安全なモバイルエージェントシステム Flyingware のための仮想ディスクの実現

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
†† 東京工業大学情報理工学研究科数理 · 計算科学専攻
††† 筑波大学電子 · 情報工学系
{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. モバイルエージェントシステム Flyingware では, Java 言語で記述されたモバイルエージェントの中に, 仮想ディスクを構築することでこの問題を解決している. 本論文では, その仮想ディスクの実現について述べる.

1 はじめに

近年, PC やサーバ機だけでなく PDA や携電話など, 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.

そこで Flyingware では仮想ディスクという概念を用い, 安全にファイルへのアクセスを許している. **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. Flyingware 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 [2]. 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 モバイルエージェントが実行時に必要なファイルにアクセスする際, 移動先の実ディスクにはアクセスしないため, 移動先の実ディスクは安全である. 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．

本論文では, 安全なモバイルエージェントシステム Flyingware のための仮想ディスクの実現について述べる．その実装の特徴は, Java 言語のストリームを利用した入出力クラス内の 4 つの基本クラスを仮想ディスクに対応させることで, ストリームを利用した入出力クラスの全てに対応していることである．また, 変換するのは個々の入出力処理に対して行うのではなく, ストリームの生成の部分だけでよい．

2 Flyingware

Flyingware は, 実行時に必要なバイトコードを自動的に抽出し, 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．

以下に Flyingware の利用例を示す．

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

これだけのコードを書けば, ローカルのみで実行可能なアプリケーションはネットワーク上を移動するアプリケーションになる．

`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, 移動後に実行される．`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（バイトコード）をまとめ, 指定されたホストに転送する．

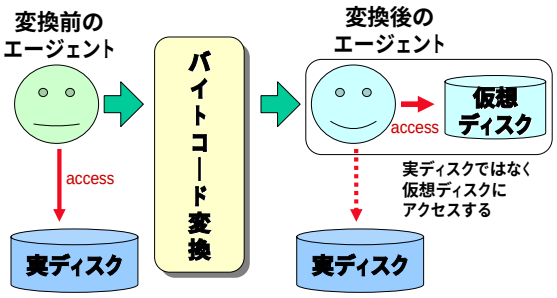

図 1: 仮想ディスクへのアクセス

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．

3 仮想ディスクによる安全なエージェントシステムの実現

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

本研究は, 仮想ディスクにアクセスするようにエージェントのプログラムをコンパイル時にバイトコードレベルで変換する方式を採用している．変換を起動時に行うことにより, 移動先の環境によって仮想ディスクと実ディスクを切り替えて動作する方法も可能であるが, それでは変換器も移動させなくてはならない．移動させるコードはなるべく少なくしたいので, 本研究ではコンパイル時で変換している．こ

の結果, エージェントの開発者だけが変換のための特別なソフトウェアをもっていればよく, エージェントを実行するマシン上に特別なミドルウェア等は必要ない. 必要なのは, 開発者が正しくプログラム変換を行ったかどうかを検査する検査器だけである. この検査器には, 標準の Java 仮想機械に付属しているアクセス制御機構を使うことができる.

本研究を利用することで, 万一開発者に悪意があり, エージェントが実ディスクにアクセスしようとしても, 例外を発生して実行をとめることができる. 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.

3.1 仮想ディスクを実現する際の問題点

Flyingware を利用したアプリケーションは Java 言語で記述される. したがって, アプリケーションが実ディスク上のファイルにアクセスする場合, Java の入出力用のクラスを利用する. Java には, 各種入出力用のクラスが `java.io` パッケージ内にある. **図 2** にパッケージ内の入出力クラスを示す.

Flyingware を利用したアプリケーションは, 図 2 のファイル入出力クラスの他に, 画像を表示するためのクラスである `javax.swing.ImageIcon` を使うと想定される. これらのクラス全てが, 仮想ディスク上のファイルにアクセスできるようにする必要がある.

しかし, この膨大な量のクラスひとつひとつに対して, 仮想ディスクのためのクラスを作成するのには大変な労力を要する.

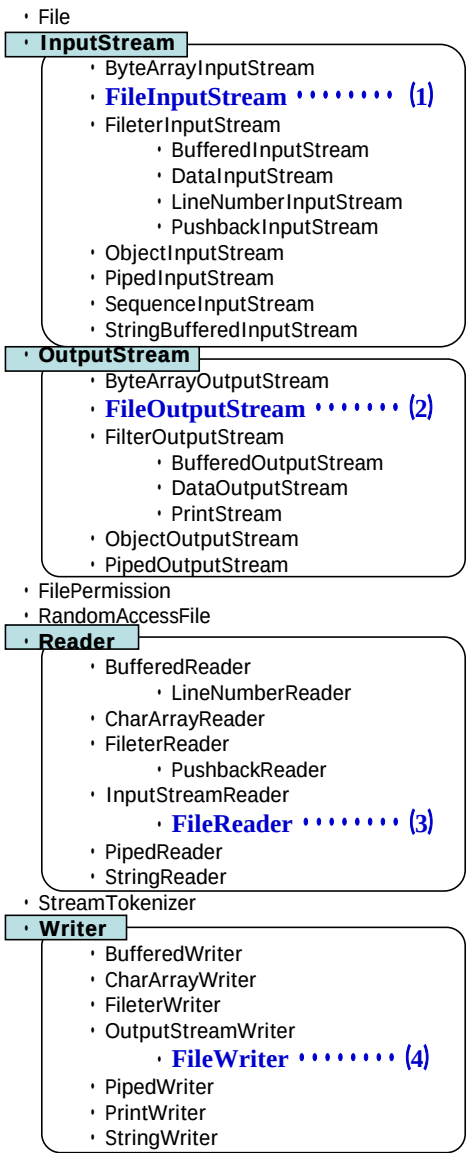

図 2: java.io パッケージ内の入出力クラス

3.2 仮想ディスクの実現方法

図 2 のように, Java には膨大な数のファイル入出力を行うクラスが存在する. しかし, ファイル入出力の 4 つの基本クラスに対して専用のクラスライブラリを作成するだけで, ストリームに基づき入出力を行うほとんどのクラスに対応することがわかった. その 4 つの基本クラスとは, バイナリ・データ入力用の `FileInputStream`, バイナリ・データ出力用の `FileOutputStream`, テキストの入力用の `FileReader`, テキストの出力用の `FileWriter` である. 以下にその理由を示す.

ファイル入出力のクラス群は, 大きく分けて 4 つのグループに分類できる. そのグループを, 図 2 では 4 つの枠組みで表している. それぞれのグループは, 共通の抽象クラスをスーパークラスとして持っている. ファイル入出力クラスは, ストリームを引数に受け取るクラスと, ストリーム以外（ファイル名など）のオブジェクトを引数に受け取るクラスに大別できる. ストリームを引数に受け取るクラスの読み込み・書き込み先は, 受け取ったストリームの接続先に依存する. そのため, 直接実ディスク上のファイルにアクセスするクラスは, ストリーム以外の引数を受け取るクラスということになる. ストリーム以外の引数を受け取るクラスはいくつか存在するが, その中でファイル名を引数に取るクラスは, 各グループ内でひとつだけ存在する. つまり, 直接ファイルにアクセスする

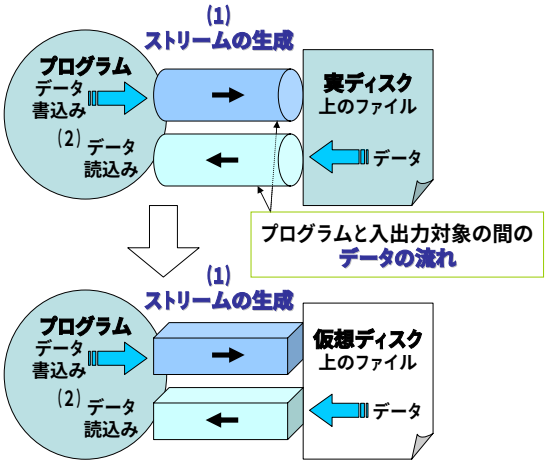

図 3: ファイル入出力

クラスは各グループにひとつしかない．そのクラスが，4 つの**基本クラス**である．図 2 では太字で示している．

4 つのグループのうち，InputStream グループを使ったファイル入力の例を以下に示す．

```
// ファイル入力例
byte[] b;
DataInputStream din = new DataInputStream(
          new FileInputStream(ファイル名));
din.read(b);
```

このように，実際にファイル名を引数に受け取るのは FileInputStream である．DataInputStream は，FileInputStream のような InputStream 型のストリームオブジェクトをコンストラクタの引数に取る．そのため，引数のストリームが仮想ディスク上のファイルに対応していた場合，DataInputStream も仮想ディスクにアクセスすることになる．このことは先ほど説明した基本となる 4 つのクラス全てに当てはまる．したがって，この 4 つの基本クラスについて仮想ディスク対応クラスライブラリを作成することで，4 つのグループ内のクラス全てに対応することができる．

Java 言語で記述されたプログラムで，ストリームを利用したファイル入出力の大まかなステップは，次の 2 つである．

(1) 目的に応じたストリーム（オブジェクト）を生成

(2) ストリームに対して入出力を実行

これら全てのファイル入出力処理を，仮想ディスクに対応するように変更することは，大変手間がかかる．この問題を，Java のオブジェクト指向の性質を利用して以下のように解決する．

(1) で，オブジェクトのストリームが実ディスク上のファイルに対応していた場合，これを仮想ディスク上のファイルに対するストリームに変更する．**図 3** では，ストリームオブジェクトが円形から角形に変更されている部分に相当する．接続先が実ディスク上のファイルから仮想ディスク上のファイルに変更されても，入出力の実行はそのまま利用できる．したがって，変更するのは (1) のストリームの生成だけで，(2) の read() や write() などの入出力の実行は変更せずそのまま利用できる．

前述したように，4 つの基本クラスは，実ディスク上のファイルへのストリームを生成する役割を持っている．そこで，これら基本クラスと同じ仕様で，仮想ディスク上のファイルへのストリームを生成する役割を持つクラスライブラリを作成する．このクラスライブラリを利用すれば，4 つのグループ内の全てのクラスが仮想ディスク上のファイルにアクセス可能になる．

4 仮想ディスクの実装

4.1 クラスライブラリ

クラスライブラリには，以下のような主要クラスがある．コード量はおよそ 350 行（約 9500byte）である．

- *VirtualDisk*
  仮想ディスク上のファイル管理をするクラス
- *VDInputStream*
  仮想ディスク上での FileInputStream の代替クラス
- *VDOutputStream*
  仮想ディスク上での OutputStream の代替クラス
- *VDReader*
  仮想ディスク上での FileReader の代替クラス
- *VDWriter*
  仮想ディスク上での FileWriter の代替クラス
- *VDImageIcon*
  仮想ディスク上での ImageIcon の代替クラス

各クラスの概要を以下に示す．

- *VirtualDisk*

```
// VirtualDisk クラス
public class VirtualDisk {
    HashMap map;

    // コンストラクタ
```

```
    public VirtualDisk() {
        map = new HashMap();
        loadData();
    }

    // ファイルの取り出し
    public byte[] getFile(String name) {
        return (byte[]) map.get(name);
    }

    // VDOutputStream からの書き込み処理
    public void putWrite(String name, byte[] buff) {
        map.put(name, buff);
    }
    :
}
```

VirtualDisk は, 仮想ディスク上のファイル管理を行うクラスである. byte 配列に変換した, アプリケーション実行時に必要なファイルを, ハッシュ表 *map* に格納している. 仮想ディスクにコピーすべき実ディスク上のファイルは, 現在の所, ある決まった（変更可能な）ディレクトリ内に全てあるものとする. エージェントが移動する際に, *AgentStart* クラス（後述）内で *VirtualDisk* クラスのオブジェクトが生成さる. これが仮想ディスク上にファイルが構築されることに相当する. このファイルの構築は, *VirtualDisk* クラスのコンストラクタで行われる. *loadData()* は, 指定されたディレクトリ内にあるファイルを全て byte 配列に変換し, *map* に格納するメソッドである. このようにして仮想ディスク上にファイルが構築され, Flyinware を使って移動する際には, エージェントと一緒にこの仮想ディスク（ byte 配列）も移動する.

移動後は, ファイルアクセスは仮想ディスク上のファイルに対して行われるようにしなければならない. 例えば, ストリームへの入力が行われると, *getFile()* が呼ばれる. これは, ハッシュ表 *map* から目的のファイルを返すメソッドである. *putWrite()* は, ストリームへの書き込みが行われる場合に, ファイルの内容をハッシュ表 *map* に保存するためのメソッドである.

AgentStart クラスは, アプリケーションから呼び出される唯一のクラスで, Flyingware におけるエージェントの移動はこのクラスが行う. *AgentStart* クラスは, アプリケーションと仮想ディスクの 2 つのオブジェクトを持っている. アプリケーションから移動の要求があると, この両方のオブジェクトをひとつの jar ファイルにまとめ, 電子メールに添付して送信する.

AgentStart は, 仮想ディスクを *VirtualDisk* クラスのオブジェクト *vd* としてフィールドに持っている. 仮想ディスク上のファイルにアクセスする際には, この *AgentStart.vd* を呼ぶ.

- *VDInputStream*

```
public class VDInputStream
                extends ByteArrayInputStream
    // 各コンストラクタ
    public VDInputStream(byte[] data) {
        super(data);
    }
    public VDInputStream(String name) {
        this(AgentStart.vd.getFile(name));
    }
    public VDInputStream(File file) {
        this(file.getName());
    }
    :
}
```

VDInputStream は, `FileInputStream` の代わりにファイル入力を行うクラスである. そのため, *VirtualDisk* の byte 配列に対するストリームを生成する. *VDInputStream* は, `ByteArrayInputStream` をスーパークラスに持ち, `FileInputStream` と同様の振る舞いをするクラスである. つまり, *VirtualDisk* の byte 配列から入力処理を行うクラスである.

byte[] を引数にとるコンストラクタは, スーパークラスである `ByteArrayOutputStream` のコンストラクタを呼び出す. String を引数にとるコンストラクタは, まずそのファイル名をもとに *VirtualDisk* クラスの *getFile()* を使って仮想ディスクから byte 配列を得る. そしてその byte[] を自分のクラスのコンストラクタに渡す. File を引数にとるコンストラクタは, その File オブジェクトから String のファイル名を取り出し, String を引数にとる自分のコンストラクタに渡す.

- *VDOutputStream*

```
public class VDOutputStream
                extends ByteArrayOutputStream
    protected String filename;

    // コンストラクタ
    public VDOutputStream(String name) {
        super();
        filename = name;
    }

    // 終了する際に，仮想ディスクに書き込む
```

```
    public void close() {
        byte[] data = new byte[this.count];
        for(int i = 0; i < data.length; i++) {
            data[i] = this.buf[i];
        }
        AgentStart.vd.putWrite(filename, data);
        super.close();
    }
    :
}
```

VDOutputStream は, FileOutputStream の代わりに, ファイル出力を行うクラスである. 書き込み先のファイル名は, コンストラクタで *filename* フィールドに保持しておく. *VDOutputStream* は, ByteArrayOutputStream をスーパークラスに持ち, FileOutputStream と同じ振る舞いをするクラスである. つまり, *VirtualDisk* の byte 配列に書き込み処理を行うクラスである.

書き込みは読み込みと同様にストリームを生成することに加えて, 仮想ディスク上にその処理を反映させなければならない. 書き込みが行われると, スーパークラス ByteArrayOutputStream のフィールドである, byte 配列 buf に, その内容が書き込まれていく. 基本的には *VDOutputStream* の *write()* が呼ばれるが, 書き込みの種類によっては呼ばれないこともある. *write()* が呼ばれなくても, buf には内容が書き込まれている. そのため close() が呼ばれたときに, *VirtualDisk* クラスの *putWrite()* を呼び, *VirtualDisk* のハッシュ表 *map* に対して書き込み *map.put()* を行う. 書き込みは, buf の有効バイト数 count 分に対して行う.

- *VDReader*

```
public class VDReader extends InputStreamReader
    // コンストラクタ
    public VDReader(String name) {
        super(new ByteArrayInputStream(
                  AgentStart.vd.getFile(name)));
    }
    :
}
```

VDReader は, FileReader の代わりにファイル入力を行うクラスである. 上記の *VDInputStream* と同様に, ByteArrayInputStream を利用して, *VirtualDisk* の byte 配列に対するストリームを生成する. Reader ストリームに変換するため, Java 標準のクラスライブラリ InputStreamReader を利用する.

ここでは *VDInputStream* ではなく, ByteArrayInputStream を用いる. *VDInputStream* は, ByteArrayInputStream を FileInputStream 用に拡張したクラスである. ここで必要なのは, byte 配列から入力ストリームを生成する機能だけで, FileInputStream 用に拡張した機能は必要はない. そのため, *VDInputStream* ではなく ByteArrayInputStream を用いる.

- *VDWriter*

```
public class VDWriter extends OutputStreamWriter
    // コンストラクタ
    public VDWriter(String name) {
        super(new VDOutputStream(name));
    }
    :
}
```

VDWriter は, FileWriter の代わりにファイル出力を行うクラスである. 上記の *VDOutputStream* から生成されたストリームを, Java 標準のクラスライブラリ OutputStreamWriter を使って Writer ストリームに変換するクラスである. 仮想ディスク上への書き込みは, *VDOutputStream* で行われる.

- *VDImageIcon*

```
public class VDImageIcon extends ImageIcon
    // コンストラクタ
    public VDImageIcon(String name) {
        super(AgentStart.vd.getFile(name));
    }
}
```

VDImageIcon は, 仮想ディスク上の画像ファイルから描画を行うクラスである. Java 標準のクラス ImageIcon には byte 配列（画像データ）から画像を描画するコンストラクタが存在する. *VDImageIcon* では, 与えられたファイル名をもとに, 画像データを *VirtualDisk* の *map* から探し, 得られた byte 配列から画像を描画させている.

図 2 で示したクラスの中で現在対応できていないのは以下のクラスのみである.

- File
- FilePermission
- RandomAccessFile

これらについてはこれから対応していく予定である.

4.2 バイトコード変換

図 3 で説明した (1) のストリームの生成を変更する処理は, バイトコード変換を利用して自動的に行う. 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.

Javassist の利用例を以下に示す.

```
import javassist.*;

ClassPool cp = ClassPool.getDefault();
CtClass cc = cp.get("file");
cc.replaceClassName("java.io.FileInputStream",
                            "VDInputStream");
cc.writeFile();
```

cp.get("file") で指定されたファイルに対して, バイトコード変換を行う. *cc.replaceClassName()* で `FileInputStream` は, *VDInputStream* に変更される. 最後に *cc.writeFile()* で書き戻す.

Javassist を利用することで, リファクタリング等のツールを利用し, 手動で変換する必要はなく, 必要な変更は全て自動的に行われる.

5 実験

5.1 アプリケーション

本研究を, アプリケーションを使って実行し, 実験を行った. そのアプリケーションの実行図を**図 4** に示す. これは, 顧客に商品カタログを送り, 注文をとってくるアプリケーションである. 実行時における画像や商品データの読み込み, また顧客の注文の書き込みなどは, 全て仮想ディスク上で行われる. 以下にこのアプリケーションが実際に書き換えられるコードを示す[1].

```
BufferedReader in = new BufferedReader(
                    new FileReader("data.txt"));
String line = in.readLine();
    :
```

これは, 商品データを読み込むプログラムの一部である. まず入力ストリームを生成し, readLine() で入力を実行し, data.txt から 1 行読み込んでいる. このプログラムを以下のように変更する.

```
BufferedReader in = new BufferedReader(
                    new VDReader("data.txt"));
String line = in.readLine();
    :
```

3.2 節で説明したように, 変更するのは基本クラスの `FileInputStream` だけで, `BufferedReader` は変

[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.

図 4: アプリケーションの例

更しない. またその基本クラスも, 変更するのはストリームの生成だけで, readLine() のような入力実行は変更しなくてよい.

5.2 安全性

Java 標準のアクセス制御機能を利用し, ファイルアクセスを禁止した状態で, このアプリケーションを実行した. 正常に実行できたことを確認した.

5.3 移動するコード量

移動するコード量を測定した. 実際に移動するのは agent.jar で, その中には**表 1** に示す内容が含まれている.

ディレクトリ	単位：byte
agent.jar	548,584
META-INF/	52
flyingware/	31,430
memory/	8,415
appli/	38,875
OBJECT.IMG	539,827

表 1: 注文前後の各ファイル（ディレクトリ）サイズ

META-INF ディレクトリ内にあるのは, jar ファイルを展開するための情報である. flyingware , memory ディレクトリは, それぞれ移動するために必要な機能, 仮想ディスクのクラスライブラリである. これらは, アプリケーションが利用しているクラスだけが含まれている. appli ディレクトリ内にあるのは, アプリケーションのクラスである. OBJECT.IMG は, 移動するオブジェクトなどの情報に加え, 仮想ディスクの情報が入っている. そのため, データ量が膨大になってしまう. 特に, このアプリケーションは, **表 2** に示すように, 画像を多く利用しているので, データ量が大きくなる.

データの種類	単位：byte
画像	532,846
商品データ	2,165

表 2: アプリケーション実行時に必要なファイル

6 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.

7 おわりに

我々は, 安全なモバイルエージェントシステム Flyingware のための仮想ディスクを実現した. 仮想ディスクの利用は, バイトコード変換を利用して自動的に行われるので, アプリケーション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, 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動作することができる.

本論文では, 仮想ディスクについて述べた. その実装の特徴は, Java 言語のストリームを利用した入出力クラス内の 4 つの基本クラスを仮想ディスクに対応させることで, ストリームを利用した入出力クラスの全てに対応していることである. また, 変換するのは個々の入出力処理に対して行うのではなく, ストリームの生成の部分だけでよい.

今後は, ストリーム以外の入出力ラスを仮想ディスクに対応させる.

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] 大塚紀子, 千葉滋, 新城靖, 板野肯三, Flyingware: バイトコード変換による安全なエージェントの実行, 日本ソフトウェア科学会第 19 回大会, 2002.

[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.