平成１３年度学士論文

不要なデータを捨てながら移動するモバイルエージェント

東京工業大学 理学部 情報科学科
学籍番号 96-0922-8
木村 信真

指導教官
千葉 滋 講師

平成14年2月7日

概要

本稿は、Java 言語で設計されたモバイルエージェントシステム Flyingware を提案する。モバイルエージェントとは、計算機間を自律的に移動することのできるプログラムである。プログラムの状態を保持して移動して、移動先の計算機で移動元で行っていた作業の続きを再開するため、ネットワークに接続するのはモバイルエージェントの移動時だけでよく、常時接続の必要がない。しかし、既存のモバイルエージェントの多くは、ネットワーク常時接続の環境を想定された設計になっていて、応用範囲も限定されてしまっている。我々の提案する Flyingware は、それらのモバイルエージェントと違い、ネットワークに非常時接続の環境でも使用可能なモバイルエージェントシステムである。Flyingware は、Java 言語で設計されており、自分自身を電子メールとして移動するため、実行環境を選ばない。さらに、通信コストを極力抑えるために、不要になったクラスを捨てながら移動できるように設計されている。既存のモバイルエージェントで、ネットワークに非常時接続環境で使用可能な物の欠点は、不要なデータをも移動してしまうことにあった。Flyingware では、移動時に自分自身の .class ファイルを自動的に解析することにより、移動先で必要なデータだけを自動的に収集することが出来る。しかし、従来の Flyingware では、自動的に移動できるのは .class ファイルとその時点で存在するオブジェクトだけである。もし画像ファイルや音声ファイルなどのデータファイルを移動しようとするならば、ユーザがプログラム中にそのことを明示的に記述しなければならない。また、それらのデータが不要になったら、そのこともユーザが明示的に記述しなければならない。これではユーザの負担が増えてしまう。

そこで、本稿では、Flyingware を拡張し、画像ファイルや音声ファイルなども自動的に移動し、不要になった時点で自動的に捨てていく方法を提案する。移動時に自分自身を自動的に解析する機能を拡張し、移動先で必要なクラスの識別だけでなく、.gif ファイルなどのファイルも識別

して移動できるようにした。また、従来の Flyingware はオブジェクトの解析を行わないため、移動先で必要なクラスの探し忘れが存在したので、オブジェクトの解析を行うようにし、クラスの探し忘れをなくすようにした。このことにより、Flyingware は、以前よりも多くのデータを、自動的に移動し、不要になった時点で捨てていくことが可能になった。そして Flyingware の典型的なアプリケーション例としてアンケートメールアプリケーションを作成し、拡張後の Flyingware が期待通りの動作を行うことを示す実験も行った。

謝辞

Flyingware の拡張を提案、実装するにあたり、多くの人から助言を頂いた。中でも、指導教官の千葉滋講師、筑波大学の立堀道昭氏、横田大輔氏、東京大学の光来健一氏には多大な助勢を頂いた。厚く感謝の気持ちを申し上げたい。また、千葉研究室の西澤無我氏、中川清志氏、柿原聡氏、栗田亮氏、宇崎央泰氏、お互いに助け合いながら卒業研究をして来たこれらの方々にも深く感謝の意を表したい。これから先も、これらの方々が、お互いに助け合いながら、自らの研究に力を注いでくれることを深く望む。

目 次

図 目 次

第1章　はじめに

インターネットの爆発的な普及により、ネットワークの通信量は増加の傾向にあり、大きな問題となっている。また、計算機間の通信遅延も大きな問題となっている。LAN 環境では計算機間の通信遅延は無視できるほど小さいが、インターネットなどのように遠くの計算機と通信する場合、通信遅延は無視できないほどの大きさになるからである。これらの問題を解決するシステムとして、モバイルエージェント・システムが注目されている。モバイルエージェントとは、計算機間を自律的に移動することのできるプログラムである。プログラムの状態を保持して移動して、移動先の計算機で移動元で行っていた作業の続きを再開する。相手計算機に移動して処理を行うことにより、通信遅延、通信回数の低減になる。特に通信回数に関して言えば、モバイルエージェントの移動後は移動元の計算機との通信の必要がなくなるので、ネットワークに接続するのはモバイルエージェントの移動時だけでよい。そのためモバイルエージェントは、ネットワークに非常時接続の環境でも非常に有用なシステムである。しかし、既存のモバイルエージェントの多くは、ネットワーク常時接続の環境を想定された設計になっていて、応用範囲も限定されてしまっている。我々の提案する Flyingware は、それらのモバイルエージェントと違い、ネットワークに非常時接続の環境でも使用可能なモバイルエージェントシステムである。Flyingware は、Java 言語で設計されており、自分自身を電子メールとして移動するため、実行環境を選ばない。さらに、通信コストを極力抑えるために、不要になったクラスを捨てながら移動できるように設計されている。今までても、ネットワークに非常時接続環境で使用可能なモバイルエージェントは存在したが、不要なデータをも移動してしまったり、あるいはそれを避けるためにユーザに負担のかかる設計になっていた。Flyingware では、移動時に自分自身の .class ファイルを自動的に解析することにより、移動先で必要なデータだけを自動的に収集することが出来る。毎回の移動時に自動的に必要なクラスだけを収集するということは、そのつど、不要なクラスを捨てて

いけるということである。しかし、従来の Flyingware では、自動的に移動できるのは .class ファイルとその時点で存在するオブジェクトだけである。もし画像ファイルや音声ファイルなどのデータファイルを移動しようとするならば、ユーザがプログラム中にそのことを明示的に記述しなければならない。また、それらのデータが不要になったら、そのこともユーザが明示的に記述しなければならない。これではユーザの負担が増えてしまう。

本稿では、Flyingware への機能拡張という形で、画像ファイルや音声ファイルなどのデータファイルを自動的に移動し、不要になった時点で捨てながら移動できるモバイルエージェントの提案を進めていく。Flyingware が既存のモバイルエージェントよりも優れている点は、その移動方式にある。不要になった .class ファイルを捨て、次の移動先で必要な .class ファイルだけを所持して移動するため、通信コストを最小限に抑えることができる。しかも、移動する際にこのことを自動的に行うため、ユーザの負担が低減される。このように、自動的に不要なデータを捨てていけるモバイルエージェントは、既存のものにはない。この利点を損なわないよう、Flyingware の機能を拡張しなければならない。まず、必要な画像ファイルなどのデータを、探し忘れのないように認識するために、自動的に自分自身の .class ファイルを解析し、必要なデータを探し出せる機能を追加した。このことにより、不要になった .class ファイルごと、不要な画像ファイルを捨てていくことができるようになった。そして、従来の Flyingware ではオブジェクトの解析を行っていなかったために、オブジェクトのクラスを必要だと認識していなかった。そのため、移動先で必要なファイルが見つからず、Flyingware が実行できないという状況が存在した。そこで我々は ObjectOutputStream クラスを拡張することにより、オブジェクトの解析を行い、オブジェクトのクラスも必要だと認識できるようにした。本稿では、２章でまず一般のモバイルエージェントについて述べる。既存のモバイルエージェントの例を挙げながら、理想的なモバイルエージェントの移動方法について述べていく。３章では、従来の Flyingware の機能、利点、欠点について述べる。Flyingware が理想の移動方式を取っていること、しかし、いくつかの欠点を持っていることを説明していく。そして４章では、それらの Flyingware の機能を拡張し、画像ファイルなどのデータを自動的に移動し、適時捨てながら移動していく方法について提案する。５章では、機能を拡張した Flyingware が正しく動作していることを示す実験と、Flyingware を用いた典型的な

アプリケーションについて述べる。最後に６章で本稿をまとめる。

第2章　モバイルエージェント

”モバイルエージェント”という言葉の定義は正確には定まってはいないが、一般的には、”計算機間を自律的に移動することのできるプログラム”のことを指す。本稿でも、そういうものとして”モバイルエージェント”という言葉を使う。

モバイルエージェントは、内部状態を保持したまま計算機間を自律的に移動し、そのつど処理を行うプログラムである。そのためモバイルエージェントにはいくつかの有用な利点が挙げられ、その応用範囲も広いと考えられている。しかし、様々な組織がモバイルエージェントの開発を手がけているが、理想的なモバイルエージェントを開発している組織は少ない。この章では、理想的なモバイルエージェントとはどのようなものかについて、モバイルエージェントの特徴、仕組み、応用例などを交えながら説明していく。

2.1　モバイルエージェントの特徴

インターネットの爆発的な普及により、ネットワークの通信量は増加の傾向にある。通信技術の進歩により通信は高速化しているとはいえ、クライアント・サーバシステムなどの従来の通信形態を用いている限り、通信大域不足の問題は解決しないものと思われる。また、LAN 環境では計算機間の通信遅延は無視できるほど小さいが、インターネットなどのように遠くの計算機と通信する場合、通信遅延は無視できないほどの大きさになるため、この計算機間の通信遅延も大きな問題となっている。これらの問題を解決する、新しい分散システムの実装方式として、モバイルエージェントは大きな注目を集めている。

モバイルエージェントは、移動の際にプログラムコード及びそのプログラムの内部状態や実行状態を保存して移動し、移動先の計算機で移動元で行っていた作業の続きを再開することができる。そのため、移動後は元の計算機との通信の必要がない。つまり、計算機間の通信はモバイル

エージェントの移動時だけでよい。このことが、モバイルエージェントの最大の利点である。モバイルエージェントを用いれば、計算機間通信の回数を低減させることができるし、通信を相手の計算機内で局所化することができるため、通信遅延は非常に小さいものとなるのである。この利点を生かした応用例として、モバイル www ロボットという検索ロボットが挙げられる。[12] 従来の検索ロボットは、1 ページごとに Web サーバーにリクエストを出し、そのレスポンスを受け取る。そのレスポンスの中には有用なデータだけでなくタグやヘッダおよび目的と異なるデータなどが混在している。これらはネットワークに対する負荷や、Web サーバーに対する負荷を無駄にあげることになる。また、毎回毎回リクエストを出すことの負荷も馬鹿にならない。これに対してモバイル www ロボットは、収集プログラムを対象となる Web サーバーに転送し、 Web サーバー内で全ての処理を行う。よって、毎回のリクエストに伴うネットワークに対する負荷が無い。サーバー内で不要なタグなどの除去、目的のデータの抽出、収集したデータの圧縮等を施した後に転送元の計算機に戻ることができるので、全体としての通信量を低減することができる。また、Web サーバーとの通信は最初と最後の 2 回のみになるので、通信遅延の影響を受けずスループットも向上する (図 2.1)。

その他のモバイルエージェントの応用例としては、負荷分散が上げられる。モバイルエージェントは、実行時の状態を保ったまま遠隔の計算機に移動し、処理を続行することができるため、負荷の高い計算機から負荷の低い計算機へタスクを移送し処理を続行することができるのである。また、複製を生成して複数の計算機上で並列処理させることもできるし、故障やシャットダウンの危険性のある計算機から安全な計算機に退避して処理を実行する、といった考え方も実現可能である。これら全ての応用例に共通するモバイルエージェントの利点は、実行状態を所持して移動するので、移動後は元の計算機との通信が必要ないということである。モバイル www ロボットの例のように、計算機がネットワークに接続される必要があるのはモバイルエージェントの移動時だけである。モバイルエージェントは、一時の断線などの障害にも強く、ネットワークに常時接続でない環境にとっても有用なシステムなのである。逆を言えば、モバイルエージェント・システムを設計する場合、移動後にはなるべく元の計算機との通信が起こらないようなシステムを設計する必要があると言える。

このように様々な利点が挙げられる反面、モバイルエージェントの実

図 2.1: モバイル www ロボットの例

現にはいくつかの制約がある。モバイルエージェントは不特定多数の多種多様な計算機上で実行されるため、以下の特徴を必要とするのである。

- ハードウェアや OS に依存しない
- 不正なモバイルエージェントが、移動先のホストを攻撃しない
- 不正なホストがモバイルエージェントを攻撃しない

モバイルエージェントは、不特定多数の多種多様な計算機間を移動し、そのつど実行されるので、どのような環境でも実行できなければならない。従って、モバイルエージェントの実行環境は、ハードウェアや OS に依存しないことが必要とされる。さらに、セキュリティにも十分注意しなければならない。モバイルエージェントが移動先のホストを故意ではないにしろ攻撃してしまうと、そのモバイルエージェントは、モバイルエージェントという名前のコンピューターウィルスということになってしまうし、不正なホストがモバイルエージェントのデータを盗み見たり改ざんしたりできてもいけない。モバイルエージェントはどんな環境でも実行できるが故に、安全面には十分注意しなければならないのである。

以上のように、モバイルエージェントの実現には様々な制約があるが、現在、モバイルエージェントの設計には Java 言語を用いるのが主流となっている。その理由として、Java 言語は、特定のプラットフォームに依存しないという利点がある他、以下のように豊富なセキュリティ機能が備えられていることが挙げられる。

- アドレスに対する演算の禁止
- 配列の添え字チェック
- 拡張可能な SecurityManager
- サンドボックス
- 電子著名

そのため、Java 言語とモバイルエージェントは非常に相性がよく、一般に知られているモバイルエージェントの多くは Java 言語を用いて設計されたものである。しかし、Java 言語を用いたとしても、セキュリティ上絶対に安全ということはなく、特にエージェントの秘密情報を不正ホストから守ることは非常に困難とされている。

2.2　モバイルエージェントの移動

モバイルエージェントの移動は以下の手順で行われる。

1. 現在いるホストでモバイルエージェントの実行を停止させる。
2. データを直列化する。
3. 相手先のホストに送信する。
4. 移動先でデータを複合化しモバイルエージェントに変換する。
5. モバイルエージェントの実行を再開する。

移動先で実行を再開するためには、プログラムコードと、プログラムの状態を表すデータを移動しなければならない。この移動するデータの内容によってモバイルエージェントを分類すると、大きく 3 つに分けることができる。

1. プログラムコードのみの移動
2. プログラムコードとデータ領域の移動
3. プログラムコードとデータ領域と実行状態領域の移動

データ領域とは、モバイルエージェントの持つデータを格納する領域のことで、インスタンス変数などが対象となる。実行状態領域は、局所変数やプログラムの呼び出し関係を保存するスタックと、プログラムの現在の実行箇所を示すプログラムカウンタを格納する領域である。Java 言語等におけるスレッドは、実行状態領域に格納される。1 のプログラムコードのみの移動では、プログラムの状態を表すデータは移動しないため、移動先では初期状態からプログラムを実行することになる。Java Applet や Servlet がこの手法を用いたものである。
2 と 3 ならば、データ領域や実行状態領域も移動するので、移動先の計算機で、移動元で行っていた作業の続きを行うことができる。2 では、プログラムの実行状態を移動しないため、移動直前の状態を再現するのが難しい。この場合、移動先ではシステムや利用者があらかじめ指定したメソッドが呼び出され、作業が再開される。そのため、利用者はローカルに動作するプログラムとは異なる作法で、移動することを意識してプログラムを書く必要がある。 その分プログラムの記述がやや複雑になる

が、この移動方法は機構が単純であるため既存のプログラミング言語処理系上に実現することが容易である。また、実行状態領域を移動しないため、全体としてのデータ量が少なくなるという利点がある。Java 言語の直列化機構はこの手法に基づいているので、Java 言語を用いたモバイルエージェントシステムにはこの方式を採用しているものが多い。具体的には、Aglets[1]、Voyager[2]、AgentSpace[4][5]、Flyingware などがある。

３は、ローカル変数やスタック領域、プログラムカウンタなども移動するので、移動後も移動直前の実行箇所から処理を再開することができる。そのため、プログラムを記述する際、移動性備えた特別な記述が不要で、エージェントの移動を含んだプログラムを自然にかつ簡潔に記述することが可能である。しかし、この分類に属するモバイルエージェントは、既存のプログラミング言語処理系に適用することが難しい。例えば、Java 言語を用いて設計された Moba[6] などは、独自に Java VM を拡張して、実行状態領域の転送を可能にしている。しかしそのために、オリジナルの Java VM との互換性が失われてしまい、このモバイルエージェントを実行するためには、その実行環境を計算機にインストールしなければならない。このように、このカテゴリに分類されるモバイルエージェントは、様々なプログラミング言語で独自の実行環境を実装しているため、その実行環境のインストールに手間がかかってしまう。また、実行状態領域を移動するということは、移動するエージェント全体のデータ量も増えてしまうため、実行速度、通信速度を向上させることも難しくなる。データ領域の移動だけでも、移動元ホストで行っていた作業を移動先ホストで再開することは可能であるし、移動するデータの内容に関しては、どちらの方が良いとは一概には言えない。

現在、様々な研究グループが、開発したモバイルエージェントを一般に公表しているが、それらのモバイルエージェントを前述の移動内容によって分けると、以下のようになる。

1. プログラムコードのみ
   例：Java Applet、Servlet
   移動先では初期状態から実行を再開する。

2. プログラムコード+データ領域
   例：Aglets、Voyager、AgentSpace、Flyingware
   与えられた行程表に基づきサーバーを移動して処理を行える

3. プログラムコード＋データ領域＋実行状態領域
   例：Telescript[3]、Moba
   実行状態も送るため移動前の実行状態を正確に再現できる

一般には 2 番以降のものをモバイルエージェントという。この他にも、プランニング情報を一緒に移動することで、状況に応じた行動が取れる Plangent エージェント [7] など、様々なアイディアのものが知られている。上に挙げたモバイルエージェントの特徴などについては次節で述べる。

モバイルエージェントの移動で最も重要になるのが、データの移動方式である。モバイルエージェントは、移動するデータの内容によって大きく 3 つに分けれることを説明したが、どの場合も、それらのデータを相手計算機に移動する必要がある。最低でも、モバイルエージェントの実行に必要なプログラムコードが相手計算機に存在しなければ、実行を再開することができない。データ領域、実行状態領域については、現在いる計算機での実行が終了した直後に相手計算機に転送すれば良いが、プログラムコードの移動に関しては、必ずしも直後に移動する必要はない。この、移動先で必要なプログラムコードの転送には、3 通りの方式がある。[5]

1. 移動先の計算機にも、必要となるプログラムコードをインストールしておく方法

2. 移動の際、最初に必要なプログラムコードを全て一括して送る、静的ダウンロード方式

3. 移動先で実行中に、必要に応じてプログラムコードをダウンロードする、動的ダウンロード方式

あらかじめプログラムコードをインストールしておく方法は、プログラムコードを転送しなくて良い分効率が良いが、莫大な手間がかかる。そもそもモバイルエージェントは、不特定多数の多種多様な計算機上で実行されることを想定しているため、この方式は現実的ではない。最初に必要なプログラムコードを一括して転送する静的ダウンロード方法では、プログラムコード移動後には移動元と移動先の計算機で通信の必要がなくなるので、モバイルエージェントの利点を備えることが出来る。しかしこの方法では、必要なプログラムコードだけを選んで移動できるかど

うかが問題となる。特に、Java 言語では、実行に必要なクラスが明らかであるとは限らないため、この方法では、実行に必要なクラスを移動できないことや、必要のないクラスまで移動してしまうことが考えられる。そのため、Java 言語を用いたモバイルエージェントは、３番目の動的ダウンロード方式をとっているものが多い。この動的ダウンロード方式では、最初に転送するのは必要最低限のクラスだけにとどめておき、後から必要に応じてクラスを転送するのである。この方法ならば、必要なクラスを必ず転送することができるし、必要のないクラスを転送してしまう恐れもない。しかし、この方法には以下のような欠点がある。

- 必要になった時点で動的にクラスをロードするので、移動後も必ずネットワークに接続されている必要がある。もしも通信元の計算機との回線が切断されたりしたら、クラスの転送が不可能となり、処理が継続できなくなる。

- 実行時に必要となるクラスを転送するためには、最低 1 往復分の通信遅延がかかり、必要なクラスが多いとその転送コストは無視できなくなる。

- ローディングされるクラスは必要性が判明してから転送要求を出すため、その間処理が停止し、システムのレスポンスが悪くなる。

後から必要に応じてクラスを転送するということは、ネットワークに常時接続でなければ使用が出来ないと言うことである。これではモバイルエージェントの最大の利点である、非常時接続環境での有効性が失われてしまう。また、使用する環境がネットワークに常時接続の環境であったとしても、２や３のように、通信遅延の増大という問題が出てくる。これでは、前節で述べたモバイルエージェントの利点がほとんど実現できていないことになる。

このように、プログラムコードの移動方式は、モバイルエージェントの実現にとって大きな問題となる。プログラムコードをあらかじめインストールしておく方式は先ほども述べたように現実的ではないので問題外とするが、静的ダウンロード方式も動的ダウンロード方式も、それぞれ長所、短所を持っている。動的ダウンロード方式では、不要なクラスを転送しないメリットは大きいが、ネットワークに常時接続の環境を必要とするデメリットの方がはるかに大きい。なぜなら、モバイルエージェントの最大の利点は、ネットワークに非常時接続の環境でも有用だという

ことである。そのため、モバイルエージェントは静的ダウンロード方式を採用する必要がある。をユーザ自らが指定して移動させなければならないなど、ユーザに負担をかける設計となっている。また、移動中に不要になったクラスを自動的に捨てることができず、不要なデータを所持したまま移動してしまうといった欠点も持っている。理想のモバイルエージェントの移動方式とは、これらの欠点を解決した、静的ダウンロード方式のモバイルエージェントである。つまり、モバイルエージェントの移動時に、移動先で必要なクラスが全てわかり、その必要なクラスを一括して自動的に転送することができれば、それが最も理想的なプログラムコード移動方式である。そして、この理想的な移動方式を実現しているのが、我々の研究室で開発された `Flyingware` である。次節では既存の様々なモバイルエージェント・システムについて述べ、次章で `Flyingware` について述べていく。

2.3　様々なモバイルエージェント

この節では、さまざまなグループが開発している既存のモバイルエージェントが、どのような転送方式をとっているのか、理想的な移動方式を実現しているモバイルエージェントがあるのかを交えながら、それらの利点、欠点などについて述べていく。

Telescript　`Telescript` は 1990 年にジェネラル・マジック社が提唱したプログラミング言語で、商用言語として初めてモバイルエージェント技術を実装したものである。独自のエージェント実行環境を実装しているために実行状態領域の移動が可能であるが、実行環境のインストールに手間がかかる。また、動的ダウンロード方式を取っている。

Odyssey　Odyssey は Telescript を開発した General Magic が提供する Java モバイルエージェントである [3]。Telescript のアーキテクチャをほぼ継承しているが、エージェントが Java のスレッドであり、移動には RMI が用いられるなど、Java の機能を活用して実装が行われている。Aglets と同様に移動先での実行継続はできないが、移動先でどのメソッドを起動するかを開発者がリスト形式で自由に指定できる。

Aglets　Aglets は IBM 社によって開発されたモバイルエージェント・システムで、Java 言語によって実装されている。Java のクラスライブラリ上で実装されたものとしては先駆的なものの一つである。Aglets は Java の直列化機構を利用しているためローカル変数などは移動の対象にならず、インスタンス変数のみを移動することができる。Agletes のプログラムコード転送方式は、プログラム実行中に必要に応じてダウンロードしてくる、動的ダウンロード方式である。そのため、前節で述べたように、プログラムコードの転送要求が頻繁に起こると、通信遅延が増大してしまう恐れがある。また、Aglets を用いたモバイルエージェントの実行には、指定されたエージェント・サーバが必要になる。

Voyager　Voyager は ObjectSpace 社によって開発された、Java 上でモバイルエージェント機能を実現する ORB(Object Request Broker) パッケージである。VoyagerORB を使うと、Java のクラスは実行時にリモートからアクセス可能にでき、リモートにインスタンスを生成したりできる。また、セキュリティ機能や HTTP プロトコルへの対応など、インターネットを非常に意識した作りになっている。Voyager では独自の ORB を利用しているが、移動の対象となるのは Java 直列化機構と同じようにインスタンス変数である。Agetls と同じく、動的ローディング方式を採用しており、実行には指定されたエージェント・サーバをインストールしなければならない。

MOBA　Moba (Mobile Agent Facilities on Java Language Environment) は、早稲田大学の首藤一幸氏が開発しているモバイルエージェント・システムである。Java で実装されているが、Java VM を拡張することにより実行状態領域も移動することを可能にしている。しかし、オリジナルの Java VM との互換性が失われるため、実行環境をインストールしなければならず、その対応プラットフォームも狭い。移動の際には特別な記述は必要なく、利用者はローカルなプログラムとの差をほとんど意識する必要がない。さらに耐故障性向上のため、計算途中の状態を保存する仕掛けなども備わっている。しかし、クラスの移動方式は動的ローディング方式のため、ネットワークに上接続の環境でなければ仕様が難しい。

AgentSpace　AgentSpace システムは、Java 言語により実装されたモバイルエージェント・システムで、移動する内容はプログラムコードとデータ領域である。このシステムでは、エージェントの転送中は、ZIP 形式の

データ圧縮により通信の高速化をはかっている。また、セキュリティ確保のために転送中データの暗号化も可能になっている。さらに AgentSpace システムは、他の Java ベースのモバイルエージェントシステムと違い、静的ダウンロード方式を採用している。そのため、通信コストを抑えることが可能であるが、移動先で必要となるクラスはユーザ自らが .jar コマンドを用いて統合する必要がある。また、エージェントランタイムシステムと呼ばれる実行環境上で動作するため、この実行環境をインストールする必要がある。

Plangent　Plangent エージェント・システムは，Java 言語で実装されているが、他の Java ベースのモバイルエージェントとは異なる点がある。それは、このシステムでは、エージェントに”ゴール”を与えることにより、そのゴールへ向かうための適切なプランニングを行うことができるということである。それぞれの計算機やエージェントは固有の知識ベースを持ち、それにより、”ゴール”を達成するための適切な行動プランを生成するのである。さらに、エージェントの実行環境として論理型言語を Java 上に実装し、その実行環境に基づくエージェントを移動させる点である。そのため、他のモバイルエージェントと違い、実行状態領域を移動することが可能となっている。しかし、実行には特別な環境が必要となり、複数のホストがその環境をインストールし、立ち上げた状態でなければなければ実行できない。また、静的ダウンロード方式を採用しているが、不要になったクラスを自動的に捨てていくような機能はない [13]。

以上のように、既存のモバイルエージェントの多くは、プログラムコード転送に動的ダウンロード方式を採用している。動的ダウンロード方式では、不要なクラスを転送しないメリットは大きいが、ネットワークに常時接続の環境を必要とするデメリットの方がはるかに大きい。なぜなら、モバイルエージェントの最大の利点は、ネットワークに非常時接続の環境でも有用だということだからである。さらにこの方式は、常時接続の環境であったとしても、通信遅延が増大してしまう可能性を持っている。そのため、モバイルエージェントは静的ダウンロード方式を採用する必要がある。しかし、既存の静的ダウンロード方式を採用しているモバイルエージェント、AgentSpace や Plangent では、必要なクラスの転送をユーザが手作業で行わなくてはならないなど、ユーザに負担のか

かる設計となっている。また、移動中に不要になったクラスを自動的に捨てることが出来ず、不要なクラスを所持したまま移動してしまうなど、理想的な転送方式とは言えない。理想のモバイルエージェントの移動方式とは、これらの欠点を解決した、静的ダウンロード方式である。つまり、モバイルエージェントの移動時に、移動先で必要なクラスが全てわかり、その必要なクラスを一括して自動的に転送することができれば、それが最も理想的なプログラムコード移動方式である。そして、この理想的な移動方式を実現しているのが、我々の研究室で開発された Flyingware である。Flyingware は、移動時に、移動先で必要なクラスを自動的に識別し、一括して移動するという方式をとっている。ネットワークに常時接続の環境でなくても使用可能で、ユーザが必要なクラスを選択する負担もない、理想的な移動方式であると言える。また、Flyingware は、実行時に特別な実行環境を必要としない。既存のモバイルエージェント・システムでは、実行するために、エージェント・サーバと呼ばれるような実行環境のインストールが必要となる。モバイルエージェントは、多種多様の計算機上で実行されるため、実行時に特別な環境が必要だったり、ネットワークに常時接続の環境でなければ使用できないのでは、その応用範囲も限定されたものになってしまう。Flyingware は、ネットワークに非常時接続の環境でも使用でき、実行時の特別な環境を必要としないため、様々な環境で使用が可能なモバイルエージェント・システムである。次章では、Flyingware の利点や欠点ついて述べながら、さらに Flyingware を改良していく方法について述べる。

第3章　従来のFlyingware

Flyingwareとは、電子メールの交換プロトコルであるSMTPを用いて、自分自身を転送することができるモバイルエージェントである。転送のための基本機能はFlyingwareAPIと呼ぶJavaのクラスライブラリによって提供される。Flyingwareはこのライブラリを用いて書かれた通常のJavaアプリケーションである。
Flyingwareはモバイルエージェントであるので、当然モバイルエージェントが必要とする要素を備えている。まず、電子メールを用いてJavaアプリケーションを送るので、Java言語の処理系が適切にインストールされてさえいれば、プラットフォームに依存せずに実行することができる。また、Javaのセキュリティ・マネージャを利用して、セキュリティの問題も解決している。そして、移動時点で存在しているオブジェクトも移動することができる。ただし、FlyingwareはJavaの直列化機構を利用しているので、移動の対象となるのはインスタンス変数であり、ローカル変数などは移動の対象とはならない。ここまででは、一般に公開されている他のJavaベースのモバイルエージェントとほとんど変わらないが、Flyingwareには、他のモバイルエージェントにはない大きな特徴を持っている。それは.classファイルの自動解析機能である。
Flyingwareを用いてアプリケーションを書き実行すると、Flyingwareは自分自身の.classファイルを自動的に解析して、移動先で必要になるであろうクラスを探し出す。移動先で必要になるクラスとは、Flyingwareアプリケーションのクラスと、移動先でそのアプリケーションクラスから参照される可能性のあるクラスのことである。Flyingwareは、それらの.classファイルと、その時点で存在しているオブジェクトを１つの.jarファイルにまとめ、それを電子メールの添付ファイルとして相手ホストに送る。プログラマが、転送先で必要となるクラスをいちいち指定する必要がない。Flyingwareは、最初の移動時に必要なクラスだけを一括して転送してくれるモバイルエージェントなのである。プログラムコードの移動方式は、他のモバイルエージェントでは大きな問題となっていた。

一括方式では、不要なプログラムコードまで送ってしまう恐れがあるし、遅延ローディング方式では、通信遅延の増加などの問題がある。しかし Flyingware では、最初に必要なクラスだけを一括して転送するという、理想的な移動方式を実現している。Flyingware には、このこと以外にも、他のモバイルエージェントよりも優れている点がいくつかある。この章では、Flyingware の利点、欠点、応用例などを挙げながら、Flyingware の応用範囲をさらに広げる方法について述べる。

3.1　Flyingware の利点と応用例

Flyingware には大きな特徴が 2 つある。一つは、自らを電子メールとして移動する、ということである。この面から見ると、Flyingware は、自律的に移動することのできる電子メールである、という言い方ができる。もう 1 つの大きな特徴は、移動する際に自分自身を解析して、移動先で必要になるであろう .class ファイルを選び出し、転送するということである。この面から見ると、Flyingware は、不要な .class ファイルを捨てながら移動するモバイルエージェントである、という言い方ができる。どちらの特徴も、他のモバイルエージェントにはない利点をもたらしてくれる。

まず、自らを電子メールとして移動することの利点を述べる。電子メールにはいくつかの利点があるが、その 1 つとして、特定のプラットフォームに依存しないことが挙げられる。したがって Flyingware は、Java 言語の処理系が適切にインストールされていれば、ハードウェアや OS に依存せずに実行できるばかりか、他の多くのモバイルエージェントが必要とする実行時の特別な環境を必要としない。一般に公開されているモバイルエージェントは、実行するにあたって、エージェント・サーバーあるいはプレースと呼ばれる、特別な実行環境を必要とする。これらは主に Web 上からダウンロードしてくることになり、この実行環境を持たない計算機上では、モバイルエージェントは動作できない。そもそもモバイルエージェントは、不特定多数の計算機上で実行されるものとされている。実行時に特別な環境が必要なのではこの構想から外れてしまい、モバイルエージェントの応用例も限られてしまう。現在、電子メールは非常に広く普及しているため、Flyingware のユーザーは、Flyingware の転送先の環境を気にすることなくこれを使用することができる。

電子メールの利点はまだ他にもある。最近ではブロードバンドインターネットという言葉が有名になってきて、インターネットを常時接続の環境で使うユーザーも増えてきた。しかし常時接続環境はまだまだ一般的なインターネット環境とは言えない。常時接続環境への手続きには手間がかかるし、コストもかかる。インターネットをあまり使わないユーザーにとっては常時接続の必要はないし、安全性の面からも常時接続環境に抵抗のあるユーザーもいるだろう。地域によっては常時接続が可能でない地域もたくさんある。これらの理由から、インターネットユーザーの多くは未だ非常時接続の環境にあると思われる。これらインターネット非常時接続のユーザーにとって、電子メールは有用な情報伝達手段である。電子メールであれば、一度手元のマシンに取り込んでしまえばいつでも閲覧可能であるし、相手にメールを送りたいときにだけインターネットに接続すればよい。また、電子メールは Web ページと違い、特定の人に閲覧してもらうことが可能である。Web ページは、不特定の大多数の人々に見てもらえる利点はあるが、特定の人に閲覧してもらうためには、その人に何らかの方法で Web ページのアドレスを伝えなければならない。直接その人に会って伝えるか、電話で伝えるか、メールで伝えるなどの方法があるが、どちらにせよ二度手間である。電子メールならば特定の相手を指定して送信すればその人に閲覧してもらうことが可能であるし、相手によってメールの内容をカスタマイズすることが可能である。従来の電子メールでも、メッセージを HTML 形式で記述すれば、書体や字の色や大きさを変えたり、背景色を変えたりと、Web ページと同等の表現力を電子メールに持たせることができたが、Flyingware を電子メールとして使うと、従来のテキストまたは HTML を用いた電子メールよりも、より自律的、対話的な表現が可能になる。例えば、ユーザーと対話して、その結果に応じて自動的に返信したり、別のユーザーに転送する、といった自律的なコンテンツを作成することができるようになる。

以上の電子メールの利点を考慮すると、この Flyingware は、企業の広報活動などで有効利用できると思われる。企業が顧客に対して広告宣伝を行う場合などには、電子メールを用いて商品を紹介したり、イベントの通知を行ったりと、電子メールを用いる場合が非常に多い。また、企業側から顧客側への一方通行の広報活動だけではなく、商品の申し込みや、イベントの参加通知、市場調査に対する回答など、顧客側から企業側へ電子メールで返信されることも多いものと思われる。例えば、企業が顧

客に対してアンケートのメールを送る場合を考える。企業がアンケートの結果をデータベース化する時、もし従来の電子メールを用いていたら、この作業は大変面倒なものになる。なぜなら、顧客が自由な形式で書いた返答メールを、企業側が手作業でデータベース化しなければならないからである。もし手作業でのデータベース化が面倒であれば別の方法もある。企業が、アンケートを行う Web サイトを作成しておき、その Web サイトのアドレスを書いた電子メールを顧客に転送するのである。顧客は、企業側から送られてきた電子メール中に記されている Web サイトを訪れ、その Web サイトのページを使って企業への返答をする。Web サイトを用いた方法ならば定型的なフォームで返答が送信されるので、手作業でなく、計算機で直接返答を処理できる利点はある。しかし、顧客はその Web サイトの閲覧中はインターネットに接続された環境を必要とする。

Flyingware を用いれば上記の問題が解決するほか、企業側から顧客側にメールを送信するときには大量のデータを積んでいたが、返答時には必要のないデータを切り捨てて必要な少量のデータだけを持ち帰るといったことも可能になる。例えば、旅行会社が大量の画像データを積んだパンフレットメールを顧客に送り、その返答を求める場合などには、最初は大量の画像データを積んでいるが、返ってくるときには画像データが必要なくなるので、その画像データを切り捨てれば、大幅な通信量の削減になる。

Flyingware のもう 1 つの大きな特徴が、移動時に自分自身の .class ファイルを自動的に解析して、必要なクラスだけを探し出す、ということである。このことにより Flyingware は、移動時に、必要なクラスだけを、一括して転送することができるのである。これは、モバイルエージェントのプログラムコード移動方式としては理想的な方式で、この移動方式を実現している点が、Flyingware が他のモバイルエージェントに比べて大きく秀でている点である。
Flyingware が移動可能なデータは、自分自身のアプリケーションクラス、移動先でそのアプリケーションクラスから参照される可能性のあるクラス、そしてその時点で存在しているオブジェクト、である。その時点で存在するオブジェクトを認識する作業は、ObjectOutputStream クラスで行われる。このクラスの writeObject メソッドは、引数に取るオブジェクトから参照される可能性のある全てのオブジェクトをストリームに書き

出す作業をする。この働きにより、その時点で存在しているオブジェクトを的確に認識し、移動させることができる。重要なのは、Flyingware が必要なクラスをどのように認識するか、である。Flyingware アプリケーションを他の計算機に移動させるには、fly メソッドを呼び出す。この fly メソッドは、移動させる Flyingware のアプリケーションのクラスオブジェクトと、そのクラスの中の、移動先で実行を再開するメソッド名を引数に取る。このメソッド名はアプリケーションプログラマが任意に設定することができ、本論分では今後、このメソッドのことを resume メソッドと呼ぶことにする。fly メソッドを呼び出すと、Flyingware アプリケーションはその計算機での処理を中止し、移動先で必要なクラスを探す作業を始める。この、必要なクラスを探し、１つの .jar ファイルにまとめる作業は、Flyingware.AgentOutputStream クラスで行われる。このクラスで行われる作業は、以下のとおりである。

1. Agent クラス（移動先で Flyingware アプリケーションを起動するためのクラス）を探し出し、.jar ファイルにまとめる。

2. fly メソッドを呼び出す際に引数で指定した、移動先で実行を再開する Flyingware アプリケーションのクラスオブジェクトを解析して、その時点で存在しているオブジェクトを調べ、.jar ファイルにまとめる。

3. そのアプリケーションクラスの .class ファイルを解析し、新たなクラスを参照するかどうかを調べる。

4. 参照するならば、その参照されるクラスの .class ファイルを調べ、また、新たなクラスを参照するかどうかを調べる。

5. ４を繰り返す。

このように Flyingware は、移動先で実行されるアプリケーションクラスの .class ファイルから、参照するクラスを芋づる式に見つけていくのである。アプリケーションがあるクラスを参照するということは、そのクラスはアプリケーションの実行に必要なクラスということで、逆に参照しないクラスは、アプリケーションの実行に不要なクラスということになる。そのため Flyingware は、移動先での実行に必要のなくなったクラスは、探し出さずにその場に捨てていくことができる。この働きを説明するために、以下のようなアプリケーションの例を用意した。このアプ

リケーションは、MorningTest クラス、Morning クラス、Afternoon クラス、Evening クラス、Lunch クラス、Bread クラスの 6 つのクラスからなる。（図 3.1）このアプリケーションはまず、MorningTest クラスで Morning クラスのオブジェクトを生成し、移動させる。移動先で Morning クラスは Afternoon クラスのオブジェクトを生成し、それを移動させる。Afternoon クラスは移動後に Evening クラスのオブジェクトを生成し移動させる。このようにこの例は、順次新しいクラスオブジェクトを生成しながら、それを移動させていく、というものである。

図 3.1: .class ファイルの解析の例

このアプリケーションの例で、Flyingware の働きを簡単に説明する。

最初の移動時　この Flyingware アプリケーションは、MorningTest クラスから実行される。このクラスは Morning クラスのオブジェクトを作成して移動させようとする。この時 Flyingware は、まず Agent クラスを .jar ファイルにまとめ、次にその時点で存在するオブジェクトを .jar

ファイルにまとめ、そして Morning クラスの .class ファイルを .jar ファイルにまとめる。その後、Flyingware は Morning クラスの .class ファイルの解析を行い、Morning クラスでは Afternoon クラスが参照されているのを発見する。そこで Afternoon クラスの .class ファイルを .jar ファイルにまとめ、今度は Afternoon クラスの .class ファイルを解析する。Afternoon クラスでは Lunch クラスと Evening クラスが参照されているが、Flyingware はまず Lunch クラスが参照されているのを発見するので、Lunch クラスの .class ファイルを .jar ファイルにまとめる。そして Lunch クラスの .class ファイルを解析し、そこで Bread クラスが参照されているのを発見するので、その .class ファイルを .jar ファイルにまとめる。Bread クラスでは新たなクラスは参照されていないことがわかり、探索途中だった Afternoon クラスの解析に戻る。そこで今度は Evening クラスが参照されているのを発見し、Evening クラスの .class ファイルを .jar ファイルにまとめる。さらに Evening クラスの .class ファイルを解析するが、新たなクラスは参照されていないので、必要なファイルを一つの .jar ファイルにまとめる作業はそこで終了し、.jar ファイルを移動する。（当然 FlyingwareAPI のクラスファイルも必要なものだけ移動されるが、そのことの記述は今後も省略する。）

２回目の移動時　Morning クラスは、移動先で Afternoon クラスのオブジェクトを作成し、それをまた移動させようとする。Agent クラスとその時点で存在するオブジェクト、そして Afternoon クラスの .class ファイルを一つの .jar ファイルにまとめた後、Flyingware は Afternoon クラスの .class ファイルを解析する。最初の移動時と同様、Afternoon クラスからは Lunch クラス、Evening クラス、Bread クラスが参照されるので、それらの .class ファイルを .jar ファイルにまとめ、出来上がった .jar ファイルを移動する。この時、Morning クラスは .jar ファイルにまとめられていない。つまりこの時点で、Morning クラスは必要がなくなったので、捨てられたことになる。

３回目の移動時　Afternoon クラスは移動先では Evening クラスのオブジェクトを作成し、再度移動しようとする。Agent クラスとその時点で存在するオブジェクト、そして Evening クラスの .class ファイルを一つの .jar ファイルにまとめた後、Flyingware は Evening クラスの .class ファイルを解析する。Evening クラスでは新しいクラスは参照されないので、必要なファイルの探索は終了し、.jar ファイルを再度移動する。この時点で、Afternoon クラスと、そこから参照されていた Lunch クラス、Bread

クラスは必要がなくなったので、捨てられたことになる。Evening クラスの移動後、このアプリケーションは終了する。最初の移動時には、Morning クラス、Afternoon クラス、Evening クラス、Lunch クラス、Bread クラスを所持して移動した Flyingware が、３回目の移動時には、Evening クラスの移動だけですんだことになる。
このように、Flyingware は、不要になったクラスを捨てながら、必要なクラスだけを所持して移動できるモバイルエージェントなのである。

3.2　Flyingware の弱点

Flyingware には、まだまだ不自由な点がある。その一つが、自動的に転送できるデータの種類である。Flyingware が自分自身を自動的に解析する際、移動先で必要だと認識し転送できるデータは、Java の .class ファイルと、その時点で存在するオブジェクトだけである。それ以外のファイル、例えば.gif ファイルや .jpg ファイルなどの画像ファイルや、.wav ファイルなどの音声ファイルは自動的に転送できない。もっと詳しく言えば、それらのファイルを探すことをしない。それでは、前述の旅行会社のパンフレットメールの様に画像データを送りたい場合はどうすればよいかというと、従来の Flyingware で画像データを転送したければ、ユーザが最初に、画像データをオブジェクトとして指定しなければならない。そしてその画像データを転送する必要がなくなったら、そのことをユーザーが明示的に記さなければならない。このことは、プログラマの負担を増大させることになる。また、.gif ファイルや .wav ファイルなど、画像ファイルや音声ファイルなどをそのまま移動できたほうが良い場合がある。例えば、移動先の相手に渡したい画像データを積んで移動し、移動先で全て相手に渡す場合などである。Flyingware は自分自身を .jar ファイルに圧縮して移動するので、移動先で自分自身を解凍すれば、画像データをそのまま相手に渡すことができる。この場合、画像ファイルや音声ファイルをオブジェクトにせず、元のままのファイルを所持して移動する必要がある。このように、.class ファイル以外の、画像ファイルなどに関しても、自動的に移動できるようにしたほうが、ユーザの負担が減り、Flyingware の応用範囲も広がる。特に画像ファイルは、様々なアプリケーションで頻繁に利用されるものと思われる。画像ファイルの使用でいちいちユーザに負担をかけていたのでは、アプリケーションを作ることも相当な負担になってしまう。画像ファイルなどは、ユー

ザに負担のかからないよう、Flyingware が自動的に移動してくれた方が良い。従来の、ユーザが明示的に指示する方法では、Flyingware の利点を生かせているとはいえない。Flyingware の利点は、自分自身を自動的に解析して、必要な.class ファイルだけを選び出せることである。ならば、.class ファイル以外の画像ファイルや音声ファイルに関しても同じ事ができたほうが良い。つまり、Flyingware は自分自身を自動的に解析して、どのような種類のファイルに関しても、必要なファイルだけを選び出し、移動できるようにする必要があるのである。

さらに研究の途中で、Flyingware は、オブジェクトを解析をしないことが発見された。このことはつまり、Flyingware にはクラスの探し忘れがあることを意味している。例えば、次のような場合に、必要なはずのクラスを探し忘れてしまう。Human クラスというスーパークラスと、そのサブクラスの Man クラスがあり、以下のような Flyingware アプリケーションを作成したとする。

```
public class Sample {
    public Human human;
            ⋮
    pubilc void resume(String[] args) {
                    ⋮
        flight.fly(mo, "resume");
                    ⋮
    }
}
```

簡単のため、コードの記述は省略してあるが、このアプリケーションは、Human クラスのインスタンス変数を所持して、ただ移動を繰り返すだけのものである。このクラスには main メソッドがなく、最初にこのアプリケーションを移動させるときには、そのためのクラスを作成する必要がある。ここまでは、特に問題のない、普通の Flyingware アプリケーションである。ただし、このアプリケーションを最初に移動させるクラスで、次のようなことをした場合に、問題が起こる。

```
public class SampleTest {
    pubilc static void main(String[] args) {
        Sample sample = new Sample();
        sample.human=new Man();
                ⋮
        sample.flight.fly(sample, "resume");
                ⋮
    }
}
```

このように Java 言語では、スーパークラスのオブジェクトの型の中に、サブクラスのオブジェクトを入れることが可能である。この例では、Sample クラスの Human オブジェクトの型の中に入っているのは、実際には Man オブジェクトである。SampleTest クラスを実行すると、このクラスは Sample クラスのオブジェクトを生成し、それを他の計算機に移動させる。この時、移動させるクラスは、Flyingware のクラスの他に、Sample クラス、Human クラスである。Man クラスは移動しない。なぜなら、移動の命令を受けた Flyingware は、以下のように振舞うからである。

1. 他の計算機への移動命令を受ける。
2. Agent クラスとこの時点で存在するオブジェクトを .jar ファイルに追加する。
3. 移動させるクラスオブジェクトは Sample クラスのものなので、Sample クラスの .class ファイルを .jar ファイルに追加する。
4. Sample クラスの .class ファイルを解析する。
5. Sample クラスは Human 型のインスタンス変数を所持していることを発見し、Human クラスの .class ファイルを .jar ファイルに追加する。
6. ファイルの探索を終了し、移動する。

SampleTest クラスでは、Flyingware は、Sample クラスのオブジェクトを移動させるように命令を受けている。そのため、Sample クラスの .class

ファイルから調べ始めるのだが、Sample クラスには Man クラスの記述がないため、Sample クラスの .class ファイルにも Man クラスに関する記述は出てこない。従って、Flyingware は Man クラスを探し出せないことになる。もしオブジェクトの解析を行うことができれば、Man クラスを認識することができるようになる。２番目の、オブジェクトを .jar ファイルに追加する段階で、それらのオブジェクトを解析し、元のクラスを割り出すようにすれば良い。しかし、従来の Flyingware ではオブジェクトの解析を行わないため、このような場合に、クラスの探し忘れが生じてしまうのである。

第4章　Flyingware の拡張

前述のように、Flyingwareには、二つの欠点があった。一つは、Flyingware はオブジェクトの解析を行わないため、必要なはずのクラスを探し忘れる場合がある、ということである。もう一つは、自動的に移動するデータは、.class ファイルと、その時点で存在するオブジェクトだけということである。Flyingware を利用するに当たって、典型的なアプリケーションは何かということを考えると、やはり企業の宣伝メールのようなものがあげられる。このアプリケーションを実現するためには、.class ファイル以外にも、画像ファイルや音声ファイルなどのデータファイルも転送できるようにしなければならない。なぜなら、前章の最後に述べたように、従来の Flyingware で画像データなどを転送するためには、ユーザーの負担が増大してしまうし、画像データなどの元のファイルも移動できたほうが、Flyingware の応用範囲が広がるからである。クラスの探し忘れをなくすのはもちろん、移動の際には自動的に自分自身を調べ上げ、.class ファイル以外の必要なデータファイルを相手ホストに送り、必要のないデータファイルは適時捨てていけるように Flyingware を改良する必要がある。そこで我々は、以上の Flyingware の欠点を補い、改良する方法を提案する。本論分での Flyingware の最終的な目標は、クラスの探し忘れをなくすのはもちろんだが、移動の際に自分自身を自動的に解析し、必要な画像ファイルと必要のない画像ファイルを識別することである。

4.1　ObjectOutputStream クラスの拡張

従来の Flyingware はオブジェクトの解析を行わないことは前章の最後で述べた。このため、従来の Flyingware には.class ファイルの探し忘れがある。この探し忘れをなくすためには、オブジェクトの解析を行い、そのオブジェクトが参照するクラスを全て認識する必要がある。そこで用いるのが ObjectOutputStream クラスである。Flyingware では、

このクラスの writeObject メソッドを用いて、あるオブジェクトから参照される全てのオブジェクトを認識していることは前章で述べた。さらに ObjectOutputStream クラスには、それらオブジェクトの解析を簡単に実現できる機能が最初から備わっている。それは、annotateClass メソッドである。

W riteObject メソッドは、呼び出されると、引数のオブジェクトから参照される全てのオブジェクトを、出力ストリームに書き出す作業をする。前述のクラスの探し忘れをなくすためには、それら参照されるオブジェクトの元のクラスを認識し、一緒に転送できるようにすればよい。そこで annotateClass メソッドの機能を用いる。このメソッドは、writeObject メソッドを呼び出した際に、その中から呼び出される抽象メソッドである。

writeObject メソッドは、引数のオブジェクトを調べていき、そのオブジェクトから参照される別のオブジェクトを探し出す作業をする。そしてもし別のオブジェクトを参照していれば、そのオブジェクトの元のクラスを調べる。この元のクラスが、もし初めて登場したものであれば、writeObject メソッドは annotateClass メソッドを呼び出す。したがって、annotateClass メソッドは、writeObject メソッドの引数のオブジェクトが参照する全てのオブジェクトの元のクラスを引数にとることになる。ただし、このメソッドはデフォルトでは何も記述されていないので、これらのオブジェクトの元のクラスは無視される。従来の Flyingware はオブジェクトの解析を行わないためにクラスの探し忘れがあると以前述べたが、これは正確な言い方ではない。Flyingware は ObjectOutputStream クラスを用いてオブジェクトの認識を行っていて、この際、それらのオブジェクトの元のクラスも認識しているのである。しかし、認識するだけで、そのクラスを移動する作業をしないため、クラスの移動し忘れがある、という言い方が正確な言い方である。このように、annotateClass メソッドは、オブジェクトの元のクラスを認識できるので、それらのクラスの .class ファイルを .jar ファイルにまとめるようにできれば、.class ファイルの探し忘れはなくなる。

annotateClass メソッドを実装するために、ObjectOutputStream クラスのサブクラス、MyObjectOutputStream クラスを作成し、その中で annotateClass メソッドを実装した。前述のように、writeObject メソッドは、引数のオブジェクトを調べていき、そこから参照されるオブジェクトの元のクラスで初めて登場するものがあると、そのクラスを引数にとっ

て annotateClass メソッドを呼び出す。したがって、annotateClass メソッドでは、引数にとるクラスを全て覚えけばよいということになる。具体的には、Vector クラスでヴェクトルを作っておき、それに、引数のクラスを追加する、という方法をとった。あとで必要なファイルを .jar ファイルにまとめる際に、このクラスのヴェクトルを参照すれば、どのクラスが必要なのかがわかる。これで、従来の Flyingware では存在したクラスの探し忘れが、なくなったことになる。

4.2　必要なデータファイルの認識

Flyingware の機能拡張の目標は、画像ファイルなどのデータファイルを、自動的に、必要なものと必要でないものを識別することにある。この節では、必要なデータファイルを見落としのないように探し出す方法を提案する。
話をわかりやすくするために、まず、画像ファイルの識別に焦点を絞って話を進めていく。画像ファイルを Java プログラム中で使うには、通常はそのソースファイル中に、画像ファイル名を記述する。例えば以下のような文をプログラム中に記述することになる。

例 1:

```
ImageIcon icon = new ImageIcon("sample.gif");
JOptionPane.showMessageDialog( ··· , icon);
```

例 2 :

```
String imageName = "sample.gif";
ImageIcon icon = new ImageIcon(imageName);
JOptionPane.showMessageDialog( ··· , icon);
```

JOptionPane.showMessageDialog メソッドは、画面にダイアログを表示するメソッドである。上の例のように記述すれば、"sample.gif"という .gif ファイルの画像イメージをダイアログに出力することができる。どちらの例もソースファイル中に画像ファイル名を記述しているが、ソースファイル中に記述してあるこれらのファイル名は、コンパイル後の .class ファイル中にも記述されている。したがって、必要な画像ファイルを認識するためには、.class ファイルを解析して、これらの画像ファイル名に

当たる文字列を探し出せばよい、ということになる。もっとも単純な方法としては、.class ファイル中に出てくる、ファイル名になりうる文字列を全て取り出し、そのファイルが実際に存在するかどうか、すべて探し出す、という方法が挙げられる。具体的には、.class ファイルの中のクラス情報が記されている場所を調べ、例えば、".gif"などの画像ファイルの拡張子に当たる文字列を見つけたら、それを含む文字列を画像ファイルの名前であると認識する。そして.jar ファイルにまとめる際に、その文字列の表すファイルが実在すれば一緒にまとめるし、存在しなければ無視する、といった作業をする。この方法ならば、プログラマの負担は、".gif"などの拡張子に指定だけに抑えられるが、画像ファイル名を見落としてしまう可能性がある。先ほどの例にもあるように、プログラマが Flyingware を用いてプログラムを書き、その中で画像ファイルを使おうとすれば、例えば次のように記述して、イメージオブジェクトを作成しなければならない。

例３：

```
ImageIcon icon = new ImageIcon("sample.gif");
```

ソースプログラムにこのように記述してあれば、そのクラスファイル中にも"sample.gif"という文字列が存在し、それを見つけることができる。しかし次のような例だと、.class ファイルの中には"sample.gif"という文字列は存在せず、この文字列の表すファイルは見落とされてしまう。

例４：

```
String str = "sam";
ImageIcon icon = new ImageIcon(str+"ple.gif");
```

この例４は、意味のないただ意地悪なだけの例だが、プログラム中で画像ファイルを使う上で、似たような状況になることは十分考えられる。この問題を解決するためには、プログラマにある程度の負担を背負ってもらう必要がある。つまり、画像ファイルをプログラムコード中で定義するときには、何らかのルールに従ってもらうようにする。その上で.class ファイルを調べればよい。「例４のような方法でなく、例３のような方法で書く。」というのはそのもっとも単純なルールである。このルールをきちんと決めておけば、必要な画像ファイルの認識は可能になる。これより先は、プログラマはこのルールに従ってプログラムを記述しているものとして話を進めていく。

それでは、画像ファイルの認識の具体的な方法を述べる。先ほども述べたように、最も単純な方法としては、.class ファイルの中を調べていき、何らかのファイル名を見つけたらそれを一緒に送る、というものがあげられる。例えば.gif ファイルを探したければ、.class ファイルの中に ".gif"という文字列が出てこないか探せばよい。".gif"という文字列を見つけたら、"."の１つ前の文字、２つ前の文字、と調べていき、ファイル名として適当でない文字になったら、その文字列の検索は終了する。この方法で、一つの疑問がわいてくる。もし複数の、ファイル名にあたる文字列が繋がっていたのでは、それぞれのファイル名を識別できないのではないか、ということである。例えば、"image1.gif", "image2.gif"というファイル名をソースファイル中に記述した場合、.class ファイル中で"image1.gifimage2.gif"というように、二つのファイル名が繋がっていたのでは、それぞれのファイル名を識別できず、これらを１つの.gif ファイルとして認識してしまう恐れがある。しかし、このようなことは実際には起こり得ない。.class ファイル中では、適当なファイル名の前後には、下記の例のように、何かしらのファイル名になりえない文字が入っているので、この点は心配する必要はない。

例１：image1.gif『image2.gif
例２：image1.gif　image2.gif

したがって、ファイル名になりえない文字列に当たった時点で、それまでの文字列をファイル名の候補として挙げることができる。

さらに、.gif ファイルに限らず、画像ファイルや音声ファイルなど様々な種類のファイルを認識することを考える。今述べた方法では、あらかじめ指定してある拡張子のファイルしか認識できないか、あるいはFlyingware アプリケーションのプログラマが、識別して移動させたいファイルの拡張子を指定しなければならない。例えば、.class ファイルのほかに、送りたいファイルは.gif ファイルと.wav ファイルで、という情報をアプリケーションプログラマどこかに記しておくことになる。しかしこれではプログラマの負担が増えてしまうことになるし、Flyingware の機能拡張の目標は、どのような種類のデータに関しても、必要なデータだけを自動的に識別して移動させることにある。そこで、どのような種類のファイル名に関しても、.class ファイルの中から見落としなく識別できるように、次のような方法を提案する。それは、.class ファイルの中から、何らかのファイル名である可能性のある文字列はとりあえずピックアップ

しておく。 という方法である。.class ファイルの中のある文字列がファイル名であるかどうかの判別には、まず、その文字列が"."を含むかどうかで判別できる。ファイル名には必ずファイル拡張子がついているので、ファイル名は必ず"."を含む。さらに"."の前後の文字が、ファイル名になりうる文字であれば、それらの文字列はファイル名になりうる、ということが言える。具体的には、以下のようなアルゴリズムを提案する。

1. .class ファイルの中に'.' の文字を見つけたら、それはファイル名の一部である可能性があるので次のステップに進む。

2. その'.' の前後の文字を調べ、その文字が両方ともファイル名になりうる文字であれば、その文字列はファイル名の一部である可能性があるので次のステップへ進む。

3. その'.' の前後について、ファイル名になりうる文字が続く限り文字を調べていき、それをファイル名とする。

4. そのファイル名のファイルが実際に存在すれば.jar ファイルに追加して一緒に送り、存在しなければ無視する。

.class ファイル中には、様々な情報が記述されている。このアルゴリズムを用いれば、例えば abc.dfg というような、ファイル名ではない文字列も、ファイル名の候補としてピックアップする。しかし、そのファイル名のファイルが実際に存在しないこでエラーを起こして、実行が止まってしまったのでは意味がない。それらの関係ない文字列は無視する必要があるのである。そしてこの方法で問題となるのが、ファイルが存在する場所はどこか、ということである。Flyingware が様々な計算機上を移動中は、全ての必要なファイルは.jar ファイルの中に格納してあるので問題はないが、最初に Flyingware を移動させるときが問題となる。探し出して送りたいファイルが、いたるところにちらばっていたのでは探し出せない可能性があるため、それらのファイルは、一ヶ所に集まっている必要がある。つまり、実行する Flyingware のアプリケーションファイルと同じディレクトリや、そのサブディレクトリに集まっている必要があるのである。また、送る必要のないファイルまで一緒に送ってしまう可能性もある。次の例は、移動先で"sample.gif"の画像ファイルを表示する、Hello クラスの.class ファイルをこのアルゴリズムで調べた結果ピックアップされたファイル名候補（左）と、ファイル名候補の中から実在するファイルを.jar ファイルにまとめた後の、その.jar ファイルの内

容（右）である。

```
> java FileSearch Hello.class
Hello.java
sample.gif
mail.csg.is.titech.ac.jp
csg.is.titech.ac.jp
is.titech.ac.jp
titech.ac.jp
ac.jp
csg.is.titech.ac.jp
is.titech.ac.jp
titech.ac.jp
ac.jp
```

```
> jar tf newfile.jar
Hello.java
sample.gif
```

.class ファイル中には、移動先や移動元となるホストのアドレスなどが記述されていて、これらの文字列もこのアルゴリズムによってピックアップされているのが、左の例からわかる。そして.jar ファイルにまとめる際、関係のない文字列は無視されているが、送る必要のない.java ファイルまでまとめられているのが右の例からわかる。これは、.class ファイルの中にはそれのソースファイル名の文字列が必ず記述されているということと、Hello.class ファイルと同じディレクトリに Hello.java ファイルが存在したためである。この時、ソースファイルをまとめることは想定しておらず、関係のないファイルを認識してしまったことになり、これには何らかの対処が必要となる。あらかじめソースファイルは送らないように記述しておくか、あるいは、移動させるファイルを集めてあるディレクトリには、移動させる必要のないファイルは一切置いておかない、といった方法が考えられる。後者の方法ではユーザーの負担が増えてしまうが、ソースファイルを送りたい状況というのも考えられるため、我々は後者の方法をとるようにした。しかし、.class ファイルや OBJECT.IMG などのファイルは、Flyingware に元々ある機能で識別して移動できることが決まっているので、これらのファイル名に関しては、見つけたら無視するような方法をとった。
またプログラムを記述する上で、コメントととして、ファイル名になりうる文字列を記述することがある。

```
//this.is.comment
```

上記の文字列は、前述のファイル名探索アルゴリズムを用いれば、ファイル名として認識される。しかし、このようにコメントとして記述された文字列は、.class ファイル中には記述されない。したがって、コメントとして不要なファイル名を記述したとしても、誤ってそのファイルを移動してしまうことはない。

必要なファイルを認識する他の方法として、先ほども少し触れたが、Flyingware のアプリケーションプログラマがあらかじめが情報を用意しておく、という方法も考えられる。.class ファイルのほかに、移動させたいファイルの拡張子を指定しておき、それらのファイルのあるディレクトリを指定しておくなど、移動させるファイルに関する情報をどこかに記述しておく。そうすれば先ほどのファイル名探索の方法のように、必要のないソースファイルを探し出してしまう、といったことも起こらず、確実に必要なファイルだけを選びだすことが可能になる。しかし、先ほどの方法に比べて機械側の手間が省ける反面、ユーザー側の手間が増えることということが問題になる。従来の Flyingware の利点は、Flyingware が自動的に必要なクラスだけを選び出して転送してくれる点であるし、Flyingware の機能拡張後の利点は、必要なデータファイルだけを自動的に選び出して転送することにある。ユーザー側の負担をあまり増やすような方法は取るべきではないと判断したため、この方法は採用せず、先ほどのアルゴリズムを採用した。

先ほどのアルゴリズムを用いてファイル名を探索すれば、.class ファイルの中から必要なデータファイルだけを認識して、移動させることができる。ただし、これまで述べてきたことは、機能拡張後の Flyingware を用いて作成したアプリケーションを、最初に飛ばす場合の話である。Flyingware に限らず、モバイルエージェントを使用している際には、移動先で、データが必要なくなることがある。移動中に不要になったデータをいつまでも所持して移動していたのでは、転送量を無駄に増やしてしまうことになる。移動中に不要になったデータは適時捨てていける機能が必要である。次節では、不要になったデータを適時捨てていく方法について述べていく。

4.3　不要になったデータを捨てる方法

前節まででは、必要なデータを探し忘れのないよう認識すること、および、最初の移動時に、不要なデータを一緒に転送しないことに関して述べてきた。これより先は、Flyingware はこれらの機能を追加したものとして扱う。Flyingware は、必要なデータファイルを忘れずに移動することは可能になったが、移動中に必要でないデータファイルを転送してしまう可能性がある。例えば、次のような、２度移動する Flyingware のアプリケーションを考える。このアプリケーションは最初の移動先で、画像データを表示し、２度目の移動先ではなにもしない。つまり、最初の移動時には画像データを所持して移動する必要があるが、２度目の移動時には画像データは必要なくなる。これらの振る舞いを、１つの .class ファイル（ソースファイル）に記述してある場合、２度目の移動時にも画像データを所持して移動してしまうのである。なぜなら、現段階の Flyingware は、自分自身の .class ファイルを解析して、そこに登場するファイル名で実在するものに関しては全て移動してしまうからである。たとえ最初の移動後に画像データが不要になったとしても、.class ファイル中にその画像データの情報が記述されているため、現段階の Flyingware はその画像データを必要なものとして認識してしまうのである。現段階での Flyingware の、このときの振る舞いを整理すると以下のようになる。

1. １度目の移動命令を受ける。
2. 自分自身の .class ファイルを解析し、移動先で画像データが使用されることを認識する。
3. 画像データを所持して、１度目の移動を実行する。
4. 移動先で画像データを使用し、その後２度目の移動命令を受ける。
5. 自分自身の .class ファイルを解析する。
6. 次の移動先では画像データは使用されないはずだが、.class ファイル中には画像データの情報が記述されているので、その画像データを必要なものと認識する。
7. 画像データを所持して２度目の移動を実行する。

何度も述べるが、Flyingware の機能拡張の目的は、どのような種類のデータファイルに関しても、必要なものだけを所持して移動することに

ある。不要になったデータファイルは適時捨てていけなければならない。もし単純に、この問題を解決しようと考えた場合、.class ファイル中の、画像データに関する情報を消去する、という方法が考えられる。最初の移動先で画像データを使用した後、その画像データが不要になるので、その時点か、もしくは次の移動時に、.class ファイル中から画像データの情報を消去するのである。しかし、この方法は実現が難しい。そこで我々は、従来の Flyingware の特長を生かした、不要なファイルを捨てていく方法を提案をする。Flyingware には、不要なクラスを捨てながら移動できるという利点がある。そのため、この利点を生かせば、不要なデータファイルを適時捨てていく方法が、今のところ２通り提案できる。この２つの方法を、便宜上のため、それぞれ ”方法 1”、”方法２” と呼ぶことにする。

方法 1　１つ目の方法は、画像ファイルごとにクラスを用意する、という方法である。これはもっと詳しく説明すると、不要になるであろう段階が一緒の画像ファイルごとに、クラスを用意する、ということである。Flyingware は、複数の計算機を移動するうちに、ある .class ファイルが不要になることがある。そうすると Flyingware はその .class ファイルを捨てて、次の計算機に移動をする。その場合、その捨てられた .class ファイルは、二度と解析されないことになる。ということは、もしその捨てられた .class ファイルに画像データの情報が記述されていたとしても、その情報は二度とピックアップされないということになる。例えば、前章の図 3.1 のような例に、画像データを用いることを考えた場合、図 4.1 のようになる。

細かい個所は省いてあるが、この図 3.1 のアプリケーションは、計算機を移動しながら、それぞれの計算機上で画像を表示する、というものである。そのため、それぞれの .class ファイル中には、画像ファイル名が記述されている。このアプリケーションは、前章の例と同様、MorningTest クラスから実行される。必要なファイルを認識する手順は基本的に前章の例と同じであるが、以下の作業が追加される。まず、Morning クラスの .class ファイルを解析している際に、"sunrise.jpg"という文字列を認識する。さらに、Morning クラスでは Afternoon クラスが参照されているのを発見し、Afternoon クラスの .class ファイルを解析している際に"sunshine.jpg"という文字列を認識する。同様に Evening クラスの .class ファイルを解析

図 4.1: 画像ファイルの探索

中に"sunset.jpg"という文字列を認識する。そして、これらの認識した文字列のファイルが存在すればそれらを.jar ファイルにまとめ、移動させる。つまり、最初の移動では"sunrise.jpg","sunshine.jpg","sunset.jpg"の３つの画像ファイルが一緒に移動される。２回目の移動時には、Afternoon クラスの.class ファイルの解析からファイル探索が行われ、Morning クラスが捨てられることはすでに述べた。Morning クラスが捨てられるのは、Morning クラスの.class ファイルの解析が行われないためである。Morning クラスの解析が行われないということは、"sunrise.jpg"という文字列は、他のどの.class ファイルを探索しても発見されないので、この画像ファイルは移動されないことになる。つまり、この時点で"sunrise.jpg"のファイルは捨てられたことになる。同様に、Afternoon クラスが捨てられた時に、Afternoon クラスで記述されている"sunshine.jpg"という名前のファイルも捨てられることになる。このように、.class ファイルを捨てたとき、その.class ファイルに記述されていた画像ファイル名は

もう認識できなくなるので、画像ファイルごとに.class ファイルを用意しておけば、.class ファイルを捨てる時に画像ファイルもまとめて捨ててしまえるのである。この方法ならば、画像ファイル以外にも、適切に、必要のないデータファイルを捨てていくことができる。しかし、このデータ捨てる段階ごとに.class ファイルを用意するといった方法は、プログラマにとって相当な負担となる可能性がある。作成するアプリケーションによっては、そのアプリケーション（の一部）を一つの.class ファイルではなく、データを捨てる段階ごとの.class ファイルに分けて記述するという作業が困難な場合があるからである。例えば、移動先でのユーザの応答の結果によって、不要になる画像ファイルが変わる場合などである。３つの画像ファイルを積んで相手ホストに移動し、相手ユーザの選んだ画像ファイルを捨てるようなアプリケーションを考える。どの画像ファイルを捨てるかが最初にわからないため、画像ファイルを捨てる段階ごとにアプリケーションプログラムを分けて記述することができない。３つのどの画像が捨てられても良いように３通りのプログラムを用意しておくことも可能だが、これは相当な無駄になる。”方法１”は、どの段階でどの画像ファイルを捨てるかが最初にわかっていなければ、使用することが困難である。また、データを捨てる段階がが多くなれば、その分、Flyingware のアプリケーションプログラムの記述量も増大してしまうといった欠点もある。

方法２　”方法１”では、データを捨てる段階が多くなればなるほど、プログラマの負担が増大してしまうという欠点があった。そこで、データを捨てる段階ごとに.class ファイルを分けて記述する必要のない、”方法２”の方法も提案する。”方法２”は、従来の Flyingware で画像データを移動する場合と似ている。従来の Flyingware で画像データを移動しようとしたら、最初に、必要な画像データはオブジェクトとして所持しておく。そして、画像データが不要になったら、そのオブジェクトを削除するようにプログラマが明示的に指示する。このように、不要になった画像データを明示的に削除するようにすれば、画像データを捨てる段階がいくら増えようが、その段階ごとに.class ファイルを分けて記述する必要がない。この方法を実現するにあたり、注意しなければならないのは、移動させるべき.class ファイル中に、ファイル名が記述されていなければならない、ということである。必要なデータファイルの認識は前節の方法で行うため、移動する.class ファイル中にファイル名が記述されて

いなければ、そのファイルを移動することができない。従って、以下のような例ではデータファイルを移動することができない。

```
public class System2 {
    pubilc String imagename;
        ⋮
    pubilc void resume(String[] args) {
        imagename = null;
            ⋮
        flight.fly(this, ”resume”);
            ⋮
    }
}
```

```
public class System2Test {
    pubilc static void main(String[] args) {
        System2 s2 = new System2();
        s2.imagename = ”sample.gif”;
            ⋮
        s2.flight.fly(s2, ”resume”);
            ⋮
    }
}
```

このアプリケーションは`System2Test`クラスから実行する。このアプリケーションの狙いは、最初の移動時には`"sample.gif"`を所持して移動し、その移動先で画像ファイル名を削除して二度目の移動時に画像ファイルを捨てていく、ということだが、これはうまくいかない。なぜなら最初の移動時には`System2`クラスを移動させるため、`System2`クラスの`.class`ファイルから解析を始めるのだが、そこには`"sample.gif"`の文字列は登場せず、この`.gif`ファイルは移動されないことになるからである。そのため、移動させるクラスの`.class`ファイル中には、移動させるべきデー

タファイル名が記述されていなければならないが、以下のような例でも問題がある。

```
public class System2 {
    pubilc String imagename = ”sample.gif”;
            ⋮
    pubilc void resume(String[] args) {
        imagename = null;
                ⋮
        flight.fly(this, ”resume”);
                ⋮
    }
}
```

```
public class System2Test {
    pubilc static void main(String[] args) {
        System2 s2 = new System2();
                ⋮
        s2.flight.fly(s2, ”resume”);
                ⋮
    }
}
```

先ほどの例を少し変えたこの例ならば、移動させる.class ファイル中に`"sample.gif"`というファイル名が登場するため、このファイルを移動させることができる。しかし、この画像ファイルを捨てていくことができない。オブジェクトを削除しても、`.class` ファイル中の`"sample.gif"`という文字列は削除できないからである。従って、移動させるアプリケーションの`.class` ファイルには、将来不要となることが予測されるデータファイル名を記述してはならない。”方法 1”のように、捨てることのできる`.class` ファイルにデータファイル名を記述し、その`.class` ファイルごとデータファイルを捨てていく、という方式を取る必要がある。この

ことを実現するために、”方法２”は、Java におけるクラスの継承関係を用いた。この方法ではまず、画像ファイルを定義するためのスーパークラスを用意しておく。そして、そのサブクラスで実際に画像ファイルを定義していく。例えば、以下のようなクラスを用意する。

```
public class MyImage {
    pubilc String imagename; {
}
```

```
public class SubImage extends MyImage {
    super.imagename = ”sample.gif”; {
}
```

そして、以下のようなアプリケーションを作成し、実行する。

```
public class System2 {
    pubilc MyImage image;
            ⋮
    pubilc void resume(String[] args) {
        image = null;
                ⋮
        flight.fly(this, ”resume”);
                ⋮
    }
}
```

```
public class System2Test {
    pubilc static void main(String[] args) {
        System2 s2 = new System2();
        s2.iamge = new SubImage();
                ⋮
        s2.flight.fly(s2, "resume");
                ⋮
    }
}
```

System2 クラスで、MyImage オブジェクトの中に入っているのは、実際には MyImage クラスのサブクラスのものである。Flyingware には、移動時にオブジェクトの解析を行う機能が追加された。そのため、System2 クラスのオブジェクトを調べれば、それが SubImage オブジェクトであることがわかり、SubImage クラスも移動先で必要なクラスだと認識する。そして、SubImage クラスに記述されている画像ファイルも一緒に移動できるのである。さらに、System2 クラスには、SubImage クラスに関する記述がないため、このオブジェクトを削除すれば、SubImage クラス及び SubImage クラスに記述されている画像ファイルをその場に捨てていくことができるのである。少々面倒な方法ではあるが、継承関係を用いないと、適切にデータファイルを捨てていくことができない。例えば、上の例で継承関係を用いていないと、以下のようなアプリケーションになる。

```
public class MyImage {
    pubilc String imagename = "sample.gif"; {
}
```

```
public class System2 {
    pubilc MyImage image;
        ⋮
    pubilc void resume(String[] args) {
        image = null;
            ⋮
        flight.fly(this, "resume");
            ⋮
    }

}
```

```
public class System2Test {
    pubilc static void main(String[] args) {
        System2 s2 = new System2();
        s2.iamge = new SubImage();
            ⋮
        s2.flight.fly(s2, "resume");
            ⋮
    }
}
```

この場合、MyImageオブジェクトを削除しても、System2クラスにはMyImageクラスの記述があるため、MyImage クラス及び"sample.gif"を捨てていくことができない。多少プログラムの記述量が増えるが、データファイルを適切に捨てていくためには、クラスの継承関係を用いた方法をとる必要がある。そしてこの方法を用いれば、アプリケーションプログラムを捨てる段階ごとに分けて記述する必要がなくなる。ただ、最初にSystem2Testクラスを使ってSystem2 クラスを飛ばす方法は、"方法１"の手法である。これは移動させる.class ファイル中に画像ファイル名などを登場させないためで、必ず使用しなければならない方法である。つまり、"方法２"は、"方法１"と併用する必要があるのである。

不要になったデータファイルを適時捨てていく方法として、”方法１”と”方法２”を提案したが、これらの方法は、状況に応じて使い分けたほうが良い。データファイルを捨てる段階が多い場合や、その段階がプログラム記述時にわからない時には、”方法２”（”方法１”と併用）を使用する必要があるが、”方法１”のみの方がプログラマの負担を抑えられる場合もある。データファイルを捨てる段階が極めて少なく、一度に多量のデータファイルを捨てる場合などである。例えば、多量の画像ファイルを所持して相手ホストに移動し、そこで全ての画像ファイルを捨てて元のホストに戻ってくるアプリケーションなどは、”方法１”を用いた方が、プログラムの記述量は抑えられる。以上の２つの方法を使用することにより、拡張後の Flyingware は、必要な画像ファイルなどを自動的に移動し、不要になった時点で捨てていけるモバイルエージェントとなった。

第5章　アプリケーションの例と実験

前章では、不要なクラスを捨てながら移動できるモバイルエージェントFlyingwareの機能を拡張する方法を提案した。拡張後のFlyingwareは、様々な種類のデータファイルを所持して移動し、不要になったものはそのつど捨てていけるモバイルエージェントである。この章では、拡張後のFlyingwareを用いた典型的なアプリケーションの紹介と、拡張後のFlyingwareが適切にデータを捨てていく様子を紹介する。

5.1　実験

我々は、必要なデータファイルを所持して移動し、不要になった時点で捨てていけるモバイルエージェント、Flyingwareを提案し、実装した。そして、このFlyingwareが期待通りの動作をしていることを示すための実験を行った。おおまかな実験内容は、以下の通りである。

- 同じ大きさの画像ファイルを十個用意し、最初にそれらを所持して移動する。
- 一回移動するたびに、画像ファイルを一つずつ捨てていく。
- 移動のたびに、必要なデータファイルを.jarファイルにまとめる時間と、.jarファイルのデータサイズ及び内容を調べる。

上記の実験内容を、”方法１”を用いて実装したものと、”方法２”（”方法1”と併用）を用いて実装したものの二つを用意し、以下の環境、条件で実験を行った。

- CPU AMD AthlonProcessor1200Mhz
- メモリ261.600KB RAM

- OS Windows2000
- 画像データのサイズは全て４０ kb とする。
- アプリケーションのクラスは、画像を運ぶことを目的とした、簡易なものとする。
- .jar をまとめるのに要した時間は、それぞれ１０回ずつ計測し、その平均値を取った。

結果　図 5.1 及び図 5.2 は上記の環境、条件で実験を行った結果のグラフである。グラフの縦軸は .jar ファイルをまとめるの際に要した時間と、その .jar ファイルのデータサイズである。横軸は、Flyingware の移動回数である。どちらものグラフでも、データサイズがだんだん減少していることから、データが順調に捨てられていっていることがわかる。また、それにつれて、ファイルをまとめる時間も減少していく様子がうかがえる。どちらの方法でも、移動先で必要なファイルはだんだん減っていく設計になっているので、それらをまとめる時間も減少して当然である。また、”方法１”、”方法２”どちらの方法を用いても、データの総量には大差がないように見える。これは最初の移動時のデータ総量、移動中のデータ総量、移動終了時のデータ総量、全ての場合について言えることである。しかし、これは移動の際にデータ圧縮を行っているためで、実際には、データ量には大きな開きがある。”方法１”のアプリケーションクラスの総ファイルサイズは、13.88kb で、１回の移動で 1.08kb の .class ファイルが捨てられていき、最終的には 1.77kb になる。”方法２”ではアプリケーションクラスの総ファイルサイズは 6.45kb で、一回の移動で 257b の .class ファイルが捨てられていき、最終的には 2.58kb になる。最終的なデータサイズは”方法１”の方が小さいが、最初のデータサイズは圧倒的に”方法１”の方が大きなものになっている。.class ファイルのサイズで 6kb 以上の差は、無視できるものではない。さらに、データサイズだけでなく、プログラムの記述も”方法２”の方が簡易であるので、このように移動回数が多い場合は、”方法２”を用いた方がユーザの負担は低減できる。ただし前章で説明したように、”方法１”を用いた方がデータサイズを抑えられる場合もある。移動回数が少なく、さらに一度に大量のデータを捨てていくアプリケーションの場合である。この場合のデータサイズなどを検証するため、１０個の画像データを所持して移動し、移動先で全て捨て、元のホストに戻ってくるアプリケーションを、”方法１”、”

方法２”の２通りの方法を用いて実験を行った。実験環境は前述の通りである。この実験についてはグラフの掲載はせず、結果だけ簡単に説明するが、この場合は”方法１”を用いた方が、総データサイズ、それをまとめる時間ともに少ない結果になった。なぜならこの場合、”方法１”のアプリケーションクラスの総ファイルサイズは 2.878kb で、”方法２”のアプリケーションクラスの総ファイルサイズは 4.137kb になったからである。このように、移動回数が少なく、一度に大量のデータを捨てていくアプリケーションの場合は、”方法１”の方がプログラムの記述量を抑えられる。また、移動回数が少なければ、”方法１”でも比較的楽にプログラムを記述することができる。以上のように”方法１”、”方法２”どちらが有効かは、アプリケーションによって変わる。そのため、Flyingware のアプリケーションプログラマは、この両方の方法を使い分けるべきである。

図 5.1: system1 の実験結果

図 5.2: system2 の実験結果

5.2　Flyingware を用いたアプリケーションの例

以上のように Flyingware では、自分自身を自動的に解析し、移動先で必要となる、.class ファイル以外の画像ファイルなどのデータファイルも転送することが可能になった。このことにより、旅行会社のパンフレットメールのようなアプリケーションの作成も可能になったのである。この Flyingware を用いたアプリケーションとして、我々は以下のものを実装したので、紹介する。

旅行のアンケートメール　あるグループで旅行をすることになったときに、Flyingware を用いて、グループメンバーの旅行可能日から希望旅行先までアンケートをとることが可能である。旅行の幹事は、旅行先の画像イメージと、見所や代金などのデータを積んだ Flyingware を移動させる。Flyingware は、それぞれのホストでどこに行きたいかのアンケートをとり、集計結果によってだんだん行き先を絞り、不要になったデータ

を捨てていく。このアプリケーションの実装には、”方法２”（”方法１”と併用）を用いた。移動先でのユーザの応答により捨てるべき画像ファイルが変わるため、最初に、どの段階でどの画像ファイルを捨てるかがわからないためである。さらに、このアプリケーションには、旅行可能日を入力してもらい、幹事のホストに戻ってきて、メンバーに共通の旅行可能日を表示する機能を追加することも可能である。従来の電子メールでこのようなアンケートを行う場合、幹事はメンバー全員にアンケートメールを送り、その返答メール全てに目を通し、データを集計しなければならない、ということになる。また、Flyingware を用いたほうが、より視覚的に見やすいものになる。視覚的に見やすいという点では、Web ページを用いれば視覚的にかなり見やすいアンケートページを作成できる。しかしその場合、３章でも説明したように、そのページの閲覧中はインターネットに接続された環境を必要とする他、第三者に閲覧される可能性もある。

第6章　まとめ

我々は、従来のFlyingwareを拡張して、様々な種類のデータファイルを所持して移動し、不要になった時点で捨てながら移動できるモバイルエージェントの提案と実装を行ってきた。まず移動時に自分自身の.classファイルを解析して文字列探索を行うことにより、.classファイル以外の必要なデータファイル名を認識し、移動することが可能になった。また、オブジェクトの解析を行うことにより、クラスの探し忘れをなくし、より多くのクラスを移動することを可能にした。そして、不要になったデータファイルを捨てていく方法を2通り提案した。この拡張後のFlyingwareが既存のモバイルエージェントシステムよりも優れている点は、プログラムコードの転送方式と、移動できるデータの種類にある。既存のモバイルエージェントシステムでは、ほとんどのものが動的ローディングによりプログラムコードを転送するため、ネットワークに常時接続の環境でなければ使用することが困難であった。また、静的ローディング方式を採用しているモバイルエージェントシステムでも、移動先で必要なプログラムコードを移動するためには、ユーザが明示的に必要なものを指示しなければならなかった。Flyingwareでは、自分自身を解析にすることにより、移動先で必要なプログラムコード及びデータファイルを自動的に探し出し、一括して転送する、理想的転送方式を実現することができる。そのためFlyingwareは、ネットワークに常時接続ではない環境や、ネットワークの不安定な環境でも使用することができる。さらにFlyingwareは、必要のなくなったプログラムコード及びデータファイルは適時捨てていける機能を持ち、通信コストを極力低減することができる。また、電子メールを用いて移動するため、既存のモバイルエージェントが必要とする実行時の特別な環境も必要がなく、様々な環境で手軽に使用できるモバイルエージェントであると言える。このFlyingwareの今後の課題としては、移動中にはデータファイルを追加できないということが挙げられる。Flyingwareは、最初に所持したデータファイルを捨てながら移動するだけで、移動中のホストで新たな画像ファイルを受け取る、といっ

たことはできない。このことが可能になれば、アプリケーションの応用範囲はさらに広がると思われる。

参考文献

[1] IBM, Aglets Software Deveropment Kit, Online publishing, URI http://www.trl.ibm.com/aglets/index-j.html

[2] Tomen Information Systems, Object Space, Online publiching, URI http://www.tomen.co.jp/tisco/Voyager/vgrCompFeatContent.-htm

[3] GeneralMagic, Online publiching, URI http://www.generalmagic-.com/

[4] 佐藤一郎, AgentSpace: モバイルエージェントシステム, 日本ソフトウェア科学会 Workshop on Muti Agent and Cooperative Computation, (MACC'98), December, (1998)

[5] 佐藤一郎, AgentSpace Online publishing, URI http://research.-nii.ac.jp/ ichiro/agent/index.html

[6] Kazuyuki Shudo, Moba: Mobile Agent Facilities on JavaTM Language Environment, Online publishing, URI http://www.shudo.net-/moba/index_j.html

[7] Toshiba, Plangent, Online publishing, URI http://www2.toshiba.-co.jp/plangent/index_j.htm

[8] Shinya Wada, Shigeru Chiba, Kouzou Itano, 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 会大, 筑波大学電子情報工学系, HLLA テクニカルノート HLLA-585 Sep 19, 2000

[9] Toramatsu Shintani, Tadachika Ozono, Naoki Fukuta, モバイルエージェントの応用 -マルチエージェントシステムのためのモビリティの利

用-, 人工知能学会誌 Vol.16, No.4, pp.488-493 人工知能学会誌 2001 年 07 月.

[10] Kazuhiko Kato, 新しいネットワーク基盤とグローバルネットワークへの応用 -ソフトウェアをインターネットへ解き放つ- 科学技術振興事業団 さきがけ研究 21 発表会予稿集，2000 年 12 月.

[11] Katsumi Kishimoto, 移動エージェントを使った Web キャッシュプロキシシステム, pp.4–10

[12] Youichi Muraoka, モバイルエージェントの性能比較, Online publishing, URI http://www.seckey.net/docs/agent/20001004/-rejime.html

[13] 知的モバイルエージェント, Online publishing, URI http://www-.icot.or.jp/FTS/REPORTS/H10-reports/AITEC9903Re2_Folder/-AITEC9903R2-sec3-6.htm|