修 士 論 文

Group Communicationに基づいた 耐故障モバイルエージェント

指導教官　片山卓也 教授

審査員主査：片山卓也 教授

審査員：渡部卓雄 助教授

審査員：篠田陽一 助教授

北陸先端科学技術大学院大学

情報科学研究科 情報システム学専攻

910092

林原尚浩

2001年2月15日

要 旨

モバイルエージェントはホスト間を移動しながら処理を行なうプログラムである. モバイルエージェントを実用的なアプリケーションとして扱う場合, ホストやネットワークの障害に対する何らかの機構が必要である. 一般的なアプローチとしてはモバイルエージェントの複製を作成することによって冗長性を確保し障害に対処するという方法を用いている. しかし, この方法では同じモバイルエージェントが複数存在することになり, それらが何らかの障害により一つ以上のモバイルエージェントが並行して同じ処理を行なうという問題が生じる.

本研究では Group Communication を用いることによって, モバイルエージェントの耐故障性を高めるシステムを提案した. また, 冗長性を確保したことによって起る処理の重複などの問題にも対処している. さらに, 本システムは特定のエージェントシステムに依存しない構造になっており, 異なるエージェントシステムにおいても同じサービスを容易に提供できるように考慮している. 最後に, 実際のエージェントシステムへサービスを提供することの容易性を, AgentSpace へ本システムのプロトタイプを実装することによって考察した.

目 次

図 目 次

表 目 次

第 1 章

はじめに

1.1 背景

近年，ネットワークインフラやインターネットの爆発的な普及により以前の情報一局集中管理から分散化が図られてきた．また, 携帯電話などに代表されるアドホックネットワーク[1][河 98] は 20 世紀後半から 21 世紀初頭にかけて急速に発展してきた.

このような背景により，ネットワークを介するリモートオペレーションを効率的に行なう手法に関する研究は多種多様になされてきた．そのなかに RPC(Remote Procedure Call) などのクライアント・サーバ型のシステムがある．RPC は一方のクライアント・コンピュータから他方のサーバ・コンピュータのプロシージャを呼び出すものである．この手法はクライアントが計算能力の低いコンピュータであった場合でもプロシージャの実行はサーバ側で行なうため非常に有効な手段である.

しかし, RPC においてクライアントはサーバにプロシージャの実行を要求してその応答が返ってくるまで常にネットワークの接続を要求する．よって, プロシージャの規模が大きくなるにつれてネットワーク資源の占有率は高くなる．また, これは LAN の非常に高速なネットワークを仮定しており, 公衆回線などの低速なネットワーク環境では現実的な手段であるとは考えにくい.

リモートオペレーションを行なう手法として，本研究ではモバイルエージェント (エージェントシステム) に着目する.

モバイルエージェントとは，現在の状態を保存しつつネットワークを通じて処理対象のデータがある他の計算機に移動し，その計算機上で処理を続行するというプログラムで

[1]必要に応じて一時的に構築するネットワーク環境．頻繁に接続, 切断が起る

ある.

モバイルエージェントを用いたリモートオペレーションは図 1.2 のようにネットワークへの接続を必要とするのはエージェントを転送する時だけで, 処理を行なっている間は接続を必要としない. そのため, RPC に比べてネットワーク資源効率が良いと考えられる.

また, ネットワークの接続が不安定な場合やアドホックネットワークなどの環境において処理の間接続を保たなければならない RPC は現実的ではない. モバイルエージェントは移動後の処理時は接続を必要としないのでこの様な環境では有効である.

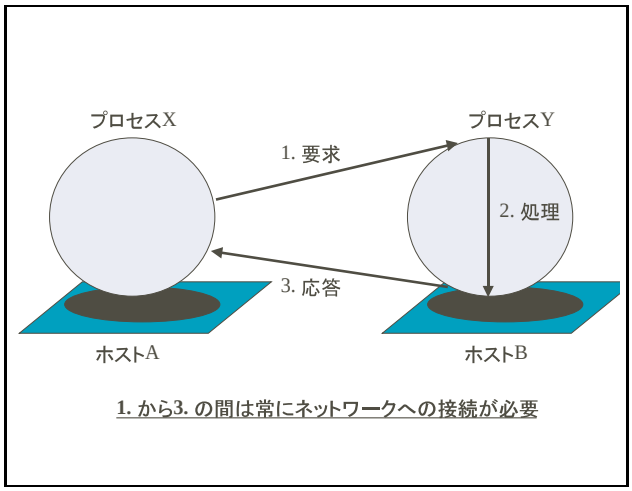

図 1.1: RPC のリモートオペレーション

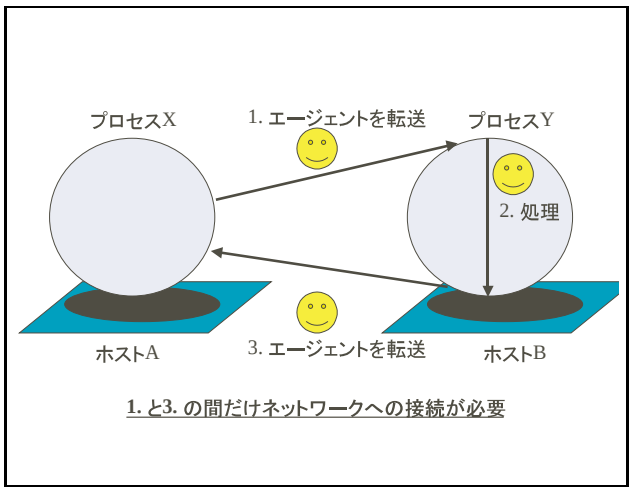

図 1.2: モバイルエージェントのリモートオペレーション

1.2 問題点

モバイルエージェントの転送は基本的に非同期である. そのためネットワークを介した計算機間の転送における信頼性は低く，また, モバイルエージェントを用いて構築するアプリケーションもまた信頼性が低くなる.

この対処法としては, モバイルエージェントの複製を作って回避する方法が一般的である. しかし, 単純にこの方法を適用しただけでは問題が起る. 一つの例としてオリジナルのモバイルエージェントと複製のモバイルエージェントが並列に動作した場合である. この場合, 両方のモバイルエージェントが同じ処理を重複して行なってしまう. 重要なことはどのモバイルエージェントがオリジナルで複製かという見分けをつけることである.

これらはモバイルエージェントが商用のアプリケーションとして用いられている例が少ない要因の一つである.

1.3 解決法の概要

前述の問題点の解決法としては Group Communication に基づいた複製技術を用いる. Group Communication とは分散システムのプロセスをグループとして管理する技術である. 既に, この技術を用いた分散システムに対する耐故障性機構は実装例が存在する. 本論文では, Group Communication を用いてモバイルエージェントを複製しグループとして管理する手法を述べる.

これによって, モバイルエージェントの耐故障性を高め, 処理の重複などの問題を解決する.

1.4 論文の構成

本論文の構成を以下に述べる.

- **第 2 章:** モバイルエージェントの概念, エージェントシステムの概要を述べる. また, 現存するエージェントシステムについていくつかの例を挙げて解説する.
- **第 3 章:** 一般的な耐故障機構について述べる. また, Group Communication の説明や実装例について述べ, エージェントシステムへの適用を行なう場合に生じる問題点などを述べる.

- **第 4 章:** 本研究で提案するシステムの概要について述べる. 特に, システムの構造や特色などを解説する.
- **第 5 章:** システムの下位層である Group Communication Layer について解説する. ここでは, 障害処理の手続きやそれに伴うサーバとモバイルエージェントの通信プロトコルを述べる.
- **第 6 章:** システムの上位層に当たる Agent Management Layer について述べる. ここでは, 障害が起った際に Group Communication Layer では対応できない部分について対処する機構を解説する.
- **第 7 章:** エージェントシステム”AgentSpace” へのプロトタイプ実装を解説する.
- **第 8 章:** 本研究で提案した手法の有用性を述べ, 考察を行ない, 最後に本研究の展望を述べる.

第 2 章

エージェントパラダイム

本章では, モバイルエージェントやそのシステムの概念と既存のエージェントシステムについて述べる.

2.1 モバイルエージェント

近年, ネットワークの普及により情報が分散して存在する傾向にある. これに伴い, 大量の情報の中から自身のニーズに合う情報を探すことは非常に大変な作業となっている. また, ネットワークへの接続に関しても, モバイル環境の増加と共に, アドホックネットワークが非常に多く見られる.

この様な背景において, 1990 年代初期にリモート・オペレーションを効率的に行なうツールとしてエージェントの概念が考え出された. エージェントの概念は以下の 2 種類が存在する [岩 98].

- 位置固定エージェント (Stationary Agent)
  エージェントの中でも, 実行を開始したシステム上だけで実行するタイプのエージェント
- モバイルエージェント
  実行を開始したシステムに拘束されないエージェント

本研究では後者のモバイルエージェントに焦点をおく.

2.1.1 用語の定義

本論文で用いるモバイルエージェントに関する語彙の定義を行なう.

エージェント： 単にエージェントと表記されている場合はモバイルエージェントを指す.

エージェントシステム： エージェントを生成, 複製, 実行等を制御するランタイム環境を指す.

エージェントタイプ： エージェントを生成したエージェントシステム (つまり, そのエージェントが属するエージェントシステム) の種別を指す.

2.1.2 概念

本研究におけるモバイルエージェントの定義は, 「実行を開始したシステムには拘束されない自律的な活動を行なうプログラムである」という OMG の MASIF[OMG97] の定義を採用する.

エージェントは, それを実行するランタイムであるエージェントシステムの上で実行する. そのため, エージェントの実行にはエージェントシステムが起動している必要があり, また, エージェントが移動して処理の継続を行なう場合も移動先のホストにエージェントシステムが起動している必要がある.

エージェントの概念として新しい部分は実行状態を保存しつつ移動し, 処理を継続できると言うことである. 実行状態の保存については実装している言語の制約などによって異なるが, Java の場合は, Serialize 可能なオブジェクトやインスタンス変数は保存できるが, プログラムカウンタや Serialize 不能なオブジェクトなどは移動後に初期化される.

特徴

モバイルエージェントの特徴として以下のようなものが挙げられる [岩 98].

- **移動性:** プロセスが処理を継続しつつ, ネットワークで接続されたホスト間を移動することができる.
- **局所性:** 各々のエージェントはシステム全体の情報を持っている必要はなく, 必要に応じてシステムの情報を取得できる.

- **自律性:** エージェントがユーザの手を離れ, 外部からのイベントなどの刺激がなくても処理を継続できる.
- **協調性:** エージェント対エージェントで通信し, 情報などの交換を行なうことができる. それによってエージェント自身の情報や行動に反映させることができる. 但し, 現時点において異なるエージェントタイプ間ではプロトコルなどの違いから通信することはできない.
- **柔軟性:** エージェントを取り巻く環境の変化に適応して, 自身の目的達成のために適切な行動を取ることができる[1].

利点

以上のような特徴を持つモバイルエージェントの利点としては次のものが挙げられる.

- **接続資源の有効活用:** エージェントは移動する時だけ接続を要求する. つまり移動後, リモートで処理を続行している間は接続は必要としない.
- **携帯端末等のサポート:** 携帯端末等は処理能力が低いものが多い. 処理自体を他のホストで実行するエージェントはローカルの計算機資源を消費しないため, この様な端末での有効性は大きいと考えられる.
- **負荷分散:** エージェントというプロセスは移動することができる. この利点を利用して, ホストで負荷が大きくなった場合 (つまり, エージェントが集中した場合), エージェントを移動させることによって容易に自然な形で負荷分散が可能となる.

2.2 エージェントシステム

エージェントシステムとはエージェントの実行などを制御するランタイムである. 基本的にはホストでエージェントシステムが起動し, その上でエージェントが稼働するという構造になっている.

現在, エージェントシステムは数多く開発されており, 実装している言語も多岐にわたっている. 中でも, Java で実装されているエージェントシステムはその多くを占めている.

[1] この特徴については単なる例外処理を指しているのが現状である.

この背景には, Java 言語がネットワークアプリケーションとして高い機能[2]を持っていることや OS に依存しないこと等の利点を認知されつつあるからである. 実際に, Web ブラウザに Java VM が組み込まれていたり, 組み込み用 Java により多くの電化製品に Java が組み込まれようとしている.

エージェントシステムはエージェントを制御するために概ね以下のような動作をする.

- エージェントの生成
- エージェントの終了
- エージェントの送信
- エージェントの受信

この基本的な機能に加え複製を作ったり, 二次記憶に退避させる機構等が備わっているエージェントシステムもある.

2.3 既存のエージェントシステム

2.3.1 Telescript

General Magic 社の Telescript は 1990 年代初期に開発された [山 96]. このシステムはモバイルエージェントの概念を実装した初めてのシステムとされており, "Telescript エンジン" と呼ばれるエージェントシステム上でエージェントが実行される仕組みになっている. また, エージェントを作成するために独自の言語体系を持っている.

Telescript は以下の概念から構成されている.

- **プレース (Places):** モバイルエージェントにサービスを提供する場である. Telescript 技術はネットワークがどれだけ巨大であっても, それはプレースの集まりであると考えている. Telescript エンジンはエージェントをプレース毎に管理している
- **エージェント (Agents):** エージェントは通信型アプリケーションであり, プレースに存在する.
- **トラベル (Travel):** トラベルはエージェントがプレースからプレースへ移動することを指す

[2]プログラムのシリアライズや RMI を用いたメソッドの遠隔呼び出しなどの API が備わっている.

- **ミーティング (Meetings):** エージェント同士が同一プレースに移動して出会い, プロシージャを相互に呼び出せる
- **コネクション (Connections):** 異なるプレース, 異なるホストにいるエージェントと通信すること
- **オーソリティ (Authorities):** エージェントやプレースはオーソリティーを識別することができる. エージェントやプレースのオーソリティとは, 現実世界でそれらが表す個人 (ユーザ) や組織である
- **パーミット (Permits):** エージェントやプレースにおいてオーソリティの行動を制限すること

2.3.2 Aglets

Aglets はIBM 社が開発したエージェントシステムである [DBL98][MO98]. このエージェントシステムは Java ベースのエージェントシステムである.

特徴としてはクラスをロードする際, 動的なローディングか静的なローディングかを選択できる点がある. エージェントを作成する際, Aglet クラスのサブクラスを定義する以外には以下の 3 つの特徴を持ったエージェントを作ることができる.

- **Slave:** 移動パターンを指定することができるエージェント
- **Messenger:** Message クラスを送るエージェント. Aglets は Message クラスを送ることでエージぇん間の通信を実現している
- **Notifier:** 移動先で処理を行ない, 一定時間毎にその結果を返すエージェント

Aglets のシステムは下位層の Communication 層と上位層の Aglets Runtime 層から出来ています. 上位層の Aglets Runtime 層は Aglets の様々な API に加えて, エージェントの移動時に必要なエージェントの符号化, 復号化, クラスローディング, ガベージコレクション等を行なう.

2.3.3 AgentSpace

AgentSpace は Java をベースとしてお茶の水大学の佐藤一郎氏によって開発されたエージェントシステムである [Sat99].

エージェントの実体ははAgentクラスのサブクラスを定義し, Javaのアプリケーションと同じくコンパイルを行なった後, jar形式で圧縮したものである.

AgentSpaceの構成

```
(agentspace)-|-system-|-agentspace-|-*.java
                      |            |(システムのソースプログラム)
                      |            |
                      |            |-*.class
                      |            |(システムのクラスファイル)
                      |
                      |-agent------|-*.java
                      |            |(サンプルエージェントの
                      |            |ソースプログラム)
                      |            |
                      |            |-*.class
                      |            |(サンプルエージェントの
                      |            |クラスファイル)
                      |
                      |-*.jar  (サンプルエージェントの実体)
```

システムのソースコードは全て公開されており, さらに, 構造がシンプルであるためシステムに手を加えることが比較的容易に行なえる.

第 3 章

耐故障システム

3.1 概要

システムの耐故障性を高めるためのアプローチとしてハードウェアによるアプローチとソフトウェアによるアプローチがある. 前者はハードウェアの冗長性を高めたり, 低レベルのアプリケーション (マイクロコンピュータなど) で故障を検知する. 後者は全てソフトウェアで故障に対応するという方法であり, 本稿では後者のソフトウェア的アプローチの耐故障性システムを構築する. また, 本稿において以降, **耐故障性システム**と記述してある場合はソフトウェア的アプローチを指すことにする.

後述する Group Communication はソフトウェア的な耐故障性システムの実装例であり, 本研究ではこの機構や概念をエージェントに応用することを考える.

3.2 耐故障性アプローチ

耐故障性システムを構築するためには複製による冗長性の確保が必要である. 複製を用いた耐故障性を実現するアプローチとして, 以下に 2 つの代表的なアプローチを示す.

どちらのアプローチもシステムは複数台サーバマシンから構成されており, 目的はサーバ側で故障が発生しても他のサーバで故障したサーバの代りに処理を行なうことで故障をマスクし, クライアントから見てそれらがあたかも一つの信頼性の高いサーバであるようにサービスを提供することである.

3.2.1 Statemachine アプローチ

Statemachine アプローチ [MUL] は *Active Replication* とも呼ばる. サービスを提供するシステムは複数台のサーバで構成されている. クライアントからサーバへのリクエストは失敗することはない. なぜなら, クライアントからのリクエストは全てのサーバへ送信されるからである. つまり, 中央集権的なサーバはなく, サーバの中の一つが故障したとしても, 他のサーバもリクエストを受け取っているのでリクエストは受理[1]される.

送られたリクエストは全てのサーバで計算され, その結果は各々クライアントに返信される. クライアントはいくつかのサーバからの返信を Voting し結果を得る.

耐故障性システムを構築する際には**t fault-tolerant** を考慮する必要がある. この定数 t は, あるインターバルの間に faulty になるコンポーネントもしくは, サーバが t 以上にはならないという指標である. これは, システムの設計時にどのレベルの故障まで耐えるシステムにするかということを踏まえて決定される. この指標を元にしてサーバの台数などを決定する.

たとえば, t fault-tolerant システムで単純なサーバの故障による停止を考えるなら $t+1$ のサーバで対応できるが, より悪性の故障に対応するためにはそれ以上の台数を用意する必要がある.

t falut-tolerant の概念は次節の Primary/Backup アプローチにも言えることである.

3.2.2 Primary/Backup アプローチ

Primary/Backup アプローチは複製を用いて冗長性を高める手法の一つであり, Statemachine アプローチとは「あたかも一つのシステムによって実装されたサービス」を提供するという点では同じである. この Primary/Backup アプローチと前述の Statemachine アプローチとの違いはクライアントからサーバへのリクエストは失敗することがあると言うことである.

システム全体は Statemachine アプローチと同様に複数台のサーバから構成されているが, サーバの役割が異なる. Primary/Backup アプローチではサーバの中の一つが Primary として実際の処理を行ない, その他のサーバは Backup として Primary の処理のバックアップを担当する. また, クライアントは Primary のみにリクエストを送る. そのため, Primary が故障して, その故障を検知して新たな Primary を Backup の中から選びクライアントに報告するまでクライアントのリクエストは失敗し続ける. このクライアントからのリクエ

[1] リクエストが正しく処理される準備が整った状態

ストが失敗し続ける時間のことを *Failover-time* と言う.

シンプルな Primary/Backup のプロトコルを図 3.1 に示す.

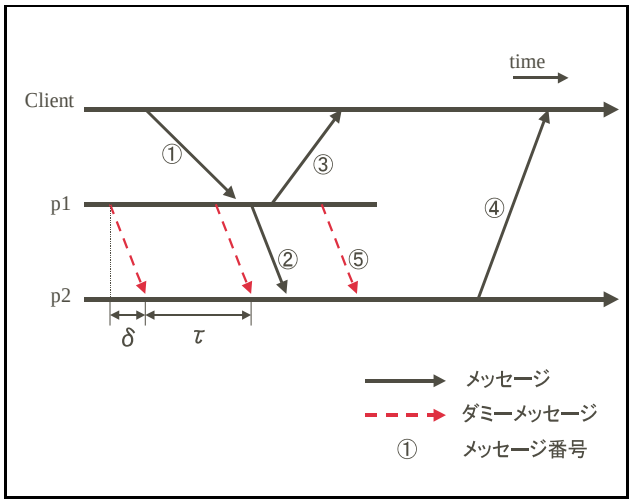

図 3.1: Simple Primary/Backup Protocol

まず, このプロトコルの説明を容易にするため以下の仮定をおく.

- 2 つのサーバプロセス p1, p2 が異なるホストの異なるプロセッサで動作する
- 全てのリンク[2]は point-to-point で結ばれている
- リンクは全て non-faulty リンク[3]である
- メッセージの送信と受信の間の最大メッセージ遅延は δ とする
- Client, p1, p2 が動作しているプロセッサのクロックは同期しているものとする

図中の p1 と p2 はサーバであり, 現在 p1 が Primary サーバで p2 が Backup サーバであるとする. p1 は τ 秒毎にダミーメッセージを p2 に送る. もし, p2 が $\tau + \delta$ 秒でそのメッセージを受け取らなかった場合, p2 は Primary サーバになる.

①のメッセージはクライアントからのリクエストである. p1 がリクエストを受け取るときはいつでも以下の動作を行なう.

1. リクエストを処理し, 情報を更新する
2. 更新した内容を p2 に送る. このメッセージ②を**State Update メッセージ**と言う.
3. p1 は p2 からの受取り通知を待たずにクライアントに応答 (メッセージ③) を返す

図 3.1 では p1 がメッセージ③を p2 に送り, ダミーメッセージ⑤を送信した後何らかの故障が起っている. この場合, p2 はその後ダミーメッセージを受け取れないので Primary サーバとなり, メッセージ④をクライアントに送信して Primary サーバの変更を通知する.

Primary サーバが故障したことは, ダミーメッセージの間隔は最大 $\tau + 2\delta$ なので最悪でも $\tau + 2\delta$ 秒後には分かる. つまり, Failover Time はクライアントへ Primary サーバが変更したことを通知する時間を含め $\tau + 3\delta$ になる.

Primary/Backup アプローチは以上のような手順で故障を発見しクライアントに通知する. このアプローチは Statemachine アプローチよりコストが低く一般的である.

3.3 Group Communication

本節では Group Communication の概要と現存する Group Communication ツールキットの例を解説する.

[2]ホスト間を結ぶ物理的な媒体

[3]リンク障害が起らないリンク. 実際には存在しないが, 説明を簡単化するための仮定として用いられる.

3.3.1 概要

Group Communication は一般的に**メンバシップ・サービス**と**マルチキャスト・サービス**からなる分散システムを管理するツールキットである. 特に, プロセスをグループとして管理し, プロセス間通信においてグループへのマルチキャストなどを提供する.

Group Communication ではメッセージ通信は基本的に非同期で行なう. しかし, それではメッセージ順序が受け取り側によって異なってしまう可能性がある. Group Communication ではこの問題の解決策として**仮想同期 (Virtual Synchrony)**[Bir96](下図 3.2) を使ってあたかも同期してメッセージ通信を行なっているかのように見せかけている.

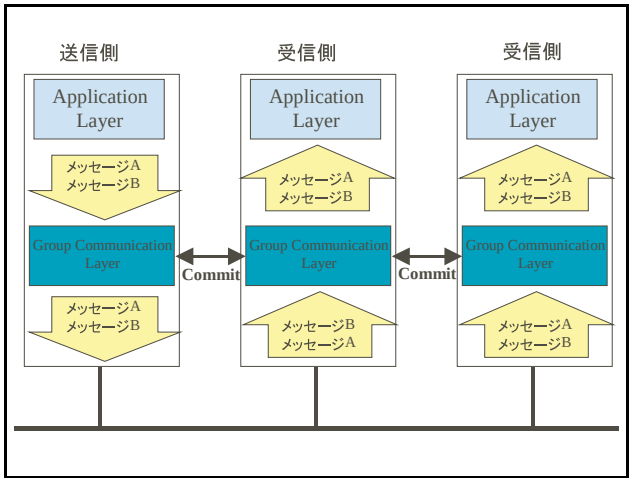

図 3.2: 仮想同期

メッセージの送信順番の決定は理論的に**グローバル時間順 (grobal time ordering)**[4]で行なわなければならない. しかし, これは実装において非常に困難であるため, 多くのシステムでは, ある起点を設けてそれを基に順序を決定する**一貫性時間順 (consistent time ordering)** を用いている.

メンバシップ・サービス

メンバシップ・サービスはプロセスの参加および脱退に加え, グループの生成や削除を行なう.

このサービスでは, グループに関するあらゆる情報を管理する. 各々のグループにはグループの識別子またはアドレスが割り振られており, 特定のグループのプロセスを指定す

[4]実時間において送信した順番

ることもできる.

グループのメンバは同じホストにいる必要はなく, しばしば分散している. もちろん, 分散したホストでもグループのメンバを管理する必要があり, また, 全てのホストで各グループのメンバに関して一貫性を保つ必要がある.

また, プロセスは一定間隔でサーバに生存していることを知らせるための通知を行なう. これによってプロセスが故障した場合, サーバがその通知が届いていないことを検出してグループメンバを変更する. もし, あるプロセスが故障したことを他のプロセスが知らなかったなら, コミットの処理はグループ内の全プロセスの返信を待つため, 永久にブロックし続ける. グループメンバを変更することによってこの問題を防ぐことが出来る.

マルチキャスト・サービス

マルチキャスト・サービスはメンバシップ・サービスをもとにグループメンバへのブロードキャストを提供するサービスである.

しかし, メッセージ通信においては順序の入れ替わる問題が頻発するのでマルチキャスト・サービスでは様々なポリシーを持つブロードキャストを提供している. 以下に, 代表的なブロードキャストを紹介する.

- FIFO ブロードキャスト
  メッセージ msg_1, msg_2 の順にブロードキャストしたなら msg_1, msg_2 の順で受け取るか, msg_1 か msg_2 のどちらかしか受け取らない

- Causal ブロードキャスト
  メッセージ msg_1 のブロードキャストが msg_2 より因果的に先行しているなら msg_1 が受け取られるまで誰も msg_2 を受け取らない

- Atomic ブロードキャスト
  受け取ったメッセージは全てのプロセスが受け取っているものとする. もし, あるプロセス p がメッセージ msg_1 を受け取っていなければ, 全てのプロセスは msg_1 を受け取れない

3.3.2 既存の Group Communication

ISIS

ISIS システム [KPB][Bir96] はコーネル大学の K.Birman らによって開発された. このシステムは株取引を調停する分散アプリケーションなどを構築している. ISIS は完全なオペレーティングシステムというよりも UNIX などの既存のオペレーティングシステム上で動作するプログラムの集合である.

ISIS の通信プリミティブは以下のものが提供されている.

- ABCAST
  通常のグループメンバへの通信やデータ転送に使われる.
- CBCAST
  仮想的な同期通信を提供する. 因果関係も考慮してメッセージ順序を決定する.
- GBCAST
  ABCAST に似ているが, データを転送するためではなくグループのメンバシップを管理するために使用される.

Transis

Transis システム [Ami] は ISIS と同じく Group Communication サービスを提供するシステムである. このシステムは図 3.3 のような形状を持っている.

Transis システムではホストにおけるプロセスの管理はホストに常駐している**Transis daemon** によって管理されている. プロセスは複数のグループに属してもよい. Transis daemon は管理しているプロセスがどのグループに属しているかを知っており, 特定のグループへマルチキャストが行なわれた場合には, Transis daemon から該当するグループに属するプロセスへ配送される.

また, Transis は以下の API を持っている.

- **Connect:**
  Transis への接続を始める IPC(Inter Process Communication) ハンドルをユーザプロセスに作成する. 一つのプロセスは複数の接続を作れる.
- **Disconnect:**
  接続を終了する.

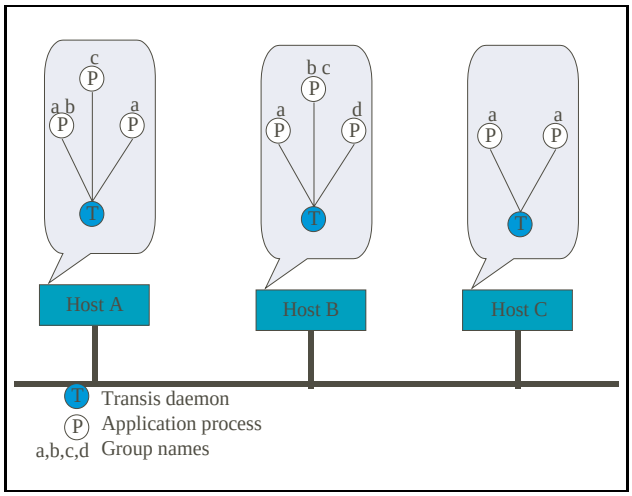

図 3.3: Transis システム

- **Join:**
  接続した状態でプロセスのグループに参加する．これにより membership notification[5]が発生する．

- **Leave:**
  プロセスが自発的にプロセスのグループから抜ける．

- **Multicast:**
  特定のグループにメッセージをマルチキャストしその配送を保証する．また, 配送される順番も保証される．

- **Receive:**
  配送されたメッセージをプロセスが受け取る．

3.4 エージェントシステムへの適用

3.4.1 概要

Primary/Backup アプローチを用いて[6]エージェントの複製を作ることでエージェントの耐故障性を高める．しかし, 複製を作ると同じ処理を複数回実行してしまう可能性があ

[5]メンバの変更が生じたときに発行される命令

[6]Primary/Backup アプローチはクライアントにかかる負荷が Statemachine アプローチより少ないことからこのアプローチを用いる．

る. これはエージェントの信頼性を低くする原因である. この問題に対して耐故障性機構も備えた GroupCommunication を用いてエージェントをグループ化して管理することによって実際に処理を行なう primary エージェントが高々1 つであることを保証し, エージェントの耐故障性と信頼性を高める.

Group Communication は本来プロセスのグループを管理するものであるが, 本研究においてはエージェントのグループを管理する. エージェントのグループはユーザが生成したエージェントとユーザまたは, システムが複製したエージェントから成る. エージェントのグループはユーザが生成したエージェントとシステムが複製したエージェントから成る. ユーザが生成したエージェントを Primary エージェント, 複製されたエージェントを Backup エージェントと呼ぶ.

これによってグループ内で Primary エージェントが高々一つであることを保証する.

3.4.2 問題点

Group Communication はプロセスのグループを管理するために考案されたシステムであるため, グループのメンバが移動するという概念を持っていない. つまり, エージェントは移動するためエージェントが移動した後に移動元のホストではそのエージェントが故障したと見なしてしまう. 同じく移動先に到着した場合でも新しいメンバを追加する動作が生じる. よって, エージェントの移動時と到着時は常にグループの変更が行なわれることになる. グループの変更は各ホストのサーバのデータと一貫性を持たせるためにコミット等を含む複雑なプロトコルを用いている. それゆえグループの変更行なうのは非常にコストが高い操作である. そのため, エージェントにそのまま適用することは現実的ではないと考えられる.

3.4.3 適用方法

エージェントの移動を開始する直前にローカルホストのサーバに移動することを通知し, そのホストでの障害処理を停止する. 次に到着した場合, 移動元のホストに到着したホストのアドレスを通知することによって再び移動先でも障害処理を再開するように変更を加える.

これによってエージェントが移動することによるメンバ変更は行なわれないので, 比較的効率よくエージェントの移動が行なわれる.

第 4 章

モバイルエージェントのための耐故障システム

本章においては，本研究で提案するシステムの全体的な構造について述べる．

4.1 概要

エージェントを用いたアプリケーションを実現するためには, エージェントシステムをより頑健なシステムにする必要がある．なぜなら, エージェント自体が処理の実行中や移動中の環境の変化に影響を受けやすいからである．特に, 処理実行中のプロセッサの故障や移動中のリンク障害はエージェントのプロセスが消滅してしまうので深刻な問題である．

本機構はエージェントシステムに障害処理を組み込むことでエージェントの耐故障性を高めることを目的としたものである．

また, エージェントシステムを特定せずにどのエージェントシステムでも容易に同じサービスを組み込めるように考慮している．

4.2 構成

本機構は以下のような構成となっている．

- エージェントシステム用パッケージ”FTA”(以下 FTA)
- サーバプロセス”GroupManager”(以下 GM)

前者はエージェントに耐故障性機構を追加するためのパッケージである．このパッケージをエージェントシステムに追加することによって，プログラマはエージェントをプログラムする際に障害処理を意識することなく耐故障性を実現できる．また，このパッケージはエージェントシステム毎に存在し，パッケージをエージェントシステムに追加することで異なるエージェントでも同様のサービスを提供することができる．

後者はエージェントを管理するためのサーバである．このサーバはホストに一つだけ起動しており，そのホストにいる耐故障エージェント (FTA をインポートしているエージェント) の管理をおこなう．

4.2.1 動作概要

ここでは, 本機構の動作を概観する．まず, 必要な語彙の定義を以下に行なう．

語彙定義

プライマリエージェント (Primary Agent): ユーザが新たに生成したエージェント

複製エージェント (Backup Agent): ユーザまたは, GM の命令によってプライマリエージェントを元に複製されたエージェント

グループ (Group): 一つのプライマリエージェントとそれから派生した複数の複製エージェントを含んだ集合

耐故障エージェント (Fault-Tolerant Agent): パッケージ FTA をインポートして作成したエージェント．これ以降, 単にエージェントと記述されている場合, 耐故障エージェントを指すものとする．

つぎに, 動作の全体的な概要を説明する．ユーザがパッケージ FTA をインポートしたエージェントをエージェントシステムによって作成したときに, エージェントはそのことを GM にメッセージを送信し通知する．この通知により GM は新たなグループとエージェント識別子 (Agent_ID) を生成し, そのエージェントに返信する．

このエージェントは生成された後, GM へ様々なメッセージを送信する．メッセージの種類は以下の 2 種類がある．

- 通常メッセージ (Regular Messages)
  エージェントが GM へ自身の持つ情報や状態の変化を通知するもの. 単にメッセージと記述している場合にはこちらを指す.

- ハートビートメッセージ (HeartBeat Message)
  エージェントがホスト上で存在していることを GM に通知するもの. ある一定の間隔で GM に送られており, このメッセージが一定時間内に GM に届かなかった場合, GM 側ではそのエージェントは消滅したと認識される. このメッセージは GM 同士も行なう.

サーバプロセス GM は一つのホストに一つだけ起動し, 他のホストの GM とハートビートメッセージで生存確認をする.

エージェントは生成された後ハートビートメッセージを GM に送信する. ハートビートメッセージには Agent_ID が含まれており, GM 側では一定間隔でハートビートメッセージが届いているかどうかを調べる. もし届いていない場合にはそのエージェントが故障したと見なし障害処理を行なう.

エージェントが移動する場合には GM に Dispatch メッセージを送信し移動することを通知する. エージェントが移動先に到着した場合には移動元の GM に Arrive メッセージを送信して到着したことを通知する. GM 側ではこのエージェントに関して Dispatch メッセージと Arrive メッセージの間は行なわない.

プライマリエージェントが複製エージェントを作った場合, 複製エージェントは Join メッセージを GM に送信する. GM はこのメッセージによりプライマリエージェントが属するグループの新しいメンバとしてこの複製エージェントを加える.

エージェントが故障した場合は GM 側で検知し, 障害処理を行なう. エージェントは一連の処理を終えたあと Complete メッセージを GM に送信する. GM はこのメッセージが送信された場合, そのエージェントが属するグループを消去し管理対象から外す.

4.2.2 システムアーキテクチャ

本機構のシステムアーキテクチャを図 4.1 のように定義する. *Mobile Agent Application Layer* と *Agent System Layer* は一般的なエージェントシステムの構造である. 本機構は *Group Management Layer* で障害処理のサービスを提供する.

Group Management Layer は下位層である *Group Communication Layer* と上位層である *Agent Management Layer* から構成されている. 下位層は主に通信やグループの管理等

を担当し, 上位層はエージェントに関する障害処理やトランザクションの重複実行の防止等を担当する.

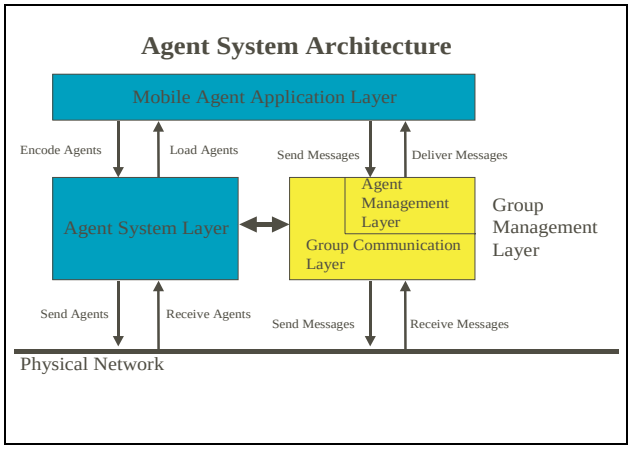

図 4.1: 本機構のシステムアーキテクチャ

4.2.3 特色

本機構はエージェントシステムに依存していないため，エージェントシステムの枠にとらわれずに同じサービスを提供することが可能である．異なるエージェントシステムに本機構を適用するためには，そのエージェントシステム用のパッケージをエージェントを作る際にインポートするだけで良い.

本稿においては 6 章で例として AgentSpace への実装に関して述べる.

また, エージェントタイプの異なるエージェント同士である場合, 現行のシステムではプロトコルの違いなどから相互通信することはできないが, Group Management Layer を用いてプロトコルを統一することによって通信が可能となる.

4.2.4 故障モデル (Failure Model)

障害の種類は以下のようなものに分類される.

- Crash 障害
  Fail-stop 障害とも呼ばれ, 正常なシステムが単に停止し応答しなくなる障害. 最も良性な障害である.

- Crash+Link 障害
  Crash 障害に加え他のホストとの通信リンクが断線等の故障で通信不能になる障害.

- Omission 障害
  システムは正常に動作しているように見えるが, 送信されるべきイベントやメッセージが送信されなかったり, 受信されるべきイベントやメッセージが受信されなかったりする障害.

- Timing 障害
  システムは正常に動作しているように見えるが, 入出力イベントに対する応答が極端に早かったり遅かったりする障害.

- Byzantine 障害
  Crash, Crash+Link, Ommision, Timing 障害を含む一般的な障害. 障害が発生しても正常に動作しているように見える最も悪性な障害である.

以上を図示したものが下図 4.2 である.
本機構は図 4.2 のようにホストの故障やリンク障害に対応している.

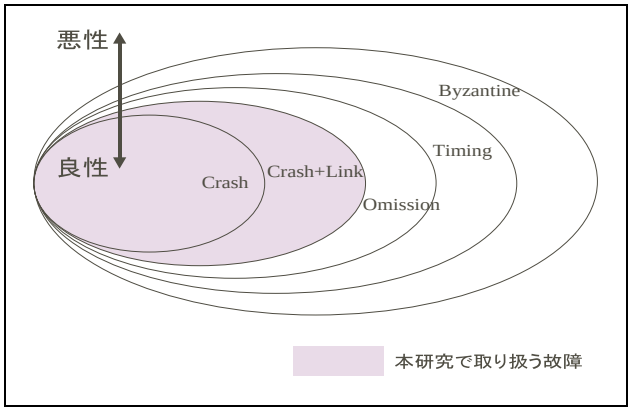

図 4.2: 故障モデル

4.2.5 仮定

本機構は以下のようなエージェントシステムを仮定している.

- エージェントシステムは Java で実装された Java のアプリケーションである
  本研究では Java で実装されたエージェントシステムを対象とする.

- エージェントは Java のアプリケーションである
  エージェントのプログラム中で Java の API が使用可能であることを前提とする.

- エージェントのライフサイクル (状態遷移) は概ね図 4.3 を満たす
  移動可能なモバイルエージェントを対象とする. また, 耐故障性を高めるために複製を作ることから, エージェントシステムにエージェントの複製を作る機能が無い場合は本システムのサービスのフルセットを提供することはできない.

下図 4.3 はエージェントの一般的な状態遷移 (ライフサイクル) を表したものである．本機構はこの状態遷移を元に障害処理をマッピングさせており，この様な状態遷移を含むエージェント (エージェントシステム) ならば，本機構を適用することが可能である.

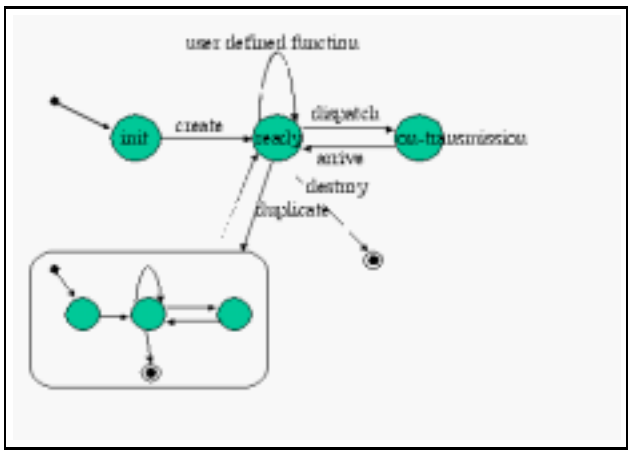

図 4.3: エージェントの状態遷移

第 5 章

Group Communication Layer

本章では, 前章で紹介した Group Management Layer の下位層である *Group Communication Layer* について述べる. また, GM 側で行なう処理とエージェント側で行なう処理とに分けて説明を行ない, それらのインタラクションのためのメッセージプロトコルを述べる.

5.1 概要

この階層では主にエージェントの状態の把握やエージェントのグループ化, グループ化したエージェントへのマルチキャスト等のサービスを提供する. また, エージェントの耐故障性を高めるために複製を作るサービスもこの階層で行なう.

5.1.1 特色

従来の Group Communication は管理対象がプロセスであったため, 管理対象が移動することを考慮されていなかった. 実際, 従来の Group Communication をエージェントに適用するとエージェントが他のホストへ移動することによって障害処理が発生し, グループのメンバ変更が行なわれる. エージェントが移動先のホストへ到着すると, ここでもグループのメンバ変更が行なわれる. これは非常にコストが高い処理となってしまう. また, 同時に従来の Group Communication ではメンバ変更が起って追加されたメンバは新しいメンバと見なされてしまう. しかし, ここでは同一メンバ (エージェント) が同じグループに存在しながら移動しているので, 同一のメンバとして見なしたい.

本機構では, エージェントが移動を始める直前に GM にエージェントの移動を通知し,

そのエージェントへの障害処理を中断する. 次に, 移動が完了したエージェントがエージェントの移動完了を GM に通知することによって移動先のホストで再び障害処理を行なうという仕組みを持っている. つまり, エージェントが移動することによってメンバ変更は起らないので, エージェントが移動してもグループメンバが変更することはない.

5.1.2 GM の構造

GM の構造を図 5.1 のクラス図で説明する. エージェントを管理するためにエージェン

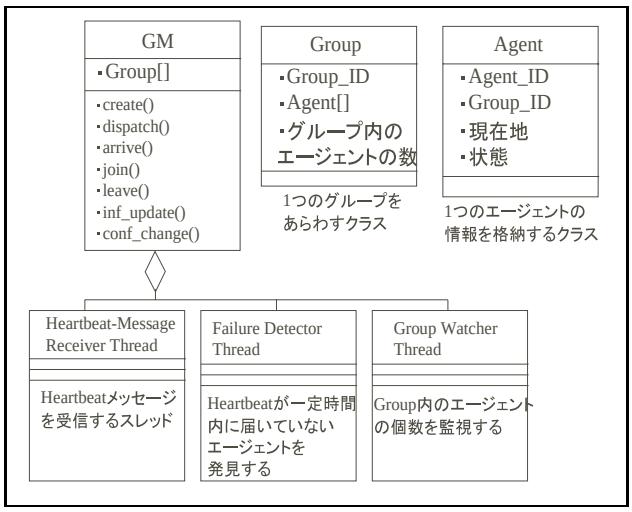

図 5.1: GM の構成

トの情報を格納するクラス Agent が存在する. クラス Agent には Agent_ID, Group_ID, 現在地 (IP アドレス), 状態[1] また, このエージェントをグループ化して扱うために Group_ID とクラス Agent を配列として持つクラス Group が存在する. GM はこれらを包括的に管理するためにクラス Group をインスタンスに持つの配列を持つ.

GM は現在同一ホストにいるエージェントとそのエージェントが属しているグループを管理する.

また, GM は以下の 3 つのスレッドを持ち, これらは GM とは独立して動作している. 役割は以下の通りである.

- Heartbeat-Message Receiver
  ハートビートメッセージを受信するスレッド

[1] 状態は Normal, On-Transmission, Suspect, Faulty がある.

- Failure Detector
  エージェントからのハートビートメッセージを監視し, 一定時間内に受信していないエージェントを発見する

- Group Watcher
  グループ内のエージェントの数を管理する. 故障によって減った分を Primary エージェントに複製を作らせることで補う.

5.2 障害処理のための手続き

本節においては，耐故障性モバイルエージェントを実現するための障害処理の手続きについて述べる.

手続きに関しては実装上大きく 2 つに分かれる．一つはモバイルエージェントが GM に対して発行する要求であり，もう一つは GM 側での要求に対する処理，または，エージェントへの応答である.

この要求と応答の詳しいプロトコルについては次節 (4.3) に述べる.

5.2.1 GroupManager における手続き

ここではサーバプロセス GM が行なう手続きについて述べる.

Create procedure

Create 手続きはエージェントから Create メッセージを受信することによって呼び出される. まず, Agent_ID を生成する. つぎに, 新たなグループを初期化し, そのエントリの一番最初に生成した Agent_ID を書き込む. 最後にエージェントへ Agent_ID を返信する.

Dispatch procedure

Dispatch 手続きはエージェントから Dispatch メッセージを受信することによって呼び出される. Dispatch メッセージに含まれる Agent_ID を読み取り, その Agent_ID を持つエントリのエージェントの状態を *Normal* から *On-Transmission* に変更する.

Arrive procedure

Arrive 手続きはエージェントから Arrive メッセージを受信することによって呼び出される. Arrive メッセージに含まれる Agent_ID と現在地 (Current) を読み取り, その Agent_ID をもつエントリのエージェントの状態を On-Transmission から Normal へ変更し, 現在地を上書きする.

このあと, 変更情報を各メンバを管理している GM へ通知するため後述の *Information Update procedure* を呼び出す. Configuration Change は行なわない.

Join procedure

Join 手続きはエージェントから Join メッセージを受信することによって呼び出される. ここでは Create メッセージと同じく Agent_ID を発行してエージェントに返信する. この変更を他のホストの GM に伝達するために Configuration Change を呼び出す.

Leave procedure

Leave 手続きは GM の Failure Detector スレッドがエージェントの故障を検出した際に呼び出される. この手続きはグループメンバを変更するための処理を行ない, その変更を他のホストの GM に伝達するために Configuration Change を呼び出す.

Complete procedure

Complete 手続きはエージェントからの Complete メッセージを受信することで呼び出される. Complete メッセージはエージェントの一連の処理が終了したことを表すので, そのエージェントが所属するグループおよびグループメンバを全て消去する.

Duplicate procedure

Duplicate 手続きは GM の Group Watcher スレッドがグループの数が故障によって減少したことを検出した場合に呼び出される. この手続きによって Duplicate メッセージがそのグループの Primary エージェントに送信され Primary エージェントは複製を一つ生成する.

Infomation Update procedure

エージェントが移動を完了して Arrive メッセージを送った後, 現在地の情報の変更をそのエージェントが所属するグループを持つホストへ伝達する.

Configuration Change procedure

Configuration Change は何らかの原因でエージェントのプロセスが故障してグループのメンバが減ったり, Primary エージェントが複製を作って新しいメンバが増えたりした場合に呼び出される手続きである. つまり, グループのメンバが変更する毎に呼び出される.

Configuration Change の手続きを図 5.2 に示す[2].

図 5.2 について簡単に説明する. 1 行目の引数に与えている com は Configuration Change によって行なわれる操作を示しているものである. つまり, メンバを追加 (join) するのか, 削除 (leave) するのかのどちらかが com という変数の中に入っている. また, groupNo というのは Configuration Change の対象となっているグループが格納されている配列の添字を示している.

変数の info は変更内容を送信先に通知するためのものである. また, decision はコミットの結果を示す変数[3]が入る.

8 行目から 10 行目までの繰り返しは, マルチキャストするために対象グループに所属する各エージェントの現在地 (IP アドレス) を集めている. joinGroup() はマルチキャストするための送信先アドレスを登録する関数である.

12 行目ではコミットを行なっている. 送信先は対象グループに所属するエージェントの現在地の GM である. もし, コミットが失敗 (abort) したなら, 11 行目に戻りもう一度コミットを試みる. この場合のコミットは, ただ単にメッセージをマルチキャストし, コミットの対象がコミット可能であるかどうかを確認するだけである.

その後, 16 行目では更新情報のマルチキャストを行なっている. また, 17 行目では更新情報を送った先からの Ack を待っており, 全ての Ack が帰ってきた場合には Configuration Change が終了したことを更新情報の送信先に通知する. Ack を待っている間にタイムアウトが成立してしまった場合には, もう一度 11 行目からやり直す.

この手続きでは 11〜24 行目にかけてグループのメンバを更新するためにコミットを使った Atomic ブロードキャストを行なっている.

[2]図中の || は文字列を連結する中置演算子
[3]成功 (success) か失敗 (abort) が入る変数

Configuration Change procedure

```
01 procedure conf_change(com, agentID, groupNo);
02    var msg := φ;
03       info := agentID;
04       groupNum := Group_Array[groupNo].num;
05       decision := φ;
06       count := 0;
07    begin
08        for i:= 0 to length of Agent_Array;
09            joinGroup(Group_Array[groupNo].Agent_Array[i].addr);
10        endfor
11    label Abort;
12       decision := commit();
13       if decision = "abort";
14         then goto Abort;
15       endif;
16             sendMulticast(msg || . || com || . || info);
17       repeat receive(ack);
18           count := count + 1;
19       until timeout;
21       if count < groupNum;
22           goto Abort;
23       endif;
24       sendMulticast(decision);
25    end;
```

図 5.2: Configuration Change procedure

5.2.2 エージェントにおける手続き

ここでは, 耐故障エージェントが行なう手続きについて述べる．ここで説明する手続きは, エージェントシステム毎に作成するパッケージとして実装されている.

エージェントが障害処理のために行なう手続きは以下のものがある.

- Create
- Dispatch
- Arrive
- Join
- Complete

また, GM がエージェントの障害処理をする為にはエージェントの状態などの情報を知る必要がある. エージェントは GM に通常メッセージによって状態の変化や更新した情報を通知する.

エージェントから GM への通常メッセージは以下のものがある.

- Create メッセージ
  Create 手続きによって GM に送信される．エージェントが生成されたことを通知する
  - メッセージの形式
    メッセージシグナル
- Dispatch メッセージ
  Dispatch 手続きによって GM に送信される. エージェントが他のホストへ移動することを通知する
  - メッセージの形式
    メッセージシグナル + Agent_ID
- Arrive メッセージ
  Arrive 手続きによって移動元の GM に送信される. エージェントが移動先のホストへ到着したことを通知する

  - メッセージの形式
    メッセージシグナル + Agent_ID + 現在地 (IP Address)

- Join メッセージ
  Join 手続きによって GM に送信される. 複製エージェントが生成されたことを通知する

  - メッセージの形式
    メッセージシグナル + Group_ID

- Complete メッセージ
  Complete 手続きによって GM に送信される. エージェントの一連の仕事が完了したことを通知する

  - メッセージの形式
    メッセージシグナル + Group_ID

Create procedure

手続き Create は, ユーザが新しいエージェントを生成したときにエージェント側で呼び出される手続きであり, GM にエージェント識別子 (以降 Agent_ID) を要求する. この Agent_ID は本機構独自のものであり, エージェントシステム固有のエージェント識別子とは全く関係のないものである. Agent_ID は" HostAddress.Group_No.Agent_No" という形式を取っている. この ID はエージェントが移動しても変ることはなく常に一意に特定のエージェントを指し示している. Group_No はグループを格納する配列の添字で GM における Create の手続きで新たに作成されたグループが格納された場所である. Agent_No も Group_No と同じくそのエージェントの情報が格納された配列の添字を表している. エージェントが他のホストに移動したときは GM が適当な空き場所にグループやエージェントの情報を格納するため, Group_No と Agent_No は共に一意にそのグループやエージェントを指し示すものではない. Create 手続きは図 5.3 の通りである.

Create 手続きは先ず, GM に Create メッセージを送信する. Create メッセージにはメッセージシグナル (sig) と返信を受けるための番号 (reqNo) が含まれている. reqNo はメッセージの固有番号の代りとして使っている. エージェントが自分宛のメッセージかどうかを確認するために GM は送信対象の Agent_ID を添付してメッセージを送信する. しかし, この時点では Agent_ID は決定していないのでその代りに reqNo という番号を付けて送信

し, GM はこの番号を resNo として返信する. これにより, Agent_ID が無くてもエージェントが適切なメッセージを受け取れるようにしている.

その後 GM からの返信を受信するまではブロックする. GM からの返信待ちはタイムアウトが設定されており, タイムアウトが起きるとエージェントは Create メッセージを再送する. GM から resNo と Agent_ID が返信される. 基本的に GM からのエージェントへのメッセージの送信や返信はローカルホスト上の全てのエージェントへブロードキャストされる.

reqNo の値は説明の簡単化のためランダムな値を代入するように書いてあるが, 実際はランダムな値が衝突する可能性があり, 時刻等の様な衝突する可能性が低い値を代入するようにしている.

GM からの返信を受信した場合は, その返信に含まれる Agent_ID と Agent_ID から導き出した Group_ID を格納する. 図中の getGID() という関数は Agent_ID から Group_ID を返す関数である.

```
procedure create_agent();
    var Agent_ID := φ;
        Group_ID := φ;
        sig := "create"; /* message signal */
        reqNo := rand(); /* peculiar number of this message */
    begin;
        send(localhost, sig, reqNo);
        while timeout do
            waitfor receive(resNo, msg);
                if reqNo = resNo;
                    then break;
                endif;
        endwhile;
        Agent_ID := msg;
        Group_ID := getGID(Agent_ID);
    end;
```

図 5.3: Create procedure

Dispatch procedure

手続き Dispatch はユーザまたは, エージェント自身が他のホスト (または, 同一ホストの異なるエージェントシステム) への移動要求をエージェントシステムに発行したときに呼び出される (つまり, 他のホストに移動する直前).

Dispatch 手続きは GM に Dispatch メッセージを送信する. Dispatch メッセージにはメッセージシグナルと Agent_ID が含まれる. ここでの通信は非同期である.

Dispatch 手続きの例を以下 (図 5.4) に示す. また, 非同期通信である Arrive, Leave, Complete の各手続きは同様にして行なわれる.

```
procedure dispatch_agent(Agent_ID);
  var sig := "dispatch";
  begin;
    send(localhost, sig, Agent_ID);
  end;
```

図 5.4: Dispatch procedure

Arrive procedure

手続き Arrive は移動中のエージェントが目的のホストのエージェントシステムに到着したときに呼び出される.

Arrive 手続きは GM に Arrive メッセージを送信する. Arrive メッセージにはメッセージシグナル, Agent_ID と現在地の IP アドレスを移動元の GM に送信する. この通信も非同期で行う.

Join procedure

手続き Join はユーザまたは, GM がエージェントに複製命令 Join_req(エージェントにその複製の生成を指示すること) を発行したときに生成される子プロセスが一番最初に呼び出す手続きである.

Join 手続きは先ず, GM に Join メッセージを送信する. Join メッセージにはメッセージシグナル, 要求番号と Group_ID を含んでいる. その後, Create 手続きと同様 GM から

の返信が返ってくるまでブロックする. 返信が返ってきたら, そこに含まれている新しい Agent_ID を格納する. ここでも GM との通信にタイムアウトが設定されており, タイムアウトを起こすと Join メッセージを再送する.

Leave procedure

手続き Leave はユーザ, GM, エージェント自身がエージェントを消滅させる直前に呼び出される.

Leave 手続きは GM に Leave メッセージを送信する. Leave メッセージにはメッセージシグナル, Group_ID, Agent_ID が含まれる. この通信は非同期である.

Complete procedure

手続き Complete はエージェントの一連の仕事が終了したときに呼び出される.

Complete 手続きは GM に Complete メッセージを送信する. Complete メッセージにはメッセージシグナル, Group_ID が含まれる.

5.3 障害処理プロトコル

5.3.1 エージェントの障害処理プロトコル

エージェントはハートビートメッセージを一定間隔で GM に送信することで自らの生存を報告する. GM はそのメッセージを受け取ることでエージェントの生存を確認する. このメッセージの送信間隔は決められており, GM がその送信間隔[4] でエージェントからのハートビートを受け取らなかった場合, エージェントは故障したと見なす.

以下に障害処理のプロトコルを示す (図 5.5).

図 5.5 は Host A のエージェントにおける障害処理の様子を記述している.

まず, エージェントからのハートビートが途切れたことによって GM1 は Configuration Change の手続きを行なう. また, これを同じグループのエージェントを持つリモートホストの GM2, 3 に Configuration Change メッセージ (メッセージ①) を送信する. このメッセージには更新した内容が含まれている.

[4]実際はネットワーク遅延などを考慮して少し長めに設定してある.

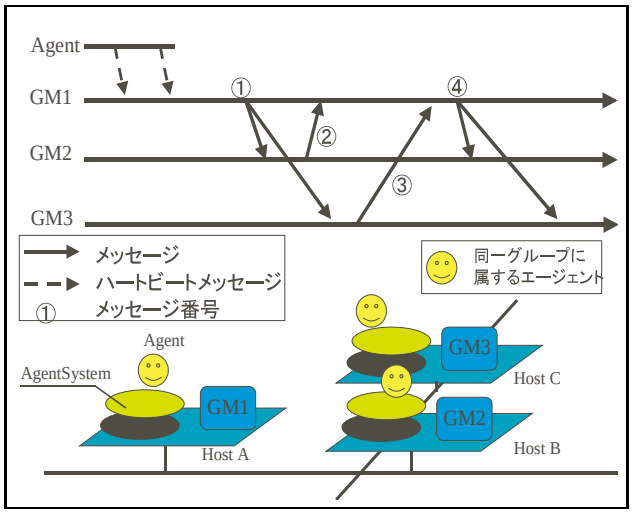

図 5.5: エージェントの障害処理プロトコル

次に, Configuration Change メッセージを受信し, グループメンバの更新を行なったホストからの Ack が返信される. これがメッセージ②, ③である. Configuration Change を送った全てのホストからの返信を受信するまで GM1 はブロックする. ここではタイムアウトが設定されており, 全てのメッセージを受け取る前にタイムアウトが発生した場合にはもう一度①のメッセージを送るところからやり直す.

最後に, メッセージ④を送信してグループメンバの更新が終了したことを通知する.

5.3.2 GM の障害処理プロトコル

GM もエージェントと同じくハートビートメッセージを出して自らの生存を報告している. 但し, 送信先は予め静的に決められているホストの GM である.

以下に障害処理のプロトコルを示す.

図 (?.?) の場合, 予め GM1 には GM2, 3, 4 が, GM2 には GM1, 3, 4 がと言うように各々通信すべき相手が決まっているとする. この場合, 各々が決められたホストへ一定間隔でハートビートメッセージを送信しているが, 説明を簡単にするため GM1 が故障したときの他の GM の動作に焦点を当てるため GM1 のハートビートメッセージしか記述していない.

GM1 からのハートビートが届かないことを GM2, 3, 4 の順番で気づいたとする. 最初に気づいた GM2 は GM1 以外で送信先が決まっているホスト (GM3, 4) に Suspect メッセージを送る (メッセージ①). 同様にして GM3, 4 も各々Suspect メッセージを送信する (メッ

セージ②, ③).

この場合, 全ての送信先から Suspect メッセージを受信した場合に GM1 が故障であると判断する. メッセージ待ちの時間はタイムアウトが設定されており, タイムアウトが起ると GM1 は故障と判断されない.

ネットワークパーティションの検出については予めネットワークパーティションが発生し得るパターン (どことどこが通信不能になったらネットワークパーティションが成立するというパターン) を設定しておく必要がある.

5.4 まとめ

本章では Group Communication Layer で行なう手続きと障害処理におけるプロトコルを一通り解説した. これらを行なうことでエージェントや GM の障害処理を行なうことができる. これによってエージェントの耐故障性を高めることができる.

また, Group Communication Layer で行なう通信はエージェントシステムの API を全く使用していないので, これを応用することによってエージェントタイプの異なるエージェント同士でも通信することは可能である.

第 6 章

Agent Management Layer

本章では本研究で提案するシステムの上位層である Agent Management Layer について述べる.

6.1 概要

モバイルエージェントなどの分散アプリケーションにおいて, トランザクションの一貫性を保つことは非常に重要である. トランザクションの一貫性を保つためにはモバイルエージェントの一意性を保証する必要がある. 特に, 電子商取引に関するアプリケーション等のように, ミッション・クリティカルなアプリケーションにおいてその要求は非常に大きい.

しかし, この要求を満たすことは非常に困難である. なぜなら, 管理対象からの生存確認が途絶えた場合, リンク障害であるかホストの故障であるかをネットワークを介して診断することは事実上不可能だからである. さらに, ネットワークパーティションが発生した場合などは対象不能になることが多い.

また, 本機構ではモバイルエージェントの耐故障性を高めるために複製を用いて実現している為, モバイルエージェントの一意性やトランザクションの一貫性を喪失する可能性がある.

このような背景を踏まえ, 特定の条件下でモバイルエージェントの一意性を保証し, トランザクションの一貫性を保つ手法を提案する.

6.2 Majority Voting

Majority Voting とは Voting の一種で, グループのメンバに投票を行なわせ, その投票を取り仕切る何らかのプロセスが過半数を越える投票結果を優先するという方法である.

まず, 本研究における Majority Voting を以下のように定義する.

n 個[1]のプロセスをメンバに持つグループ P とそれら個々の Voting に対する応答の集合 R, さらに, P と R の関係 (つまり Voting) を以下のように表す.

定義 1

$P = \{p_1, p_2, p_3, ..., p_n\}$

$R = \{a, d\}$ **where** a is a response of agree from a process, d is a respose of disagree from a process

$P \xrightarrow{v} R$

$\forall p \in P$ **where** $v(p) = a \vee v(p) = d$

プロセスのグループ P で Voting 行なうことを $V(P)$ で表す. P に Voting を行なうと, P の全てのメンバはその Voting に対して賛成 (Agree), 反対 (Disagree), いずれかの応答を返すか応答を返さない. 賛成の応答の集合を A, 反対の応答の集合を D とする. 応答なしの場合は反対と見なす. 但し, 非同期システムを想定しているため, 応答は有限時間内に返ってくるものと仮定する.

$A = \{p \in P | v(p) = a\}$

$D = \{p \in P | v(p) = d\}$

今, プロセスグループの中にリーダとなる以下のようなプロセスが存在するとする. このプロセスはグループ内の他のプロセスとは異なる特別な役割を担っている.

リーダプロセス

$P_l \in P$

各プロセスへの Voting とその返答をを以下のように表す (定義 2).

次に Majority Voting において Voting の成功と失敗を以下のように定義する.

[1] n は自然数

定義 2

$v(p_n) = a$ の時 p_n は賛成
$v(p_n) = d$ の時 p_n は反対

Voting の結果の定義

$V(P) = a \Leftrightarrow |D| < |A| \vee |D| = |A| \rightarrow P_l \in A$ Voting 成功
$V(P) = d \Leftrightarrow |D| > |A|$ Voting 失敗

実際, エージェントのグループメンバがネットワークパーティションの中と外に分けられてしまった場合, 通信がとれないためグループの一貫性は喪失してしまう. また, このことは Primary エージェントが 2 つ以上存在し, それらが並行に実行してしまうことを示唆している. Primary エージェントが 2 つ以上存在しそれらが並行に実行されると, 同じ処理を 2 度以上行なってしまう可能性がある.

しかし, ネットワークパーティションが発生したとしても通信がとれる範囲にグループメンバがどれだけの数だけ存在するかを確認することはできる. また, グループのメンバ数は各 GM が把握している. このことを Majority Voting を使って解決する. 但し, Majority Voting を行なっている間はネットワークパーティションが 1 つ以上存在しないことを仮定する.

例として図 6.1 のような場合を考える. Group$B$ のメンバは b_1, b_2, b_3 はである. ネットワークパーティションが起きたことによって b_1, b_2 とは互いに通信できるが, b_3 は b_1 または b_2 と互いに通信できなくなるとする.

この場合, Group$B$ はサブグループ B_1 とサブグループ B_2 に分割されたと考える. 各々のエージェントは元のグループのメンバ数を知っている. ここで, 各々のエージェントが全グループメンバへ Voting を行なう. エージェントが現在属しているサブグループのメンバ数が元のグループのメンバ数に関して過半数 (Majority) の数を確保していれば, Voting は成功し, そのエージェントはそのまま継続して処理を行なう. そうでない場合 (Minority) は Voting が失敗し, そのエージェントは消滅する. 但し同数のサブグループに分割された場合は, Primary エージェント (P_l に相当する) が存在するサブグループを Voting 成功とする.

ネットワークパーティションが起った場合, GM は下図のような手続きで Majority Voting

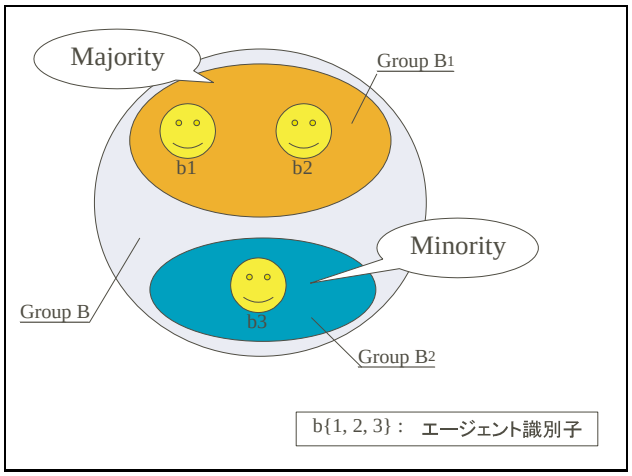

図 6.1: MajorityVoting

を行なう. ネットワークパーティションが起っているかどうかは GM よる障害処理時に検出する.

図 6.2 の Majority Voting について解説する. 図の左端の数字は行番号とする.

まず, 引数には group_ID, groupNo[2], hostAddress をとる.

4 行目で voting として送るメッセージ (msg) を生成している. メッセージの形式はメッセージシグナル+group_ID+hostAddress である. メッセージシグナルはこの場合 vote である. さらに voting の対象となっているグループの識別子 group_ID と返信アドレスである hostAddress を付加している.

次に 7 行目から 9 行目までの for 文では対象となっているグループのメンバをマルチキャストのグループとしてアドレスを登録している. また, 10 行目ではマルチキャストのグループとして登録してあるアドレスへのマルチキャストを行なっている.

11〜14 行目では vote メッセージの返信を待っている. マルチキャストした数の返信を全て受信するか timeout が起るとループを抜ける. ここでメンバの過半数を上回る数の返信を受信していなければローカルホストに存在する対象グループのエージェントは消去されることになる. 17〜23 行目ではローカルホストに存在し, かつ, 対象グループに属しているエージェントにプロセスを終了するためのメッセージを送信している. もし, 過半数を越えていれば対象グループのエージェントはそのまま続行して処理を行なう.

この Majority Voting はネットワークパーティションが一つしか存在しないときは前述

[2]配列 Group_Array の添字, この手続きは voting するためのものなので対象となるグループは予め決まっているものとする.

Majority Voting

```
01 procedure  voting(group_ID, groupNo, hostAddr);
02    var  num:= Group_Array[groupNo].num,
03       agentID:= φ;
04       msg:= "vote" || . || group_ID || . || hostAddr;
05       count:= φ;
06    begin
07       for  i := 0  to  num;
08           joinGroup(Group_Array[groupNo].Agent_Array[i].addr);
09       endfor
10       sendMulticast(msg);
11       repeat
12          waitfor receive(msg);
13          count:= count + 1;
14       until (num < count) ∨ timeout;
15       if count < (num div 2);
16          then
17             for i := 0  to  num;
18                if Group_Array[groupNo].Agent_Array[i].local = 1;
19                   then
20                      agentID := group_Array[groupNo].Agent_Array[i].Agent_ID;
21                      msg:= || "destroy";
22                      send(msg);
23             endfor;
24    end;
```

図 6.2: Majority Voting Algorithm

のように対処できるが, Majority Voting を含む障害処理の間に新たなネットワークパーティションが起ると対処不能である. 本研究では, 障害処理を行なっている間は新たなネットワークパーティションが発生しないことを仮定している.

6.3 Transaction Commitment

Primary エージェントの一貫性が保たれていても 1 つの処理を 2 度行なってしまう可能性はある. 例えば, Primary エージェントがある処理を行なった後ネットワークパーティションで通信不能になってしまった場合, 他の Backup エージェントは Primary エージェントが行なったその処理を知らないのでもう一度行なってしまう可能性がある.

この様な問題が発生しないように重要な処理には行なう前にコミットを行なうこととする. 以下に Two-phase commit を使ったコミットの例を示す.

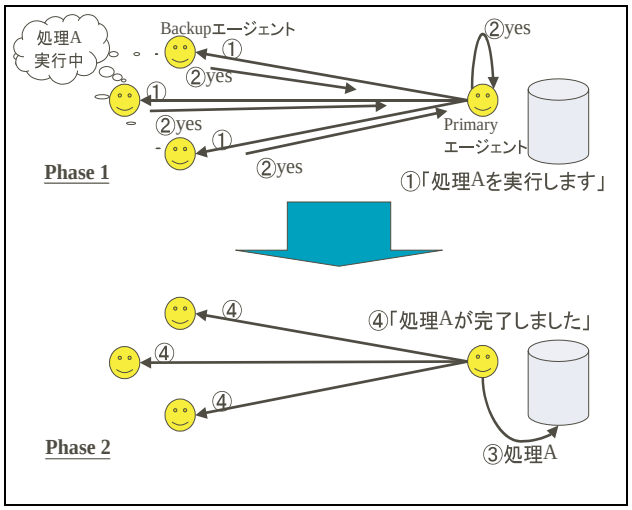

図 6.3: コミット (Two-phase commit) の例

図 6.3[3] のように Primary エージェントが処理 A を実行する前に Backup エージェントに処理 A を実行することを通知する (①). もし, 全 Backup エージェントが yes と回答した場合, エージェントは処理 A を実行し (③), 処理 A が完了したことを Backup エージェントに通知する (④).

Backup エージェントは①から④までの間, Primary エージェントが処理 A を行なっていると思い込んでいる.

[3] 図中の①〜④は処理を行なう順序を示している.

次に, コミットが成立しないケースを 2 つ例として挙げている.

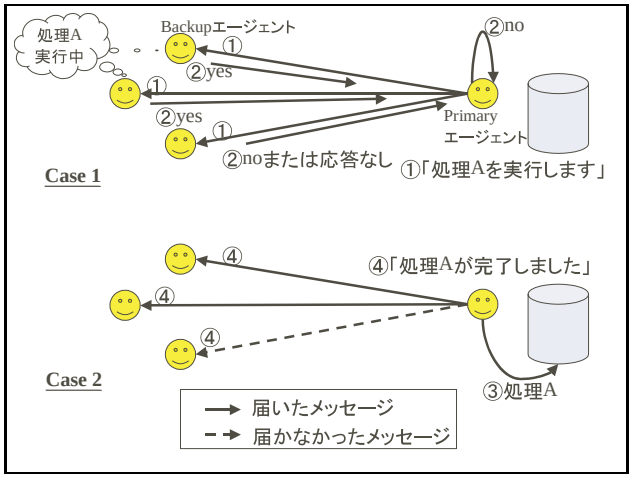

図 6.4: コミットが失敗する例

図 6.4 の Case 1 の場合, 一つのエージェントがコミットの Phase 1 で no または応答がないので Primary エージェントはもう一度最初から Phase 1 を行なう. エージェントが応答するまでこの処理を繰り返す. しかしこの間に, GM の障害処理によって Configuration Change が起こり, 応答がないエージェントがグループのメンバから外された場合はコミットが成立する.

Case 2 の場合, ④メッセージのメッセージが届いていない. これは, Primary エージェントによって例外がハンドルされる. この場合は④のメッセージを再送する.

図 6.4 の Phase 1 が完了して③の処理 A を行なっていなければ Phase 2 を行なっている最中に Primary エージェントがいるホストに障害が発生したり, Primary エージェント自体に障害が発生した場合でも GM が障害処理を行ないグループメンバが変更するので Backup エージェントは Primary エージェントからの④のメッセージを待ち続けることはない. しかし, ③を行なってから故障が発生した場合は対処不能である. ゆえに, ③と④ の間では Primary エージェントがいるホストの障害, または, Primary エージェントの障害が発生しないことを仮定する.

6.4 まとめ

本章では Agent Management Layer について述べた. この階層では Group Communication Layer では対処できない障害 (ネットワークパーティションなど) に対処するための

機構を提供する. 特に, Primary エージェントは高々一つでなければならないというプロパティを喪失する可能性のある障害で Primary エージェントの一意性を保証できる可能性を示唆した.

また, トランザクションの一貫性をグループのエージェント全てに持たせるために, 重要な処理をおこなう前後にコミットを用いることを考えた. この事はエージェントを商用アプリケーションとしている場合, 非常に重要な要素であると考えることができる.

第 7 章

実装

本章では本研究で提案したシステムを実際のエージェントシステム上に実現出来ることを示すためのプロトタイプ実装について述べる.

7.1 概要

本研究で提案したシステムが実際のエージェントシステムに組み込めることを示すためにプロトタイプの実装を試みた. この実装の目的はエージェントシステムの内部機構を変更せずに本システムのサービスを組み込むことが出来ることを示すことである. これを示すことによって他のエージェントシステムへ同じようなサービスを用意に実装できることを表すことができる.

7.1.1 実装環境

- OS: Solaris7
- Java: jdk1.2(Java2)
- 実装対象: AgentSpace

7.2 AgentSpace への実装

本研究で提案したシステムのプロトタイプの実装を AgentSpace に行なった. このプロトタイプ実装はシステムのサブセットであるが, Group Communication Layer の機能をほぼ網羅したものである.

7.2.1 AgentSpace の仕組み

AgentSpace への実装のを説明するためにまず, AgentSpace 自体の仕組みとエージェントの作成方法を簡単に解説する.

エージェントを作成する際には Agent.class を extends して表 7.1 に定義されているメソッドを実装することによって様々なアクションに対応させている.

表 7.1: AgentSpace のコールバックメソッド

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

また, 以下に簡単なエージェントのコードを示す. このコードをコンパイルし, jar 形式に圧縮したものが AgentSpace のエージェントの実体である. このエージェントは単に 110 × 70 のウインドウを作成し, “Hello World” という文字を表示するだけのエージェントである. このエージェントが移動した場合, 移動元のホストからウインドウが消え, 移動先に再び”Hello World” という文字を表示するウインドウが現れる.

7.2.2 パッケージ FTA の実装

パッケージ FTA は以下のクラスから構成されている.

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

public class HelloWorldAgent extends Agent {
    public HelloWorldAgent() {}
    public void create() {
        add(new Label(``Hello World''));
        setSize(110, 70);
        show();
    }
    public void arrive() {show();}
    public void dispatch(URL url) {dispose();}
    public void destroy() {dispose();}
    public void duplicate() {dispose();}
    public void child(AgentIdentifier paid) {show();}
    public void parent(AgentIdentifier caid) {show();}
}
```

図 7.1: HelloWorldAgent(通常のエージェント)

- FTA
  Agent クラスを拡張したクラス. 耐故障エージェントを作る場合このクラスを extends する

- MessageSender
  GM に通常メッセージを送信するクラス

- MessageReceiver
  GM からのメッセージを受信するクラス

- HeartBeatSender
  ハートビートメッセージを GM に送信するクラス

エージェントの障害処理を実現するため以下のメソッドを追加した. また, 5.2.2 で述べたエージェントの障害処理のための手続きとは以下のように対応づけられている.

表 7.2: エージェントの障害処理の手続きとの対応表

メソッド名	障害処理の手続き (5.2.2)
void sendCreate()	Create 手続き
void sendDispatch(String Agent_ID)	Dispatch 手続き
void sendArrive(InetAddress sourceHost, String Agent_ID)	Arrive 手続き
void sendJoin(String Group_ID)	Join 手続き
void sendComplete(String Group_ID)	Complete 手続き

7.2.3 耐故障エージェントの作り方

耐故障エージェントを実装するためには FTA クラスを extends してエージェントを作成する. 基本的に障害処理に関する処理は FTA.java で定義されているため, parent を除く全てのメソッドでスーパクラスの同名のメソッドを呼び出している.

このエージェントはホストを移動する毎に最大 5 個までエージェントを複製する. さらに, エージェントがいるホストが消滅してしまった場合にはその分を補うために Primary エージェントが新しいエージェントの複製を作成する.

この実装においてはユーザが意図的にエージェントを消滅させた場合は, このエージェントの処理が全て終了したものと見なし GM はそのエージェントが属するグループのエージェントを全て消滅させる.

```
import agentspace.*;
import fta.*; /* パッケージ FTA */
import java.awt.*;
import java.net.*;

public class FTHelloWorldAgent extends FTA {

  public Hello() {} // Constructor
  public void create() {
    super.create();
    add(new Label("Hello World"));
    setSize(100, 60);
    show();
  }
  public void arrive() {super.arrive(); show();}
  public void dispatch(URL url) {super.dispatch(url); dispose();}
  public void destroy() {super.destroy(); dispose();}
  public void duplicate() {super.duplicate(); dispose();}
  public void child(AgentIdentifier paid) {super.child(); show();}
  public void parent(AgentIdentifier caid) {show();}
}
```

図 7.2: FTHelloWorldAgent(耐故障エージェント)

7.3 考察

本章では AgentSpace へのプロトタイプの実装を試みた. その結果, エージェントシステムの内部機構は殆んど変更せずに本研究で提案したシステムを組み込むことができた. これにより, 他のエージェントシステムへの本システムの実装は比較的容易に行なえると考えることができる.

また, 耐故障エージェントを従来のエージェント作成方法とほぼ同じ方法で作成できることからプログラマは障害処理などを気にする必要はない.

今後の展開としては, Aglets や Bee-gent のような比較的機能の複雑なエージェントシステムへ本システムのフルセットの実装を試みていく予定である. また, フルセットの実装を行なった場合にどれだけの信頼性が得られるかという実験的検証も行なう.

第 8 章

結論

本章では本研究をまとめ, 考察し, 今後の展望を述べる.

8.1 本研究のまとめ

エージェントはホスト間を移動するプログラムでありユーザの元を離れる時間が長い. ゆえに, エージェントの故障は非常に発見しづらく, また, 対処することが困難な問題である.

本研究ではモバイルエージェントの耐故障性を高めるために Group Communication をベースとした耐故障システムを提案した. Group Communication は分散システムの耐故障性を目的としたプロセス管理に用いられている. この技術は, 元々管理対象がプロセスであるため, 管理対象が移動することは考えられていない. つまり, エージェントへ適用するためには管理対象が移動するという概念を追加する必要がある. 本研究ではこの概念を追加した Group Communication を本システムの Group Communication Layer として提案した.

本システムはパッケージ FTA をインポートし通常どおりエージェントを生成するだけで耐故障性の高いエージェントを実現できるものである. ホストなどの故障を回避するためにエージェントの複製を用いている. 複製を用いることによる問題は同じ処理を複数のエージェントが行うことや, 処理の実行後にエージェントが故障した場合, 故障したエージェントが行なった処理を繰り返し行なうこと等が挙げられる. しかし, 本システムではその問題も Agent Management Layer で Primary エージェントの一意性や処理の重複的な実行を防止している.

8.2 本研究の考察

次に本研究で提案したシステムと実装したプロトタイプについて述べる. 本システムを設計する上で重要だったことは, エージェントシステムに依存しない構造を持ち, Java で実装されたエージェントシステムであるなら容易に本システムのサービスを実装できることである. このことは AgentSpace にプロトタイプを実装することによって実証された.

また, 本システムの Group Communication Layer を用いて通信することによって, 現在は実現されていない異なるエージェントタイプのエージェント同士の通信などを実現することを考慮している.

8.3 今後の課題

今後の課題としては以下の点が挙げられる.

- 本システムのフルセットの実装と検証
  本稿ではプロトタイプの実装のみを行ない, 本システムの構造の優卯要請について確かめたが, 今後, 本システムの全体の実装を行なってシステム自身の有用性を確かめる必要がある.

- 耐故障機構の拡張
  既存の多くの Group Communication は機能の拡張性が高く, また, ユーザの要求に柔軟に答えられる設計となっている. 本システムも機能の付加が容易な構造にしていく必要がある. また, ユーザが必要としている障害処理などに柔軟に答えていくことも重要である.

- 悪意のある攻撃に対する対処
  本稿で実装したプロトタイプは処理を実行している途中にエージェントシステムを強制終了した場合, 全てのエージェントが行なうべき処理が終わったと見なしている. しかし, これが悪意のある行為でユーザが意図していない場合は全ての処理が完了しないままエージェントが終了してしまうことになる. このことに対処するため, リモートホストではエージェントを終了できないように変更することが必要である.

8.4 今後の展望

今後の展望としては以下のことが考えられる.

- ユーザによる障害処理の柔軟な選択
- Group Communication Layer を用いた異種エージェント間通信
- 様々なエージェントとの協調動作

エージェントを作る側のユーザは本システムの全ての機能を必ずしも必要としない事が多いと考えられる. よってユーザがエージェントを作成する際に障害処理の機能を選択できる柔軟性を本システムに付加する必要がある. これによってユーザの要求に答えられる信頼性の高いエージェントを生成することが可能になる.

また, 異種エージェント間通信については現在実現されておらず非常に興味深い. 本システムはエージェントシステムに依存しないため様々なエージェントシステムへの適用が可能である. つまり, 異なるエージェントシステムから生成されたエージェント同士ではプロトコルの違いなどから通信は不可能であるが, 本システムが実装されていれば本システムの同一のサービスを用いて通信が可能である. これにより異種エージェントでの協調動作が可能となるかもしれない.

本システムではユーザが生成したエージェントとその複製を一つのグループとしており, それ以外は同じグループに入れることはできない. 様々なエージェントを協調 (並行) 動作させつつ耐故障性を高めるためには本システム自体を拡張する必要がある.

謝辞

本研究にあたって熱心にご指導, ご助言頂きました片山卓也教授, 権藤克彦助教授に深く感謝の意を表し, 心より御礼申し上げます. また, ゼミなどでお世話になったアデルシェリフ助手をはじめソフトウェア基礎講座の助手の先生方に深く感謝致します. 最後に, 本研究を行なうに当たって様々な面で面倒を見て頂いた豊島真澄先輩をはじめソフトウェア基礎講座の皆様方に感謝の意を表します.

参考文献

[Ami] Yair Amir. Replication using group communication over a partitioned network.

[Bir96] Kenneth P. Birman. Building secure and reliable network applications, 1996.

[DBL98] Mitsuru Oshima Danny B. Lange. Programming and deploying java mobile agents with aglets, 1998.

[KPB] Robbert Van Renesse Kenneth P. Birman. Reliable distributed compluting with the isis toolkit.

[MO98] Guenter Karjoth Mitsuru Oshima. Aglets spacification 1.1 draft, 1998.

[MUL] SAPE MULLENDER. Distributed systems second edition.

[OMG97] OMG. Masif (mobile agent specification interoperability and facility), Nov 1997.

[Sat99] Ichiro Sato. Hierarchically structured mobile agents and their migration, 1999.

[河 98] 河口信夫. アドホックネットワークにおけるモバイルエージェントの応用. 名古屋大学大学院 工学研究科, 1998.

[岩 98] 岩井俊弥. Java モバイル・エージェント, 1998.

[山 96] 津田宏 山崎重一郎. Telescpipt 言語入門, 1996.