成蹊大学理工学研究報告
J. Fac. Sci.Tech., Seikei Univ.
Vol.55 No.2 (2018)pp.1-6

モバイルエージェントシステム AgentSphere のためのマイグレーションおよびセキュリティに関するコアシステムの強化

ディダ ベサリ[*1], 甲斐 宗徳[*2]

Improvements in Security and Code Migration Related Core System Functionality for Mobile Agent System AgentSphere

Besar Dida[*1], Munenori Kai[*2]

ABSTRACT：We at the software laboratory at Seikei University are developing a platform that enables parallel and distributed processing in an autonomous manner. This platform is called AgentSphere and one of its main traits is a strong code migration which henceforth shall be referred to as "Strong Migration". Through this migration, autonomous code which is referred to as "Agent" migrate between the platform while retaining its previous run state. In this paper, we talk about two main improvements to this system. First, the current serialization was using unoptimized java serialization which was slow and had overhead. We solve this problem by implementing a customizable serializer which serializes in Json format and is much faster than the default one. Second, we close off security gaps created by the unorthodox usage of our Strong Migration which accepts unknown classes and deserializes them inside the runtime.

Keywords：parallel and distributed systems, mobile agent, strong migration, autonomous system, security

(Received November 30, 2018)

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。

[*1]：理工学専攻博士後期課程学生
[*2]：理工学専攻教授 (kai@st.seikei.ac.jp)

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．既存の直列化

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。

3．JavaFlowマイグレーション

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。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［2］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。

4．新たな直列化方式

本論文では直列化の問題点および今後のAgentSphereの拡張性を確保するために直列化方式に新たな手法の実装した。

直列化されたデータは従来のバイトデータではなくJsonデータベースファイルとして出力し、この時に用いるライブラリはGoogleのオープンソースgoogle-gson[1]を用いた。

4．1　新たな直列化方式の必要性

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 章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。

Json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。

4．2　Gsonとは

ここで用いるGoogleのgoogle-gsonはJavaオブジェクトのシリアライズをJson形式で行うオープンソースの中でもよりオーバーヘッドが少ない方式である。Json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の直列化では例外に対する対応が柔軟であり未知のクラスや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が専用のインスタンスクリエイタを記述することにより可能となった。直列化不可能なクラス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Serializableをインプリメントしていないため直列化できなかった。これらの場合は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。

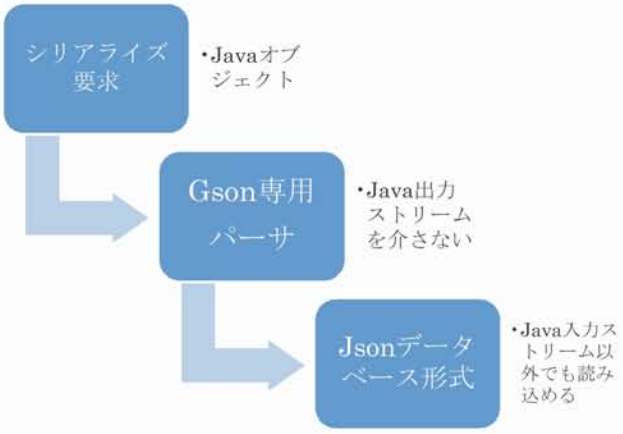

図 1．Gsonを用いた直列化フロー

Gsonでは既存のシリアライズと大まかな流れは同じだが、直列化と復元に関するところだけが変更されている。まず、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の型であるAbstractAgentが何なのかを学習させるために（Jsonとしてどのように出力させるために）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。

4．3　機能評価

AgentSphereを拡張する平均的な500行ぐらいの単純クラスで試してみた結果が下記の表の通りだった。

シリアライズ時：

Java Serialization	Google Gson
～103ms	～32ms

デシリアライズ時：

Java Serialization	Google Gson
～123ms	～30ms

異なるファイルサイズの場合の実験も行った。場合によっては、Continuationデータが莫大でありオーバーヘッドが余計にかかってしまうのではないかという問題も残っているため、それについても測定を行ってみた。ファイルサイズを1KBから開始し、１０倍ほど増加させつつ上記と同じ方法でマイグレーションを行った結果が下記のグラフのとおりとなった

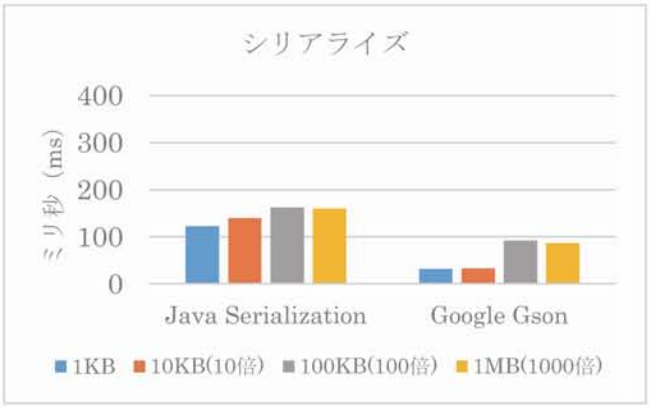

図 2.　1KB, 10KB, 100KB, 1MBのシリアライズ時間（実時間のみ）

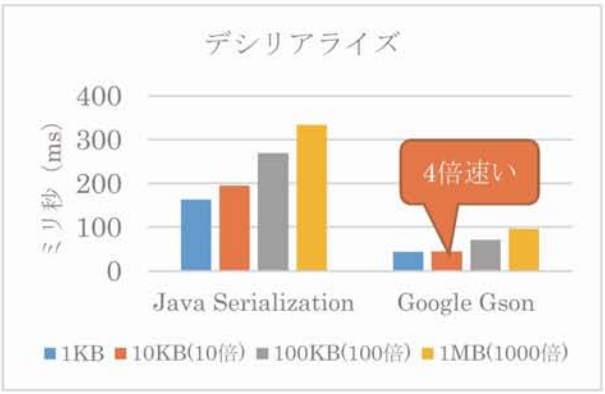

図 3.　1KB, 10KB, 100KB, 1MBのデシリアライズ時間（実時間のみ）

Google Gsonの導入により、AgentSphereの階層型クラスローダの一部が廃止された。本来、Javaでは未知のクラスが入ってきたときはそのクラスパスがないのでインスタンスを作成することが不可能となっている。AgentSphereではそれを階層型クラスローダで解消しており、未知のクラスに受け入れ元に対応するパスを取得し与えている。

しかし、Gsonのインスタンスクリエイターでは読み込まれたJson形式のオブジェクトをクラスとして開始できるが、JavaFlowとの互換性もあるため階層的クラスローダのJavaFlowのContinuationデータをアペンドする部分までは残っている状態である。よって、最終的にクラスローダ内では従来必要だった過程が不必要になった。ここではこのコンポーネントを削除せずに、場合によっては過去のエージェントのバックアップやデータの互換性が必要となるであろうと予想して、ユーザの設定により現状のまま使用できるように残してある。

5．先読みを使用した新たなセキュリティ方式

直列化されたデータは出力ストリームに渡すことで元に戻すことができる。直列化の場合では直列されたデータ内のオブジェクトが読み込まれる前にそのクラスの記述子が読み込まれる。この時点でJavaにすでに使用されているresolveClass[4]メソッドをオーバーロードし、未知のクラスが来た場合に処理を中止させることができる。現状のAgentSphereでは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を実装しているためクラスやメソッドの動的なアップデートが可能であるが、本研究で提唱する先読みを使用した方式はオプションとして存在し、動的アップデートを犠牲に強固なセキュリテ

ィを得ることができる。ここではこれを逆に利用し、ユーザが指定した特定のクラスや記述が検知された場合に読み込みを中止、直列データのロードも中止する。

図 4．左から右へ進む直列化データの復元

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[3]のデシリアライズライブラリを導入する。

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１．SerialKillerの導入

SerialKillerには様々な機能が備わっている。

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。

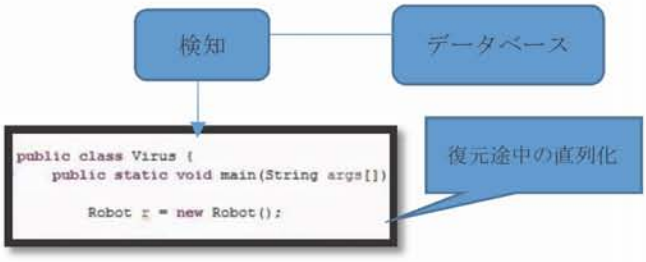

図 5．有害なクラスの検知

ブラックリストとは別にホワイトリストも含まれている。ブラックリストとは違い、ホワイトリストを見ているときのモードではホワイトリストの中身以外を例外で停止させることができる。しかし、エージェントとは多種多様で類似しているものはAbstractAgent内の記述のみであるため、ホワイトリストを用いることは困難である。

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。

4．1　先読みデータベースの実装

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ような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。これによりAgentSphereは基本的な脆弱性及び今後ユーザが定義していく脆弱性に対して耐性を持つことができた。

6．おわりに

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。

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] Google-Gson, https://github.com/google/gson

[2] Java Security, O'REILLY 出版, 1998 年, ISBN1-56592-403-7E

[3] ikkisoft,SerialKiller,https://github.com/ikkisoft/SerialKiller