Flyingware：バイトコード変換による安全なエージェントの実行

Flyingware: Secure Execution of Agents with a Bytecode Translator

大塚 紀子 † 千葉 滋 †† 新城 靖 ††† 板野 肯三 †††
Noriko OHTSUKA Shigeru CHIBA Yasushi SHINJO Kozo ITANO

† 筑波大学大学院理工学研究科
Master's Program in Science and Engineering, University of Tsukuba
†† 東京工業大学情報理工学研究科数理 · 計算科学専攻
Dept. of Mathematical and Computing Sciences, Tokyo Institute of Technology
††† 筑波大学電子 · 情報工学系
Institute of Information Sciences & Electronics, University of Tsukuba
{noriko,yas,itano}@hlla.is.tsukuba.ac.jp
chiba@is.titech.ac.jp

モバイルエージェントシステムにおいて実用的なアプリケーションを記述するためには, 実行に必要なファイルも一緒に移動させる必要がある. しかし, 移動先で安全のために OS や言語処理系がファイルへのアクセス制限をしている場合, エージェントは移動先にファイルを移動することができない. 本論文では, Java 言語で記述されたモバイルエージェントのバイトコードを, 仮想的にメモリ中に作られたファイルシステムをアクセスするように書き換えることで, この問題を解決する方法を提案する.

Mobile agents must often move together with external files, which are necessary for execution. However, if the operating system or the language runtime at the destination does not allow the agents to access a file system, the agents cannot take the files to the destination. We propose a solution of this problem for Java. With this solution, the bytecode of the agents is transformed so that the agents would access a file system virtually created on memory at the destination.

1 はじめに

近年, PC やサーバ機だけでなく PDA や携帯電話など, 様々な機器がネットワーク接続機能を持つようになってきている. そのようなネットワーク接続機能を持つ機器を効率的に利用するための技術として, モバイルエージェントシステムが注目を集めている. モバイルエージェントとは, ネットワーク上を移動し, 処理を行うことができるプログラムのことである. 従来のクライアントサーバモデルに基づく分散システムと比較して, モバイルエージェントシステムの利点は実行中にネットワーク通信を必要としないことである. たとえば, 無線ネットワークでは, 電波状況が悪い場所では通信が不能になったり, 通信が低速になってしまう. このような場合でも, エージェントは実行中に通信しないので快適に利用することができる.

我々は, Java 言語を対象としたモバイルエージェントシステム Flyingware [1, 2] を開発している. Flyingware の特徴は, エージェントの移動に電子メールを用いる点と, 実行に必要なクラスを自動的に抽出し, 一括して移動先へ転送する点である. しかし旧版の Flyingware では, 移動先に転送されるのはクラスだけであり, 画像ファイル等, 実行に必要なファイルを転送する機能はなかった. たとえば画像ファイルを読み込んだり, 計算結果を一時的にファイルに保存したりする場合など, モバイルエージェントと一緒にファイルを移動させたい要求がある. ファイルの移動を簡単に実現するには, ファイルを移動先のディスクにコピーすればよい. しかし, それでは移動先の OS や言語処理系が, モバイルエージェントによるファイルへのアクセスを, 安全のために禁止している場合, ファイルをコピーすることができない. このような

場合旧版の Flyingware では, アプリケーション開発者がプログラムを, ファイルを利用しないように書き換える必要があった. これはアプリケーション開発者の大きな負担になっていた.

本論文では, Flyingware に新たに追加した, 指定された画像ファイル等をエージェントと一緒に転送する方法について述べる. この方法では, 仮想ディスクという概念を用いて, 移動先の安全性を保ったまま, ファイルを転送する.

仮想ディスクとは, 仮想的にメモリ中に作られたファイルシステムのことである. 本論文で提案する方法では, 通常の API を使ってファイルをアクセスするプログラム（バイトコード）を, 仮想ディスクだけをアクセスするように自動的に変換する. また, **実ディスク**（実際のディスク上のファイルシステム）を元にメモリ中に仮想ディスクを自動的に作成する. 以後, エージェントが移動する際には, その仮想ディスクを一緒に転送する.

この方法の利点は, 第 1 に移動先が安全のため, モバイルエージェントによる実ディスクへのアクセスを禁止していたとしても, モバイルエージェントは一緒に移動したファイルを利用できることである. 第 2 に, 開発者は仮想ディスクに関する知識なしに, 安全にファイルにアクセスするモバイルエージェントのプログラムを作成できることである. また, 変換されたモバイルエージェントは実ディスクに一切アクセスしないので, 携帯端末など, 実ディスクをもたない端末上に移動しても動作することができる.

2 Flyingware

Flyingware は, 実行時に必要なクラスを自動的に抽出し, それを電子メールに添付させて転送するモバイルエージェントシステムである. 従来のエージェントシステムは, エージェント独自のプラットフォームをインストールする必要があるものが多い [3, 4]. また初めから端末に組み込むもの [5] もあるが, これは組み込まれていない機器では利用できない. このようなエージェントシステムと比較して, Flyingware の特徴は, 特別なプラットフォームをインストールする必要はなく, 電子メールが送受信でき, Java 言語が実行できる環境であれば利用できることである. 要求に応じてクラスを転送させるエージェントシステムと比較して, Flyingware の利点は, クラスをまとめて一度に移動させるので, 実行時には通信機能を一切必要としないことである.

2.1 アプリケーション例

Flyingware のアプリケーション例として, 高い表現力を持った電子メールがある. 従来の電子メールは, 個人化されたデータを個人宛に直接送ることができ, また閲覧中には通信機能が不要であるという利点があるが, 対話的な処理ができないなど表現能力が低いという欠点がある. World Wide Web はその逆の特性がある. Flyingware を用いれば, 両者の利点をあわせて実現できる.

2.2 従来の Flyingware の問題点

Flyingware には実行に必要なバイトコードを自動的に抽出して送信する機能がある. 以下に Flyingware のアプリケーションの例を示す.

```
public static void main(...) {
        :
    Foo foo = new Foo(...);
    SmtpRoundTrip flight =
                new SmtpRoundTrip(...);
    flight.fly(foo, "resume");
}
```

`fly()` は, 移動する時に呼び出すメソッドである. その結果として, 引数で渡されたオブジェクトの指定された名前のメソッドが, 到着後に実行される. `fly()` では, 引数として受け取ったオブジェクトからたどることができるクラスを解析し, 実行に必要な全てのクラスを得る. そしてそれらのクラスをまとめ, 指定されたホストに転送する.

従来の Flyingware はファイルを移動させる機能を提供していなかった. そのため開発者は, ファイルをアクセスするプログラムを手動でファイルをアクセスしないプログラムに書き換える必要があった. 具体的には, 例えば移動前にファイルの内容をメモリ中の byte 配列にコピーし, 移動後は, ファイルではなくメモリ中の byte 配列を使うようにプログラムを書き換える必要があった. たくさんのファイルにアクセスするアプリケーションでは, 開発者の負担になっていた.

3 バイトコード変換と仮想ディスクによる安全なエージェントの実行

前章で述べた問題をバイトコード変換と仮想ディスクを使って解決する. **仮想ディスク**とは仮想的にメモリ中に作られたファイルシステムである. 仮想ディスクは, 実際のディスク上のファイルシステム（以

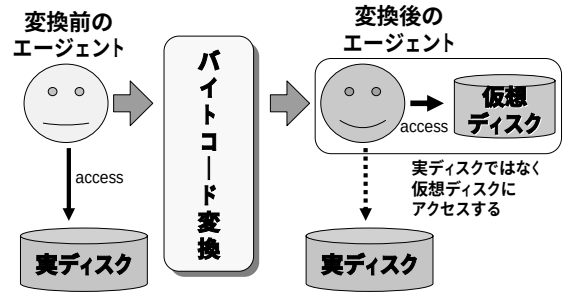

図 1: 仮想ディスクへのアクセス

後**実ディスク**と呼ぶ）と同じ API でアクセスできる（図 1 参照）.

この方法では, 各アプリケーション開発者は, まず通常の API を使って実ディスク上のファイルにアクセスするプログラムを記述し, コンパイルしてバイトコードを得る. そのバイトコードをその仮想ディスクだけをアクセスするように変換する. 変換したプログラムを実行し, 初めにファイルにアクセスした時, または移動する時に, 実ディスク上のファイルを元に仮想ディスクを自動的に生成する. 以後, エージェントが移動する際には, その仮想ディスクも一緒に移動するようにする. これ以降は全てのファイルアクセスは仮想ディスクに対してのみ行われる.

このようにして, 移動先が安全のために実ディスクにアクセスすることを禁止していても, ファイルにアクセスするモバイルエージェントは動き続けることができる. さらに, 開発者は仮想ディスクに関する知識なしに, モバイルエージェントのプログラムを作成することができる. また, 作成したモバイルエージェントは実ディスクに一切アクセスしないので, 携帯端末など, 実ディスクをもたない端末上に移動しても動作することができる.

この方法では 2 つのソフトウェアを用いる. ひとつはバイトコード変換を行うプログラムで, もうひとつは仮想ディスクを実現するクラスライブラリである.

3.1 バイトコード変換

実ディスク上のファイルにアクセスするプログラムを, 仮想ディスク上のファイルにしかアクセスしないように変更するために, 本研究では Javassist[6] を利用する. Javassist は, Java のリフレクション API を拡張してプログラムの振る舞いを変更できるようにするバイトコード変換器である. Javassist では, バイトコードレベルで変換するので, ソースがなくても利用することができるという利点がある. さらにソースコードを変換するよりも処理速度も速くなる.

実際に, 実ディスク上のファイルにアクセスするアプリケーションプログラムをどのように変更したらよいかを示す.

Java は, ファイルの入出力を行うための基本的なクラスとして `FileInputStream` と `FileOutputStrem` を持っている. `FileInputStream` クラスを利用して, データの読み取りを行うプログラムの例を, 以下に示す[1].

```
/* 変換前のプログラム */
FileInputStream fin =
        new FileInputStream(ファイル名);
fin.read();
  :
```

このプログラムは, まず `FileInputStram` クラスのオブジェクトを生成している. そしてそのオブジェクトの `read()` メソッドにより, データを読み込んでいる. このプログラムを以下のように変換する[2].

```
/* 変換後のプログラム */
ByteArrayInputStream fin =
        VirtualDiskIn.factory(ファイル名);
fin.read();
  :
```

変更する部分はオブジェクトを生成する式だけであり, その後の `fin.read()` のようなメソッド呼び出しは変更する必要はない. `VirtualDiskIn` クラスは, 仮想ディスクにアクセスするためのクラスライブラリである. 詳しくは 3.2 節で述べる.

`FileInputStream` の他に対応すべきクラスとしては, ファイルの書き込みを行う `FileOutputStream` や, `ImageIcon` などがある. これらのクラスについても, `FielnputStream` と同様にそれぞれ専用のクラスライブラリを呼び出すように変換する.

Java の標準クラスには, 内部的に `FileInputStream` オブジェクトを生成し, ファイルにアクセスするものもある. 仮想ディスクにアクセスするように, そのような標準クラス中の `FileInputStream` の利用を全て `VirtualDiskIn` の利用に変換する方法も考えられる. しかし, これには移動先の標準クラスをバイトコード変換したもの

[1]説明のためにソースプログラム形式で示しているが, 実際にはバイトコードレベルで変換される.

[2]実際には `ByteArrayInputStream` ではなく `FileInputStream` と同じメソッドを定義したクラスに変換する.

に置き換える必要が生じる．それでは, 2 章で述べたように特別なプラットフォームを必要しないというFlyingware の特徴が失われてしまう．そこで本研究では, ファイルアクセスを行う標準クラスについては, 内部を書き換えずに, それぞれ専用のクラスライブラリを呼び出すように変換する．

3.2 仮想ディスクを実現するクラスライブラリ

FileInputStream のための仮想ディスクを実現するクラスライブラリの概要を以下に示す．

```
public class VirtualDiskIn {
    static Hashtable files;

    public static ByteArrayInputStream
                factory(String fileName) {
       if(files != null) {
           makeVD();
       }
       byte b[] = (byte[])files.get(fileName);
       return new
           ByteArrayInputStream(b);
    }

    public static void makeVD(){
       files = new Hashtable();
       fileNameLists = getFileNameList();
       while(fileNameLists.available()) {
          fileName = fileNameLists.getName();
          byte b[] = read(fileName);
          files.put(fileName, b);
       }
    }
    :
}
```

このクラスは，重要な変数としてハッシュ表の files, 及び重要なメソッド factory() と makeVD() を持っている．

factory() は, 3.1 節で述べたように変換後のプログラムに現れるメソッドである．引数として与えられたファイル名を元に files を検索し, 対応する byte 配列を得る．files が null の場合は, 仮想ディスクが生成されていないことを意味する．その場合は makeVD() メソッドを呼び出して, 仮想ディスクを生成する．得られた byte 配列から Java の標準クラスライブラリである ByteArrayInputStream クラスのオブジェクトを生成し, それを factory() メソッドの返り値とする．ByteArrayInputStream クラスは, byte 配列に内容を保持している, FileInputStream とほぼ同等のメソッドを持つクラスである．

makeVD() は, 仮想ディスクを生成するメソッドである．このメソッドはまずハッシュ表を生成し, ファイル名のリストを得る．このリストを作成する方法としては, 例えば, あるディレクトリの中にあるファイルを検索する方法が考えられる．リストの全てのファイル名について, そのファイル名が示すファイルの内容を byte 配列に格納する．そしてこの byte 配列をハッシュ表に挿入する．

4 関連研究

安全性を保ったままモバイルエージェントにファイルへのアクセスを許す研究は少なくない．例えば SoftwarePot[7] では, 特別なミドルウェアにより, モバイルエージェントが全ディレクトリ木中の特定の領域にしかアクセスできないように制限している．一方, 本研究の方法は, バイトコード変換によって安全なコードを生成するので, 移動先のマシンに特別な安全性を確保するソフトウェアが必要ない．

5 おわりに

本稿では, 移動先でエージェントを安全に実行するために, 実ディスクを使うプログラムは仮想ディスクを使うように自動的に変換する方法を提案した．さらに, 仮想ディスクを実現するクラスライブラリの設計を示した． 今後は, 提案したバイトコード変換を行うプログラムと, クラスライブラリを実装していく．

参考文献

[1] Shigeru Chiba, Flyingware: An Email-based Mobile Agent System, OOPSLA Workshop on Experiences with Autonomous Mobile Objects and Agent Based Systems, 2000.

[2] 和田慎也, 千葉滋, 板野肯三, Flyingware: Java によって表現力を強化した電子メール, 日本ソフトウェア科学会第 17 回大会講演論文集, 2000.

[3] Ichiro Satoh, A Mobile Agent-based Framework for Active Networks, In Proc. IEEE Systems, Man, and Cybernetics Conference, pp.71-76, 1999.

[4] IBM, Aglets, http://www.trl.ibm.com/aglets/

[5] OMRON Business Development Group, Jumon: Agent Based Distributed Middleware, 2001.

[6] Shigeru Chiba, Load-time Structural Reflection in Java, ECOOP 2000 – Object-Oriented Programming, Springer LNCS 1850, pp. 313-336, 2000.

[7] Kazuhiko Kato, Yoshihiro Oyama, Katsunori Kanda, and Katsuya Matsubara, Software Circulation using Sandboxed File Space – Previous Experience and New Approach, In Proc. 8th ECOOP Workshop on Mobile Object Systems, 2002.