卒業論文

題目

エージェント指向言語MALS

指導教授

所 真理雄 教授

慶應義塾大学

理工学部 電気工学科

1995年度

69202702

勝野 恭治

論文要旨

学科	電気工学	学籍番号	69202702	氏名	勝野 恭治

(論文題目)

エージェント指向言語MALS

大規模のネットワーク上で広域に分散した計算機間にわたる作業を行う場合、あるいは携帯計算機をネットワークに接続する場合、モーバイルエージェントを用いたプログラミングは有効な手法であると考えられている。モーバイルエージェントはネットワーク上の計算機間を自律的に移動し、各計算機上の情報を収集し、利用しながらプログラマの要求を満たす。

しかし、既存のモーバイルエージェントのためのプログラミングシステムにおいて、複数の操作の原子的実行を保証するシステムは存在しない。複数の操作の原子的実行は、モーバイルエージェントを記述する際にしばしば用いられるため、モーバイルエージェントのためのプログラミングシステムは、複数の操作の原子的実行を保証し、プログラマを支援する必要がある。

本研究では、エージェント指向言語MALS(Mobile Agent Language System)の提案、実装を行う。MALSでは、言語システムが複数の操作の原子的実行を保証している。これにより、プログラマは、プログラム実行中の障害を意識する必要がなくなる。このため、MALSは高い記述性を持ち、プログラマの負荷を軽減することができる。本論文ではさらに、性能評価を行ない、MALSの複数の操作の原子的実行を実現する部分のオーバーヘッドの割合を明らかにすることでMALSの実用性を実証する。

慶應義塾大学理工学部

◎ 内容の要旨は約600字

Summary of Bachelor's Thesis

Course	Student No.	SURNAME, Firstname
Electrical Engineering	69202702	KATSUNO, Yasuharu

Title

Agent Oriented Language, MALS

Mobile agent has been proposed as a programming model suited for communications over largely distributed network and for mobile computers connected to the network. A mobile agent satisfies the requirements from programmer by autonomously migrating from a computer to another, looking for the computational resources, and gathering the information it need.

However, previous proposals lack an important programming functionality: guarantee of atomic execution of a set of operations. In applications suited for mobile agents, there are many chances that a set of operations are meaningful if and only if they are all executed. Without programming systems support for such a guarantee, a programmer must guarantee the atomic execution of operations by herself, which would be a cumbersome task.

This study proposes an agent oriented language *MALS (Mobile Agent Language System)*, and shows its implementation. *MALS* guarantees atomic executions of many operations on the language system. Under *MALS* system, programmer doesn't need to be conscious of the conflicts or failures while executing the program. Therefore,*MALS* can reduce programmer's load by its programmability. Performance evaluation of *MALS* shows its overhead against language systems without atomic execution guarantees, and proves its feasibility and applicability.

Faculty of Science and Technology, KEIO UNIVERSITY

⊚ The contents of English summary should be written in about 300 words.

目 次

図 目 次

表 目 次

第 1 章

序論

コンピュータの利用形態は、時代と共に変化している。かつては1台のコンピュータをネットワークに接続せずに使用していた。続いて、複数のコンピュータをネットワークで接続するようになった。さらに利用形態が集中型から分散型へと変化して、分散システムが誕生した[21]。

近年、ネットワークの利用状況が激しく変化してきている。インターネットが爆発的な勢いで成長している。利用者は現在推定4000万人以上といわれ、西暦2000年には3億人に達するという予想がされている。その結果、World Wide Web[20]上における情報検索など、大規模のネットワーク上で、広域に分散した計算機間にわたる作業が増加している。また、モーバイルコンピュータが出現した。A4サイズ、重さ2kg程度のコンピュータと無線ネットワークを用いることによって、ネットワークに接続でき、様々なサービスを受けられるようになった。

従来から分散システムにおいて、異なるホスト上に存在する手続きをプログラマのいるホストから実行させるための機構として、RPC(Remote Procedure Call)[26]が幅広く利用されている。RPCは、プログラマに対し位置透過なプログラミング環境を提供する。これより、プログラマは、どのようなコンピュータがどこに設置され、どのように接続されているかといったことを意識せずプログラミングを行うことができる。

LAN(Local Area Network)などの小規模のネットワークにおいて、RPCは有効な手法である。しかし、大規模のネットワーク上で広域に分散した計算機間にわたる作業をする際に、RPCを用いると、

- 大きな伝達遅延が生じる通信を多数行うため、手続きを呼び出してから返答が返ってくるまでの時間が長い。
- データを管理するサーバが、他のホストに変わった際の対応が困難になる。

ということが問題になる。また、モーバイルコンピュータを無線ネットワークを用いて、広域ネットワークに接続する際に、RPCを用いると、

- モーバイルコンピュータをネットワークに接続し続けなければならない時間が長い。

ということが問題となる。

そこで、その問題点を克服するために登場したのがリモートプログラミング[23] [7] [25] [24]である。リモートプログラミングとは、プログラムを目的のホストに移動させ、実行させる手法である。リモートプログラミングを用いると、大きな伝達遅延が生じる通信の回数が減少するため、手続きを呼び出してから、返答が返ってくるまでの時間が

短くなる。また、ネットワークと接続し続けなければならない時間が短くなる。しかし、リモートプログラミングもRPCと同様に、データを管理するサーバが、他のホストに変わった際の対応が困難になるということが問題となる。

RPC、リモートプログラミングの問題点を克服するために登場したのがモーバイルエージェントである。モーバイルエージェントとは、ネットワーク上を自律的に移動し、各計算機上に存在する情報を収集、利用しながらユーザの要求を満たしていくプログラムである。モーバイルエージェントを用いることによって、大きな伝達遅延が生じる通信の回数が減少するため、手続きを呼び出してから、返答が返ってくるまでの時間が短くなる。また、ネットワークと接続し続けなければならない時間が短くなる。さらに、データを管理するサーバが、他のホストに変わった際に対応できるようにもなる。

既存のモーバイルエージェント記述用言語[5] [6]には、複数の操作の原子的実行を支援した言語は存在しなかった。複数の操作の原子的実行とは、複数の操作を全て実行するか、全く実行しないというall or nothingの実行を保証する機能である。複数の操作の原子的実行は、モーバイルエージェントを用いる際にしばしば用いられる機能である。既存のモーバイルエージェントでは個々の操作は独立しており、複数にわたる操作で1つの意味のある作業をさせる場合、プログラマの記述量が多くなる。しかし、複数の操作の原子的実行を支援することによって、モーバイルエージェントを記述するプログラマは、プログラム実行中の障害を意識する必要がなくなる。これら独立した一連の操作に対し、複数の操作の原子的実行を定義できるようにすることで、モーバイルエージェントを記述するプログラマの負担を軽減することができると考えられる。

本研究では、既存のモーバイルエージェント記述用言語の問題点を踏まえて、複数の操作の原子的実行を支援するエージェント指向言語MALS(Mobile Agent Language System)を提案し、実装を行う。さらに、このエージェント指向言語MALSの実装と評価を通して、分散システムにおけるMALSの有効性を検証する。

本論文は、本章も含めて7章から構成されている。第2章では、広域ネットワークの利用形態の変化を説明した後、関連研究について説明する。第3章では、本研究の骨子であるエージェント指向言語MALSのシステムモデルについて説明する。第4章では、本研究で用いた実装言語Javaを概観し、第3章で説明したMALSの実装方法を述べる。第5章では、第4章で実装したエージェント指向言語MALSの評価と検討を通して、分散システムにおけるMALSの評価と検討を行う。第6章では今後の課題、将来への展望などを検討し、第7章で本研究の結論を述べる。

第 2 章

背景

本章では、関連研究についてまとめ、近年のネットワークの利用状況の変化について概観する。

2.1 関連研究

本節では、本研究について関連のある

- RPC
- リモートプログラミング言語
- ネットワーク上を移動するモーバイルエージェントを用いたシステム

について、これまでに行われた研究を紹介する。

2.1.1 RPC

分散システムにおいて、異なるホスト上に存在する手続きをプログラマのいるホストから呼び出すために、従来から幅広く利用されていた技術として RPC(Remote Procedure Call) [26] という手法がある。RPC は、異なるホストに存在する手続きを同一プログラムの中に定義された手続きと同じように呼び出すための機構を提供する。

RPC では、実際の通信はメッセージ伝達で行われる。そこで、プログラミング言語で記述された手続き呼び出しを、メッセージ伝達に変換する必要がある。そのためスタブと呼ばれる手続きを用いる。スタブとは、クライアントのホストとサーバのホストに置かれ、呼び出しに対してはメッセージを組み立てて送信し、呼び出される側ではそのメッセージを元に手続き呼び出しを行う。

RPC の注目すべき点は、クライアントの操作を、通常の手続き呼び出しと同じようにしたという点である。手続き呼び出しは、プログラミングの手法として長年に渡って親しまれており、プログラマにとっても非常に理解がしやすい。このため、これらの手順をパッケージにまとめて、内部が利用者に見えないようにしてしまえば、プログラマは、どのようなコンピュータがどこに設置され、どのように接続されているかといったことをほとんど意識することがなくなる。つまり位置透過性が得られるのである。

RPC の実行経過を図 2.1に示す。この図において、プログラマが自分で記述するのは、斜線の 2 つの長方形の部分だけでよい。スタブはスタブジェネレータというトランスレータによって自動生成される。この結果、プログラム自体もすっきりと分かりやすいものになる。

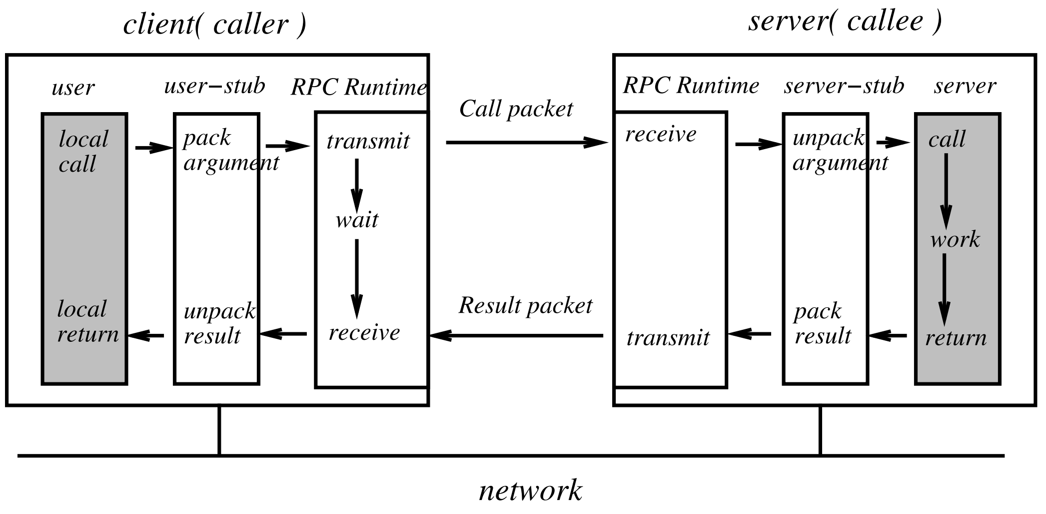

図 2.1: RPC の構造

一般に分散システムには、異なる機種の計算機同士が通信し合うプログラムは異なる言語で記述されることがある。こうした異種環境での通信を支援する RPC として、Sun ONC (Open Network Computing) の RPC [21]、Mach の MIG(Mach Interface Generator) [27]、HRPC(Heterogenerous Remote Procedure Call)[28] の 3 つを紹介する。

Sun ONC の RPC は、XDR(eXternal Data Representation) と呼ぶデータ表現変換システムを備えている。これは異機種計算機間の通信において、バイト順、語長、浮動小数点形式の変換を行う。

Mach の Matchmaker は、異なるプログラミング言語で共通仕様の RPC を利用できるようにするためのインタフェース言語である。MIG は、Matchmaker のサブセットで、C および C++のための RPC インターフェースを生成する。プログラマは、MIG 定義ファイルと呼ばれる仕様ファイルを作成する。このファイルはデータ型の定義と操作を記述したものである。配列や構造体の定義は Pascal 風の構文規則に従う。そのほか、ポインタ型や多相型も考慮している。また、データ型を定義した C のヘッダファイルを取り込むことも可能である。MIG は、この定義ファイルをもとに、クライアント、サーバ双方のスタブ、およびコンパイラが使用する各種宣言を含んだヘッダファイルを生成する。スタブルーチンは、Mach のタスク間の通信機能を用いて、通信を行うようになっている。

HRPC は、RPC を構成する、バインディング、データ転送、制御、データ表現の間に明確なインタフェースを定義し、これらのモジュールを組み合わせてシステムを構築しようというアプローチをとっている。各モジュールの組合せにより、既存のシステムの

多くをエミュレートすることができる。

2.1.2 リモートプログラミング言語

リモートプログラミングとは、プログラマのいるローカルホストからリモートホストへプログラムを送る手法である。リモートプログラミングによるホスト間の手続き呼び出しは、基本的に、次のような手順で動作する。

1. ローカルホストにいるプログラマが、リモートホストに送るプログラムを作成する。
2. ローカルホスト上のシステムは、リモートホストと接続し、そのプログラムをリモートホストに送る。送り終わったらリモートホストとの接続を切る。
3. リモートホスト上のシステムは、送られてきたプログラムを実行する。
4. 実行し終わったら、リモートホスト上のシステムは、ローカルホストと接続し、実行結果をローカルホストに送る。送り終わったら、ローカルホストとの接続を切る。
5. ローカルホスト上のシステムは、実行結果を受け取る。

リモートプログラミングの動作を、図 2.2に示す。

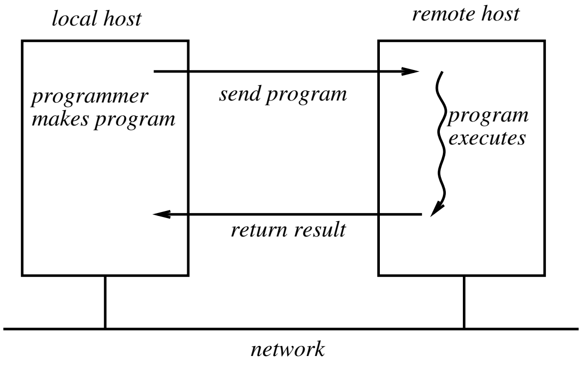

図 2.2: リモートプログラミングの動作

リモートプログラミング言語としては、Obliq[23]、Emerald[24]、Distributed Smalltalk[25]、Phantom[7] などがある。ネットワーク上のホスト間を移動するプログラムは、ホスト間の通信回数を減少させるために移動する。

また、RPC では、リモートホストにある手続きは常にリモートホスト上で実行されてしまう。そのため、リモートホストに負荷が集中してしまうという問題点がある。リ

モートホスト上に存在するリソースにアクセスしない手続きなら、リモートホスト上で実行しなくても構わない。負荷分散の観点から、このような手続きは、リモートホストとは別のホストで実行されることが望ましい。

以上の問題点を解決するために Phantom[7] では、リモートホストからローカルホストにプログラムを送り、ローカルホスト側で手続きを実行する。図 2.3に動作の概要を示す。図 2.3で示しているように、リモートホストからローカルホストへプログラムを移動させ、ローカルホストで実行することによって、リモートホストの負荷を分散することができる。

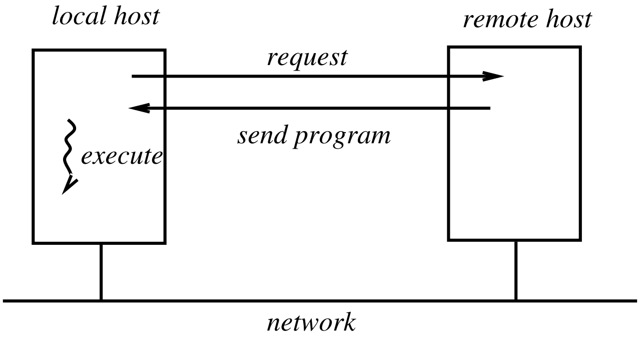

図 2.3: Phantom

分散環境を考慮した言語として、従来から分散プログラミング言語というものがある[29]。しかしその多くは、リモートホストに存在する手続きを実行させるのに RPC を用いている。

2.1.3 モーバイルエージェントを用いたシステム

モーバイルエージェントといっても、その定義は論文によって異なっている [5] [6] [10] [14] [15] [16] [17] [18] [19]。これらの論文で提案されているモーバイルエージェントの機能について大きく 2 つにまとめる。

- ネットワーク上の情報を収集し、自律的に移動する

  ネットワークに存在するホスト上のシステムと通信を行うことによって、情報を収集する。収集した情報は、ホスト間を移動するのに利用されたり、プログラマの大まかな要求に対して、最も適切な行動を取るプランニングで利用されたりする。

- モーバイルエージェント同士が通信をする

  モーバイルエージェント同士が通信を行うことによって、情報収集を行ったり、複数のモーバイルエージェントで協力して1つの仕事を行ったりする。

モーバイルエージェントによるホスト間の手続き呼び出しは、基本的には、次のような手順で動作する。

1. ローカルホストにいるプログラマが、モーバイルエージェントを作成する。
2. ローカルホスト上のシステムは、リモートホストと接続し、モーバイルエージェントを移動させる。モーバイルエージェントが移動し終わったら、リモートホストとの接続を切る。
3. モーバイルエージェントは、ホスト上のシステムから情報を収集し、その情報を元に、移動する。そして、プログラマの要求を満たすことのできるホスト上で実行する。
4. モーバイルエージェントが、プログラマの要求を全て満たしたら、モーバイルエージェントは、現在いるホスト上のシステムに移動の要求を出す。すると、ホスト上のシステムは、ローカルホストと接続し、モーバイルエージェントをローカルホストに移動させる。モーバイルエージェントが移動し終わったら、ローカルホストとの接続を切る。
5. ローカルホストはモーバイルエージェントを受け取る。

モーバイルエージェントの動作を、図2.4に示す。

モーバイルエージェントは、手続きを持っているホストに負荷が集中するという問題がある。しかし、RPC、リモートプログラミングと比べて、ホスト間の通信回数が少ない、ネットワークと接続する時間が短い、データを管理するサーバが他のホストに変わった際に対応することができるといった利点がある。負荷集中よりも、その利点の方が重要である。

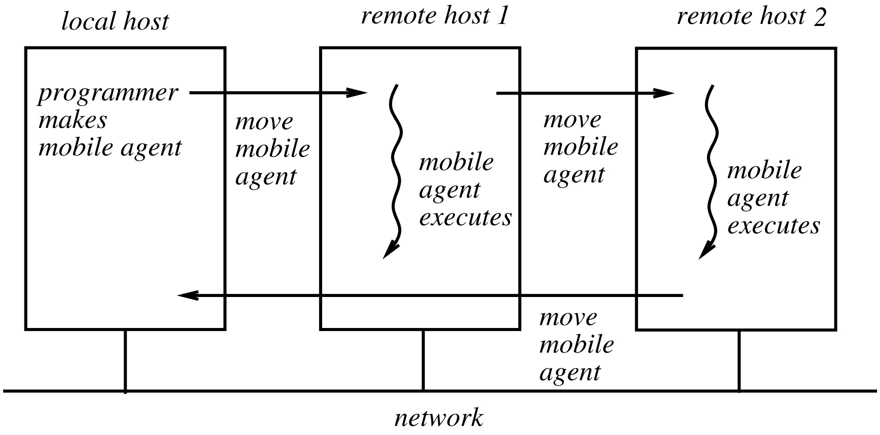

図 2.4: モーバイルエージェントの動作

一般に、モーバイルエージェントといっても、モーバイルエージェントがどの様な目的で何をするかによって、その役割は異なる。本節ではネットワーク上を移動するモーバイルエージェントを用いたシステムを比較する。

Telescript

Telescript[5] [6] とは、General Magic 社が開発した言語であり、ネットワーク上を動き回り、プログラマの要求を満たすモーバイルエージェントを用いる。Telescript では、モーバイルエージェントは各ホストに存在しているステーショナリーエージェントや、他のモーバイルエージェントと通信をすることによって情報を収集することができる。

利用方法としては、買物をすることを主眼に置いていて、モーバイルエージェントの効果的な使用法として以下の 3 点を挙げている。

- 他のホスト上のデータを監視し、変化が生じた場合、プログラマに知らせに行く。
- ネットワークを動き回って情報収集をする。
- 複数の要求を 1 度に指定する。

Flage

Flage [8] は、プログラマなどの他のエージェントからの要求の変化に対し、稼働時に動的かつ自動的に適応したり、極力人手を労さずに適応できる柔軟なソフトウェア (適応型ソフトウェア) を構築することを目標としている。Flage ではそのためエボルーショナ

ルエージェントというものを用いる。エボルーショナルエージェントの定義は以下のようになっている。

> エボルーショナルエージェント
>
> 自律的にネットワーク上を渡り歩き、ネットワークのノード上に存在するソフトウェアを獲得することで、自分自身を進化させ要求の変化に対応させる。

Flageの利用例を紹介する。プログラマのいるホストにデータベースを操作するmanipulate-data agentというエージェントが存在する。プログラマがmanipulate-data agentにmergeというオペレーションを要求したとする。しかし、プログラマのいるホストにあるmanipulate-data agentにはmergeというオペレーションが存在しない。通常のシステムの場合、manipulate-data agentからプログラマにエラーメッセージが返る。しかし、Flageではmanupulate-data agentがネットワークを歩き回り、mergeオペレーションを実行できるフリーソフトウェアを探し、発見することができたら、そのフリーソフトウェアを取り込む。その後、プログラマのいるホストに戻り、mergeオペレーションを実行する。

エボルーショナルエージェントと、Telescriptのモーバイルエージェントとの根本的な違いとして挙げられるのは、ネットワークを動き回る目的である。つまり、Telescriptのモーバイルエージェントがネットワークに存在する情報を収集したり、リモートホストに存在するリソースを利用するのが目的なのに対して、エボルーショナルエージェントは、ネットワークに存在するフリーソフトを探し、自分自身を進化させるのが目的である。

NMW-System

NMW-System[11] [12] [13]は、ネットワークワームと呼ばれる、ネットワークを動き回る管理エージェントを用いている。管理エージェントがネットワーク上を動き回る目的は、ネットワークの管理を行うためである。NMW-Systemは、ネットワークの管理に利用することを目的として開発されたシステムである。

現在のインターネットにおけるネットワーク管理には、ネットワークの管理をネットワークを介して行なっている。そのため、ひとたびネットワークに障害が発生すると、ネットワーク全体の状態を取得、制御することができなくなってしまうという欠点がある。

その欠点を解決するためにNMW-Systemでは、ネットワークを移動しながら自分自身の判断でネットワークを調べられる管理エージェント、ネットワークワームを用いる。ネットワークワームをネットワーク管理に用いると、以上のような利点が考えられる。

- ネットワーク上を移動しながらの調査、分析、制御が可能となる。

  移動しながら調査することができるので、複数のホストの状況に応じた作業が出来る。

- 管理作業に必要な通信量を削減できる。

  ワームのプログラムによって、必要な情報のみを選別できるので、作業に必要な通信量が削減できる。この特徴は、管理者が極端に回線容量が低い環境から操作する場合や、ネットワークが輻輳を起こしている際に有効である。

- ネットワークの障害時にも対応できる。

  ネットワークが障害を起こした場合、通常の方法では、障害によって切断されたネットワークの状態を知ることはできない。しかしワームを用いれば、ワーム自身の判断で障害に対応することができる。対応の例としては、障害が復旧するまで待つ方法や、管理用の経路でワームの実行結果を管理者に送るという方法などがある。

2.2 ネットワークの利用形態の変化

近年、ネットワークの利用形態が変化してきている。ネットワークの利用形態の変化として以下の2点が挙げられる。

- 大規模のネットワーク上で、広域に分散した計算機間にわたる作業の増加

  近年、インターネットは爆発的な勢いで成長し、また World Wide Web [20] が進歩した。その結果として、大規模のネットワーク上で情報検索などの広域に分散した計算機間にわたる作業が増加している。

- モーバイルコンピュータのネットワークへの接続

  モーバイルコンピュータを、無線ネットワークを用いて、ネットワークに接続するようになった。しかし、無線で接続しているので、接続が不安定になってしまうという難点がある。そのため、モーバイルコンピュータとネットワークとを接続し続ける時間を、できるだけ短くする必要がある。

2.2.1 より、上記の利用状況で、RPC、リモートプログラミング、モーバイルエージェントを用いた場合について考える。

2.2.1 RPC

広域に分散した計算機間にわたる作業

RPC を用いる際に問題になる点として、以下の 2 点が挙げられる。それぞれ説明する。

- 手続きを呼び出してから、返答が返ってくるまで時間が大きい。
- データを管理するサーバが、他のホストに変わった際の対応が困難になる。

RPC は、そもそもプログラマのいるローカルホスト、目的のリモートホスト間の通信回数が多い。なぜなら、ローカルホストからリモートホストに存在する手続きを 1 回実行するのに、ローカルホスト、リモートホスト間で 2 回のメッセージ伝達が要求されるからである。つまり、リモートホストに存在する手続きを n 回実行させるのに、ローカルホスト、リモートホスト間で $2n$ 回のメッセージ伝達が要求されることになってしまう。図 2.5にローカルホストにいるプログラマがリモートホストに存在する手続きを RPC を用いて 3 回実行した場合を示した。

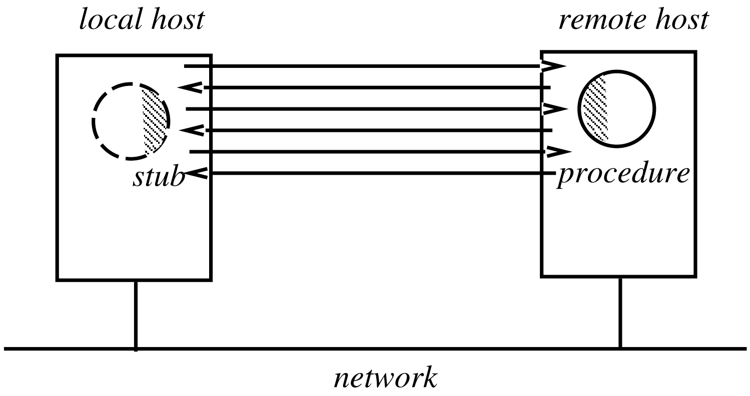

図 2.5: リモートホストにある手続きを RPC を用いて 3 回実行した場合

伝達遅延の小さい小規模ネットワークで作業をする際は、通信回数が多いことが問題にはならなかった。しかし、大規模ネットワークで広域に分散した計算機間にわたる作業をする際は、伝達遅延が大きいので、通信回数が多いことが問題となる。なぜなら伝達遅延が大きいため、ローカルホストからリモートホストに存在する手続きを呼び出してから、返答が返ってくるまでの時間が大きくなるからである。

また、大規模のネットワークでは、プログラマがネットワーク全体を把握することができない。そのため、プログラマの知らない間にデータを管理するサーバが他のホストに変わる場合がある。例えば、あるデータベースを管理するホストが、リモートホスト1からリモートホスト2に変更されたとする (図2.6)。その様なことが起きても、ネットワークが大規模のため、プログラマは、データベースを管理するホストが変更したということを知ることができない。そのため、プログラマはリモートホスト1に対して、データベースにアクセスする手続きの呼び出しを行う。しかし、リモートホスト1にはプログラマの求めるデータベースがないので、エラーメッセージが返ってくる。そのエラーメッセージにより、プログラマは、データベースがリモートホスト1からリモートホスト2に移動したことを知る。しかし、リモートホスト2にある手続き2に対して、手続き呼び出しを行わなければならないので、通信回数が増加してしまう。また、データを管理するサーバが、他のホストに変わるごとにプログラマが対応しなければならないので、プログラマの負担が大きくなってしまう。

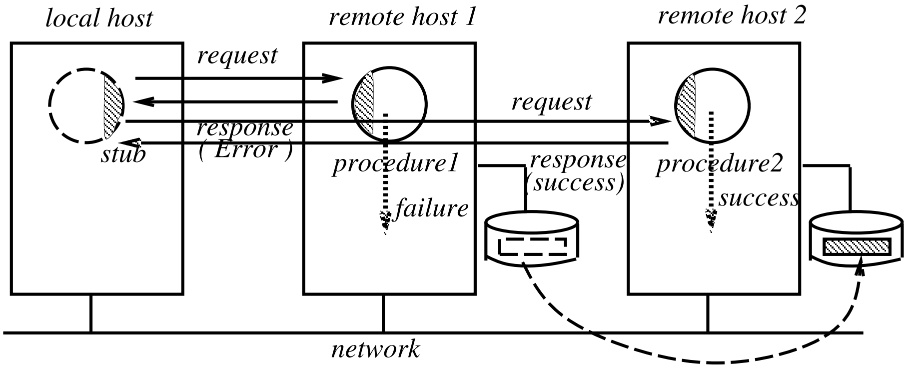

図2.6: データを管理するサーバが他のホストに変化した際のRPCの対策

モーバイルコンピュータのネットワークへの接続

ネットワークを接続したり、切断したりするためには、接続や切断をチェックするプログラムコードがいろいろな所に必要になる。プログラムの単位が手続きで、ホスト間の通信回数が多いRPCでは、ネットワークの接続や、切断のチェックを行うのは適切ではない。そのため、RPCは、手続き呼び出しを行ってから返答が返ってくるまで、ローカルホストとリモートホスト間を接続し続けなければならない。ローカルホストからリモートホストに存在する手続きを3回実行した場合を、図2.7に示した。大規模のネットワーク上において広域に分散した計算機間にわたる広域な作業を行う際に、伝達遅延が

大きいという現状を考えると、手続きを呼び出してから、返答が返ってくるまでの時間が増える。つまり、RPC を用いると、ネットワークと接続し続けなければならない時間が増えてしまう。

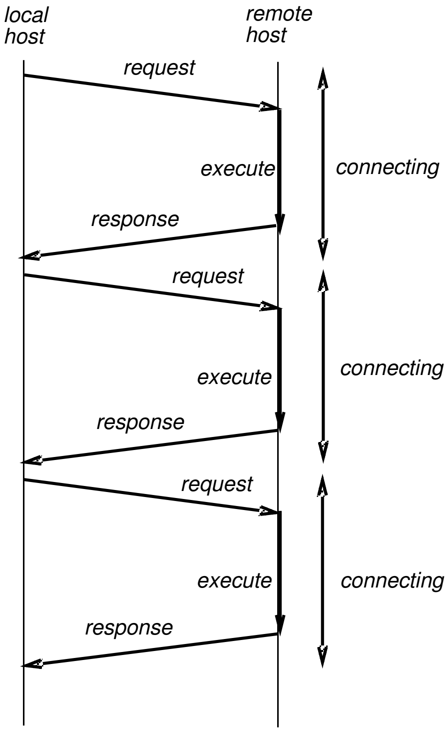

図 2.7: RPC を用いた場合のネットワークと接続している時間

2.2.2 リモートプログラミング

広域に分散した計算機間にわたる作業

RPC では、プログラマのいるローカルホストからリモートホストに存在する手続きを、n 回実行させるのにローカルホスト、リモートホスト間で $2n$ 回のメッセージ伝達を要求する必要がある (図 2.5参照)。しかし、リモートプログラミングでは、一度ローカルホストからリモートホストにプログラムを送れば、すべての実行が終了するまでローカルホスト、リモートホスト間の通信はない。つまりリモートホストに存在する手続きを 1 回実行する場合においても、n 回実行する場合に置いてもローカルホスト、リモートホスト間の通信回数は 2 回になる。図 2.8にローカルホストにいるプログラマがリモートホストにある手続きをリモートプログラミングで 3 回実行した場合を示した。

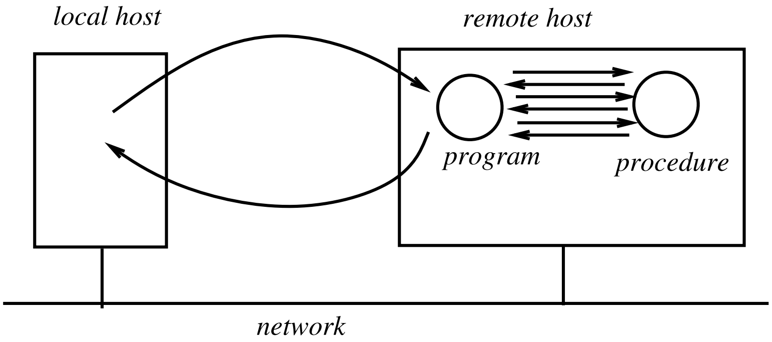

図 2.8: リモートホストにある手続きをリモートプログラミングで 3 回実行した場合

また、データを管理するサーバが、他のホストに変わった際の対応について考えてみる。前節で用いた例を再び用いる。

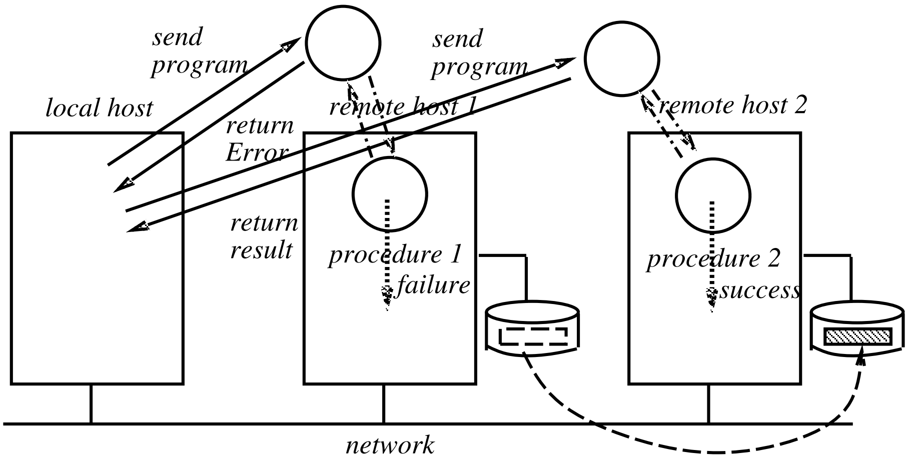

図 2.9: データを管理するサーバが他のホストに変わった際のリモートプログラミングの対策

あるデータベースを管理するホストがリモートホスト 1 からリモートホスト 2 に変更されたとする (図 2.9)。その様なことが起きても、ネットワークが大規模のため、プログラマはデータベースを管理するホストに変更があったことを知ることができない。そのため、プログラマはリモートホスト 1 にプログラムを送る。しかし、リモートホスト 1 にはプログラマの求めるデータベースがないので、プログラムを実行できない。そして、エラーメッセージが返ってきて、そのエラーメッセージにより、プログラマはデータベースがリモートホスト 1 からリモートホスト 2 に移動したことを知る。しかし、リモートホスト 2 にある手続き 2 に対してプログラムを送らなければならないので、通信

回数が増加してしまう。また、データを管理するサーバが、他のホストに変わるごとに、プログラマが対応しなければならないので、プログラマの負担が大きくなってしまう。

モーバイルコンピュータのネットワークへの接続

リモートプログラミングは、RPC に比べて、プログラミングの単位が大きく、また、ホスト間の通信回数が少ない。そのため、ネットワークとの接続や切断に関するプログラムコードを組み込んでも問題はないと考えられる。

よって、リモートプログラミングの場合、ローカルホスト、リモートホスト間を接続し続けなければならないのは、ローカルホストからリモートホストへプログラムを送る時と、リモートホストからローカルホストへ実行結果が返ってくる時のみである。つまりリモートホストでローカルホストから送られたプログラムが実行している間は、ローカルホストとリモートホストを接続し続けなくてもよいのである。プログラマのいるローカルホストからリモートプログラミングで、リモートホストに存在する手続きを 3 回実行した場合を、図 2.10に示した。

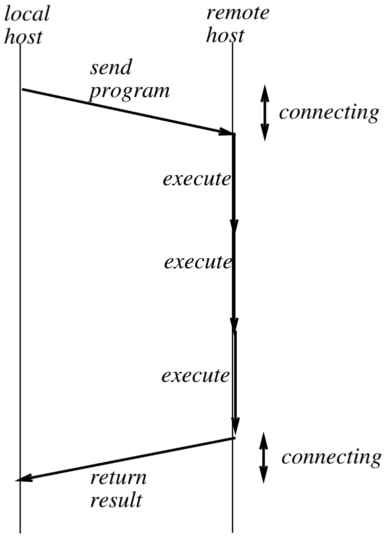

図 2.10: リモートプログラミングを用いた場合のネットワークと接続している時間

2.2.3 モーバイルエージェント

広域に分散した計算機間にわたる作業

モーバイルエージェントを用いると、リモートプログラミングと同様にローカルホストから、リモートホストに存在する手続きを呼び出すのに、ローカルホスト、リモートホスト間の通信は常に2回になる。そのため、リモートプログラミングと同様、手続きを呼び出してから返答が返ってくるまでの時間が、RPCに比べて減少する。

また、モーバイルエージェントを用いると、プログラマの知らない間にデータを管理するサーバが他のホストに変わった場合、システム側で対処するので、プログラマが対処する必要がない。前節で用いた例を再び用いて具体的に説明する。

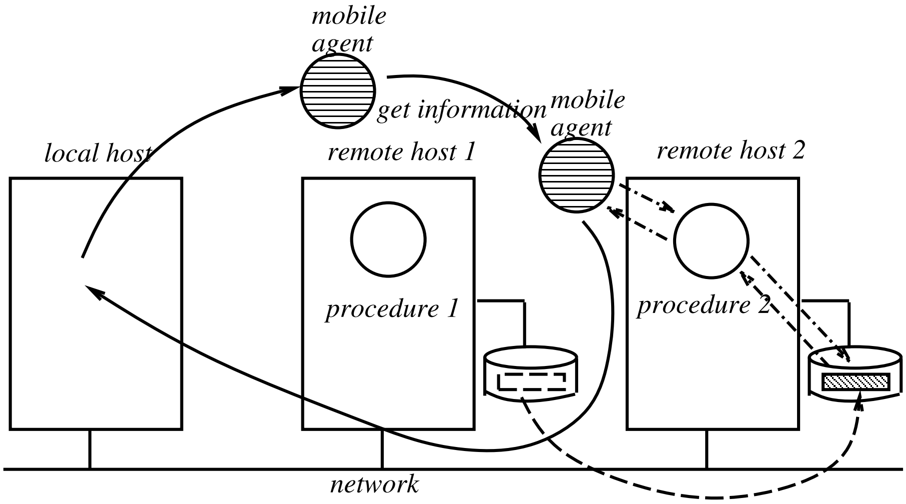

図2.11: データを管理するサーバが他のホストに変わった際のモーバイルエージェントの対策

あるデータベースを管理するホストが、リモートホスト1からリモートホスト2に変更されたとする(図2.11)。その様なことが起こっても、ネットワークが大規模なため、プログラマはデータベースを管理するホストに変更があったことを知ることができない。そのため、プログラマは、リモートホスト1にモーバイルエージェントを送る。しかし、リモートホスト1にはプログラマの求めるデータベースがないので、モーバイルエージェントは実行することができない。しかし、モーバイルエージェントがリモートホスト1から、情報を収集して、目的のデータベースがリモートホスト2に変更されたことを知ることができる。そして、モーバイルエージェントは、リモートホスト2に移動し、目的のデータベースにアクセスする。よって、RPCやリモートプログラミングに比べて、

プログラマの負担が軽減される。

モーバイルコンピュータのネットワークへの接続

　前節で説明したリモートプログラミングと同様に、ローカルホスト、リモートホスト間を接続し続けなければならないのは、ローカルホストからリモートホストへモーバイルエージェントを送る時と、リモートホストからローカルホストへモーバイルエージェントが返ってくる時だけである。つまり、リモートホストでモーバイルエージェントが実行している間は接続し続けなくてもよい。

2.3 まとめ

以上、大規模のネットワーク上で、広域に分散した計算機間にわたる作業を行う際や、モーバイルコンピュータをネットワークに接続する際に、RPC、リモートプログラミング、モーバイルエージェントを用いた場合をまとめると、次の表 2.1のようになる。

	広域にわたる作業	モーバイルコンピュータ
RPC	×	×
リモートプログラミング	△	○
モーバイルエージェント	○	○

○	問題がない
△	一部問題がある
×	問題がある

表 2.1: RPC、リモートプログラミング、モーバイルエージェントの比較

本章では、まず、本論文との関連研究について議論した。そして、近年のネットワークの利用形態の変化について検討し、RPC、リモートプログラミングでは問題があるということを示した。そして、その解決策であるモーバイルエージェントを説明した。

3 章では、現状のモーバイルエージェントのためのプログラミングシステムの問題点を解決する本論文の対象でもある、MALS のシステムモデルについて詳しく見ていく。

第 3 章

エージェント指向言語 MALS のシステムモデル

本研究では、現状のモーバイルエージェント記述言語の問題点を指摘し、解決する、エージェント指向言語 MALS(Mobile Agent Language System) を提案する。本章では、複数の操作の原子的実行、MALS の目的、MALS エージェント、また、MALS のシステムモデルについて説明する。

3.1 複数の操作の原子的実行

システム管理者が、複数の設定ファイルに対して、全てをアップデートするか全くアップデートしないといった実行を行うことを考える。この場合、アップデートできなかった場合を考慮して、プログラムを記述しなければならない。つまり、プログラマが、プログラム実行中の障害を意識しなければならないのである。すると、プログラマの記述量が多くなってしまう。

そこで、複数の操作の原子的実行が支援されていると、上記の問題は解決される。複数の操作の原子的実行は、複数の操作を全て実行するか、全く実行しないかを保証する[22]。これにより、プログラマが障害を意識する必要がなくなり、プログラマの負担が軽減する。

本論文では、障害として、トランザクション障害を考えている。トランザクション障害は、特定の複数のデータに対するひとまとまりの処理を行っている際に、全ての実行を取消し、実行をする前の状態に戻さなければならなくなったことによる障害である。

3.2 MALS の目的

既存のモーバイルエージェント記述言語では、複数の操作の原子的実行が支援されていなかった。モーバイルエージェントを用いる際に、複数の操作の原子的実行はしばしば用いられる機能である。例えば、飛行機、フェリーのチケット、現地のホテルの予約を 3 つ全部確保したい場合を考える。しかし、実際には 3 つ全部確保できない時がある。また、5 つのファイルをアップデートする際に、全てをアップデートするか、全てをアップデートしないかのどちらかにしたい、などといった利用は数多く存在する。既存のモーバイルエージェント記述言語では、複数の操作の原子的実行が支援されていないため、複数の操作の原子的実行を実現するためには、モーバイルエージェントの記述がかなり複雑になってしまう。そのため、モーバイルエージェントを記述するプログラマの大きな負担となる。

本論文の提案する MALS(Mobile Agent Language System) は、既存のモーバイルエージェント記述言語が、複数の操作における原子的実行が支援されていないという問題点

を解決することを目的としている。MALS は、プログラマが指定したホストに移動して、プログラマの要求を満たし、プログラマのいるホストに戻ってくる MALS エージェントを実現する言語システムである。MALS では、複数の操作の原子的実行を支援しており、MALS エージェントを記述するプログラマの負担を軽減させる。

3.3 MALS エージェント

本論文の定義する MALS エージェントは、ローカルホストにいるプログラマによって作成され、プログラマの要求を満たすために、ネットワークに接続されたホスト間を自律的に移動し、ホスト上の MALS システム (3.4.2 節参照) と通信をする、つまり以下の

- ネットワーク上の情報を収集する
- ネットワークを自律的に移動する

ということを行うプログラムである。

自律的に移動するとは、MALS エージェントを作成したプログラマが移動を記述していなくても移動を行うということである。例えば、プログラマが、リモートホスト 1 に存在するデータベースをアクセスする、という MALS エージェントを作成したとする。この場合、プログラマは、リモートホスト 1 に存在するデータベースをアクセスする、という箇所を記述する。そして、そのデータベースが、リモートホスト 1 からリモートホスト 2 に変更された場合を考える。MALS エージェントが、目的のデータベースがリモートホスト 1 からリモートホスト 2 に変更された、という情報を収集することができたとする。すると、システム側で、MALS エージェントをリモートホスト 2 に移動させる。本論文では、プログラマが移動を行うことを記述していなくても移動を行うことを、自律的に移動すると定義する。

3.4　MALS のシステムモデル

MALS は以下に示す構成要素

- MALS エージェント
- MALS システム

からなる。それぞれを順に説明していく。

3.4.1　MALS エージェント

MALS エージェントは図 3.1のような構造をしている。

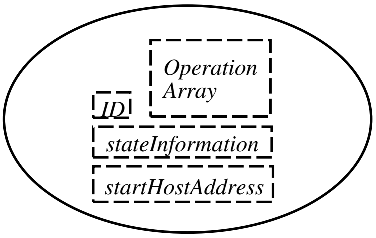

図 3.1: MALS エージェントの構造

点線で囲まれている四角形が、MALS エージェントが保持している情報である。

[**MALS エージェントが保持している情報**]

ID

MALS エージェントを識別するために用いられる世界で唯一の文字列である。

Operation Array

プログラマが、MALS エージェントを作成する際に記述したプログラムが保持されている。

stateInformation

ホスト上で得た情報を保存する際に用いられる。例えば、リモートホスト 1 に存在するファイルから得た情報を、リモートホスト 2 に存在するファイルに書き込む場合、リ

モートホスト 1 に存在するファイルから得た情報を MALS エージェントが保持して、リモートホスト 2 に移動しなければならない。その際に、MALS エージェントは、そのファイルから得た情報を stateInformaiton に保持する。

startHostAddress

MALS エージェントが、プログラマのいるホストに戻る際に用いられる。その MALS エージェントを作成したプログラマのいるホストの名前が用いられる。

3.4.2 MALS システム

MALS システムは各ホストに存在し、MALS エージェントと通信を行なうことによってさまざまなサービスを提供する。MALS システムは図 3.2のような構造をしている。

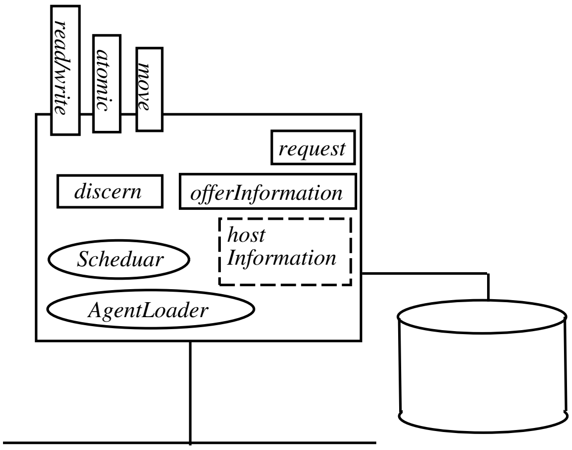

図 3.2: MALS システムの構造

実線で囲まれている楕円形は、MALS システムが保持している機構であり、点線で囲まれている四角形が、MALS システムが保持している情報であり、実線で囲まれていて、外側の四角形の内部にある四角形が、MALS システム自身が用いるメソッドであり、実線で囲まれていて、外側の四角形の外部にある四角形が、MALS エージェントに公開しているオペレーションである。それぞれを説明していく。

[**MALS システムが保持している機構**]

Schedular

ホスト上のデータをアクセスする read/write オペレーションや、複数の操作の原子的実行を実現する atomic オペレーションの処理順を決める機構である。

AgentLoader

ホストに移動してきた MALS エージェントをロードする機構である。MALS エージェントは、AgentLoader にロードされることによって、そのホスト上で実行することができる。

[**MALS システムが保持している情報**]

hostInformation

ネットワークの状況、そのリモートホストの状況、他のホストの情報などである。

[**MALS システム自身が用いるメソッド**]

discern

discern とは、MALS エージェントを識別するオペレーションである。ホスト上で公開されている全てのデータが、全ての MALS エージェントに対して公開されているとは限らない。例えば、システム管理のファイルなら、そのホストを管理している管理者の作成した MALS エージェントのみに公開する。また、有料な情報なら、情報料を支払ったプログラマの作成した MALS エージェントのみに公開する。以上のような状況を考えると、MALS エージェントを識別する機能は重要である。MALS システムは、discern オペレーションを用いることによって、MALS エージェントを識別することができる。具体的には、MALS エージェントに対して request オペレーションを用いて ID を要求し、その ID を discern オペレーションを用いて検査し、MALS エージェントを識別する。そして、リモートホスト上に存在するデータに対するアクセスを許可するかを決定する。もしアクセスを許可しないのなら、MALS エージェントにエラーを返す。

request

request とは、MALS エージェントに要求を出すオペレーションである。例えば、データベースに対するアクセスを考える。ホストの負荷を分散するために、データベースを、

読み書きのできるマスターと、読むことのみできるスレーブに分けて、それぞれ別々のホストに置くことが考えられる。その場合、例えばデータベースを検索するのみなら、スレーブ側のデータベースで十分である。マスター側のデータベースが存在するホストが、たくさんの MALS エージェントが来ているため、負荷が高くなっている場合を考える。すると、データベースの検索のために移動してきた MALS エージェントを、スレーブ側のホストに移動させなければならない。そのため、マスター側のホストに新たに来た MALS エージェントのアクセス要求が、もし検索するのみだったら、その MALS エージェントに、アクセス要求を中断し、スレーブ側のホストに移動するように要求を出す。よって負荷を分散することができる。以上を図式化すると、図 3.3になる。

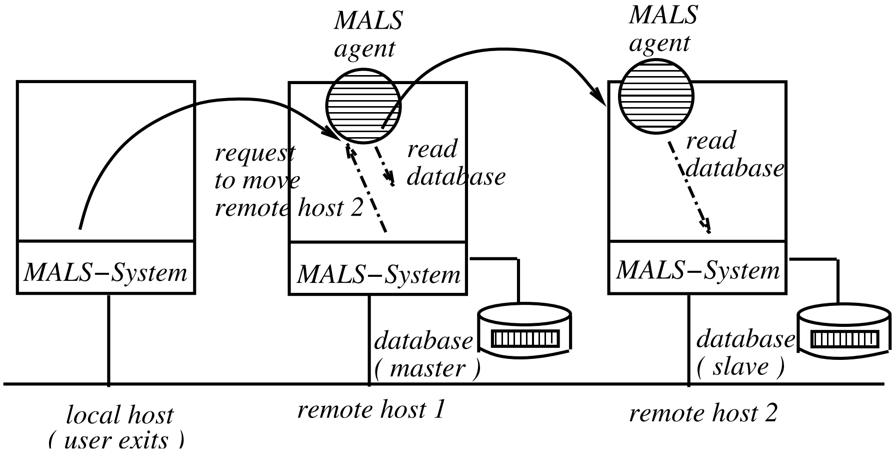

図 3.3: MALS システムの request の使用例

また、MALS エージェントを識別するために、MALS エージェントに対して識別子を要求する際にも用いる (discern を参照)。

offerInformation

offerInformation とは、ホストで所有している情報を MALS エージェントに提供するオペレーションである。ホストが所有するデータベースやファイルなどが、別のホストに移動したことを考える。そのような変化が起こっても、プログラマのいるホストでは知ることができない。そのため、プログラマから移送されてきた MALS エージェントも、当然変化があったことを知ることができない。しかし、MALS システムが MALS エージェントに対して、目的のデータを保持するホストが変化した、という情報を提供することによって、MALS エージェントは変化を知ることができる。

例えば、リモートホスト 1 にあったデータベース 1 が、リモートホスト 2 に移動した状況を考える。リモートホスト 1 にあるデータベース 1 をアクセスするために、ローカルホストにいるプログラマから MALS エージェントが移送されてきたとする。MALS エージェントは、リモートホスト 1 の MALS システムにファイル 1 に対する読み込みの要求、read オペレーションを出す。しかし、データベース 1 は、リモートホスト 1 には存在しない。そこで、MALS システムは MALS エージェントに対して、データベース 1 がリモートホスト 1 からリモートホスト 2 に移動した、という情報を提供する。その情報を得た MALS エージェントはリモートホスト 2 に移動し、データベース 1 にアクセスすることができるようになる。以上を図式化すると、図 3.4になる。

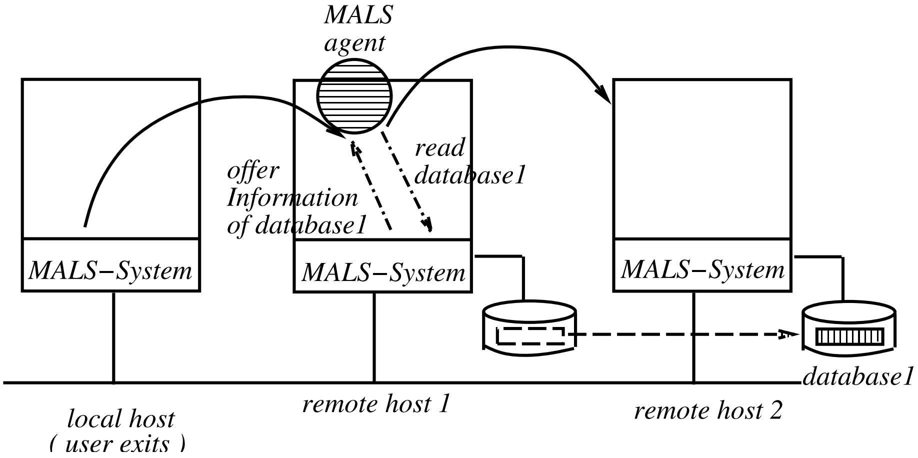

図 3.4: MALS システムの offerInformation の使用例

[**MALS** エージェントに公開しているオペレーション]

move

move とは、MALS エージェントがホストからホストへ移動する際に用いられるオペレーションである。MALS エージェントが move オペレーションを実行すると、現在、MALS エージェントがいるホスト上の MALS システムに対して、移動目的のホストに移動する、という要求を出し、MALS システムがそれを実行する。よって、MALS エージェントは移動することができる。その際に、MALS エージェントは現在の状態を保存したまま移動することになる。移動先のホストは、ホストの名前で指定しなければならない。

もし、移動先のホストにMALSシステムがなかったり、MALSエージェントが指定したホストそのものが存在しなかった場合は、MALSエージェントにエラーが返る。

read/write

read/writeとは、MALSエージェントがホスト上に存在するファイル、データベースなどといったデータに対して読み込み/書き込みをする際に用いるオペレーションである。MALSエージェントは、ホスト上のデータに直接アクセスすることはできない。MALSエージェントがホスト上のデータにアクセスするには、MALSシステム上で公開されているread/writeオペレーションを用いて、データの読み込み/書き込みを行う。その結果が、MALSエージェントへと返ってくる (図3.5参照)。

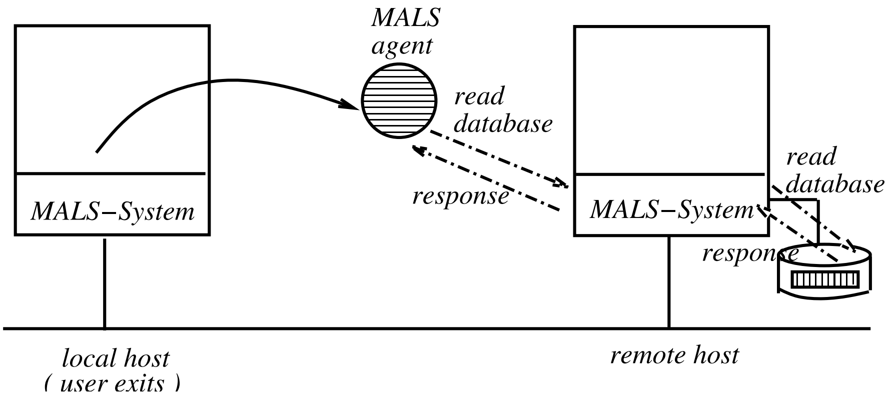

図 3.5: MALS システムの read の使用例

MALSエージェントが、直接ホスト上のデータにアクセスできない理由として、次の2点が挙げられる。

まず、MALSエージェントが、ホスト上のデータに不当にアクセスするのを防止することが挙げられる。ホスト上のデータには、MALSエージェントには公開したくないデータも当然存在する。read/writeオペレーションでは、指定されたデータしかアクセスすることができない。そこで、MALSエージェントが、ホスト上のデータをアクセスする際に、MALSシステムのread/writeオペレーションを使用することになれば、MALSエージェントのデータに対するアクセスを制限することができる。MALSエージェントの認証については、MALSシステムのdiscernオペレーションを用いて認証をする。

次に、MALSエージェントのサイズを小さくし、様々なデータに対応させることが挙げられる。一般に、データベース、ファイルへのアクセス方法は、各データベース、ファイルシステムで異なっている。もし、MALSエージェントがデータベース、ファイルに直

接アクセスするのであれば、MALS エージェント自身がそれぞれのデータベース、ファイルシステムのアクセスに対応していなければならない。しかし、ローカルホストのプログラマは、目的のホスト上のデータベース、ファイルシステムの種類を知ることができるのはまれので、MALS エージェントが考えられる全てのデータベース、ファイルシステムのアクセスに対応していなければならない。そうすると、データベース、ファイルシステムが新たに変更された場合に対応することができなくなってしまう。また、MALS エージェント自身のサイズが非常に大きくなってしまう。そこで、データベースに対するアクセスを MALS システムが行えば、データベースが新たに変更された場合は、MALS システムの read/write オペレーションをその変更に対応させればよいということになる。

read オペレーションは、ホスト上の情報を収集する際にも用いる。read オペレーションを用いることにより、MALS エージェントはホスト上の情報を収集することができる。

例えば、以前リモートホスト 1 に存在したデータベース 1 がリモートホスト 2 に移動したとする。その際、リモートホスト 1 のシステム管理者が、データベース 1 がリモートホスト 2 に移動した、という情報をリモートホスト 1 に保存したとする。その情報を、リモートホスト 1 を訪れた MALS エージェントが read オペレーションを用いて収集することができる。ホストの状況や、ネットワークの状況は常に変化する。その変化の情報をホスト上に記録しておき、そのホストを訪れた MALS エージェントがその情報を得ることによって、変化に対応することができる。当然、ホスト上の情報は変化に応じて、常に変わっていく。

atomic

atomic とは、MALS エージェントが、ホスト上に存在するデータに対して複数の読み込み/書き込みを全て実行するか、全く実行しないということを保証する場合、つまり、複数の読み込み/書き込みの原子的実行を保証する場合に用いるオペレーションである。

このオペレーションの具体的な利用例を挙げると、

- システム管理者が複数の設定ファイルに対して、全てアップデートをするか全くアップデートしないといった実行を行いたい。
- 旅行を計画しているプログラマが、飛行機のチケット、フェリーのチケット、ホテルの予約を取る。その際、3 つのうちの 1 つでも取れなければ、旅行に行くことができないので、取るのなら 3 つ全部取りたい。

などが挙げられる。複数の操作における原子的実行の記述は、MALS システム側で支

援されていなくても実現することができる。しかし、さまざま問題が生じてしまう。具体的な例を取り上げて説明する。例えば、

ファイル 1 から 1 行読み込んで、それをファイル 2、ファイル 3、
ファイル 4、ファイル 5 に書き込む

という操作を考えてみる。複数の操作の原子的実行を考えない実行では、図 3.6の様に記述できるとする。

```
String St = read( "file1" );
write( St, "file2" );
write( St, "file3" );
write( St, "file4" );
write( St, "file5" );
```

図 3.6: 複数の操作の原子的実行を考えない実行

もし、この複数のオペレーションに対して原子的実行を保証したい場合、つまり 5 つのオペレーションのうちどれか 1 つでも失敗した場合 (ファイルが存在しない、MALS システムから許可されていないデータをアクセスしようとしてアクセスすることができない) に、全てのオペレーションを取り消して、元の状態に戻したい場合を考えると、図 3.7のようなプログラムが考えられる。

図 3.7のプログラムを説明すると、read/write ともにアクセスに失敗すると ERROR を返す。また全てのオペレーションをキャンセルする際には、書き込みを行ったファイルを元に戻さなければいけないので、ホスト上の MALS システムの、ファイルの回復を行う recovery というオペレーションが、モーバイルエージェントに公開されていなければならない。

上記のプログラムは 2 つの問題点がある。まず、複数の操作の原子的実行を考えない場合の記述と比べて、かなり複雑である。そのため、MALS エージェントを記述するプログラマの負担が大きくなる。続いて、MALS エージェント自身が、書き込んだファイルを回復させなければならないので、ホスト上の MALS システムにある recovery オペレーションが、MALS エージェントに対して公開されていなければならないということである。つまり、MALS エージェントが、ホスト上のデータに対して回復を行うオペレーションを、自由に用いることができることになってしまう。そのような状況で、図 3.8の様な悪質な MALS エージェントが移動してきたら、ホスト上に存在するデータの一貫性が崩れてしまう恐れがある。MALS エージェントに recovery オペレーションを公開するのは適切ではない。

```
String St;
int i, j, m;

if ( ( St = read( "file1" ) ) == ERROR )
        exit();

if ( ( i = write( St, "file2" ) ) == ERROR )
        exit();

if ( ( j = write( St, "file3" ) ) == ERROR ){
        recovery( "file2" );
        exit();
}

if ( ( k = write( St, "file4" ) ) == ERROR ){
        recovery( "file2" );
        recovery( "file3" );
        exit()
}

if ( ( m = write( St, "file5" ) ) == ERROR ){
        recovery( "file2" );
        recovery( "file3" );
        recovery( "file4" );
        exit()
}
```

図 3.7: 複数の操作の原子的実行が支援されていない場合

```
int i;

String S = read( "file1" );
if ( ( i = write( S, "file2" ) ) == ERROR ){
        recovery("file1");
        recovery("file2");
}
```

図 3.8: 悪質な MALS エージェント

本論文の MALS エージェントは、図 3.9のような記述により、複数の読み込み/書き込みにおける原子的実行を支援する。beginAtomic, endAtomic に囲まれた複数のオペレーションに対して原子的実行が保証される。図 3.9のプログラムと図 3.7のプログラムを比較する。

```
beginAtomic{
        String St = read( "file1" );
        write( St, "file2" );
        write( St, "file3" );
        write( St, "file4" );
        write( St, "file5" );
}endAtomic
```

図 3.9: 複数の操作の原子的実行が支援されている場合

まず、記述が複雑になってしまうという問題があった。しかし、複数の操作の原子的実行を考えない場合 (図 3.6) とほぼ変わらない記述で表現できる。よって、MALS エージェントを記述するプログラマの負担が少なくなる。

続いて、ファイルの回復をする recovery オペレーションが MALS エージェントに公開されていることにより、ホスト上のデータの一貫性が崩れてしまう、という問題があった。しかし、図 3.9で示しているように、MALS エージェントが recovery オペレーションを用いる必要がないので、MALS エージェントに recovery オペレーションを公開する必要がなくなる。よって、ホスト上のデータの一貫性が崩れるといった事態を防ぐことができる。

原子的実行の仕組みを説明する。原子的実行の対象となるオペレーションを実行する前に、MALS エージェントは、MALS システム上の atomic オペレーションを実行する。すると、MALS システム上の Schedular に beginAtomic というメッセージが送られる。そして、MALS エージェントが読み込み/書き込みを行う。原子的実行の対象となるオペレーションが全て終了したら、モーバイルエージェントは、再び MALS システム上の atomic オペレーションを実行する。すると、MALS システム上の Schedular に endAtomic というメッセージが送られる。すると Schedular は、beginAtomic から endAtomic までの read/write オペレーションに対しての原子的実行を保証する。

3.5 まとめ

本章では、まず、MALSエージェント、複数の操作の原子的実行について説明した。続いて、本論文の提案するMALSエージェントを用いるエージェント指向言語MALS(Mobile Agent Language System)の目的、システムモデルについて説明した。

第4章では、MALSの実装について詳しく述べる。

第 4 章

エージェント指向言語 MALS の実装

本章では、MALS の実装について説明する。本論文では、プロトタイプとして Java 言語を用いて MALS を実現している。まず、Java 言語について説明し、続いて MALS の実装方法について説明する。

4.1 Java 言語

Java 言語 [1][2][3] とは Sun Microsystems によって開発されたプログラミング言語である。Java 言語の目的は、ソフトウェア開発者の作業負荷を C/C++を使用する時よりも軽減させるということである。Java 言語の特色を以下で説明する。

- オブジェクト指向

  Java 言語では、各種オブジェクトを収めた既存ライブラリを用いることによってネットワーク、グラフィカルインターフェースなど各種の機能を利用できる。

- 信頼性

  Java のメモリ管理モデルは、基本的にポインタ操作やポインタ演算を行わないので、これまで C や C++のプログラマが悩まされてきたポインタ操作に関するエラーが排除される。

- 中立性と移植性

  Java 言語は、ハードウェアアーキテクチャの種類やオペレーティングシステム、または言語環境には依存することなく、ネットワーク環境で動作するアプリケーションを支援することを主眼として設計されている。Java 言語コンパイラにより、アーキテクチャに依存しないバイトコードが生成される。

- インタプリタ形式

  Java 言語が持つインタプリタ形式という特性により、全てのプラットフォームで同じ Java 言語のバイトコードが実行でき、従来のようなバイナリ形式による提供やバージョン管理の問題が解消される。Java インタプリタは、インタプリタとランタイムシステムを移植したマシンであれば機種を問わずに、Java バイトコードを直接実行できる。

- マルチスレッド対応

  Java 言語に組み込まれたマルチスレッド機能により、アプリケーション中で複数スレッドの同時処理を行うことができる。

本論文でプロトタイプとして Java 言語を用いた理由として以下の点が挙げられる。

- インタプリタ言語である

  分散システムに接続されているホストは、全て同じ機種、同じ OS ではない。しかし、Java 言語はインタプリタ言語なので、Java のインタプリタ、ランタイムが移植されていれば機種、OS を問わず、同じ Java のプログラムを実行させせることができる。

- ソースコードをチェックする

  Java 言語では、Java のソースを中間コードに変換するが、その過程で不当なデータ変換が行われていないか、などといったチェックを行う。

- 中間コードをネットワークを介してロード可能である

  Java では、中間コードをネットワークを介して、ロードすることができる。よって、モーバイルエージェントをソースではなく、チェックされた中間コードとして送ることができる。

- 自動的なメモリ管理である

  メモリ管理を全てシステム側で行うので、ユーザの作成した MALS エージェントがメモリクラッシュを起こすことがない。

4.2 MALS の構成

本論文では、Java Release1.0 Alpha3 版を用いて、MALS を実装した。実装のシステム構成を図で示すと、図 4.1のようになる。

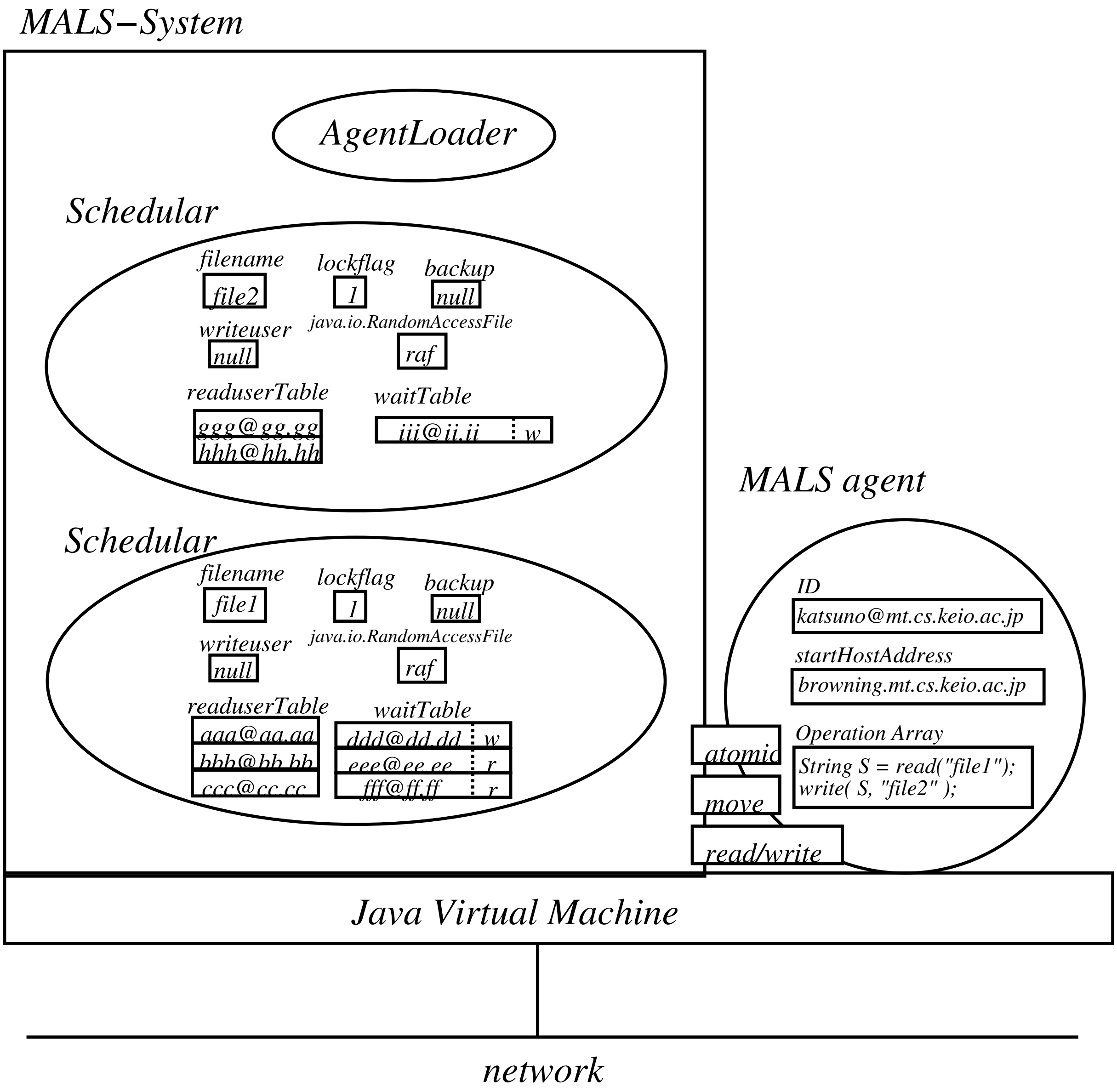

図 4.1: MALS の構成

4.3 節は、図 4.1を用いて、MALS システムの実装方法について説明する。

4.3 MALS システムの実装方法

4.3.1 read/write, atomic, Schedular

本節では、ホスト上に存在するファイルを MALS エージェントがアクセスする際に使用されるオペレーション、機構を説明する。

MALS エージェントに公開するファイルには、それぞれ Schedular が作成される。図4.1を見ると、file1、file2 に対する Schedular が存在するので、file1、file2 が公開されていることが分かる。

MALS エージェントがファイルにアクセスする場合、MALS システム上で公開されているオペレーション read/write、atomic を用いる。MALS エージェントが read/write、atomic オペレーションを実行したら、Schedular に要求を出す。そして、Schedular で処理をして、処理の結果が MALS エージェントに返る。

Schedular とは、複数の MALS エージェントが、ホスト上のデータをアクセスするという状況において、ホスト上のファイルの一貫性を保証する機構である。Schedular では、ファイルの一貫性を保証するために 2 相ロック [22] という手法を用いている。2 相ロックについて説明する。

2 相ロックとは、衝突を生じる操作に対し、その操作の順序関係を決定するためにロック機構を用いる手法である。データを操作する場合は、まず、そのデータのロックを獲得することから始まる。ロックを獲得できたトランザクションは、実際に、そのデータに対し、操作を進めることができる。トランザクションは、一度ロックを解除すれば、そのトランザクション内で他のデータに対しても、再びロックを獲得することはできない。この方針をとることで、トランザクション間の順序関係を決定することが可能となる。また、2 相ロックでは、複数のトランザクションが互いにロックが獲得できるのを待ってしまい、デッドロックを引き起こすという欠点があり、これを回避するための機構を必要とする。そのため本論文では、Schedular にデッドロックを回避する機構が備わっている。

Schedular を構成している要素を、図 4.1の Schedular の図を用いて説明する。

filename

この Schedular がスケジューリングの対象としているファイルの名前である。

lockflag

ファイルのロック状況を表す。本論文では、3 種類のロックを導入している。それぞれ説明する。

- 読み込みロック (**lockflag:1**)

  ファイルの読み込みに対するロックである。あるファイルに対して読み込みしか行わないトランザクションは、同じファイルに対して読み込みしか行わないトランザクションと決して競合することがない。つまり、複数のトランザクションが読み込みロックを共有することがありえる。読み込みは書き込みに比べて頻繁に行われるので、この方法は並行性の向上に効果があると考えられる。

- 書き込みロック (**lockflag:2**)

  ファイルの書き込みに対するロックである。書き込みロックされているファイルに、他のトランザクションからアクセスされるとそのファイルの一貫性が失われてしまうので、他のトランザクションのアクセスが禁止される。

- 書き込み予約ロック (**lockflag:3**)

  複数のトランザクションの読み込みロックに対する共有を認める方法は、並行性の向上に効果がある。しかし、読み込みしか行わないトランザクションが、書き込みを伴うトランザクションの進行を妨げてしまうことになる。そこで本論文では、書き込み予約ロックを導入した。書き込み予約ロックとは、トランザクションがあるファイルに書き込みをする際、そのファイルに読み込みロックがかかっていた場合に、そのファイルに対して書き込み予約ロックをかける。書き込み予約ロックをかけられたファイルには、現在読み込みを行っているトランザクション以外、読み込みを行うことはできない。そして、現在読み込みを行っているトランザクション全ての読み込みが終了したら、書き込み予約ロックをかけたトランザクションが書き込みを行うことができる。この書き込み予約ロックによって、読み込みしか行わないトランザクションが、書き込みを伴うトランザクションの進行を妨げてしまうことが減少すると思われる。

以上、3 つのロックをまとめると表 4.1のようになる。

backup

トランザクションがアボートをする際には、そのトランザクションが書き込みを行ったファイルを、書き込みをする前の状態に戻さなければならない。そのため、ファイル

要求ロック ＼ データのロック状態	ロックなし	読み込み	書き込み予約	書き込み
読み込み	可	可	待ち	待ち
書き込み	可	待ち	待ち	待ち

表 4.1: ロック規則

に書き込みを行う際に、backup という Java 言語の byte 型のデータタイプにバックアップを取る。

writeuser

ファイルの書き込みを行っている MALS エージェントの ID を示す。

raf

ファイルアクセスを行うために、Java 言語の java.io.RandomAccessFile クラスのインスタンス raf を用いている。ファイルの読み込みを行うために、ファイルを先頭の 1 行を読み込む readLine メソッドを、ファイルの書き込みを行うために、ファイルの先頭の 1 行に書き込む writeBytes メソッドを使用している。

readuserTable

ファイルの読み込みを行っている MALS エージェントの ID を示す。

waitTable

MALS エージェントはファイルに対する読み込み/書き込みが lockflag の関係上、実行することのできない場合、MALS エージェントの ID と、アクセスモード (読み込みなら r、書き込みなら w) を waitTable に登録する。waitTable に登録された MALS エージェントは、10ms ごとに lockflag と、waitTable を見る。もし、waitTable の先頭になっていて、登録したアクセスモードが実行できる状態になっていたら、waitTable から削除し、ファイルにアクセスすることができる。10ms ごとに waitTable を見る、ということを 20 回行っても自分の番になっていなかったら、全てのオペレーションをアボートし、オペレーションを始めからやり直す。しかし、アボートする MALS エージェントが、atomic オペレーションを実行していてたら、atomic オペレーションを用いないオペレーション

を始めからやり直すことになる。waitTable に登録されている MALS エージェントが、このようにアボートすることによりデッドロックが防がれる。

4.3.2 AgentLoader

MALS エージェントは、Java 中間コードの形式でホスト間を移動する。AgentLoader とは、ホストに移動してきた Java 中間コードをロードする役割をなしている。Java 中間コードをロードするのに、Java 言語で用意されている java.util.ClassLoader クラスを利用している。ClassLoader クラスはメモリ上の Java 中間コードをロードすることができない。そのため、移動してきた Java 中間コードを一度ファイルに落し、そのファイルからロードしている。

4.4 MALS の実行手順

本節では、本論文で実現した MALS の実行手順を説明する。ローカルホストにいるユーザが、リモートホストに存在するファイルにアクセスする際の手順を説明する (図 4.2)。

図 4.2を説明する前に、図 4.2の中で出てきた、トランスレータ mals、コンパイラ javac、結果ファイルについて説明する。

トランスレータ mals

トランスレータ mals は、MALS のソースを Java のソース.java ファイルに変換する。本論文では、mals を Perl5.000 を用いて作成した。mals は、MALS のソースを Java のソースに変換すること以外に、send モードと receive モードという 2 つのモードを用いて、次のことを行う。

send モード

send モードとは、図 4.2のローカルホスト側 (左側) の全ての実行、つまり、MALS のソースを Java のソースに変換してから、結果ファイルを受け取るまでの手順を行うモードである。構文としては以下のようになっている。

mals -s *User'sE-MailAdderss LocalHostName malsfile*

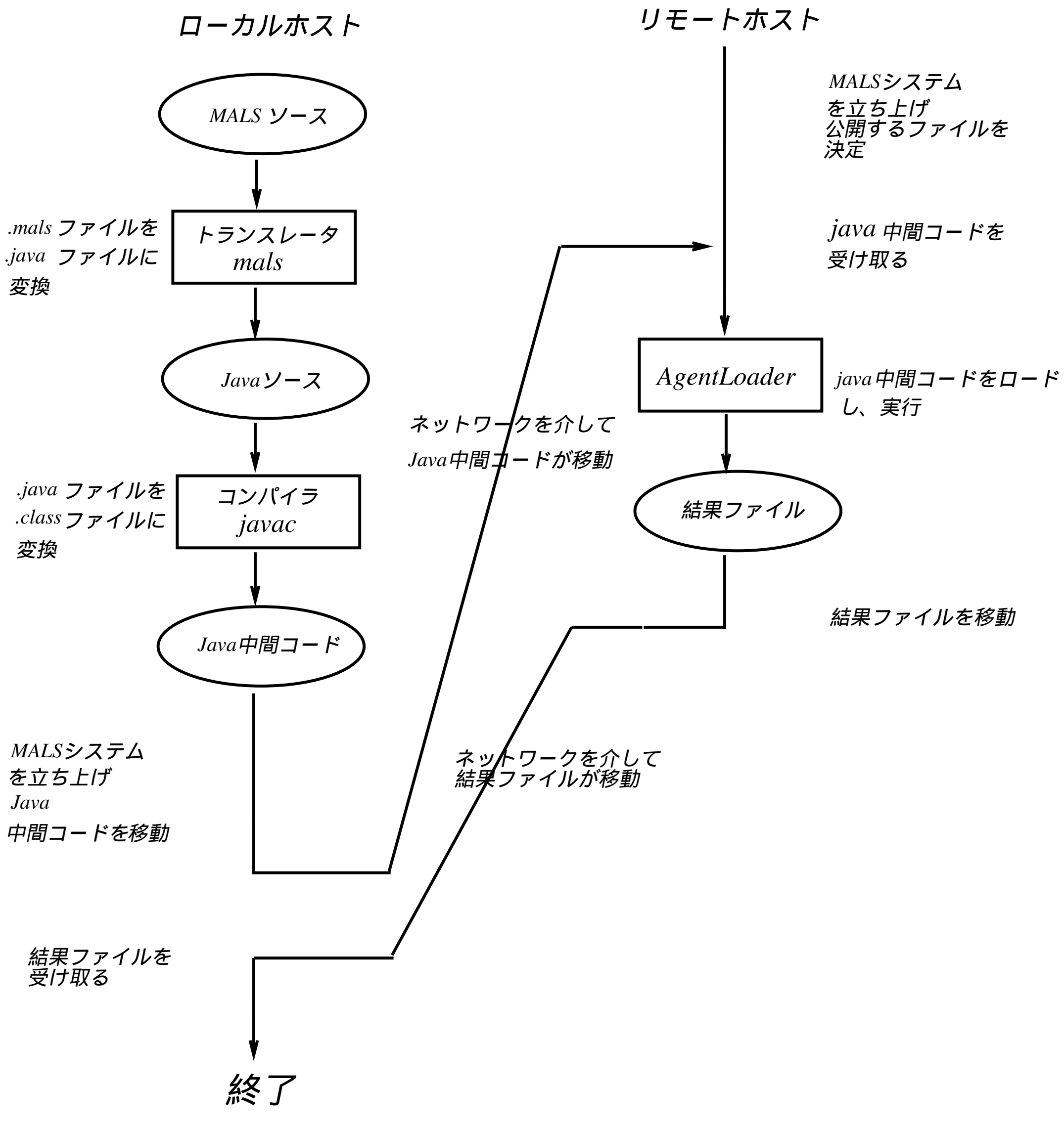

図 4.2: MALS の実行手順

receive モード

receive モードとは、図 4.2のリモートホスト側 (右側) の全ての実行、つまり、MALS システムを立ち上げ、公開するファイルを決定してから、結果ファイルを移動させるまでの手順を行うモードである。構文としては以下のようになっている。

mals -r *filename* *filename* ....

コンパイラ javac

コンパイラ javac は、Java のソースをコンパイルし、Java 中間コードを作成する。javac は Java Release1.0 Alpha3 版を用いている。

結果ファイル

本論文では、MALS エージェントの stateInformation を実現していない。よって、MALS エージェント自身にリモートホストで行った read/write の結果を保持させることができない。そのため、本論文では、リモートホストで行った read/write のログを結果ファイルという形でローカルホストに移動させる。結果ファイルに記録されるログを以下にまとめる。

- **Read for File** *filename* **,success.**

  ファイル *filename* に対する読み込みが成功した。

- **Write for File** *filename* **,success.**

  ファイル *filename* に対する書き込みが成功した。

- **File** *filename* **doesn't exist.**

  ファイル *filename* が存在せず、読み込み/書き込みができなかった。

- **File** *filename* **doesn't exist, so abort.**

  ファイル *filename* が存在せず、読み込み/書き込みができず、アボートした。

実行手順

図 4.2をローカルホスト側とリモートホスト側に分けて説明する。

ローカルホスト側

1. ユーザが MALS エージェントを記述し、拡張子.mals のファイル、MALS のソースを作成する。
2. ユーザが先程の拡張子.mals のファイルに対して、トランスレータ mals を send モードで実行させる。
3. トランスレータ mals が拡張子.mals の MALS のソースを拡張子.java の Java のソースに変換する。
4. コンパイラ javac が拡張子.java の Java のソースを拡張子.class の Java の中間コードに変換する。
5. MALS システムを立ち上げて、先程の Java 中間コードをユーザに指定されたリモートホストに送る。
6. リモートホストから送られてきた結果ファイルを受け取る。

リモートホスト側

1. リモートホスト側のシステム管理者が、MALS エージェントに公開するファイルを指定するために、トランスレータ mals を receive モードで実行させる。
2. MALS システムが立ち上がり、MALS エージェントに公開するファイルを指定する。
3. ローカルホストから送られてきた Java 中間コードを受け取る。
4. MALS システムの AgentLoader を実行し、ローカルホストから送られてきた Java 中間コードをロードし、MALS エージェントを実行する。
5. MALS エージェントが実行した結果、生成された結果ファイルを、ローカルホストに送る。

4.5 まとめ

本章では、まず、プロトタイプとして使用したJava言語について説明し、前章で述べたMALSがどのように実装されたかを説明した。

第5章では、本章で実装したMALSの評価を行う。

第 5 章

評価及び検討

5.1 評価

数値の評価は、全てFUJI XEROX Argoss 5250(CPU: Sparc2、OS: Solaris 2.4) 上で行った。そして、時間を測定するのに、Java言語のjava.systemクラスのcurrentTimeMillionsメソッドを用いた。currentTimeMillisメソッドは、msの単位でプログラムの実行時間を測定することができる。しかし、UNIXのプロセススケジューリングの機構上の制約から、実際に測定できるのは10msまでなので、測定した時間の1msのオーダーは四捨五入した。

5.1.1 複数の操作の原子的実行に関する記述性の評価

本節では、MALSの複数の操作の原子的実行に関する記述性を評価する。例として、ホスト上に存在するファイルに、MALSエージェントがアクセスするという状況を考える。まず、複数の操作の原子的実行を保証しない場合、図5.1のような記述になる。

```
agent EVA {

    beginLocation( browning.mt.cs.keio.ac.jp ){

        String S = read( "alpha" );
        write( "beta",  S );
        write( "gamma", S );
        write( "delta", S );

    }endLocation
}
```

図5.1: 複数の操作の原子的実行を保証しない場合

図5.1は、browning.mt.cs.keio.ac.jp上に存在するファイルalpha, beta, gamma, deltaに対して、alphaから読み込んだ先頭の1行をbeta、gamma deltaに書き込むという操作を行うEVAという名前のモーバイルエージェントを記述したプログラムである。

次に、複数の操作の原子的実行を保証する場合のプログラムを考える。複数の操作の原子的実行が支援されていなくても、図5.2のように記述ができると考えられる。

プログラム中のオペレーションを説明すると、read/writeともにアクセスに失敗するとERRORを返す。また、全ての操作をキャンセルする際には、書き込みを行ったファイルを元に戻さなければならない。そこで、ホスト上のMALSシステムが保持してい

```
agent EVA {

    beginLocation( browning.mt.cs.keio.ac.jp ){

        String S, T, U, V;


        if ( ( S = read( "alpha" ) ) == ERROR )
            exit();

        if ( ( U = write( "beta",  S ) ) == ERROR )
            exit();

        if ( ( T = write( "gamma", S ) ) == ERROR ){
            recovery( "beta" );
            exit();
        }

        if ( ( V = write( "delta", T ) ) == ERROR ){
            recovery( "beta"  );
            recovery( "gamma" );
            exit();
        }

    }endLocation
}
```

図 5.2: 複数の操作の原子的実行が支援されていない場合

る、ファイルの回復を行う recovery というオペレーションが、MALS エージェントに公開されていなければならない。

このプログラムには問題点が 2 点ある。それぞれを説明する。まず、複数の操作の原子的実行を保証しない場合の記述 (図 5.1) と比べて、かなり複雑になってしまうという問題点がある。browning.mt.cs.keio.ac.jp 上に存在するファイル alpha, beta, gamma, delta, zeta に対して、alpha から読み込んだ先頭の 1 行を beta, gamma, delta に書き込む。もし beta に書き込むことができなかったら、代わりに zeta に文字列を書き込む、といった if 文が入ってしまうと図 5.3のように更に複雑になってしまうと考えられる。よって、複数の操作における原子的実行が支援されていないと、複数の操作における原子的実行を保証しない場合と比べ、記述がかなり複雑になってしまい、MALS エージェントを記述するプログラマの大きな負担になってしまうことが分かる。

次に、ホスト上のファイルの一貫性が崩されてしまう恐れがあるという問題点がある。これは、ホスト上の MALS システムが保持している、ファイルを回復させる recovery オペレーションが、MALS エージェントに対して公開されていなければならないためである。つまり、MALS エージェントが、ホスト上に存在するファイルの回復を自由に行ってしまう可能性が考えられるからである。そのような状況では、プログラマが悪質な MALS エージェントを作成し、ホスト上のファイルの一貫性が崩してしまう恐れがある。よって、複数の操作の原子的実行が支援されていないと、ホスト上のファイルの一貫性が崩れてしまう恐れがあることが分かる。

本論文の MALS エージェントは、複数の read/write オペレーションにおける原子的実行を支援しており、先程の 2 つの問題点を解決する。それぞれ説明する。まず、記述が複雑になるという問題があったが、先程の例の、browning.mt.cs.keio.ac.jp 上に存在するファイル alpha, beta, gamma, delta に対して、alpha から読み込んだ先頭の 1 行を beta、gamma delta に書き込むという複数の操作における原子的実行を保証するプログラムは図 5.4のような記述になる。

また、browning.mt.cs.keio.ac.jp 上に存在するファイル alpha, beta, gamma, delta, zeta に対して、alpha から読み込んだ文字列を beta, gamma, delta に書き込む。もし beta に文字列を書き込むことができなかったら、代わりに zeta に文字列を書き込む、といった if 文が入っている場合に対しても図 5.5のような記述になる。

上記の 2 つのプログラム (図 5.4, 図 5.5) を、まず、複数の操作の原子的実行を保証しない場合の記述 (図 5.1) と比較してみる。すると、複数の操作の原子的実行を保証しない実行の記述と比較して、if 文を含んだ際も、ほぼ変わらぬ記述で表現できることが分かる。また、複数の操作における原子的実行が支援されていない場合の実行の記述 (図

```
agent EVA {

    beginLocation( browning.mt.cs.keio.ac.jp ){

        String S, T, U, V, W;


        if ( ( S = read( "alpha" ) ) == ERROR ){
                exit();
        }

        if ( ( U = write( "beta",  S ) ) == ERROR ){
            if ( ( W = write( zeta, S ) ) == ERROR )
                exit();

            if ( ( T = write( "gamma", S ) ) == ERROR ){
                recovery( "zeta" );
                exit();
            }

            if ( ( V = write( "delta", T ) ) == ERROR ){
                recovery( "zeta"  );
                recovery( "gamma" );
                exit();
            }
        }
        else {
            if ( ( T = write( "gamma", S ) ) == ERROR ){
                recovery( "beta" );
                exit();
            }

            if ( ( V = write( "delta", T ) ) == ERROR ){
                recovery( "beta"  );
                recovery( "gamma" );
                exit();
            }
        }

    }endLocation
}
```

図 5.3: 複数の操作の原子的実行が支援されていなく、if 文を含んでいる場合

```
agent EVA {

    beginLocation( browning.mt.cs.keio.ac.jp ){

        beginAtomic{

            String S = read( "alpha" );
            write( "beta",  S );
            write( "gamma", S );
            write( "delta", S );

        }endAtomic

    }endLocation
}
```

図 5.4: 複数の操作の原子的実行が支援されている場合

```
agent EVA {

    beginLocation( browning.mt.cs.keio.ac.jp ){

        String T;

        beginAtomic{

            String S = read( "alpha" );

            if ( ( T = write( "beta",  S ) ) == ERROR ){
                write( "zeta", S );
            }

            write( "gamma", S );
            write( "delta", S );

        }endAtomic

    }endLocation
}
```

図 5.5: 複数の操作の原子的実行が支援されていて、if 文を含んでいる場合

5.2, 図 5.3) と比較してみると、記述が複雑になっていない。よって、MALS エージェントを記述するプログラマの負担は小さくて済むと考えられる。

次に、ホスト上のファイルの一貫性が崩れてしまう恐れがあるという問題点が図 5.2、図 5.3にはあった。しかし、図 5.4、図 5.5では、モーバイルエージェントが、ファイルを回復させる recovery オペレーションを用いてない。ファイルの回復は、ホスト上の MALS システムが行っているので、recovery オペレーションを MALS エージェントに公開する必要がなくなる。よって、MALS エージェントによって、ホスト上のファイルの一貫性が崩されることがなくなると考えられる。

5.1.2 Schedular そのものにかかるオーバーヘッドの測定

本節では、ファイルのバックアップを行う際のオーバーヘッド、スケジューリングを行う際のオーバーヘッド、データの衝突が起きた際のオーバーヘッドを除いた Schedular そのものにかかるオーバーヘッドを測定する。そこで、

- ホスト上の 10Kbyte のファイル A の先頭の 1 行を、Schedular を通して読み込む場合
- ホスト上の 10Kbyte のファイル A の先頭の 1 行を、Schedular を通さないでみ込む場合

の 2 通りについて、実行時間を測定した。それぞれの場合を、5 回測定し、実行時間の平均を取り、その 1 回分を測定値とした。表 5.1に測定結果を示した。

Schedular を通す場合	Schedular を通さない場合
9.048 ms	8.016 ms

表 5.1: Schedular そのものにかかるオーバーヘッド

表 5.1を見ると、Schedular を通す場合と、通さない場合とで時間の差は、1.032ms となった。また、Schedular そのものにかかるオーバーヘッドは、12.87 %となった。実行時間が 12.87 %長くなってしまうが、データの一貫性が保証されるので、まずまずの数値と考えられる。

5.1.3 ファイルのバックアップを行う際のオーバーヘッドの測定

本節では、ファイルのバックアップを行う際のオーバーヘッドを測定する。

MALS エージェントは、ファイルに対する読み込み/書き込みにおける操作がアボートする際に、書き込んだファイルを書き込む前の状態に戻す。そのため、ファイルに書き込む時にバックアップをメモリ上に取る。そこで、

- ホスト上の 10Kbyte のファイル A の先頭の 1 行を読み込み、ファイル B に書き込むという操作を、Schedular を通して行う場合、つまりバックアップを取る場合
- ホスト上の 10Kbyte のファイル A の先頭の 1 行を読み込み、ファイル B に書き込むという操作を、Schedular を通さないで行う場合、つまりバックアップを取らない場合

の 2 通りについて、ファイル B のサイズを変化させた時の実行時間を測定した。それぞれの場合を 5 回測定し、実行時間の平均を取り、その 1 回分を測定値とした。図 5.6に測定結果を示した。

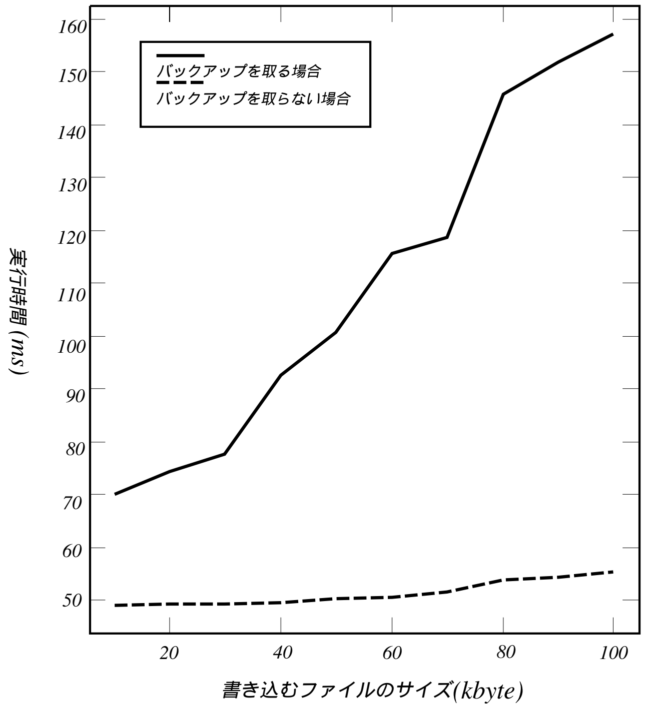

図 5.6: ファイルのバックアップを行う際のオーバーヘッド

本論文では、バックアップを取る対象のファイルを、全部メモリ上に保管するという方法を用いている。つまり、バックアップを取る対象のファイルが 100kbyte なら、100kbyte

全部をメモリ上に保管する。バックアップの方法としてファイルのコピーを用いなかった理由は、バックアップを取るためのオーバーヘッドを少なくするためである。そのため、バックアップを取るファイルのサイズが大きくなればなるほど、その分バックアップを取る時間が長くなると考えられる。

図 5.6を見ると、予想通り、バックアップを取らない場合には書き込むファイルのサイズが大きくなっても、実行時間がさほど大きくならないのに対して、バックアップを取る場合は、書き込むファイルのサイズが大きくなると、その分実行時間が大きくなっている。理想的には、ファイルのバックアップは書き込まれた部分のみを取ればよいので、今後、改良を行う必要がある。

5.1.4 スケジューリングを行う際のオーバーヘッドの測定

本節では、スケジューリングを行う際のオーバーヘッドを測定する。そこで、

- 複数の MALS エージェントが、それぞれ同一のホスト上の 10Kbyte のファイルに対して、あるファイルの先頭の 1 行を読み込み、それをあるファイルの先頭の 1 行に書き込むという操作を Schedular を通して行う場合
- 複数の MALS エージェントが、それぞれ同一のホスト上の 10Kbyte のファイルに対して、あるファイルの先頭の 1 行を読み込み、それをあるファイルの先頭の 1 行に書き込むという操作を Schedular を通さないで行う場合

の 2 通りについて、MALS エージェントの数を変化させた時、最初に実行を開始した MALS エージェントの開始時間から、最後に実行を終えた MALS エージェントの終了時間までの時間を測定した。それぞれ MALS エージェントのオペレーション中でデータの衝突はないものとする。それぞれの場合を 5 回測定し、実行時間の平均を取り、その値を測定値とした。図 5.7に測定結果を示した。

図 5.7を見ると、MALS エージェントの数が増えるにしたがって、データの一貫性を考えた場合と、データの一貫性を考えない場合との差が広がってきている。その理由として、MALS エージェントの数が増えるにしたがって、Schedular の実行時間が長くなったためと考えられる。

Schedular を通して行う場合は、Schedular を通さないで行う場合と比べて、実行時間が 20 ~ 30 %長くなってしまう。しかし、その分、データの一貫性が保証されるので、まずまずの数値と考えられる。

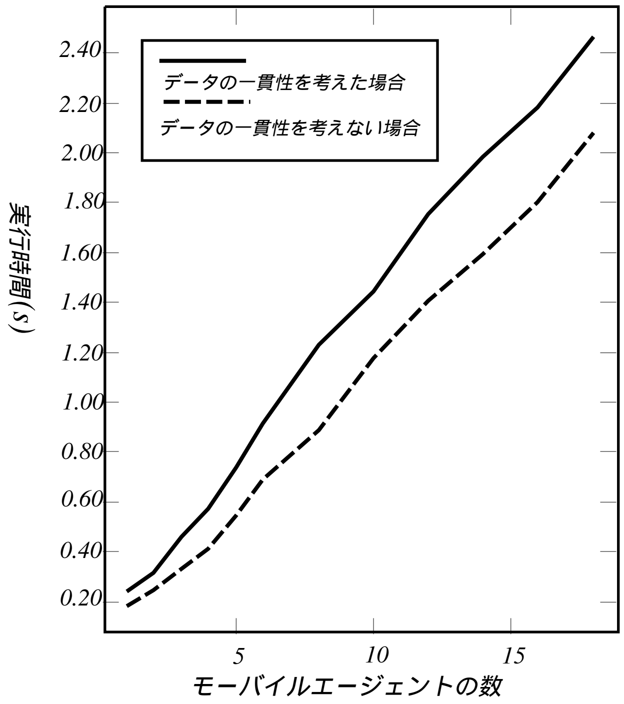

図 5.7: スケジューリングを行う際のオーバーヘッド

5.1.5 データの衝突が起きる際のオーバーヘッドの測定

本節では、データの衝突が起きる際のオーバーヘッドを測定する。まず、衝突率という言葉を説明する。

$$\frac{\text{各 }MALS\text{エージェントの全体の衝突を起こすオペレーション}}{\text{各 }MALS\text{エージェントの全体のオペレーション}} \times 100(\%)$$

を衝突率とする。例を挙げると、20 の MALS エージェントがいて、全体で 20 回衝突があったとすると、衝突率は、

$$\frac{20}{20 \times 20} \times 100 = 5(\%)$$

複数の MALS エージェントが、ホスト上の 10kbyte のファイルに対して、ファイルの先頭の 1 行を読み込むという操作を Schedular を通して 20 回行う場合について、MALS エージェントの数、衝突率を変化させた時、最初に実行を開始した MALS エージェントの開始時間から、最後に実行を終えた MALS エージェントの終了時間までの時間を測定した。

データの衝突については、実際には衝突は起きない状況の中、Schedular に偽の衝突を起こさせるようにした。衝突があった場合、MALS エージェントは 100ms スリープするとする。5 回測定し、実行時間の平均を取り、その値を測定値とした。

図 5.8に測定結果を示した。

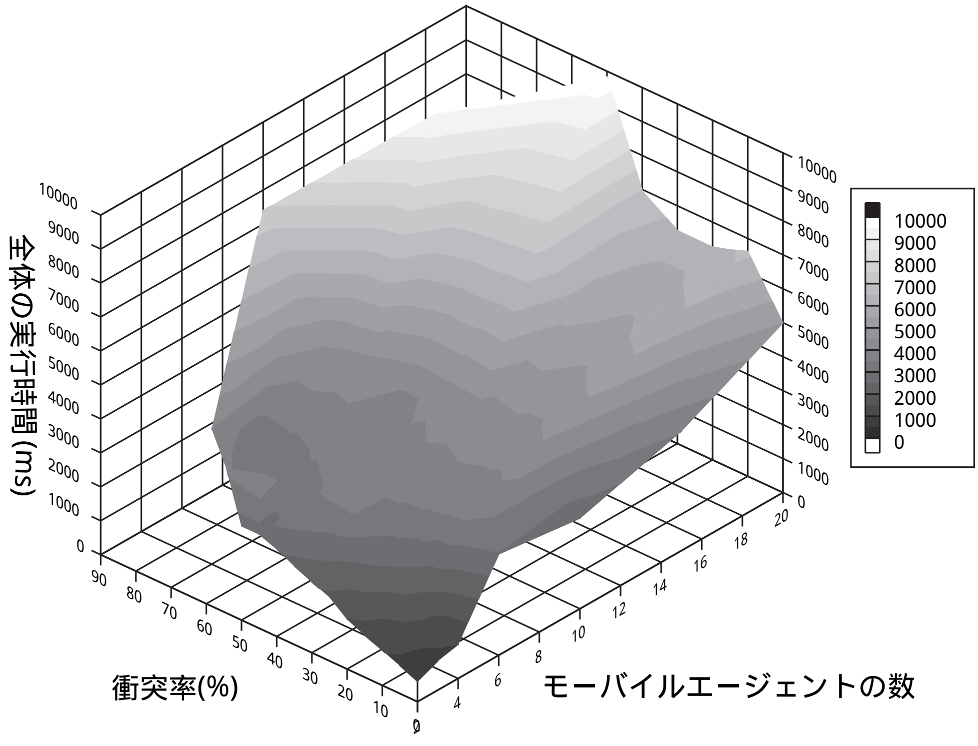

図 5.8: データの衝突が起きる際のオーバーヘッド

MALS エージェントの数、衝突率が増えるにしたがって全体の実行時間が長くなると考えれる。図 5.8を見ると、予想通り、MALS エージェントの数、衝突率が増えるにしたがって全体の実行時間が増えている。衝突率が増えるにしたがって、全体の実行時間は最大約 5 倍まで長くなってしまう。データの一貫性が保たれているが、全体の実行時間がかかりすぎなので、今後、考えていかなければならない問題である。

5.2 MALS を用いたアプリケーションの検討

インターネット上における MALS のアプリケーションについて検討する。

5.2.1 買物

インターネットは、ビジネスの分野でも用いられている。その例が買物である。現在、本や CD、また花までもがインターネット上で購入することができる。今後、現実世界で売っているさまざま品物がインターネット上で買うことができると考えられる。そこで、MALS を用いてインターネット上で買物をすることを考えると、現状の以下の 2 点の問題点が解決されると考えられる。

まず、現在、モーバイルコンピュータでインターネットを用いて買物をする際に、長時間インターネットと接続しなければならないという問題点がある。前述したように、モーバイルコンピュータは、ネットワークとの接続が不安定、通信料金が高額という理由で、ネットワークと接続する時間を出来るだけ短くしたいものである。そこで、ネットワークとの接続している時間が短い MALS を用いると、モーバイルコンピュータを用いて、インターネット上で買物をする、ということが現実的なことになる。例えば、営業で外回りをしているビジネスマンが、突然出張することになり、飛行機のチケットを予約しなければいけなくなった。その際、旅行代理店に行くことなく、自分が持っているモーバイルコンピュータを用いて、飛行機のチケットを取れる。また、MALS エージェントがチケットの予約をしてくれるので、外回りの仕事を続けることができる。

次に、複数の買物を全て行うか、全く行わないということを実現しようとすると大変な手間がかかってしまうという問題点がある。例えば、インターネット上で飛行機のチケット、フェリーのチケット、ホテルの予約をするということを考える。以上 3 つの買物をする場合、現状では、3 回アクセスして予約を取らなければならない。もし、どれか 1 つでも予約が取れなかった場合は、今まで予約した分をキャンセルしなければならない。そのため、予約を行ったホストに再びアクセスしなければならなく、大変な手間がかかってしまう。Telescript[5][6] を用いた場合も同様のことが起きてしまう。しかし、チケットを販売している全てのホストに MALS システムがあれば、MALS を用いて、ユーザが MALS エージェントを作成し、それをインターネットに送れば、チケットの予約や、キャンセルも行うことができる。

5.2.2 情報検索

現在、インターネット上の情報を検索するということが頻繁に行われている。しかし現状では、プログラマがインターネット全体を把握することができないため、データを管理するサーバが、他のホストに変わった際の対応が困難という問題点がある。

しかし、MALS を用いることによってその問題は改善されると考えられる。例を挙げて説明する。

例えば、ローカルホストにいるユーザがリモートホスト 1、リモートホスト 2、にあるデータベースを、リモートホスト 1、リモートホスト 2 の順番で情報検索を行うことを考える。しかし、リモートホスト 2 のデータベースが、リモートホスト 3 に移動していて、ローカルホストのユーザはその変化を知らないとする。もし Mosaic, Netscape, HotJava[4] などの既存のブラウザを用いると (図 5.9)、データを管理するサーバが他のホストに変わった際に、対応が困難になる。

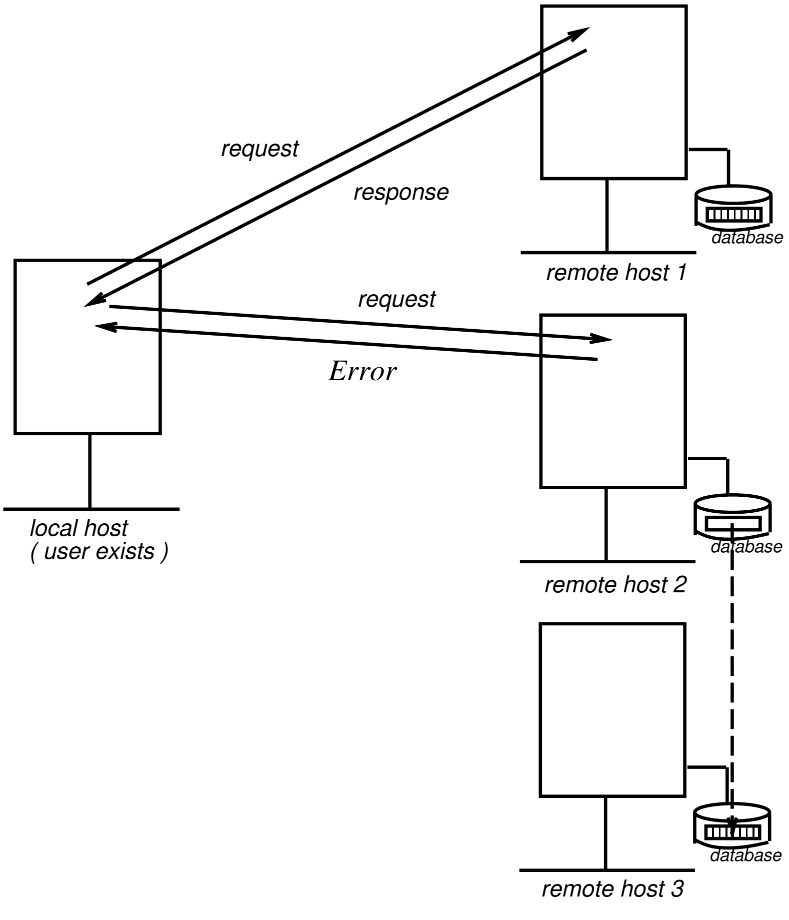

図 5.9: 既存のブラウザで情報検索をする場合

しかし、MALS を用いると (図 5.10)、MALS エージェントが、リモートホストに存在する MALS システムから、データを管理するサーバが、他のホストに変わったことを聞き出し変化に対応できる。

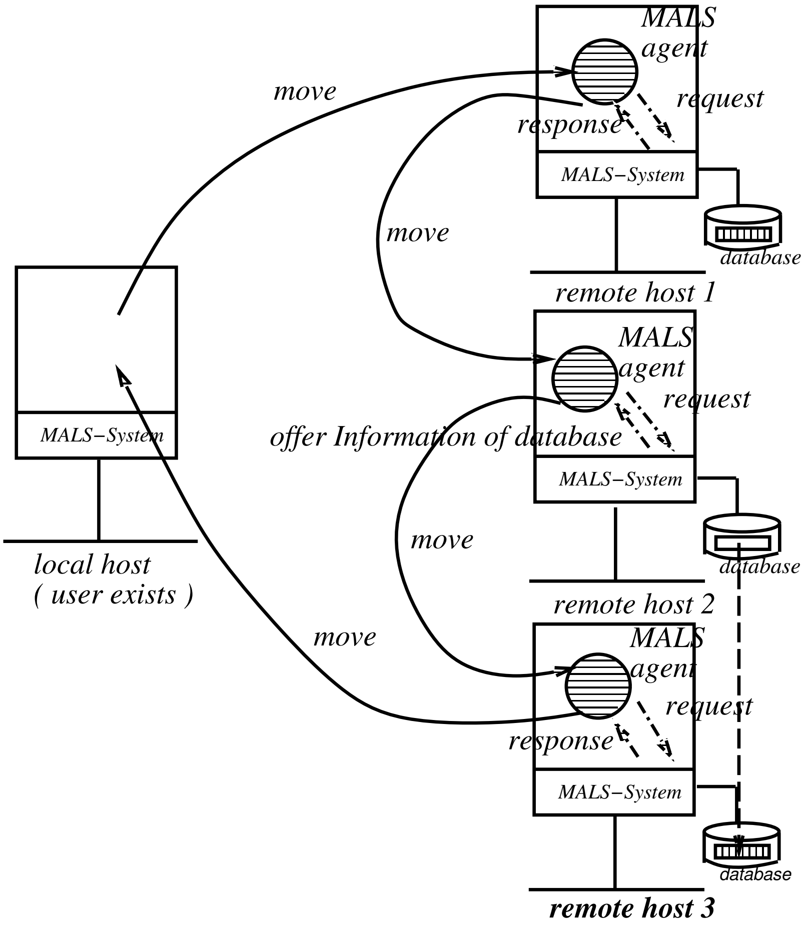

図 5.10: MALS で情報検索をする場合

第 6 章

今後の課題

6.1 分散環境下での利用

本論文のシステムでは、単一ホストにおいて原子性を保証している。モーバイルエージェントは複数のホストに動き回るので、複数のホスト上で行ったオペレーション列に対しても複数の動作に対する原子性を保証しなければならない。

6.2 MALS システム、モーバイルエージェントの全てのオペレーションの実現

本論文の実現では、リモートシステムは、リモートホスト上のファイルに対する readData/writeData オペレーションを、モーバイルエージェントは move オペレーション、read/write オペレーションを実現した。しかし、システムモデルで提案したリモートシステムの offerInformation オペレーション、request オペレーション、discern オペレーションや、モーバイルエージェントの getInformation オペレーション、putInformation オペレーションを実現していない。今後、実現されていないオペレーションを全て実現する必要がある。

6.3 モーバイルエージェントの永続性

本論文では、Java 言語を用いた。しかし Java 言語では、オブジェクトの永続性、つまりオブジェクトの状態を保存する機構が支援されてない。そのために今日作成した MALS のモーバイルエージェントも状態が保存できない。本論文で定義しているモーバイルエージェントは、状態を保持したままホスト間を移動することを前提としているので、今後、Java 言語を拡張する必要がある。

6.4 セキュリティ

MALS では不特定多数のユーザからリモートホストにモーバイルエージェントが送られる。そのため、ローカルホストからリモートホストへコンピュータウィルス機能を持つモーバイルエージェントや、暴走してしまうモーバイルエージェントなど、リモートホストに害を与えるものが送られてくることも当然考えられる。本論文の時点の実装では何も対処していないので、今後対処しなければならない問題である。

第 7 章

結論

大規模のネットワーク上で、広域に分散した計算機間にわたる作業を行う場合、また、携帯計算機をネットワークに接続する場合において、ネットワーク上の計算機を自律的に移動し、各計算機上の情報を収集し、利用しながらユーザの要求を満たすモーバイルエージェントは有効な手法であると考えられている。しかし、既存のモーバイルエージェントのためのプログラミングシステムにおいて、複数の操作の原子的実行を保証するシステムは存在していない。

そこで、本研究では、エージェント指向言語MALS(Mobile Agent Language System)の提案、実装を行った。MALSでは、言語システムが複数の操作の原子的実行を保証している。よって、プログラマは、プログラム実行中の障害を意識する必要がなくなる。このため、MALSは、プログラマの負荷を軽減し、記述性を向上させることができる。

本論文ではさらに、MALSの実装と性能評価を行ない、MALSが複数の操作の原子的実行を行う場合のオーバーヘッドの割合を明らかにした。

今後の課題を以下にまとめる。

- 分散環境化での利用
- MALSシステム、モーバイルエージェントの全てのオペレーションの実現
- モーバイルエージェントの永続性
- セキュリティ

モーバイルエージェントに原子性をサポートしたことは、今後のモーバイルエージェントを発展させる上で重要な意味を持つと考えられる。

謝辞

本研究の機会を与えて下さり、また日頃から懇切なご指導を頂きました慶應義塾大学理工学部教授 所 真理雄博士に心から深く感謝致します。

また、本研究に対し、多大なる御指導と貴重な助言を下さった同学理工学研究科博士課程 伊藤 純一郎氏、修士課程 中原 真則 氏に深く感謝致します。

そして大変お忙しい中、論文の査読をして頂き、数々の助言や御指摘を下さいました同学理工学研究科博士課程 松林 慶 氏、修士課程 猿渡 隆介 氏、成田 多良 氏に深く感謝致します。

また、本研究に関し活発な議論をして頂き、貴重な意見を下さいました本研究室m^3グループの皆様に深く感謝致します。

そして1年間苦楽を共に研究した4年生をはじめ、本研究室の皆様に深く感謝致します。

最後に、研究生活を支援してくれた両親に深く感謝致します。

1996年2月

Appendix A

MALS エージェントの記述構文

MALS エージェントの記述構文について記述例 (図 A.1) を用いて説明する。

```
agent EVA {

    beginLocation( browning.mt.cs.keio.ac.jp ){

        beginAtomic{

            String S = read( alpha );
            write( beta,  S );
            write( gamma, S );
            write( delta, S );

        }endAtomic

    }endLocation
}
```

図 A.1: MALS エージェントの記述例

MALS は Java 言語の記述構文に以下に説明する部分を加えた形になっている。

- **agent** *MobileAgentName* { **.....** }

  Java 言語では、オブジェクトを記述する際に、**class** *ObjectName* { **.....** } で囲まなければならなかったが、MALS では、MALS エージェントを記述する際に、**agent** *MobileAgentName* { **.....** } と囲まなければならない。MALS の **agent** と Java の **class** は、メソッドがないという大きな違いがある。また、*MobileAgentName* の部分は、MALS ファイルの拡張子を除いた部分と同じにしなければならない。例えば、図 A.1の場合なら、MALS ファイルが EVA.mals になる必要がある。

- **beginLocation**(*String RemoteHostName*) { **.....** }**endLocation**

  RemoteHostName で指定されたホストで行う実行を、括弧でくくられている部分に記述する。ホスト上のファイルにアクセスするためには、read/write オペレーションを用いなければならない。また、本論文ではユーザが指定できるリモートホストは 1 つなので、*RemoteHostName* で複数のホストを指定したり、**beginLocation**(*String RemoteHostName*){ **.....** }**endLocation** を複数記述することは出来ない。

- **beginAtomic** { **.....** }**endAtomic**

beginAtomic{ ..... }endAtomicでくくられた部分の特定のオペレーションに対する原子性を保証する。特定のオペレーションに該当するのは、read/writeオペレーションである。また、**beignAtomic{ ..... }endAtomic**でくくられた部分にif文、for文、while文を入れることも可能である。

- **String read**(*String filename*)

  *filename*で指定されたファイルの先頭の行を読み込んで、その行を文字列として返す。*filename*で指定されたファイルが存在しなかったらnullを返す。もし、**beginAtomic{** .....**}endAtomic**で囲まれていて*filename*で指定されたファイルが存在しなかったら、**beginAtomic{ ..... }endAtomic**で囲まれたread/writeオペレーションを全て取り消す。

- **void write**(*String filename* , *String S*)

  文字列Sを*filename*で指定されたファイルの先頭の行に書き込む。もし、**beginAtomic{** ..... **}endAtomic**で囲まれていて*filename*で指定されたファイルが存在しなかったら、**beginAtomic{ ..... }endAtomic**で囲まれたread/writeオペレーションを全て取り消す。

参考文献

[1] Goslig, J. and McGliton, H. The Java language environment: A White Paper. In *Sun Microsystems*. 1995.

[2] Sun Microsystems. The Java language: A White Paper. In *Sun Microsystems*. 1995.

[3] Sun Microsystems. The Java Language Specification: Release1.0 Alpha3. In *Sun Microsystems*, May, 1995.

[4] Sun Microsystems. The HOTJAVA Browsers: A White Paper. In *Sun Microsystems*, 1995.

[5] James E. White, Bradshaw, Jeffrey(ed.). Mobile Agents. In *AAAI Press and MIT Press*, Menlo Park, CA., 1996.

[6] General Magic,Inc. An Introduction to Safety and Security in Telescript. In *General Magic,Inc*, 1995.

[7] Antony Courtney. Phantom: An Interpreted Language for Distributed Programming. In *USENIX Association*, Conference on Object-Oriented Technologies(COOTS) - June 26-29, 1995.

[8] Fumihiro Kumeno, Yasuyuki Tahara, Akihiko Ohsuga, and Shinichi Honiden. Agent Oriented Programming Language, Flage. In *IMSA Workshop '95: Proceedings of the International Symposium on New Models for Software Architecture* , Oct, 1995.

[9] 田原 康之, 大須賀 昭彦, 永井 保夫, 本位田 真一. エージェント指向アーキテクチャの提案 - 開放型分散環境における動的スクリプト生成機能の実現 -. In 日本ソフトウェア科学会第 *12* 回大会, 1995.

[10] Tim Finin, Jay Weber, James McGuire, Stuart Shapiro, and Chris Beck. Specification of the KQML Agent-Communication Language. In *Palo Alto,CA,Technical Report EIT TR 92-04*, Jun, 1993.

[11] 清水 亮博, 大野 浩之. ネットワークワームを利用したネットワーク管理手法. In 情報処理学会第 *47* 回全国大会, Oct, 1993.

[12] 清水 亮博, 大野 浩之. ネットワークワームを利用したネットワーク管理システムのためのセキュリティ向上手法. In 情報処理学会第 *49* 回全国大会, Sep, 1994.

[13] 清水 亮博, 大野 浩之. ネットワークワームを利用したネットワーク管理システムの資源探索に関する一考察. In 情報処理学会第 *49* 回全国大会, Sep, 1994.

[14] Oren Etzioni, and Daniel Weld. A Softbot-based Interface to the Internet. In *Communications of the ACM*, July 1994.

[15] Colin G.Harrison, David M.Chess, and Aaron Kershenbaum. Mobile Agents: Are they a good idea?. In *Research Report, IBM Research Division*, .

[16] Michael R. Genesereth, and Steven P. Ketchpel. Software Agents. In *Communications of the ACM*, July 1994.

[17] Ernest A. Edmonds, Linda Candy, Rachel Jones, and Bassel Soufi. Support for Collaborative Design: Agents and Emergence. In *Communications of the ACM*, July 1994.

[18] Donald A. Norman. How Might People Interact with Agents. In *Communications of the ACM*, July 1994.

[19] Pattie Maes. Agents that Reduce Work and Information Overload. In *Communications of the ACM*, July 1994.

[20] Tim Berners-Lee, Robert Cailliau, Ari Luotonen, Henrik Frystyk Nielsen, and Arthur Secret. The World-Wide Web. In *Communications of the ACM*, Aug, 1994.

[21] 前川 守, 所 真理雄, 清水 謙多郎. 分散オペレーティングシステム. In 共立出版, 1993.

[22] P. Bernstein, V. Hadzilacos, and N. Goodman. Concurrency Control and Recovery in Database Systems. In *Addison Wesley*, 1987.

[23] Luca Cardelli. A Language with Distributed Scope. In *Computing Systems*,vol8,no 1,pp.27-59,MIT Press, 1995.

[24] Andrew Black, Norman Hutchinson, Eric Jul, and Henry Levy. Object Structure in the Emerald System. In *OOPSLA '86 Proceedings*, Sep, 1986.

[25] John K. Bennett. The Design and Implementation of Distributed Smalltalk. In *OOPSLA '87 Proceedings*, Oct, 1987.

[26] Andrew D. Birrel, and Bruce Jay Nelson. Implementing Remote Procedure Calls. In *ACM Transaction on Computer Systems*, Vol 2, No. 1, Feb 1984.

[27] Jones, A.K. and R.T.Rashid. Mach and Machmaker: Kernel and Language Support for Object-Oriented Distributed Systems. In *ACM SIGPLAN Notices*, Vol 21, No. 11, 1986.

[28] Bershad,B.N., et.al. A Remote Procedure Call Facility for Interconnecting Heterogeneous Computer Systems. In *IEEE Transaction Software Engineer*, Vol SE-13, No. 8, 1987.

[29] Henri E.Bal, Jennifer G.Steiner, and Andrew S. Tanenbaum. Programming Languages for Distributed Computing Systes. In *ACM Computing Surveys, 21(3)*, Sep. 1989.