーバヘッドが発生しているからである。このため、ユーザは信頼のおけるネットワーク上ではこれをオフにすることも可能だが、セキュリティを改善するために多少のオーバヘッドを被るかどうかを選ぶことができる。

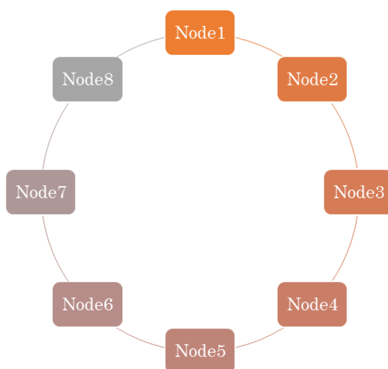
SerialKiller の他の機能としては、デシリアライズされるクラスの中身を控えるためのプロファイリングモードがあり、これにより普段デシリアライズされるであろうクラスをユーザが把握し、その中からブラックリストやホワイトリストを使うかどうかを選択することができる。SerialKiller を停止（AgentSphere を停止）させることなく設定ファイルを更新できるリフレッシュ機能も備わっている。信頼性が売りである AgentSphere にとってセキュリティアップデートでいったん機能を停止しなければならない場面を避ける。

図 5.2-1　左から右へと直列化されるデータのフロー

Copyright © 2018 by
The Institute of Electronics, Information and Communication Engineers and
Information Processing Society of Japan All rights reserved.

5.3 ネットワーク

AgentSphere の特徴の一つとして DHT が使用されている。DHT（分散ハッシュテーブル）とはハッシュテーブル（key、value）に似たような分散システムを提供するアルゴリズムである。エージェントたちをノードとして、key に対応した value を容易に取り出すことが出来る。各ノードはお互いの key と value を把握、つまりマッピングしていることから、この DHT の内部でノードの追加および削除が容易になっている。
DHT では各ノードの IP アドレスとそれに対応したデータの key を Hash 関数で変換させて空間をマッピングする。各ノードは空間上に割り当てられる。DHT の使用例としては peer-to-peer、DNS、ファイル分散システムなどが挙げられる。本研究室で開発している AgentSphere には DHT を基礎とするアルゴリズムの一つである Mercury を使用している。

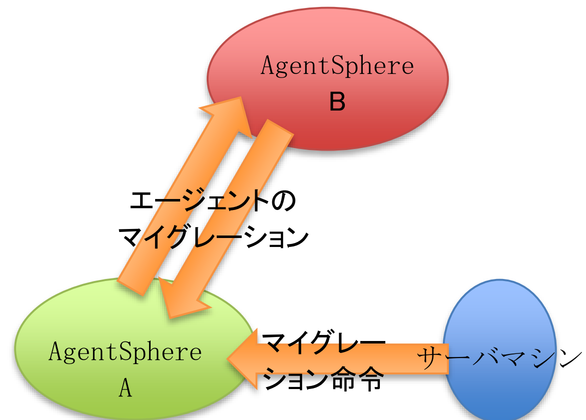


コード 5.3.-1. Mercury のネットワーク

Mercury とは広範囲で複数属性の探索を可能としているアルゴリズムである。Mercury は特徴として Hash 関数を使用せず、連続した値の順に配置する。これを使用することで範囲指定をし、データの検索が可能となる。さらに、Mercury はネットワークを構築し、ネットワーク上でデータの key を使用して管理する。

上記の DHT および Mercury アルゴリズムによって AgentSphere ではネットワークを通じてお互いの居場所およびメッセージを送りあうことができる。過去では常に他エージェントを総当たりで探すエージェントを常駐させる必要があったが、今では探さずにピンポイントで居場所に移動して処理を行うことが可能となっている。それを利用して AgentSphere では新たに Web ブラウザを介してエージェントをモニタリングし、それによって例えばエージェントの集中している AgentSphere からエージェントを移動させるなどのコントロールツールが作成されることになった。このツールの特徴としては使いやすく、なおかつスマートフォンやタブレットからも使用出来るような環境を開発した。

AgentSphere ではサーバ上で動作をする Java プログラムである Servlet [11] を用いて、Tomcat で構築されたサーバ上で実行することにした。図 5.3-2.では、AgenSphere をモニタリングする役目をもったサーバマシンが常駐し、そこから同ネットワーク内の AgentSphere に存在するエージェントらにマイグレーション命令や状態の監視が行えるところが表示されている。



コード 5.3.-2. サーバを介した通信

このモニタリングは AgentSphere らが構築している上記の Mercury DHT ネットワークがあって実現されている。DHT により現在の AgentSphere の数と、その中に在中しているエージェントを把握することが可能。
　ユーザに使いやすいことを掲げている AgentSphere にとってモニタリングツールは必要不可欠であったが、現状のモニタリングツールはローカルでしか起動させることができなかった。この新たなモニタリングツールを使用すればユーザはローカルマシンからだけではなくリモートのマシンから AgentSphere のネットワークを管理運営することが可能となった。

6. おわり

　AgentSphere は本論文で紹介した通り、専門的な環境や知識を有していないユーザが手軽に並列分散処理を行うことが可能なように開発されてきている。本論文で紹介された JSON を用いたシリアライズ・デシリアライズの方式によって AgentSphere はより軽く、柔軟にマイグレーションを行うことができた。著者らが進めてきた複数のマイグレーション方式についての考慮も行い、その中でも最適であったものが使用された
　また、JSON 形式は容易に Javascript や他言語に読み込まれるため、このシリアライズ方式で本来発生するサイズや再変換の手間を既存のシリアライザに存在する無駄をなくす形で補い、AgentSphere のエージェントが Java 専用ではなくより汎用で多目的に設計できるようになった。この汎用性をいかに利用するかどうかは今後の課題ではあるが、ユーザビリティを掲げる AgentSphere においては重要な要素である。

Copyright © 2018 by
The Institute of Electronics, Information and Communication Engineers and
Information Processing Society of Japan All rights reserved.

しかし、近年で主流となっているマシンの性能に左右されるディープラーニングも注目を浴びている。学習データの積み重ねなどは非常に重要な課題であるため、自由にネットワークを移動し高い信頼性をもったエージェントたちにそれらのデータの運用を任せると偶発的な事故を抑えることが可能となるという点に着目し、今後はディープラーニング用の環境作りおよび他言語間のマイグレーションについての可能性の研究も進める予定である。

参考文献

[1] **長塩征幸**、“DHT に基づく検索機能を用いた AgentMonitor の改良”、成蹊大学理工学部情報科ソフトウェア研究室学士論文、2012 年
[2] **赤井雄樹**、“JavaVM 上での非手続オブジェクト転送を可能とする直列化方式の構築”、成蹊大学理工学部情報科ソフトウェア研究室学士論文、2010 年
[3] **加藤 史彬,田久保 雅俊,櫻井 康樹,甲斐 宗徳**「コード変換による強マイグレーション化モバイルエージェントの実現」,FIT2007, B-024, Sep.2007
[4] **Java Reflection**, http://www.oracle.com/technetwork/articles/java/javareflection-1536171.html
[5] **Google-Gson**, https://github.com/Google-Gson
[6] **The JSON Data Interchange Format**,http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-404.pdf
[7] **Json for Office, VBA における JSON の利用**, http://www.jsonforoffice.com/json-javascript-object-notation/
[8] **Json.org**, http://json.org/
[9] **Java Security**, O’REILLY 出版, 1998 年, ISBN1-56592-403-7E
[10] **ikkisoft,**SerialKiller,https://github.com/ikkisoft/SerialKiller
[11] **宮本信二「基礎からのサーブレット/JSP 第 3 版」**、ISBN978-4-7973-5928-2、2011 年

Copyright © 2018 by
The Institute of Electronics, Information and Communication Engineers and
Information Processing Society of Japan All rights reserved.