卒業論文　1998年度(平成10年度)

計算資源共有利用を可能とする動的分散ソフトウェアツールキットの構築

指導教員
慶應義塾大学環境情報学部
徳田　英幸
村井　純
楠本　博之
中村　修

慶應義塾大学　環境情報学部
藤村　浩士

卒業論文要旨　　1998年度(平成10年度)

計算資源共有利用を可能とする動的分散ソフトウェアツールキットの構築

本研究は，広域分散環境に適合する動的分散ソフトウェアモデルを提案し，同モデルを実現するためのソフトウェアツールキットを提供する．

クライアント・サーバモデルに代表される従来の分散ソフトウェアモデルは，サブシステムが計算機に常駐して移動しないという特徴を持つ．しかし，クライアント・サーバモデルでは，サブシステムに障害が発生するとシステムの運用を継続できない．さらに，実行ファイルのメンテナンスが困難なため，多くの協力者なしには大規模運用できない．このように，従来の分散ソフトウェアモデルでは，広域分散環境を十分に活かせない．

本研究で提案する動的分散ソフトウェアモデルは，プログラムのモジュールが計算機の制約を受けず，自律的に計算機間を移動する．これにより，動的負荷分散の実現，耐故障性の向上，大規模分散性の向上，計算環境の最適化，小型携帯端末の処理支援などを実現できる．本モデルを実現するために，動的分散ソフトウェアの構築を支援するツールキットであるMeta Spaceを開発した．Meta Spaceを用いることによって，プログラマは動的分散ソフトウェアを容易に構築できる．

本論文では，まず従来の分散ソフトウェアモデルの問題点を指摘し，これを解決するための分散ソフトウェアモデルとして，動的分散ソフトウェアモデルを提案する．次に，動的分散ソフトウェアの容易な構築を可能にするMeta Spaceの概要，設計，実装について説明する．最後に，定量的および定性的評価をもとに動的分散ソフトウェア構築時の有用性を実証する．

慶應義塾大学　環境情報学部
藤村 浩士

Abstract of Bachelor's Thesis

Design and Implementation of Dynamic Distributed Software Toolkit

In this research, we propose a software model for the dynamic distribution of software components across a wide-area network, and we provide a software environment to realize the proposed model.

In the existing distributed software models, especially in the client-server model, software components remain stationary. Therefore the client-server model is not suitable for a wide-area network because a whole system stop working due to the trouble in the subsystem, and it takes too many system administrators to maintain executable files on a large scale.

In the dynamic distributed software model, software modules can independently migrate among computers. This feature enables dynamic distributed software to have the advantage of dynamic load balancing, fault tolerance, scalability, optimization of computing environment, and the processing support for small-size portable devices. Moreover, we designed and implemented a dynamic distributed software toolkit named "Meta Space". By the toolkit, programmers can easily construct dynamic distributed software whose modules can independently and transparently migrate among distributed computers.

In this thesis, we analyze the problems of existing distributed software models, and propose a dynamic distributed software model as the new resolution to the problems. Then, we describe the overview of Meta Space and explain the design and implementation. We finally discuss the usability of Meta Space based on the result of qualitative and quantitative evaluations.

Hiroshi Fujimura

Faculty of Environmental Information
Keio University

目次

図 目 次

表 目 次

第1章

序論

1.1 本研究の背景

今日，計算機の処理性能は著しい発達を遂げている．1971年から1997年の26年間でマイクロプロセッサに集積されるトランジスタの数は，インテル社の製品を例にあげると，4004プロセッサの2,300個からPentiumIIプロセッサの750万個まで，3,200倍以上に増大している[1]．このようなチップの大規模集積化は，十分な処理能力を持った計算機の価格を一般家庭の手の届く範囲にまで押し下げ，計算機を広く普及させる原動力となった．加えて，1980年代にはまだごく一部の研究者にしか利用されていなかったインターネットが，一般ユーザにまで普及し始めた．インターネットに関して調査を行なっているNetwork Wizards社は，世界中でインターネットに接続している計算機が，1998年7月の時点で36,700,000台以上に達したと発表した[2]．このような広域分散環境の発達により，インターネットで接続された膨大な数の遠隔計算機に対して，プログラムを分散し大規模な分散計算を行なわせる環境が整いつつある．さらに，マルチプラットフォームを実現したJava言語[3]の登場により，異なるアーキテクチャの計算機へJavaプログラムを移送し，実行できるようになった．これにより，小型携帯端末上のJavaプログラムを高性能計算機へ移送し，高速に実行するなどの計算環境の最適化が可能になった．

ところが，クライアント・サーバモデルに代表される従来の分散ソフトウェアモデルによって構築されるシステムの多くは，同一計算機にサブシステムが常駐し，計算機間を移動しない．このような分散ソフトウェアモデルには，いくつか問題点が存在する．第一の問題点は，プログラムの実行に先立って，ユーザが各計算機上に実行ファイルを配置する必要があるため，多くの協力者なしには大規模運用できない点である．第二の問題点は，計算機やネットワークの障害発生時に，サブシステムを移送して正常な計算機上で実行再開できない点である．第三の問題点は，サブシステムをより高性能な計算機へ移送して，実行するなどの計算環境の動的最適化が行なえない点である．以上から，従来の分散ソフトウェアモデルでは，広域分散環境を十分に活せないと言える．

本研究では，これらの問題を解決するための広域分散環境に適合する分散ソフト

ウェアモデルを提案する．

1.2　本研究の目的と特徴

本研究は，広域分散環境に適合する分散ソフトウェアモデルを提案し，同モデルを実現する環境の提供を目的としている．

本研究では，インターネット上の計算機を共有資源として利用できる分散基盤環境と，その上で動作する分散ソフトウェアの構築を支援するツールキットから成るMeta Spaceを開発した．Meta Spaceを用いることによって，プログラマはプログラムのモジュールが異なる計算機に，動的かつ位置透過的に移動する分散ソフトウェアを，容易に作成できる．Meta Spaceによって実現される分散ソフトウェアの概念を図1.1に示す．Meta Space が提供する分散基盤環境は，マルチプラットフォームを実現するJavaの実行環境であるJava Virtual Machine[4](以下JVMと呼ぶ)上に構築される．Meta Spaceを用いて構築された分散ソフトウェアは，Meta Spaceが提供する分散基盤環境上で動作し，ソフトウェアのモジュールが計算機間を位置透過的に移動可能となる．

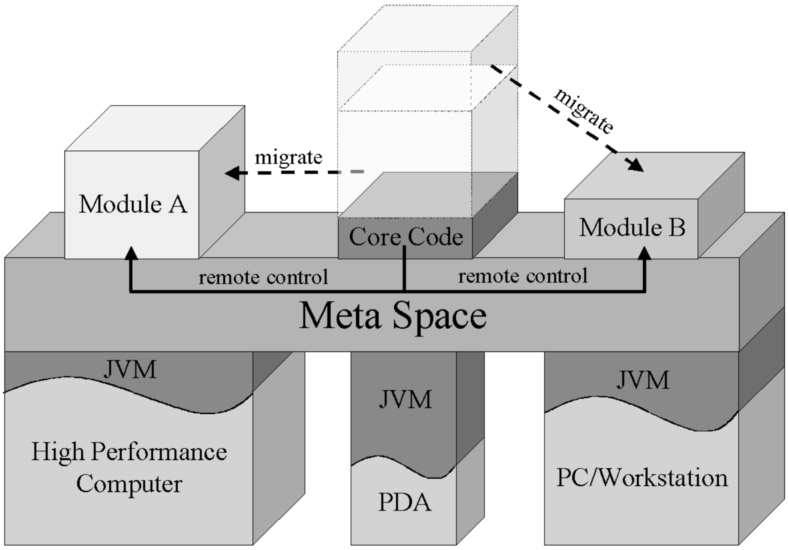

図 1.1: Meta Spaceと動的分散ソフトウェアの概念

1.3　本論文の構成

本論文では，第2章で従来の分散ソフトウェアモデルの問題点を考察した上で，これを解決する新たな分散ソフトウェアモデルとして動的分散ソフトウェアモデルを提案する．

第3章では，動的分散ソフトウェアツールキットの機能を検討し，これに基づいて既存の分散ソフトウェア構築ツールの問題点を明らかにする．

第4章では，動的分散ソフトウェアを容易に構築するためのツールキットであるMeta Spaceの概要，全体構成，動作手順，設計方針，各サブシステムの設計について述べる．

第5章では，Meta Spaceプロトタイプシステムの実装環境，各サブシステムの実装の詳細を述べる．また，Meta Spaceを用いた動的分散ソフトウェアの構築方法についても述べる．

第6章では，実装したプロトタイプシステムの定量的評価と評価結果の考察，類似研究との機能比較を行なう．

第7章では，全体としての結論を述べ，また今後の課題を明らかにし，本論文を締めくくる．

第 2 章

動的分散ソフトウェアモデルの提案

本章では，広域分散環境に適した動的分散ソフトウェアモデルの提案を行なう．

まず，本論文が想定する広域分散環境について定義する．次に広域分散環境下で動作する従来の分散ソフトウェアモデルについて考察し，その問題点を明らかにする．さらに，この問題点を解決すべく本論文で提案する動的分散ソフトウェアモデルについて説明し，その利点について述べ，動的分散ソフトウェアモデルが広域分散環境に適合するモデルであることを示す．

2.1　広域分散環境

本研究で想定する広域分散環境とは，インターネットに接続した多種多様な計算機群で構成された環境を指す．本研究で想定している広域分散環境の特徴と問題点を以下に示す．

大規模性 多数の計算機が接続されている．

多様性 PDA，PC，大型計算機など，多様なアーキテクチャを持った計算機から構成される．

低信頼性 計算機間は切断の可能性があるネットワークで接続されている．

2.2　静的分散ソフトウェアモデル

分散ソフトウェアとは，異なる計算機上で動作するサブシステム間の協調動作によって全体の機能を実現するソフトウェアである．システムの一部の機能を提供するサブシステムはプロセスとして実装され，サブシステム間の通信はソケットライブラリを用いるか，Remote Procedure Call (以下，RPC と呼ぶ) [6] によってリモート手続き呼び出しの形で行なわれるのが一般的である．ソケットや RPC によって作成される従来までの分散ソフトウェアの特徴は，サブシステムが計算機間を移動しない点である．本論文では，このような特徴を持つ分散ソフトウェアを静的分散ソフトウェアと呼ぶ．

クライアント・サーバモデルで構築されたソフトウェアの多くは，静的分散ソフトウェアに該当する．静的分散ソフトウェアの利点は，実装の容易さにある．各サブシステムが動作する計算機は常に固定しているため，プログラマはサブシステム間の協調動作を容易に記述できる．しかし，いくつか問題点が存在する．

第一の問題点は，耐故障性が低い点である．ネットワークの障害や計算機自体の障害などにより，一部のサブシステムが利用不能になると，分散ソフトウェア全体としての機能が損なわれる．

第二の問題点は，実行ファイルのメンテナンスが困難な点である．システムの起動に先立って，システム管理者が実行ファイルを配置せねばならない．このような作業はソフトウェアが大規模に分散されるほど，システム管理者の負担が増大するため，静的分散ソフトウェアは大規模分散実行に適さない．

以上の考察から静的分散ソフトウェアは，広域分散環境の利点である大規模性および多様性を活かし，課題である低信頼性を補うことはできないと言える．

2.3　動的分散ソフトウェアモデル

静的分散ソフトウェアモデルが持つ問題点を解決すべく，本論文では広域分散環境に適合する分散ソフトウェアモデルとして，動的分散ソフトウェアモデルを提案する．

動的分散ソフトウェアモデルとは，プログラム中の各モジュールが計算機に常駐せず，最適な実行環境を求めて自律的に計算機間を移動するモデルである．したがってプログラムは，必要に応じて各モジュールに分割され，そのモジュールにとって最適な計算機上で実行される．動的分散ソフトウェアモデルの利点を以下に示す．

動的負荷分散の実現

計算資源を必要とするモジュールをプログラムから分離して，異なる計算機上で実行させることにより，動的に負荷分散を実現できる．しかも分離したモジュールは実行中であっても計算機間を自由に移動できるため，最適な実行環境の計算機を求めて移動を繰り返すことができる．

耐故障性の向上

ネットワークの断絶や，計算機自体の障害により，接続先のモジュールが利用できない場合，新たなモジュールを異なる計算機に移送して処理を継続できる．これにより，モジュールの移送先計算機が存在する限り，システムの運用を継続できる．

大規模分散性の向上

モジュールの実行に必要なファイルは動的に移送されるため，モジュールが分散する計算機の台数が増加しても，システム管理者の負担は増大しない．したがって，例えばインターネット上の10000台の計算機で分散並列実行されるソフトウェアを作成しても，少人数で運用できる．

計算環境の最適化

例えばマルチメディア処理モジュール，浮動小数点演算モジュールなどの特色を持った処理を，それぞれの処理を得意とするCPUへ移動させ，高速に処理できる．

小型携帯端末の処理支援

各モジュールを異なる計算機で分散実行することで，移送元のプログラムのメモリ消費量を抑制できる．これにより，実メモリの少ない計算機でも高度なプログラムを実行できる．

以上の考察から動的分散ソフトウェアモデルは，静的分散ソフトウェアモデルに比べて優れた特性を持つことがわかる．表2.1に，従来までの静的分散ソフトウェアモデルと動的分散ソフトウェアモデルの比較を示す．動的分散ソフトウェアモデルが持つ特徴は，広域分散環境の特徴である大規模性および多様性を活かしつつ，課題である低信頼性を補うものである．したがって，動的分散ソフトウェアモデルは，広域分散環境に適合する分散ソフトウェアモデルと言える．

2.4　本章のまとめ

本論文で想定している広域分散環境は，インターネットに接続した多種多様な計算機群で構成された環境を指す．このような広域分散環境は大規模性，多様性という特

表 2.1: 静的分散ソフトウェアモデルと動的分散ソフトウェアモデルの比較

項　　目	静的分散ソフトウェアモデル	動的分散ソフトウェアモデル
実装の容易さ	○	△
動的負荷分散	×	○
耐故障性	×	○
大規模分散性	×	○
管理コスト削減	×	○
計算環境の動的最適化	×	○
省メモリ	×	○

徴を持ち，信頼性が低いという問題を持つ．

本論文では，広域分散環境下で動作する従来の分散ソフトウェアモデルを，静的分散ソフトウェアモデルと呼ぶ．静的分散ソフトウェアモデルはサブシステムが同一計算機上に常駐して移動しないという特性を持つ．したがって静的分散ソフトウェアは，サブシステムに障害が発生すると，システム全体としての機能が損なわれるという問題点が存在する．また，システム管理者が実行に先立って実行ファイルを配置する必要があり，メンテナンスが困難という問題点も持つ．したがって従来型の静的分散ソフトウェアモデルでは，広域分散環境の特徴である大規模性および多様性を活かせず，また課題である信頼性を補えない．

一方で，本研究が提案する動的分散ソフトウェアモデルは，モジュールが動的かつ自律的に計算機間を移動するため，広域分散環境の特徴である大規模性および多様性を活かしつつ，課題である信頼性を補える．以上から，本研究が提案する動的分散ソフトウェアモデルは広域分散環境に適合する分散ソフトウェアモデルである．

次章では動的分散ソフトウェアを効率的に構築できる動的分散ソフトウェアツールキットについて述べる．

第 3 章

動的分散ソフトウェアツールキット

前章では，動的分散ソフトウェアモデルが広域分散環境に適合することを述べた．しかし動的分散ソフトウェアの構築には，モジュールの自律的移動，実行ファイルの移送など複雑な機構を必要とするため，プログラマの負担は大きい．したがって，プログラマの負担を軽減するために，動的分散ソフトウェアの構築を支援するツールキットが必要である．

本章では，動的分散ソフトウェアツールキットに必要とされる機能について述べる．次に，既存の分散ソフトウェア構築ツールを例示し，既存の各ツールで動的分散ソフトウェアを構築する際の問題点を明らかにする．

3.1 動的分散ソフトウェアツールキットの機能

動的分散ソフトウェアツールキットとは，動的分散ソフトウェアの構築を支援するコマンドおよびライブラリから成るソフトウェア群である．動的分散ソフトウェアツールキットを提供することにより，プログラマは高度な動的分散ソフトウェアを効率的に構築できる．動的分散ソフトウェアツールキットが提供すべき主要機能を以下に示す．

実行ファイルの自動転送機能

> 分散するモジュール (以下，分散モジュールと呼ぶ．) の実行に必要となるファイルを，分散モジュールの移動先計算機へ自動的に転送できる．

分散モジュールの自動復旧機能

> 移動先計算機に障害が発生した場合に，分散モジュールを新たな計算機へ移動させ，運用を継続できる．

分散モジュールの遠隔制御機能

> 遠隔に存在する分散モジュールを必要に応じて制御できる．

以上の3つの機能は，全ての動的分散ソフトウェアが共通して備えるべき必須機能である．したがって，動的分散ソフトウェアツールキットによって提供することで，プログラマの負担を軽減できる．この他に，ソフトウェアの記述を簡便にするための機能として以下の2つがあげられる．

分散モジュールの遠隔手続き型制御機能

> プログラマから煩雑な制御メッセージ通信を隠蔽し，分散モジュールの制御を遠隔手続き呼び出し[1]で制御できる．これより，分散モジュールの制御部分の記述を簡便化できる．

分散モジュールの位置透過的記述機能

> 分散モジュールの移動をプログラマから隠蔽することにより，プログラマは分散モジュールの位置に依存しないソフトウェアを記述できる．

3.2 関連システム

分散ソフトウェア構築ツールを用いることで，プログラマは分散ソフトウェアを効率的に構築できる．しかし，既存の分散ソフトウェア構築ツールは，静的分散ソフトウェアモデルを前提として設計されているため，動的分散ソフトウェアの構築には適さない．本節では，既存の分散ソフトウェア構築ツールとして，ソケット，Remote Procedure Call，Object Request Broker，モーバイルエージェントシステムについて示し，各ツールを用いて動的分散ソフトウェアを構築する際の問題点を明らかにする．

---

[1] ローカルの手続きを呼び出すことによって，遠隔計算機上に存在する手続きを呼び出すこと．詳しくは3.2.2項で解説．

3.2.1 ソケット

バークレー版UNIXで開発されたソケット[5]は，アプリケーションに対して，OSのネットワーク入出力機構へのインタフェースを提供する．ソケットのデータ入出力操作は`read()`，`write()`システムコールによる入出力を基本パラダイムとして設計されている．これによりプログラマは，ソケットに対するデータ入出力を，ファイルに対するデータ入出力と同様の操作で記述できる．またソケットは，様々な通信プロトコルをサポートするが，アプリケーションから通信プロトコルを隠蔽するため，プログラマは通信プロトコル透過な記述ができる．図3.2にソケット通信のモデル図を示す．

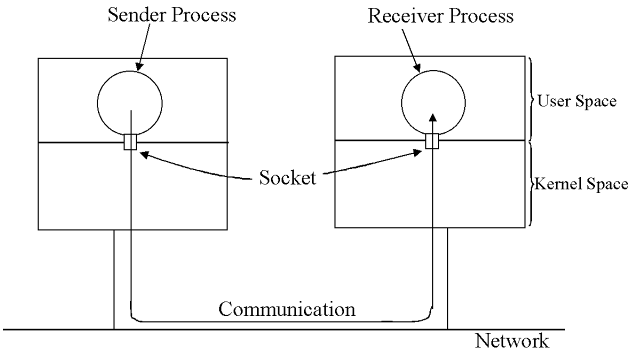

図 3.1: ソケット通信のモデル図

3.2.2 Remote Procedure Call

Remote Procedure Call (以下，RPC)は，遠隔計算機上に存在する手続きをローカルの手続きとほぼ同一の操作で呼び出すことを可能にする．この方法はBirrellとNelsonによる論文[6]ではじめて発表され，データ入出力をプログラマから隠蔽する方法として用いられるようになった．RPCの実装には，Open Network Computing (ONC)[7]，Courier[9]，Network Computing Architecture (NCA)[8]などがある．

RPCのモデル図を図3.2に示す．通常RPCには，呼び出し側であるクライアントの手続きと，呼び出される側であるサーバの手続きとの間に，両者の仲介を行なうクライアントスタブとサーバスタブが介在する．RPCによる一連の遠隔手続き呼び出しは以下の手順で行なわれる．

1: アプリケーションによってクライアント手続きが呼ばれると，クライアント手続きはクライアントスタブへ引数を渡す．

2: クライアントスタブは引数を，ネットワークを通じてサーバスタブへ送信する．

3: 実際のサーバの手続きを呼び出すために，サーバスタブはクライアントスタブから受信した引数と共にサーバの手続きを呼び出す．

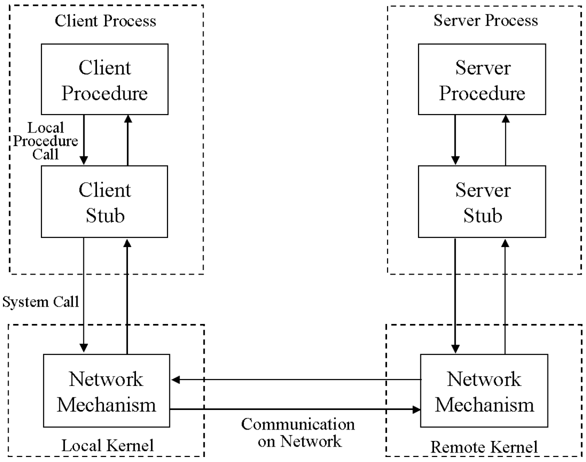

図 3.2: RPCのモデル図

4: サーバスタブはサーバ関数から返り値を受け取り，これをクライアントスタブへ送信する．

5: クライアントスタブは，返り値を受信し，これをクライアント手続きの返り値として，クライアント手続きに戻る．アプリケーションにとっては，通常のローカルの手続き呼び出しから戻ったように見える．

3.2.3 Object Request Broker

Object Request Broker (以下，ORB)はネットワークが介在するオブジェクト間の通信メカニズムを提供する．ORBを用いることにより，プログラマは遠隔のオブジェクトを遠隔手続き呼び出しによって制御できる．ORBの実装には，Java RMI[10]，HORB[11]，またObject Management Group (OMG) によって作成され，ORBの標準仕様となったCORBA[12]などがある．ORBの実現方法はRPCと類似しており，ローカルオブジェクトとリモートオブジェクトの間に，クライアントスタブとサーバスタブが介在して，両者の通信を仲介する．

3.2.4 モーバイルエージェントシステム

モーバイルエージェントシステムは，エージェントと呼ばれるプログラムがネットワーク上の計算機を移動しながら，情報の収集やサービスの提供を行なうシステムである．電子市場や分散情報検索，個人秘書エージェントなどの基礎技術として研究されている．モーバイルエージェントシステムの実装には，Aglets[13]，AgentSpace[14]などがある．モーバイルエージェントのモデル図を図3.3に示した．

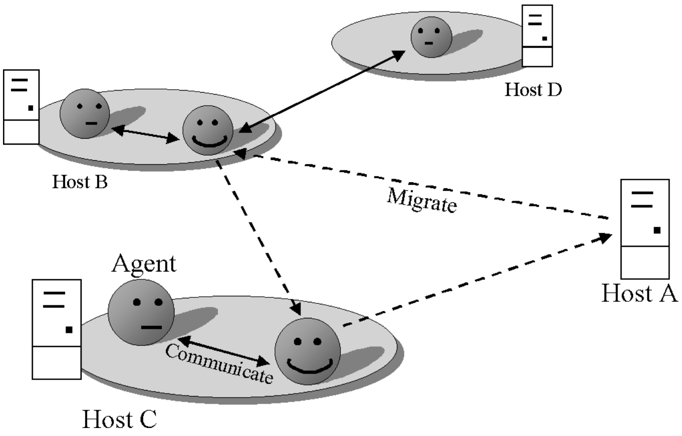

図 3.3: モーバイルエージェントのモデル図

表 3.1: 既存ツールの機能提供状況

	ソケット	RPC	ORB	エージェント
動的分散ソフトウェアに共通する必須機能				
実行ファイルの 自動転送機能	×	×	一部のシステムで提供	○
分散モジュールの 自動復旧機能	×	×	×	一部のシステムで提供
分散モジュールの 遠隔制御機能	○	○	○	一部のシステムで提供
ソフトウェアの記述を簡便にする機能				
分散モジュールの 遠隔手続き型制御機能	×	○	○	一部のシステムで提供
分散モジュールの 位置透過的記述機能	×	×	×	○

3.3 関連システムにおける問題点

既存の分散ソフトウェア構築ツールを用いることにより，静的分散ソフトウェアを容易に構築できる．しかし，これらのツールは静的分散ソフトウェアの構築を前提として設計されているため，動的分散ソフトウェアの構築に必要な機能は提供されていない．第3.2節で紹介した既存の各種ツールについて，動的分散ソフトウェア構築に必要となる機能の提供状況を表3.1に示す．

表3.1より，動的分散ソフトウェア構築ツールとして最も有利なのは，モーバイル

エージェントシステムである．実行ファイルの自動転送機能と，分散モジュールの位置透過的記述機能を，完全に提供している．しかし，分散モジュールの自動復旧機能，分散モジュールの遠隔制御機能，分散モジュールの遠隔手続き型制御機能は，多くのモーバイルエージェントシステムで提供されていない．多くのモバイルエージェントシステムでは，コールバックメソッド[2]と呼ばれる特定のメソッドが操作インタフェースとしてあらかじめ定義されている．プログラマはこのメソッドに分散モジュールが行う処理を記述することで，モーバイルエージェントを分散モジュールとして応用できる．しかし，コールバックメソッドが呼び出されるのは移動直後に限られるため，アプリケーションが必要に応じてエージェントを実行できない．また，分散モジュールの操作インタフェースは特定のコールバックメソッドに限られるため，プログラマは粒度に応じた操作インタフェースを独自に定義できない．これらの制約が存在するため，モーバイルエージェントシステムは動的分散ソフトウェアの実現には適さない．

表3.1の機能比較の結果，動的分散ソフトウェア構築ツールが提供すべき機能を全て提供する既存のツールは存在しない．これらのツールを用いて動的分散ソフトウェアを構築する場合，ツールが提供していない機能はプログラマが独自に実装する必要がある．したがって，動的分散ソフトウェアの効率的な構築を支援するツールキットを提供する意義は大きい．

3.4 本章のまとめ

動的分散ソフトウェアは構築時に，分散モジュールの移動，実行ファイルの移送など複雑な機構を必要とするため，プログラマの負担は大きい．したがって，プログラマの負担を軽減するために，動的分散ソフトウェアの構築を支援するツールキットが必要である．

動的分散ソフトウェアツールキットが提供すべき主要機能としては，実行ファイルの自動転送機能，分散モジュールの自動復旧機能，分散モジュールの遠隔制御機能，分散モジュールの遠隔手続き型制御機能，分散モジュールの位置透過的記述機能がある．

既存の分散ソフトウェア構築ツールとして，ソケット，RPC，ORB，エージェントなどがある．しかしこれらの既存ツールは，静的分散ソフトウェアの構築を前提として設計されているため，動的分散ソフトウェアの実現には適さない．したがって，動的分散ソフトウェア構築を支援するツールキットを提供する意義は大きい．

---

[2] 移動後，直ちに呼ばれるメソッド

第 4 章

動的分散ソフトウェアツールキットの設計

既存の分散ソフトウェア構築ツールは，静的分散ソフトウェアの構築を前提として設計されているため，効率的に動的分散ソフトウェアを構築できない．

したがって，本研究では動的分散ソフトウェアの構築を効率的に行なえるツールキットであるMeta Spaceを開発した．本章では，Meta Spaceの設計について詳細に述べる．まず．Meta Spaceの設計方針，概要，全体構成，動作手順について述べる．次に設計方針に基づいて，Meta Spaceの主要サブシステムであるMPool部，MSupporter部，MClient部，MWorker部の詳細な設計を行なう．

4.1 設計方針

本研究の目的の1つは，動的分散ソフトウェアの構築を効率的に行なえるツールキットであるMeta Spaceを提供することである．これを達成し，かつアプリケーションプログラマおよび，Meta Spaceのシステム管理者の負担を軽減するために，システムの設計方針として耐故障性，大規模分散性，柔軟性，安全性，簡便性をあげる．

耐故障性

動的分散ソフトウェアでは，計算機の障害，ネットワークの輻輳や断絶といった様々な障害によって，分散モジュールに対する接続性が絶たれる可能性がある．Meta Spaceでは，これらの障害から分散モジュールを復旧することで，システムの継続的な運用を可能とする．

大規模分散性

分散対象となる計算機の台数が多いほど，大規模分散実行を必要とするソフトウェアには有利となる．Meta Spaceでは，インターネット上の不特定多数の計算機を共有計算資源として提供および利用可能とする．

柔軟性

Meta Spaceでは，個々のユーザや組織が持つ多様な運用ニーズに対応可能なシステムとするため，計算資源提供ポリシの柔軟な設定を可能とする．

安全性

インターネット上の不特定多数のアプリケーションに対して，計算資源を提供する計算機は，不特定多数の分散モジュールを受け入れて，実行せねばならない．これらの中には悪意ある分散モジュールが存在する可能性がある．Meta Spaceでは，悪意ある分散モジュールによるアタックやウィルス感染を未然に防止し，安全なシステムの運用を可能とする．

簡便性

計算資源の提供を簡便にすることで，一般ユーザから計算資源の提供を募ることが可能となる．これは分散モジュールの実行環境拡大に寄与する．また，システムの管理を簡便にすることで，多数の計算機を管理しなければならないシステム管理者の負担を軽減する．したがってMeta Spaceは，導入および管理の手間を簡便なものにする．

4.2 Meta Spaceの概要

前節で述べた設計基本方針を基にして，本研究では動的分散ソフトウェアツールキットであるMeta Spaceを開発した．

Meta Spaceは，動的分散ソフトウェアの構築を支援するためのライブラリおよびコマンドから成るソフトウェア群である．Meta Spaceを用いることにより，プログラマは高機能な動的分散ソフトウェアを効率的に開発できる．Meta Spaceの特徴は，インターネット上の多数の計算機を共有計算資源として，提供，管理，分配する機構を提供した点である．提供された計算資源（計算機）は分散モジュールの実行環境として利用可能なため，大規模分散実行を必要とする分散ソフトウェアを構築できる．

計算資源の提供は，計算機の所有者が共有計算資源として，計算資源管理サーバであるMPoolへ登録することで行なわれる．MPoolは登録された計算資源を管理し，計算資源要求に応じてアプリケーションへ分配する．

提供された共有計算資源の利用は，Meta Spaceが提供するライブラリを用いて開発されたアプリケーションによって行なわれる．プログラマがアプリケーション中の分散したいモジュールを分散可能なMWorkerとして実装する．このアプリケーションが実行されると，アプリケーションに対してMPoolから共有計算資源が割り当てられる．アプリケーションは，割り当てられた共有計算資源へMWorkerとMWorkerの実行に必要なファイルを移送する．移送完了後，MWorker はアプリケーションの要求に応じて，移送先の計算機で実行される．

4.3 全体構成

本節では，Meta Space全体の構成をハードウェア，ソフトウェア両面から述べる．

4.3.1 ハードウェア構成

Meta Spaceの動作に必要なハードウェア構成を図4.1に示す．Meta Spaceの想定する分散環境はインターネットで接続された異機種混合計算機群である．

4.3.2 ソフトウェア構成

Meta Spaceは，大きく分けてMPool部，MSupporter部，MClient部，MWorker部，Ticket部の5つのサブシステムで構成される．

MPool部

計算資源を集約するサーバプログラム．MPoolに蓄積された資源は，必要に応じてMClient（後述）に分配される．本システムでは，資源を提供する計算機をサポートホストと呼ぶ．

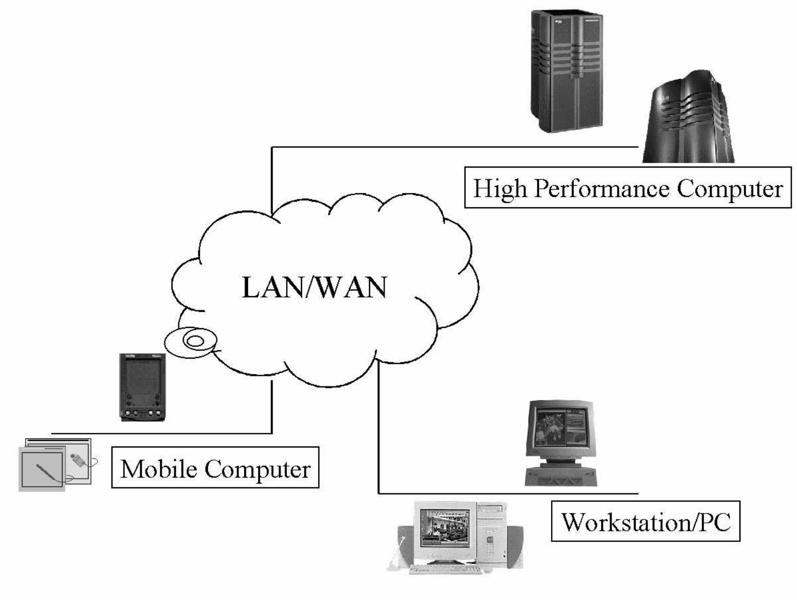

図 4.1: ハードウエア構成概念図

MSupporter 部

サポートホスト上に常駐し，自らの計算機を共有計算資源として MPool へ登録するサーバプログラム．登録はサポートホストの所有者が決定する計算資源提供ポリシに応じて行なわれる．

MClient 部

MPool に蓄積された計算資源を利用するために，アプリケーション内部でインスタンス化されるクラス．MClient は MPool に対して計算資源要求を行ない，分配された計算資源(サポートホスト) に対して MWorker(後述) オブジェクトを送り込む．

MWorker 部

Meta Space を利用するアプリケーションプログラマによって拡張され，分散処理する内容が実装されるクラス．MWorker は MClient によってサポートホスト上に転送され，アプリケーションからのリモートメソッド呼び出しによって制御される．

Ticket 部

MSupporter 内で発行され，サポートホストの情報(ホスト名，CPU タイプ，メモリ容量など) および，提供条件(最大提供時間，セキュリティ制限など) が格納されるクラス．MPool への登録時に MSupporter から渡される．Ticket は複数発行でき，それぞれに対して別々の提供条件を記述できる．

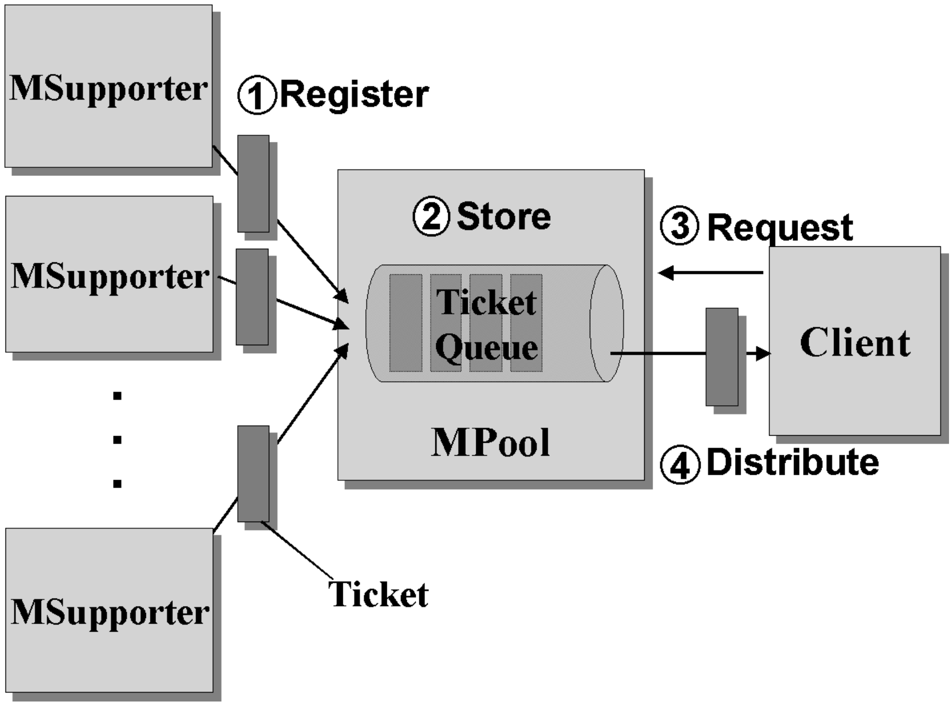

図 4.2: サポートホストの登録と計算資源要求の手順

4.4　動作手順

本項では,前項で述べたMeta Spaceサブシステム群の動作手順を述べる．本システムの動作手順は「サポートホストの登録と計算資源要求のフェーズ」，「計算資源共有利用のフェーズ」の2つに分けられる．

はじめに，サポートホストの登録と計算資源の要求の手順を図4.2に示す．

1: MSupporterは自らの計算機が計算資源として提供可能であることを示すTicketオブジェクトを作成して，あらかじめ指定されたMPoolへTicket オブジェクトを発行する．

2: MPoolはTicketオブジェクトを受け取り，MPool内部のキューに追加して登録を完了する．

3: 新たな計算資源を必要とするMClientはMPoolに対して，Ticket オブジェクトを要求する．

4: MPoolは，MClientへTicketオブジェクトを分配する．

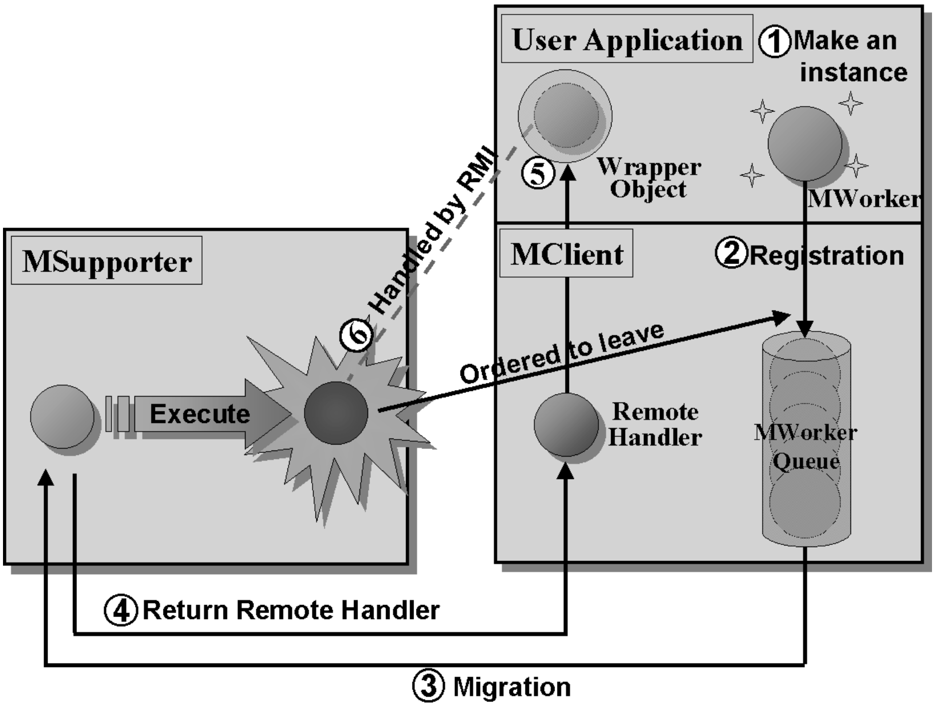

図 4.3: 計算資源共有利用の手順

これ以降はMPoolを介さずに，MClientとMSupporterで直接通信を行なう．MClientはTicketオブジェクトから取得した情報を元にMSupporterに接続し，MWorkerを送信する．

次に，計算資源共有利用の大まかな手順を図4.3に示す．

1: MWorkerオブジェクトはMeta Spaceを利用するアプリケーション内部でインスタンス化される．

2: MWorkerオブジェクトはMClient内の移送待機キューに追加され，MSupporterへ移送されるまで待機する．

3: MClientは移送待機キュー内にMWorkerオブジェクトが存在すれば，先に述べたサポートホストの登録と計算資源要求の手順3にしたがって，MSupporterに接続する．MSupporterはMClientが保持しているTicketオブジェクトの有効性[1]を確認した後，MWorkerオブジェクトを受け入れる．

4: MSupporterはMWorkerオブジェクトを受け入れると，MWorkerオブジェクトを遠隔から操作できるリモート参照をMClientに返す．

---

[1] そのTicketが偽造された物でなくかつ，提供条件を満たしていること

5: MClientは，リモート参照を内部に保持したラッパオブジェクトを生成してユーザアプリケーションに送る．

6: ユーザアプリケーションがラッパオブジェクトのメソッドを呼び出すと，ラッパオブジェクトの内部で対応するリモート参照のリモートメソッドが呼ばれ，サポートホスト上のMWorkerオブジェクトが操作される．

例外発生時

MWorkerオブジェクトの実行中に，サポートホストやネットワークの障害が発生する，あるいはMSupporterから退去命令が出されるなどの例外が発生する可能性がある．例外が発生した場合は，後述するチェックポインティング機構によってバックアップされたMWorkerが，移送待機キューへ再登録される．その後，MWorkerは手順3,4にしたがって他のMSupporterへ移送され，計算を再開する．

4.5 MPool部の設計

本節ではMSupporterから提供された計算資源を管理し，MClientへ分配するMPool部の設計について述べる．

4.5.1 機能

計算資源を管理するMPoolは，主に以下の2つの機能を提供する．

Ticketの登録

MPoolには各MSupporterから提供条件が記述されたTicketが登録される．MPoolは内部にTicketを管理するためのTicketキューを持ち，登録されたTicketをTicketキューへ格納する．

Ticketの分配

MPoolはMClientからTicketの要求を受けると，管理しているTicketの中から要求に最も適合したTicketを選出し，MClientに分配する．

Ticketの分配方式には2つの方式が考えられる．第一の方法は，MClientからのTicket要求に応じて，MPoolがTicketを分配する受動的分配方式である．第二の方法は，MPool側から分配するMClientを選出してTicketを分配する能動的分配方式である．受動的分配方式による利点は，MPoolがチケットを豊富に管理している場合，即座にチケットが手に入ることである．しかし，TicketがTicketキューにない場合は，MPoolに対して定期的にチケット要求を行う必要がある．これは，MPoolが数千のMClientからアクセスされる場合を想定すると，MClient-MPool間の通信トラフィックの増加を招く．したがって能動的分配方式を採用する．

4.5.2 識別子

全てのMPool，MSupporter，MClientには，インターネット上で一意な識別子を付けることにより，各サブシステムの特定および認証を可能とする．MPool，MSupporter，MClientで用いられる識別子のフォーマットを図4.4に示す．

//ホスト名:ポート番号/名前

図 4.4: MPoolの識別子のフォーマット

ホスト名は各サブシステムが動作している計算機である．ポート番号はそれぞれのサブシステムが内部で利用するポート番号である．また，その後に続く名前はユーザが設定ファイルの中で自由に設定できる．

図4.5に名前を`mpool-k2`として，`k2.ht.sfc.keio.ac.jp`という計算機上の1999番ポートでMPoolを起動した場合の例を示す．

//k2.ht.sfc.keio.ac.jp:1999/mpool-k2

図 4.5: MPoolの識別子例

4.5.3 MPool間でのTicket再配分機能

計算資源の提供量に対して要求量が上回った際，同一グループ内の他のMPoolに対して，Ticketの譲渡を要求できる．これによって，同一グループ内の計算資源をMPool間で最適配分し，MClientの計算資源待ち時間を減少できる．

MPoolは，PoolGroupと呼ばれるグループに加入でき，同一PoolGroup内のMPool間ではTicketの譲渡が無条件に行なわれる．また，MPoolは他のPoolGroupのMPoolに接続することも可能で，このようなMPoolをGroupGatewayと呼ぶ．GroupGatewayになったMPoolはPoolGroup間でのTicket譲渡を担当する．双方のGroupGatewayはPoolGroup間でのTicket譲渡を行なう際の提供条件[2]を取り決める必要がある．これらの設定はシステム管理者によって，MPool起動時に読み込まれる設定ファイルで行なわれる．

4.5.4 アクセスコントロール機能

不当なMClientによる計算資源不正使用および，不当なMSupporterによるMWorkerの破壊，改竄を防ぐために，MPoolはMClientとMSupporterの接続条件を設定できる．

---

[2] 提供可能な計算機種の限定，提供時間帯など

また，PoolGroupを偽り外部から不正にTicket要求を行なうMPoolに対処するため，MPoolはTicket要求を行なったMPoolが，正当なPoolGroupの構成メンバであることを審査できる．審査を行なうMPoolはGroupMasterと呼ばれ，GroupMasterはPoolGroup内で唯一の存在とする．GroupMasterの設定ファイルに記述されている条件に当てはまるMPoolのみが正当なPoolGroupメンバとして認められる．条件にはMPool識別子による直接指定，ドメイン単位での許可[3]などを記述できる．

4.5.5 運用レベル設定機能

MPoolには運用レベルとして，ノーマルモードとセーフモードの2つの運用レベルを設定できる．

ノーマルモードで運用中は，MClient，MSupporter，MPoolからの接続や，Ticketの授受は通常通り行なわれる．セーフモードに切り替わると，外部からの接続を全て拒否し，蓄積していたTicketも一時的に各発行元MSupporterへ返却される．これにより，MPool運用中にMClientおよびMSupporterの接続条件を変更する必要が生じる場合にも，MPool内部で保持しているデータの一貫性を保ちつつ，動的な設定変更が可能となる．設定変更は必ずセーフモードで行なわれ，MPoolが終了する際もセーフモードを経由して終了されるものとする．

4.5.6 Javaアプレットによる一括管理機能

システム管理者はMSupporterに対して，Javaアプレット[4]による遠隔ホストからの動的な設定変更ができる．

管理するMPoolの増加につれ，MPoolの設定変更におけるシステム管理者の管理コストは増大する．これに対処するためMPoolの内部に簡易HTTPサーバを実装することで，Javaアプレットによる設定変更を可能とする．このメンテナンス・アプレットを用いることで，遠隔ホストからの動的な設定変更が可能となる．また，GroupMasterとなったMPoolのメンテナンス・アプレットには，グループ内の全てのMPoolを一括管理する機能を提供する．これにより，MPoolが動作する全ての計算機に`telnet`などで接続して，設定変更を行なう作業からシステム管理者を解放できる．

4.5.7 まとめ

これまでに述べてきたMPoolの主な機能を表4.1に示す．

---

[3] 特定ドメイン内のMPoolであれば構成メンバになることを許可すること

[4] Javaプログラムの一種で，HTTPによってブラウザのJVMにロードおよび実行できるプログラム．

表 4.1: MPoolの主な機能

機　　能	説　　明
Ticketの管理	MSupporterからTicketを受け取り管理する．
Ticketの分配	MClientへTicketを分配する．
PoolGroup内部での Ticketの譲渡	PoolGroup内部で無条件にTicketの譲渡を行なう．
PoolGroup間での Ticketの譲渡	PoolGroup間で条件付きでTicketの譲渡を行なう．
MPoolに対する アクセスコントロール	Ticketの譲渡はGroupMasterの設定ファイルで示された 正当なPoolGroup構成メンバのみに限定する．
MSupporterに対する アクセスコントロール	Ticketの登録はMPoolの設定ファイルで指定された MSupporterのみに限定する．
MClientに対する アクセスコントロール	Ticketの分配はMPoolの設定ファイルで指定された 正当なMClientのみに限定する．
運用レベルの変更	通常運用のためのノーマルモードとメンテナンス用の セーフモードの2モードに変更できる．
アプレットによる 動的設定変更	メンテナンス・アプレットを用いて遠隔ホスト上で 動的（起動状態のまま）設定変更が行なえる．
アプレットによる 一括管理	GroupMasterのメンテナンス・アプレットからPoolGroupに 所属する全てのMPoolを一括管理できる．

4.6 MSupporter部の設計

本節では計算資源を提供し，MClientから送られたMWorkerを受け入れ実行するMSupporterの設計を行なう．

4.6.1 機能

MSupporterは以下の2つの機能を提供する．

Ticketの発行
計算資源の提供条件を記述したTicketを発行する．発行するTicketの数に制限はなく，それぞれ別々の提供条件を記述できる．

MWorkerの実行
MClientから差し出されたTicketが正当であれば，MWorkerを受け入れ実行する．

4.6.2 動作手順

以下に，MSupporter の動作手順を示す．

1: Ticket の作成
MSupporter は起動直後に，提供条件を記述した設定ファイルを読み込み，提供条件を反映した Ticket を作成する．

2: Ticket の発行
設定ファイルに記述されている MPool に対して，Ticket を発行する．

3: 待機
MClient から MWorker 受け入れ要求を受けるまで待機する．

4: Ticket 確認
MWorker 受け入れ要求の際，MClient から MWorker と共に Ticket を受け取る．この Ticket は MClient が捏造した可能性があるため，自らが発行した正当な Ticket であることを確認する．さらに，MWorker が Ticket に記述されている提供条件を満たすものかどうかを確認する．

5: MWorker の認証
MClient や MWorker が識別子を偽り，サポートホストの計算資源を不正利用することを防ぐため，MClient および受け入れる MWorker の認証を行なう．

6: リモート参照の返信
Ticket，MClient，MWorker が正当であると確認されると，MWorker を受け入れる．この時，MWorker を実行するために必要な全てのクラスファイルは，MClient から動的にロードされる．MWorker を受け入れた後，MWorker のリモート参照を MClient へ返信する．

7: MWorker の実行
ユーザアプリケーションのリモートメソッド呼び出しによって MWorker は MSupporter 上で実行される．ただし，MWorker の実行は Ticket の提供条件に制約される．

8: MWorker の終了
ユーザアプリケーション内部でラッパオブジェクトへの参照がなくなった場合は，MClient はこれを検知し，MSupporter へ通知する．通知を受けた MSupporter はリモート参照の実体である MWorker を終了（削除）する．

4.6.3 セキュリティ機構

運用形態によっては，MSupporter は不特定多数の MWorker を受け入れる可能性があるため，強固なセキュリティ機構を提供する必要がある．これにより成り済まし

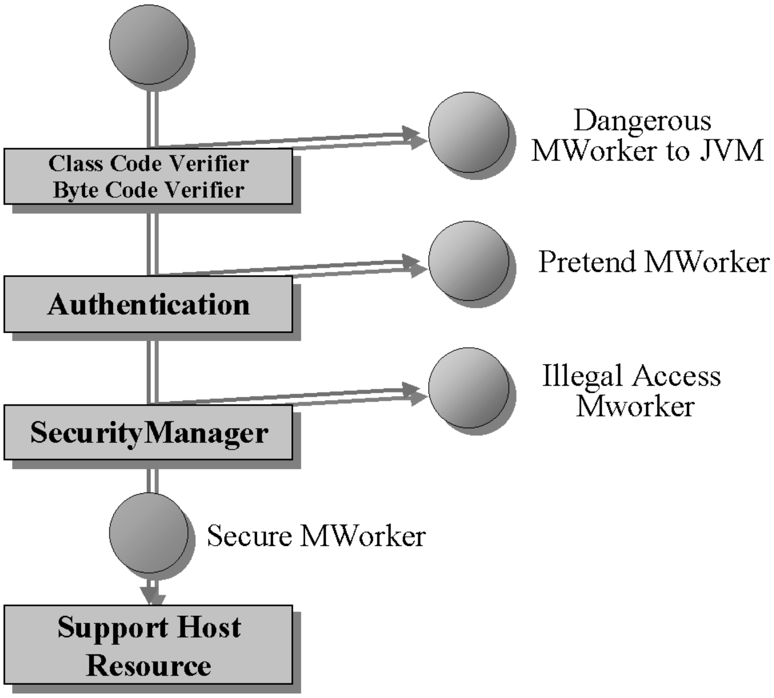

図 4.6: MSupporter のセキュリティ機構

MWorker や，サポートホスト上の資源へ不正アクセスを試みる MWorker から，サポートホストの資源を守ることができる．

本システムは図 4.6 に示したように，バイトコードベリファイア/クラスファイルベリファイア，認証機構，セキュリティマネジャによる 3 段階のセキュリティチェックを行ない，悪意ある MWorker からサポートホストの資源を守る．

クラスコードベリファイア / バイトコードベリファイア

クラスコードベリファイアおよびバイトコードベリファイアは，JVM 内部で実現されているセキュリティ機構である．クラスファイルをロードした後，クラスフォーマットや内部の整合性チェック，およびオペランドスタックに格納される型と値の一貫性チェックなどを行なう．これにより，故意またはコンパイラのバグによる不正な Java バイトコードを含む MWorker が，JVM を破壊する危険を回避できる．

認証機構

認証機構によって，サポートホストの計算資源を不正利用する成り済まし MWorker を排除できる．

セキュリティマネージャ

本システムが独自に実装するセキュリティマネージャにより，サポートホスト上

のファイルのアクセス禁止や特定のホストおよび，ポートへのアクセス禁止などを可能とする．

4.6.4 ネットワーククラスファイルローディング機能とキャッシング機能

MSupporterは，受け入れたMWorkerの実行に必要なクラスを，MClientの動作計算機からネットワーク経由で自動的にロード可能である．

これにより，クラスファイルをサポートホスト上のファイルシステムに配置するというシステム管理者の作業を省略できる．また，MWorkerが分散する計算機の台数が増加しても，システム管理者の負担は増大しないため，大規模分散実行されるソフトウェアを少人数で運用できる．

ネットワーククラスファイルローディングの問題点は，ローカルファイルをロードする場合に比べて，一般的に多くのロード時間を必要とする点である．

また，ロード時間がネットワークの状態に依存する，ネットワークの帯域を消費するという問題も存在する．したがって，ネットワークによるクラスファイルのロードは必要最小限にすべきである．MSupporterはMWorker消滅後も内部に一定時間クラスファイルを保持することで，同一クラスファイルを必要とするMWorkerによって再利用可能とする．

4.6.5 CPU最大占有値設定機能とMWorkerの優先度設定機能

計算資源提供者は，MWorkerが消費するCPUの最大占有率を設定可能である．この機能を利用すると，提供者は計算機上でCPUの全計算能力のうち，50%の計算能力を必要とする作業を行ないつつ，残り50%をMWorkerへ提供するといった運用が可能となる．

またMWorkerを複数受け入れる場合，計算資源提供者はMWorker間の実行優先度を設定できる．MWorker間の実行優先度設定機能を利用すると，自分が所属するドメイン内のMWorkerには優先度を高く設定して，それ以外のドメインのMWorkerには優先度を低くするといった設定が可能となる．

4.6.6 提供条件設定機能

MSupporterの計算資源提供条件は柔軟に設定できる．これにより，個人や組織の様々な運用ニーズにも対応可能なシステムとなる．提供条件の設定は，Ticketごとにファイルに記述し，MSupporterが起動時にこれを読み込んで発行する．柔軟で容易な設定を可能とするために，提供条件の記述は本システムで独自に設計した専用言語を用いる．MWorker受け入れ許可対象の指定および，不許可対象の指定は表4.2に示した指定方法による選択を可能とする．

計算資源の提供条件は，指定された対象に対して，複数の提供条件項目の組み合わせで設定する．表4.3に提供条件項目の一部分を示した．

表 4.2: MWorker 受け入れ許可／不許可対象の指定方法

指定方法
PoolGroup 名による指定
ドメイン名による指定
ネットワークセグメントによる指定
ホスト名による指定
MClient の識別子による指定

4.6.7 強制退去機能と強制消滅機能

MSupporter は，現在受け入れている MWorker に対して強制退去命令を発行できる．強制退去命令が発行されるのは，設定されていた提供時間を越えた場合，MWorker が不正な挙動をした場合，MSupporter がセーフモードに移行した場合などである．加えて，提供者の都合による明示的な強制退去命令の発行も可能である．これは，提供者が計算機を利用したいと思ったときに，ただちに MWorker が計算資源を明け渡すことを可能にするためである．

強制退去命令が発行されると，MWorker は制限時間内に MSupporter を退去せねばならない．制限時間を越えても依然として MSupporter 内に常駐している場合は，MSupporter は MWorker を強制的に消滅させる．また，MSupporter によって MClient との通信が断絶したと判断された場合も，MSupporter は MWorker を強制的に消滅させる．

4.6.8 まとめ

これまでに述べてきた MSupporter の主な機能を表 4.4 に示す．

4.7 MClient 部の設計

本節では，アプリケーションから本システムによって構築された分散環境を利用するためのユーザインタフェースを提供する MClient の設計を行なう．

4.7.1 機能

MClient は主に以下の役割を担当する．

Ticket の要求

MClient 内部に存在する移送待機キューに対してアプリケーションから MWorker が追加されると，MPool に対して Ticket を要求する．

表 4.3: 資源の提供条件の項目

提供条件項目
提供時間に関する条件
指定した提供時間帯に提供する
計算機が待機中の時に提供する
指定した最大時間を越えない範囲で提供する
CPUの最大占有値を越えない範囲で提供する
通信に関する条件
ローカルのポート使用を許可する(しない)
指定したホストへの接続を許可する(しない)
指定したポートへの接続を許可する(しない)
ファイルアクセスに関する制限
指定したファイル名及び，ディレクトリ名の読み込みを許可する(しない)
指定したファイル名及び，ディレクトリ名の書き込みを許可する(しない)
指定したファイル名及び，ディレクトリ名の実行を許可する(しない)
ネイティブ・コードに関する制限
新規プロセスの生成を許可する(しない)
ネイティブメソッドのロードを許可する(しない)
システム・プロパティに関する制限
指定したシステムプロパティ名への読み込みを許可する(しない)
指定したシステムプロパティ名への書き込みを許可する(しない)
スレッドに関する制限
スレッドの生成を許可する(しない)
スレッドの停止を許可する(しない)
スレッドの優先順位の変更を許可する(しない)
チケットの有効期限に関する制限
有効日時による指定
有効時間による指定

MWorkerの送信

MPoolからTicketを受け取るとTicket発行元のMSupporterへ接続し，MWorkerを送信する．

ラッパオブジェクトの生成

MWorkerの送信が完了するとMSupporterからMWorkerに対するリモート参照が返信される．MClientはこのリモート参照を内部に保持したラッパオブジェクトを生成して，アプリケーションへ返す．アプリケーションによるラッパクラスのメ

表 4.4: MSupporterの主な機能

機　　能	説　　明
Ticketの発行	提供条件が記述されたTicketを発行する．
MWorkerの実行	MWorkerを受け入れて実行する．
MClient及びMWorkerの認証	MClientとMWorkerの正当性を確認する．
セキュリティマネージャによるアクセス制限	サポートホスト上の資源へのアクセスを制限する．
ネットワーククラスファイルローディング	MWorkerの実行に必要なクラスファイルをネットワーク経由で自動的にロードする．
クラスファイルのキャッシング	MWorkerの実行に必要なクラスファイルをキャッシュして再利用する．
提供条件の設定	詳細な提供条件を設定できる
強制退去命令の発行	強制的にMWorkerを退去させる
運用レベルの変更	通常運用のためのノーマルモードとメンテナンス用のセーフモードの2モードに変更できる．
アプレットによる動的設定変更	メンテナンス・アプレットを用いて遠隔ホスト上から動的（起動状態のまま）設定変更が行なえる．
アプレットによる一括管理	指定したMPoolのアプレットからMSupporterを一括管理できる．

ソッド呼び出しはリモート参照経由でMSupporter上のMWorkerへ伝えられる．

4.7.2 動作手順

MClientの動作手順を示す．

1: MClientの初期化
MClientはMPoolの識別子とともに，アプリケーション内で初期化される．初期化後，アプリケーションによってMClient内の移送待機キューにMWorkerが追加されるまでMClientは待機する．

2: チケットの要求
移送待機キューにMWorkerが追加されると，初期化時に指定されたMPoolへ接続を試みる．接続が確立されるとMPoolにTicket要求を行なう．Ticket要求後，MPoolからTicket利用可能通知があるまで待機する．

3: MWorker送信
MPoolからTicketを受け取ると，Ticket発行元のMSupporterに接続して，M-

Workerを送り込む．

4: ラッパオブジェクトの生成
MWorkerの移送完了後，MWorkerのリモート参照がMSupporterから返される．MClientは，リモート参照を内部に保持したラッパオブジェクトを生成して，アプリケーションへ渡す．

5: MPoolへの接続が確立できない場合
手順3で指定されたMPoolと接続が確立されなければ，MClientは一定間隔で接続を試みる．その間，MWorkerはローカルの計算資源を使って実行される．MPoolとの接続が確立して，Ticketを入手できたら，ただちにMWorkerはサポートホスト上へ移送され実行される．

4.7.3 資源選定機能

分散対象となる計算資源は，アプリケーション側から選定できる．これにより，多様な目的を持つ分散ソフトウェアを開発できる．計算資源を選定するには，アプリケーションプログラマがRequestオブジェクトに資源選定方法を記述して，MWorkerオブジェクト内へ設定しなければならない．選定作業は，Requestオブジェクトと，計算資源情報が記述されているTicketオブジェクトを元にMPoolが行なう．表4.5にTicketに記述される情報の一例を示す．

表 4.5: Ticketに記述される資源情報の例

項　目
計算機の機種名
計算機のベンダ名
インターネット上のホスト名
CPUのアーキテクチャ
CPUのクロック速度
メモリの容量
ネットワークの帯域
ベンチマーク値
ロードアベレージ
オペレーティングシステムの名前
オペレーティングシステムのバージョン名
JDKのベンダ名
JDKのバージョン名

4.7.4 リモートガーベッジコレクタ機能

MClientは不要になったラッパオブジェクトを検出し，それに対応したMSupporter上のMWorkerを終了できる．これにより，MSupporterは不要になったMWorkerを削除し，計算資源を新たなMWorkerへ提供可能となる．MWorkerの終了を検知する方法は2つある．

第一の方法はMWorkerが不要になった時点で，アプリケーションプログラマが明示的にMWorkerの終了手続きを記述する方法である．終了手続きによってMSupporterへMWorker終了依頼メッセージが伝えられ，MSupporterはMWorkerを終了できる．この方法の利点は，MWorkerが不要になった時点で，MSupporter はMWorkerを消滅できる点である．しかし，プログラマが終了手続きの記述を怠った場合，不要なMWorkerがMSupporter内に常駐するという問題点がある．不要なMWorkerが常駐している間は，MSupporter内で常にMWorker一つ分の計算資源が占有されている状態が続く．

第二の方法はアプリケーション内でMWorkerが不要になったことを自動的に検知して，MSupporterにMWorker終了依頼メッセージを送る方法である．アプリケーションはラッパオブジェクトを用いて間接的にMWorkerを操作する．したがって，ラッパオブジェクトがアプリケーション内で参照されなくなった時点で，MWorkerが不要になったと見なせる．この方法の利点は，MWorkerが不要になったことを自動的に検出するので，確実にMWorkerの消去が行なわれる点である．しかし問題点は，プログラム内でラッパオブジェクトが不要になるタイミングと，ラッパオブジェクトの参照が消滅するタイミングが，必ずしも一致しないという点である．MWorkerが不要になるタイミングに対して，参照が消滅するタイミングが遅れる場合が多い．

以上の考察より，MWorkerの終了を検知する最良の方法は，両方を採用することである．これにより，プログラマ側で明示的に終了手続きを呼び出すことで，MSupporterは直ちに新たなMWorkerへ計算資源を提供できる．また，プログラマが終了手続きの呼び出しを怠っても，システムが自動的に終了処理を行なう．

4.7.5 まとめ

これまでに述べてきたMClientの主な機能を表4.6に示す．

4.8 MWorker部の設計

本節では，アプリケーションの一部分としてMSupporter上で実行される，MWorkerの設計を行なう．

4.8.1 動作手順

MWorkerの動作手順を以下に示す．

表 4.6: MClientの主な機能

機　能	説　明
Ticket要求	MPoolへ接続して，Ticketを要求する．
MWorker送信	MSupporterへ接続して，MWorkerを送り込む．
ラッパクラスの作成	MSupporter上のMWorkerを位置透過的に操作できるラッパクラスを作成してアプリケーションに渡す．
資源選定機能	Requestオブジェクトによって分散対象となる計算機を選定できる．
リモートガーベッジコレクタ	MWorkerが不要になったことを自動的に検出してMSupporterに終了を依頼する．

1: 初期化
MWorkerはアプリケーション内部で初期化され，MClient内の移送待機キューへ追加される．

2: 実行
MSupporterによって移送が完了すると，アプリケーションからのリモートメソッド呼び出しに応じて実行する．

3: 再移送
実行中にMSupporterから強制退去命令が出されると，MClientの移送待機キュー内へ移動し，再び移送されるのを待つ．

4: 終了
MClientからMSupporterへ終了依頼メッセージが送られると，MWorkerは終了させられる．

4.8.2 識別子

全てのMWorkerにはインターネット上で一意な識別子をつけることにより，MPool，MSupporterによるMWorkerの特定および認証を可能とする．図4.7は識別子のフォーマットを示している．

MClientの識別子[シリアル番号]/名前

図 4.7: MWorkerの識別子のフォーマット

MWorkerの識別子はMClientの識別子の後にシリアル番号と名前を付加したものである．シリアル番号はMClient内部でMWorkerを一意に識別するためのIDでMClient

が自動的に設定する．名前はアプリケーションプログラマがMWorkerを識別するために独自に設定できる．

4.8.3 制御インタフェース

MWorkerの制御方法には2つの方法が考えられる．

第一の方法は，移送直後，強制退去直前，終了直前などにMSupporterによって呼ばれるメソッドを決めておき，サブクラスでこれらのメソッドの内部処理を実装する方法である．このような手続きは，テンプレートメソッド[15]あるいはコールバックメソッドと呼ばれ，アプレットや多くのモバイル・エージェントシステムで用いられている．この方法の利点は，決まったメソッドの内容を実装するだけで良いためプログラマの負担が少ないこと，移送後MSupporterから直ちにメソッドが呼ばれるため効率的であること，同一計算機上のMSupporterから直接呼ばれるため，メソッド呼び出し中はネットワークが断絶していても支障がないことなどがあげられる．しかし，プログラマがMWorkerを操作するメソッドを独自に定義できない，MWorkerをインタラクティブに操作できないなどの問題点がある．

第二の方法はプログラマがMWorkerを操作するリモートメソッドを独自に定義して，MWorkerに実装する方法である．この方法の利点は，プログラマが独自にリモートメソッドを定義できること，MWorkerをインタラクティブに操作できることなどがあげられる．しかし，リモートメソッド呼び出しと実際のメソッド呼び出しの間でネットワークによる遅延が発生すること，ネットワークが断絶するとMWorkerを操作できないなどの問題点がある．

以上の考察より，両者の方法はそれぞれの欠点をうまく補完し合うことがわかる．したがって，MWorkerに対する手続き呼び出しの方法には，両者の方法を採用する．プログラマはMWorkerの性質に応じて，MWorkerの操作方法を選択できる．

4.8.4 位置透過型メソッド呼び出し機能

MWorkerが計算機間を移動した場合，ラッパオブジェクトがリモート参照を自動的に変更することにより，アプリケーションはMWorkerに対して継続的にアクセスできる．

この機能はラッパクラスによって実現される．ラッパクラスは本システムが提供するラッパジェネレータコマンドによって生成される．アプリケーションがラッパオブジェクトのメソッドを呼び出すと，内部でリモート参照の対応するリモートメソッドが呼び出される．MWorkerが異なる計算機へ移動すると，内部のリモート参照が自動的に切り替わるため，アプリケーションから位置透過的にMWorkerを操作できる．

4.8.5 チェックポインティング機能

計算機の異常など何らかの理由により，MWorkerが実行途中で中止された場合，MWorkerを異なる計算機に移送し，中止された地点から計算を再開できる．

科学計算を目的とした利用では，MWorker が行なう計算が長時間に及ぶものが多く，実行途中に強制退去命令が出されたり，ネットワークやサポートホストの障害に遭遇する可能性も高くなる．途中で中止された場合に備えて，計算の途中結果（MWorker オブジェクトのスナップショット）をバックアップしておき，中止された地点から計算を再開するチェックチェックポインティング機構を提供する必要がある．これにより，障害によって紛失した MWorker の復旧も行なえる．

チェックポインティングを実現する最も高度な方法は，コンパイラによる自動的なチェックポイントの挿入である．これは，プログラマに全く負担をかけない方法であるが，高度な実装が必要なため現在のバージョンでは見送ることにする．

第二の方法はプログラマが必要に応じて，プログラム内に明示的にチェックポインティングメソッドを挿入する方法である．実行スレッドが計算ルーチン中に挿入されたこのメソッドに遭遇すると，それまでの実行結果を含んだ MWorker オブジェクトをラッパクラス内へ待避する．この方法の利点は，プログラマの計画に沿ってチェックポインティングが行なえることである．しかし，プログラマはこの MWorker の実行が再開した際に，計算ルーチン中の既に済んでいる部分をスキップするように，アルゴリズムを工夫しておかねばならないという問題点がある．

第三の方法はラッパクラスのメソッドが呼ばれる度に，ラッパクラスが自動的にチェックポインティングを行なう方法である．この方法は第一の方法と同じくプログラマに全く負担をかけない．加えて実装も容易である．しかしチェックポイントが，ラッパオブジェクトのメソッド呼び出しの粒度でしか行なえないという問題がある．さらに複数のスレッドによって，同時にラッパオブジェクトのメソッドが呼ばれると，バックアップされた MWorker の一貫性を損なう可能性がある．

本システムでは第二及び，第三の方法を採用する．ただし第三の方法は，マルチスレッドで MWorekr が制御された場合，バックアップされた MWorker の一貫性を損なう可能性があるため，付加的な機能とする．

4.8.6　まとめ

ここまで述べてきた MWorker の主な機能を表 4.7 に示す．

4.9　本章のまとめ

既存の分散ソフトウェア構築ツールは，静的分散ソフトウェアの構築を前提として設計されているため，効率的に動的分散ソフトウェアを構築できない．したがって，本研究では動的ソフトウェアの構築を効率的に行なえるツールキットである Meta Space を開発した．

本章では，Meta Space の設計方針として，耐故障性，大規模分散性，柔軟性，安全性，簡便性を重視することを述べた．次に，この設計方針に基づいて MPool 部，MSupporter 部，MClient 部，MWorker 部の各サブシステムについて，動作手順と必要な機能につ

表 4.7: MWorkerの主な機能

機　能	説　明
MWorkerの実行	MSupporter上でアプリケーションの一部分の処理を実行する．
サポートホスト間の移動	サポートホスト間を移動し，実行を継続する．
テンプレートメソッド呼び出し機能	MWorkerの決められた手続きを実装することでMWorkerの実行処理を容易に記述できる．
リモートメソッド呼び出し機能	ユーザが呼び出す手続きを定義することでインタラクティブにMWorkerを実行できる．
位置透過型手続き呼び出し	MWorkerが計算機間を移動しても透過的に手続きを呼び出せる．
明示的チェックポインティング機能	MWorker内部にチェックポイントメソッドを明示的に記述することでチェックポインティングを行なう．
透過的チェックポインティング機能	MWorkerの手続き呼び出しの度に，ラッパクラスが自動的にチェックポインティングを行なう．

いて述べた．特に各サブシステムに必要な機能については，各種実現方式を比較検討した上で実現方式を決定した．

次章では，本章の設計を基にMeta Spaceプロトタイプの実装について述べる．

第 5 章

動的分散ソフトウェアツールキットの実装

本章では，動的分散ソフトウェアの構築を容易にするMeta Spaceプロトタイプの実装について詳細に述べる．

まず，Meta Spaceの実装環境について述べる．次にMPool部，MSupporter部，MClient部，MWorker部の実装について詳細に述べる．本章の最後では，Meta Spaceを用いたアプリケーション構築方法について，具体例と共に述べる．

5.1 実装環境

Meta Spaceの実装はJava言語を用いて行なった．実装言語としてJavaを選択した理由を以下に示す．

安全性

Java はJava Virtual Machine (JVM)と呼ばれるインタプリタによって実行される．JVMにはクラスファイルベリファイア，バイトコードベリファイア，セキュリティマネージャ，別々のネーミングスペースを持つクラスローダなど，Javaプログラムを安全に実行するための様々な機構が導入されている．これらの機構を利用することにより，外部から侵入を試みる有害なMWorkerからサポートホストを防衛できる．

マルチプラットフォーム性

Javaプログラムは JVMが動作する計算機であれば，プログラムを再コンパイルすることなく実行できる．小型携帯端末の計算処理支援を研究の成果として掲げている本研究にとって，この特性は有益である．この特性により，本システムを用いて作成された小型携帯端末用ソフトウェアの分散モジュールは，高性能計算機上へ移動して，豊富な計算資源を利用できる．

ネットワーク指向型

Javaのオブジェクトは，ネットワークを通じて遠隔計算機上へ移送できる．また，Java RMIを利用することでリモートオブジェクトに対して透過的にアクセスできる．これらの特性および機構を利用することで，計算機間を移動するMWorkerを容易に実装できる．

本システムが必要としている機能を考慮すると，これらの特徴を持つJavaを実装言語として採用することは，現時点において最良の選択肢の一つである．

問題点は，Javaの実行速度である．Javaの実行速度は，C++に比べて3 ~ 10倍遅い[16]．しかし，実行時にネイティブコードへの動的なコンパイルを行なうJITコンパイラ[17]やHotSpot[18]，Javaバイトコードの実行に最適化されたCPU[19]などの新技術を現在各社が開発しており，今後この問題は，改善されると予想する．表5.1にMeta Spaceの実装環境を示す．

表 5.1: 実装環境

項　　目	説　　明
ハードウェア	Sun Microsystems Ultra 30
オペレーティングシステム	Sun Solaris 2.6
実装言語	Java言語(Sun JDK1.2fcs)

5.2 MPool部の実装

本節ではMSupporterから提供された計算資源を管理し，MClientへ分配するMPool部の実装を行なう．

5.2.1 実装概観

図5.1にUnified Modeling Language (以下，UMLと呼ぶ) に基づくMPool部のクラス図を示す．

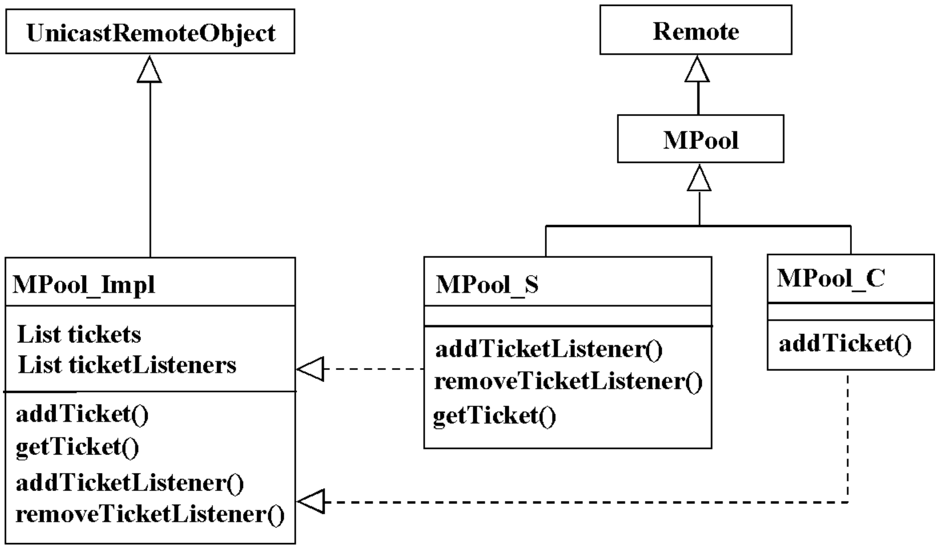

図 5.1: MPoolのクラス図

MPool_SとMPool_CはそれぞれMSupporterとMClientから呼び出されるリモートメソッドを定義したインタフェースである．MSupporterとMClientの両者から共通して呼び出されるリモートメソッドはRemoteインタフェースを継承したMPoolインタフェースで定義している．リモートメソッドをMPool_SインタフェースとMPool_Cインタフェースで別々に定義することにより，他者のリモートメソッドを誤って呼び出すトラブルを予防している．それぞれのインタフェースで定義されたリモートメソッドはUnicastRemoteObjectを継承したMPool_Implというクラスで実装される．

5.2.2 Ticketの登録機能と分配機能の実装

図5.2にTicketの登録から分配までのUMLに基づくシーケンス図を示した．MPoolのTicket登録は，MPoolのリモートメソッドであるaddTicketメソッドをMSupporterが呼び出すことで行なわれる．addTicketメソッドの引数には追加したいTicketオブジェクトと，認証を行なうためのMSupporterオブジェクトが必要である．Ticketオブ

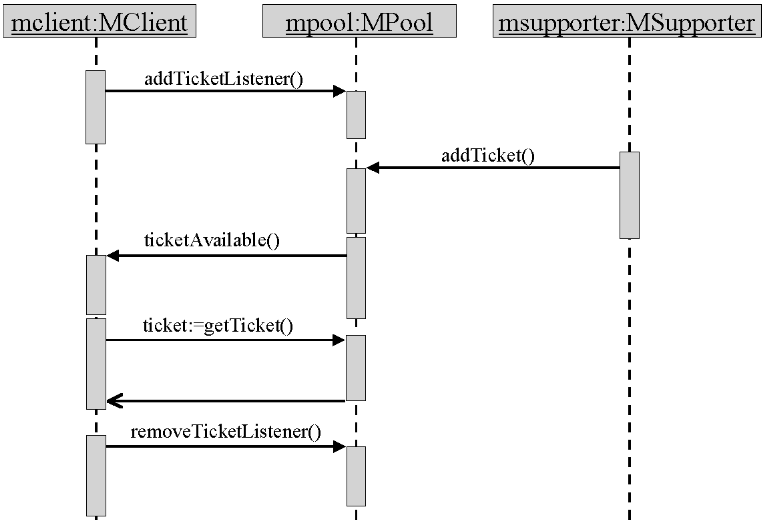

図 5.2: Ticketの登録と分配のシーケンス図

ジェクトはListクラスのインスタンスであるticketsに追加される．なお，addTicketメソッドは複数のM_Supporterから呼び出されるため，排他制御されなければならない．addTicketの実装の概要を図5.3に示す．

```
public void addTicket(MSupporter msupporter, Ticket ticket)
  throws RemoteException, RunLevelException, NotTrustedMSupporterException{

    if(runLevel == NormalMode)  //ノーマルモードならMClientの認証
      authenticate(msupporter);

    }else{                     //セーフモードなら例外発生
      throw new RunLevelException();
    }

    synchronized(tickets){     //TicketをTicketプールに追加
        tickets.add(ticket);
    }

    ticketAvailable();         //Ticketが到着したことをMClientに知らせる．
}
```

図 5.3: addTicketメソッドの実装の概要

MClientがMPoolからTicketを取得する場合，MClientはMPoolに対してTicketが必要なことを，あらかじめ通知しておく必要がある．これはMPoolのリモートメソッドであるaddTicketListenerメソッドをMClientが呼び出すことで行なわれる．addTicketListenerメソッドを図5.4に示した．addTicketListenerメソッドを呼び出したMClientは，Ticket配布待ちMClientとして，MPool_Impl内部のticketListenersリストへ追加

される．ticketListenersリストの中に，Ticket配布待ちMClientが複数存在する場合は，MPool_Implがリストの中からTicket配布候補を選択する．Ticket配布候補に選ばれると，MPoolによってMClientのリモートメソッドであるticketAvailableメソッドが呼ばれ，Ticketが配布可能であることが通知される．

```
public void addTicketListener(TicketListener listener) throws RemoteException,
                                  RunLevelException, NotTrustedMClientException
```

図 5.4: addTicketListenerメソッド

MClientはMPoolからTicketが配布可能であることが通知される（MPoolによってMClientのticketAvailableメソッドが呼ばれる）と，MPoolのリモートメソッドであるgetTicketメソッドを呼び出してTicketを取得する．引数には要求するTicketの選定方法が記述されたRequestオブジェクトと，認証を行なうためのMClient_IFのオブジェクトが必要である．全て成功すると返り値には，Ticketオブジェクトが返される．getTicketメソッドの実装の概要を図5.5に示す．

MClientは計算資源(Ticket)を必要としなくなった場合，MPoolのリモートメソッドであるremoveTicketListenerメソッドを呼ぶ．removeTicketListenerメソッドを呼び出したMClientは，ticketListenersリストから削除され，以後MPoolからTicket配布可能メッセージは届かない．

5.2.3 まとめ

表5.2に実装状況を示す．

表 5.2: MPoolプロトタイプの実装状況

機　　能	実装状況
Ticketの登録	○
Ticketの分配	○
PoolGroup内部でのTicketの譲渡	×
PoolGroup間でのTicketの譲渡	×
MPoolに対するアクセスコントロール	×
MSupporterに対するアクセスコントロール	×
MClientに対するアクセスコントロール	×
運用レベルの変更	△
アプレットによる動的設定変更	×
アプレットによる一括管理	×

```
public Ticket getTicket(MClient_IF mclient, Request request)
  throws RemoteException, RunLevelException,
  NotTrustedMSupporterException, NoTicketException{

    if(runLevel == NormalMode)  //ノーマルモードならMClientの認証
      authenticate(mclient);

    }else{                     //セーフモードなら例外発生
      throw new RunLevelException();
    }

    synchronized(tickets){
                               //提供できるTicketの引換券をMPoolが作成
        List selected = selectTicket(mclient);

                               //引換券が一枚もなければ例外発生
        if(selected.size() == 0)
            throw new NoTicketException();

                               //Requestオブジェクトが引換券の中から選定．
        Ticket ticket = reqest.select(selected);

                               //選定基準を満たす引換券が一枚もなければ例外発生
        if(ticket == null)
            throw new NoTicketException();

                               //引換券を渡して本物のTicketを得る．
        ticket = exchange(ticket);
    }
    return ticket;
}
```

図 5.5: getTicket メソッドの実装の概要

5.3 MSupporter 部の実装

本節では計算資源を提供し，MClient から送られた MWorker を受け入れ実行する MSupporter の実装を行なう．

5.3.1 実装概観

図5.6に UML に基づく MSupporter 部のクラス図を示す．

MSupporter_C と MSupporter_P は，それぞれ MClient と MPool から呼び出されるリモートメソッドを定義したインタフェースである．MClient と MPool の両者から共通して呼び出されるリモートメソッドは，Remote インタフェースを継承した MSupporter インタフェースで定義している．それぞれのインタフェースで定義されたリモートメソッドは，UnicastRemoteObject を継承した MSupporter_Impl というクラスで実装さ

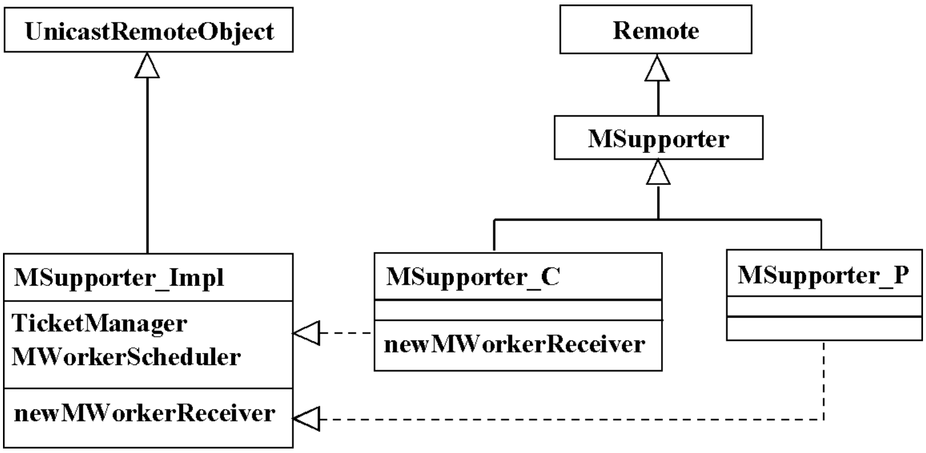

図 5.6: MSupporterのクラス図

れる．またMSupporter_Implは，Ticketの作成および発行を担当するTicketManagerと，複数のMWorker間の実行優先度やCPUの占有率を管理するMWorkerSchedulerを属性として保持する．

5.3.2　ネットワーククラスローディング機能の実装

MSupporterは，MWorkerのネットワーク経由による自動クラスローディングを実現している．ネットワーククラスローディングは，MClient_ImplによってMSupporter_Cインタフェースの newMWorkerReceiverメソッドを呼び出すことで開始される．newMWorkerReceiverメソッドを図5.7に示す．

```
public void newMWorkerReceiver(MClient_S mclient, String CodeBase)
                              throws RemoteException, RunLevelException,
                              NotTrustedMClientException, MigrateException
```

図 5.7: newMWorkerReceiverメソッド

newMWorkerReceiverメソッドは，第一引数にMClientのリモート参照，第二引数にMWorkerのロード先URLを指定する．以下にnewMWorkerReceiverの動作手順を示す．

1: newMWorkerReceiverメソッドのCodeBase引数で指定されたURLから，クラスファイルをロードするURLClassLoaderを生成し，MWorkerの受信を担当するMWorkerReceiverをロードする．

2: MWorkerReceiverは，ソケット通信によってMClientからMWorkerオブジェクトを受信する．

3: MWorkerオブジェクトの実行に必要な全てのクラスファイルは，URLClassLoaderがnewMWorkerReceiverメソッドの引数で指定したCodeBaseのURLからロードする．

4: MWorkerReceiverは，MWorkerオブジェクトを受信するとMClientのmigrationPerformedメソッドを呼び出してMWorkerオブジェクトのリモート参照を返す．

newMWorkerReceiverメソッドの第二引数に指定するCodeBaseは，インターネット上のWWWサーバやMClient内部のWWWサーバを指定できる．インターネット上のWWWサーバを指定した場合，ユーザはそのWWWサーバがアクセスできるディレクトリにMWorkerの実行に必要な全てのファイルを配置しておかねばならない．MClient内部のWWWサーバをCodeBaseに指定した場合，ユーザはクラスファイルの位置をWWWサーバの検索パスに指定する．

MSupporterがネットワーククラスローディングを行なうに際には，MWorkerの受け入れタイミングによって，次のような問題が発生する可能性がある．

1: MSupporterが，あるMClientからMWorkerAを受け入れた．

2: URLClassLoaderがMWorker Aを実行するために必要なClass Aという名前のクラスをネットワーク経由でロードした．

3: MSupporterは異なるMClientからMWorker Bを受け入れた．

4: MWorker Bも同一の名称で内容の異なるClass Aが必要だったので，URLClassLoaderは既にステップ2でロードしたClass AをMWorker Bにリンクした．

結果: MWorker BはMWorker AのClass Aを誤ってリンクしてしまい，プログラマの期待とは異なる挙動をした．

Class AはMWorker Bに対して誤ってリンクされたにも関わらず，MWorker Bに対するアクセスが可能となる．仮にClass Aが悪意のあるクラスだった場合，Class AはMWorker Bの情報改竄，盗聴，実行妨害などを行える．

本システムでは，送信元のMClientごとにURLClassLoaderを生成することにより，クラス名の衝突を防ぎ，他のMWorkerが利用するクラスを誤ってリンクする危険を排除した．MClientとURLClassLoaderの関係を図5.8に示す．図5.8において，SystemClassLoader，URLClassLoader A，URLClassLoader BがロードしたそれぞれのClassAは，クラスローダが異なるため名前が衝突することはない．

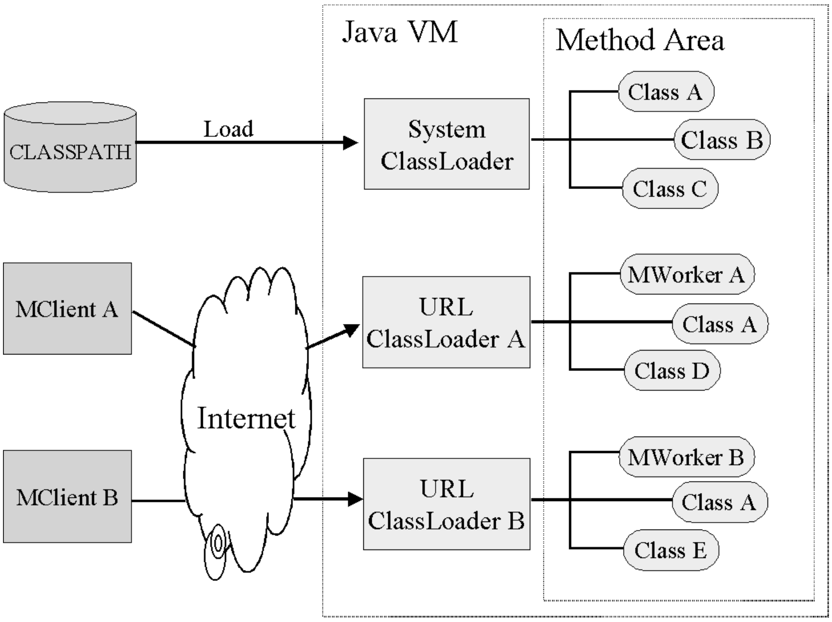

図 5.8: MClient とクラスローダの関係

5.3.3 キャッシング機能の実装

MSupporter にはクラスファイルのキャッシング機能を実装した．この機構は JDK1.2 から新しく追加された Reference 機構を利用して実現している．java.lang.ref.SoftReference オブジェクトによって参照されているオブジェクトは，他のオブジェクトから参照されなくなった後もガーベッジコレクタによる回収を極力遅らせることができる．

本システムでは SoftReference オブジェクトに URLClassLoader を参照させることで，URLClassLoader を再利用するようにしている．URLClassLoader は 2 回目以降はネットワーク経由でのクラス転送を必要としないため，迅速なロードが可能となる．

5.3.4 MWorker のスケジューリング機能の実装

MSupporter には，MWorker が消費する CPU の最大占有率を制御できかつ，MWorker 間で実行優先度に基づく実行スケジューリングを行なうことができる．MWorkerScheduler の実装にあたって，Java の仕様上いくつかの問題点が存在した．

Java プログラムの実行はオペレーティングシステムのプロセス・スケジューラによって制御されており，Java プログラムはこのスケジューリングに影響を及ぼすことはできない．したがって Java プログラムが消費する CPU 占有率の正確な制御は，ネイティブメソッドを組み込んでオペレーティングシステムのプロセススケジューリングを直接制御する必要がある．

しかし，Java プログラム内でスレッドの suspend と resume をうまく利用して，Java プログラムが消費する CPU の最高占有率を保証することは可能である．これを実現す

るアルゴリズムは，オペレーティングシステムによって毎回割り当てられるプロセス実行時間（クォンタム）の内，常に最高占有率分だけ実行し，残りの時間は全スレッドをsuspendするというものである．例えば，クォンタムが100msのオペレーティングシステムにおいて，Javaプログラムが占有するCPU時間の最高占有値を50%にしたければ，常に50msだけ実行し，残り50msは全スレッドをsuspendさせる．全スレッドをsuspendするとJavaのプロセスはブロック状態になるため，オペレーティングシステムは即座に他のプロセスへ実行権を切り替える．このとき，当該計算機上で他にプロセスが存在していなければ，Javaプログラムの CPU 占有率は50%となる．

このアルゴリズムは，Javaプログラムが消費するCPUの占有値が指定した数値を越えないことを保証できるが，問題点もある．仮にCPU占有率が50%に指定されたJavaプログラムが実行されており，他のプロセスは全てブロック状態であると仮定する．今，ユーザが50%のCPU占有時間を必要とするMPEGビデオソフトウェアを新たに起動した場合，両者の実行優先度が同じであればオペレーティングシステムはJavaプログラムが今まで消費していたCPU占有時間を100% から50%へ抑え，残り50%をMPEGビデオソフトウェアへ割り振る．その結果，本来はJavaプログラムは50%の占有率のままMPEGビデオソフトウェアと共存できるが，実際は50%から25%へ自らを抑制する．このため25%のCPUアイドル状態が発生する．MPEGビデオソフトウェア起動前と，起動後のCPU占有状態の変化の様子を図5.9に示す．

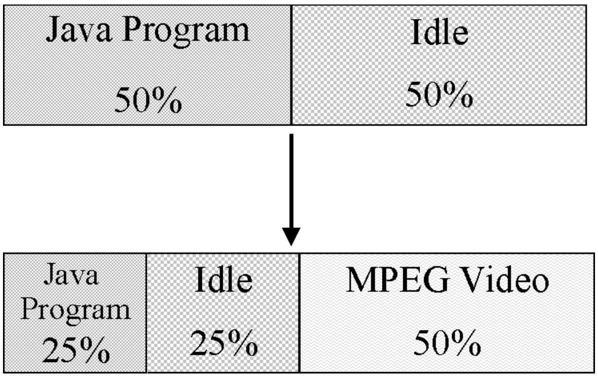

図 5.9: CPU 占有状態の変化

このような問題点はあるが，この方法はCPUの占有値が指定した値を越えないことを保証できること，Javaプログラムだけで実装可能なためプラットフォーム透過性を損なわないことなどの利点がある．したがって，MWorkerSchedulerの実装にはこのアルゴリズムを採用した．

第二の問題は，RMIの仕様に起因する問題である．MWorkerSchedulerでスレッドを管理するためには，スレッドオブジェクトをMWorkerSchedulerに登録する必要がある．しかし，MWorkerのリモートメソッドを実行するスレッドは，RMI が起動する内部スレッドであるため外部から参照できない．これを解決するために，ラッパジェネレータ（後述）でラッパオブジェクトを生成する際，MWorkerの実装クラスの各メソッ

ドにスレッド登録ルーチンが自動的に挿入されるようにした．

第三の問題はCPUの最大占有値を保証するためにMWorkerSchedulerの実装にsuspendメソッドを利用したという点である．suspendメソッドは，スレッドを同期ブロックの内部でロックオブジェクトを獲得したままsuspendさせてしまうため，デッドロックを引き起こす可能性がある．そのためJDKではsuspendメソッドの利用を推奨していない．MWorkerSchedulerはsuspendさせたThreadは必ずresumeされることを保証しているため，デッドロックの危険性はない．しかし，推奨されないsuspendメソッドを用いているため，MWorkerSchedulerは付加的な機能とした．

5.3.5 まとめ

表5.3に実装状況を示す．

表 5.3: MSupporterプロトタイプの実装状況

機　能	実装状況
Ticketの発行	○
MWorkerの実行	○
MClient及びMWorkerの認証	×
セキュリティマネージャによるアクセス制限	△
動的クラスファイルローディング	○
クラスファイルのキャッシング	○
提供条件の設定	△
強制退去命令の発行	△
運用レベルの変更	△
アプレットによる動的設定変更	×
アプレットによる一括管理	×

5.4 MClient部の実装

本節ではアプリケーションから本システムによって構築された分散環境を利用するためのユーザインタフェースを提供するMClientの実装を行なう．

5.4.1 実装概観

図5.10にUMLに基づくMClient部のクラス図を示す．

MClient_SとMClient_Pは，それぞれMSupporterとMPoolから呼び出されるリモートメソッドを定義したインタフェースである．MSupporterとMPoolの両者から共通

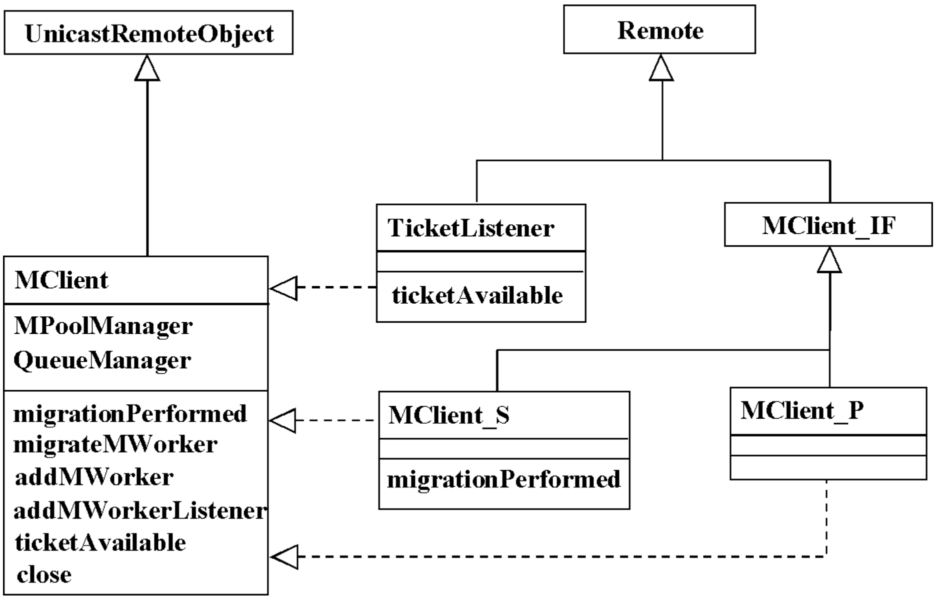

図 5.10: MClientのクラス図

して呼び出されるリモートメソッドはRemoteインタフェースを継承したMClient_IFインタフェースで定義している．それぞれのインタフェースで定義されたリモートメソッドはUnicastRemoteObjectを継承したMClientというクラスで実装される．MClientは内部に移送待機中のMWorkerを管理するQueueManagerと，MPoolを管理するMPoolManagerを持っている．

5.4.2 内部スレッドの構成

MClientでは以下の3つのスレッドから構成され，それらの協調動作によって全体としての機能を実現する．

QueueManagerスレッド

QueueManagerの中に登録されているMWorkerをMSupporterへ移送するスレッドである．移送すべきMWorkerが存在しない場合や，MPoolからTicket配布を待つ間はWait状態になる．addMWokerメソッドが呼ばれて新たなMWorkerが登録されたり，MPoolからticketAvailableメソッドが呼ばれるとRunnning状態になって処理を再開する．QueueManagerスレッドは優先度[1] 6のデーモンスレッド[2]として，動作する．

---

[1] 優先度の数字が大きいスレッドほど，JVMのスレッドスケジューラによって優先的に実行される．通常，デフォルトの優先度は5に設定されている．

[2] デーモンスレッド以外の全てのスレッドが終了すると自動的に終了するスレッド

MPoolCheckerスレッド

接続候補ではあるが，まだ接続できないMPoolに対して定期的に接続を試みるスレッドである．デーモンスレッドとして動作する．

MWorkerFinalizerスレッド

MWorkerFinalizerは，普段はWait状態にあり，アプリケーション内でMWorkerのラッパオブジェクトが不要になったことを検知すると，Running状態へ移行する．Running状態になると，不要になったラッパオブジェクトが保持しているリモート参照先である実体のMWorkerを削除する．MWorkerFinalizerスレッドは優先度1で動作して，有効MWorkerの数が0にならない限り停止しない．MWorkerFinalizerスレッドを通常のスレッドとして動作させることで，遠隔計算機上に移送されている全てのMWorkerが完全に終了することを保証している．

5.4.3 MWorker移送機能の実装

図5.11に示したmigrateMWorkerメソッドは，MWorkerを遠隔のMSupporterへ移送するためのメソッドである．引数に移送したいMWorkerを指定してこのメソッドを呼び出すと，返り値として移送されたMWorkerのリモート参照を保持したラッパオブジェクトが返される．このラッパオブジェクトを操作することで遠隔のMWorkerを操作できる．

```
public synchronized MWorker migrateMWorker(MWorker_Impl mworker_i)
```

図 5.11: migrateMWorkerメソッド

図5.12に示したaddMWorkerメソッドはMClient内の移送待機キューにMWorkerを登録するためのメソッドである．addMWorkerメソッドの引数で指定されたMWorkerは移送待機キューへ追加され，移送される順番を待つ．移送が完了するとアプリケーションプログラムのmigrationPerformedメソッドが呼び出され，MWorkerのリモート参照を保持したラッパオブジェクトが返される．migrateMWorkerメソッドよりもやや複雑な実装を必要とするが，移送待機キュー内のMWorkerは複数のスレッドによって効率的に移送される．したがって，大量のMWorkerを移送したい場合に有効なメソッドである．また，アプリケーション中のmigrationPerformedは異なるスレッドから呼ばれるので，複数のMWorkerを同時実行する場合にも有効である．

```
public synchronized void addMWorker(MWorker_Impl mworker_i)
```

図 5.12: addMWorkerメソッド

5.4.4 まとめ

表5.4に実装状況を示す．

表 5.4: MClient プロトタイプの実装状況

機　　能	実装状況
Ticket 要求	○
MWorker 送信	○
MWorker_Wrapper 作成	○
資源選定機能	○
リモートガーベッジコレクタ	○

5.5 MWorker 部の実装

本節ではアプリケーションの一部分としてMSupporter 上で実行されるMWorkerの実装を行なう．

5.5.1 実装概観

図5.13にUML に基づくMWorker部のクラス図を示す．

MWorkerインタフェースには計算の途中結果を待避するcheckpoint メソッド，MWorkerが終了したことをMSupporterへ明示的に知らせるrelease メソッドを定義している．ユーザが独自のMWorkerを作成するにはMWorkerインタフェースを拡張して，任意のリモートメソッドを定義する．次にユーザはMWorker_Impl を拡張したクラスに，このリモートメソッドの実装を行なう．図5.13ではユーザが定義するインタフェースとその実装クラスをそれぞれUserMWorker とUserMWorker_Impl としている．

一方，MWorker_Impl_Stub_Wrapper はMWorker_Implのリモート参照を保持して，自動checkpointing，MWorkerのリモート参照の自動切り替えなどを行なうクラスである．UserMWorker_Impl_Stub_Wrapper は本システムが提供するラッパジェネレータによって生成されるラッパクラスで，MWorker_Impl_Stub_Wrapper を継承し，UserMWorkerインタフェースを実装している．ただし今回のバージョンではラッパジェネレータの実装が完了していないので，UserMWorker_Impl_Stub_Wrapper は自分で記述する必要がある．

5.5.2 ラッパクラスの実装

UserMWorker_Impl_Wrapper クラスの記述は，UserMWorker_Impl クラスをもとにある程度機械的に行なえる．

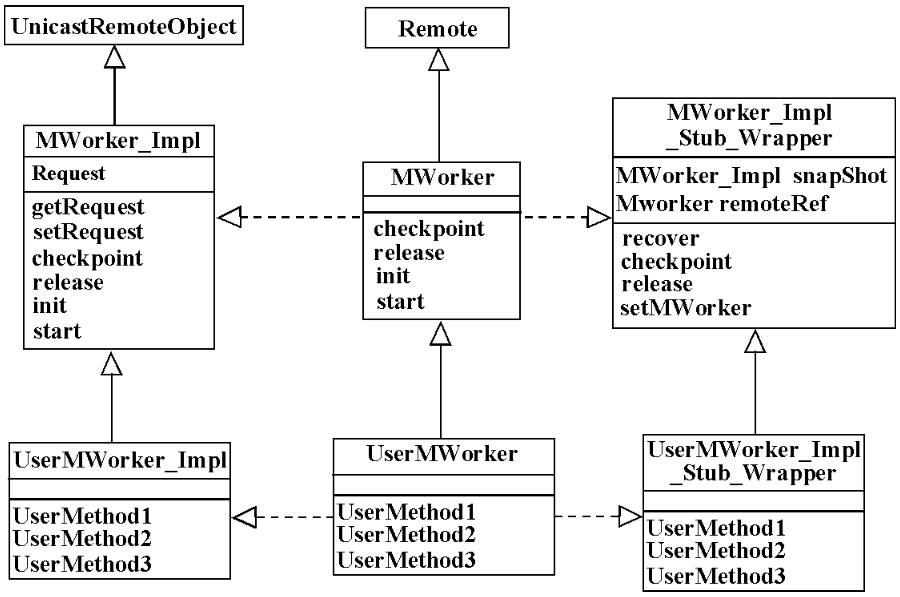

図 5.13: MWorkerのクラス図

図5.14にMyMWorker_Implクラスのラッパクラスを記述した例を示す．

ラッパクラスはMyMWorker_Implが移送中でないことを確認して，内部でリモートメソッドを呼び出す．正常に実行が終了すればcheckpointメソッドによって終了後のオブジェクトの状態（スナップショット）をバックアップしてから値を返す[3]．この間，例外が発生すると，recoverメソッドによってスナップショットが正常なホストへ移送され，新たなリモート参照へremoteRef変数が切り替えられる．その後，同じ手順を繰り返す．

5.5.3 資源選定機能の実装

MWorkerは自らを実行したい計算機を指定できる．これを行なうには資源選定方法を記述したRequestオブジェクトを生成し，MWorker_ImplクラスのsetRequestメソッドによってMWorker_Impl内へ設定すればよい．このRequestオブジェクトはTicket要求時にMClientからMSupporterへ渡される．なお，Requestオブジェクトの設定を省略するとデフォルトのRequestオブジェクトを用いてTicket要求が行なわれる．

計算資源選定方法の記述はRequestオブジェクトのselectメソッドをオーバライドして選定方法を記述する．例えば移送先計算機をSun Microsystems社のワークステーションであるUltra 30（オペレーティングシステムはSolaris 2.6）に限定する場合は，図5.15のように記述する．

---

[3] checkpointは付加的な機能である．checkpointを挿入するにはラッパジェネレータの起動時に明示的な引数を必要とする．

```
/*MyMWorker_Implの内容*/

public class MyMWorker_Impl extends MWorker_Impl implements MyMWorker{

    public MyMWorker_Impl() throws RemoteException {}

    public String sayHello() throws RemoteException{
      return "Hello!";
    }
}


/*MyMWorker_Impl_Stub_Wrapperの内容*/

public class MyMWorker_Impl_Stub_Wrapper
extends MWorker_Impl_Stub_Wrapper implements MyMWorker{

   …省略 …

    public String sayHello() throws RemoteException{
        while(true){
            try{
                   //MyMWorker_Implが移送中なら完了するまで待つ
                if(isMigrating()){
                    synchronized(this){
                        this.wait();
                    }
                }
                   //MyMWorker_Implのリモートメソッドを呼び出す
                String result = ((MyMWorker)remoteRef).sayHello();

                   //計算の途中結果 (スナップショット) を待避
                checkpoint();

                return result;  //結果を返す

             }catch(Exception e){
                   //エラーが発生した場合は，スナップショットを他のホストに移送
                recover();
             }
        }
    }

}
```

図 5.14: ラッパクラスの記述例

5.5.4 まとめ

表5.5に実装状況を示す．

```
public Ticket select(List tickets){
    Itelator i = tickets.itelator();
    Ticket t = null;
    while(i.hasNext()){
        t = (Ticket) i.next();
        if(t.computer.productName.equals(Ticket.COMPUTER_PRODUCTNAME_ULTRA30)&&
           t.os.name.equals(Ticket.OS_NAME_SOLARIS2_6))
        break;
    }
    return t;
}
```

図 5.15: Ticket選定方法の記述例

表 5.5: MWorkerプロトタイプの実装状況

機　　能	実装状況
MWorkerの実行機能	○
サポートホスト間の移動機能	○
テンプレートメソッド呼び出し機能	○
リモートメソッド呼び出し機能	○
位置透過型手続き呼び出し機能	○
明示的チェックポインティング機能	○
透過的チェックポインティング機能	○

5.6 動的分散ソフトウェア構築の実際

本節ではMeta Spaceを用いて動的分散ソフトウェアを構築する方法について述べる．例題として，与えられた値の範囲にある素数を求めるモジュールを，MWorkerとして実装する方法について述べる．MWorkerとして実装された素数探索モジュールは分散モジュールとして異なる計算機上で実行できる．

MWorkerの実装はリモートメソッドの定義，リモートメソッドの実装，ラッパジェネレータによるラッパクラスの生成の3ステップを必要とする．それぞれJava RMIのリモートメソッド定義，リモートメソッド実装，rmicコマンドによるスタブ及びスケルトンクラスの生成に対応しているため，Java RMIの利用経験があるユーザなら，MWorkerの実装も比較的容易に行なえる．

本例ではリモートメソッドを定義したPrimeNumberインタフェースと，その実装クラスのPrimeNumber_Impl，さらにラッパクラスであるPrimeNumber_Impl_Stub_Wrapperを作成する．各クラスの関係を図5.16に示す．

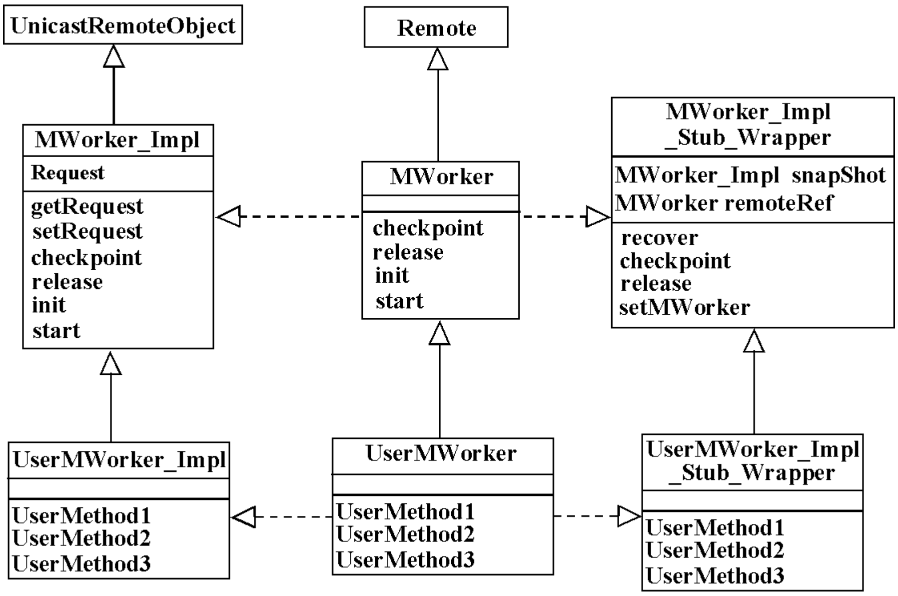

図 5.16: PrimeNumber のクラス図

5.6.1 リモートメソッドの定義

MWorker インタフェースを拡張したインタフェースにリモートメソッドを定義する．このインタフェースで定義されたリモートメソッドを呼び出すことで MSupporter 上の MWorker を操作することができる．PrimeNumber インタフェースに searchPrimeNumber メソッドを定義した例を図5.17に示す．searchPrimeNumber メソッドは，引数で指定した最小値から最大値までに存在する素数のリストを返すメソッドである．

```
import jp.ac.keio.sfc.ht.metaspace.mworker.*;
import java.rmi.*;

public interface PrimeNumber extends MWorker{
    public List searchPrimeNumber(int min, int max) throws RemoteException;
}
```

図 5.17: リモートメソッドの定義例

5.6.2 リモートメソッドの実装

MWorker_Impl を拡張したクラスにリモートメソッドを実装する．図5.18には PrimeNumber_Impl クラスに searchPrimeNumber メソッドを実装した例を示す．

```
import jp.ac.keio.sfc.ht.metaspace.mworker.*;
import java.rmi.*;
import java.util*;

public class PrimeNumber_Impl extends MWorker_Impl implements PrimeNumber{

    public List searchPrimeNumber(int min, int max) throws RemoteException{

        //ここにminからmaxの範囲内に存在する素数を
        //求める処理を記述する．

        return result ;
    }
}
```

図 5.18: リモートメソッドの実装例

5.6.3 ラッパクラスの生成

ラッパクラスはラッパジェネレータによって生成される．PrimeNumber_Impl クラスから PrimeNumber_Impl_Stub_Wrapper を生成する場合，コマンドラインで図5.19のようにして生成できる．なお，自動チェックポインティングを可能にしたい場合は-cp オプションをつける必要がある．

```
    % wg PrimeNumber_Impl
```

図 5.19: ラッパクラスの生成方法

ただし，現在のバージョンではラッパジェネレータの実装は完了していないため，ラッパクラスをユーザが記述する必要がある．PrimeNumber_Impl クラスのラッパクラスである PrimeNumber_Impl_Stub_Wrapper クラスを図5.20に示した．

MWorker の移送と実行

構築した MWorker を MSupporter 上に移送して実行する方法は2通り存在する．第一の方法はアプリケーション内部で MClient の migrateMWorker メソッドを呼び出す方法である．このメソッドは引数に指定された MWorker を即座に移送させ，返り値で MWorker のラッパオブジェクトを受け取ることができる．図5.21に migrateMWorker メソッドを用いて PrimeNumber_Impl を移送して実行する方法を示した．

第二の方法はアプリケーション内部で MClient の addMWorker メソッドを呼び出す方法である．このメソッドの引数で指定された MWorker は，一旦 MClient 内部の移送待機キューへ追加される．ラッパクラスはイベントによって返されるため，イベントを受け取るためには migrationPerformed メソッドが定義された MWorkerListener イ

```
import jp.ac.keio.sfc.ht.metaspace.mworker.*;
import jp.ac.keio.sfc.ht.metaspace.mclient.*;

public class PrimeNumber_Impl_Stub_Wrapper
extends PrimeNumber_Impl_Stub_Wrapper implements PrimeNumber{

    public PrimeNumber_Impl_Stub_Wrapper(PrimeNumber_Impl_Stub remoteRef,
                                         PrimeNumber_Impl orig, MClient mclient){
        super((MWorker)remoteRef, (MWorker_Impl) orig, mclient);
    }

    public List searchPrimeNumber(int min, int max) throws RemoteException{
        while(true){
            try{
                   //PrimeNumber_Implが移送中なら完了するまで待つ
                if(isMigrating()){
                    synchronized(this){
                        this.wait();
                    }
                }
                   //PrimeNumber_Implのリモートメソッドを呼び出す
                List result = ((MyMWorker)remoteRef).searchPrimeNumber(min, max);

                   //計算の途中結果(スナップショット)を待避
                checkpoint();

                return result;  //結果を返す

             }catch(Exception e){
                   //エラーが発生した場合は，スナップショットを他のホストに移送
                recover();
             }
        }
    }
}
```

図 5.20: ラッパクラスの記述方法

ンタフェースを実装する必要がある．図5.22にaddMWorkerメソッドを用いて複数のPrimeNumber_Implを移送して実行する方法を示した．

```
PrimeNumber_Impl pn_i = null;
MClient mclient = null;

try{                   //引数に接続するMPoolの識別子を指定して
                       //MClientを初期化する．
    MClient mclient = new MClient("//k2.ht.sfc.keio.ac.jp:7510/mpool-k2");

                       //MWorkerを初期化する．
    pn_i = new PrimeNumber_Impl();

                       //PrimeNumber_Implを移送する．
                       //返り値にラッパオブジェクトが得られる．
    PrimeNumber pn = (PrimeNumber) mclient.migrateMWorker(pn_i);

                      //0から50000000までの素数を求める．
                      //実行は遠隔計算機上で行なわれる．
    result = pn.searchPrimeNumber(0, 50000000);

}catch(Exception e){ …省略 …}
```

図 5.21: migrateMWorkerメソッドを用いたMWorkerの移送と実行方法

5.7 本章のまとめ

本章では，動的分散ソフトウェアの構築を容易にするMeta Spaceプロトタイプの実装について述べた．

まず，Meta Spaceの実装環境について述べた．次にMPool部，MSupporter部，MClient部，MWorker部の実装について詳細に述べた．

本章の最後では，Meta Spaceを用いたアプリケーション構築方法について，具体例と共に述べた．

```
public class Sample implements MWorkerListener{

     …省略 …

    public void aMethod(){

        PrimeNumber_Impl pn_i[] = PrimeNumber_Impl[5];
        MClient mclient = null;
        try{               //引数に接続するMPoolの識別子を指定して
                           //MClientを初期化する．
            MClient mclient = new MClient("//k2.ht.sfc.keio.ac.jp:7510/mpool-k2");

                          //ラッパオブジェクトを受け取るリスナーを指定する．
            mclient.addMWorkerListener(this);

                          //5個のPrimeNumber_Implを移送待機キューに追加する．
            for(int i=0; i<pn_i.length; i++){
                          //MWorkerの初期化．
                pn_i[i] = new PrimeNumber_Impl();
                          //個々のMWorkerを区別するために識別子をつける．
                pn_i[i].setName("PrimeNumber");
                          //MWorkerをMClient内の移送待機キューに追加する．
                mclient.addMWorker(pn_i[i]);
            }

        }catch(Exception e){ …省略 …}

    }
                          //MWorkerの移送が完了するとラッパオブジェクトが返される．
    public void migrationPerformed(MWorker mworker){

        List result = null;
        synchronized(this){
            count++;      //countの初期値は0
        }

        try{
                          //識別子がPrimeNumberなら素数探索を行なう．
             if(mworker.getName().equals("PrimeNumber")){

                PrimeNumber pn = (PrimeNumber)mworker;
                          //素数探索は10000000づつ，5つのMWorkerに分割して行なう．
                result = pn.searchPrimeNumber(10000000*(count-1), 10000000*count);
            }

        }catch(Exception e){ …省略 …}
    }
}
```

図 5.22: addMWorkerメソッドを用いたMWorkerの移送と実行方法

第 6 章

動的分散ソフトウェアツールキットの評価

本章では，Meta Space プロトタイプの評価を行なう．

本章の前半では，Meta Space プロトタイプを用いて構築された並列計算型アプリケーションのパフォーマンスについて定量的評価を行なう．これより，並列計算型アプリケーションはMeta Space を用いることで，パフォーマンスが向上することを実証する．また同時に，Meta Space を用いた並列計算型アプリケーションのパフォーマンスは，MPool へ追加される計算資源の量に比例することも実証する．

本章の後半では，Meta Space を類似システムと機能比較し，Meta Space が類似システムに比べて機能的に優れていることを示す．

6.1 定量的評価

本節ではMeta Spaceを用いて構築された並列計算型アプリケーションのパフォーマンスについて定量的評価を行なった．本測定により，並列計算型アプリケーションはMeta Spaceを用いることでパフォーマンスが向上することを実証する．また同時に，Meta Spaceを用いた並列計算型アプリケーションのパフォーマンスは，MPoolへ追加される計算資源の量に比例することも実証する．本節では，まず測定方法について述べ，測定環境，結果および考察について述べる．

6.1.1 測定方法

測定用アプリケーションには5.6節で紹介した素数探索を行なうアプリケーションを利用した．素数探索を行なうMWorkerを64個作成し，各MWorkerに0から50000000の範囲に存在する素数を探索させた．MPoolへ追加する計算機台数を1台，2台，4台，8台，16台，32台，64台と増加させて，それぞれの場合で探索開始から全てのMWorkerの素数探索が完了するまでの時間を計測した．また，Meta Spaceを用いずに通常の方法で構築した素数探索アプリケーションの計測も行なった．

6.1.2 測定環境

測定には100Base-Tで相互接続された，Sun Microsystems社のワークステーションであるUltra 30を64台使用した．図6.1に測定環境を示す．また表6.1にUltra 30の性能を示す．

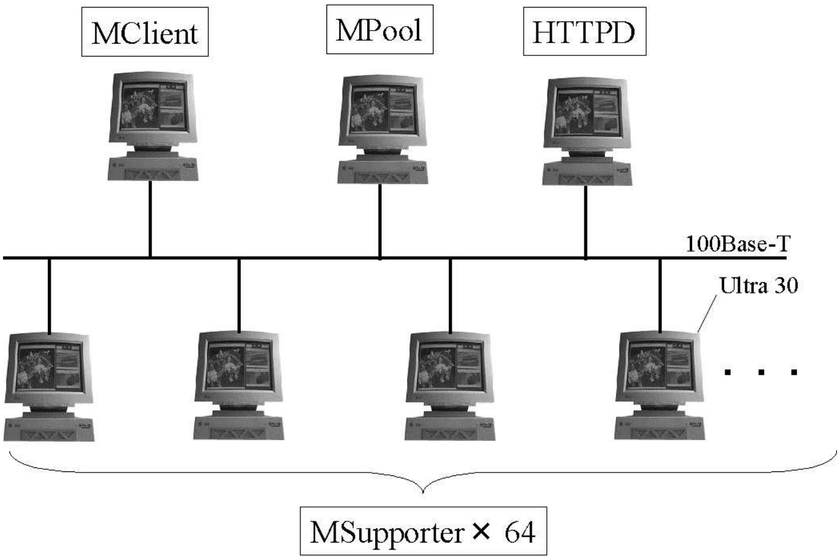

図 6.1: 測定環境

表 6.1: 測定環境

CPU	UltraSPARC-II 248MHz × 1
メモリ	128MB
OS	Solaris2.6

6.1.3 測定結果

測定結果を表6.2に示す．

表 6.2: 測定結果

	測定値 (millisecond)	速度比
システムなし	1178061	1.00
MPoolに1台追加した場合	1413399	0.83
MPoolに2台追加した場合	680442	1.73
MPoolに4台追加した場合	347960	3.39
MPoolに8台追加した場合	178154	6.61
MPoolに16台追加した場合	95876	12.29
MPoolに32台追加した場合	51802	22.74
MPoolに64台追加した場合	27417	42.97

表6.2中のシステムなしの計測値は，Meta Spaceを用いずに構築したアプリケーションの測定値を示す．これより，Meta Spaceを用いずに構築したアプリケーションの測定値が，Meta Spaceを用いて構築したアプリケーションでMPoolに1台の計算機を追加した場合の測定値より小さいことがわかる．この原因は，Meta Spaceを利用した場合に発生するサブシステム間の通信やクラスファイル転送によるオーバヘッドが影響したと考えられる，

図6.2にMeta Spaceを用いたアプリケーションの利用計算機台数における速度比のグラフを示す．これより利用計算機台数に対して速度比が直線的に向上することがわかる．測定用アプリケーションにおいて利用計算台数と速度比の関係は式6.1で表される線形回帰直線で近似できる．

$$f(x) = 0.91x \qquad \text{(ただし}x\text{：利用計算機台数)} \tag{6.1}$$

図6.2には測定値の他に，通信遅延や計算機間の計算速度のばらつきが全くなかった場合の値を，傾き1の理論値としてプロットしている．これより測定値がMPoolへ追加する計算機の台数が多くなるにつれて，理論値からはずれることがわかる．この原因として，いくつか考えられる．

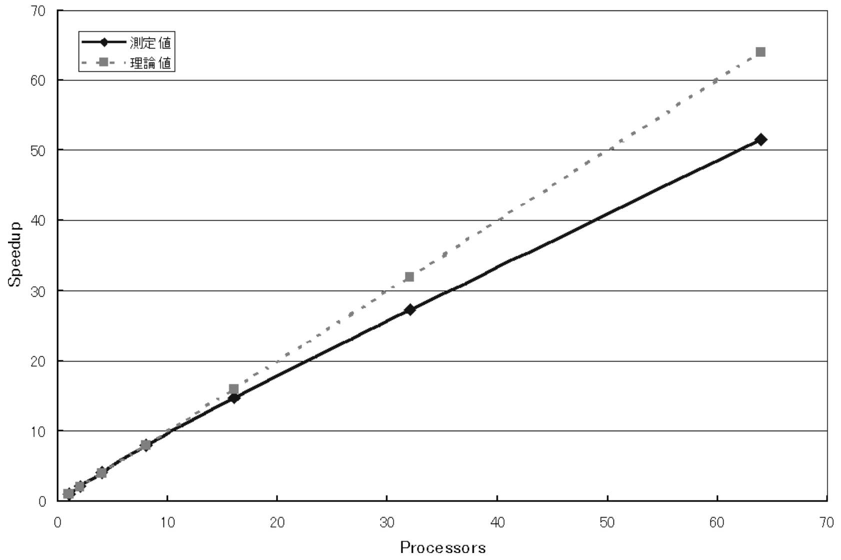

図 6.2: 測定結果のグラフ

ネットワーククラスローディングのオーバヘッド
MSupporterはMWorkerを受け入れると，MWorkerの実行に必要なクラスファイルをネットワーク経由でロードする．ただし，ネットワーク経由でのロードは初回だけで，2回目以降はキャッシング機構によってクラスファイルを再利用する．したがって分散率[1]の増加に伴って，全実行時間に占めるクラスローディングオーバヘッドの相対的割合が増加する．

ネットワークの輻輳
x個のMWorkerを分散率dで分散させた場合，xd回のクラスファイル要求がWWWサーバに対して同時発生する．したがって分散率が増加するほどWWWサーバの負荷とネットワークの輻輳が発生するため，ロード時間が大きくなる傾向にある．

6.2 類似システムとの機能比較

本項では複数の計算機を用いた大規模分散計算を目的とする類似システムとしてPVM，Javelin，Ninfletを取り上げ，Meta Spaceと機能比較を行なう．評価項目および評価結果を表6.3に示す．

PVM (Parallel Virtual Machine)[20]は，ネットワークに接続された異機種UNIXコンピュータ群を，並列コンピュータとして利用できるライブラリ及び，コマンドである．PVMはセキュリティ機構を提供していないため，不特定多数の分散モジュールを実行する大規模分散計算には適さない．また，マルチプラットフォーム性を実現して

---

[1] x個のMWorkerをn台の計算機に分散させた場合の分散率を$\frac{n}{x}$とする．

表 6.3: 類似システムとの機能比較

	PVM	Javelin	Ninflet	Meta Space
実行ファイルの自動転送機能	×	○	○	○
分散モジュールの自動復旧機能	×	×	○	○
分散モジュールの遠隔制御機能	○	×	△	○
分散モジュールの遠隔手続き型制御機能	×	×	△	○
分散モジュールの位置透過的記述機能	×	×	○	○
プラットフォーム非依存性	×	○	○	○
セキュリティ機構	×	○	○	○
提供ポリシの設定	×	×	×	○
提供計算機のCPU占有率制御機能	×	×	×	○

いないため，計算機のアーキテクチャごとに実行ファイルをメンテナンスするという煩わしさがある．

これに対しJavaで記述され，マルチプラットフォーム性を実現したシステムとしてJavelin[21]やNinflet[22]などがあげられる．

Javelinは本システムのMPoolにあたるBrokerへの登録のために，Broker AppletのあるWebページを開くという方法を採用している．この方法は手軽に登録できるという利点はあるが，計算資源を提供している間は常時ブラウザを開いておかねばならないため常連ユーザを得られにくいという欠点を持つ．

Ninfletは本システムと類似した構成を持つ．本システムのMPool，MSupporter，MClient，MWorkerがNinfletのDispatcher，Server，Client，Ninfletにそれぞれ対応する．分散するNinfletの記述はWorkerクラスを拡張して，Workerのあらかじめ定められたテンプレートメソッドをオーバライドする．この方法の利点は定められたメソッドの内容を実装するだけで良いためプログラマの負担が少ないことがあげられる．しかし，ユーザが分散モジュールの制御インタフェースを独自に定義できない，分散モジュールをインタラクティブに操作できないなどの問題点がある．またNinflet，Javelinとも大規模分散計算を前提としているため，厳しいセキュリティ方針をとっている．一方，Meta Spaceでは，セキュリティ方針を柔軟に設定可能なため，多様な運用ニーズに対応できる．

6.3 本章のまとめ

本章では，Meta Spaceプロトタイプの評価を行なった．

本章の前半では，Meta Spaceプロトタイプを用いて構築された並列計算型アプリケーションのパフォーマンスについて定量的評価を行ない，並列計算型アプリケーションはMeta Spaceを用いることで，パフォーマンスが向上することを実証した．同時に

パフォーマンスは，MPoolへ追加される計算資源の量に比例することも実証した．

本章の後半では，Meta Spaceを類似システムと機能比較し，Meta Spaceが類似システムに比べて機能的に優れていることを示した．

第 7 章

結論

7.1　まとめ

本論文では，サブシステムが同一計算機に常駐する従来までの分散ソフトウェアモデルを，静的分散ソフトウェアモデルと呼び，その問題点を考察した．静的分散ソフトウェアモデルの問題点は，サブシステムに障害が発生すると復旧できないため耐故障性が低い点，また実行ファイルのメンテナンスが困難な点である．本研究ではこれらの問題点に対して有利なモデルである，動的分散ソフトウェアモデルを提案した．動的分散ソフトウェアは，サブシステムが動的に計算機間を移動できるため，動的負荷分散，耐故障性の向上，大規模分散性の向上，計算環境の最適化，小型携帯端末の処理支援を実現できる．

本研究では，動的分散ソフトウェアを容易に構築するためのツールキットであるMeta Spaceを開発した．Meta Spaceを用いて構築したアプリケーションは，モジュールが計算機間を自由に移動可能となる．また分散モジュールは，Meta Spaceのサポートホストへ分散可能となる．Meta Spaceの設計にあたっては，耐故障性，柔軟性，大規模分散性，安全性，簡便性を設計方針とした．この設計方針に基づき，計算資源を管理分配するMPool部，MWorkerを受け入れ実行するMSupporter部，アプリケーションから計算資源を要求するMClient部，分散モジュールとして実行されるMWorker部などのサブシステムを設計および実装した．Meta Spaceを用いて動的分散アプリケーションを構築する際，分散モジュールであるMWorkerの実装は，Java RMIのリモートメソッドの実装方式に類似しているため，Java RMIの利用経験があるプログラマなら，MWorkerの実装も比較的容易に行なえる．

定量的評価では，Meta Spaceプロトタイプを用いて構築された並列計算型アプリケーションのパフォーマンスについて評価した．これにより，並列計算型アプリケーションはMeta Spaceを用いることで，パフォーマンスが向上すること，またパフォーマンスは，MPoolへ追加される計算資源の量に比例することを実証した．さらに定性的評価では，類似システムとしてPVM，Javelin，Ninflet を取り上げ，機能面において本システムが優れていることを実証した．今後は，ラッパジェネレータやMPool間のTicket融通機構，認証機構など，未実装部分の実装に取り組む．

7.2　今後の課題

Meta Spaceの今後の課題として以下の3点があげられる．

MWorkerの自律性向上

現在のバージョンではMWorkerは強制退去命令が出された場合のみ，計算機間を移動する．しかしMWorkerは，自らより良い実行環境を求めて，自律的に計算機間を移動するのが理想的である．これを実現するために，MWorkerの自律的移動機構，移動の判断基準となるサポートホストの負荷およびベンチマークの測定機構などを提供する必要がある．

コンパイラによるチェックポインティングの自動挿入

現在のバージョンでは，チェックポインティングはプログラマが明示的にコード内にチェックポイントメソッドを挿入する方法か，MWorkerのリモートメソッド呼び出し時に行なう方法しか提供していない．両者には，それぞれ第4.7.4項で考察した問題点が存在する．

したがって両者の問題を解決するために，コンパイラによるチェックポインティングの自動挿入機能を提供する必要がある．これにより，プログラマに負担をかけることなく，安全なチェックポインティングが実現できる．

コンパイラによるMWorker自動生成

現在のバージョンでは，プログラマが分離するモジュールを決定し，これをMWorkerとして実装する必要がある．したがって，プログラマはMeta Spaceのアーキテクチャを理解し，MWorkerの作成方法を習得せねばならない．この問題を解決するために，コンパイラによるMWorker自動生成機能を提供する必要がある．コンパイラがプログラム中の分離可能なモジュールを解析し，これをMWorkerとして自動生成することにより，プログラマは特別な知識や経験を必要とせずに，Meta Spaceを用いた動的分散ソフトウェアを構築可能となる．

謝辞

本研究を進めるにあたり，絶えず懇切丁寧な御指導を賜わりました，慶應義塾大学環境情報学部教授徳田英幸博士に深く感謝いたします．

慶應義塾大学徳田・村井・楠本・中村研究会の諸先輩方には，折りにふれ貴重な示唆や御助言、御指導を頂きました．特に慶應義塾大学政策・メディア研究科修士1年の中澤仁氏，大越匡氏には，本論文執筆にあたって絶えざる励ましと御指導を賜りました．また、Keio Media Space Family-Architecture(KMSF) 研究グループのメンバーには，本研究に関する様々な議論をして頂きました．ここに，深い感謝の意を表します．

平成11年2月3日
藤村 浩士

参考文献

[1] Intel Corporation, “マイクロプロセッサの歴史”, http://www.intel.co.jp/jp/home/cpc/museum/hof/hof_main.htm

[2] Network Wizards, “Internet Domain Survey, July 1998”, http://www.nw.com/zone/WWW/report.html

[3] James Gosling, Bill Joy, Guy Steele, “The JAVA Language Overview”, Sun Microsystems Inc. (1998).

[4] Tim Lindholm, Frank Yellin, “The Java Virtual Machine Specification”, Sun Microsystems Inc. (1996).

[5] “UNIX Programmer’s Manual, 4.2 Berkeley Software Distribution, Virtual VAX-11 Version”, Computer Science Division, Department of Electrical Engineering and Computer Science, University of California at Berkeley, August (1983).

[6] Birrell,A.D., and Nelson,B.J, “Implementing Remote Procedure Calls”, ACM Trans.on Computer Systems,vol.2,pp.39-59, Feb 1984

[7] Sun Microsystems Inc., “Remote Procedure Call Protocol Specification Version 2”, RFC 1057 (1988).

[8] Zahn,L., Dineen,T.H., Leach,P.J., Martin,E.A., Mishkin,N.W., Pato,J.N., and Wyant,G.L. “Network Computing Architecture”, Prentice Hall, Englewood Cliffs,N.J. (1990).

[9] “Courier: The Remote Procedure Call Protocol”, XNSS 038112, Xerox Corp., (1981).

[10] Sun Microsystems Inc., “JAVA REMOTE METHOD INVOCATION”, http://www.java.sun.com/marketing/collateral/rmi_ds.html (1998).

[11] Satoshi Hirano, “Horb Home Page”, http://ring.etl.go.jp/openlab/horb/

[12] Object Management Group, “The Common Object Request Broker: Architecture and Specificatoin”, http://www.omg.org

[13] 日本アイ・ビー・エム 東京基礎研究所, “Aglets Software Development Kit”, (1998).

[14] 佐藤一朗, “AgentSpace A Mobile Agent System”,

http://www.islab.is.ocha.ac.jp/agent (1998).

[15] Erich Gamma, Richard Helm, Ralph Jphnson, John Vlissides, Addison Wesley, “Design Patterns”, (1995).

[16] エリック・アームストロング, “C++コンパイラvs.JITコンパイラ”,JAVA WORLD 7月号 (1998).

[17] IBM, “Java JIT Compiler”, http://www.trl.ibm.co.jp/projects/jit/detail_e.htm

[18] David Griswold, “The JavaTMHotSpot Virtual Machine Architecture”, Sun Microsystems Inc.(1998).

[19] 独古康昭, “JavaChipの可能性と応用”, 情報家電WGワークショップ(1998).

[20] Al Geis, Adam Beguelin, Jack Dongarra, Weicheng Jiang, Robert Manchek, Vaidy Sunderam, “PVM 3 USER’S GUIDE AND REFERENCE MANUAL”, (1993).

[21] Bernd O.Christiansen，Peter Cappello，Mihai F.Ionescu，Michael O.Neary，Klaus E.Schauser，and Daniel Wu，“Internet-Based Parallel Computing Using Java”，ACM Workshop on Java for Science and Engineering Computation，(1997).

[22] 高木，松岡，中田，関口，佐藤，長嶋，“Ninflet:JavaによるWorld-Wide High Performance Computing環境”，Internet Conference’97.