修 士 論 文

信頼性の高い移動エージェントシステムの構成方法

指導教官　渡部卓雄 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

新堀健治

1999 年 2 月 15 日

要 旨

本研究の目的は、信頼性の高い移動エージェントシステムを構成するための方法を考案することである。移動エージェントとは、ネットワークで結合されたホスト間を内部状態を保持したまま移動できるプログラムのことである。移動エージェントには、自律性、協調性などの性質を備えているため、柔軟なアプリケーションを作ることができる。しかし、移動エージェントの技術を現実的なアプリケーションに利用するにあたってセキュリティと信頼性の確保をする必要がある。本研究では信頼性の問題について取り扱う。信頼性の問題とは、移動エージェントが障害によってその目的を達成することができなくなることである。ネットワークを移動し、処理を行う移動エージェントは、通信障害、ホストダウンなどの障害に対処しなければならない。しかし、既存の移動エージェントシステムで作成された移動エージェントは、計算機環境の障害に関して十分考慮された設計になってない。そのため、計算機環境に障害が発生した場合、移動エージェントの目的を完結することができない。本研究では、計算機環境で発生する様々な障害を分析し、それぞれの障害レベルに応じた処理をイベント処理として実現する機構を提案する。

本研究で提案する信頼性の高い移動エージェントシステムを作成するための基本フレームワークを Coop と名付ける。Coop のアーキテクチャは、a) 障害対処機構と b) 管理モジュールの 2 つから構成される。移動エージェント自身が障害対処機構を持って移動することで、障害に対処することができ、障害対処機構で対処できない処理については管理モジュールが行う。また、Coop のアーキテクチャは移動エージェントのプラットフォームに依存しない設計で構成されている。基本障害処理とは、Coop が保証している障害処理のことで、基本的な障害処理は Coop が行う。また、基本障害処理をカスタマイズすることで、移動エージェントの目的に合った障害処理をすることができる。移動エージェントの実験プラットフォームは、Java ベースの移動エージェントシステムである AgentSpace を用いて実際に作成した。

目 次

図 目 次

表 目 次

第 1章

序論

本論文は、信頼性の高い移動エージェントの構成方法を提案する。本章では本研究の背景と目的を述べ、最後に本論文の構成について述べる。

1.1 本研究の背景

コンピュータネットワークを利用したアプリケーションは、一般にRPC(Remote Procedure Call)を用いたクライアント・サーバ型のシステムが利用されてきた。RPCとは、一方のコンピュータから他方のコンピュータのプロシージャを呼び出すもので、ネットワーク上に流れる通信データはプロシージャに渡す引数か、プロシージャの実行結果である。最初にサーバーに対してプロシージャ実行を依頼して、次にサーバーから実行結果が送られてくるまでの間、クライアントとサーバーは常に接続されていなければならないのも、PRCの特徴である。LANのような比較的ネットワークの規模が小さく、通信環境が常に安定しているようなネットワークにおいては、RPCは非常に強力なプログラミング方法と言える。しかし、無線LANやPHS、携帯電話を利用した通信環境が不安定な場合、RPCを用いたネットワークアプリケーションは有効な手法とは言い難い。また、RPCを利用してアプリケーションを設計する場合は、全体を把握して緊密に各プログラムの役割を明確にしなければならない。何故ならRPCを用いてアプリケーションを設計する場合、リモートアクセスするプロシージャがシステムにどう影響を及ぼすのか、またプロシージャへの引数及び、どのような結果を返すか予め決めておかなければならないからである。WANのように広域で集中管理が不可能なネットワーク環境の場合、ネットワーク全体を把握することは困難で、それ故RPCでは柔軟なアプリケーションの設計が難しくなる可能性がある。

RPC とは異なるネットワークアプリケーション作成方法として RP(Remote Programming) という方法がある。RP はコンピュータ通信において、あるコンピュータから別のコンピュータにプロシージャをコールするだけでなく、実行されるプロシージャそのものを転送する。この場合ネットワーク上に流れる通信データは、プロシージャとそれに与える引数のデータである。また、プロシージャは送り手のコンピュータで処理が行われている途中の状態でも送ることができ、受け手側でその続きをすることができる。この場合のデータはプロシージャの移動する前の内部状態である。この内部状態を持ったプロシージャは、送り手、受け手のどちらでも実行することができることを強調して、移動エージェントと呼ばれている。移動エージェントは、プロシージャを送ったらネットワークを切断しても処理が継続できるため PHS や携帯電話などの通信環境には有力であることが、RPC と大きく異なる特徴である。また移動エージェントは自律的に動作するため、連携する他のソフトウェアやコンポーネントに与える影響は少ない。故に、急激に変化する分散システムに対応することが可能で、近年多くの企業も注目している。多くの利点を持っている移動エージェントだが、移動エージェント技術を現実的なアプリケーションに利用するにあたって、幾つかの問題を解決しなければならない。エージェント転送などのオーバーヘッドも問題の一つだが、その中で最も重要な問題は、セキュリティの問題と信頼性の問題である。移動エージェントは他のマシンに入り処理をするため、悪意のある移動エージェントからコンピュータを防がなければならない。このため移動エージェントの認証、アクセス制限などが必要とされている。また悪意のあるユーザから、クレジットカードのような秘密保持が必要とされる情報を保護しなければならない。このようにセキュリティは重要な問題である。信頼性の問題は、移動エージェントが障害によって移動エージェントの目的が達成することができなくなることで、例えばホストダウンによる移動エージェントの消滅などがある。銀行業務などのトランザクション処理を移動エージェントで実現したとする。この移動エージェントは入金情報などのデータを確実に相手に届くことが求められる。しかし、既存の移動エージェントシステムは、このような障害に対処していないため重要で信頼性が求められる処理を行うことはできない。本論文では信頼性の問題に関して述べる。

1.2 本研究の目的

本研究の目的は、信頼性の高い移動エージェントシステムを構成するための方法を考案することである。本研究でいう信頼性とは移動エージェントが計算機環境の物理的な障害に対応できる性質とする。既存の移動エージェントシステムである Aglets [1]、Kafka [2]

などは、計算機環境の障害に関して十分考慮された設計になってない。そのため計算機環境に障害が発生した場合、移動エージェントの処理を完結することができない。この時、計算機環境で発生する様々な障害を分析し、それぞれの障害レベルに応じた処理をイベント処理として実現するフレームワークである Coop を提案する。次に Coop を備えた移動エージェントシステムの実装を行なう。最後に例題を作成し、実装したシステムを評価する。

1.3 本論文の構成

本論文の構成は次のようになっている。２章では本研究で扱う移動エージェントに関する概念を説明し、幾つかの既存の移動エージェントシステムを紹介する。３章では、障害に関する基礎概念を説明し、移動エージェントシステム上で発生する障害に関して考察し、障害に関して幾つかの対処法を提案する。また、巡回移動エージェントがクラッシュ障害に対処するための１つの方法として分散アルゴリズムの説明する。４章では、Coop のシステムアーキテクチャを述べる。５章では、４章で提案したアーキテクチャを Java を用いて作成し、実際に作成した例題を紹介する。６章では、関連研究との比較や Coop の問題点を議論する。７章では、本論文をまとめ、今後の課題について述べる。付録では、本研究で利用した AgentSpace [13] や Coop の使用方法を説明する。

第 2章

移動エージェントシステム

本章では移動エージェントシステムの基本概念を説明する。次に既存の移動エージェントシステムを幾つか紹介する。

2.1 概念

現在、移動エージェントという用語に対してその定義が定まったとは言い難く、各々の論文によって定義が異なる。ここでは、一般的に言われている移動エージェントの性質について幾つか述べる。

- 自律性:外部から情報を取り入れ、各自の意志決定機構に従って行動することができる。
- 協調性:互いに情報をやり取りして、協調的に動作することができる。
- 柔軟性:環境の変化に対応することができ、適切な行動をすることができる。
- 移動性:ネットワーク上において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。

などが、よく言われる移動エージェントの性質である。しかし、これらの性質を十分に利用した例題は少なく、有効なプログラミング方法として利用されていないのが現状である。ここでは、広い意味でプログラムコードと実行状態を保持したまま移動することができるプログラムのことを移動エージェントと定義する。

一般的な移動エージェントのライフサイクルは生成、消滅、移動、到着、保存、復活、複製などがある。図 2.1 は、移動エージェントのライフサイクルを示した図である。

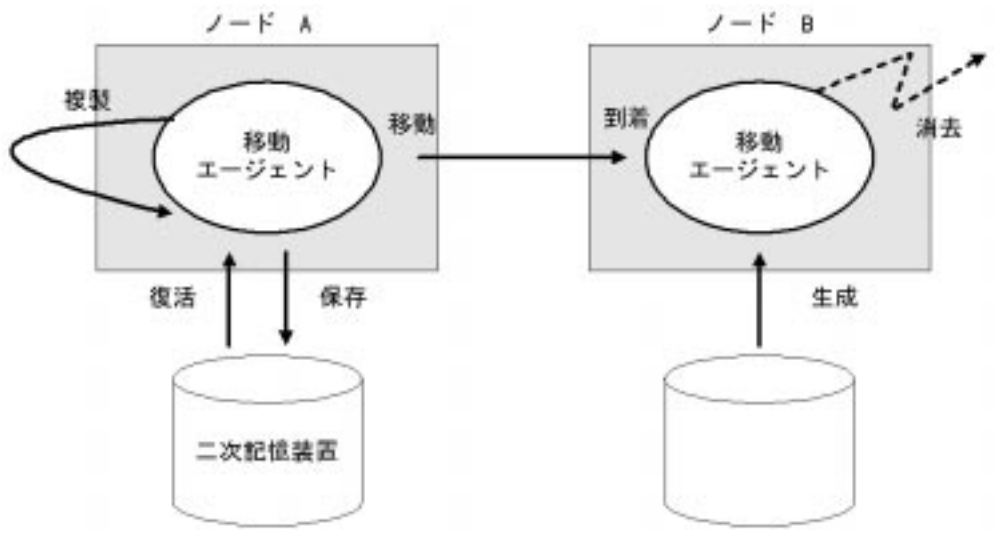

図 2.1: Agent のライフサイクル

2.2 既存の移動エージェントシステム

移動エージェントがあるノードで実行を途中で中断させて、その続きを別のノードで開始することをマイグレーションという。ソフトウェアエージェントは理論的に実行スタックやレジスタなど、すべての実効状態を凍結し別のノードでその続きから開始するのだが、現実にはシステムによっては幾つか制限がある。移動エージェントシステムの述べるにあたって、欠かすことのできないシステムとして Telescript 技術 [5] がある。Telescript は Telescript 言語を用いて移動エージェントを作成する。また、Telescript のマイグレーションは、プロセスの状態も移動することができる。現在、主流となりつつあるのが Java ベースの移動エージェントシステムである。Java が選ばれている理由として、プラットフォームを選ばない、ネットワーク上での安全確保を基本としたソフトウェア実行環境などが挙げられるが、特にシリアライゼーション (Serialization)、RMI(Remote Method Invocation)、ClassLoader を用いることで非常に簡単に移動エージェントシステムを実現することができることも、Java が選ばれる理由である。シリアライゼーションとは、オブジェクトを通信やファイルの入出力に適した形式に変換することである。シリアライズ

されたオブジェクトはデシリアライゼーション (Deserialization) することで、オブジェクトに変換され再び処理することができる。Java のシリアライゼーションで注意しなければならないのは、すべての状態をシリアライズすることはできないことである。これは Java 仮装マシンのアーキテクチャにより実行スタックやプログラムカウンタなどはシリアライズできないためである。そのためすべての Java ベースの移動エージェントシステムのマイグレーションは、実行スタックの状態を持って移動できない。この問題を解決しているが、Mobidget [14] で Java に非常に似た言語ではあるが Java 仮装マシンと互換性はない。RPC に似ている RMI は、実際にはサーバのコードを実行しているが、クライアントは自分のプロセス内のメソッド呼び出しと同じように見える。ClassLoader は、Java クラス・データを実行中の Java 仮装マシン上にロードして、Class として参照可能なオブジェクトに変換するためのクラスである。これらの Java の機能を用いることで、比較的簡単に移動エージェントシステムを作成することができる。Java ベースの移動エージェントの作成方法で参考となる文献として [4] を紹介する。次に代表的な既存の移動エージェントシステムを幾つか紹介する。

2.2.1 Telescript

移動エージェントの概念を初めて商用的なレベルに乗せたのは、General Magic 社の Telescript 技術 [5] である。Telescript は、プレース (Place)、エージェント (Agent)、トラベル (Travel)、ミーティング (Meeting)、コネクション (Connection)、オーソリティ (Authorities)、パーミット (Permite) の基本概念で実装されている。

- プレース：Telescript は、コンピュータネットワークをプレースの集まりとして考えている。プレースは移動エージェントが動作する「場」であり、１つのホストに複数のプレースがあっても良い。
- エージェント：通信型アプリケーションをエージェントと呼ぶ。
- トラベル：プレースから別のプレースへ移動することである。プレースはどんなに離れていてもよい。移動は、「go 命令」によっては別のプレースへ移動できる。また行き先はエージェントがもつ Ticket と呼ばれるオブジェクトに記述されている。
- ミーティング：エージェントが同じプレースでミートできる。相手のエージェントやミートするときの条件を引数として「meet 命令」で実行される。
- コネクション：異なるプレースにいるエージェント間で通信ができる。相手のエージェントや接続条件などを引数として「connect 命令」で実行される。

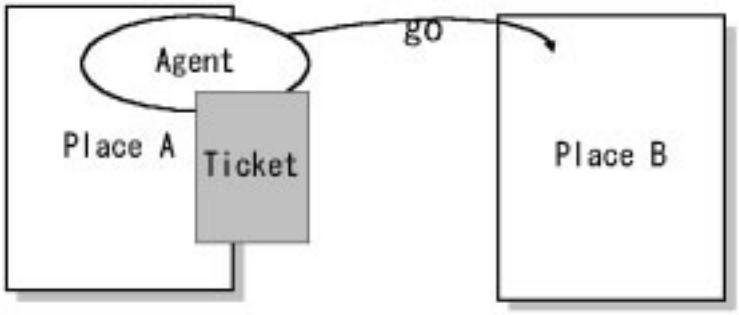

図 2.2: Agent の移動

- オーソリティ:エージェントやプレースを識別するためにオーソリティがある。「name 命令」によって調べることができる。
- パーミット：エージェントやプレースにパーミットを与えることで、オーソリティでの行動を制限することができる。エージェントやプレースは自分自身のパーミット値を見ることはできるが、増やすことはできない。「permit 命令」によって他のエージェントやプレースのパーミット値を知ることができる。

商用の移動エージェントとして、Telescript は既存の移動エージェントシステムに与えた影響は大きい。しかしプラットフォームが、Solaris、HP-UX などアーキテクチャに依存したため普及しなかった。障害処理としては、Telescript は命令が実行できなかった場合に例外を発生させ、例外処理をすることができる。現在 General Magic 社は Telescript からの移植で Java ベースの移動エージェントシステムである Odyssey[6] を提供している。Odyssey やこの後に紹介する Voyager[15] は、CORBA(Common Object Request Broker Architecture)[7] や Microsoft の DCOM(Distributed Component Object Model)[8] と連携しているため、商用として使える移動エージェントシステムと言えるだろう。Telescript を分かり易く解説している文献として [9] を紹介する。他の移動エージェントも CORBA や DCOM と連携してくることは十分考えられるので、CORBA や DCOM などの分散オブジェクトの動向を詳しく紹介している文献 [10] を紹介する。

2.2.2 Aglets

初めて Java ベースの移動エージェントシステムを商用的レベルで提供したのは、IBM 社の Aglets [1] である。Aglets のキーワードを幾つか挙げて紹介する。

- J-APPI(Java Aglet API):J-APPI で作られた Aglets は、J-APPI をサポートするホスト上なら、移動先のホストのハードウェア環境や OS を気にせずに動作できることが保証されている。
- AgletContext：Aglet は AgletContext を通じて、環境（ホスト）と対話する。
- Message：Message は他の Aglet に同期、非同期でメッセージを送られるオブジェクトである。
- AgletProxy：Aglet は AgletProxy を利用して他の Aglet と Message をやり取りする。

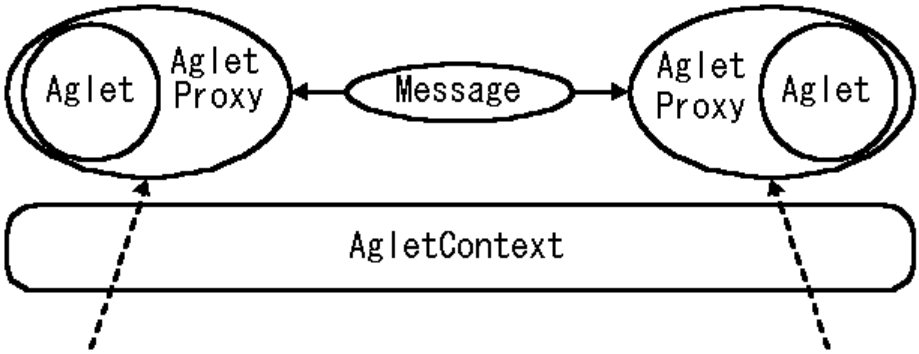

図 2.3: Aglet Object Model

- FutureReply：非同期な仕事をするときのインターフェースを提供している。レシーバーはこのインターフェースを使って直ちに返答するか、指定した時間内に返答がこなかった場合に仕事が続けられるようにタイムアウトの指定をすることができる。
- Itinerary：Aglet の移動パターンで、non-trivial な旅行計画や routing の抽象化を提供する。
- AgletIdentifier：それぞれの Aglet に識別子が割り当てられ、それは大域的に unique で、Aglet の生存期間中不変である。
- ATP(Agent Transfer Protocol)：ATP[11] は分散エージェントベースの情報システムに対するアプリケーションレベルにおける標準プロトコルである。ATP は OMG(Object Management Group)[12] の Mobile Agent Facility のプロポーザルとして提出されている。

- Tahiti：ユーザは Tahiti と呼ばれている GUI ベースの管理ツールを使って Aglet の生成、消去、送受信、セキュリティの設定などを制御することができる。
- Fiji：WWW ブラウザを通じて Aglet を実行することができる。

また、Aglets は移動したエージェントを引き戻すことも Tahiti を用いることで可能である。

Aglets は、単純な障害処理をサポートしている。移動エージェントが完全に移動先に到着したことが分かってから移動元の移動エージェントを消すことで、移動中の通信障害に対処している。また、移動先に到着しなかった場合は別のホストに移動するなどを例外処理として定義できる。

2.2.3 Kafka

Kafka[2] は Java ベースの移動エージェントシステムで富士通研究所が提供している。他の移動エージェントシステムと異なる顕著な特徴として、リフレクション、分散ディレクトリがある。

リフレクションは、エージェントの振る舞いを記述している action の定義をランタイムに変更でき、このことによりシステムに柔軟性を与えられる。この action はクラス Action のサブクラスとして定義され、このサブクラスは add(),set(),get(),remove() によって関連するクラスの追加・削除が行われる。add() インターフェースを使って、UNIX のファイルシステムのように属性を変更できるのも Kafka の特徴である。移動エージェントのコード転送は RPC を用いたアプリケーションに比べると通信のオーバーヘッドが大きい。しかし移動先の環境に必要なコードだけを転送することで、移動コードのオーバーヘッドを削減できる可能性があり、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に動的に対応できるようになることが、リフレクションの利点である。元々Java は、リフレクティブな実行環境を構築することができる。Kafka では不要なクラスを削除する機能を持つ ReflectiveClassLoader が定義してある。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。上記１，２は Kafka ではプログラマーに任されている。上記３に対しては Kafka で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ようなっている。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。

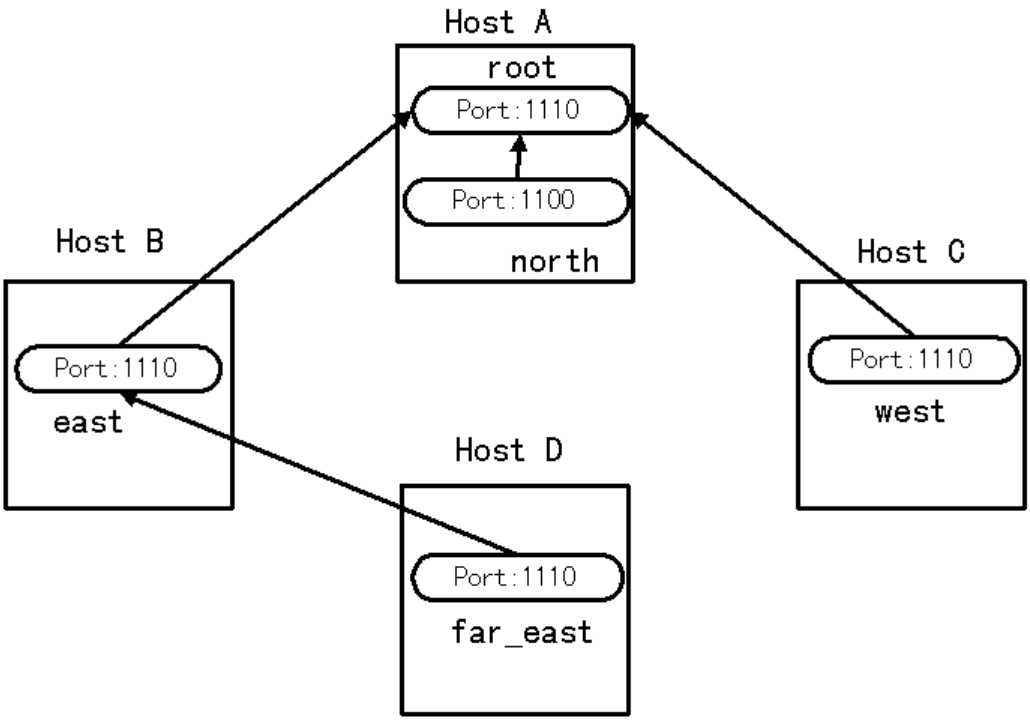

図 2.4: 分散ディレクトリの例

分散ディレクトリは名前管理サービスのことである。名前管理サービスとは、移動エージェントサーバを図 2.4 のように構築することで、移動エージェントはアドレスを知ることなしに、エージェントサーバ名からエージェントのリファレンスが取得でき、巡回することができる。図 2.4 では、ホスト A に”root”と”north”、ホスト B に”west”、ホスト C に”east”、ホスト D に”far _ east”というノードが立ち上がっている。ディレクトリ名は root を省略して”:”で区切られるため、それぞれ”:root”,”:north”,”:west”,”:east”,”:east:far _ east”となる。ホスト D の”far _ east”でエージェント名が”Test-Agent”という名前のエージェントを起動した場合、この”Test-Agent”のグローバル名は”:east:far _ east:Test-Agent”となる。このグローバル名を使えばどのホストからでも、”Test-Agent”のリファレンスが取得できる。分散ディレクトリを用いることでエージェントを検索することもできる。検索手順は、ローカルディレクトリ、子ディレクトリ、親ディレクトリ、と再帰的に検索して、root に到達するまでに見つからないと例外を発生させる。同じエージェント名が複数存在した場合は、比較的近いエージェントのリファレンスが取得でき、エージェント名が分散ディレクトリで唯一の場合は常にエージェント名だけで検索できるという利点がある。現在、Kafka は Action のみ移動でき、Agent が移動する部分は未実装である。

2.2.4 Voyager

Voyager[15] はJavaベースの移動エージェントシステムでObjectSpace社が提供している。Voyagerの特徴として、vcc(Virtual Class Compiler) と呼ばれるプリコンパイラを使い、リモート先でも同じようにアクセスできるVirtualクラスを作成する。このVirtualクラスはリモート先でインスタンスを生成することができ、ロケーションを意識しないVirtual参照をすることができる。さらにVoyagerが他のJavaベースの移動エージェントシステムと大きく異なる点は、独自のORB(Object Request Broker) を持っていることである。これによって異なるVM上ではRMIより1.2〜2倍高速に通信できる。Aglets同様にMessengerを用いてMessage Passingを実現している。Messengerのサブクラスとして、Sync（同期)、Future（非同期)、OneWay（戻り値なし）の３つのクラスが実装されている。グループ別に移動エージェントを送ることができる、マルチキャストがサポートされている。障害処理としてシンプルなチェックポインティングを採用しているのもVoyagerの顕著な特徴である。

2.2.5 TACOMA

FT(fault tolerance) を備えている移動エージェントシステムであるTACOMA[16] は、Tcl上で実装されている。プログラムやデータをfolderに入れ、meetコマンドによってfolderの受け渡すことによって移動という概念を抽象化している。エージェントが移動するときはエージェントをfolderに入れ、folderを送る能力のあるエージェントに渡し、送られる。TACOMAはチェックポインティングを採用し、FTを備えている。また、図2.5のようなシンプルな移動方法に対して図2.6は１つの仕事に対してレプリカを複数生成して複数のホストに送ることで、ホストダウンの障害に対処することもできる [17]。

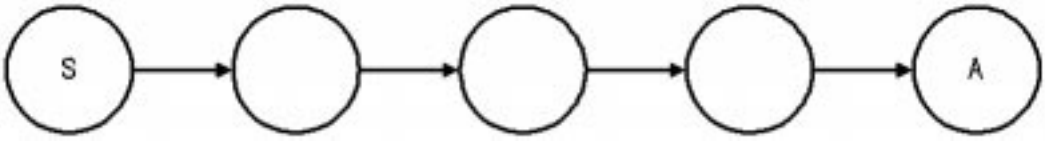

図 2.5: A simple agent computation

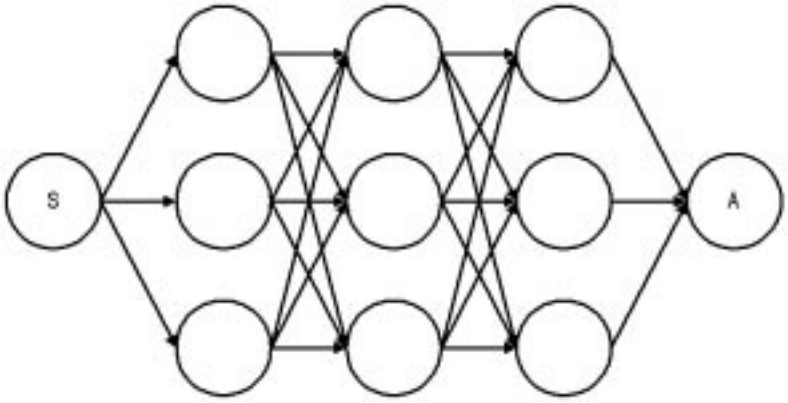

図 2.6: Replicated agent computation with voting

2.2.6 Ajenta

Ajenta[18] は Java ベースの移動エージェントシステムである。他の移動エージェントシステムと異なる顕著な点は、例外処理に関して考慮してあることが挙げられる。Ajenta や Aglets は、移動エージェントを移動先から強制的に回収することができる。このような移動先のエージェントを遠隔操作する場合、セキュリティの問題を解決しなければならない。システムが移動エージェントの遠隔操作のセキュリティを保証しているのは Ajenta だけである。Ajenta は、移動エージェントに仕事を頼んだオーナーのみが移動エージェントを遠隔操作で強制的に移動先から回収したり、オーナーが遠隔操作で移動先の移動エージェントの処理を終わらせることができる。この機能を用いることで、例外が発生した場合に対処することができる。例えば予期できない例外が発生して、移動エージェントが操作できなくなった場合には、回収したり、そのホストで終了させたりできる。また例外が発生したとき、その状態を保持した移動エージェントを回収し、正しい状態に訂正することができた場合、その続きから処理を再開できる。ただし Java のシリアライズを使って実行状態を取得しているため、スレッドレベルの実行状態は取得できない。

2.2.7 AgentSpace

AgentSpace は、Java ベースの移動エージェントシステムで、高階移動エージェントシステムである。高階移動エージェントとは、エージェントの中にエージェントを包含することがきることをいう図 (2.7)。AgentSpace は、他の Java ベースの移動エージェントシステムと異なる点として、移動方法を挙げている。既存のシステムである Aglets、Kafka、

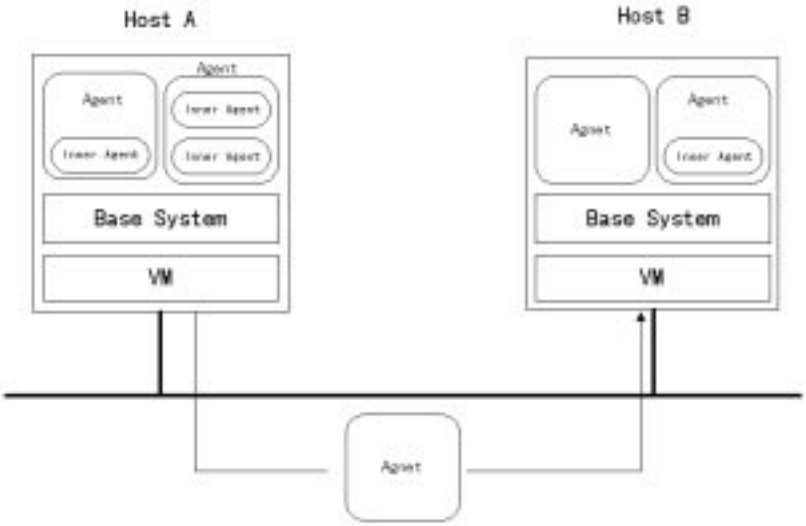

図 2.7: AgentSpace システム構成

Voyager、Odyssey などは移動方法に関して問題がある。これらのシステムは Java 言語の動的クラスローディングを基礎としている。転送後ある程度の時間が経過してからクラスコードが必要となる場合に、クラス転送が必要とされることがある。このため、通信回数が１回とは限らず、通信状態が不安定なホスト上で、動的クラスローディングを行った場合、通信ができなかったときには、それ以上の処理を継続することが不可能となる。このため、AgentSpace では一つの移動エージェントを構成する実行状態や必要とするすべてのクラスファイルをひとつにまとめ、移動エージェントを転送する方法を採用している。この方法によって、通信回数を削減できる。また、他のシステムの方法では一つメッセージの通信遅延が大きくなればなるほど転送時間が増大化するという問題を避けることができるため、遠隔化されたコンピュータ間でのエージェント移動には有効である。AgentSpace は GUI ベースの管理ツールを使って移動エージェントの基本的な動作を管理することができる。Aglets のようにセキュリティの設定や、移動した移動エージェントを自分のホストに呼び戻す機能はない。また、AgentSpace はソースコードを学生の研究用に公開されている。

第 3章

障害処理

本章では、はじめに障害の基礎概念を説明する。次に移動エージェント上で発生するの障害例を幾つか挙げ、その障害に対する障害処理について述べる。次に Coop で保証する最低限の障害処理について述べ、最後に巡回移動エージェントで、クラッシュ障害に対処できる単純な分散アルゴリズムの紹介する。

3.1 障害

移動エージェント上で発生する障害を述べる前に、障害についての基礎概念を述べる。障害には非ビザンチン的障害 (non-Byzantine fault) とビザンチン障害 (Byzantine fault) がある。非ビザンチン的障害とは障害の内容が動作の時間的遅れのみに影響する障害のことである。障害を発生した要素が動作を停止させ、外部に何らかの応答もしない場合は、クラッシュ障害 (crash fault) であり、規定の時間に応答しない場合はメッセージ抜け障害 (omission fault) またはタイミング障害 (timing fault) である。この種の多くはタイムアウト (time-out) で検出される。ビザンチン障害は障害を発生した要素があたかも正常であるかのように振る舞いながら、誤ったメッセージやデータを送り/返し続ける。意図的にこのようにプログラムされた場合もこれに含まれる。本研究で扱う障害はハードウェアに起因される非ビザンチン的障害であり、ビザンチン障害に関しては、文献 [3] で詳しく説明されている。以後、本論文では特に断りがない場合、「ハードウェアで起因される非ビザンチン的障害」を「障害」とする。

次に障害に対処するために必要と思われる手法を述べる。クラッシュ障害などに対処できる有効な手法として、チェックポイントとレプリカを用いて対処する方法がある。チェックポイントとは、プログラムの中断点 (breakpoint) のことで、障害から復帰したときに

チェックポイントを利用することで、処理をその途中から再開できる。データの多重化であるレプリカは、信頼性向上の最も重要な手段の１つであるが、レプリカの一貫性を保たなければならない問題がある。多重化にもいろいろは方法がありその用途も幾つかあるが、主に複数のレプリカを用いて同時に処理をすることで信頼性を向上させる方法と、本体に障害が発生したときにレプリカに切り替えて処理を続ける方法がある。移動エージェントの場合、チェックポインティングは移動エージェントをある状態まで戻して、仕事を再開することから、結果的にレプリカを用いて同じことをすることができる。本研究では、レプリカで代用するチェックポインティングを擬似チェックポインティングと呼ぶことにする。

ソフトウェア障害に対処する古典的な方法は、同じ目的の仕事に対して異なるアルゴリズムや異なるプログラミング方法を用いて移動エージェントを作成し、ソフトウェア障害に対処する方法がある。また、同じ目的の仕事をする移動エージェントを異なる移動エージェントシステムで動かすことで、信頼性を高めることができ、これらは有効な手段だと考えられる。ただし、厳密に行わなければならないような処理を除けば、このような方法は現実的ではないと思われる。それ故、例外処理をうまく用いて実行中のエラーを回避する方法を考える必要がある。例外処理に関しては、７章の今後の課題で述べる。

3.2 障害例

本研究では移動エージェント上で発生する計算機資源の枯渇問題 (resource starvation problem) も障害として扱うことにする。現在、移動エージェント上で発生するは下記のような障害があると考えている。

1. ホストがダウンする場合。
    - 予想できる場合。
    - 予想できない場合。
2. 移動先で期待していた計算機資源が使えなかった場合。
    - メモリが少ない、CPU のパワー不足など。
3. 通信が不可能になった場合。
    - 移動前に移動先に通信障害が発生した場合。
    - 移動中による通信障害が発生した場合。

4. ユーザーの期待している時間内に移動エージェントが処理結果を得られない場合。

   - 移動エージェントが処理を回復できないなど。

上記１は、ホストダウンによる障害で予想できる場合とそうでない場合の２種類がある。予想できる場合（移動エージェントが消滅しない）は、バッテリー切れ、UPS（無停電電源装置）の起動、検出可能なシャットダウンなどが挙げられる。この場合の障害に対しては、移動エージェントはホストダウンが発生する前に二次憶装置へ永続化し、障害回復後に計算を再開するか、別のホストに移動して処理を続けるなどの対処する。予測できない場合（移動エージェントが消滅する）としては、停電などによる突然のホストダウンなどがある。この場合の障害に対しては、チェックポインティングを用いる方法で対処する。移動エージェントは一定間隔の時間で、その実行状態を二次記憶装置に保存するチェックポインティングを行う。これによりホストが回復した後、チェックポイント（記録された実行状態）からその処理を再開することができる。また、本研究では巡回エージェントにおいて、このクラッシュ障害に対処する１つの解決策として単純な分散アルゴリズムを提案し、この障害に対処する。

上記２は異動先のホストでメモリや CPU のパワーなど期待していた計算機資源が得られなかった場合がある。この場合の障害に対しては、計算機資源の解放を待つか、別のホストに移動するなどの対処をする。

上記 3 は、移動前の通信障害と移動中の通信障害がある。移動前の通信障害に対しては、移動先を変えて仕事を続けるか、移動先に依存した仕事をする場合は通信障害が回復するまで待ち続けるなどの対処をする。移動中の通信障害に対しては、移動する前に移動エージェントのレプリカを作成し、確実に移動先に到着するまで繰り返し転送するなどの対処をする。Aglets などがこの方法で障害処理を行っているが、この障害を対処していない移動エージェントは、この方法で対処する。

障害４は、ユーザが時間内に移動エージェントの処理結果を得ることができない場合である。この場合、障害を確定することは難しい。例えば、チェックポインティングした実行状態が破壊された場合や長時間ホストダウンが回復しなかった場合などが考えられる。この障害に対する処理は、移動エージェント自体に問題がないと想定して、移動先を変えて仕事を継続する方法で対処する。

本研究ではこれらの障害に関して、対処できる処理機構を提案する。

3.3 基本障害処理

本研究のシステムが最低限の保証をする障害処理を基本障害処理と呼ぶ。障害例で挙げた障害処理は類似した処理をしているため、これらの障害処理を基本障害処理として扱う。下記のような基本障害処理を考えている。

- 移動エージェントが動作しているホスト上の障害に対する処理。
  - 別のホストに移動して、仕事を続ける。
  - 移動エージェントを二次記憶に保存して、障害回復後に処理を再開する。
- 移動先に行けない障害に対する処理。
  - 移動先を変更して仕事を続ける。
  - 移動先が回復するまで待機する。

基本障害処理の例を図 3.1 で示す。１, ２は、ホスト A からホスト B へ移動、ホスト B で計算機資源不足を検知したので、ホスト C へ移動する例である。　１', ２' は、ホスト A においてホスト B の通信障害を検出、ホスト C へ移動先を変更し移動、到着後にホスト C でシャットダウン情報を検出したので、二次記憶装置へ永続化するという例である。これらの障害を基本障害処理として移動エージェントは備えている。基本障害処理をカスタマイズすることで、個々の仕事に合った障害処理をすることもできる。

3.4 分散アルゴリズム

クラッシュ障害に対処できる方法として本研究で提案する分散アルゴリズムを紹介する。このアルゴリズムの目的は、一部のホストダウンによって移動エージェントの仕事が途中で停止することが起きないように、最低限何かしら仕事を行い、開始ホストまで戻ってくることを目的としている。分散アルゴリズムの前提を以下で述べる。

1. 巡回移動エージェントはホスト 1,2,3,4, ···· ,n,n+1,n+2, ···· と巡回し、最後に開始ホストに戻ってくる。
2. すべてのホストで仕事を行わなくても良い仕事のみを対象とする。

この分散アルゴリズムは特定の条件の仕事に対して有効であり、単純にブロードキャスト方式などと比較することは無意味である。この分散アルゴリズムを用いた有効な例題と

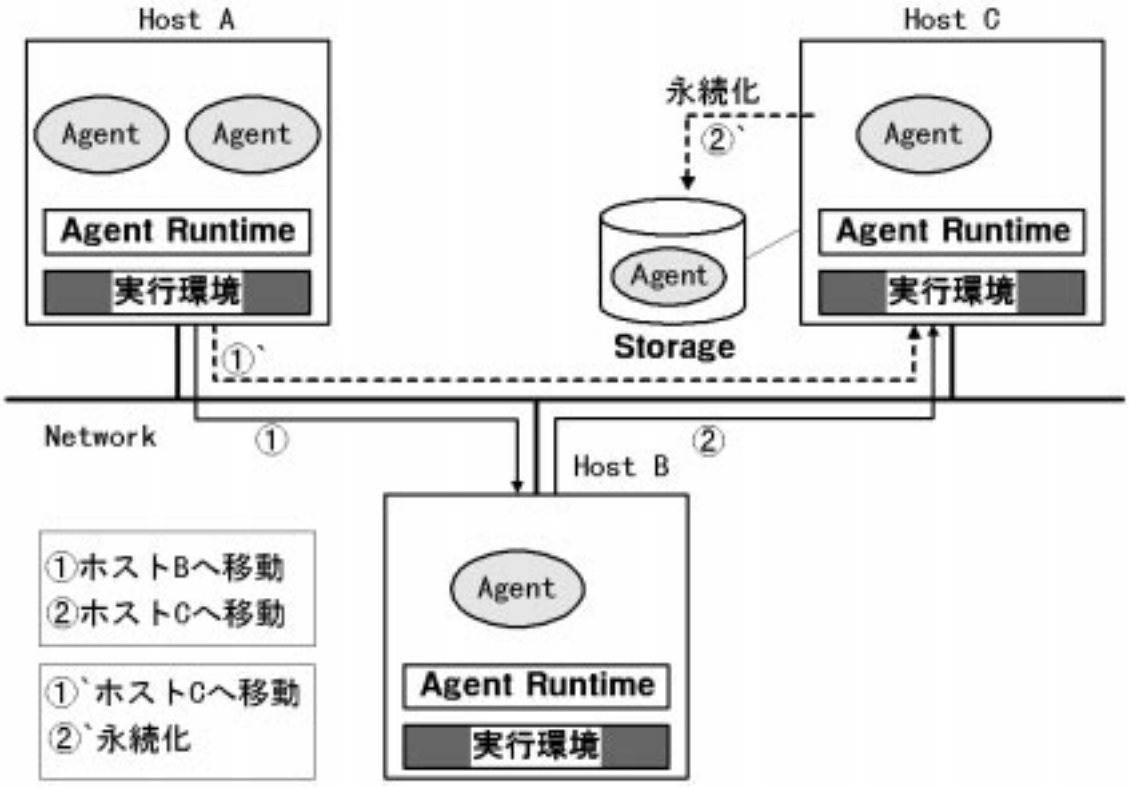

図 3.1: 基本障害処理の例

して回覧板のような仕事が考えられる。回覧板については５章の例題で詳しく説明する。この分散アルゴリズムは移動エージェントの他に下記のものを用いて実現している。

- レプリカ：移動エージェントのレプリカで、別のホストに移動するときに作成され、二次記憶装置に永続化する。障害が発生したと思われる時に、二次記憶装置から活性化（二次記憶装置に保存してある状態からの復活）され別のホストへと移動する。
- ACK(acknowledge):移動エージェントがホストに到着したときに、到着したことを移動元へ知らせるために用いる。本論文のアルゴリズムでは、タイマーが起動されているホストに対してのみ ACK を返す。
- 管理タイマー：移動エージェントが移動した後に管理タイマーが起動されて、ACK を監視する。障害が発生したと思われた場合、二次記憶装置で永続化しているレプリカを活性化する。

この分散アルゴリズムは、各ホストで管理タイマーを起動して、障害が発生したと判断したときにレプリカを、障害が発生したと思われる次のホストへ送るアルゴリズムである。このアルゴリズムは、すべてのホストで管理タイマーを起動しなくてもいいようになっている。何故なら、実際に利用する場合はネットワークに非常時接続のノートパソコンや、無線LANなどの通信状態が不安定なホスト上で、管理タイマーを起動することはこの分散アルゴリズムでは有益ではないからである。通信環境が不安定な計算機上では、移動エージェントは実行や移動のみの処理をすることが望ましい。管理タイマーの数を増やすことで、高い確率で実行結果が得ることができるが、その分ACKの通信量が増えてしまう。次に、移動エージェント、レプリカ、ACK、管理タイマーが幾つかの状況に対してどのように対処するのか、図を用いて説明する。

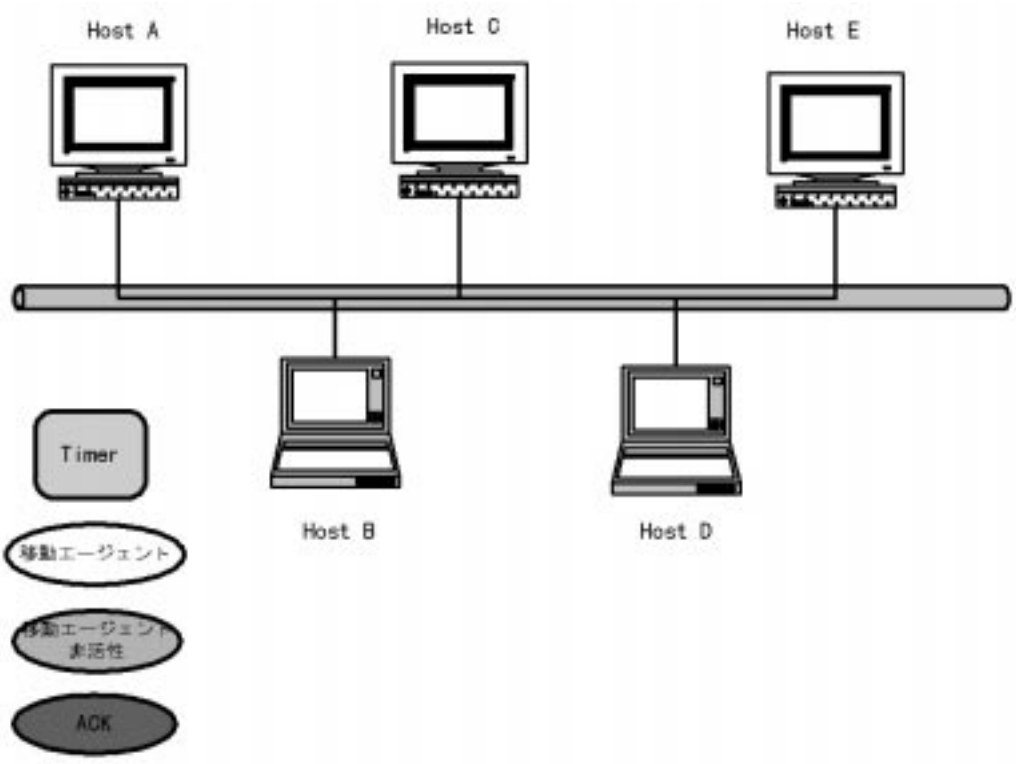

図 3.2: 構成例

図 3.2はホストA,B,C,D,Eがネットワークに接続され、各ホストで移動エージェントサーバが立ち上がっている。ホストAからホストBへ移動し、さらにホストC,D,Eと巡回して最後にホストAに戻ってくる移動エージェントを対象としている。このようなネットワークに接続されたホスト間を移動する移動エージェントを対象として分散アルゴリズ

ムの説明を述べる。

- Timer:管理タイマーで、どのホストにも起動することができる。
- 移動エージェント：巡回リスト順に各ホストへ移動する。
- 移動エージェント 非活性：移動エージェントのレプリカが二次記憶装置の保存されている状態。レプリカの実行状態は、そのホストで仕事を終えて、次のホストへ移動する直前の状態で保存してある。
- ACK：ACK は、移動先に移動エージェントが到着したことを、管理タイマーが起動しているホストへ返す。ACK は ACK エージェントを用いて実現する。

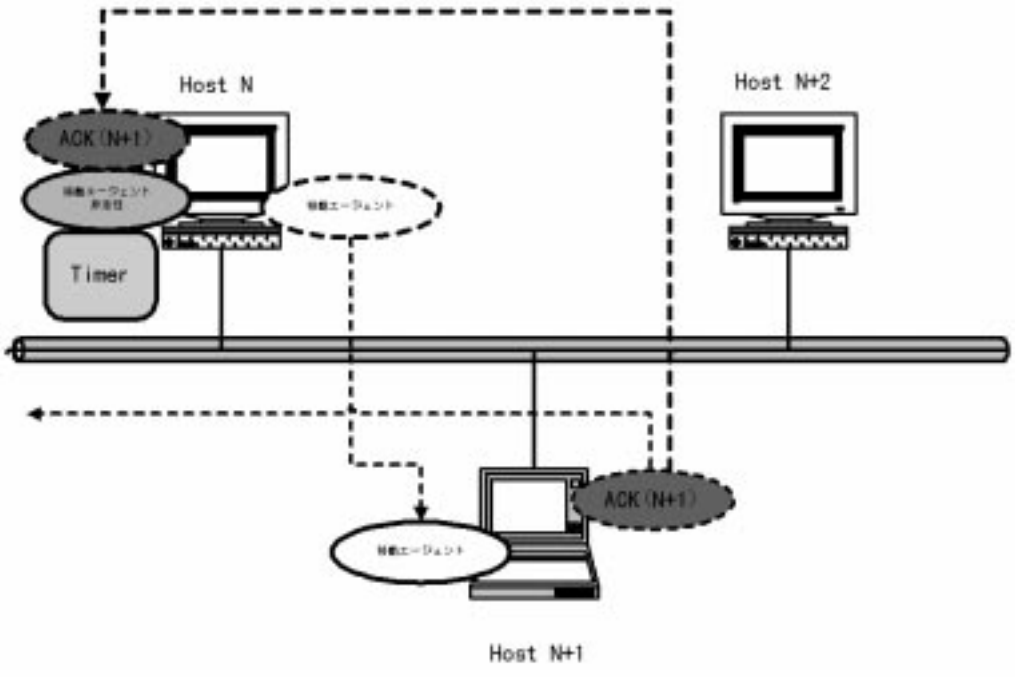

図 3.3: 移動エージェントの移動例１

(i) 移動先に移動エージェントが初めて訪れた場合。

図 3.3 はホスト N からホスト N+1 へ移動したときの図である。

移動エージェントはホスト N から移動するときに、移動直前の実行状態のレプリカを作る。作成されたレプリカは非活性され、二次記憶装置へ永続化される。本体の移動エージェントはホスト N+1 へと移動する。移動と同時にホスト N ではタイマーを起動され、

ACK の監視を開始する。移動エージェントが移動先のホスト N+1 に到着したときに、ホスト N+1 の情報を取得する。その情報は、ホスト N+1 が移動エージェントが初めて訪れたホストという情報である。移動エージェントは、初めて訪れたホストなので今まで訪れたタイマーが起動しているホスへ ACK を返し、そのホストで仕事を開始する。

ACK は送り先に到着しなければならないが、必ず届かなければならないというわけではない。ただし、ACK が届かないとタイマーが ACK を検出することはできないので、現在、移動エージェントが動作していると思われるホスト上で障害が発生したと誤認して、レプリカを今いると思われるホストの次のホストへ送る。ここで、もし障害が発生していなかった場合には矛盾が起きてしまう。この矛盾の解消のために移動エージェントは常に１つとなるように移動先のホストで、その矛盾に対処する。この対処については後で説明する。

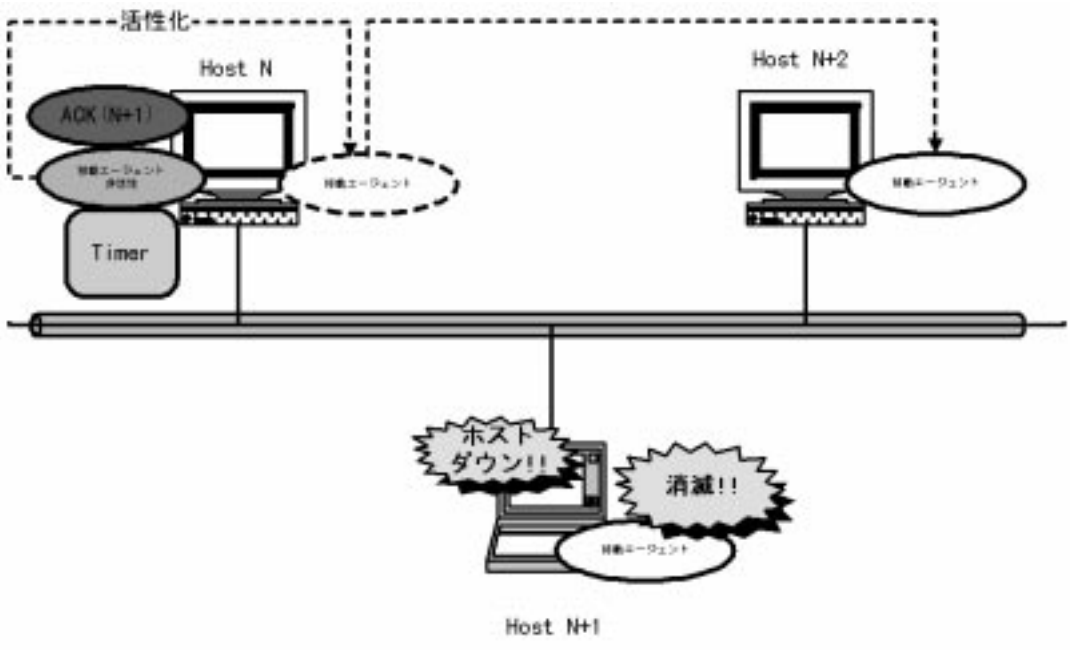

図 3.4: 移動エージェントの移動例２

(ii) ホストダウン発生した場合（図 3.4）。

移動先であるホスト N+2 は、移動エージェントが初めて訪れるホストである。

ホスト N+1 にホストダウンが発生してホスト N+1 上で動作している移動エージェントが消滅したとする。この場合ホスト N のタイマーはホスト N+2 からの ACK を受け取ることはない。ホスト N はホスト N+1 がホストダウンしたと想定して、ホスト N のレプリ

カをホスト N+2 へ送る。ホスト N+2 に到着した時に、移動エージェントはホスト N+2 から情報を取得する。取得した情報からホスト N+2 が初めて訪れたホストと判断すると、今まで訪れたタイマーが起動されているホストへ ACK を返し、移動エージェントの仕事を開始する。

もしホストダウンがホスト N+1 ではなくてホスト N+2 だった場合は、ホスト N+1 の移動エージェントはホスト N+2 へ移動することができず、ホスト N+3 へ移動するこになる。その後、同様の動作をして ACK をホスト N へ返すことになる。この場合、ホスト N はホスト N+2 が何らかの障害が発生したと判断できる。ホスト N のタイマーはホスト N+4 以降のホストから ACK を待つことになる。

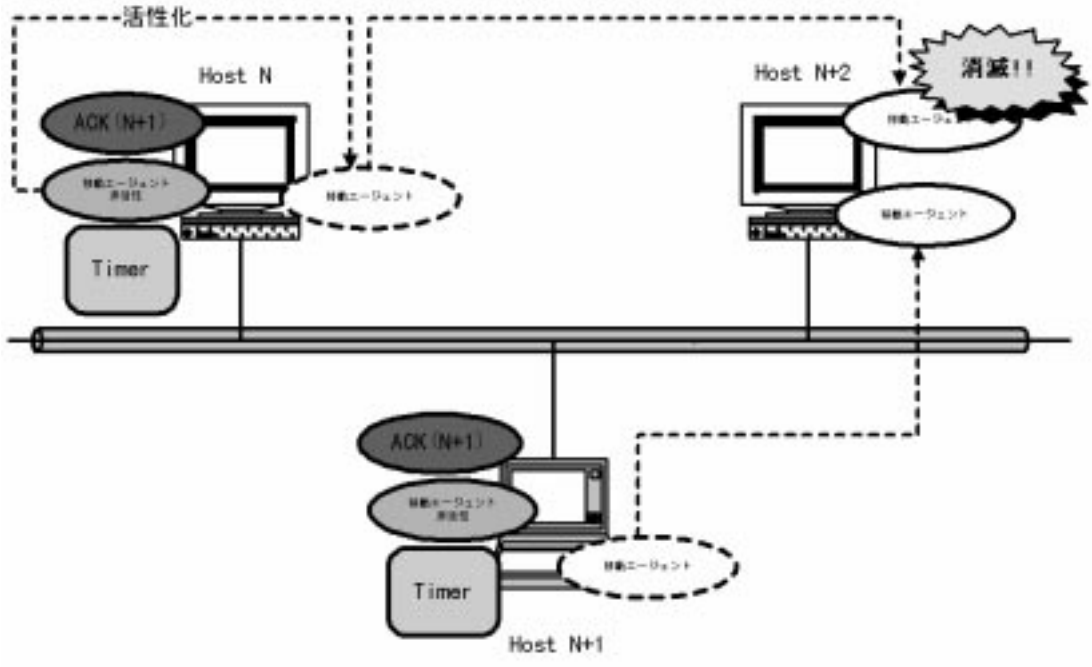

図 3.5: 移動エージェントの移動例３

(iii) 例２でホストダウンが起きていなかった場合、その１（図 3.5）。

現在、移動エージェントが動作しているホスト N+1 に一時的な通信障害が発生したとする。ホスト N のタイマーはホスト N+2 から ACK を受け取らないので、ホスト N はホスト N+1 にホストダウンが発生したと想定して、ホスト N+2 へレプリカを送ってしまう。しかし、ホスト N+1 の通信障害が回復して、しばらくしてホスト N+2 へ移動できるという例を説明する。

ホスト N のタイマーがホスト N+2 から ACK を受け取らないので、レプリカを活性化し移動エージェントはホスト N+2 へ移動する。ACK は訪れたホストでタイマーが起動しているホストのみを返すので、ホスト N+1 には返さない。ホスト N+2 でホスト N から来た移動エージェントが実行中に、ホスト N+1 から移動エージェントが到着したとする。この場合、ホスト N+1 から来た移動エージェントの方がホスト N の移動エージェントより多くのホストを訪れている（仕事量、情報量が多い）ので、ホスト N から来た移動エージェントは実行中にも関わらず消滅し、ホスト N+1 の移動エージェントが処理を始める。また、ACK はホスト N へも返すが、ホスト N のタイマーはホスト N+3 以降の ACK を待ってるので、この ACK は無視される。

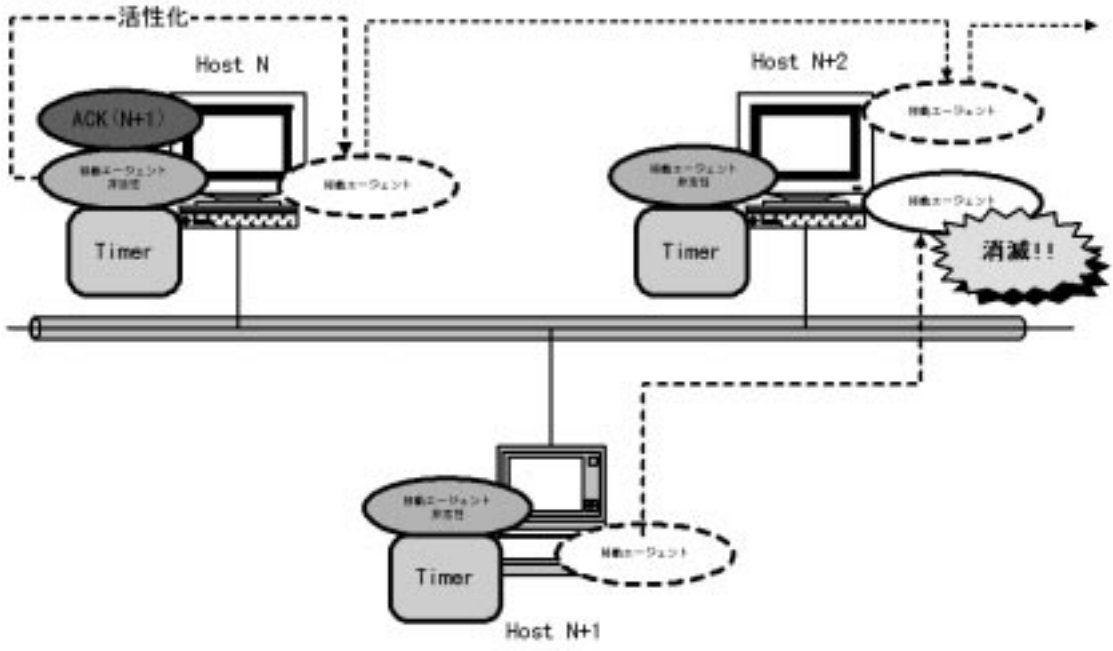

図 3.6: 移動エージェントの移動例４

(iv) 例２でホストダウンが起きていなかった場合、その２（図 3.6）。

例３同様に、ホスト N のタイマーがホスト N+2 から ACK を受け取らないため、ホスト N はホスト N+1 にホストダウンが発生したと想定して、ホスト N+2 へレプリカを送る例を説明する。

例３と異なる状況は、ホスト N からホスト N+2 へ移動した移動エージェントが、ホスト N+2 で仕事を終えてホスト N+3 へ移動した後に、ホスト N+1 からホスト N+2 へ移動エージェントが到着した場合である。実行中の移動エージェントが複数存在することを防

ぐために、ホスト N+1 から来た移動エージェントは消滅させる。

例２、例３で説明した方法で、ホストに到着した移動エージェントが消滅したり、仕事を引き継ぐことで複数のレプリカが同時に仕事をすることを防いでいる。

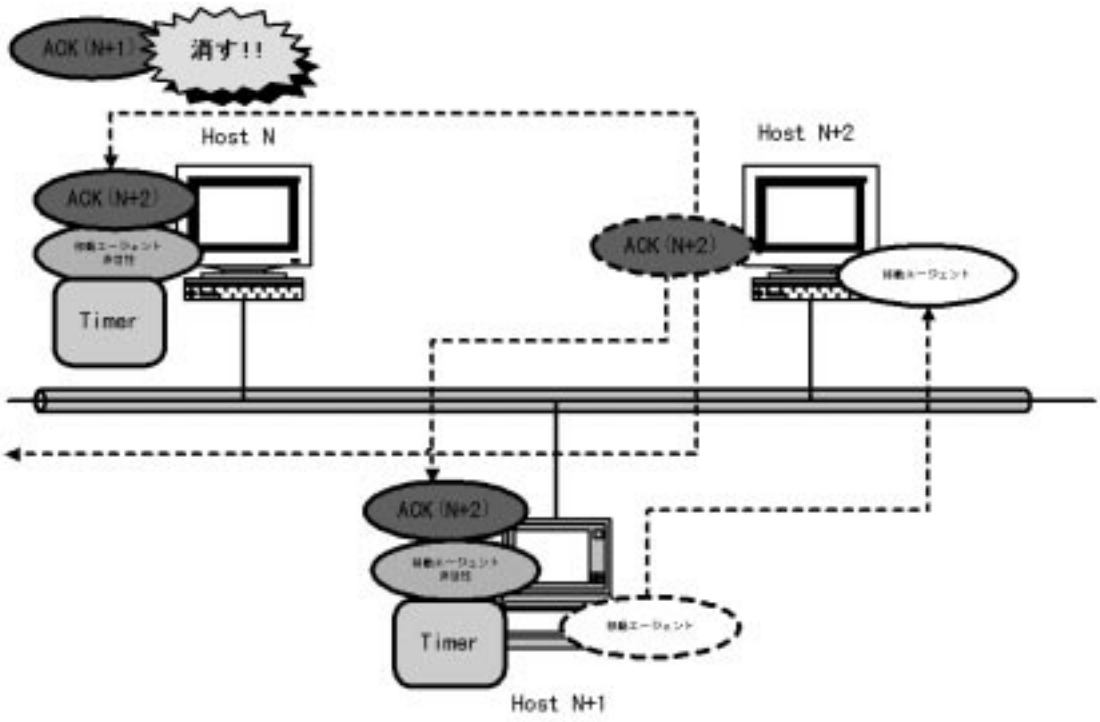

図 3.7: ACK の移動例

(v)ACK の動き（図 3.7)。

各ホストのタイマーは ACK を受け取り、常に新しい (一番ホストを訪れた移動エージェントが返す)ACK を保持する。タイマーは、現在移動エージェントが実行していると思われるホストから ACK を受け取らないとレプリカを活性化して、次のホストへレプリカを送る。

最終ホストから ACK を受け取ると、仕事は終了したと判断し、タイマーと非活性されているレプリカは消される。また、通信障害などで最終ホストから ACK を受け取れなかった場合、受け取らなかったホストのタイマーは、レプリカを次のホストへ送ることになるが、移動先のホストは、移動エージェントの仕事がすでに達成している情報を持っているので、すぐにその移動エージェントは消滅して、仕事が終了したこと知らせる ACK を移動元に返す。また、移動エージェントには寿命が設定してあるので、寿命が来ると各ホストで保存してある移動エージェント情報や、残っているレプリカ、タイマーは消去される。

この分散アルゴリズムによって起こり得る最悪の場合は、余り仕事をしていない移動エージェントがタイマーにより終了ホストに近いホストへ移動し、そのホストで、移動エージェントが仕事を終えて次のホストに移動する時である。例えば、この問題を回避するために、移動した移動エージェントよりも多くの仕事をした移動エージェントが来た時に、その移動エージェントを消さないで処理を続けたとする。その場合、最終ホストでは幾つかの移動エージェントが到着する事になってしまう。単純な情報を集めてくるような仕事の場合は、一番多い仕事をしたエージェントを採用すれば良い。この場合、移動エージェントの訪れたホストを見て移動エージェントの終了を判断するか、移動エージェントの寿命を待って、最後に一番仕事量の多い移動エージェントを採用すればいい。しかし、このような仕事内容の場合は、巡回する意味がを持たせることが難しい。

また、ユーザは各ホストで実行に必要な時間を知る必要があることと、タイマーで監視する時間を設定する必要がある。タイマー時間は移動する時間や実行時間を考えて十分長い時間を設定すればよいと考えている。実行時間が予想できない仕事（ユーザに入力を求める場合など）は、タイムアウト機能で移動させる必要があると考えている。

第 4章

Coopのシステムアーキテクチャ

本章では、はじめにCoopの設計方針を述べる。次にCoopの処理部分について述べる。Coopは障害に対処するレベルを障害対処機構と管理モジュールに分けることができる。障害対処機構は各々の移動エージェントが持っている処理機構である。管理モジュールは、障害対処機構では対処できない処理をする。

4.1 設計方針

設計方針は、移動エージェントシステムに組み込むのではなく、移動エージェント自身に障害対処機構を持たせる方針を採用した。これにより、

- 移動エージェントシステムのプラットフォーム依存しないため移植性が高い。
- 障害処理部分が独立しているため、モジュールとして取り扱うことができる。
- 障害処理機構のカスタマイズが容易。

という利点がある。ここで、障害対処機構をシステムに組み込む場合を考えてみる。例えば、障害対処機構のカスタマイズを行ったとすると、その障害処理をするためには、すべてのホストに新しいシステムを再インストールしなければならない。これでは、柔軟な障害処理をすることは難しく、また各々の移動エージェントに合った障害処理を行うことも難しくなる。これらの機構は移動エージェントシステムより上位に組み込むため、プラットフォームに依存しないアーキテクチャで作ることができる。設計の利点・問題点については7章の議論で詳しく述べる。

Coopのアーキテクチャとして図4.1を示す。各ホストの実行環境の上で移動エージェントシステムであるRunTimeが動作している。RunTimeの上には、移動エージェントと

管理モジュールがある。移動エージェントは複数存在することができるが、管理モジュールはRunTimeに対して１つである。各々の移動エージェントはそれぞれ１つのイベントモジュールを持っている。イベントモジュールは移動エージェントの実行環境を監視して、監視しているイベント（障害）を検出するとそれに対応する障害処理メソッドを呼び出す。管理モジュールは複数の移動エージェントを管理するモジュールである。管理モジュールは位置固定エージェント(Stationary Agent)で実現することができ、システム起動と同時に処理を開始するエージェントである。移動エージェントを管理することは、移動エージェントの協調性の性質を利用して実現することができる。以上の設計方針によって、移動エージェントシステムを組み込むことなしに実現することができる。図4.1は、システムの構成を示したものである。

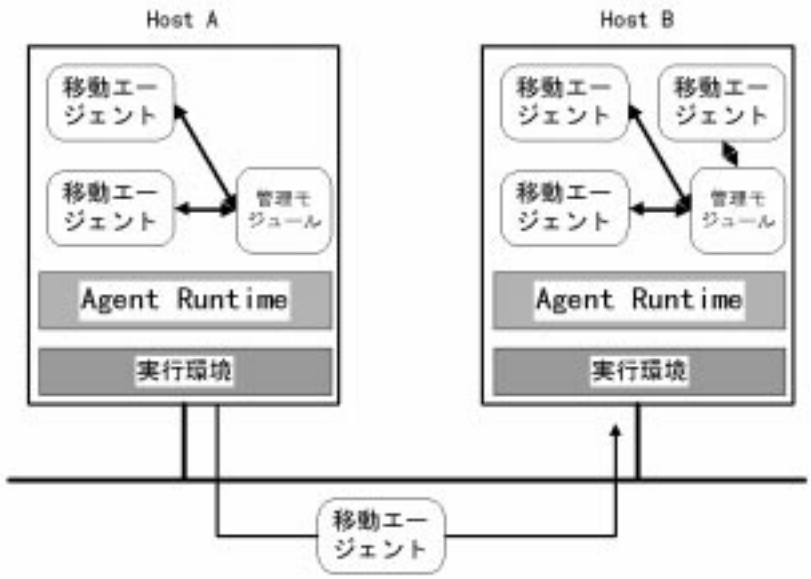

図 4.1: システム構成

4.2 障害対処機構

障害処理をするための障害対処機構は２つの処理部分から成っている。

- 障害処理メソッド：基本障害処理を実装するメソッド。
- イベントモジュール：計算機環境の障害を監視・検出し、障害処理メソッドを呼び出す。

障害をイベントとして検出し、電源の障害、計算機資源の不足、通信障害などを扱う。各々の移動エージェントがこの障害対処機構を備えている。

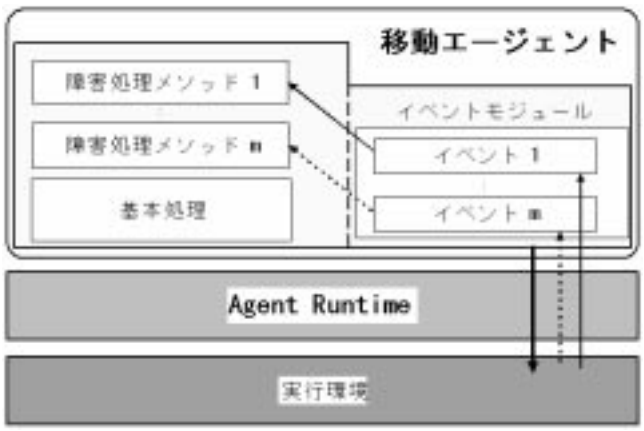

図 4.2: 障害対処機構

図 4.2 は本システムの障害対処機構のアーキテクチャを図示している。それぞれの移動エージェントは障害処理メソッド、イベントモジュール、基本処理を持っている。イベントモジュールは移動エージェントの実行環境を監視して、障害（イベント）が発生するとそのイベントに対応する障害処理メソッドを呼び出す。基本処理とは、本来移動エージェントが行う処理を記述する部分で、移動エージェントを作成するときには、最低限ユーザはこの処理部分を書くことになる。これらの障害対処機構は柔軟なカスタマイズが可能で、その利点として、

- 最低限の障害処理は障害対処機構で保証されている。
- 個々の移動エージェント特有の障害処理を実現するには、障害処理メソッドのオーバーライドやメソッドの追加、イベントモジュールのカスタマイズだけで済む。
- 障害処理をモジュールとして取り扱うことができる。

などの利点が挙げられる。これらの障害対処機構を Coop はライブラリとして提供する。

4.3 障害処理メソッド

障害処理メソッドは基本障害処理を記述する部分である。ユーザは障害処理メソッドが定義してあるクラスを継承した移動エージェントを作るだけで、システムが保証している

最低限の障害処理をする基本障害処理が実行される。また、ユーザ指定による固有の障害処理をしたい場合は、メソッドのオーバーライドすれば良い。また差分プログラミングによるカスタマイズが可能となっている。差分プログラミングによるカスタマイズとは、障害処理メソッドの処理の前後に処理を追加することである。障害処理メソッドは、１つの障害処理に対して３つのメソッド用意されている。

- before メソッド：before メソッドは障害処理メソッドの前に必ず呼ばれるメソッド。
- normal メソッド:障害処理の基本部分を記述する障害処理メソッド。
- after メソッド:after メソッドは障害処理メソッドの後に必ず呼ばれるメソッド。

差分プログラミングを行う場合は、ユーザが before・after メソッドに処理を記述すればよい。基本障害処理に加えて何か処理をしたい場合は、この２つのメソッドを用いることで、障害処理メソッドをオーバーライドし、障害処理を書き直すことなしに容易にカスタマイズできる。

4.3.1 障害処理メソッドのアーキテクチャ

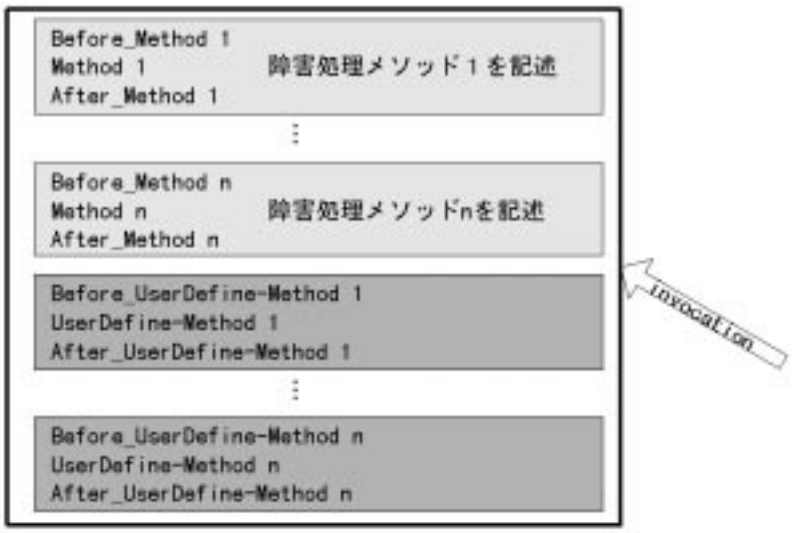

図 4.3: 障害処理メソッドの親クラス

障害処理メソッドはプログラマーがカスタマイズし易くかつ、カスタマイズの度に基本障害処理を再記述しないようにするため、設計はクラス継承を用いる。障害処理メソッ

ドを method、before_method、after_method とすると、親クラスは図 4.3 のようになる。method 1、. . . 、method n は、その障害に対する基本障害処理を記述する。その他のメソッドはカスタマイズ用に定義しているだけで、親クラスでは特に何も記述しない。これらを継承したクラスを使うことで、システムが障害に対して最低限保証することが可能となる。

障害処理メソッドのカスタマイズ方法は、継承したクラスのメソッドのオーバーライドを用いてカスタマイズすることになる。基本障害処理を定義してある method 対して、その処理が行われる前に、移動エージェントに仕事をしたい場合は、before_method のみを記述すればよい。同様に基本障害処理（method）の後に移動エージェントに仕事をさせたい場合は、after_method のみを記述すればよい。また、継承したメソッドに対して相応しくない動作や新しい障害処理を定義する場合は、ユーザ定義メソッドに記述する。図 4.4

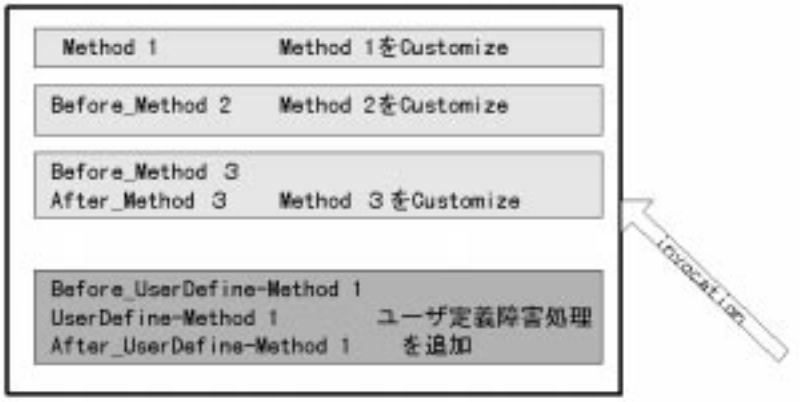

図 4.4: 障害処理メソッドのカスタマイズ

は、障害処理メソッド 1 の基本障害処理そのものをカスタマイズしている。Before_method 2 は障害処理メソッド 2 の基本障害処理の前に、Before_method 3 と After_method 3 は障害処理メソッド 3 の前後に他の処理をするように、カスタマイズしている。さらに、ユーザ定義障害処理メソッドを追加している例である。

カスタマイズをするにあたって注意しなければならないのは、method がその移動エージェントに対してどのような動作をするかを熟知しないとユーザが意図していない動きをする場合がある。このため、Coop を実装する場合は組み込む移動エージェントシステムの API を十分に理解しておく必要がある。

カスタマイズとして、次の節で説明する Battery メソッドをカスタマイズする。Battery メソッドはバッテリーが少なくなった時に移動する障害処理メソッドで、すぐに移動する

のではなく、移動前にレプリカを作成し二次記憶装置に永続化してから移動するようにしたいとする。この場合、before_Battery にレプリカを作成し永続化する記述をする。次に、移動後にバッテリー切れで移動してきたことを画面に表示したいとする。この場合には、after_Battery で前のホストでバッテリー切れが検出されたことを画面に表示するように記述する。以上で基本障害処理をカスタマイズすることができる。

4.3.2 障害処理メソッドの種類

表 4.1 では、幾つかの障害処理メソッドを紹介する。障害処理メソッドは実行中にイベント（障害）が発生する場合の他に、ホストに到着したときや生成したときに呼び出される場合がある。これは、障害は発生していないが障害処理に必要と考えられる処理を行うために障害処理メソッドを呼び出すことがある。CheckPoint メソッドとは、レプリカを用いて実行状態を保存するメソッドで、擬似的にチェックポインティングを行うことができる。イベントモジュールは実行環境に障害が発生したかどうかを監視するモジュールで、イベントモジュールで検知した障害が発生した時、その障害に対応した障害処理メソッドを呼び出す。そこで、如何に数多くの障害を予測するかで、本システムの信頼性が増す。しかし多くのイベントを監視することは、その分システムに負荷を与えることになり、パフォーマンスが低下することになる。このためイベントモジュールは、各移動エージェントで使用しないイベントに対して、監視しないようにカスタマイズすることが可能な設計にする。

4.4 イベントモジュール

イベントモジュールは障害をイベントとして検出するモジュール部分であり、検出されたイベントは、それに対応する障害処理メソッドを呼び出す。イベントを監視する部分はループしながら、常に監視を続ける。

4.4.1 イベントモジュールのアーキテクチャ

オーバーライドを用いてユーザ定義のイベントが追加・変更ができる柔軟性がある設計にするため、図 4.5 のように親クラスを設計する。 イベントモジュールのカスタマイズ方法は、親クラスのメソッドをオーバーライドすることで、実現することができる。カスタマイズは、既存のイベントをカスタマイズする場合と新しいイベントを追加する場合がある。既存のイベントをカスタマイズする時は、そのカスタマイズするメソッドをオー

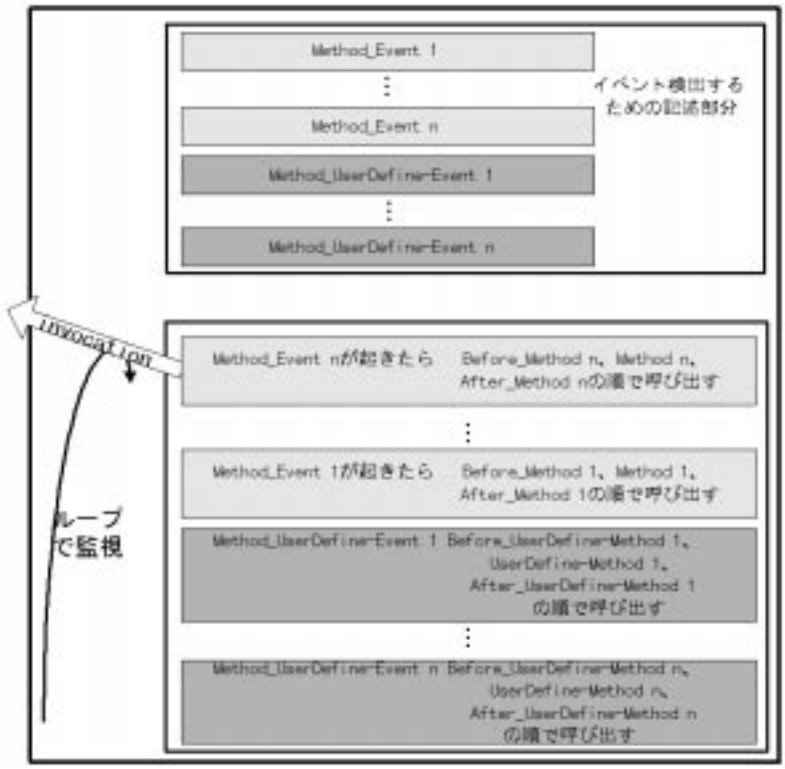

図 4.5: イベントモジュールの親クラス

バーライドする。また、新しくイベントを追加する時は、ユーザ定義イベントを記述することになる。実際に実装するに場合、これらのイベントを記述するメソッドの引数、戻り値は固定している。図 4.6 はイベント 1 をカスタマイズし、ユーザ定義イベントを 2 つ追

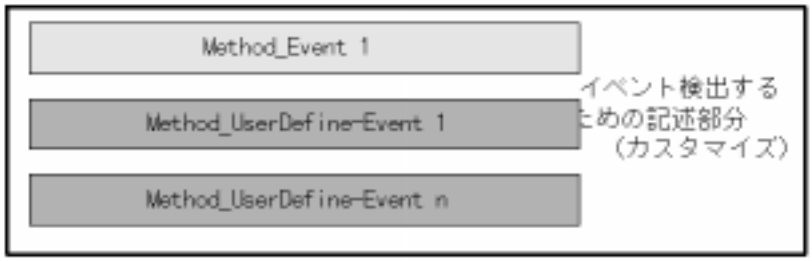

図 4.6: イベントモジュールのカスタマイズ

加している例である。そのため、イベントを追加するに時に必要と考えられるシステム情報が引数（オブジェクト）として与えられる。表 4.2 は、具体的なシステム情報を示している。新しいイベントを追加する場合、このシステム情報では情報が不足する場合が十分考えられる。イベント情報を追加したい場合には、システム情報クラスを継承したクラスに追加することで簡単に追加することができる。

4.4.2 イベントの種類

イベントモジュールで実際にイベントを検出を記述してある箇所は、図 4.5 の Method_Event の部分である。この部分に障害処理メソッドで定義した障害を検出するための処理を記述する。表 4.3 に、具体的なイベントを示す。これらのイベントを記述したメソッドは、図 4.5 のループで監視して、検出された時にその検出された障害に対処する障害処理メソッドを呼び出す。また、イベント種類のよってはイベント監視を開始してから一度しか監視をしない種類のイベントもある。

4.5 管理モジュール

実際に既存の移動エージェントシステムに障害対処機構だけを組み込むだけでは、使用に耐える移動エージェントシステムとしては機能が不足している。そのため、移動エージェントを管理する管理モジュールが必要となる。例えば、永続化された移動エージェントは当然、自分自身では活性化（二次記憶装置からの復活）できない。基本障害処理で提案した、ユーザが期待した時間にエージェントの処理結果を得られない場合の障害処理や移動中の通信障害に対応していない移動エージェントシステムに対して行う障害処理は、管理モジュールがこれを対処する。これらの管理モジュールは位置固定エージェントと移動エージェントの協調性の性質を用いることで実現することができる。管理モジュールの機能で必要な処理を幾つか挙げる。

- クラッシュ障害から回復したときに処理途中の移動エージェントを活性化する。
- 移動エージェントが設定時間内に処理結果を示さなかった場合の再送処理。
- 移動中の通信障害に対応。
- 分散アルゴリズムで使う ACK やタイマーの処理など。

この管理モジュールは、移動エージェントシステムを起動したときに一番初めに生成される位置固定エージェントである。管理モジュールは、最初に行う処理として二次記憶装置に記録してある移動エージェントの情報を取得する。この時、実行途中でホストダウンした移動エージェントがある場合には、その移動エージェントの実行状態を復元させる。当然、復元できる移動エージェントはチェックポインティングを行っていた場合に限る。Coop で行うチェックポインティングは、楽観的な考え方で行っている。楽観的な考えとは、ホストダウンしたホストがすぐに回復し、二次記憶装置に保存してある移動エージェントの実行状態が壊れていないことを想定している。Coop がチェックポインティングを行うタイミングを下記に示す。

- 生成時または到着時にチェックポインティングを行う。
- 一定間隔の時間、ホスト上で処理を行った時にチェックポインティングを行う。
- 移動する直前にチェックポインティングを行う。

これによりホストダウンが発生しても、回復時にチェックポイントした状態から仕事の続きを再開できる。

再送処理をする場合には、保存してあるレプリカを復活させて送ることになる。ユーザは、期待している時間内に移動エージェントの処理結果を得られなかった場合、以前に送ったホストに何らかの問題があると判断し、送り先を変更し再送することが可能である。また、移動先を変更するためのホストリストを変更する場合も管理モジュールが移動エージェントにホストリストを送る。

移動中の通信障害に対応していない移動エージェントの場合は、移動するときにレプリカを作成して、確実に移動先に到着したことが分かってから、管理モジュールは移動元のレプリカを消すことで対処する。

位置固定エージェントを用いる利点は、システムに組み込まないため、新しい分散アルゴリズムや管理モジュールの機能を追加する時には管理モジュールだけを変更するだけで済み、柔軟性のある移動エージェントシステムが作ることができる。

メソッド名	機能
TimeOut	各ホストで設定している実行時間が経過したときに、次のホストに移動するメソッド。
PhysicalMemory	実行環境の物理メモリが足りない場合に別のホストに移動するか、物理メモリが解放されるまで待つことになる。
VirtualMemory	実行環境の仮想メモリが足りない場合に別のホストに移動するか、仮想メモリが解放されるまで待つことになる。
DiskCapacity	現在のホストで使用しようとしている二時記憶装置の容量が不足したときに呼び出されるメソッドで、別のホストに移動する。
Cpu	現在のホストで使用しようとしているＣＰＵがエージェントの仕事に対して不都合の場合は、別のホストに移動する。(到着したときにチェックされる)
PowerSupply	ノートパソコンなどの電源が、バッテリーに切り替わったときに、実行されるメソッド。
Battery	バッテリーが指定された容量まで低下したときに、実行されるメソッドで別のホストに移動する。
ShutDown	ホストにシャットダウンを検出したときに、別のホストに移動する。
CheckPoint	チェックポイントを実行するメソッドで、一定時間が経過すると現在の実行状態を保存する。
UserDefine	システムモジュールのユーザ定義メソッドで定義した場合に呼び出されるメソッド。

表 4.1: 障害処理メソッド

変数名	内容
OS_name	OS の名前
OS_Version	OS
Power_Status	電源の状態
Battery_Status	バッテリーの状態
Resolution	ディスプレイの解像度
HD_Capacity	二次記憶装置の空き容量
Processor_Type	プロセッサーの種類
PhysicalMemory_Capacity	物理メモリの空き容量
VirtualMemory_Capacity	仮想メモリの空き容量
HostName	ホスト名
IPAddress	IP アドレス
UserName	ユーザ名
TimeZone	タイムの標準時間域
User_Language	ユーザの言語指定
Execution_Time	実行時間

表 4.2: システム情報

イベント名	内容
TimeOut_Event	設定している実行時間が経過したときに起きるイベント。
PhysicalMemory_Event	実行環境の物理メモリが足りないときに起きるイベント。（到着したときにのみ）
VirtualMemory_Event	実行環境の仮想メモリが足りないときに起きるイベント。
DiskCapacity_Event	二次記憶装置の容量が不足したときに起きるイベント。
Processor_Event	プロセッサーのタイプがエージェントの仕事に対して不都合の場合に起きるイベント。（到着したときのみ）
PowerSupply_Event	電源がバッテリーやUPS（無停電電源）に切り替わったときに起きるイベント。
Battery_Event	バッテリーが指定された容量まで低下したときに起きるイベント。
ShutDown_Event	ホストにシャットダウンを検出したときに起きるイベント。
CheckPoint_Event	一定時間が経過するとチェックポイントを実行するために起きるイベント。
UserDefine_Event	ユーザ定義イベントで定義したときに起きるイベント。

表 4.3: イベントの種類

第 5 章

実装

本章では、Java を用いて障害対処機構を作成する。実験するプラットフォームは AgentSpace を利用する。また、全てのソースコードを記することを割愛するため擬似コードで示す。ただし、AgentSpace の仕様を知ることなしに擬似コードを読むことはできないため AgentSpace の仕様を紹介してから、実装の説明をする。最後に例題として回覧板エージェントとシステム情報取得エージェントを説明する。

5.1 Java による障害対処機構の作成

まず、AgentSpace を用いて実装するために必要な説明をする。AgentSpace の構成図 5.1 となっており、ユーザが移動エージェントを作成する場合には、
/agentspace/system
上で作成しなければならない。また、移動エージェントを作成するにあたっては、
/agentspace/system/agentspace/Agent.java
を継承しなければならない。

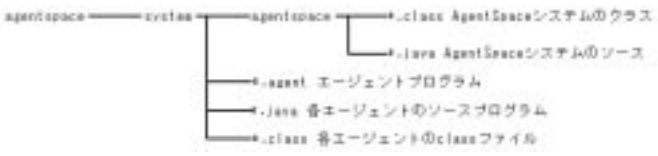

図 5.1: AgentSpace のクラス構成図

AgentSpace の API を紹介する。表 AgentSpace の API の状態とは、図 2.1 で示した、移

動エージェントの実行状態のことである。AgentSpace には、図 2.1 の実行状態に加えて初期化がある。

メソッド名	状態	機能
void init()	初期化	クラスファイルからエージェントを作成する時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。
void arrive()	到着	エージェントが、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	これはエージェントの複製が生成される直前に呼び出される。
void child(AgentIdentifier aid)	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。引数にはもう一方のエージェント (複製したエージェント) の識別子が入る。
void parent(AgentIdentifier aid)	複製	エージェントの複製が生成された際に、子エージェント側で呼び出される。

表 5.1: AgentSpace の API

AgentSpace は、ランタイムシステムとのインターフェースとしてエージェントコンテキストクラスを用いて、エージェント自身の移動、永続化、複製要求や他のエージェントに関する情報やエージェント間通信をする。エージェントコンテキストを取得するには、AgentContext getAgentContext();

で、取得できる。

メソッド名	機能
getIdentifile	自分自身のエージェント識別子を取得する。
getAgents	ランタイムシステム上のすべてのエージェント識別子を取得する。
getAgentName	エージェント名を取得する。
getCurrentHost	現在のホストのアドレスを取得する。
getPreviousHost	前のホストのアドレスを取得する。

表 5.2: エージェントコンテキストの API

エージェントは、エージェントコンテキストのメソッドを通じて自分自身の状態も変更することができる。また、取得したエージェント識別子を使ってエージェント間通信をす

メソッド名	機能
dispatch	引数のアドレスに移動する。
suspend	引数の名前で二次記憶へ永続化する。
duplicate	レプリカを作成する。
create	引数のエージェント名を二次記憶から活性化する。

表 5.3: エージェントコンテキストの API

ることができる。エージェント間通信は、send（非同期メッセージ送信）、call（同期メソッド呼び出し）、future（非同期メソッド呼び出し）の３つの API が用意されている。

初めに障害処理メソッドを AgentSpace を利用して実装した場合の記述例を示す。移動エージェントシステムに手を加えないために、Agent クラスを継承した FtAgent クラスに障害処理メソッドを記述する。この障害処理メソッドは、コールバックメソッドとして、イベントモジュールから呼び出される。Coop のフレームワークを利用して移動エージェントを作成する場合には、FtAgent を継承したクラスを用いることになる。

```
import agentspace.*;
class FtAgent extends Agent{
 //障害処理メソッド タイムアウトメソッド
 public void Before_TimeOut(SystemInfo sysinfo){}
 public void TimeOut(SystemInfo sysinfo) {
    ⋮
 dispatch(URL);
 }
 public void After_TimeOut(SystemInfo sysinfo) {}
   //障害処理メソッド
 public void Before_Method(SystemInfo sysinfo)
 public void Method(SystemInfo sysinfo) {
               /* 障害処理を記述 */
 }
 public void After_Method(SystemInfo sysinfo){}
 public void Before_UserDefine-Method(SystemInfo sysinfo){}
 public void UserDefine-Method(SystemInfo sysinfo) {}
 public void After_UserDefine-Method(SystemInfo sysinfo){}
    ⋮
```

イベントモジュールの親クラスである EventModuleBase クラスを作成する。コンストラクタで FtAgent を引数とすることで、FtAgent のメソッド（障害処理メソッド）を呼び出すことができる。イベントが検出されると、

ftagent.Before_Method();

ftagent.Method();

ftagent.After_Method();

の順に必ずメソッドが呼び出されるように、実際作成する場合はアドホックに実装する。これによって、ftagent.Method() が移動するようなメソッドであったとしても、移動先では必ず始めに ftagent.After_Method() が呼び出される。また、実際はユーザ定義イベントが定義していないときにはユーザ定義イベントが呼び出されないように実装する。

```
class EventModuleBase extends Thread implements Serializable {
 FtAgent ftagent;
 SystemInfo systeminfo; //システム情報
 EventModule(FtAgent ft) {        //コンストラクタ
  ftagent=ft;
 }
```

```
public boolean TimeOut(SystemInfo systeminfo){
 if(systeminfo.Execution_Time > 100000)
  return true;
 else
  return false;
}
public boolean Method_Event(systeminfo){
  /* イベント検出コードを記述 */
}
public boolean Method_UserDefineEvent(systeminfo){
 return false
}
      ⋮
public void run() {
 systeminfo = new SystemInfo();
 for(;;){ //ループで監視
   systeminfo.getSystemInfo(); //新しいシステム情報
   //タイムアウトを検知
  if(TimeOut(systeminfo)){
   ftagent.Before_TimeOut();      //障害処理メソッド
   ftagent.TimeOut();
   ftagent.After_TimeOut();
   this.suspend();
  }
  if(Method_Event(systeminfo)){
  ftagent.Before_Method();
  ftagent.Method();
  ftagent.After_Method();
  this.suspend();
  }
  if(Method_UserDefineEvent(systeminfo)){
  ftagent.Before_UserDefine-Method();
  ftagent.UserDefine-Method();
  ftagent.After_UserDefine-Method();
  this.suspend();
  }
      ⋮
 }
}
}
```

例として移動エージェントである TestAgent を作成する。TestAgent クラスは FtAgent クラスを継承して作成される。TimeOut メソッドの Before メソッドをカスタマイズし、さらにユーザ定義障害処理メソッドを作成して、障害処理をカスタマイズする場合を次

に示す。自分自身のスレッドを引数にイベントモジュールを作成して、イベントモジュールを開始する。イベントモジュールが開始されて、初めてイベントが監視されることになる。イベントモジュールは、生成時や処理を再開したときにイベントモジュールを作成し直さなければならない。また移動するときなどは、生成したイベントモジュールのスレッドを停止しなければならない。実際は、イベントモジュールのスレッド生成はFtAgentに記述している。イベントモジュールが開始されるタイミングは、表5.5、5.6のAIPが呼ばれる前で、スレッドの作成・開始・削除をする。よって移動エージェントを作成するプログラマーはイベントモジュールのスレッド生成・停止などは意識する必要はない。

```
class TestAgent extends FtAgent{
 Thread event;
 //障害処理メソッドのオーバーライド
 public void Before_TimeOut(); {
    /* 障害処理 */
        ⋮
 }
 //ユーザ定義障害処理の作成
 public void Before_UserDefine-Method(SystemInfo sysinfo){}
   /* 障害処理 */
        ⋮
 }
 public void UserDefine-Method(SystemInfo sysinfo) {
    /* 障害処理 */
        ⋮
 }
 public void create() {  //生成時にエージェントモジュールを作成
  event = new EventModule(this);
  event.start();
  /* エージェントの基本処理 */
        ⋮
  }
 public void arrive() {  //到着時にエージェントモジュールを作成
  event = new EventModule(this);
  event.start();
  /* エージェントの基本処理 */
        ⋮
 }
 public void dispatch(URL url) {  //移動時にイベントモジュールと止める
 event.stop();
 }

        ⋮
}
```

イベントモジュールは、EventModuleBase クラスを継承して EbentModule クラスで作成する。よってイベントモジュールをカスタマイズする場合は、EbentModule クラスに記述することになる。ここでは、例として新しいイベントであるユーザ定義イベントを追加する。

```
class EventModule extends EventModuleBase {
//イベントモジュールのカスタマイズ
 public boolean Method_UserDefine-Event(SystemInfo systeminfo){
   /* ユーザ定義イベントを記述 */
 }
}
```

FtAgent.java、EventModuleBase.java、TestAgent.java、EventModule.java の４つのファイルを説明した。移動エージェントを作成する場合、プログラマーは FtAgent.java、EventModuleBase.java を継承したクラスを利用して移動エージェントを作成する。最低限の障害処理を保証している基本障害処理のみで移動エージェントを作成する場合には TestAgent.java のみで作成すればよい。

5.2 例題

この節では、Coop を実装した AgentSpace を用いて例題を作成する。例題として回覧板エージェントとシステム情報取得エージェントを作成した。本章では回覧板エージェントを重点的に説明し、擬似コード及び実行例を示す。システム取得情報は、仕様と動作画面のみ示す。

5.2.1 回覧板エージェント

回覧板エージェントの仕様を下記に示す。

- 予め用意されたホストリストを利用して巡回する。
- 各ホストの画面上で連絡事項を表示する。
- ユーザは回覧板にコメントを入力し、次のホストに回覧板を送る。
- コメントを入力して送られた回覧板には、印が押される。
- ユーザが不在の時は、一定時間時間が経過すると、自動的に次のホストに移動する。
- ユーザ不在や障害発生したホストには、回覧板の欄チェックされる。

- 分散アルゴリズムを用いた、クラッシュ障害に対応。

次に、移動エージェントを用いて回覧板エージェントを作成する利点や障害対処機構を備えた場合の利点を述べる。

- 現実世界の回覧板とは異なり、不在者によって回覧版が止まることはない。
- 回覧板エージェントは、コメントなどのデータの保持が重要視される。
- メールを仮想的に回覧板で実現する場合との比較、
  - ブロードキャスト方式の場合は、余計なゴミメールが増える。
  - リレー方式では、ユーザ不在の場合、回覧板がその人のところで止まる。

などが挙げられる。これらの利点から、回覧板を移動エージェントで実現することは十分利点がある。また障害によって回覧板エージェントのデータが消えないために、Coop を用いることも十分利点がある。

次に、回覧板エージェントのクラスである FTKairan.java のコードを示す。

```
import agentspace.*;
import java.awt.*;
import java.awt.event.*;
class FTKairan extends FtAgent implements ActionListener ,ItemListener,WindowListener{
 private Button button1;  //クラス変数
 private TextArea text;
       ⋮
 public void FTcreate() { //エージェント生成時に基本処理をする
  AgentContext ac = getAgentContext();
  AgentIdentifier aid = ac.getIdentifier();
  initWindow(aid);
  show();
 }
 private void initWindow(AgentIdentifier aid){ //回覧板エージェントの基本処理コード
  setTitle("回覧板");
  text = new TextArea("忘年会についての案内です。… ");
  text.setEditable(false);
       ⋮
  setSize(400,300);
  addWindowListener(this);
  pack();
 }
 public void windowOpened(WindowEvent e) {}
 public void windowActivated(WindowEvent e) {}
 public void windowDeactivated(WindowEvent e) {}
```

```
 public void windowClosed(WindowEvent e) {}
 public void windowIconified(WindowEvent e){}
 public void windowDeiconified(WindowEvent e){}
 public void windowClosing(WindowEvent e){ //エージェントを消す
  AgentContext ac = getAgentContext();
  ac.send("destroy");
  dispose();
 }
 public void  actionPerformed (ActionEvent e){
       :
 }
 public void FTinit(){ //１度だけ実行される（初期化）
  Set_distributedAlgorithm();   //分散アルゴリズムを使う
  SetURLlist("150.65.193.48","150.65.193.34","150.65.193.48"); //ホストリストのセット
 }
 public void FTarrive() { //到着時にアプレットを表示
  show();
 }
 public void FTdispatch(URL url) { //移動前にアプレットを消す
  dispose();
 }
 public void FTresume() {
  show();
 }
 public void FTsuspend() {
  dispose();
 }
 public void FTdestroy() {
  dispose();
 }
}
```

実際の回覧板エージェントは図 5.2 の様になる。

図 5.2 は、酒宴案内の例である。回覧板は、幹事、ken1、ken2、ken3 の４人のいるホストを巡回をする。幹事は、タイムアウトを設定することができる。タイムアウトとは各ホストに到着後、指定した時間内にユーザーが回覧板に答えないとユーザ不在ということにして、自動に次の人に回覧板を送る時間である。このタイムアウトの設定はデフォルト”40”となっているテキストエリアに任意の時間を入力することができ、Set ボタンを押すことでタイムアウトを設定することができる。また、タイムアウトが発生した場合には判子の欄は、 **”不在 ”**とチェックされる。

回覧板を受け取るユーザは、参加・不参加のラジオボタンを押して参加の意志を伝える。また、コメントを入力できるテキストエリアがあるので一言コメントを入力することがで

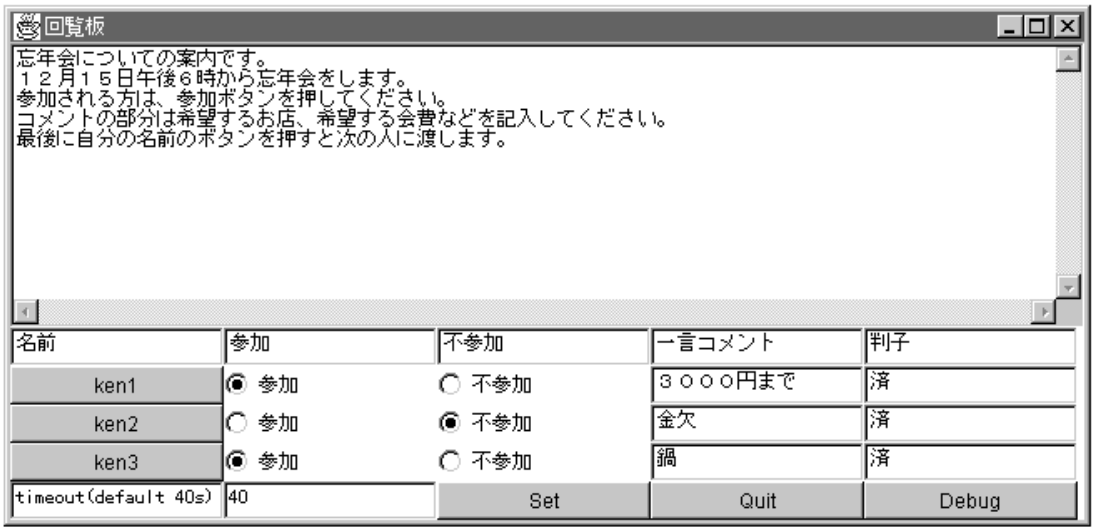

図 5.2: 回覧板エージェント

きる。ユーザは答え終わったら自分の名前のボタンを押すと、判子の欄に **”済 ”**がチェックされ、次の人に回覧板が送られる。当然、各ユーザは自分意外のラジオボタンやテキストエリアの入力することはできない。巡回が終わると、幹事に回覧板が戻ってくる。幹事は判子の欄を見ることでユーザが回覧板を受け取ったかどうかが判断することができて、判子の欄が **”障害 ”**の場合はそのホストで何らかの障害が発生したことが分かる。よって幹事は、判子の欄を見て再送するかどうかを判断し、不在者となったホストや障害が発生したと思われるホストにだけ再送することもできる。この例題は、本研究で提案した分散アルゴリズムを使用している。それ故、エージェントの矛盾を防ぐため、すでに処理し記入された回覧板が消される可能性はある。レプリカを管理している管理タイマーは、タイムアウト時間の３倍の時間を設定し、移動の時間を含めた時間を十分取っている。管理タイマーの時間が短いと、ユーザが増えたときに移動時間を考慮に入れないと、一番仕事をしている回覧板が消去される可能性があるため、この時間設定には注意する必要があると考えている。

回覧板エージェントは基本障害処理よって移動しなければならない場合には、ホストリストから次のホストへと移動する。幹事のホストでこの様な障害が発生した場合には、移動しないで二次記憶装置へ永続化される。この時に、二次記憶装置の空き容量が少なかった場合には、信頼性の高いサーバの様なホストへと移動することになる。実際に移動する場合は、信頼性の高いホストリストからホスト名を取得して移動することになる。しかし、バッテリー切れなど、緊急に移動しなければならないときは、ネットワークのタイム

アウト時間が致命的な時間なりかけない。一般に、図5.3の様なホストがネットワークに接続されていない場合やエージェントサーバが立ち上がっていない場合に移動命令が実行されると、ネットワークのタイムアウトが発生しない限り移動可能かどうかは分からない。殆どの移動エージェントシステムは、このネットワークのタイムアウトが発生すると例外が発生し、例外処理をによって別のホストに移動できる。しかし、タイムアウト時間は比較的長く設定されている場合が多いので、緊急移動の時に、このようなホストへ移動する場合には、この時間は致命的となる。そこで、Pingを用いて、ネットワークのタイムアウト時間が長い場合に対処している。返答があるときは、ネットワークに接続されてかつエージェントサーバが動作しているホストである。Pingの返答時間は、Lan環境なら１～３秒の設定で十分な時間といえるだろう。

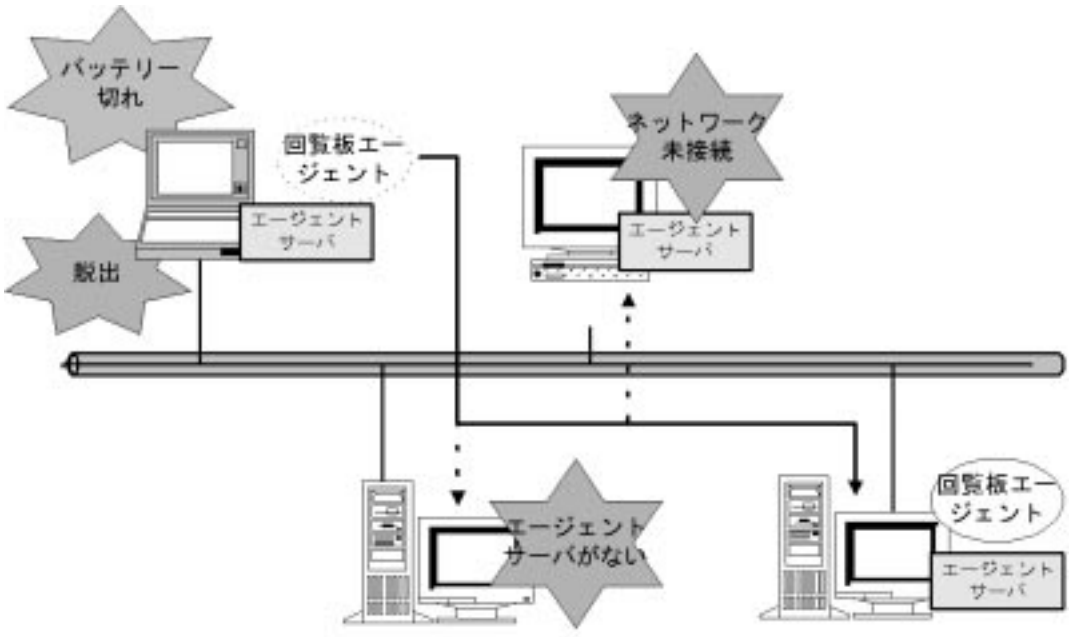

図 5.3: 緊急移動

5.2.2 システム情報取得エージェント

システム情報取得エージェントは単純な移動エージェントのため、細かい説明は割愛する。例えばシステム管理者は誰がどのようなOSを使って、HDの空き容量やJDKのバージョンなど計算機環境がどの様になっているのかを調べる必要があると考えられる。このような情報をわざわざシステム管理者が調べるよりは移動エージェントに任せればよい。システム情報取得エージェントの仕様を下記に示す。

- ホストリストを用いて巡回する。
- 各ホストで、システム情報を取得する。
- システム情報取得後、次のホストへ移動する。
- システム情報が取得できたホストには、チェック欄にチェックされる。
- ホストを巡り終えると、システム管理者（エージェントオーナー）に戻ってくる。
- 障害処理は基本障害処理のみ対応。

開始直後は、ホストリストのホスト名しか情報はない。チェックが **”済 ”**となってるホストは、詳細ボタンを押すと図5.5が画面に表示される。

図 5.4: システム情報取得エージェント

5.2.3 API

この節では、AgentSpace上でCoopのフレームワークを利用して障害処理をするために必要と思われるAPIを作成したので、それを紹介する。表5.4は、障害処理を記述するプログラマの負荷を軽減するために予め作成したメソッドである。基本障害処理で移動す

図 5.5: システム情報の例

るホストは、SetURLlist で登録されたホストに移動する。登録しないと基本障害処理で移動するホストはデフォルトのホストへ移動する。EmergencyMove は、回覧板エージェントの例で説明した緊急移動用のメソッドである。予め SetServerlist で緊急移動先ホストを登録しておかなければならない。登録しないとデフォルトのホストへ移動する。チェックポイントを擬似的に行うために作成するレプリカなどの情報は、システム管理モジュールに登録しなければならない。移動中の通信障害に対応していない移動エージェントシステムの場合は、レプリカを作成して移動エージェントを保存しておき、移動先に確実に到着した時に、レプリカを消すことになる。

表 5.5,5.6 は、AgentSpace のコールバックメソッドに相当するので、Coop のフレームワークに基づいた AgentSpace の移動エージェントを作成する場合には、少なくともこの API を把握しないと移動エージェントは作成することはできない。

メソッド名	機能
void SetURLlist(URL url)	引数のアドレスをホストリストに追加する。
void SetServerlist(URL url)	引数のアドレスを緊急移動用ホストリストに追加する。
void FTMove(URL url)	レプリカを作成し二次記憶装置に永続化し、その後本体は引数の URL へ移動する。
void EmergencyMove()	緊急移動用ホストに移動する。
void Clone()	レプリカを作成して、二次記憶装置に保存する。
void CheckPoint-Start()	主に到着直後にチェックポイントを行い、システム管理モジュールに登録する。
void CheckPoint-Current()	実行中にチェックポイントを行い、システム管理モジュールに登録する。
void CheckPoint-end()	現在のホストで仕事を終えたときに、チェックポイントを行い、システム管理モジュールに登録する。
boolean Ping(URL url)	引数のホストにエージェントシステムが立ち上がってるかどうか調べる。
void Set_distributedAlgorithm()	分散アルゴリズムを使う場合は、初期化時にこのメソッドを呼び出す。指定されたリストに URL を入力する必要がある。
void Set_redispatch()	移動途中の通信障害に対応させる。

表 5.4: 追加した API

メソッド名	状態	機能
void FTinit()	初期化	クラスファイルからエージェントを作成する時に一度だけ呼び出される。AgentSpace の init() に相当する。
void FTcreate()	生成	クラスファイルからエージェントを作成する時に一度だけ呼び出される。AgentSpace の create() に相当する。
void FTdestroy()	終了	エージェントが、消滅する直前に呼び出される。AgentSpace の destroy() に相当する。
void FTdispatch(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。AgentSpace の dispatch(URL url) に相当する。
void FTarrive()	到着	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。AgentSpace の arrive() に相当する。
void FTsuspend()	永続化	エージェントが、移動先のコンピュータに到着した直後に呼び出される。AgentSpace の suspend() に相当する。

表 5.5: 追加した API2

メソッド名	状態	機能
void FTresume()	複製	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。AgentSpace の resume() に相当する。
void FTduplicate()	複製	これはエージェントの複製が生成される直前に呼び出される。AgentSpace の duplicate() に相当する。
void FTchild(AgentIdentifier aid)	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。引数にはもう一方のエージェント (複製したエージェント) の識別子が入る。AgentSpace の child(AgentIdentifier aid) に相当する。
void FTparent(AgentIdentifier aid)	複製	エージェントの複製が生成された際に、子エージェント側で呼び出される。AgentSpace の parent(AgentIdentifier aid) に相当する。

表 5.6: 追加した API3

第 6 章

議論

本章では、本研究で提案した Coop と関連研究を比較、考察する。最後に Coop の問題点について述べる。

6.1 関連研究との比較

関連研究の中で、実際にこのような障害対処機構を持って移動し、移動エージェント自身が障害に対処する移動エージェントシステムは他には見ることはできない。FT を考慮したシステムの多くは、クラッシュ障害だけを重点に考えている。それらのシステムは、チェックポインティングを用いて障害が発生したら一番新しいチェックポインティングした実行状態を用いて、移動エージェントを復活させる方法をとっており、本研究で提案したような移動エージェント自身が様々な障害に対処するようなことはしていない。

本研究では、チェックポインティング処理を持っていない移動エージェントシステムでも、擬似的なチェックポインティングと分散アルゴリズムを用いることで代用することができた。チェックポインティングを利用して障害に対処できるシステムとしては、Tacoma、Voyager、Concordia[19] がある。ホスト又はノードがクラッシュしたときに、これらのシステムは、最後にチェックポインティングした状態から再開することができる。チェックポインティングをサポートしていない移動エージェントシステムに対しては、擬似チェックポインティングと本研究で提案した分散アルゴリズム用いれば良いだろう。ただし本研究で提案した分散アルゴリズムは単純なアルゴリズムのため、十分な考察や改良をする必要があると考えている。完成度の高いチェックポイントを備えた移動エージェントシステムに対しては、障害対処機構や管理モジュールのみを組み込むだけで、そのシステムを更に信頼性を高めることができる。本研究は、他の関連研究と比較し優劣をつけるのではな

く、優秀な移動エージェントシステムがCoopを備えることで、より良い信頼性の高い移動エージェントシステムを作成することが本研究の目的である。

本研究の利点は

- Coopのアーキテクチャは移動エージェントシステムのプラットフォームに依存しない。
- 障害処理と基本処理部分が独立しているため、障害処理をモジュールとして取り扱うことができる。
- 移動エージェント自身が障害対処機構を持つことで、
  - 障害に対処していない移動エージェントシステム上では、移動エージェント自身で障害に対処することができる。
  - 障害対処機構をカスタマイズするたびに、各ホストに新しいシステムをインストールする必要はない。
- 障害処理メソッド・イベントモジュールのカスタマイズが可能なアーキテクチャ。

などがを挙げられる。

ビザンチン障害を扱うことは非常に難しいが、非ビザンチン的ソフトウェア障害に関しては本研究でも扱わなければならない障害だと考えているので、今後の課題で述べる。

6.2 Coopの問題点

Coopの問題点としては、

- 移動エージェント自身が障害対処機構を持つことで、移動エージェントの実行コードが大きくなり通信量が増えてしまう。
- イベント監視の負荷。

が挙げられる。まず、移動エージェントのコードが大きくなる問題は、AgentSpaceで実装した障害対処機構は、移動エージェントに組み込むと約20kbyte程度、実行コードが増えてしまう。この20kbyteの大きさは通信環境に大きく左右されると考えているが、Lan環境の場合は特に重要視する程の大きさではないと考えている。

イベント監視の負荷は、移動エージェントの仕事の目的によると考えている。移動エージェントに科学技術計算をさせる必要性をここでは論じないが、移動エージェントに実行

速度が要求される場合、このイベント監視は見過ごせない負荷である。この負荷を軽減する方法の１つとしてイベント監視の sleep 時間を増やすことで、処理速度の向上が考えられる。しかし、イベント監視をしない時間が長いと障害の検出が遅れ、障害に対処できない可能性があるため、極端に sleep 時間を増やすのは障害対処機構を備える意味がなくなってしまう。sleep 時間の設定は、実験をしていないため詳しくは述べられないが、障害が発生してから移動するまでの時間や二次記憶装置に永続化する時間を考慮に入れると、極端に短い間隔で監視をする必要はないと考えている。また、各々の移動エージェントがそれぞれイベントを監視することも問題である。このイベント監視をシステムに組み込むことで処理速度は向上するが、イベント追加のカスタマイズが行い難く、各々の移動エージェント固有のイベントのカスタマイズも難しくなってしまう。イベント追加の度に移動エージェントシステムを再コンパイルし、すべてのホストに移動エージェントシステムを新しいシステムと交換しなければならない。よってこの問題は、システムの柔軟性とのトレードオフと考えている。

本研究の分散アルゴリズムは、単純なアルゴリズムで十分考慮されたアルゴリズムとは言い難い。分散アルゴリズムは改良・変更する必要があり、今後の課題でもある。また、実装して分散アルゴリズムは、容易に切り離すことができるように実装してある。

第 7 章

結論

本章では、論研究の結論を述べ、最後に今後の課題について述べる。

7.1 まとめ

本論文では、信頼性の高い移動エージェントシステムの構成方法として、Coop を提案した。Coop のアーキテクチャは a) 障害対処機構と b) 管理モジュールの 2 つから構成されている。障害対処機構とは、物理的な障害に対処できる機構で、移動エージェント自身が障害対処機構を持って移動する。それ故、移動エージェントシステムのプラットフォームに依存しない。これにより、移動エージェントシステムへの移植性が高くなっている。また、障害対処機構は容易にカスタマイズが可能なアーキテクチャであるため、柔軟な障害処理を行うことができる。さらに、障害処理と基本処理が独立しているため、これらの処理をモジュール単位で扱うことができることも利点である。障害対処機構で対処できない処理に関しては、管理モジュールで処理することを提案した。またクラッシュ障害に対処できる巡回移動エージェントを作成するために単純な分散アルゴリズムを提案した。

実験プラットフォームは、Java ベースの移動エージェントシステムである AgentSpace を用いて Coop を実装した。例題は分散アルゴリズムを使った回覧板エージェントと基本障害処理のみのシステム取得エージェントの 2 つを作成した。また、障害処理に必要と思われる API を幾つか用意し作成した。

7.2 今後の課題

今後の課題としては

- 障害処理・イベントの種類を増やす。
- 非ビザンチン的ソフトウェアの障害に対処。
- 分散アルゴリズムの考察。

を考えている。まず障害処理やイベントの種類を増やすためには、多くの障害を想定しなければならない。また、数多くの移動エージェントを作成し使用することで、予期していなかった障害が発見されると考えられる。障害対処機構はカスタマイズが容易なため、新しい障害が発見された場合でも、ユーザ定義障害処理メソッドとユーザ定義イベントを用いることでそれに対処できると考えている。

非ビザンチン的ソフトウェア障害については、本研究でも取り扱わなければならない障害だと考えている。移動エージェントが操作できなくなるような致命的な例外処理に関しては対処しているシステムは少ない。一般にこのような例外が発生した場合、エージェントサーバがこれに対処し、例外が発生した移動エージェントに対して適切な処理をしなければならない。適切な処理として古典的な方法は、エージェントサーバーは例外が発生した移動エージェントのオーナーに、移動エージェントに例外が発生して途中で処理が中断してしまったことを知らせる。これによって、移動エージェントのオーナーがその移動エージェントの処理を取り消したり、回収したりしなければならない。これとは異なる処理方法として２章で紹介したJavaベースの移動エージェントシステムであるAjantaがある。Ajantaは例外が発生した場合、その移動エージェントのオーナーに対して、例外が発生した移動エージェントとその状態を返す。オーナはこれらの情報を用いて、何故、例外が発生したのかを調べることができ、適切な状態に直すことができたら、仕事の続きを再開するという方法で例外処理を行っている。例外処理に関する研究動向についても注意したい。

本システムの問題点でも述べたが、特に本研究の分散アルゴリズムは十分考慮する必要がある。分散アルゴリズムをカスタマイズしやすアーキテクチャにすることで、アルゴリズムの改良や変更ができればよいと考えている。また、関連研究や既存の分散アルゴリズムを参考にし、完成度が高く使いやすい移動エージェントシステムに組み込むことで、より完成された信頼性の高い移動エージェントシステムを作成することを目標としたい。

謝辞

渡部卓雄助教授には本研究について多大な御指導と御助言をして頂きました。ここに感謝の意を表し、心より御礼申し上げます。二木厚吉教授にはよく御指導して頂きました。心より感謝致します。緒方和博助手には貴重な助言をして頂きました。深く感謝致します。天野憲樹氏には、御指導と御助言をして頂きました。深く感謝致します。言語設計学講座のみなさまには、議論に乗っていただきました。深く御礼申し上げます。佐藤一郎助教授には、快く AgentSpace を利用することを許可していただきました。深く感謝致します。

参考文献

[1] Danny,B.L.,Mitsuru,O. and Kazuya,K.:Aglets Programming Mobile Agents in Java. LNCS 1274, pages 253-266, Springer-Verlag. 1997.

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

[3] 前川守著.:岩波講座ソフトウェア科学７ソフトウェア実行／開発環境, 岩波書店,1992.

[4] 岩井俊弥著.;Java モバイルエージェント, ソフト・リサーチ・センター,1998.

[5] White,J.E.:Telescript technology,the foundation for the electronic marketplace.White Paper.Generan Magic,Inc.1994.

[6] http://www.genmagic.com/technology/oddyssey.html

[7] 小野沢 博文.:分散オブジェクト技術 CORBA, ソフト・リサーチ・センター 1996.

[8] http://www.microsoft.com

[9] 山崎重一郎, 津田宏編訳,:Telesciript 言語入門, アスキー出版,1996.

[10] Robert O,Dan H,Jeri E.:The Essential Distributed Objects Survival Guide,WILEY,1996.

[11] http://www.trl.ibm.co.jp/aglets/atp/atp.htm

[12] http://www.omg.org/

[13] 佐藤一郎.:AgentSpace 高階モーバイルエージェントシステム, 電子情報通信学会技術研究報告,Vol97,No627,pages41-48, 電子情報通信学会,1998.

[14] 藤田悟,.:移動・分散プログラミング言語 Mobidget, 情報処理学会題 57 回全国大会, 講演論文集 (1),pages301-302, 情報処理学会,1998.

[15] http://www.objectspace.com/voyager/

[16] Johansen,D.,van Renesse,R. and Schneider,F.B.:Operating System Support for Mobile Agents,IEEE 5th Workshop on Hot Topics in Operating System(HOTOS-V),pages42-45,1995.

[17] Fred,B.S.:Towards Fault-tolerant and Secure Agentry,11th International Workshop on Distributed Algorithms,1997.

[18] Anand,T and Neeran,K.:Protected Resource Access for Mobile Agent-based Distributed Computing,In Proceedings of the ICPP workshop onWireless Networking and Mobile Computing, Minneapolis,1998.

[19] http://wwww.meitca.com/HSL/Projects/Concordia

第 A 章

付録

A.1 AgentSpace

AgentSpace を使って、本研究で提案した障害対処機構を備えた移動エージェントを作成し、動かすための簡単な説明をする。AgentSpace の使用法は、AgentSpace をインストールしたときに作成される、HTML 形式のドキュメントを参照して欲しい。

A.2 動作環境

実装したプログラムは、Java 上で実装しているので、理論上は Java の VM 上なら、OS に依存することなしに動作する。しかし、イベントモジュールでネィティブメソッドを使用しているために、Win32API が動く OS に限定されている。他の OS に対応させるためには、その OS のネィティブメソッドを実装する必要がある。動作環境は、Windows95、Windows98、WindowsNT 4.0 の Java 言語 (JDK1.1 以上) 上のシステムで動作する。実装に使った Java のバージョンは JDK1.1.6 である。また、AgentSpace のバージョンは 1998 年 3 月 3 日版を使用した。システム管理モジュールがシステム起動時に自動的に立ち上がるように、多少 AgentSpace のシステムに 2,3 行手を加えている。

A.3 インストール方法

Java を手に入れてない場合は
http://sunsite.sut.ac.jp/java/
から JDK1.1 以上を入手する必要がある。Java2 に関しては動作テストをしていない。イ

ンストール方法は、
http://133.65.66.181/agent/download.html
からダウンロードした1998年3月3日版をインストールマニュアルにしたがってAgentSpace
をインストールした後、
http://www.jaist.ac.jp/k̃en2/FtAgentSpace.zip
をダウンロードする。ファイルを解凍後、
/agentspace/system/*.*
/agentspace/system/FTClone
/agentspace/system/agentspace/*.*
のファイル及び、ディレクトリをAgentSpaceにコピーする。

A.4 移動エージェント作成方法

ここでは、回覧板エージェントを例に、移動エージェントの作成方法を説明する。回覧板エージェントはFTKairan.javaにすべて記述されている。まず、javac FTKairan.javaで、FTKairan.classファイルを作る。次に、AgentSpaceのMakeAgent.classを使って、移動エージェントを作成する。引数として
java MakeAgent 主クラス名 エージェント名 必要クラスファイル名(s)
となっている。必要クラスファイル名は、
EventModule.class EventModuleBase.class FtAgent.class FTClone.class FTInfo.class
EventInfo.class PingCheck1.class PingCheck2.class
である。よってエージェントを作成する場合には
java MakeAgent FTKairan FTKairan.agent FTKairan.class EventModule.class
EventModuleBase.class FtAgent.class FTClone.class FTInfo.class EventInfo.class
PingCheck1.class PingCheck2.class
とすれば、FTKairan.agentエージェントが作成できる。

A.5 実行方法

./agentspace/systemのディレクトリ上で、
java agentspace/AgentServer
と入力すると、AgentSpaceが立ち上がる。エージェントモニターが起動し、さらにシステム管理モジュールA.1が自動的に起動する。システム管理モジュールが起動することを

除けばAgentSpaceとほぼ同様に動かすことができる。

図A.1は３つの移動エージェントを管理している。beginとなってる移動エージェントは、生成されたか、現在のホストに到着した状態である。currentとなっている移動エージェントは、現在のホストで実行している移動エージェントで、擬似チェックポイントが行われて、レプリカが保存されている。endとなってるエージェントは、すでに現在のホストで仕事を終えて、別のホストに移動した状態で、そのレプリカは二次記憶装置に永続化されている。現在のホストがクラッシュ障害などから回復したときに、beginとcurrentが残っていたときは、移動エージェントが現在のホストで仕事をしているときにクラッシュ障害が発生したことが分かり、保存してあるレプリカを用いて仕事が再開することができる。

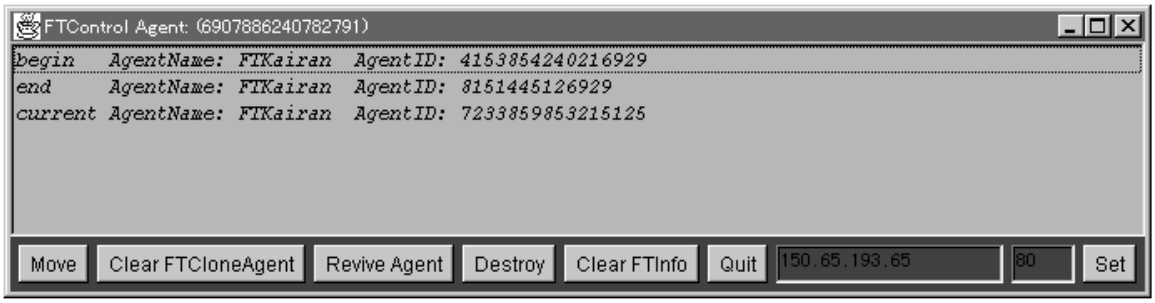

図 A.1: システム管理モジュール