修 士 論 文

モバイルエージェントの
動的部品交換方式の実験的検証

指導教官　権藤克彦 助教授

北陸先端科学技術大学院大学
情報科学研究科情報システム学専攻

松原 剛

2000 年 02 月 15 日

要 旨

本研究では、MobileAgent に自己管理能力を高める機構を提案及び実装する。

MobileAgent とは、内部状態を保持したままホスト間を移動できるプログラムである。しかし、その移動性という特性のため管理者の手の及ばない場所で活動することが多く、急激な環境変化に耐えられなくなることが多い。

例えば、決められた Web サーバを巡回し、HTML ファイルを読み取り、欲しい情報のみを抽出し mail で転送する機能を持つ情報収集エージェントの場合では、HTML の構成等を変えられると処理の続行ができなくなる。従来のアプリケーションならば管理者が抽出ポリシーを変えれば良かったが、MobileAgent の場合は管理者が直接手を加えられる場所で活動しているとは限らないため、保守を行うことが困難になる。そこで、エージェント自身に自己管理を行わせる研究が進められている。

本研究では自己管理の手段として「エージェントに自己の一部を組替えさせる」というアプローチをとる。とりわけエージェントの一部となる”部品”の配布方法に焦点を当て、エージェントの特性を生かした部品配布方法を提唱し、従来のアプリケーションに適用されている配布方法と比較検討する。

従来の部品配布方式には、エージェントに新しい部品を送りつける push 方式やエージェントに部品を取ってこさせる pull 方式がある。しかし、これらの方式の実現にはネットワークコネクションの数が増えるといったデメリットがある。そこで、本研究では、エージェント同士がお互いに保持している部品を比較し交換することによって部品を配布する方式（比較交換方式）を提案し、実装する。

実装には Java ベースの MobileAgent システムである AgentSpace を用い、実際に例題を実装し、本研究の方式を実験し従来の方法と比較した。

目 次

図 目 次

表 目 次

第 1 章

はじめに

本章では本研究の背景、目的及び特色を述べ、本論文全体の構成を述べる。

1.1 本研究の背景

本節では、本研究のテーマである MobileAgent 技術の生まれてきた過程、及び問題点を述べる。

安価な計算機の普及により処理能力や情報が分散して存在するようになり、また、それと並行してネットワーク形態の多様化が起こり、MobileAgent 技術が生まれた。しかし、MobileAgent は移動性と独立性という性質のため、エージェントのオーナの手を離れて活動している時間が大半を占める。そのため、遠隔地で活動中に、処理を行うべき対象になんらかの変化が加えられ処理を継続できなくなったとしても、エージェントはオーナが直接手を加えることができる場所にいるとは限らず、管理することが困難になるといった問題が生まれた。

1.1.1 計算機資源及び情報の分散化

近年のワークステーションやパーソナルコンピュータのめざましい普及ぶりには目を見はらされる。また、高価な計算機ほど単価当たりの性能が上がるという従来の常識が通用しなくなり、安価なマシンが高価なマシンに接続するといった形態より、複数の安価なマシンで協調作業を行う形態の方が効率がよくなるケースも増えてきた。一方、地理的に広く分散している組織では各地域で必要な計算設備をすでに個々別に拡充してきている。

こうして、計算機資源や情報が分散して存在するようになった結果、様々な問題も生まれた。例えば、各組織で同じ情報を保持せねばならないケースや逐次実行を行わなければ

ならない処理などには同期や協調といった概念を持たせる必要性が生じた。

この問題の解決には、定期的な情報のミラーリングやサーバクライアント型のコネクションによる通信などの方法が利用されているが、大量の情報を転送したり、余計なコネクションを張る必要性があり、そのオーバーヘッドは情報が大きいほど、また処理の複雑さが上がるほど増える。そこで、なるべく転送やコネクションにかけるコストを少なくしたいという要求が生まれた。

1.1.2 非継続的接続の増加

計算処理能力や情報の分散化と共にネットワークの多様化が進み、現在では、ワークステーション、パーソナルコンピュータのみならず、モバイル端末や情報家電等、様々な機器がネットワークに接続するようになった。しかし、これらの機器は継続的にネットワークに繋がっているわけではなく、むしろ必要な時にのみネットワークに接続するといったものが多い。しかも、接続している期間に課金されるといったケースが多いため、ネットワーク接続時間はなるべく短い方が好ましい。そのため、長時間ネットワーク接続を必要とするアプリケーションの実行が困難になるといった問題が生じた。

そこで、携帯端末ではネットワークの接続時に要求を行い、その処理を、ネットワーク上のホストに委託し、後にネットワークに繋がった時に結果だけを受け取るといったサービスの要求が生まれた。

1.1.3 MobileAgent 技術の登場

これらの問題を解決するために MobileAgent 技術が生まれた。MobileAgent は、現在の内部状態や実行状態を保持したままネットワーク上のホストを移動できるプログラムで、移動先のホストで前回の処理の続きを行うことが可能である。

MobileAgent の明確な定義は定まっていないが、一般的に以下の特徴を持つとされる。[12, 13, 14]

- **移動性**：プロセスがネットワーク上のホスト間を移動しながら処理を行うことができる。
- **局所性**：システム全体の情報を持つ必要はなく、局所的な情報のみで行動することができる。
- **自律性**：プログラム自身が自己完結していて、移動先で他からのアクションや刺激などが無くても、処理を維持継続できる。しかし、現段階では独立性といった程度

の意味合いが強く、エージェントのオーナーの手を離れた場所での稼動といった程度の意味に使われることが多い。

- **協調性**：エージェント同士が通信し合い、それによって自己の行動を変えることができる機能を持つこと。エージェントというひとまとめにされたアプリケーション同士がお互いに情報交換することを指す。

- **適応性**：例えば電源が落ちるといった実行環境の変化に対応して、適切な行動を選択できる。

本格的な MobileAgent の起源は 1994 年に J.White らにより開発された MobileAgent 記述言語とその実行環境である Telescript[9] である。以降いくつかのプロジェクトが起こった。しかし、これらのシステムは特定の OS 上でしか動作せず、移動性の実現には大きな壁となった。

その後、1995 年に Java が発表され、Java の実行環境は様々な OS 上に実装された。これにより、あらゆる環境へ移動できる MobileAgent の開発が可能になり、Java ベースの MobileAgent の開発が急速に広まった。

1.1.4 MobileAgent の問題点

- **自己メンテナンスの必要性**

  MobileAgent はその移動性や独立性といった性質のため、エージェントのオーナの手を離れて活動している時間が大半を占める。そのため、遠隔地で活動中に、処理を行うべき対象になんらかの変化が加えられ、処理を継続できなくなったとしても、エージェントはオーナが直接手を加えることができる場所に存在しているとは限らず、管理することが困難になるといった問題が生まれた。

  これらの問題の解決の手段として、エージェントを常に監視したり、エージェントを呼び戻すといったアプローチが考えられるが、そのためにはエージェントのランタイムあるいはエージェント同士の通信が不可欠になり、通信相手先のランタイムが常に存在していると限らない場合には結局その機構がうまく働かない。MobileAgent の局所性という特徴を生かしたい場合には不向きであると言える。

- **動的保守の必要性**

  現在、アプリケーションの保守は、一般的にアプリケーションを停止させて、変更を加え、再起動させて行うものが多いが、MobileAgent はネットワークに強く依存

しネットワークコネクションを張るものも多い。例えばプロキシの動作を行うエージェントの場合、複数のクライアントから接続要求を受け、ネットワークコネクションを常に張っている状態になる場合があり、その途中でアプリケーションを停止し、新しいプロトコルに対応させる保守を行うとネットワーク接続が切れてしまいサービスの質の低下に繋がる。そのため、なるべくネットワーク接続を切らずに保守を行いたいという要求が生まれた。

1.2 本研究の目的

本研究の目的はエージェント自体が自己の状態を管理し、自分で自分の保守を行える環境を実現することである。また、その際にはネットワーク接続や他のスレッドの処理は継続させたまま保守を行うことを目的とする。

保守の方法として、エージェントが自身の一部分を動的に組替える、動的部品交換という手段をとる。

特に、部品の配布方式に着目し、エージェントの移動性や独立性といった特色を生かした方法を考案する。また、従来のアプリケーションに適用されている配布方法と比較検討する。

従来の部品配布方式には、エージェントに新しい部品を送りつける push 方式やエージェントに部品を取ってこさせる pull 方式がある。しかし、これらの方式の実現にはネットワークコネクションの数が増えるといったデメリットがある。そこで、本研究では、エージェント同士がお互いに保持している部品を比較し交換することによって部品を配布する方式（比較交換方式）を提案し、実装する。

また、エージェントの動的部品交換に関しては実例に基づいた検証例が少ないので、実際に数種の例題について実装および検証し、どのような例に対してはどのような手法が効果的かという知見を得る。

1.3 本研究の仮定

ここで本研究で仮定している状況を述べる。

- 仮定環境
  本研究では以下の理由により Java ベースの MobileAgent システムを仮定している。
  - MobileAgent の主流は Java ベース

現在、Java の実行環境は多くの OS に移植されている。また、Java にはシリアライズ機能があり、現在の状態のスナップショットを取り易いという特徴がある。これらの機能は処理の移動や継続の際に非常に都合が良い環境を提供してくれた。このため、現在 MobileAgent の主流は Java ベースのものが大半となった。

- クラスロードの柔軟さ

  Java にはクラスロード方式に柔軟性があるのが特徴で、ファイルのみならず、メモリ上のデータ等からもクラスをロードすることができる。本研究が提案する部品交換方法は、クラスの動的なローディングと密接に繋がった交換方式を仮定しているため、必然的に Java ベースのものを対象としたものとなる。

- 用語の定義
  - **ランタイム**:MobileAgent の実行環境を指す。場ということもある。
  - **エージェント**:ランタイム上を移動し活動する実体を指す。
  - **部品**:クラス単位を指す。パーツと呼ぶ場合もある。バージョンには全順序関係があるとする。
  - **保守**:部品の交換を指す。
  - **動的**:エージェント自体を再起動することなく部品を組替えることを指す。
  - **エージェントオーナ**:エージェントを実際に使う者を指す。
  - **エージェントメーカー**:エージェントを作った者を指す。
  - **パーツメーカー**:エージェントパーツを実際に使う者を指す。

Java ではクラスの概念が重要で、ほぼ全ての操作がクラス単位で行われている。クラス単位より小さい粒度での操作は Java の実行環境に手を加えずには行いにくいので部品をクラス単位とした。

部品のバージョンに全順序関係を置いた理由は、本研究ではエージェント同士の比較交換による配布を前提としているため、部品のバージョンに全順序関係がない場合は交換の必要性の判断が困難になるためである。バグの修正や、処理対象のシステムのバージョンアップへの対応といった比較的単純に部品の上下関係を決定できるケースを想定している。

また、注意点として動的な組替えを行うにあたって、いくつかの制約がつく。例えば、古いインスタンスが処理を終えていない場合にクラスを組替える場合は注意が

必要である。本研究では動的交換のメカニズムのみを提供し、交換のポリシー（例えば交換時に古いインスタンスの処理が全て終わるまで待つ）の決定はユーザに任せている。

- 移動性、局所性の重視

  MobileAgent の特徴のうち、特に移動性と局所性に着目し、通信回数や通信量、転送量をなるべく少なくする方法を重視する。

  MobileAgent のクラスロード方式は大きく分けて動的ローディングと静的ローディングの２つに大別される。動的ローディングは実際にインスタンスを生成するときにはじめてクラスを転送する。静的ローディングは最初に全てのコードを転送しておき、必要になった時にその場でインスタンスを生成できる。

  動的ローディングではクラスが必要になるのがいつかわからないので、常にネットワーク接続が可能な状態にしておかなくてはならない。そのため、MobileAgent の局所性の特性を犠牲にしてしまう。よって、本研究では、後者の静的ローディングを行うランタイムにスポットを当てる。

1.4 本研究で提案するシステム

- 概要

  本研究では特に部品の配布方式に焦点を当てている。エージェント同士がお互いの保持している部品を比較し、自分の持っていない部品を持っていた場合はその部品を相手から取得する。

- 実現

  シナリオ

  1 移動後、同ランタイム上に存在するエージェントの部品リストを要求。同時に自分のリストも公開する。

  2 自分の保持していない部品を持っている相手に部品の要求。

  3 部品を受け取る。

以下に「時計」機能と「電卓」機能を持つエージェントの部品交換の例を示す。

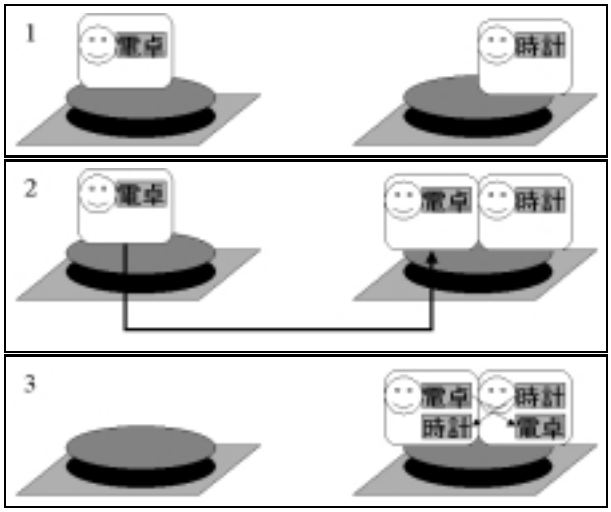

図 1.1: 部品交換イメージ

1.5 本研究の特色

本研究では特に動的組換えの部品の配布方式に焦点をあてている。

従来のアプリケーションでは稼動場所が限られており、その場所にあるコードをメンテナンスするという方法であった。交換部品はその場に送られ、その場でオーナーがメンテナンスを行うという「場」を主体とした配布交換であったが、本研究では「アプリケーション」を媒介にした配布方式であることが特徴的である。部品はアプリケーションが運び、アプリケーション自体が保守を行う。部品を得る行動もタイミングもアプリケーションが決定する。

「場」を主体とした配布の場合、「場」は固定的である必要がある。しかし、ネットワーク環境の複雑化により「場」が固定的ではなくなりつつあり、固定的な情報を埋め込んだアプリケーションは処理の継続が困難になることが多い。

本研究で提案する方式では「アプリケーション」を媒介とした配布方式なので、固定情報は極力少なくすることができ、より、柔軟な対応ができるようになりオーナやメーカーの保守作業を軽減させることが可能となる。

1.6 本論文の構成

本稿の構成を以下に挙げる。

- **第１章**：本研究でスポットを当てる MobileAgent 技術の生まれた背景と、移動するという性質のためオーナが保守を行い難くなるといった問題店があることを述べた。また、本研究の目的は動的部品交換という手段を用い、それらの問題を解決することであることを述べた。

- **第２章**：MobileAgent システムの基本的な概念、例えば、移動性や局所性といった特徴があることを述べ、後に既存の MobileAgent システムの特徴、例えばクラスロード方式の違い等について述べる。

- **第３章**：Java アプリケーションの一部を自動的組み変えて保守を行うための交換方式の概念と本研究における位置付けを行う。

- **第４章**：動的交換に使われる部品の配布方式について述べる。従来のアプリケーションに行われてきた push 方式や pull 方式といった方法の説明と固定情報の持ち歩く必要性といった特性の解説を行う。その後、MobileAgent の移動性や局所性という特性を生かした方式（比較交換方式）を提案する。

- **第５章**：４章で提案したエージェントの自動保守機構を実際に実装し、そのメソッドを利用する MobileAgent の実装方法を解説する。

- **第６章**：５章で実装したシステム上に Web のログアナライザの例題を実装及び実験し、従来の方式との特性を比較検討する。

- **第７章**：本研究全体をまとめ、実験により得られた結論を述べ、最後に本研究の展望を述べる。

第 2 章

MobileAgent システム

本章では MobileAgent システムの基本的な概念、例えば、移動性や局所性といった特徴があることを述べ、後に既存の MobileAgent システムの特徴、例えばクラスロード方式の違いについて述べる。

2.1 MobileAgent の生まれた背景

- 計算機資源及び情報の分散化

  今日、Web の普及等により情報が複数のホストに分散して存在する傾向にある。それに伴い、情報を一旦転送し処理を行うよりも、情報の存在するホストで処理を行い、結果のみを転送する方が効率が良くなるケースが増えた (図 2.1)。例えば後述の Web のサーチエンジン等が挙げられる (図 2.3.1)

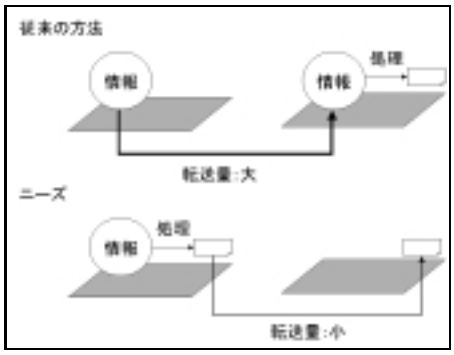

図 2.1: 遠隔処理で転送量を減らす

- 非継続的接続の増加

  モバイル端末や情報家電等は継続的にネットワークに繋がっているわけではなく、むしろ必要なときにのみネットワークに接続するといったものが多い。これらの端末上で実行するネットワークアプリケーションはネットワークが切れると、処理の中断を余儀なくされる。そこで、処理体系をネットワーク上のマシンに移動し、処理を継続させるといった要求が生まれた。(図 2.3)

2.2 MobileAgent の概念

MobileAgent はプロセスの内部状態や実行状態をもったまま、ネットワーク上の別のホストへ移動し、そこで処理を続けることができるプログラムである。

現在 MobileAgent の明確な定義は定まっているとは言いがたいが、一般的に以下の特徴を持つとされる。[12, 13, 14]

- **移動性**：プロセスがネットワーク上のホスト間を移動しながら処理を行うことができる。
- **局所性**：システム全体の情報を持つ必要はなく、局所的な情報のみで行動することができる。
- **自律性**：プログラム自身が自己完結していて、移動先で他からのアクションや刺激などが無くても、処理を維持継続できる。しかし、現段階では独立性といった程度の意味合いが強く、エージェントのオーナーの手を離れた場所での稼動といった程度の意味に使われることが多い。
- **協調性**：エージェント同士が通信し合い、それによって自己の行動を変えることができる機能を持つこと。エージェントというひとまとめにされたアプリケーション同士がお互いに情報交換することを指す。
- **適応性**：例えば電源が落ちるといった実行環境の変化に対応して、適切な行動を選択できる。

2.3 MobileAgent の利用例

現在、実用または実用が期待されている MobileAgent の利用方法を挙げる。

2.3.1 サーチエンジン

Web のサーチエンジンを作るときは、一度全ての HTML コンテンツを自ホストにダウンロードし、キーワードの抽出といった作業を行わなくてはならない (図 2.2左)。しかし、MobileAgent システムを使えば、抽出プログラムのみを送りつけ、キーワードの抽出結果と URL のみを自ホストに転送すればよく、データのネットワーク転送量を大幅に軽減することができる。そのため、ネットワークの接続状態が不安定な状況にも強い。

また、従来の方法で実現した場合、コンテンツのダウンロード中は常に自ホストを起動しておかなくてはならないが、MobileAgent を利用するならば自ホストの処理は終了しておいて、結果だけを待つこともできる (図 2.2右)。

まず、MobileAgent を HTML ファイルが有るサーバに送り、そこで処理を行いキーワードのデータベースを作り、データベースのみを転送する。

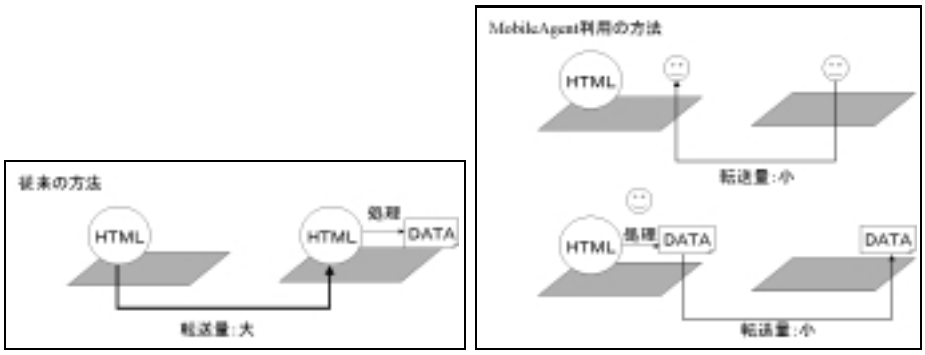

図 2.2: サーチエンジン

2.3.2 処理継続 Agent

今日、モバイル端末の利用が増えているが、モバイル端末の状態は電源供給状況やユーザの物理的な移動等の要求によって変化しやすい性質を持つ。そのため、長時間の作業等には向かない。そこで、現在行っている処理を他の端末に引き継がせるために MobileAgent 技術が利用できる。そして、モバイル端末の再起動時にまた処理を戻すことによって長期の作業も行えるようになる。(図 2.3)

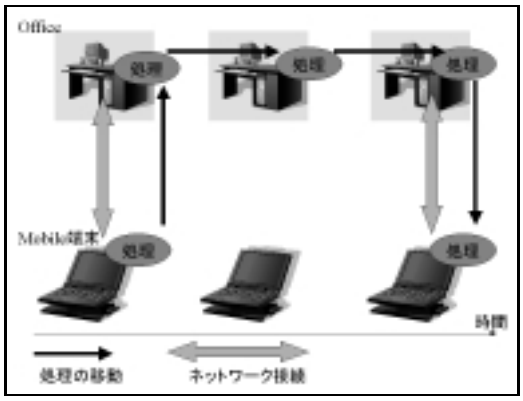

図 2.3: 処理の移動

2.4 MobileAgent の構造

Java ベースの MobileAgent の構造について解説する。

2.4.1 エージェントとランタイム

基本的にはエージェントのランタイムを各端末に起動し、エージェントがその上で稼動するという形になる。(図 2.4)

ランタイムはエージェントの移動や起動を管理する。

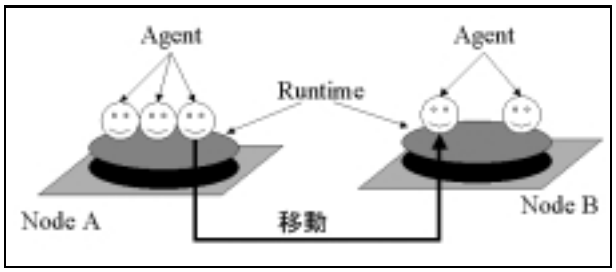

図 2.4: エージェントとランタイム

2.4.2 移動のシナリオ

図 2.5はエージェントの移動時のイメージを示す。

1 エージェントの内部状態をバイトコードに変換 (シリアライズ) する

2 ネットワークを通じてバイトコードを移動させる

3 バイトコードからインスタンスを再生する

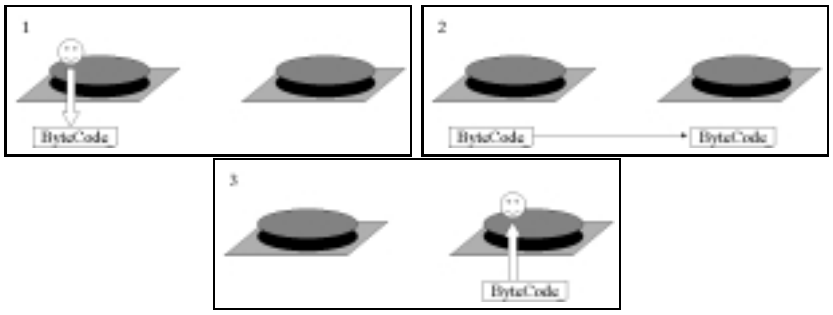

図 2.5: エージェントの移動

2.4.3 クラスロード方式

Java はクラスロード方式に柔軟性があり、ファイルのみならず、メモリ上のデータや、ネットワーク的に離れているサーバのクラスを必要に応じて読み込みインスタンスを生成することができる。Java ベースの MobileAgent は、この性質を利用してクラスをロードし、エージェントを再生する仕組みになっているが、クラスを読み込む先によって実装方法が大きく分けて 2 通りのアプローチに分かれた。

- 静的クラスロード

  エージェントの移動はインスタンスの内部状態と、エージェントのコードの移動を指し、移動した場所でインスタンスを再生する。(図 2.6)

  - メリット : 全てのコードが転送されているため、ネットワークコネクションが一回ですむ。移動してしまった後はネットワークが切断されても処理の継続が可能。コードを一括管理できる。
  - デメリット : 移動するバイトコードが肥大化し、移動のオーバーヘッドが大きくなる。

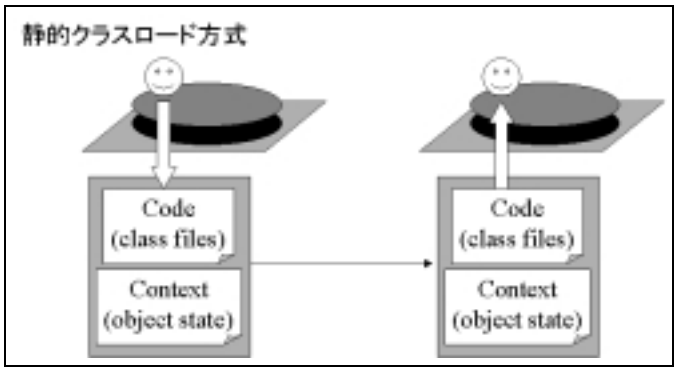

図 2.6: 静的クラスロード

- 動的クラスロード

  エージェントの移動はインスタンスの内部状態のみの移動を指し、エージェントのコードは元のサーバに置いておき、移動先に移動したエージェントがその元のクラスを読み込みインスタンスを再生する。(図 2.7)

  - メリット : 移動先で使わないクラスの転送は行わなくて済むため、転送データ量が大幅に減る。そのため、エージェント自体の移動が早い。
  - デメリット : 移動先でクラスが必要になったとき、新たにネットワークコネクションをはるため、コネクション数が多くなる。ネットワークが切断された場合に処理の継続ができない。

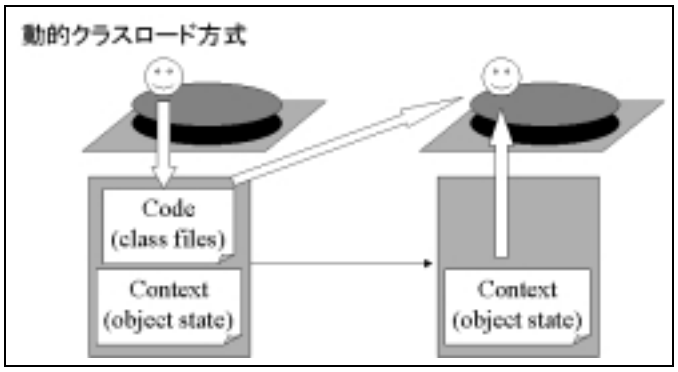

図 2.7: 動的クラスロード

2.5 既存の MobileAgent システム

様々なシステムが開発されているが、特徴的なものを挙げる。

2.5.1 Aglets

IBM 社が開発した Java ベースの移動エージェントシステム。[10]

- ロード方式:動的静的ローディング (必要に応じてユーザが選べる)
- ランタイム:AgentContext
- エージェント:Aglet, Slave, Messenger, Notifier

特徴は、Aglets 様々な移動パターンやロード方式を選べる柔軟さである。Aglet は移動エージェントとして Aglet のサブクラスを定義するほかに３つの利用パターンがある。

- **Slave** 移動行程を指定することができる一般的な移動エージェント。
- **Messenger** Message クラスを送るエージェント。
- **Notifier** 移動先で仕事を行い一定時間ごとにその結果を返すエージェント。

また、このほかにも移動したエージェントを自分のホストに呼び戻す機能等があり、移動の概念に重点を置いたシステムなので、移動中心のエージェントシステムの構築に向いている。また、他のエージェントシステムは実験段階なものが多いが、Aglets は早くから”たび can”[11] 等、商用レベルでの実用に足るシステムを提供している。

2.5.2 Kafka

富士通研究所が開発した Java ベースの移動エージェントシステム。[3]

- ロード方式:動的ローディング
- ランタイム:Agent
- エージェント:Action

他のエージェントシステムと比較し、ユニークな特徴はリフレクションである。

リフレクションはエージェントの振舞いの定義を動的に変更できる機能である。動的な負荷分散やエージェント間通信の応答速度の改善のために動的に対応できる。リフレクションをサポートする場合の問題として、

1 削除されたメッセージインタフェースに対してメッセージが送られてきた場合、

2 メッセージに添付される引数の制約が変更になった場合、

3 アクションの動作中にアクションの定義に対して変更要求がきた場合

などがある。

1,2 に関してはユーザに任されている。3 に関しては動作中のスレッドと使用しているアクションの関係を内部で管理しているので、動作中のアクションの定義は関連するスレッドが消滅するまでクラス削除は延期される仕様になっている。これによってアクションの動作とアクションクラスの置き換えが非同期に発生しても、クラス変更に伴うフォールトトレランスは保証されている。リフレクションを使いエージェントの一部を組替え、動的な保守が行えるが、動的ローディング方式を使っているためコード全てを保持しておらず、部品を置いてあるサーバとネットワークコネクションが切れた場合に保守が行えなくなる可能性がある。

2.5.3 AgentSpace

Java ベースの MobileAgent システム。静的クラスローダータイプの MobileAgent システム。ソースコードを公開し、修論・卒論生の研究用テストベッドなどを目的とし開発された。[2]

- ロード方式:静的ローディング
- ランタイム:AgentServer
- エージェント:Agent

他の MobileAgent システムとの違いに移動方法を挙げている。他のシステムには移動方法に関して問題がある。これらのシステムは Java の動的クラスローディングを基礎としている（オンデマンド方式)。この方式は、転送後に実際にインスタンスを生成する時にクラス転送を行う。このため、エージェントが移動した後にも移動元サーバとの通信が必要になり、局所性のメリットが失われてしまう。また、移動元のサーバが通信不可能になった場合に処理の続行ができなくなってしまう。しかし、AgentSpace では全てのクラスコードを転送するため、移動先で処理が完結する。シンプルな設計のため、理解や改造がしやすい。

2.6 問題点

- 自己メンテナンスの必要性

  MobileAgent は移動して処理を行うというその特性のため、MobileAgent のオーナーはエージェントの現在の稼動位置の特定が困難になり易く、保守を行うことが困難になるケースが増えてきた。その解決の一手段として、移動後の位置の特定を行う機構を新たに設けるという手段があるが、位置の特定にはエージェントのランタイムあるいはエージェント同士の通信が不可欠になり、通信相手先のランタイムが常に存在していると限らない場合には結局その機構が上手く働かない。MobileAgent の局所性という特徴を生かしたい場合には不向きであると言える。

- 動的保守の必要性

  現在、アプリケーションの保守は、一般的にアプリケーションを停止させて、変更を加えアプリケーションを再起動して行うものが多いが、MobileAgent はネットワークに強く依存しネットワークコネクションを張るものも多い。そのため、例えばプロキシの動作を行うエージェントの場合、複数のクライアントから接続要求を受け、ネットワークコネクションを常に張っている状態になる場合があり、その途中で新しいプロトコルに対応させる保守を行うとネットワーク接続が切れてしまいサービスの質の低下に繋がる。そのため、なるべくネットワーク接続を切らずに保守を行いたいという要求が生まれた。

第 3 章

動的部品交換

本章ではソフトウェアの自己管理方式の一つの動的部品交換の概念と本研究における交換の仮定を述べる。

3.1 部品交換の概念

今日、ネットワークやソフトウェア実行環境が多様化してきている。そのため、ソフトウェア開発者はソフトウェアのリリース時点で、後に起こる全ての環境変化を予想できなくなってきた。そこで、リリースした後に環境変化が起こった場合、それらの変化にアプリケーションを対応させるため技術（パッチ技術等）が開発されてきた。

Java 言語では、機能となるクラスを動的にロードすることができるため、比較的環境変化に対し柔軟に対処することができる。

3.2 仮定

本研究における動的部品交換の用語の定義を行う。

- **部品**:クラス単位を指す。パーツと呼ぶ場合もある。バージョンには全順序関係があるとする。
- **保守**:部品の交換を指す。
- **動的**:エージェント自体を再起動することなく部品を組替えることを指す。

Java ではクラスの概念が重要で、ほぼ全ての操作がクラス単位で行われている。クラス単位より小さい粒度での操作は行いにくいので部品をクラス単位とした。

部品のバージョンに全順序関係を置いた理由は、本研究ではエージェント同士の比較交換による配布を前提としているため、部品のバージョンに全順序関係がない場合は交換の必要性の判断が困難になるためである。バグの修正や、処理対象のシステムのバージョンアップへの対応といった比較的単純に部品の上下関係を決定できるケースを想定している。

また、注意点として動的な組替えを行うにあたって、いくつかの制約がつく。例えば、古いインスタンスが処理を終えていない場合にクラスを組替える場合は注意が必要である。本研究では動的交換のメカニズムのみを提供し、交換のポリシー（例えば交換時に古いインスタンスの処理が全て終わるまで待つ）の決定はユーザに任せている。

3.3 クラスローダ

3.3.1 仕組み

通常、Java 仮想マシンはローカル・ファイル・システムからプラットフォームに依存した方法でクラスをロードする。例えば UNIX システムでは、仮想マシンは環境変数 CLASSPATH によって定義したディレクトリからクラスをロードする。

しかしクラスは必ずしもファイルからロードする必要はない。これらは、ネットワークから、またアプリケーションから直接生成するといったように、他のリソースからロードすることもできるのである。defineClass メソッドはバイトの配列を Class クラスのインスタンスに変換する。こうして新しく定義したクラスは、Class クラスの newInstance メソッドを使用してインスタンスを生成することができるようになる。

クラスローダが生成するオブジェクトのメソッドやコンストラクタは、その他のクラスを参照することができる。Java 仮想マシンは参照するクラスを決定するため、元のクラスを生成したクラスローダの loadClass メソッドを呼び出す。クラスの存在確認や、存在する場合のスーパークラスの情報のみが必要である場合、resolve フラグは false に設定される。しかし、クラスのインスタンスを実際に生成する場合や、そのメソッドのいずれかを呼び出す場合、クラスの参照は解決しておく必要がある。その際は、resolve フラグは true に設定されるので、resolveClass メソッドを呼び出すようにする。

MobileAgent はこの機能を利用し作られている。例えば、AgentSpace では ClassLoader クラスを継承し、AgentClassLoader クラスを定義し、エージェントを移動してきたバイ

トコードからエージェントを再生している。

3.3.2 クラスローダの使用例

実際にクラスローダを拡張し、動的にクラスロード行う例を解説する。Hello.class(図 3.3) を他のクラスから動的に読みこむ例を解説する。

まず、ClassLoader クラスを継承したクラス、CLoader クラス (図 3.1) を実装する。

```
public class CLoader extends ClassLoader{
  public CLoader(){
  }

  public Class loadClass(String classname,boolean boo){
    Class cl = null;
    try{
      cl = findSystemClass(classname); // ←この部分を変える
      System.out.println("Class : "+classname+" Loading");
    }catch(ClassNotFoundException e){}
    return cl;
  }
}
```

図 3.1: CLoader.java

loadClass の返値に動的に作ったクラスを指定すれば良い。今回は findSystemClass() を使い、CLASSPATH から読みこむ実装にしてあるが、もし CLASSPATH にそのファイルが無かった場合にはまた別のリソース（ネットワーク上や、メモリ中のバイトコード）からクラスを生成するといった処理を行うことができる。

次に実際に前述の CLoader クラス (図 3.1) を使い Hello.class を呼び出すクラスを実装する (図 3.2)。（クラスローダが findSystemClass() を呼び出しているだけなので new で Hello クラスを生成した場合と大差がない)。

```
public class CLtest {
  public static void main(String[] argv){
```

```
    ClassLoader loader = new CLoader();
    try{
      loader.loadClass("Hello").newInstance();  // 呼び出すクラスを指定し
    }                                           // インスタンスを生成する
    catch(ClassNotFoundException e){}
    catch(InstantiationException e){}
    catch(IllegalAccessException e){}
  }
}
```

図 3.2: CLtest.java

呼び出される Hello.class(図 3.3)。

```
public class Hello {
  public Hello(){
    System.out.println("Hello World");
  }
  public static void main(String[] argv){
    new Hello();
  }
}
```

図 3.3: Hello.java

第 4 章

部品配布方式

本章では、まず、従来のアプリケーションに行われてきた部品交換方法の説明を行い、その後、MobileAgent の特性を生かした比較交換方式を提唱する。

4.1 従来の配布方式

本節では、従来のアプリケーションに適用されている部品配布方式を説明する。

4.1.1 push 方式

部品メーカーは部品のアップデートを行う場合、各ランタイムに部品を送る。(図 4.1)

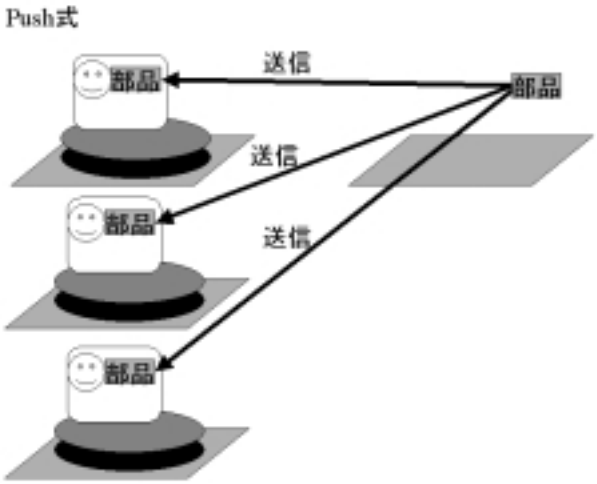

図 4.1: 部品配布方式-Push 方式

- メリット:部品メーカーは部品のアップデートをしたいときにだけネットワークに繋げれば良い。
- デメリット:頻繁にアップデートを繰り返す場合、通信量が多くなる。

4.1.2 pull 方式

エージェントが常に倉庫を監視し、アップデートがあった場合にパーツを取得する。(図 4.2)

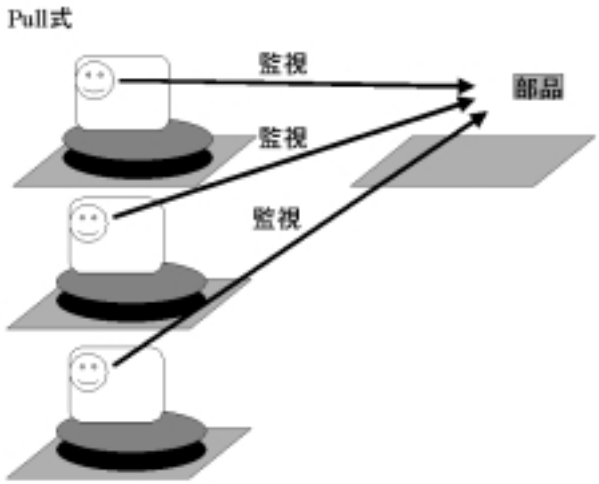

図 4.2: 部品配布方式-Pull 方式

- メリット:部品メーカーはパブリックな場に部品を置いておくだけで良い。
- デメリット:アップデートの監視のため、部品の置いてあるサーバにアクセスが集中する。よって、部品を置くサーバは常にネットワーク接続が可能な状態にしておかなければならない。

4.1.3 従来の配布方式の問題点

- 固定情報を持ち歩くリスク

  本研究では特に動的組換えの部品の配布方式に焦点をあてている。

  従来のアプリケーションでは稼動場所が限られており、その場所にあるコードをメンテナンスするという方法であった。交換部品はその場に送られ、その場でオーナー

がメンテナンスを行うという「場」を主体とした配布交換であったが、本研究では「アプリケーション」を媒介にした配布方式であることが特徴的である。部品はアプリケーションが運び、アプリケーション自体が保守を行う。部品を得る行動もタイミングもアプリケーションが決定する。

「場」を主体とした配布の場合、「場」は固定的である必要がある。しかし、ネットワーク環境の複雑化により「場」が固定的ではなくなりつつあり、固定的な情報を埋め込んだアプリケーションは処理の継続が困難になることが多い。

本研究で提案する方式では「アプリケーション」を媒介とした配布方式なので、固定情報は極力少なくすることができ、より、柔軟な対応ができるようになりオーナーやメーカーの保守作業を軽減させることが可能となる。

4.2 比較交換方式

これらの問題を解決するための方式を提案する。エージェント同士がお互いの持っているパーツを比較し、お互いに補完しあう方式を提案する。(図 4.3)

例えば Web サーバが動いているサーバを巡回し、アクセスログの解析を行う log analyzer エージェントを作るとする。Web サーバには IIS, Apache, Enterprise 等があり、それぞれの解析手法は異なり、異なったルーチンが必要になる。

例えば自分は Apache 用の解析部品しかもっていない場合に、IIS のサーバへ移動した場合、解析はできない。ところが、その場に IIS 用の解析部品を持つエージェントが居た場合、その部品を取得すれば今後自分でも IIS の解析が行えるようになる。また、相手が Apache 用の解析部品を持っていなければ相互に補完し合うことによって、システム全体に部品が浸潤してゆく。

- メリット:エージェントの移動とともに部品が運ばれるので、部品配布のために新たなコネクションを張る必要がない。
- デメリット:従来方式に比べて部品配布時間がかかるケースがある。確実に全エージェントに部品が渡るという保証がない。媒介となるエージェントが少ない場合には交換が行われない。

4.2.1 部品追加のシナリオ

1 同サーバ内の相手のパーツリストとパーツの情報を要求

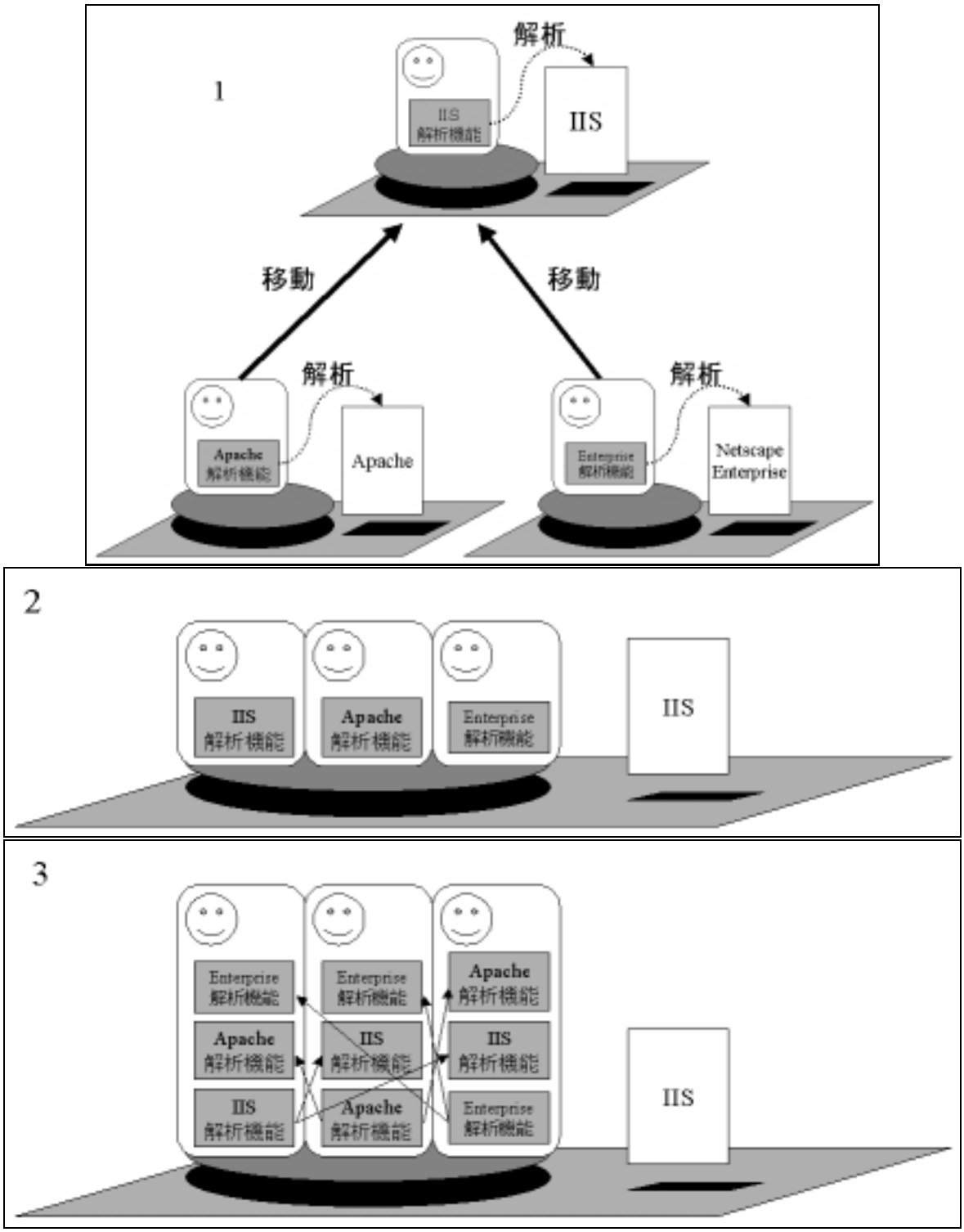
1
解析
IIS
解析機能
IIS
移動
移動
解析
解析
Apache
解析機能
Apache
Enterprise
解析機能
Netscape
Enterprise
2
IIS
解析機能
Apache
解析機能
Enterprise
解析機能
IIS
3
Enterprise
解析機能
Enterprise
解析機能
Apache
解析機能
Apache
解析機能
IIS
解析機能
IIS
解析機能
IIS
解析機能
Apache
解析機能
Enterprise
解析機能
IIS

図 4.3: 相互部品補完

2 自分のパーツと比較し、必要なものがあればパーツを要求

3 パーツを取得

4-1 コードの中身を入れかえる（オーバーライトモード）

4-2 別枠にコードをストックする（ストックモード）

5-1 移動時に古いインスタンスを破壊する。

5-2 古いインスタンスが全て消滅したときに古いコードを破棄する。

4.2.2 比較交換方式の特徴

従来の方式では交換部品はアプリケーションの活動場所に送られ、その場でオーナーがメンテナンスを行うという「場」を主体とした配布交換であったが、比較交換方式では「アプリケーション」を媒介にした配布方式であることが特徴的である。部品はアプリケーションが運び、アプリケーション自体が保守を行う。部品を得る行動もタイミングもアプリケーションが決定する。

「場」を主体とした配布の場合、「場」は固定的である必要がある。しかし、ネットワーク環境の複雑化により「場」が固定的ではなくなりつつあり、固定的な情報を埋め込んだアプリケーションは処理の継続が困難になるケースが増えてきた。比較交換方式は「アプリケーション」を媒介とした配布方式なので、固定情報は極力少なくすることができ、より、柔軟な対応ができるようになり、オーナやメーカーの保守作業を軽減させることが可能となる。

第 5 章

実装

本章では部品交換部分の API の実装し、その API を利用する MobileAgent の実装方法を解説する。

5.1 実装環境

- OS:Windows98/Soralis2.7
- Java:JDK1.7(Java1.1)
- MobileAgent システム:AgentSpace

5.1.1 AgentSpace の仕組み

AgentSpace を用いて実装するためにまず、AgentSpace 自体の構造と簡単なエージェントの作成方法を解説する。

静的クラスロード方式の MobileAgent システムである。図 5.1 は AgentSpace の階層図である。

```
(agentspace)-|-(system)-|-agentspace-|-*.class   AgentSpace システムのクラス
                        |             |-*.java    AgentSpace システムのソース
                        |-*.agent       エージェントプログラム
                        |-*.java        各エージェントのソースプログラム
                        |-*.class       各エージェントの class ファイル
```

図 5.1: AgentSpace のディレクトリ階層図

AgentSpace の API を解説する。表 5.1 は Agent.class に定義されているメソッドである。エージェントにこれらのメソッドを実装することによって様々なアクションに対応させることができる。

簡単なエージェントのコードを示す。このコードをコンパイルし、makeagent でエージェント化すると図 5.3のエージェントが作成される。

```
import agentspace.*;
import java.awt.*;
import java.net.*;

public class HelloWorld extends Agent {
  public HelloWorld() {}
  public void create() {
    add(new Label("Hello World"));
    setSize(100, 60);
    show();
  }
}
```

図 5.2: HelloWorld.java

図 5.3: Hello.agent

表 5.1: AgentSpace のコールバックメソッド

メソッド	機能名	概要
void init()	初期化	クラスファイルからエージェントのファイルを合成するときに一度だけ呼び出される
void create()	生成	エージェントが生成される際に一度だけ呼び出される。
void destroy()	終了	エージェントが、消滅する直前に呼び出される。
void dispatch(URL url)	移動	エージェントが、他のコンピュータに移動するときに、今いるコンピュータを離れる直前に呼び出される。引数 url には移動先の URL アドレスが入る
void arrive()	到着	ージェントが、移動先のコンピュータに到着した直後に呼び出される。
void suspend()	永続化	エージェントが永続化される直前に呼び出される。
void resume()	活性化	永続化されたエージェントの再び動作可能になる際に呼び出される。
void duplicate()	複製	エージェントの複製が生成された際に呼び出される。
void parent(AgentIdentifier aid)	複製	エージェントの複製が生成したエージェント側のみ呼び出される。引数には複製したエージェントの識別子が入る。
void child(AgentIdentifier aid)	複製	複製されたエージェント側のみ呼び出される。引数には複製したエージェントの識別子が入る。

5.2 比較交換方式の実装

本研究では AgentSpace にエージェント同士が部品の比較交換を行うための機構を実装した。AgentExchange.class に部品交換のための API 群を実装した。また、それらの機能を簡単使うために Agent クラスのコールバックメソッド及びフィールドを拡張した。

5.2.1 部品交換のための追加 API

AgentExchange.class を新たに追加し、表 5.2に示す関数を作った。

表 5.2: 追加 method

メソッド名	内容
int getParts(String classname,AgentContext ac)	必要な部品を取得する
int removeParts(String classname,AgentContext ac)	保持コードから部品を消去する
Object loadParts(String classname,AgentContext ac)	部品からインスタンスを生成する
int readPartsVersion(String classname,AgentContext ac)	部品のバージョンを読みこむ
String readPartsCategory(String classname,AgentContext ac)	部品のカテゴリを読みこむ
String readPartsField(String classname,AgentContext ac)	ユーザ定義の部品情報を読みこむ

5.2.2 Agent クラスの拡張

新たに得た部品を実際に利用するために Agent クラスに新たなコールバックメソッドを付け加えた。また、比較交換を行うために部品となるクラスに情報を埋め込む必要があるためいくつかのフィールドを追加した。(表 5.3)

表 5.3: Agent クラスの拡張

変数名	内容
void execParts(String pats)	部品を得たときに実行されるコールバックメソッド。 得た部品をどう扱うかを記述できる
int partsVersion	部品のバージョン
String partsCategory	部品の種類

5.3 実装例題

ここでは実際に、システムを使って部品交換を行う例題を組む方法を解説する。
例題は秘書エージェント。

5.3.1 例題の概要

時計や電卓といった小物ツールを使える秘書エージェント。

他の秘書エージェントに出会った時に相手のツールと比較してお互いに最新バージョンの機能を補完し合う。

(見易くする例題のため GUI を多用し、実用には向かない)

5.3.2 例題のイメージ

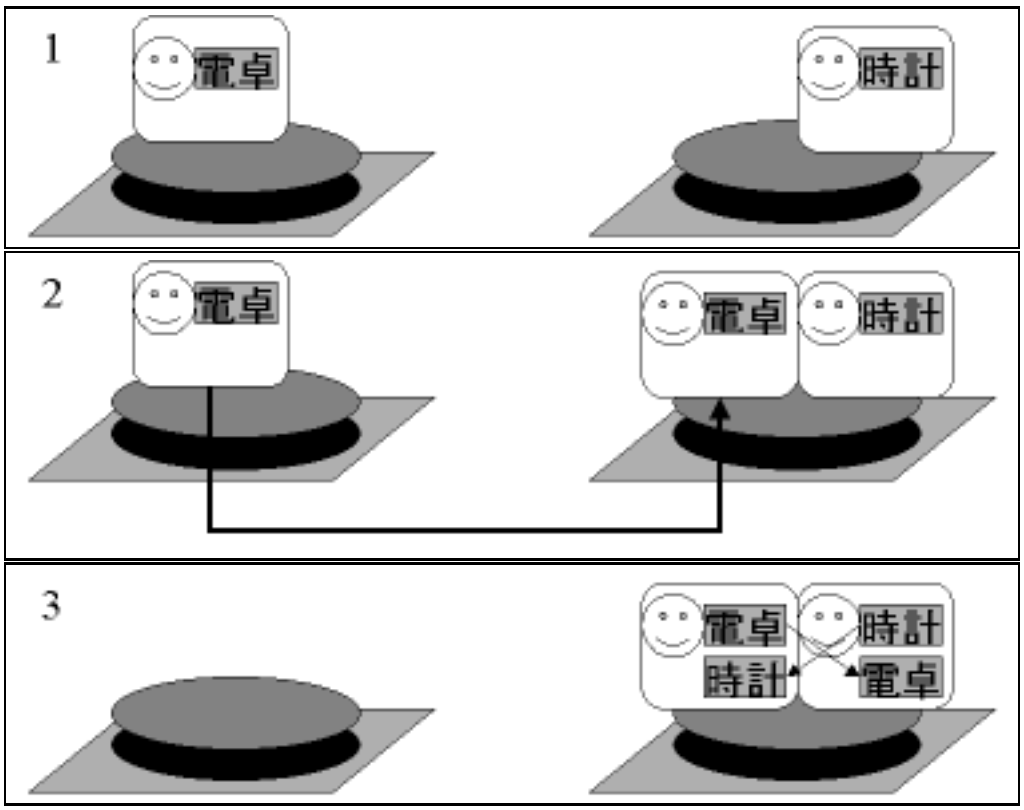

図 5.4: 秘書エージェント

5.3.3 エージェント本体のコード

```
public class Hello extends Agent{
  public boolean action(Event evt,Object arg){
    String classname = null;
    classname = (String) bnametoclass.get((String)arg);
```

```
    Object obj = AgentExchange.loadParts(classname,ac);
    int ver = AgentExchange.readVersion(classname,ac);
    instancetable.put(obj,new Integer(ver));

    mf.additem(obj);
    return true;
  }
  public void execParts(String cl){        // parts を得たときに呼び出される
    parts.addElement(cl);

    Class cla = AgentExchange.readClass(cl,ac);
    String classtype = cla.getSuperclass().getName();

    // parts 情報読み出し
    String labe  = AgentExchange.readPartsField(cl,ac);
    String category = AgentExchange.readPartsCategory(cl,ac);
    int ver = AgentExchange.readPartsVersion(cl,ac);

    // secretary 特有 (applet と secretary parts のみボタン配置)
    if (classtype.equals("java.applet.Applet")){
      addbutton(labe,ver,cl);
      return;
    }
    if (category == null) return;
    if (category.equals("secretary")){
      addbutton(labe,ver,cl);
    }
  }
 :
}
```

5.3.4 パーツリスト

- 時計 version1

図 5.5: 秘書エージェント部品-時計 Version1

```
public class Clock2 extends Applet implements Runnable {
  public static String buttonlabel   = "時計";
  public static String partsCategory      = "secretary";
  public static int partsVersion      = 1;
  public void init() {
:
  }
  public void start() {
:
  }
}
```

- 時計 version2

図 5.6: 秘書エージェント部品-時計 Version2

```
public class Clock2 extends Applet implements Runnable {
  public static String buttonlabel   = "時計";
  public static String partsCategory      = "secretary";
```

```
  public static int partsVersion      = 2;
  public void init() {
                :
  }
  public void start() {
                :
  }
}
```

- 電卓 version1

図 5.7: 秘書エージェント部品-電卓 Version1

```
public class Calc extends Panel {
  public static String buttonlabel   = "計算機";
  public static String partsCategory      = "secretary";
  public static int partsVersion      = 1;
:
  public Calc(){
:
  }
}
```

5.4 例題実行例

初期状態では server1 に 時計 version1 計算機 version1 を持つエージェントが存在し、server2 に時計 version2 が存在している。

図 5.8: 組替え前 server 1

図 5.9: 組替え前 server 2

ここで server2 から server1 へ時計 Version2 を持つエージェントが移動してくると、組替えが起こり、以下の状態になる。

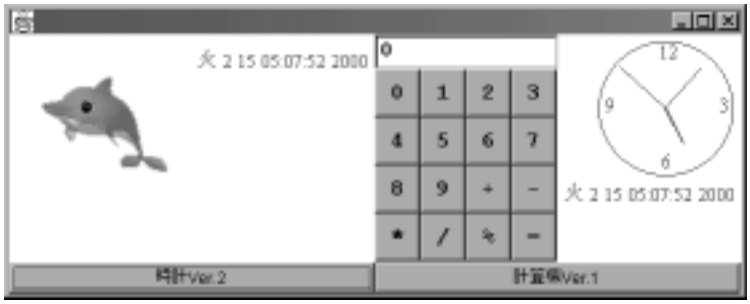

図 5.10: 組替え後

第 6 章

実験

本章では前章で実装したシステム上に例題を実装し、従来の方式との特性を比較検証する。

6.1 実験例題

エージェントのランタイムと同時に Web サーバが立ちあがっているホストを巡回し、ログファイルの解析を行い、複数のサイトの情報からなるアクセスランキングを作るエージェントの例題を実装した (図 6.1)。

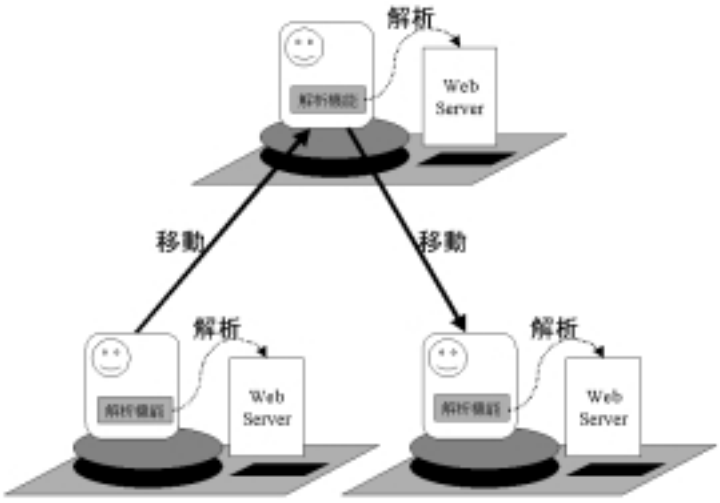

図 6.1: log analyzer

6.2 基礎実験

シミュレートを行うにあたって、目安となる数値を設定するために、実装したサンプルエージェントを用いて基礎的なデータを測定した。

表 6.1: 基礎データ

エージェントサイズ	4109 bytes
部品サイズ	3809 bytes
エージェント移動時間 (専用線間)	245 ms
エージェント移動時間 (PPP 接続間)	4124 ms
ログ解析時間	6980 ms

6.3 実験フィールドの仮定

実験フィールドとして以下の環境を仮定する。

- 100 ノードにそれぞれ Web サーバと MobileAgent のランタイムが稼動している。
- Web サーバは Apache,IIS,Enterprise,NCSA, その他, の種類があり、100 ノード中以下の割合で存在している (1997 年 8 月調べ [15])。

表 6.2: サーバ比率

Apache	41 node	Enterprise	18 node
IIS	18 node	NCSA	10 node
Others	13 node		

- 上記のノード上に、あらかじめ Apache 用の部品のみを持った log analyser エージェントが数体いる。

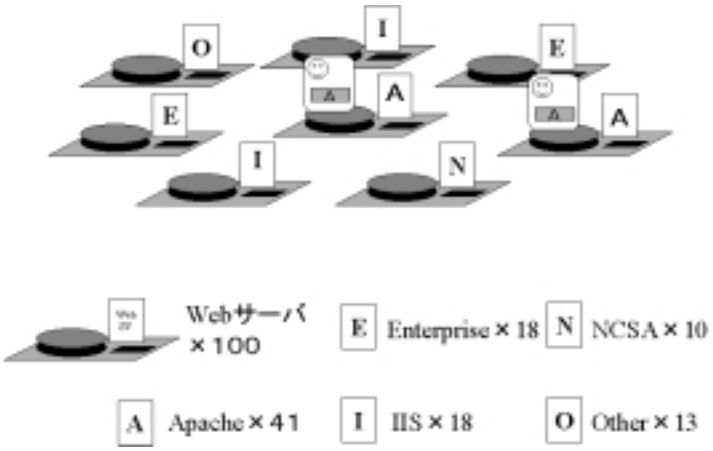

図 6.2: 初期設定

- エージェントは以下のポリシーで稼動する。
  - 現在居る場所の Web サーバが解析可能な場合：システム全体の仕事量＋１、処理終了後、次の未処理サーバへ移動。
  - 現在居る場所の Web サーバが解析不可能な場合：次の未処理サーバへ移動
- メーカーは上記上のノードには存在せず、サーバまでは電話回線で繋がっている携帯端末とする。

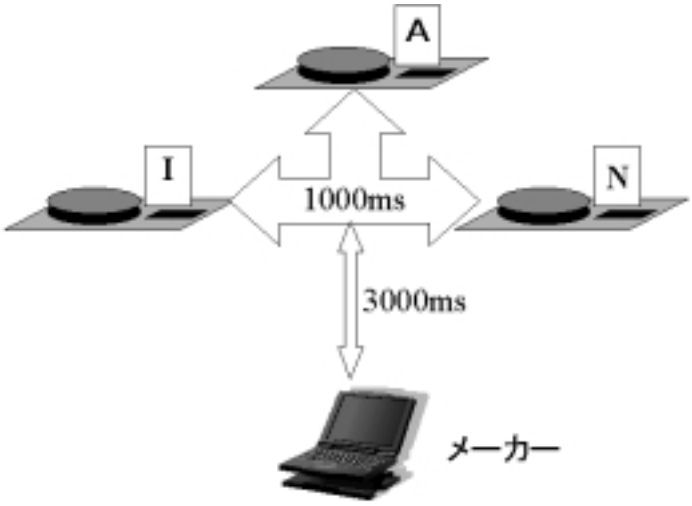

図 6.3: 初期設定

- ノード間/メーカー間の移動時間, 処理時間は以下表に示す通りである。

表 6.3: 移動/処理時間

ノード間の移動時間	1000 ms
メーカーノード間の移動時間	4000 ms
処理時間	6000 ms

6.4 実験シナリオ

上記のフィールド上のエージェントに新しい部品を配布する実験を行う。

- 比較交換方式：上記フィールドに IIS,Enterprise,NCSA の部品をもったエージェントを各一体ずつ参加させ、配布状況を調べる。
- push 方式：上記フィールドのエージェント一体一体に IIS,Enterprise,NCSA の部品を持たせる保守を順次行う。携帯端末からノード群までの回線は電話回線のため、配布には 4000ms かかる。
- pull 方式：本来は携帯端末を常に見張り、アップデートがあったときに部品を送付してもらうというアプローチであるが、アップデートの監視時間を細かく取ると、push 方式とほぼ同じ結果になるため、本実験では push 方式のみ実験を行った。

6.5 実験結果

6.5.1 実験環境

100 スペースのエージェントスペースをシミュレートするシミュレータを作り、基礎実験 (表 6.1) の結果をもとに設定したパラメータ (表 6.3) でシミュレーションを行った。

6.5.2 実験結果-部品配布スピード

部品配布率

以下の図は部品配布開始後 30 秒間で全エージェントの何%に各部品が配布されたかを示す。

- エージェント数/ランタイム数 = 0.04 の場合。

  100 スペースに対し 4 体のエージェントが活動したが、一度も出会わずに活動したため、比較交換方式での部品の配布は行われなかった。

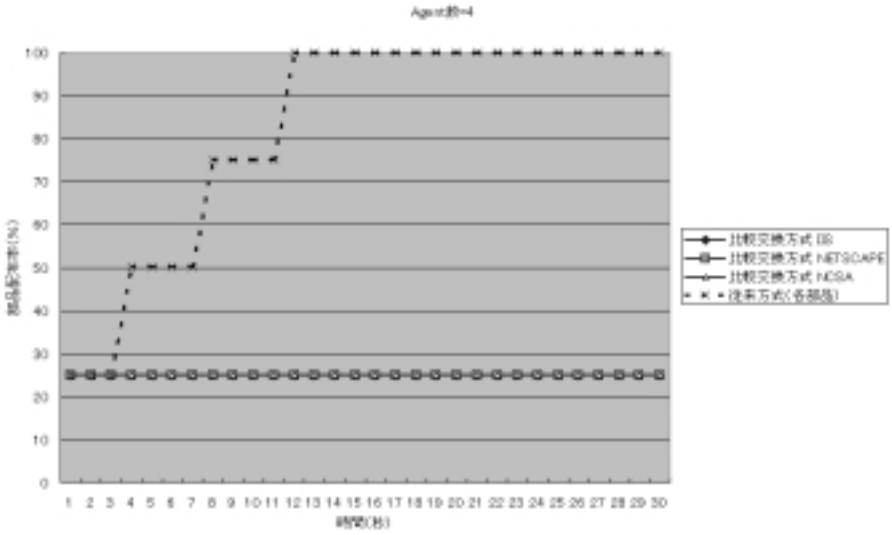

図 6.4: 部品配布率-エージェント数：4

- エージェント数/ランタイム数 = 0.1 の場合。

  100 スペースに対し 10 体のエージェントが活動。まだ従来方式のほうが確実に部品を配布している。

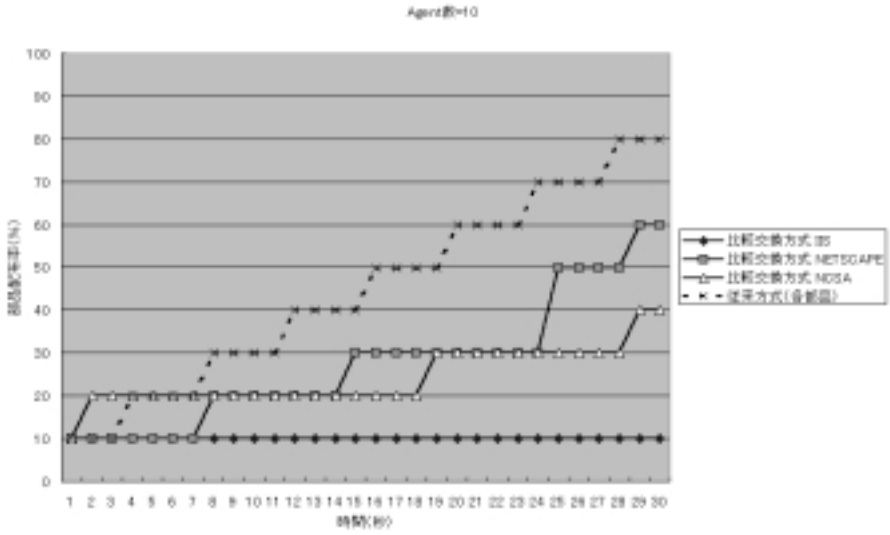

図 6.5: 部品配布率-エージェント数：10

- エージェント数/ランタイム数 = 0.25 の場合。

  100 スペースに対し 25 体のエージェントが活動。従来方式での配布が全エージェントに行き渡らなくなっている。IIS 用の部品を持つエージェントが早期の段階で他のエージェントに出会ったため、急激に配布率を伸ばし、エージェント全体の約 80%のエージェントが IIS 用の部品を持った。従来方式とほぼ同程度の結果が期待できる。

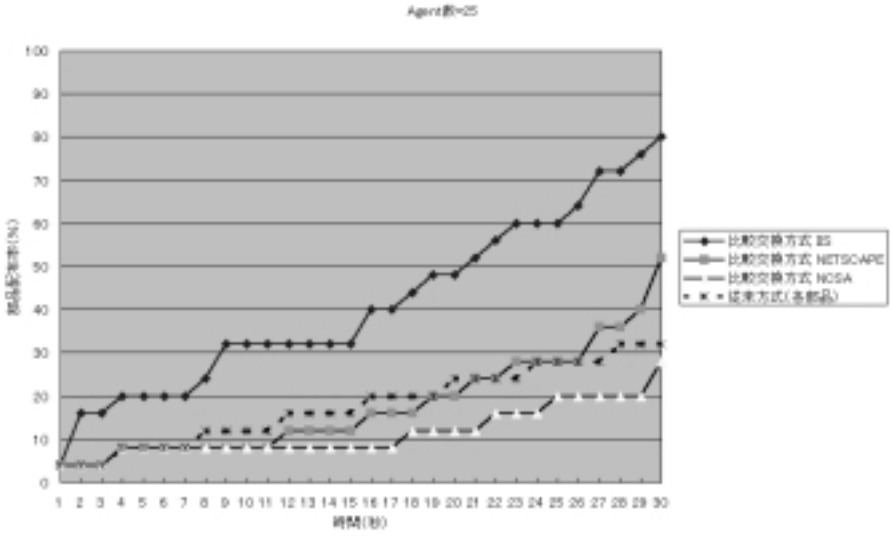

図 6.6: 部品配布率-エージェント数：25

- エージェント数/ランタイム数 = 0.5 の場合。

  全スペースの半分にエージェントが存在している状態。従来方式では全体のわずか10%程度にしか部品を配布できない。

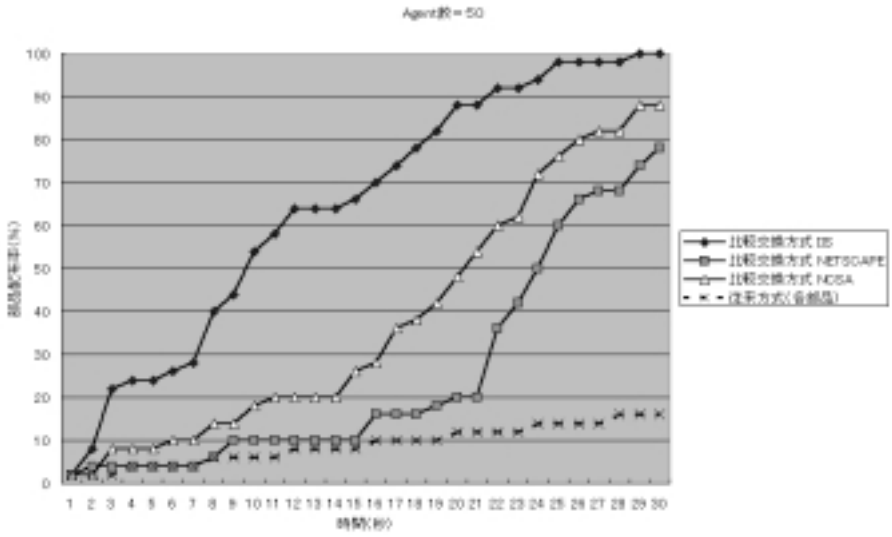

図 6.7: 部品配布率-エージェント数：50

- エージェント数/ランタイム数 = 1 の場合。スペースの数と同数のエージェントが存在している状態。ほぼ全てのエージェントに部品が配布された。

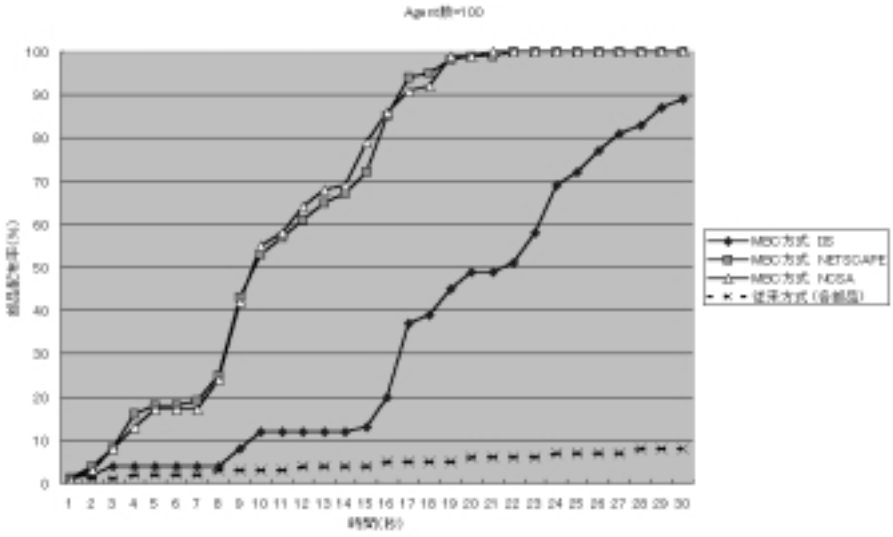

図 6.8: 部品配布率-エージェント数：100

総仕事量

システム全体で部品配布開始後３０秒間に全エージェントが行った総仕事量を示す。
従来方式

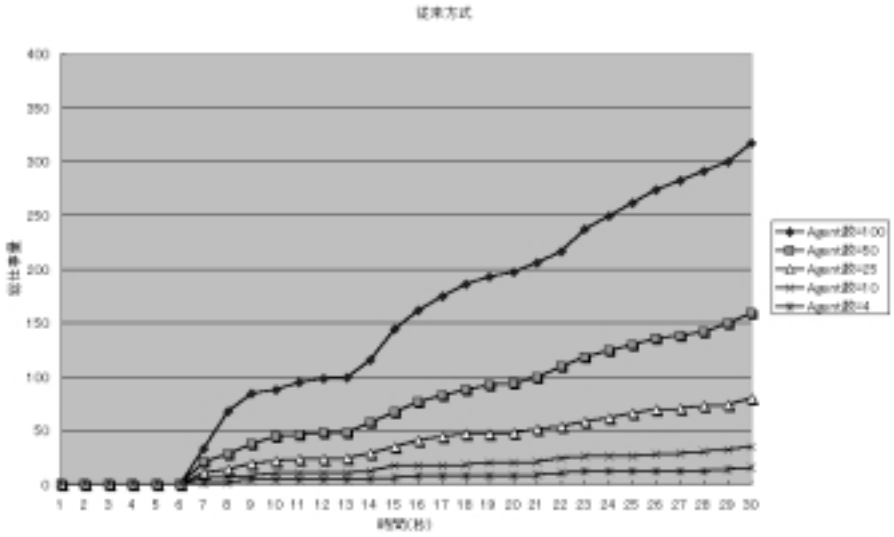

図 6.9: 総仕事量-従来方式

比較交換方式

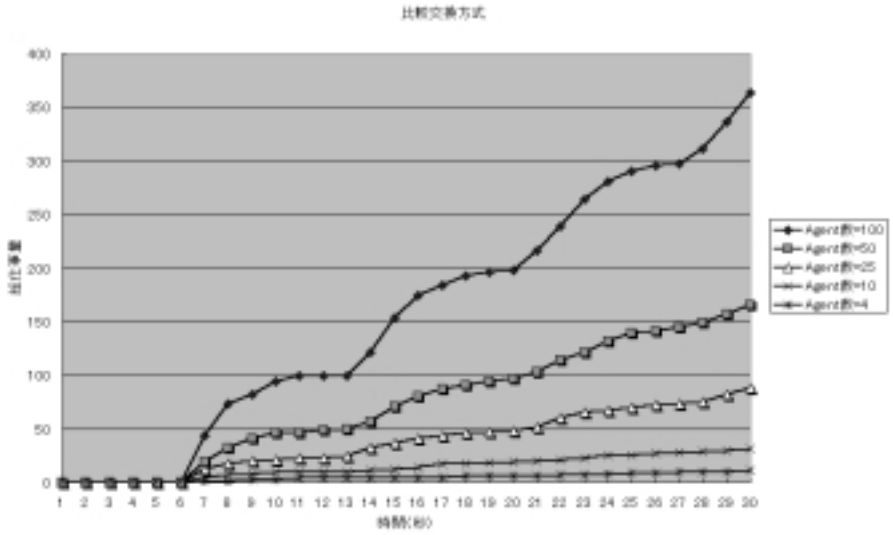

図 6.10: 総仕事量-比較交換方式

仕事率

従来方式と、比較交換方式の１エージェント当たりの行った仕事量を示す。

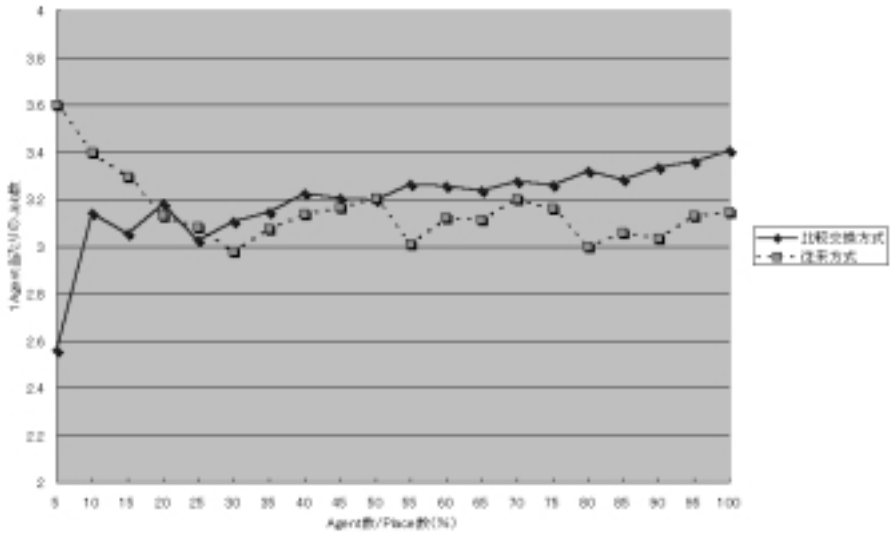

図 6.11: １エージェント当たりの仕事量

エージェントの数が 20 体を超えたあたりから、比較交換方式のほうが仕事量が勝っている。

6.5.3 結論

本例題での実験では、エージェントの数が多くなればなるほど、比較交換方式のほうが迅速に配布が行われた。このため、場に対して比較的エージェントの数が少ないシステムでは従来の方式を利用し、エージェントの数が比較的多くなる場合には比較交換方式のほうが有利である。

第 7 章

おわりに

本章では本研究をまとめ、最後に、本研究の展望を述べる。

7.1 まとめ

現在、計算機資源や、情報が分散して存在する傾向にあり MobileAgent 技術が生まれた。しかし、その移動性と独立性という性質のため、オーナーの手を離れて活動する時間が長い。MobileAgent は活動先での環境の変化に、柔軟に対応できないといった問題が生まれた。

例えば、決められた Web サーバを巡回し、HTML ファイルを読み取り、欲しい情報のみを抽出し mail で転送する機能を持つ情報収集エージェントの場合では、HTML の構成等を変えられると処理の続行ができなくなる。従来のアプリケーションならばオーナが抽出ポリシーを変えれば良かったが、MobileAgent の場合はエージェントの活動中の場所がオーナが直接手を加えられる場所であるとは限らないため、保守を行うことが困難になる。

そこで、本研究の目的は動的部品交換という手段を用いそれらの問題を解決することにした。

本研究を行うにあたって、MobileAgent システムの基本的な概念、例えば、移動性や局所性といった特徴があることを述べ、後に既存の MobileAgent システムの特徴、例えばクラスロード方式の違い等について解説した。次に Java アプリケーションの一部を自動的組み変えて保守を行うための交換方式の概念と本研究における位置付けを行い、動的交換に使われる部品の配布方式について述べた。従来のアプリケーションに行われてきた push 方式や pull 方式といった方法の説明と固定情報の持ち歩く必要性といった特性の解

説を行った。

その後、MobileAgent の移動性や局所性という特性を生かした方式（比較交換方式）を提案した。提案したエージェントの自動保守機構を実際に実装し、そのメソッドを利用する MobileAgent の実装方法を解説した。実装したシステム上に Web のログアナライザ等の数種の例題を実装及び実験し、従来の方式との特性を比較検討した。

7.2 結論

本研究で提案した比較交換方式の実験結果を述べる。本方式は、場に対するエージェントの数の割合が高い程、効率よく部品を配布することができ、従来の Push 方式や Pull 方式よりもネットワーク接続を張る回数を少なく組替えを行うことができる。さらに、従来の Push 方式や Pull 方式が有効に働く場面でも本研究の方式を併用することができ、より効率の良い組替えを提供することができる。

また、従来の方式に比べ、「場」の情報といった、固定的な情報を埋め込む必要がないので、「場」が流動的に変化する場合でも「場」と独立して組替えを行うことができる。

7.3 今後の課題

- **セキュリティ**:現在、部品交換のポリシーはバージョンの高いものなら盲目的に信じて取りこむというポリシーにしているが、セキュリティの面を全く考慮していない。部品の認証等の必要性がある。

- **拡散スケジュール**:現在、部品の拡散は各エージェントの移動ポリシーに委ねられているため、エージェントの数が少ない場合はなかなか拡散が行われないケースも多い。保持部品の情報交換を積極的に行うための機構の必要性がある。

- **関係のある部品のパッケージ化**:現在、部品のクラスに継承関係がある場合や、組替えを行った後、古いインスタンスをどうするかといった処理はユーザに委ねている。これらの処理を安全に、かつ効率的に行えるメソッドが必要である。

7.4 今後の展望

- **部品の柔軟性**:現在は一意に決まっている部品優先度をもっと柔軟性を持たせると、保守のみならず、エージェントが独自に新しい機能を付け加えていくといった可能

性が広がる。

- **協調性、適応性の付加**:現段階では従来方式に、移動性、局所性、独立性を考慮した交換方式であるが、協調性を取り入れることにより、より効率的に部品の配布が可能となると予測する。例えば、現状ではエージェント同士が交換する情報は、相手が持っている部品のリストのみであるが、「○○という部品をだれそれが持っていた」という間接的な情報も交換すれば、現在、処理することができないことでも、エージェントが独自に判断して問題解決のために移動し、部品を得るといったことが可能になる。

- **自律性の付加**：現在はオーナやメーカーに提供された部品をエージェントが組み込み、部品が拡散していくだけであるが、そのうち、エージェント自体が部品や環境から学習し、新しい部品を作っていけるようなインテリジェンスを持つようになると、自律的に自己進化してゆくエージェント群が現れるようになるかもしれない。例えば、サーチエンジンエージェントの場合は、様々な Web サーバを解析するうちに、各サーバを解析する部品の差分を取ることによって、初めて扱うサーバにも同じ方法が使えないかどうかを指向錯誤を行って、新たに部品を構築することが可能になるかもしれない。

謝辞

本研究を行うにあたり、有益な御指導、助言および援助を頂きました権藤克彦助教授に深く感謝の意を表し、心より御礼申し上げます。また、本研究について様々な議論をして頂いた片山卓也先生をはじめとするソフトウェア基礎講座のメンバーの皆様方に深く感謝致します。

参考文献

[1] M.T. Tu,F.Griffel,M.Merz,and W.Lamersdorf: A Plug-in Architecture Providing Dynamic Negotiation Capabilities for Mobile Agents, LectureNotes in Computer-Science 14,MA'98,MobileAgents, 1998.

[2] 佐藤一郎: Hierarchically Structured Mobile Agents and their Migration, to appear in Workshop on Mobile Object Systems, 1999.

[3] T.Nishigaya.Design of Multi-Agent Programming libraries for Java.Http://blacky.fujitsu.co.jp/hypertext/free/kafka/paper/,1997.

[4] 飯島正, 山本喜一, 土居範久: 遠隔ソフトウェアメインテナインスのためのオブジェクト実行時バージョンアップ, 日本ソフトウェア科学会１５回大会, 1998.

[5] 飯島 正, 山本喜一, 土居範久:移動エージェントのための共生・寄生モデル,IPS55, 1998.

[6] 粂野　文洋, 佐藤 仁孝, 加藤 哲男, 本位田 真一: エージェント指向言語 Flage によるネットワーク移動型エージェントとその応用, 第 17 回 IPA 技術発表会, http://www.ipa.go.jp/NEWSOFT/public/Flage/index.html, 1998.

[7] 天野憲樹, 渡部卓雄:動的適応可能なソフトウエア・ モデルのための言語的アプローチ, SPA'98,1998.

[8] Danny,B.L.,Mitsuru,O.and Kazuya,K.:Aglets Programming Mobile Agents in Java. LNCS 1274,pages 2553-266,Springer-Verlag. 1997.

[9] White,J.E.:Telescript technology,the foundation for the electronic market-place.White Paper.General Magic,Inc.1994.

[10] http://www.trl.ibm.co.jp/aglets/

[11] http://www.tabican.ne.jp/

[12] http://www.voc.or.jp/i-kyo/s2-331.html

[13] http://www.jisa.or.jp/activity/report/1995/agent95.html

[14] http://www.melco.co.jp/rd_home/java/monja/paper.html

[15] http://www.hitachi-ns.co.jp/pub/w3survey/9708/