博 士 論 文

疎結合分散環境における耐故障性と適応性を実現するソフトウェアの構成に関する研究

指導教官　片山 卓也 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

豊島 真澄

2001 年 4 月 19 日

要 旨

本論文では疎結合分散環境における動的な資源割り当ての手法としてRAFT資源管理システムを提案する。RAFTシステムは、関数型の計算パラダイムに基づいた耐故障技術であるAPR複製技術によってスケジューリングされたタスクの、計算資源への割り当てと実行の制御を実現する。

本研究ではまずはじめにAPRの関数起動アルゴリズムの定式化と、APR計算による資源利用に関する考察を行なう。これによって関数型のプログラムを分散環境上で実行する際のスレッドの状態や消費する資源について明確にする。

次にRAFTプロセスを導入する。RAFTプロセスはAPRの論理的な計算活動を、分散環境に存在する計算資源に割り当てるために定義された細粒度のプロセスである。

続いてRAFT資源管理システムを提案する。RAFT資源管理システムは、疎結合分散環境に存在する計算資源に関する情報を管理する。さらに計算を実行するRAFTプロセスに対する資源の公平な割り当てを行い、計算中のプロセスに対する制御も行なうことによって、クラッシュ、バリューの各障害に対して、一貫性を保証しながら速やかなリカバリを行うという特徴を持つ。

論文の最後ではRAFT資源管理システムを適用したAPRの実装について述べる。実装は関数型のプログラミング言語とグループコミュニケーションシステムを用いて行なう。

本研究により、計算のモデル化とプログラムの記述から実装まで、一貫して関数型のパラダイムを用いて耐故障性を保証する手法の有効性と、その分散環境における実装の持つ特徴が明らかになった。

目 次

図 目 次

第 1章

はじめに

1.1 研究の背景と目的

インターネットや組織内でのグループウェアの普及などによって、近年では多くの廉価な計算機がネットワークを介して接続されるようになってきた。このため、これらネットワークに接続された多くの PC の計算能力を用いることにより、組織内だけでなく組織をまたがって大規模な計算を行う試みが多くなされている。

しかしこのように一般の PC を含む疎結合分散環境においては、システムを構成する要素の数が多く、またそれらの性能も均一であるとは限らず、さらに通信経路はしばしば切断される。このような環境において大規模な計算プログラムを実行するためには、計算開始から計算完了までの間にシステム構成要素の一部に障害が発生した場合においても計算を続けられる能力、すなわち**耐故障性**を備えていることが強く求められる。また不均一かつ変化する環境において計算を実行するために、実行時システムは実装環境の変化に適応する能力、すなわち**適応性**を備えなければならない。

耐故障性を高めるための技術については、ハードウェアの分野においてはすでにいくつかの技術が実用化され、計算機システムの信頼性の向上に寄与している。しかし計算機の利用者は計算機システムに対して入力を行い、計算機システムが仕事を完了することを期待しており、利用者に対して計算機システム全体の信頼性を向上させるためには、ソフトウェアによる信頼性や耐故障性の向上もハードウェアのそれと同様に不可欠である。

耐故障性を持ったソフトウェアを実現するための研究そのものは 1970 年頃から行われており、いくつかの要素技術は存在する。既存の耐故障性を持つソフトウェアを作成するための技法の多くは、これら個々の要素技術をシステム内の特定の部分に対して適用し、命令的あるいは手続き的な計算モデルによって表現している。このような命令的計算モデルは動作が直観的で理解しやすいという反面、障害の特定が困難、計算状態の退避や障害が発生する前の正常な状態への回復が煩雑、障害前と後での状態の一貫性の保証が困難、といった欠点が指摘されている。

これらの問題を克服するための技術として、関数型の計算パラダイムに基づいた APR 複製技術が 1998 年に Cherif によって提案された [CK98]。APR では計算モデルから実装まで一貫して、耐故障性を保証するための計算方法を提案している。また APR は耐故障性の保証を行なうと同時に計算完了までの時間の短縮も実現するという特徴を持つ。プログラマは関数型のプログラミングスタイルで問題を記述することによって、並列計算と耐故障性の向上の 2 つの利益を同時に獲得することができるとされている。

現在のところ APR 複製技術に関しては、アルゴリズムと障害への対処方法が提案されているが、分散環境に存在する計算資源の利用方法や分散配置されたプログラム間の通信に関する考察はなされておらず、実装は与えられていない。

これらの背景により本研究ではまずはじめに、論理的な関数を分散環境に存在する PE 上で実行するために、より細粒度の *RAFT* **プロセス**を提案する。また分散環境に存在する資源の管理や RAFT プロセスのこれら計算資源への割り当ておよび起動を行なう *RAFT* **資源管理システム**を提案する。また、分散環境を考慮した詳細な分析と設計を行ない、グループコミュニケーションを利用した実装方法を提案する。これにより、計算モデルから実装に至るまで一貫して関数型のパラダイムに基づきながら耐故障性を保証するシステムについて、その有効性と特徴を明らかにすることを目的とする。

1.2 本論文の構成

本論文ではまずはじめに、第 2 章において耐故障性に関する基本的な用語の定義を行なうとともに、実装で用いるグループコミュニケーションの概念に関して

概略を述べる。

次に第 3 章では、APR 複製技術の基となっている関数型の計算モデルの概要を述べ、APR の計算起動アルゴリズムを実例を用いて紹介する。

第 4 章では、APR アルゴリズムを実装した際の資源利用に関する詳細な考察を行ない、資源割り当てアルゴリズムを実装するために必要な APR 関数起動アルゴリズムの定式化を行なう。

第 5 章ではこれらの考察に基づき、RAFT プロセスを導入する。また、計算資源の管理および RAFT プロセスの計算資源への割り当てを行う RAFT 資源管理システムを提案する。

第 6 章では疎結合分散環境における実装アーキテクチャを示し、グループコミュニケーションシステムを利用した実装例を示す。

第 2章

耐故障技術

本章ではまずはじめに耐故障性に関するいくつかの基本的な用語の定義を述べる。次に、対処する障害の性質を明確に述べるために必要な、障害モデルについて述べる。また、耐故障性を高めるための基本となるいくつかの主要な要素技術について説明を行なう。

本章の後半では、本論文で提案するシステムの実装においてシステム構成要素間の通信とグループ管理を行なうために用いるグループコミュニケーションについて、必要ないくつかの概念を紹介する。

2.1 耐故障性

2.1.1 用語の定義

本研究ではシステムの耐故障性を高めることは主要な要求の一つである。「耐故障」とは、計算機システムの信頼性を高めるための方法の一つであり、一般には以下に示すIFIP ワーキンググループ 10.4 による定義が受け入れられている [Lap92]。

障害 (failure)、誤り (error)、フォールト (fault)：

> 計算機システムが、利用者が期待する動作を行わないとき、正確にはシステムに対して定められた仕様から逸脱した動作を行ったときに、この計算機システムには障害 $(failure)$ が発生したという。誤り $(error)$ とは、障害をもた

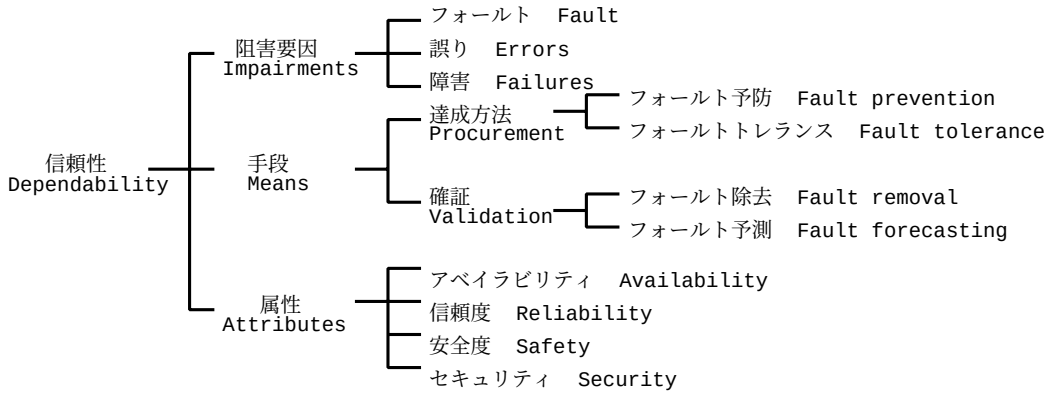

図 2.1: 信頼性に関する用語の定義 (IFIP WG10.4[Lap92])

らす可能性があるシステム内部の状態であり、誤りがシステム外部に影響を与えたときに、それは障害となってあらわれたということになる。そして、誤りの原因はフォールト (*fault*) と呼ばれる (図 2.1)。

耐故障 (fault tolerance)

フォールトが生じても仕様に示された仕事の遂行を可能とする方法。

つまり本研究の重要な目的のひとつである「耐故障性を高める」とは、システム内部にはフォールトが存在することを仮定し、これに対処することによって、許容される範囲内での実行を保証するということである。

2.1.2 障害モデル

システムの障害はその要因と考えられる箇所によっていくつかに分類することができる。この分類を用いることによって、システムがどのような障害に対して耐故障性を保証するのかという範囲を明確に定義することができる。障害モデルおよび障害のクラス分けには文献により様々なものが存在するが、分散システムの障害に関して特に注目している Schneider による障害モデルの分類 [Sch93] を以下に示す。

- **Failstop** プロセッサは停止することによって障害を起こす。プロセッサの状態は、停止以後変化しない。障害の発生は、他のプロセッサによって検出される。

- **Crash** プロセッサは停止することによって障害を起こす。プロセッサの状態は、停止以後変化しない。障害の発生は、他のプロセッサによって検出されない。

- **Crash+Link** プロセッサは停止することによって障害を起こす。プロセッサの状態は、停止以後変化しない。通信リンクも障害を起こし、いくつかのメッセージは失われが、メッセージの遅延や壊れた内容のメッセージが送受信されることは無い。

- **Receive Omission** プロセッサはメッセージの一部のみしか受け取らない。もしくは、メッセージの一部のみを受け取った状態で停止する。

- **Send Omission** プロセッサはメッセージの一部のみしか送信しない。もしくは、メッセージの一部のみを送信した状態で停止する。

- **General Omission** プロセッサはメッセージの一部のみしか受け取らない。もしくはプロセッサはメッセージの一部のみしか送信しない。上記のいずれかの状態か、その状態で停止する。

- **Byzantine Failures** プロセッサは仕様と異なった任意の振る舞いを起こして停止する。

以上の障害モデルの分類の各々おいて耐故障性を保証するのは、一般に後にあげた項目ほど困難である。特に Byzantine Failures は任意の振る舞いまでをそのクラスに含んでおり、このクラスに含まれる全ての障害に対処するのは不可能である。

しかし Byzantine Failures をさらに分類することによって、その一部に対処することができる。[Jal94] では Byzantine Failures のサブセットである、*incorrect computation* モデルを定義している。

- **incorrect computation** プロセッサは入力に対して誤った出力を返す。

本論文ではこの障害モデルを**バリュー障害モデル** (*value failure model*) と呼ぶ。

障害に対処するシステムを構築するにあたり、障害がいくつ発生した場合までを仮定するかということについて、その数を明確に示したい場合が多い。このため t 個の障害に対処するシステムを、*t-fault tolerant* システムと呼ぶ。例えば 2 つ

のノードのクラッシュ障害までを対称として耐故障性を保証するシステムは、「2-クラッシュ障害モデルに対処する」ということによって、対処する障害モデルと障害の数を明確に表すことができる。

2.1.3 耐故障性ソフトウェアを実現する技術

システムの耐故障性を高めるための技術は、ハードウェアによる方法とソフトウェアによる方法の 2 種類に分類することができる。本研究で議論の対象とするのは後者であり、本論文においてはソフトウェアフォールトトレランス (software fault-tolerance) と呼ぶ[1]。

ソフトウェアフォールトトレランスを実現するために既に提案されているいくつかの代表的な技術の中から、特に本研究と関連が深いチェックポインティング、リカバリブロック、複製の 3 つの技術についてその基本的な概念を紹介する。

チェックポインティング

計算途中において、ある時点での計算の状態や値を記憶装置に保存することを、チェックポインティング (*checkpointing*) を行うという。また、チェックポインティングを行う時期をチェックポイントと呼ぶ。チェックポインティングは、計算が長時間に及ぶようなアプリケーションを実行する際に利用される一般的な手段である。

計算中に障害が発生し、直前のチェックポイントの計算状態を取り出す動作を、ロールバック (*rollback*) という。また、ロールバックを行い、さらに障害発生直前のチェックポイントから実行を行って、以前障害が発生した計算状態まで戻ることを、リカバリ (*recovery*) と呼ぶ。

チェックポインティングを計算途中に行った後に障害が発生した場合、ロールバックを行ってリカバリを行うことによって、計算を最初からではなく、チェックポイントから再起動することができる (図 2.2)。チェックポインティングを行うことにはコストが必要であることは言うまでもない。しかし長時間の計算途中での

[1] "software fault tolerance" は、「ソフトウェアに内在する障害に対処する (software-fault tolerance)」という意味で用いられることもある [Lyu95] が、本論文では本文中に示したように "software fault-tolerance" の意味で用いる。

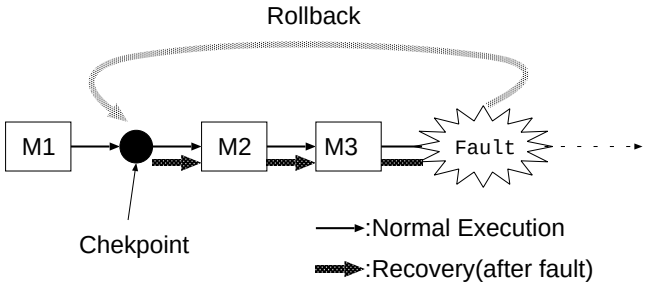

図 2.2: チェックポインティングとリカバリ

障害による再起動は、チェックポインティングを行わない場合は完全に最初から計算を行う必要があり、この計算のやり直しに伴うコストは非常に大きい。これに対してチェックポインティングを行う計算では、障害発生時にはその時点での直前のチェックポイントの状態から計算を再開することができる。

チェックポインティングを行う場合、計算機システムが利用者に与えられた計算を実行するのに要する実際の時間 T_t は以下のように表わすことができる。

$$
\begin{aligned}
T_t &= T_u + T_c + T_o \\
T_t &: \text{応答時間,} \\
T_u &: \text{ユーザに与えられた計算を実行する時間,} \\
T_c &: \text{チ ェックポインティングを行うのに要する時間,} \\
T_o &: \text{ロールバックとリカバリを行うのに要する時間}
\end{aligned}
$$

チェックポインティングを行わない場合の計算時間の期待値は、障害の発生を考慮にいれた場合、ユーザに要求される計算にかかる時間に対して指数的に増加するが、チェックポインティングを行う場合のそれは線形に増加することが知られている [VFN95]。このためチェックポインティングは、障害発生時の実行時間の増加を抑制するための基本的な手法の 1 つとして受け入れられている。

リカバリブロック

チェックポインティングを用いて実現される耐故障性を高める手法の一つとして、リカバリブロックがある。リカバリブロックの例を、図 2.3 に示す。

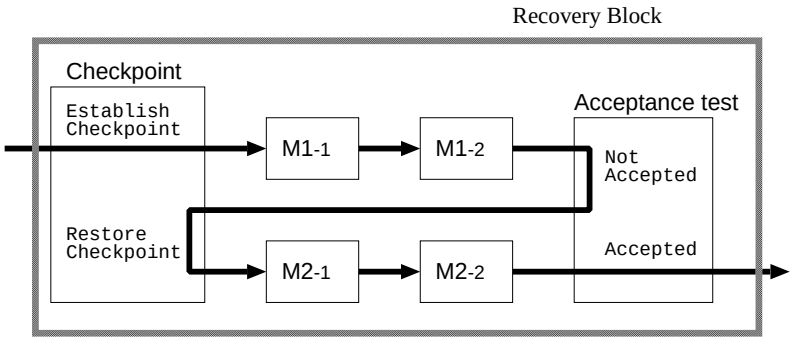

図 2.3: リカバリブロック

図 2.3 にある大きな太線の枠の内側が 1 つのリカバリブロックを示している。処理の流れがリカバリブロックの中に入ると同時に、チェックポインティングが行われる。リカバリブロックには、同じ意味の計算を行う実行単位が複数用意されている (図中では 2 つ、M1 と M2)。

チェックポインティングが終わると、まず最初に 1 つめの選択肢である M1 が実行される。M1 は、M1-1, M1-2 という 2 つの小さな計算単位の続きであり、M1 の完了、すなわち、M1-2 が完了した時点で M1-2 の状態や出力を用いてアクセプタンステスト (検定:$Acceptance\ \ Test$) が行われる。

図 2.3 では M1 からの出力がアクセプタンステストによって妥当ではないと判断されたため、このリカバリブロックに入ったときに保存したチェックポイントの値を取りだし (Restore)、別の選択肢の計算である M2 を実行する。M2 は M2-1, M2-2 からなり、M2-2 の最終状態がアクセプタンステストにかけられる。M2-2 の状態はアクセプタンステストによって妥当であると判断され、このリカバリブロック全体の計算が終了する。

リカバリブロックはネストさせることにより、柔軟に障害への対処を記述することができる。つまり、M1 や M2 をそれ以上は分割できない小さなモジュールと仮定すると、図 2.3 のリカバリ-ブロック全体 (RB-A とする) は、さらに大きなリカバリ-ブロックである RB-B の一部とすることができる。リカバリブロック RB-B のうちの計算の選択肢の 1 つである RB-A 全体がアクセプタンステストで妥当でないとされた場合は、RB-A 内の M1 や M2 の計算結果は全て破棄され、RB-B の他の選択肢である計算が実行される。

複製

最も古くから知られている耐故障性を高めるための手法の一つとして、複製 (*replication*) がある (図 2.4)。

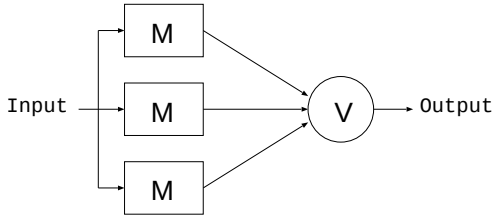

図 2.4: 複製と選択関数

複製は文字通り、ある機能を実行する 1 つのモジュール M の複製 (**レプリカ**：*replica*) を複数個作成し、これを同時に実行することにより複数の出力を得る。そして、選択関数 V に対して M で得られた複数の出力を入力すること (*voting*) により正しい出力を得る。特に、選択関数 V が多数決によって結果を決めている場合は、多数決 (*majority voting*) 関数と呼ばれることがある。

選択関数 V の動作は、仮定する障害モデルによって異なる。例えばクラッシュ障害のみを仮定する場合、レプリカから得られる出力は全て正しいことから、V へのもっとも早い入力が即時に出力となる。

一方 n-バリュー障害を仮定している場合、同一の出力が $n + 1$ 個得られた時点で V はその値を正しい結果として出力する。

2.2 複製技術

システムの耐故障性を高めるためにはシステム構成要素の冗長性を高めることは最も基本的な要件であり、本研究で使用する APR も複製技術の一つである。本節では代表的な複製技術として、ステートマシンアプローチとプライマリ・バックアップアプローチについて概要を述べ、最後に APR の概要を述べる。

2.2.1 プライマリ・バックアップアプローチ

プライマリ・バックアップアプローチ [AD76] はパッシブリプリケーションとも呼ばれ、ハードウェアを含めた商用の耐故障システムなどに最も広く用いられている。プライマリ・バックアップアプローチでは、入力は複数存在するレプリカのうちのプライマリレプリカに対してのみ行なわれる図 2.5。計算の実行は通常は

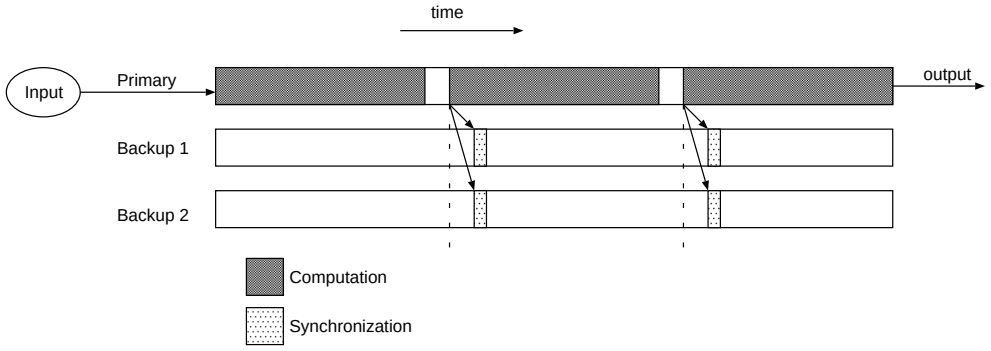

図 2.5: プライマリ・バックアップアプローチ

プライマリレプリカにおいてだけ実行され、出力もプライマリレプリカからのみ行なわれる。計算中のある時刻においてチェックポインティングを行なうことにより全てのレプリカ間で同期をとる。プライマリレプリカに障害が発生した場合には、バックアップのうちの 1 つがプライマリとなり、チェックポインティングで保存したデータを用いて実行を行なう。

プライマリ・バックアップアプローチはクラッシュ障害に対処することができる。またその実装の容易さから、多くの商用システムで用いられている反面、プライマリにおける障害のリカバリには必ず直前のチェックポイントまでのロールバックを必要とするという特徴がある。

2.2.2 ステートマシンアプローチ

ステートマシンアプローチ (*State Machine Approach*)[F.S90] は Active Replication とも呼ばれる複製技術の一つである (図 2.6)。入力は全てのレプリカ (ステートマシン) に対して行なわれる。全てのレプリカは計算を実行し、実行中のチェックポイントにおいて全てのレプリカ間で同期を取ることによって実行が進められる。

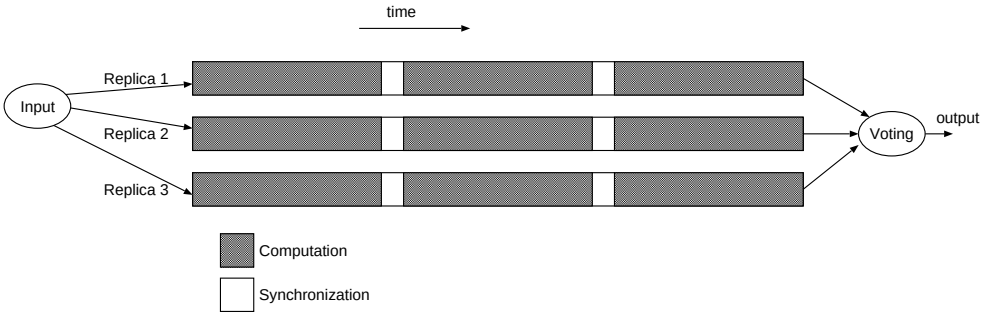

図 2.6: ステートマシンアプローチ

ステートマシンアプローチはクラッシュ障害から Byzantine 障害を含む任意の障害に透過的に対処できるが、複数のステートマシンの一貫性を保つためにレプリカ間の同期をとることが必須である。またアプリケーションの定義する計算に対して、レプリカ数に比例する計算資源を消費するというのも特徴である。

2.2.3 APR 複製技術

APR (*Active Parallel Replication*) は複製による耐故障性の保証に加え、並列計算によって計算時間の短縮を実現する新しい複製技術であり、Cherif らによって 1998 年に提案された [CK98]。APR は主に長時間にわたる大規模な計算アプリケーションを対象としており、単一のフレームワークでクラッシュ障害とバリュー障害の 2 つの障害モデルに対処する。APR では全てのレプリカに対して入力を行い、

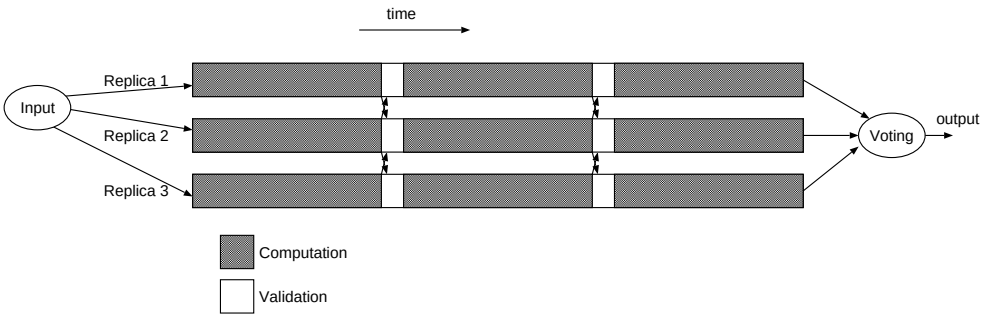

図 2.7: APR 複製技術

全てのレプリカが計算を実行する。これによりクラッシュ障害およびバリュー障害に対処する。

さらに計算の実行は、アプリケーションを複数の独立な関数として記述し、レプリカ毎に異なる順序で関数計算を起動することによって行なわれる。また計算結果をレプリカ間で共有することにより、並列計算による計算の高速化を実現している。APR 複製技術に関する詳説は第 3 章で行なう。

2.3 グループコミュニケーション

ネットワークによって接続された複数の計算機による活動は、個々の計算機内部の計算の実行及び計算機間の通信による情報の送受信の 2 種類に大別できる。本章では後者に関する技術であるグループコミュニケーションについて、既存の通信技術を紹介する。

分散環境における通信

分散システムにおいて複数プロセスの協調動作を実現する場合、設計者および実装者は、システムを構成する全ての要素に関して、実装した個々のシナリオが仕様を満たすかに関して多くの分析を行なわなければならない。例えば疎結合分散環境において複数のレプリカによる計算を実現する場合は、システム内の全ての複製において一貫性を保ちながら、複製されたある一つの値に対する更新操作が行われなければならない場合などがある。

現実の分散システムにおいては、システムを構成する多数の要素がネットワーク上に分散して存在するため、システム内の全ての構成要素における時刻の完全に正確な同期を行うことは不可能である。このことから、全ての複製上で特定の時刻に操作を起動することによって一貫性を保証するという方法は用いることができない。この問題に対する解決として、同期に必要な時刻をシステム内の論理的な時刻、すなわち構成要素間を流れるメッセージの送受信の順序関係として扱う方法がいくつか提案されている [BM93][HT93]。

メッセージ送受信の順序という論理的な時刻を扱うことにより、分散システム内部の全てのプロセスにおいて一貫性を保持しながら実行を行うことが可能になる。しかしシステム内部に障害の発生を仮定する場合、非同期システムにおいて

はコンセンサスの形成やアトミックマルチキャストを実現することは不可能であることが証明されている [FLP85]。このことから現実の非同期システムにおいては、個々のアプリケーションに応じたタイムアウトを用いることによってシステムの障害を検出することが多い[2]。

プロセスグループ

プログラマが全ての通信に関して論理時計に注意を払いながらシステムを構成することは非常に手間がかかるため、通信を多く用いるシステムにおいてはバグの混入などの原因となる。この問題に対して、複数のプロセス[3]をグループとして扱う方法が、オペレーティングシステムや分散システムを設計、実装する際の重要な抽象として 1980 年代から提案されている [CZ85][Bir86]。複数プロセスの抽象とそれらの通信手段を提供するシステムを、グループコミュニケーションシステムと呼ぶ。とくにプロセスのクラッシュや通信リンクの障害への対処を考慮したグループコミュニケーションシステムとして、ISIS[Bir93]、Totem[AMSM92]、Horus[vRBM96]、Transis[DM96]、Ensemble[Hay98] などが提案されている。

グループに含まれる要素、つまり通信を行うプロセスをメンバと呼ぶ。グループコミュニケーションシステムが提供する主なサービスには、**メンバーシップサービス**と**グループマルチキャストサービス**[4]がある。

メンバーシップサービス

メンバーシップサービスは、分散環境上に存在するプロセスをメンバーと見なし、それら複数のメンバーからなるグループを構成する。これによりプログラマおよびユーザは、個々のメンバーに対してではなく、対象とするグループ全体の特徴を考慮することによって、様々な管理や操作を行うことができる。具体的には基本的な操作として、各メンバーがグループに対する参加や脱退を行う操作 (*join*,

[2] このタイムアウトは論理的には、非同期分散システムにおいて障害検出機構が誤って有限時間で障害を検出するというアイデアにより実現されるものであり [CT96]、有限時間内にメッセージが必ず到着することを仮定する同期システムとは根本的に異なる。

[3] ここでは、分散環境に存在して通信及び計算を行う実体を総称して「プロセス」と呼ぶ。

[4] グループコミュニケーションの分野においては、グループ全体にブロードキャストを行なうことを一般にマルチキャストと呼ぶことから、本文でもこの慣習に従う。

leave) を提供する。グループに参加している全てのメンバーは、メンバーシップに関する共通の**ビュー**、すなわちグループ名や現在の全てのメンバーに関する情報を内部に保持している。

メンバーシップの管理は通常**ハートビート** (自プロセスが動作していることをグループに知らせる定期的なメッセージ) を用いて行われる。ハートビートは送信するメンバーが正常に動作していることをグループに知らせるものである。グループ内では常に、ビューの変更の際に一時的に特別な役割を果たす**コーディネータプロセス**が 1 つだけ存在する。あるメンバーからのハートビートが他のメンバーによって一定時間以上観測されない場合、観測を行えなかったメンバーはグループコーディネータに対して *suspect* メッセージを送信する。グループコーディネータは *suspect* メッセージを受信したことを全てのメンバーに送信する。この時点で全てのメンバーはグループへのメッセージの送信を一時停止する。コーディネータによる送信に対するリプライが一定時間以上起こらない場合、他の全てのメンバーが内部に持っているビューから *suspect* されたメンバーが削除され、新たなグループビューがコーディネータから配布される。これによってビューは更新され、新たなビューに含まれる全てのメンバーにおいてビューが更新された後に通常の通信が開始される。

グループマルチキャストサービス

複数のメンバーがグループに対して連続してブロードキャストを行う場合、その単一性や順序関係についてしばしば問題になる事は既に述べた。これに対してグループコミュニケーションシステムは、プログラマが要求する特定の通信の性質、すなわち順序関係や単一性に関していくつかの通信プリミティブを提供する。

たとえば Ensemble グループコミュニケーションシステムでは、メッセージの到着順序に関してグループ毎に性質を指定することができる。具体的には FIFO, causal, total 各々の順序関係をグループの属性として定義することによって、必要に応じて保証することができる。

いくつかのグループコミュニケーションシステムは、広域ネットワークを含む商用の分散システムにおいて、データの共有、分散トランザクションの実現、データの複製の管理のために使用されている。

本研究では上記に述べたメンバーシップサービスとグループマルチキャストサービスを、レプリカの管理とレプリカ間のデータの送受信に用いる。

第 3章

APR複製技術

APR は単一のフレームワークの中で、クラッシュおよびバリューの 2 つの障害への対処を実現する複製技術である。計算の並列化と保証する耐故障性への対処は全て実行時システムの内部において実現され、ユーザおよびプログラマはその実現のために特別な設計や実装を行なう必要がない。

APR 複製技術は、その計算の基本的な方法である *APC*(Active Parallel Computation) と障害に対処するいくつかの手法から定義される。本節ではまずはじめに APC の基盤となっている FTAG 計算モデルを紹介する。

3.1 FTAG 計算モデル

ここではまずはじめに、APR が基としている計算モデルである FTAG 計算モデルについて、APR を理解するために必要な基本概念を説明する。

3.1.1 FTAG 基本モデル

FTAG 計算モデル [SKS94] では、全ての計算を**モジュール**と呼ぶ純粋に数学的な関数の集まりとして記述する。各モジュールは複数の入力と出力を持つことができる。入力 $x_1, \ldots, x_n$ および出力 $y_1, \ldots, y_m$ を持つモジュール M を以下のように記述する。

$$M(x_1, \ldots, x_n \mid y_1, \ldots, y_m)$$

$x_1, \ldots, x_n,\ y_1, \ldots, y_m$ をモジュール M の**属性**と呼ぶ。この場合 $x_1, \ldots, x_n$ を**相続属性** (または入力属性)、$y_1, \ldots, y_m$ を**合成属性** (または出力属性) と呼ぶ。

モジュール M が十分に単純な場合は、出力は相続属性から直接計算される。このようなモジュールを**基本モジュール**と呼び、以下のように記述する。

$$M(x_1, \ldots, x_n \mid y_1, \ldots, y_m) \Rightarrow return \quad \textbf{where} \quad \mathbf{E}$$

ここで E は $y_1, \ldots, y_m$ が $x_1, \ldots, x_n$ からどのようにして計算されるかを表す式の並びで、**属性関係式**と呼ばれる。M_j のある相続属性 x が M_i のある合成属性 y を単にコピーされるだけの場合、x の代わりに y を M_j の入力として記述することで $y = x$ という関係式を省略することができる。

M が複雑な場合はより単純な複数の**サブモジュール**に分解される。M が $M_1, .., M_k$ に分解されることを以下のように記述する。

$$M(x_1, \ldots, x_n \mid y_1, \ldots, y_m) \Rightarrow M_1 \ldots M_k \quad \textbf{where} \quad \mathbf{E}$$

$M_1 \ldots M_k$ と E の組を M の**分解**と呼び、以後 D で表現する。

モジュールの分解には条件によって複数の方法がある場合が考えられる。**条件付き分解**は以下のような一般形で記述される。

$$\begin{array}{l} M(x_1, \ldots, x_n | y_1, \ldots, y_m) \Rightarrow \\ \quad [\quad C_1 \qquad\qquad \rightarrow D_1 \\ \quad | \quad \vdots \\ \quad | \quad C_n \qquad\qquad \rightarrow D_n \\ \quad | \quad \textbf{otherwise} \quad \rightarrow D_{def} \quad] \end{array}$$

条件 $C_1, \ldots, C_n$ はこの順でテストされ、C_i が真になった時点で対応する分解 D_i が適用される。どの条件も満たされない場合はデフォルト分解 D_{def} が適用される。

FTAG におけるプログラムは、全てが基本モジュールで記述できるようになるまでモジュール分解を繰返し適用することによって実行される。

計算の過程や結果は**計算木**と呼ばれる図 3.1 に示すような属性付きの木構造によって表現される。相続属性はモジュールに向かって計算木を上から下へ流れ、合

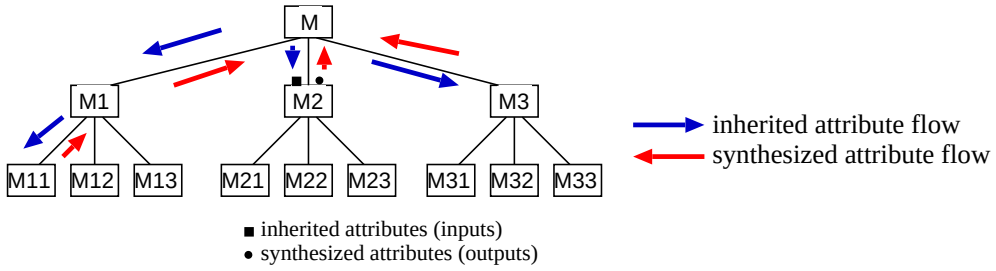

図 3.1: FTAG 計算木

成属性はモジュールから出て下から上へと流れる。

3.1.2 再実行

FTAG は計算木の一部を別の計算木で置換する**再実行**という機構を持っている。これは障害が発生した (正しくない結果が得られた、または結果が得られなかった) 部分の計算を実行しなおすために利用される。

ここでは、我々の興味の対象となっている障害は全て値の誤りに反映されることで検出可能であるという仮定を設けている。この仮定はソフトウェア障害を取扱うときに一般的なものである [Ran75]。一方プロセッサの故障などの障害は適当な属性の値が $\perp$ となることで検出できるものとする。

図 3.2 は再実行のもっとも単純な場合を表している。モジュール M は M_1, M_2, M_3

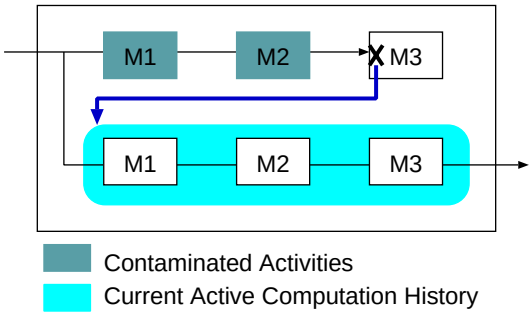

図 3.2: FTAG の再実行

に分解されるが、M_3 の実行中のある時点で値に誤りがあることが検出され、原因を分析したところその原因が M_1 の実行中にあることがわかったと仮定する。このとき、M_1 以降のすべての計算結果は破棄され、M_1 およびそれに続く M_2, M_3 は再計算される必要がある[1]。このような再計算をモジュール分解の特殊な形と考え、**再実行分解**と呼ぶ。再計算が正しく実行された後では、再計算された部分が現在の実行履歴として保存される。

図 3.3 は再実行によって実行木がどのように変化するかを表している。再実行が

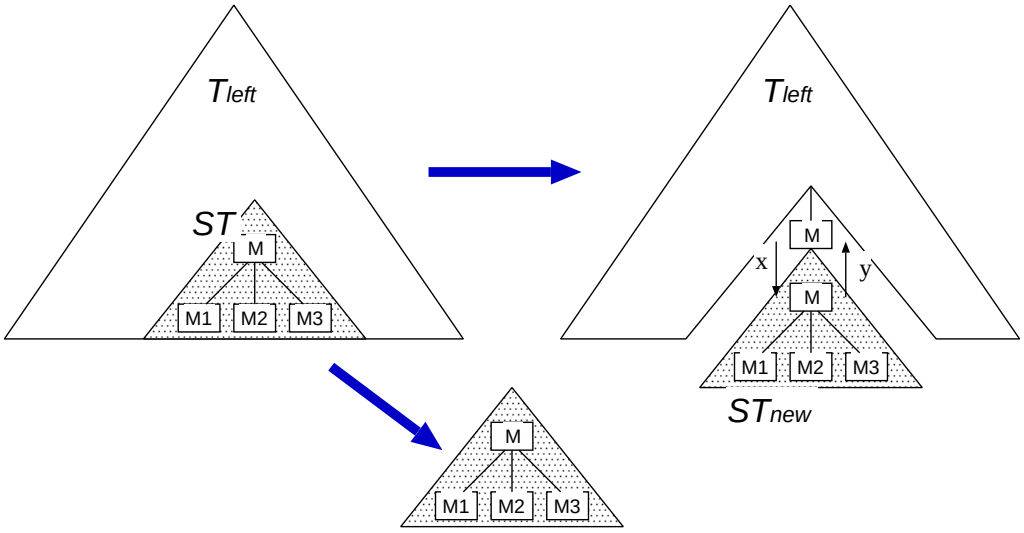

図 3.3: 再実行発生時の実行木の変化

発生すると、まず誤った値を含んだ部分木 ST を実行木から切り放す。残った部分を T_{left} と呼ぶことにする。次にモジュール M を根とする新しい実行木 ST_{new} を作製し、T_{left} における M の全ての入力を ST_{new} に渡す。ST_{new} は M の再実行が進行するにつれて成長する。実行が完了すると、ST_{new} は ST と置換されるような形で T_{left} に継ぎ木され、ST_{new} の M の出力が正しい計算結果として T_{left} に渡される。

一般に計算木には複数の M に相当するノードが存在するが、どのノード (および部分木) が再実行されるかは実行木を解析することによって決定される。この例では再実行の対象はもっとも最近に実行された M であり、これは実行木上で M_3 からルートに到るパス上で最初に出現する M として定義される。

[1] この単純な例では障害は一過性、すなわち再計算の時には発生しないことを仮定している。

3.1.3 複製

障害時にもサービスを継続して行なうための標準的な手法として**複製** (*replication*) がある。ハードウェア故障に対応するためには、同一のプログラムを異なるマシン上で並列に実行する方法が考えられる。この手法を形式化したものとしては**複製状態機械モデル** (*replicated state machine*)[F.S90] が有名である。一方ソフトウェアの故障に対応するためには、複数の異なるプログラムを用意して同時に実行する。N**バージョンプログラミング**はこの手法の実現方法のひとつである [Avi85]。

FTAG ではモジュール分解において同一の相続および合成属性の集合をもつ複数のサブモジュールが存在するとき、それらは複製であると解釈される。例えば、

$$M(x \mid y) \Rightarrow M_1(x \mid y)\ M_1(x \mid y)\ M_1(x \mid y)$$

では 3 個の M_1 はそれぞれが同一の相続属性 x および合成属性 y を持つ。このような形態の分解を**複製分解** (*replicated decomposition*) と呼び、分解された M_1 の集合を**レプリカ** (*replica*) と呼ぶ。複製分解の解釈は以下のようになる。各モジュールは通常の分解と同様同時に実行されるが、どれかひとつだけが M の正しい結果となる。当然ながら、耐ハードウェア故障のためには各 M_1 はすべて別のプロセッサ上で実行されなければならない。

3.2 Active Parallel Computation

APC (*Active Parallel Computation*) は、関数型計算モデル FTAG に対して複製に関する新たな機構を導入した計算手法である。APR では、計算起動前に必ずルートモジュール (計算木の根となるモジュール) の複製を行なってから計算を開始し、全てのレプリカにおいて APC で定めるアルゴリズムによって計算を実行する。

APC での計算は、レプリカ毎に計算起動順序に関して異なるポリシーを持つことによって行なわれる。例えば、以下に示すような FTAG アプリケーションプログラムを考える。

$$
\begin{array}{lcllcl}
M(x|y) & \Rightarrow & M_1(x_1|y_1) & M_2(x|y) & \Rightarrow & M_{21}(x_{21}|y_{21}) \\
 & & M_2(x_2|y_2) & & & M_{22}(x_{22}|y_{22}) \\
 & & M_3(x_3|y_3) & & & M_{23}(x_{23}|y_{23}) \\
 & & \textbf{where} & & & \textbf{where} \\
 & & \vdots & & & \vdots \\
M_1(x|y) & \Rightarrow & M_{11}(x_{11}|y_{11}) & M_3(x|y) & \Rightarrow & M_{31}(x_{31}|y_{31}) \\
 & & M_{12}(x_{12}|y_{12}) & & & M_{32}(x_{32}|y_{32}) \\
 & & M_{13}(x_{13}|y_{13}) & & & M_{33}(x_{33}|y_{33}) \\
 & & \textbf{where} & & & \textbf{where} \\
 & & \vdots & & & \vdots
\end{array}
$$

ルートモジュール M は、3 つのサブモジュール M_1, M_2, M_3 に分解されて計算が行なわれる。上記の FTAG プログラムコードを、APC によって 3 つのレプリカによって実行している様子を図 3.4 に示す。

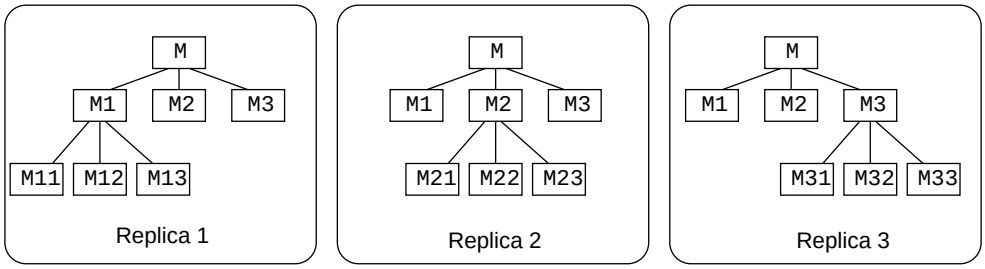

図 3.4: APC (Active Parallel Computation)

図 3.4 は計算実行途中の計算木である。レプリカ 1 では最も左側の計算木から、レプリカ 2 では中央の計算木から、そしてレプリカ 3 では最も右側の計算木から計算を実行する。

計算がレプリカ毎にどのように進行しているのかを明らかにするために、図 3.5 に *Gannt chart* を用いた計算進行状況のシミュレーションを示す。チャート中では縦にプロセッサおよびレプリカ、横に時間の軸が各々とられている。チャート中の各々のマスでモジュールの名前が記されているものは、そのモジュールが計算中であることを表す。黒く塗り潰された部分はプロセッサがアイドルであることを示している。

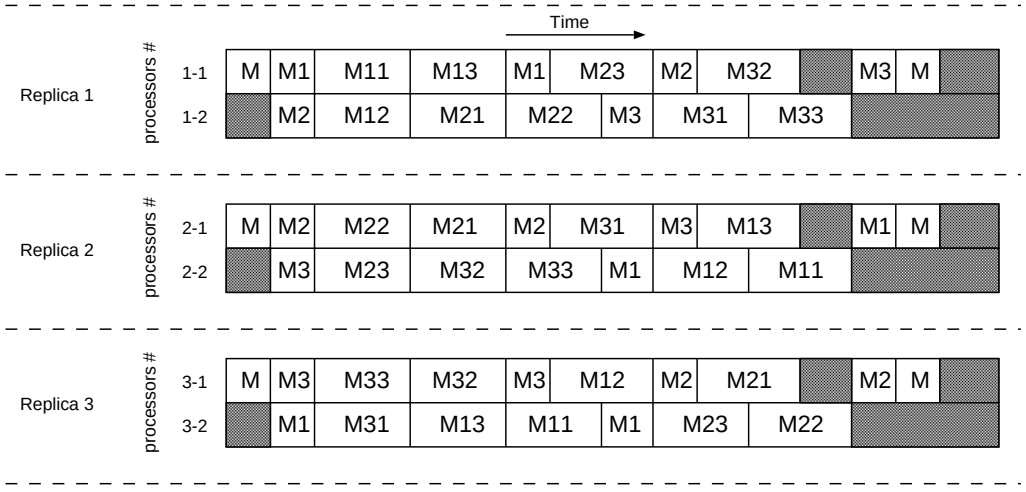

図 3.5: Gantt Chart of APC

ここでは全てのモジュールの計算時間とプロセッサの処理速度を一定と仮定している。全てのレプリカは計算順序に関して異なるポリシーで計算を行なう。レプリカ 1 では M_1, M_2, M_3 の順で計算を行う。レプリカ 1 では M_1 の計算において関数分解を行うため、M_2 の計算を行う前に M_{11}, M_{12}, M_{13} の計算を先に実行する。同様にレプリカ 2 においては M_2 の部分木から、レプリカ 3 においては M_3 の部分木からそれぞれ実行を開始する。

全てのレプリカは全ての部分木の計算が完了するとルートモジュール M の出力を返すことにより計算を完了する。

3.3 APR

APC では FTAG 計算モデルを基にして、各々のレプリカにおけるモジュールの計算起動アルゴリズムを決定した。APR ではこれに加えて、障害への対処を考慮した計算起動順序を定義している。具体的には、クラッシュ・バリューのどちらの障害モデルを仮定するか、またその障害の数としていくつを仮定するかによって、実行時システムが各々の場合の計算順序と計算結果の扱いに関する制御を行なう。

3.3.1 クラッシュ障害モデルの場合

クラッシュ障害を仮定する場合、各々のモジュールの出力は全てのレプリカの中のいずれかから、1 つだけ得られれば良い。なぜならクラッシュ障害の仮定より、出力結果は必ず正しいからである。

APR では、全てのレプリカは基本的に APC で定められた計算順序にしたがって計算を起動する。ただしクラッシュ障害を仮定する場合、既に関数分解結果もしくは出力が得られているモジュールに関する計算の起動は行なわない。

全てのレプリカはモジュールの分解および計算が完了すると、自分のレプリカにおける関数分解結果およびモジュールの出力結果を、正しい計算結果として安定記憶に保存する。保存と同時に、これらの計算結果は他のレプリカに送信される。他のレプリカから出力を受け取ったレプリカは、この結果を正しい値として安定記憶に保存する。もしも受信した結果に関するモジュールを計算中の場合は、計算を中断して計算資源を解放する。

クラッシュ障害を仮定して図 3.4 の計算木で示した計算を行ない、実際には障害が発生しなかった場合の Gannt Chart を図 3.6 に示す。各々のレプリカは APC に

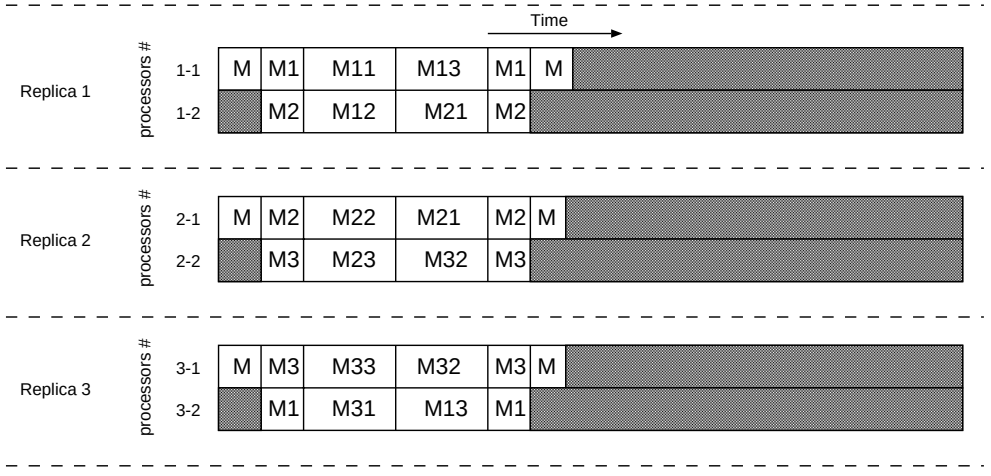

図 3.6: APR の実行 (クラッシュ障害モデルを仮定した場合)

定義されている計算順序にしたがって計算を行なっている。さらに計算を行なう過程で他のレプリカからの出力を受信する。例えばレプリカ 1 では、$M21$ を計算完了した段階において、既にレプリカ 2 から受信した $M22, M23$ の計算結果を用いて $M2$ の計算を起動している。

図 3.6 の例では深さが 3 という小さな木構造を仮定しているために 2 つのレプリカによって計算されている部分 (M_{21}, M_{13} など) の割合が高く見える。しかし一般に APR では、クラッシュ障害を仮定して障害が発生しない場合、計算木中のほとんどのモジュールに関して 1 回のみ計算を行う。このように APR でクラッシュ障害を仮定したときには、計算完了までに必要な時間は大幅に短縮される。

3.3.2 バリュー障害モデルの場合

APR ではバリュー障害モデルを仮定した場合もクラッシュ障害の場合と同様に、APC による計算の起動を行なうが、関数分解結果および計算結果に関する扱い、そして関数の起動決定がより複雑になる。

バリュー障害を仮定した場合は、モジュールからの結果を 1 つだけ獲得した時点では、その値が正しいとすることはできない。つまり出力結果や分解結果は複数のレプリカからの出力によってその妥当性の確認を行なわなければならない。n-バリュー障害を仮定する場合は、その仮定から、$(n+1)$ 個の同一の結果が得られたときにその計算結果は妥当であると言うことができる。

このため全ての関数分解結果および出力結果には妥当性の確認が完了しているかしていないかを示す属性が付加される。計算の起動は APC の計算順序およびこの妥当性を示す属性によって決定される。

1-バリュー障害を仮定して図 3.4 に示した計算木の計算を APR で行ない、実際には障害が発生しなかった場合の Gannt chart を図 3.7 に示す。図 3.7 において、レプリカ 1 は M_{32}, M_{33} の計算を行なっていない。これは、M_{32}, M_{33} の結果が既にレプリカ 2 とレプリカ 3 によって計算され、$(n+1) = 2$ 個の同一の計算結果が得られた、つまり妥当性を確認されている計算結果をレプリカ 1 が受信し、既に持っているからである。

このように、バリュー障害を仮定している場合においては計算機中のそれぞれのモジュールはいずれかのレプリカによって合計で 2 回だけ評価されるため、APR によって計算完了までの時間が短縮される。

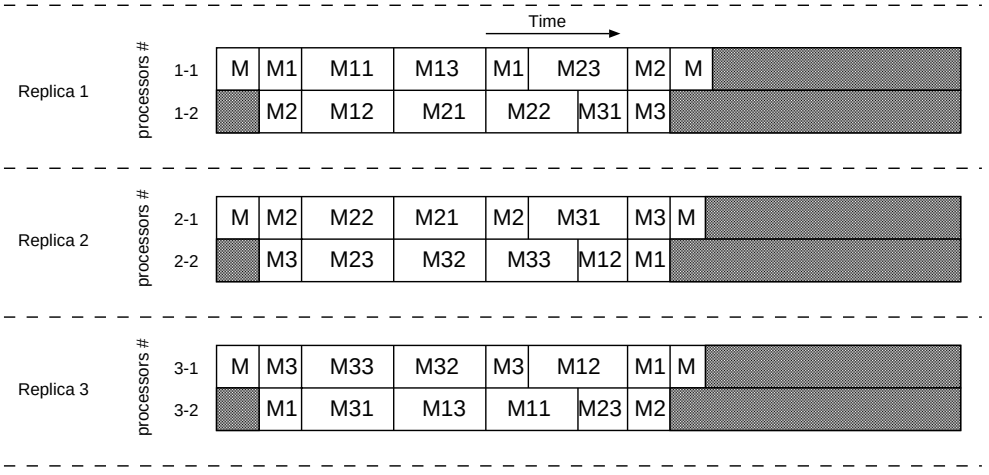

図 3.7: APR の実行 (バリュー障害モデルを仮定した場合)

3.4 ACMS

APR においてバリュー障害を仮定する場合、計算木の深さが深い場合は障害発生時のリカバリに必要な時間が大きくなる可能性がある。

例えばあるレプリカにおいて、計算木を分解している途中で障害が発生した場合を考える。この障害を検出するためには他のレプリカからの関数分解結果または出力属性を得ることによって計算結果を比較し、バリュー障害を検出する必要がある。

ここでアプリケーションによって与えられる計算木の深さが大きい場合を考える。(ある時刻 t_i では検出されていないが) 障害を持つ出力を行ったレプリカ R_d は、通常の APR の計算起動順序にしたがって計算を続行する。また R_d 以外の他のレプリカも時刻 t_i 付近において、APR によって定められた順序通り、障害とは親子関係にない部分木の計算をリーフモジュール (FTAG の基本モジュール) まで計算する。R_d 以外のいずれかのレプリカがその後の時刻 $t_n(t_n > t_i)$ において、ようやく他の部分木の計算を開始し、このレプリカがいつか (時刻 $t_d(t_d > t_i)$) 必ず[2] R_d の出力との比較を行い障害が検出される。

このように、障害が起きたレプリカが計算した部分木と同じ部分木の計算を他のレプリカが実行するまでの時間が長くなり、障害検出までの時間は非常に長く

[2] n-バリュー障害の仮定と、その仮定に対処するために APR が同一モジュールを最低で $n+1$ 個のレプリカにおいて計算するというアルゴリズムより、明らか。

なる可能性がある。この結果として、レプリカ R_d が障害を持つ関数分解結果を用いて計算を行なった部分木、すなわちリカバリによって破棄される部分木が大きくなるという問題が生じる。

この問題に対処するために $ACMS\,(Adaptive\ Computation\ Management\ Scheme)$ アルゴリズムが提案されている。ACMS では一定時間毎に各々のレプリカにおける計算順序に関するポリシーを変更し、同一モジュールの関数分解結果および出力を複数のレプリカにおいて獲得するまでの時間を短縮する。これによりバリュー障害を仮定する場合のリカバリに要する最悪実行時間を制限することができる。

第 4章

APRタスクの分析

APR の基本的な計算方法と障害への対処方法はすでに定義され、疎結合分散環境への実装が適しているということが指摘されている [Che98] が、具体的なアルゴリズムや実装の詳細に関する考察は現在のところ行なわれていない。本章では、APR 複製技術の主要な構成要素である APR 計算起動アルゴリズムの詳細について実装環境を考慮にいれた分析を行い、実装を行なうために必要ないくつかの定式化を行なう。

4.1 資源消費の分析

APC と APR によるモジュールの起動および障害への対処の方法に関する概要は第 2 章で述べた。本節では APR の論理的な側面を明確にするため、APR のスレッド[1]の状態とそれらの消費する資源に関する分析を行なう。

4.1.1 計算スレッドの状態に関する分析

APR アルゴリズムに従う計算では、関数の定義とその関数に対する入力が決定された後に、APR スケジューラ (APR アルゴリズムを実装している実行時システム) からの起動要求が起こった時点で計算の起動が行われる。さらに実行中のスレッドが関数分解を起こしてサブモジュールの計算を行なうスレッドを起動する

[1]本章では一つの処理の論理的な流れを一般に「スレッド」と呼ぶ。

場合、それらサブモジュールもさらに別のスレッドとして実行されるため、呼び出し側のスレッドは実行を一時停止してサブモジュールからの出力属性を待つ。

今後ことわりなく「起動」と記述する場合は、関数の計算を行なうスレッドの起動を表すものとする。また関数の定義と入力の決定が行われているが実行に使用する資源を割り当てられていないモジュールを「起動可能 (Ready)」と言う。図 4.1 にスレッドの起動から終了までの全ての状態を示す。

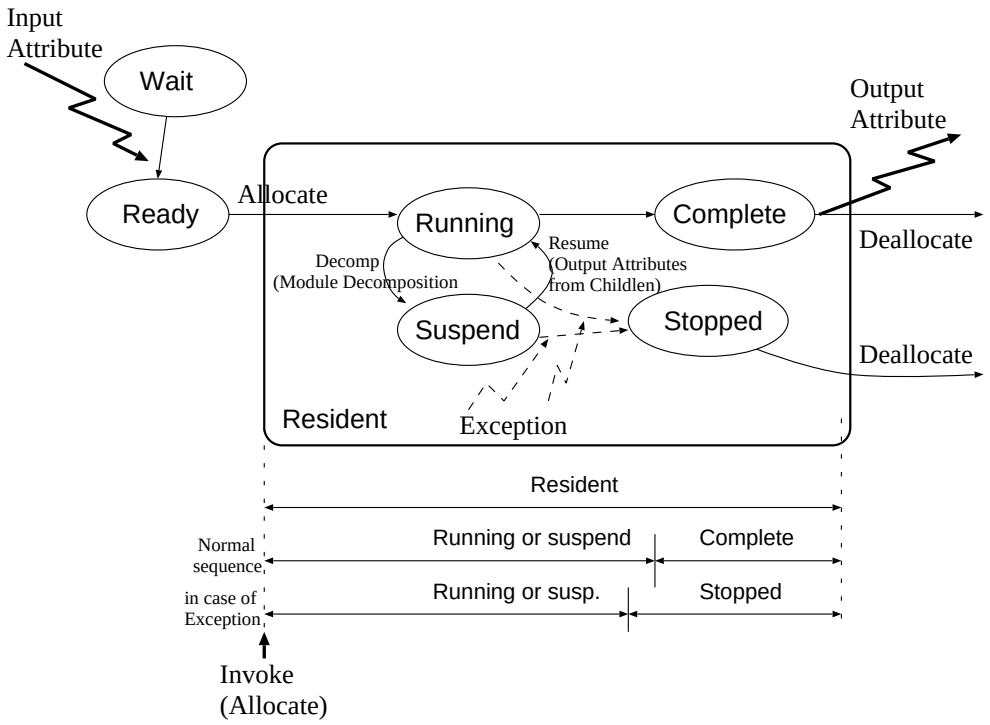

図 4.1: 起動されたスレッドの状態遷移

計算木のルートモジュールはユーザからの計算開始要求により、それ以外のモジュールは親モジュールを実行するスレッドの関数分解によって生成される。入力属性が全て決定している場合は起動可能 (Ready) になり、APR のスケジューラによるスケジューリングが行われる。入力属性の全てもしくは一部が決定していない場合、Wait 状態で入力属性を待つ。APR のスケジューラによって起動が決定されたモジュールは、資源を割り当てられる $(allocate)$ と一つのスレッドとして起動され、Resident 状態に遷移する。起動直後のスレッドは直ちに Running 状態に入り関数の計算を開始する。Running 状態で関数分解を起こしたスレッドは Suspend 状態でサブモジュールからの出力属性を待つ。サブモジュールからの出力属性を

得た関数は再起動 ($resume$) し、この属性を用いて再び計算を行う。計算が完了すると Complete 状態において自らの出力属性を親モジュールに渡した後にスレッドを終了して Resident 状態から出ると同時に実行を終了する。

実行時システムによる障害の検出に起因する再実行の発生や、資源削除要求が発生した場合は実行時システムから exception が発生する。この場合スレッドは計算の実行を中断して Stopped 状態に遷移する。

計算木の中間に存在するモジュールでは必ずサブモジュールからの出力属性待ちとその属性を用いた計算が行われる。一方リーフモジュール (サブモジュールを持たない計算木の末端のモジュール) に関してはサブモジュールからの出力属性待ちは無いため、Suspend 状態はとらない。exception が起こらない場合の Resident な計算スレッドの状態遷移のシナリオには、以下の二種類のみが存在する。

- リーフモジュール　　Running→Complete
- 中間モジュール　　Running→Suspend→Running→Complete

4.1.2　各状態における消費資源に関する分析

ある関数を実装したスレッドが使用する資源の量を、実行前の入力属性が決定していない段階で正確に見積もることは不可能である。しかし、資源割り当てアルゴリズムを決定するためには、実行中のスレッドがいつどのように資源を消費するのか、その資源の種類と消費の傾向を明らかにする必要がある。このため本節では、実行中のスレッドの各状態における消費資源に関して、プロセッサ実行時間と記憶領域の使用量に関する分析を行う。

FTAG 言語によって記述されたアプリケーションでは、関数分解が実行される時点では既に当該モジュールの全てのサブモジュールの入力属性は定義されている。このため関数分解は常に、APR スケジューラに対するスレッド起動要求を直ちに発生する。これら起動要求を行われたモジュールは、APR のスケジューラによってスケジューリングされた後に、なんらかの資源割り当てアルゴリズムによって決定する PE に上で起動される[2]。資源の割り当て (allocate) および起動されたサ

[2]資源割り当てアルゴリズムについては 5 章で具体的に述べる。

ブモジュールの計算を行うスレッドは、Running 状態に入り計算を開始する。この Running 状態においてスレッド th により消費される資源 $R_{ri(th)}$ は、CPU 時間と記憶領域であり、具体的には以下のように表すことができる[3]。

$$R_{ri(th)} = \{\ C_{ri(th)}\ ,\ \int_{Alloc}^{Susp} (Mt_{th} + Md_{th}(t))dt\ \} \tag{4.1}$$

ここで、

$C_{ri(th)}$： スレッド th が消費する CPU 時間

Mt： 静的記憶領域

Md： 動的に確保する記憶領域

$Alloc$： allocate される時刻

$Susp$： Suspend 状態に遷移する時刻

次に計算実行中のスレッドにおいて関数分解が発生してサブモジュールの関数を呼び出し、このサブモジュールの評価を行うスレッドからの出力を待っている状態が、Suspend 状態である。スレッド th が Suspend 状態において消費する資源 $R_{susp(th)}$ は、

$$R_{susp(th)} = \int_{Susp}^{Run} (Mt_{th} + Md_{th})dt \tag{4.2}$$

である。R_{susp} は、計算木上のリーフモジュール以外の全てのモジュールがサブモジュールからの出力を待つ間 Suspend 状態にあり、記憶領域を消費する一方で CPU 時間を消費していない状態を表している。

関数分解を起こして Suspend 状態にあったモジュールは、サブモジュールからの出力を用いて**再起動** $(resume)$ して Running 状態に遷移を行い、関数の合成属性の計算を行う。このときにスレッド th が消費する資源 $R_{rs(th)}$ は式 4.1 と同様に、記憶領域と CPU 時間の消費を行う。

$$R_{rs(th)} = \{\ C_{rs(th)},\ \int_{Run}^{Compl} (Mt_{th} + Md_{th}(t))dt\ \} \tag{4.3}$$

つまり論理的には、相続属性を用いた属性関係式の評価に必要な資源が R_{ri} であり、この評価によってサブモジュールへの入力が決定する。一方、サブモジュー

[3]ここで R の添字 ri は、相続属性 (inherited attributes) を用いた実行中 (running) を示し、後述の rs の s は合成属性 (synthesized attributes) を示す。

ルの合成属性を用いた属性関係式の評価に必要な資源が R_{rs} であり、この評価によって当該関数の合成属性 (出力) が得られる。今後、R_{ri} と R_{rs} の双方をあわせて R_{Run} と呼ぶ。

あるスレッド th の計算実行によって資源される R_{th} は以下のように表すことができる。

$$R_{th} \quad = \quad \{ \int_{Alloc}^{Compl} (Mt_{th} + Md_{th}(t))dt,\ C_{th} \} \tag{4.4}$$

ただし $C_{th} = C_{ri(th)} + C_{rs(th)}$

C_{th} は計算する関数と入力属性、そして PE の単位 CPU 時間あたりの計算能力によって決定される。一方記憶領域の消費はアプリケーションと入力属性によって決定する絶対量と、Suspend 状態も含む Allocate から Deallocate までの時間に関係する。

4.1.3 通信に関する分析

APR のターゲットであるアプリケーションは、長時間にわたる大規模な科学技術計算などの計算である。計算には多くの PE が使用され、レプリカを作成する場所として広域のネットワークを含むこともある。しかしながら疎結合分散環境においてマルチプロセッサを用いて並列に計算を行う際には、計算を行う構成要素の増加に伴ってそれらの通信量や通信コストは増加するという問題がある。このため一般にプロセッサ数に対して理想的に、比例して性能が向上することはない。

このため本節では主に、APR を実行するために必要な通信についてまず分析を行った結果について述べる。また通信方法やソフトウェアの構成方法によって影響される、スケーラビリティに関する分析を行う。本研究では対象アプリケーションとして大規模で粗粒度の計算アプリケーションを対象にしているため、主にスケーラビリティに着目して評価を行なう。これらにより第 4.2 節以降では、アルゴリズムや通信方法の明確な定義およびソフトウェアの構成方法を決定する。

通信の分類

APR では図 4.2 に示すように、大別して二種類の通信が発生する。第一の通信

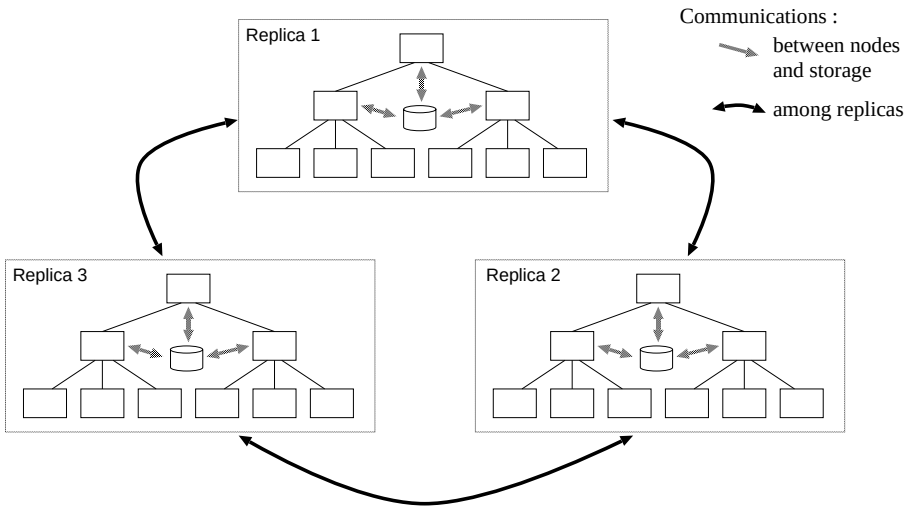

図 4.2: APR における 2 種類の通信

は個々のレプリカ内部で起こる通信であり、これを**レプリカ内通信**と呼ぶ。APR ではクラッシュ障害およびバリュー障害に対処するため、全ての計算結果と分解結果は、レプリカ内の安定記憶に保存する。このため以下に示す各段階においてレプリカ内通信が行われる。

- 計算の開始 (親モジュールからサブモジュールへの入力属性)
- 計算完了 (サブモジュールから親モジュールへの出力属性)

第二の通信は実行中の複数のレプリカ間で行われる通信であり、これを**レプリカ間通信**と呼ぶ。レプリカ間通信は以下の各段階において発生する。

- あるレプリカにおける出力属性を他のレプリカに渡す
- レプリカメンバーに関する情報の送受信

上記出力属性の送受信は、APR におけるレプリカ間での計算結果の共有を実現する。レプリカメンバーに関する情報の送受信に関しては、第 6 章で詳しく述べる。

通信コストの分析

レプリカ内通信について： レプリカ内通信に関しては、出力属性値と関数分解結果を管理するモジュールを実行している PE への通信の集中が問題になる可能

性がある。通信集中の発生を決定する要因として以下のような事項が挙げられる。

1. 実行するアプリケーションの粒度

2. 計算資源の数

3. レプリカを管理する実行時システムの配置とその計算資源の通信性能

実行するアプリケーションの粒度、つまりアプリケーション中で定義される個々の関数の実行に要する時間は、入力アプリケーションと PE の計算能力に依存する。

2,3 の事項に関しては、実装環境を考慮に入れて評価を行う必要があり、第 5 章で詳しく述べる。

レプリカ間通信について： 耐故障性を保証するためには、レプリカは障害の発生に関して各々隔離 (isolate) されている必要がある。つまりあるレプリカ R_i の障害は他の全てのレプリカ $R_n(n \neq i)$ に波及してはならない。このためレプリカは各々物理的に離れた場所に配置されることが耐故障性の保証という観点からは望ましい。しかし異なるレプリカに存在する PE 間の通信コストは、同一レプリカ内に存在する PE 間の通信コストと比較して著しく高い場合が多い。さらにレプリカの数が増加した場合はリモートに存在するレプリカとの通信量の増加を招き、レプリカ数のスケーラビリティを確保するためには問題となる。

しかし、耐故障性を持つシステムを構築する際には TMR(Triple Modular Redundancy) が 1-value failure と 2-crash failure に対処するために十分であることから、一般にレプリカ数は 3 程度が多く用いられる [Jal94]。さらにレプリカの数が増加した場合においても、分散環境において複数の実体が値を分配するために必要なコストは、論理的にはその実体の数に対してたかだか線形の増加で抑えることができるということが知られている [Lam78][LK00]。これらの事実から、現実にアプリケーションで要求されるレプリカ数に関するレプリカ間通信のスケーラビリティは確保されているものとする。

4.2 APR 関数起動アルゴリズムの定式化

4.2.1 データ構造およびアルゴリズム

ここではRAFT 資源割り当てアルゴリズムを定義するために最小限必要な、APR 関数起動アルゴリズムに関する定式化を行なう。

レプリカ識別子 (r_id)

定義 1 (レプリカ識別子 (r_id)) システム起動時に全てのレプリカに 1 から始まる連続する自然数でユニークな識別子を付け、これを r_id(レプリカ ID) と呼ぶ。レプリカと r_id の対応は、システムの起動から終了まで変化しない。

□

r_id はシステムの論理的な構成要素を指定する普遍の数値であり、システムの再構成が行われる場合にも変化しない。

モジュール分解リスト ($Dlist$)

APR ではレプリカ毎にモジュールの計算順序を変えるアルゴリズムを提案している。このアルゴリズムを実装するために、単一の関数分解に注目してこの分解によって新たに生成されたモジュールに対して、起動要求に関する順序付けを行ったものを $Dlist$ として定義する。

定義 2 (モジュール分解リスト ($Dlist$)) 総レプリカ数が r で、r_id が $j(1 \leq j \leq r)$ であるレプリカ R_j のあるモジュール M_k が関数分解によって n 個のサブモジュール $M_m(1 \leq m \leq n)$ を生成することを考える。このとき R_j 上の M_k における分解リスト $Dlist_{(R_j,M_k)}$ は、

$$Dlist_{(R_j,M_k)} = (M_p, M_{(p+1)}, \ldots, M_{(n-1)}, M_n, M_1, M_2, \ldots, M_{(p-1)}) \tag{4.5}$$

$$\text{where } p = \begin{cases} j & (\text{if } j = n \times m \ (n \in \mathcal{N})) \\ j \bmod m & (\text{otherwise}) \end{cases}$$

ただしモジュール M_a がモジュール M_b の出力に直接依存している場合、すなわち

$M_a(x_1, \cdots, x_n \mid y_1, \cdots, y_m)$ における $x_1, \cdots, x_n$ 中のいづれかの要素が M_b の出力で定義される場合、$Dlist$ 中の M_a のエントリに依存属性 $dep(M_b)$ を付加する。

□

モジュールの状態 (S_M)

定義 3 (モジュールの状態 (S_M)) 起動後のモジュール M_i の状態 S_{M_i} は必ず、Wait, Ready, Running, Suspend, Complete, Stopped のいずれかである。

□

起動要求リスト

起動要求リスト $Wlist_{R_j}$ は、レプリカ j 全体において、その時点における起動要求の高い順序で、定義3(モジュールの状態) に示した属性 S_M を持つモジュールを並べたリストである。$Wlist_{R_j}$ はレプリカを生成した直後に、$S_M =$ Ready なるルートモジュールだけを要素とするリストとして作成される。

定義 4 (起動要求リスト (*Wlist*)) あるレプリカ R_j 中の $Wlist_{R_j}$ は

$$Wlist_{R_j} = \text{list of } MS_i \tag{4.6}$$

ただし、$MS_i = \text{pair of (name of } M_i; S_{M_i})$

□

上記の定義 4 に従ってルートモジュールのみのリストとして生成された $Wlist_{R_j}$ は、関数分解が進行すると以下のアルゴリズムに従って次々に更新される。

起動要求リストのアップデート

実行中に同一レプリカ内のモジュールにおいて $Dlist$ が作成されると直ちに、$Wlist_{R_j}$ に対して以下に定義するアップデートが行われる。

アルゴリズム 1 (起動要求リストのアップデート ($Wlist.update$)) あるレプリカ R_j 中のモジュール M_k が関数分解を行ってリスト $Dlist_{(R_j,M_k)} = M_{k1}, M_{k2}, \ldots, M_{kn}$ からなるサブモジュールのリストを生じた場合、レプリカ R_j における起動要求リスト $Wlist_{R_j}$ は以下に示す操作により再構成される。

$$Wlist_{R_j} = (\ldots, MS_k) \ @ \ f(Dlist_{(R_j,M_k)}) \ @ \ (MS_{(k+1)}, \ldots, MS_{(j-1)}) \qquad (4.7)$$

ただし関数 f は、$Dlist$ から全ての要素が Ready または Wait 状態であるような MS_i のリストを生成する関数である。

□

計算起動ポリシーの変更 ($ACMS$)

バリュー障害を仮定している場合は、障害発生時の最悪のリカバリ時間を低くするため、ACMS によって起動優先順序の変更が行なわれる (第 3.4 節)。レプリカ R_i における起動優先順序ポリシーは、現時点まで R_{i+1}(存在しなければ R_1) が持っていた起動優先順序ポリシーに置き換えられる。

アルゴリズム 2 (ACMS 操作) 総レプリカ数 r のシステムで、定義 2、定義 4、アルゴリズム 1 に関して、レプリカ IDj を j_n ($j_n = j + 1 (\leq r,$ otherwise $1)$) と読み替える。レプリカ内およびモジュール内に $Dlist_{j_n}, Wlist_{j_n}$ が既に存在する場合は、これを使用し、$Dlist_{j_n}$ の起動要求を行なう。新たな $Dlist_{j_n}, Wlist_{j_n}$ が無い場合は新規に作成し、ルートモジュールから計算が行なわれる。

□

4.2.2 実行例

以上のデータ構造とアルゴリズムの定義に従い、APR の関数分解が進行する様子を図 4.3 に示す[4]。

[4]ここでは説明を容易にするため、入力属性の依存関係が存在せず、ACMS が適用されない場合の例を示す

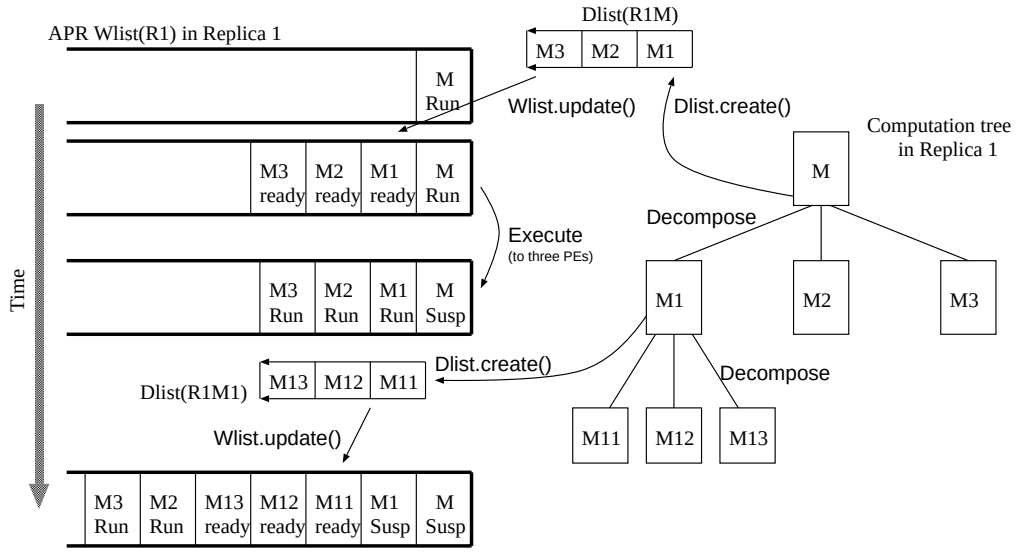

図 4.3: APR キューの論理的な構造と動作

$Wlist_{R_1}$ はレプリカ R_1 の起動時に、ルートモジュール M の名前と M の状態である Ready の組、すなわち $(M, Ready)$ のみを要素として作成される。計算が起動されると、$Wlist_{R_1}$ の最も前にある Ready 状態にあるモジュールから順に開始される。M は Running に状態を変更されて起動し、関数分解を行う。この関数分解により M の $Dlist_{(R_1,M)} = (M_1, M_2, M_3)$ が生成される。$Dlist_{(R_1,M)}$ の全ての要素は Ready 状態として $Wlist_{R_1}$ に追加される。

M_1, M_2, M_3 およびそれらのサブモジュールについても同様に、$Dlist$ の作成と $Wlist$ の更新、そして計算の起動が行われることによって、レプリカ R_1 における全ての関数分解が完了するまで継続して実行される。

第 5 章

RAFT 資源管理システム

APR は第 2 章で述べたように、関数型計算モデルを用いて分散環境下で耐故障性を保証するための複製技術として提案されている。しかし、主に計算の起動順序を定義している一方で、使用可能な計算資源をそれらの計算で使用する具体的な方法については言及していない。

本研究で提案する RAFT(Resource Allocation for Fault Tolerance) は、FTAG 言語で記述された APR アプリケーションの計算タスクを分散環境に存在する利用可能な資源に割り当てる際に必要なアルゴリズムと操作を提供する。

具体的にはまず、APR の論理的な計算単位であるモジュールを分散環境に存在する計算資源上で実行するために、より細粒度の RAFT プロセスという処理の単位に分解する。また RAFT 資源管理システムは、分散環境に存在する利用可能な計算資源の状態を把握してこれらの管理を行う。

さらに RAFT は、障害発生時には基本的な資源割り当てアルゴリズムとは異なるいくつかの操作を提供する。クラッシュ障害発生時には物理的な計算資源と論理的な計算モジュールとの対応付けに関する管理を行う。またバリュー障害発生時には、リカバリに必要な計算に対する資源割り当てを特別に扱うことによってリカバリ時間の削減を行なう。

これらによって RAFT は、計算資源利用効率の向上と計算終了までの時間およびリカバリに要する時間の短縮を実現する。RAFT アルゴリズムの位置づけを図 5.1 に示す。

ユーザは関数型で記述したアプリケーションプログラムをシステムに入力する。

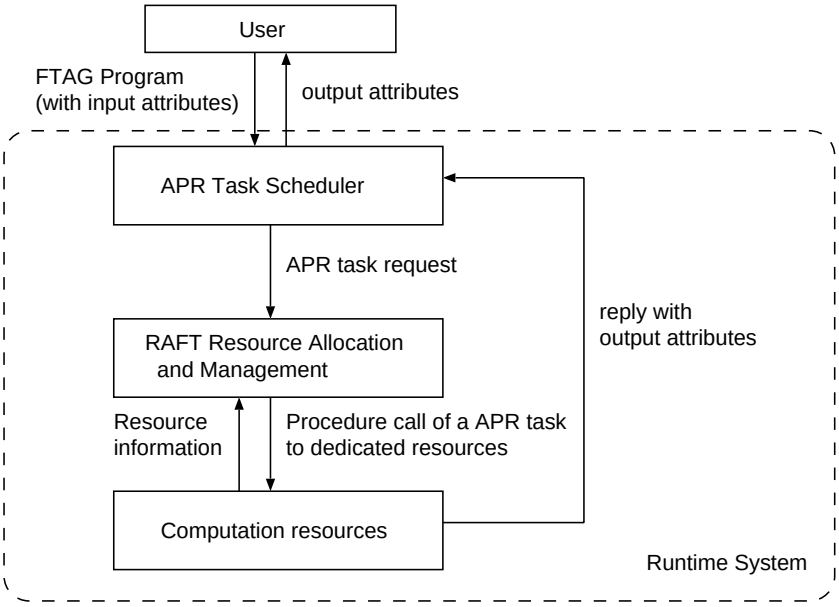

図 5.1: RAFT の位置づけ

APR スケジューラによって、入力されたアプリケーションプログラムは関数の単位で解釈され、仮定する障害モデルに応じて、計算起動順序が随時決定される。RAFT 資源管理システムは、APR スケジューラによってスケジューリングされた論理的なモジュールという実体を、RAFT プロセスという分散環境での実行に適した単位に分割し、計算資源に割り当てて実行を行う。

5.1 RAFT への要求

RAFT アルゴリズムの決定に際して基本的に考慮するべき要求は、故障の存在の仮定とそれへの対処、および並列計算による計算完了までに必要な時間の短縮である。具体的には以下に示す事項を主要な要求として考慮した。

第一に資源割り当ての決定は動的に行われなければならない。FTAG 計算モデルに定義されている通り、関数型の計算モデルで意味のある計算を行うためには中間モジュールにおいて条件に応じた関数分解を行うことが必須である。このことから入力属性が決定する以前、つまり計算開始前に静的にスケジューリングおよび資源割り当てを行うことは不可能である。また、一般的には関数型のプログラミングスタイルで書かれたプログラムには再帰的なアルゴリズムが含まれるこ

とが多く、実行前に計算に必要な時間を見積もって静的なスケジューリングを行うことは困難である。

第二に、対象とする計算機環境が分散システムであり、対象アプリケーションが長時間に渡る科学技術計算であることから、計算実行中の利用可能計算資源の変化に柔軟に対応できるような資源割り当て方法である必要がある。利用可能資源の変化は受動的にはPEやリンクの障害によって発生する一方で、積極的に高速な計算資源を追加したり、障害発生の可能性が高い資源を代替の資源に置き換えるという行動によっても引き起こされる可能性がある。

第三に考慮した点として、この資源割り当てアルゴリズムはスケーラブルでなければならない。大規模な計算アプリケーションを短時間で完了するためには、多くの計算資源を用いなければならない。つまり、利用可能な計算資源の規模や計算タスクの数に対して、十分なスケーラビリティを確保しながら可能な限り精度の良い資源割り当てを行う必要がある。

5.2 RAFTへの入力

RAFT実行時システムは各レプリカ上に1つ存在し、以下に示す3つの情報を入力として計算を行い、資源の割り当てを決定する。

- APRの起動要求リスト ($Wlist$)
- 利用可能な計算資源に関する情報
- 仮定する障害モデルに関する情報

本章の以降の節ではまずはじめに、第4.2節で定義した$Wlist$によって各レプリカに対して与えられるAPRの論理的な計算タスクを、分散環境上のPE上のプロセスに対応づける方法について述べる。次に利用可能な計算資源に関する情報の管理について述べる。続いて基本的なRAFTの資源割り当てアルゴリズムを述べた後、障害への対処時の資源管理のアルゴリズムについて述べる。本章の最後では利用可能資源の変化への対応と、操作コストに関する考察を行う。

5.3 RAFT プロセス

第 4 章における考察と定式化によって、計算実行時の APR のモジュールが持つ特徴を論理的な側面から明らかにした。本節ではこれらの振る舞いが明確になったモジュールを、分散環境における現実のプロセス[1]に対応させる方法を述べる。

5.3.1 細粒度プロセスの必要性

4.1.2 節で述べた通り、論理的には APR におけるスレッドは関数分解を起こした後、サブモジュールを実行するスレッドからの出力属性を待つ間は Suspend 状態にある。その後サブモジュールからの出力属性の入力が完了すると $resume$ して再び Running 状態に遷移し、CPU 時間とメモリの消費 R_{rs} を再開する。

上記の論理的な一つのモジュールの一連の活動を全て、分散環境における物理的な一つのプロセスとして実装することは、以下に示すような問題がある。

第 1 に、単一 PE におけるクラッシュ障害の発生が複数のモジュールの障害を引き起こす。Suspend 中の全てのモジュールはサブモジュールの計算が進行している

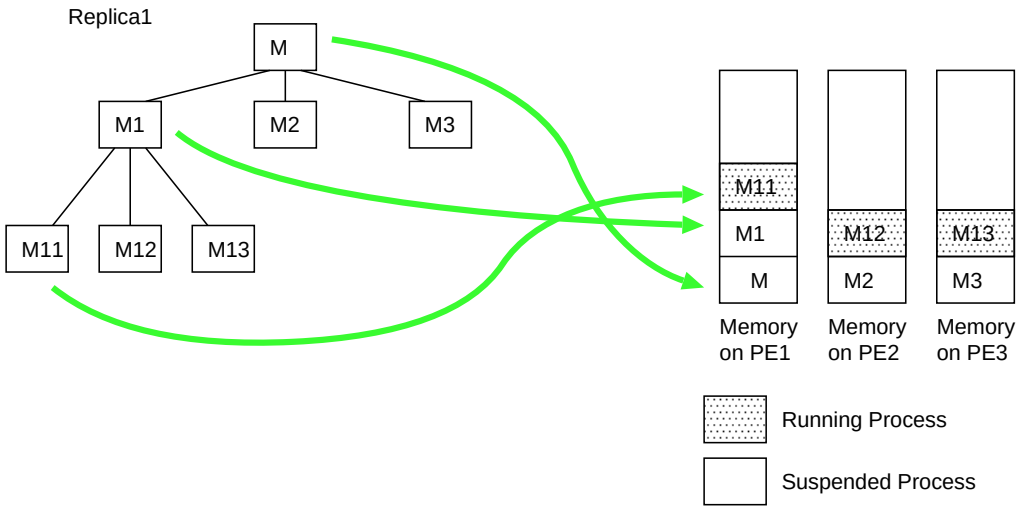

図 5.2: Suspend モジュールの資源消費

間 Resident 状態、つまり PE 上に存在しており、この単一の PE のクラッシュ障害

[1]本章では分散環境における物理的な処理単位であるオペレーティングシステム上でのプロセスおよびスレッドを総称して「プロセス」と呼ぶ。

が、複数の論理的に依存関係を持たない計算を同時に停止させる結果となる。

例えば図5.2に示す計算中の状態を考える。この時点ではすでに、M, M_1, M_2, M_3 の関数分解が行われ、M_{11}, M_{12}, M_{13} の関数分解もしくはリーフノードとしての属性の計算が実行されている。このとき M_{11} を計算中の PE1 にクラッシュ障害が発生した場合、M_1 および M の計算も障害によって中断され、リカバリを行うためにはこれらのモジュールの計算も再実行される必要がある。

第 2 に、Suspend 中のモジュールが消費し続ける資源 R_{susp} が、単一の PE に関して大きくなるという問題がある。論理的なモジュールの計算スレッドを単純に単一のプロセスに対応づけた場合の記憶領域の消費を、図 5.2 の右側に図示する。例ではレプリカ 1 において、図の計算木に示す部分までの関数分解が行なわれた時点における各々の PE の記憶領域の消費を表している。図に表わされている時刻において、M, M_1, M_2, M_3 は関数分解を起こした後に Suspend 状態で記憶領域を消費し続けている[2]。R_{susp} はモジュールの計算が起動されてから出力属性が得られるまでの間に渡って消費されるため、M は全体の計算の実行が開始されてから完了するまでの間に渡り、R_{susp} の消費を続ける。

第 3 の問題として、これら複数の Suspend 状態のモジュールがいつサブモジュールからの出力属得て $resume$ し、Running 状態に遷移するかを予測することは非常に困難である[3]。このためある PE 上で短時間に複数のモジュールが resume 可能な状態に遷移すると、当該 PE における負荷が著しく高くなる (もしくは起動待ちのタスクが増える) 可能性がある。これは事実上、ロードバランシング (実行中のプロセスを制御対象にした負荷分散) の実現を不可能にする。

これらの理由により RAFT では、モジュール実行の論理的なスレッドを、以下に示すような複数の物理的な処理の流れに分割する。

[2]各々のモジュールをオペレーティングシステム上のスレッドで実装したときのローカル変数に関しても、別のプロセスとして実装した場合と記憶領域の規模は異なるが同様の議論が成り立つ。

[3]当該スレッドを親モジュールとする部分木全体の計算が終わることを予測するのは、APR が他レプリカからの属性の受信するという特徴を考慮すると不可能である。

5.3.2 RAFT プロセスの定義

RAFT プロセスは、FTAG で定義されたモジュールの論理的な活動を、分散環境における複数の細粒度のプロセスに対応づけたものである。RAFT プロセスの定義を行なう前にここで再度 FTAG の関数分解の定義に注目する。

FTAG の特徴

FTAG の全てのアプリケーションは、以下の 3 種類のモジュール計算の組合わせで表されるような純粋な関数の集合であった (第 3.1 節参照)。
基本モジュール：

$$M(x_1, \ldots, x_n \mid y_1, \ldots, y_m) \Rightarrow return \quad \textbf{where} \quad \textbf{E}$$

モジュール分解：

$$M(x_1, \ldots, x_n \mid y_1, \ldots, y_m) \Rightarrow M_1 \ldots M_k \quad \textbf{where} \quad \textbf{E}$$

条件付き分解：

$$\begin{array}{l} M(x_1, \ldots, x_n | y_1, \ldots, y_m) \Rightarrow \\ \quad [\quad C_1 \quad\quad\quad\quad \rightarrow D_1 \\ \quad | \quad\quad \vdots \\ \quad | \quad C_n \quad\quad\quad\quad \rightarrow D_n \\ \quad | \quad \textbf{otherwise} \quad \rightarrow D_{def} \quad] \end{array}$$

ある FTAG のモジュール $M(x_1, \ldots, x_n \mid y_1, \ldots, y_m)$ が関数分解および条件付き関数分解を実行するとき、関数分解の結果を得るのに必要な情報は入力属性 $(x_1, \ldots, x_n)$ のみである。つまり関数分解に必要かつ十分な情報は以下の 2 つである。

1. 関数分解の定義 (モジュール M のプログラム)
2. M に対する全ての入力属性 $(x_1, \ldots, x_n)$

関数分解の定義は、モジュール M の計算を実行するプログラム中に含まれる。モジュール M が起動されるのは M の状態属性 S_M が Ready の場合であるから、M に対する入力属性 $(x_1, \ldots, x_n)$ は実行時にはすでに決定している。

M が基本モジュールの場合は関数分解が行われないため、上記 2 つの情報から直接 M の出力属性 $(y_1, \ldots, y_m)$ が得られる。

また、属性関係式 $\mathbf{E}$ に含まれる関数は、入力属性とサブモジュールからの出力属性によって、当該モジュールの出力を評価するための関数である。つまり FTAG においてあるモジュール M の出力属性を定義するために必要かつ十分な情報は、

1. M への入力属性 $(x_1, \ldots, x_n)$
2. 条件付き関数分解の結果 D_i
3. 上記分解によって実行される全てのサブモジュールの出力属性
4. M の出力を定義する属性関係式 $\mathbf{E}$

である。

上記 M への入力属性 $(x_1, \ldots, x_n)$ は APR の実行時システム内の計算木構造および安定記憶に保存された属性によって、M の起動時に決定している。各々のモジュールにおける条件付き関数分解の結果は、定義 2 によって定められた通り、モジュール M の $Dlist$ によって保存される。3 番目の、サブモジュールからの全ての出力属性は、サブモジュールにおける計算が全て完了した時点で得ることができる。また 4 番目の属性関係式 $\mathbf{E}$ の定義はモジュールの計算を実行するプログラムコード中に含まれる。

M の出力属性の合成は、上記の 4 つの情報全てがそろった任意の時刻と場所で行なうことができる。

モジュールのプロセスへの分割

以上の考察より RAFT では、関数計算という FTAG における 1 つの論理的な活動を、図 5.3 に示すように 2 つのプロセスに分割する。

図 5.3 の上半分は、FTAG におけるモジュールの論理的な状態の遷移を表している。この例は中間モジュールにおける計算の実行の様子であり、実行を始めるとまずはじめに関数分解を行い、サブモジュールを起動からの出力属性を待って Suspend 状態に入る。サブモジュールからの出力を全て得たモジュールは、再び Running 状態に遷移して合成属性の計算を行う。

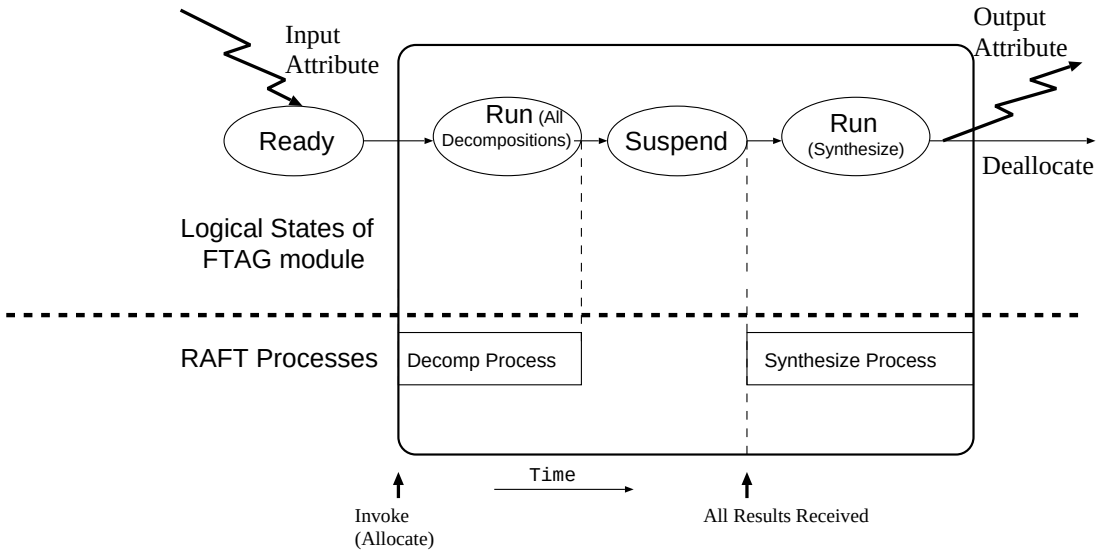

図 5.3: RAFT プロセス

図 5.3 の下半分はこのようなモジュールの活動に対応する RAFT プロセスを表す。図 5.3 において、この関数への入力はプロセス起動時に決定している。起動した $RAFT$ **分解プロセス** $(RAFT\ Decomposing\ process)$ は関数分解の処理を実行する。これにより条件付き関数分解の結果が決定し、このモジュールにおける $Dlist$ として保存される。全ての関数分解が完了すると関数分解プロセスは実行を終了する。

APR では、計算の結果は全て (他のレプリカに送信するために) 実行時システムに保存されるため、全てのサブモジュールにおいて出力属性が出力されたことを RAFT 実行時システムは検出することができる。この検出によって実行時システムは $RAFT$ **合成プロセス** $(RAFT\ Sythesizing\ process)$ を起動する。合成プロセスは先に示した 3 つの入力、すなわち入力属性、属性関係式、そしてサブモジュールからの出力によって出力属性を合成する。

定義 5 (RAFT プロセス) 関数分解を行なう論理的なモジュールを、2 つの RAFT プロセス RP によって実装する。モジュール M の全ての関数分解、すなわち M のサブモジュールの起動リスト $Dlist$ への挿入を行なうまでのプロセスを $RAFT$ **分解プロセス** $(RAFT\ Decomposing\ process)$ と呼び、$RP_d(M)$ と書く。サブモジュールの出力属性が全て揃った後に起動し、入力属性、サブモジュールからの出力属

性、そして属性関係式を用いて出力属性を計算するプロセスを $RAFT$ **合成プロセス** ($RAFT$ *synthesizing process*) と呼び、$RP_s(M)$ と書く。モジュールが FTAG 基本モジュール[4]である場合はそれを $RP_p(M)$ と書く。

□

以上に定義した RAFT プロセスを、その入力と出力の型に注目してまとめると以下のように書くことができる。

RAFT 分解プロセス： モジュール M の RAFT 分解プロセスは、

$$\mathtt{RP_d} : (\mathtt{Mname} * \mathtt{in\_attr}) \rightarrow (\mathtt{Cname} * \mathtt{Dlist} * \mathtt{attr\_c}) \tag{5.1}$$

ここで、

- `Mname`：モジュール名 M
- `in_attr`：モジュール M への相続属性
- `Cname`：条件付き分解のケース C
- `Dlist`：M の関数分解結果
- `attr_c`：Dlist 中の M のサブモジュールへの入力属性

RAFT 合成プロセス： モジュール M の RAFT 合成プロセスは、

$$\mathtt{RP_s} : (\mathtt{Mname} * \mathtt{in\_attr} * \mathtt{Cname} * \mathtt{c\_out\_attrs}) \rightarrow \mathtt{out\_attr} \tag{5.2}$$

ここで、

- `c_out_attrs`：モジュール M のサブモジュールの出力のリスト
- `out_attr`：モジュール M の出力属性

RAFT における合成プロセスは APR のモジュールの起動要求と同様に、$Wlist$ に以下に示すような拡張を加えることによって資源に割り当てられる。RAFT プロセスのリスト $RPlist$ を以下のように定義する。

[4]それ以上関数分解が行われないモジュール。第 3.1 節参照。

定義 6 (RAFT プロセス起動要求リスト (*RPlist*)) あるレプリカ R_j 中の $RPlist_{R_j}$ は

$$RPlist_{R_j} = \text{list of } (RP, S_{rp}) \tag{5.3}$$

ここで、

$RP = \{RP_d, RP_s, RP_p\}$,

S_{rp} は RP の状態で、{Wait, Ready, Running, Stopped, Complete} のいずれかである。

□

Suspend 状態は、モジュールの RP_dおよび RP_s への分解により排除されるため、RAFT プロセスでは APR のモジュールの論理的な状態遷移とは異なり、Suspend 状態は取らない。

RAFT プロセスの生成および実行完了時には、実行時システムは $RPlist$ を以下の規則にしたがって更新する。

定義 7 (RPlist のアップデート) レプリカ i において、APR のモジュール M に対する起動要求によって、状態 S_{rp} = Ready である RAFT 分解プロセス $RP_d(M)$ を $RPlist_{R_i}$ に追加する。M が FTAG 基本モジュールである場合は $RP_p(M)$ を同様に追加する。RAFT 分解プロセス $RP_d(M)$ または $RP_p(M)$ が実行を開始するとその状態を Running に変更する。$RP_d(M)$ が出力を行なって実行を完了すると、RAFT 合成プロセス $RP_s(M)$ を Wait 状態で $RPlist_{R_i}$ に追加する。$RP_p(M)$ が出力を行った場合は、$RPlist_{R_i}$ 中の $RP_p(M)$ の状態を Complete に変更する。$RP_s(M)$ の実行に必要な属性が全て出力されている場合、そのプロセスの状態 S_{rp} を Ready に変更する。$RP_s(M)$ が出力を行なって実行を完了すると $RPlist_{R_i}$ 中の $RP_s(M)$ の状態を Complete に変更する。

□

以下に定義 6 で行った RPlist の定義、および定義 7 に述べた RPlist のアップデートの操作を行うプログラムの疑似コードを図 5.4 に示す。

```
type ListenEvents = E_APRinvoke | E_RPstart | E_RPcomplete
type Srp = Wait | Ready | Running | Stopped | Complete
type RP = {rpname: RPname ; srp: Srp}
type RPlist = RP list
type M = modulename

let createRP (m:M) = ({m_d ; Ready} : RP)
let makerunRP (rp:RP) = ({rp ; Run} : RP)
let makecompleteRP (rp:RP) =
  ({rpname ; Complete} : RP)
let RPlist.replace rplist rp =
  List.replace (List.find rp.rpname rplist) rp

(* RPlist アップデート操作 *)
let RPlist.update rplist m =
  match rplist with
    [] -> createRP m
  | l -> l :: (createRP m)

(* RAFTプロセスの実行開始 *)
let RPstart rplist (rp: RP) =
  RAFT.invoke_ rp ;
  rplist := RPlist.replace rplist (makerunRP rp)

(* RAFTプロセスの実行完了 *)
let RPcomplete (rp: RP) =
  rplist := RPlist.replace rplist (makecompleteRP rp)
```

図 5.4: RPlistアップデートアルゴリズム

5.3.3 RAFT プロセスに関する考察

RAFT プロセスによって得られる効果

RAFT プロセスの導入によって、論理的に Suspend 状態にあるモジュールが物理的に PE 上に長時間 (サブモジュールの出力属性が計算されるまでの時間) 存在することによって障害発生時のリカバリのコストが大きくなるという問題を回避できる。

また、Suspend 状態のモジュールを PE 上に配置しないことにより、記憶領域 R_{susp} の消費を抑制する。

さらにプロセスを分割することによって、負荷分散の機構を実装するのが容易になる。仮に Suspend を含めて 1 つのプロセスとして実装する場合は、resume 時のスレッドは既に特定された PE 上に存在するために、Suspend 状態にあるプロセスに対するマイグレーションの操作が必要になる。しかし耐故障性の保証のためには、論理的に実行中のモジュールに対するマイグレーションの操作中には障害が発生してはならず、これは分散環境において障害が発生する (及びこれらに対処する) という仮定と矛盾する。このことから、Suspend 中のモジュールをそのままマイグレーションさせるというアプローチをとることはできない。

RAFT プロセスの導入により、実行時システムは、論理的に一つであったモジュールを、リカバリ可能な 2 つのプロセスに分解して扱っているという見方ができる。このため実行時システムは、モジュールの起動、すなわち分解プロセスへの資源割り当てと同様の方法で、合成プロセスを利用可能資源に割り当てることができ、これにより負荷分散を考慮にいれた資源割り当て (第 5.5 節) を耐故障性を保証しながら行うことができる。

RAFT プロセスは APR のモジュールという論理的な計算単位をより細粒度に分割したものであり、モジュールと同様に関数としての性質を持っている。つまり RAFT プロセスは式 (5.1), 式 (5.2) に示したように、入力が完了してプロセスが起動してから出力を行うまでの間に、外部からのデータを待ってブロックするという事は無い。このため、資源を占有するプロセスがブロックするため起こる飢餓状態は発生しない。

プロセス生成コスト

RAFT プロセスに関しては、プロセス生成のコストが問題になる可能性がある。RAFT プロセスを導入したことによって、1 つの論理的な中間のモジュールに対応するプロセスの生成は、2 回必要である。プロセス生成のコストは特に、属性関係式における計算量が小さなアプリケーションが再帰的に関数分解を行なうようなアプリケーションにおいて顕著になる。細粒度の再帰計算に関するこの問題は、関数型の計算を積極的に複数の計算資源に割り当てる際における一般的な問題として知られており、多くの場合アプリケーションプログラマがプログラム中に言語毎に定義されたアサーションを挿入することで回避される [PvE93]。すなわち資源割り当てを行う必要がないレベルの細粒度の計算に対しては、並列化及び計算資源の新規割り当てを行わないような記述を行う[5]。

また記憶領域が豊富に利用可能な環境においては、RAFT プロセスを実行するスレッド (またはプロセス) をあらかじめ起動しておき、相続属性が入力された時点で計算を開始する評価器のようにして用いることも可能である。

FTAG を用いたアプリケーションの記述では、該当する計算を一つの FTAG 基本モジュールとして記述することにより、プログラマは RAFT プロセスへの分割を行われない単一のプロセスを宣言することができる。FTAG 基本モジュールの最適な大きさに関しては、使用する PE などの利用可能資源によって大きく異なるため、次節以降で資源情報に関する詳細を述べた後、本章の最後で議論する。

5.4 資源情報の管理

RAFT 実行時システムは各々のレプリカに存在し、それぞれのレプリカが計算に利用できる全ての PE に関する情報を保持している。これら PE は完全にアイドルな PE が含まれるのは言うまでもないが、いくつかの計算が割り当てられて実行中の状態にある PE も含む。さらに利用可能な PE には様々な仕様のものがあり、RAFT はこれら PE が計算を行う能力に関する情報を持つ必要がある。

[5]逆に並列化を行う部分のみに対してアサーションを記述する方法も数多くある ([J.H99][Ser99] など)。

PE の処理能力に関する情報

個々の PE の計算能力を示すために、PE が持つ属性として以下に示す PF 属性 (PE Factor) を導入する。

定義 8 (*PF* 属性の定義) PE の計算能力を以下の PF によって表す。

$$PF(t) = F_{pf}(S_{cpu}, M_m, LD_k(t), M_a(t))$$

ここで、

S_{cpu}: CPU 単体の処理速度

M_m: 搭載する実メモリ

$LD_k(t)$: 時刻 t における CPU の負荷

$M_a(t)$: 時刻 t における使用可能メモリ

F_{pf}: PF 算出関数

各 PE は T_{pf} 時間毎に PF を算出する。算出した $PF(t)$ が $PF(t-1)$ よりも $Thr_{pf}\%$ 以上の変化を検出した場合、PF の変化を RAFT 実行時システムに通知する。

□

上記パラメータの中で実行中に変化するものは LD_k, M_a である。PF を決定するために RAFT 実行時システムは LD_k および M_a の情報を定期的にオペレーティングシステムから獲得する。

F_{pf} の定義の妥当性は、実行環境やアプリケーションに強く依存する。例えば記憶領域に注目すると、搭載する実メモリ M_m は与えられる利用可能なマシンに依存し、使用可能メモリはある時刻 t において PE 上で実行されている RAFT プロセスを含む全てのプロセスの記憶領域の使用量に依存する。このため RAFT 資源管理システムは、ユーザがこれらの情報を実行時システムに与えるためのインターフェースを設ける必要がある。

PE の集合に関する情報

RAFT は内部属性として pe_table を保持する。pe_table は、個々の PE の識別子である pe_id をキーとして、定義 8 に示した PF、当該 PE 上で実行中の RAFT プロセスに関する情報をデータとして持つ以下に示すようなデータ構造である。

定義 9 (*pe_table* の定義) RAFT 実行時システムは以下に定義する pe_table によって資源情報を管理する。

$$pe\_table = \text{list of } (pe\_id, PF(t), RPl(t))$$

(ただし、$RPl(t)$ は時刻 t においてその PE 上で実行中の RAFT プロセス名のリスト)

□

pe_table の情報のうち、$PF(t), PRl(t)$ は時間の関数であり、計算実行中に常に変化する。このため RAFT 実行時システムは pe_table に含まれる情報の更新を実行中に行う。

具体的には $PF(t)$ の更新は定義 8 で述べた通り、PF の値に Thr_{pf} 以上の変化が生じた時に行われる。また RPl の変更は実行時システムによって行われる。実行時システムは RAFT プロセスの起動を行うため、起動時には必ず RPl を変更することができる。また RAFT プロセス実行完了時には出力属性が RAFT プロセスから実行時システムに対して必ず送信されるため、この通信をトリガとして RPl を更新することができる。

5.5 資源の割り当て

5.5.1 計算起動時の資源割り当て

これまでに APR による論理的なモジュールという単位でのスケジューリングと、分散環境で実行可能な RAFT プロセスへの分割を行い、資源情報の管理方法についても述べた。本節では RAFT プロセスを利用可能資源に割り当てる方法について述べる。

RAFT は APR からの資源割り当て要求が発生した時点における pe_table の情報をもとに、最適な資源を要求された RAFT プロセスに割り当てる。

タスクのスケジューリングに関する特定のアルゴリズムを考慮せずに木構造の計算を行う場合には、その木構造の根に近い部分木から順に大きな部分木に対して資源を割り当てていくことが、公平な資源割り当てを行なうために一般的には有効であることが知られている [Knu73][Kui89]。

一方 APR ではそれぞれのレプリカが $Wlist$ のアップデートアルゴリズム (アルゴリズム 1) で定義されるアルゴリズムに従って木構造中の異なる部分から局所的に計算を行っていくことにより、耐故障性と計算時間の短縮を実現する。さらに RAFT では 1 つのモジュールを 2 つの RAFT プロセスに分解した結果、実際に計算を行うプロセスのリストである $RPlist$ には、関数分解、関数合成の各々のプロセスが挿入される。資源割り当ては基本的には、RAFT プロセスのリストである $RPlist$ の先頭から逐次適切な資源を割り当てる形で行う。

ただし RAFT では、関数分解によって複数のサブモジュールが得られたときに、$RPlist$ 中で最も前に現れるモジュールについて、このモジュールを実行する RAFT プロセスを親モジュールと同じ PE に割り当てる。これはあるモジュールが関数分解を行った直後は、論理的にそのモジュールは Suspend 状態に遷移する、つまり物理的には RAFT 分解プロセスが終了するため、PE は 1 つのプロセスを実行可能な状態になることが常に期待できるからである。

最初のサブモジュールを計算する RAFT プロセスを親モジュールと同一の PE に割り当てた後に、RAFT は 2 番目以降のモジュールの RAFT プロセスに対して、pe_table 中の資源を以下のアルゴリズムにしたがって割り当てる。

アルゴリズム 3 (RAFT 基本アルゴリズム) レプリカ i において本アルゴリズムが起動されたとき、Wait 属性を持つ RAFT プロセスが存在する場合はその RAFT プロセスに相当するモジュール M_j の持つ $dep(M_j)$ 属性を調べることにより依存しているモジュールの出力属性が存在するかを判定し、既に存在している場合は Ready に状態を変更する。次に RAFT は $RPlist_{(R_i)}$ の先頭から Ready 属性を持つモジュールを検索する。$RPlist_{(R_i)}$ の中で Ready 属性を持つモジュールが、$Dlist$ の先頭のモジュールである場合はを親モジュールの RAFT 分解プロセスと同一 PE に割り当てる。次に、$RPlist_{(R_i)}$ の続くモジュールを pe_table 中で $RPl = \phi \wedge max(PF)$ を満たす PE に割り当てる。

□

RAFT 基本アルゴリズムは特別な場合[6] を除き、 1 つ以上の PE において実行中の RAFT プロセスが存在せず、かつ $RPlist \neq []$ の場合のみ起動される。

[6]障害への対処を含む特別な場合については、第 5.6 節で述べる。

アルゴリズム 4 (RAFT 基本アルゴリズムの起動時刻) RAFT 実行時システムは PE 上のプロセスの終了と $RPlist$ のアップデートの全ての通知を受ける。$Pl_R = \phi$ である PE がシステム中に存在し、かつ $RPlist \neq []$ の場合、RAFT 基本アルゴリズムを起動する。

□

アルゴリズム 3 およびアルゴリズム 4 において定義したアルゴリズムの疑似コードを以下に示す。

```
type RPlist = RP list
type thispe = pe_id

let RAFTbase rplist idlepe=
  (* Wait 状態の RAFT プロセスを探す。可能ならば状態を変更 *)
  inspect_waiting rplist ;
  (* Ready 状態の先頭のプロセスを選択 *)
  let n = select_proc rplist in

  (* Dlist 中の先頭のモジュールに相当するならこの PE で起動 *)
  if (hd(dlist_of(n)) = n) then
    (* invoke on this pe *)
    RAFT.invoke n thispe
  else
    RAFT.invoke n idlepe
```

図 5.5: RAFT 基本アルゴリズム

図 5.6 は RAFT 基本アルゴリズムの動作例である。この例では論理的には M_1 のサブモジュールは $Wlist$ 上において M_2 の前に挿入されるため、M_2 への資源割り当ては M_1 をルートとする部分木への資源割り当てよりも後に遅らされる。また、M_1 のサブモジュール M_{11}, M_{12}, M_{13} のうちの M_{11} のみが M_1 と同じ PE に対して割り当てられている。

実行時システム内部における RAFT プロセスの起動までの操作は以下に示すようになる。レプリカ 1 において、ルートモジュール M は分解を行って $Dlist_{R1,M} =$

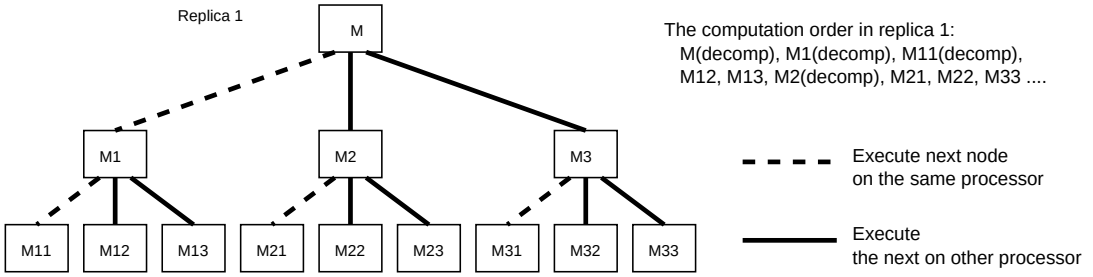

図 5.6: RAFT 基本アルゴリズム実行中の計算木

(M_1, M_2, M_3) を生成する。このモジュール分解によってレプリカ 1 は、以下の $Wlist$ のアップデートを行う。

$$\begin{aligned} Wlist_{R_1} &= Wlist.update\ Wlist_{R_1}\ Dlist_{(R_1,M)} \\ &= Wlist.update\ MS\ Dlist_{(R_1,M)} \\ &= MS\ @\ Dlist_{(R_1,M)}\ @\ [] \\ &= (MS, MS_1, MS_2, MS_3) \end{aligned}$$

ここで MS_i は定義 4 で定義した、状態に関する属性付きのモジュールである。$Wlist$ は既に存在する M を示す MS に、新たに $Dlist$ として作成されたサブモジュールを追加した形になる。

$Wlist.update$ 操作は $Wlist$ の更新の完了と同時に RAFT 実行時システムに通知され、RAFT プロセスの生成を行う。実行時システムは RAFT プロセスの生成後直ちに $RPlist$ を更新する。

$$\begin{aligned} RPlist_{R_1} &= RPlist.update\ RPlist_{R_1}\ Dlist_{(R_1,M)} \\ &= RPlist.update\ [(RP_d(M), \mathsf{Wait}), (RP_s(M), \mathsf{Wait})], Dlist_{(R_1,M)} \\ &= [(RP_d(M), \mathsf{Wait}), (RP_s(M), \mathsf{Wait}), (RP_d(M_1), \mathsf{Ready}), \\ &\quad (RP_d(M_2), \mathsf{Ready}), (RP_d(M_3), \mathsf{Ready})] \end{aligned}$$

RAFT 実行時システムは、RAFT プロセスを生成して $RPlist$ を更新すると、RAFT 基本アルゴリズムを起動する。RAFT 基本アルゴリズムを実行することにより、Ready 状態にあった $(RP_d(M_1),\ RP_d(M_2),\ RP_d(M_3))$ が PE を割り当てられて状態を Running に変更され、各々の PE 上で実行を開始する。

5.5.2 不要になったプロセスの実行中断

APR では 1 つまたは複数のレプリカによってすでに妥当性を得た計算結果に関しては、他のレプリカは計算を行わない。例えばクラッシュ障害を仮定する場合は、あるモジュール M の合成属性は 1 つだけ必要であり、他のレプリカはこの合成属性を M の正しい計算結果として用いて計算を進める。このため、M の合成属性を計算する RAFT プロセス $RP_s(M)$ は、任意のある 1 つのレプリカによって計算されれば良く、$RP_s(M)$ の結果があるレプリカにおいて出力された時刻において、他のレプリカにおける $RP_s(M)$ は即時に停止して計算資源を解放することができる。

また n-バリュー障害を仮定する場合も同様に、M の $n+1$ 個の同一の出力が得られたときにその値は妥当であるとして保存し、モジュール M の正しい出力として他の計算で使用することができるため、他のレプリカにおいて計算中の $RP_s(M)$ は即時に停止して計算資源を解放することができる。

このため RAFT は図 4.1 の exception を起動する。これにより実行途中の RAFT プロセスの実行を中断して資源を解放し、他のレプリカから得られた出力を正しい結果として扱う。

妥当性が確認されて保存された出力は、次に RAFT 基本アルゴリズムが起動されたときに、Wait 状態にあるプロセスがあれば直ちに利用される。

5.5.3 操作コストに関する考察

pe_table からある時点における最適な資源を取り出す操作は、常に pe_table が PF(t) の値に従ってソートされている場合は $O(1)$ で完了する。しかしながら pe_table を常にソートしておくためには pe_table のアップデート操作 1 回毎に $O(n_{pe})$ (n_{pe} は利用可能な PE の数) の計算が必要であることから、pe_table が持つ情報の正しさと pe_table の管理に必要なコストのトレードオフが問題になる。またこれは PE 数が増加したときには特に大きなコストとなり、PE 数に関するスケーラビリティを保つことができない。このため RAFT では、pe_table を常にソートされた状態に保つことはせず、APR からの資源要求発生時に pe_table から最適な資源を発見するために RAFT 実行時システムが $O(n_{pe})$ の操作を行うこととする。

5.6 障害への対処

本システムが対象とする利用可能資源に発生する障害には大きく分けて 2 種類が存在する。第 1 の障害は PE のクラッシュにより発生する障害であり、第 2 の障害は関数の計算結果および分解結果に関する妥当性試験によって発生するバリュー障害である。

RAFT では基本的に、RAFT プロセス実行中の PE のクラッシュ障害に対しては、その RAFT プロセスを実行しているレプリカ内部の実行時システムによって対処する。またレプリカ全体を制御する実行時システムの障害に対しては、レプリカ単位のクラッシュと見なして対処を行う。

バリュー障害の検出は、レプリカ間で RAFT プロセスの出力を共有することによって行う。共有する出力には、RAFT 合成プロセスの出力属性だけでなく、RAFT 分解プロセスの関数分解結果も含まれる。クラッシュ障害のリカバリとは異なりバリュー障害のリカバリは、幾つかの理由によって迅速なリカバリが要求される。RAFT ではバリュー障害の発生に対して迅速にリカバリを行うアルゴリズムを提供する。

本節の以降では、クラッシュおよびバリュー障害に対する RAFT システムの対応に関して述べる。

5.6.1 クラッシュ障害

あるレプリカ R_i の内部に存在する単一の PE におけるクラッシュは、レプリカ R_i 内部に存在する実行時システムによって対処を行なう。クラッシュ障害の発生は利用可能資源の変更とリカバリ、つまり当該 PE 上で実行していた RAFT プロセスの再スケジューリングを必要とする。

クラッシュ障害の仮定より、クラッシュした PE が既に出力した結果は全て正しい。つまりその PE が過去に出力して実行時システムに保存された RAFT 分解プロセスの結果と RAFT 合成プロセスの出力は、実行時システムが安定記憶から取り出して障害発生の後もそのまま使用する。

クラッシュに対するリカバリの操作は単純である。まず始めに、障害が発生した PE 上に割り当てられていた RAFT プロセス (RP_r と呼ぶ) は、実行時システム

が持つpe_table内の要素RPlから検索される。PEへの割り当てを既に行なわれたこのRAFTプロセスRP_rの実行に必要な入力は、必ず実行時システムによって既に保存されていることから、直ちにReady状態のプロセスとして$RPlist$に挿入される。最後に実行時システムは、障害を起こした資源のエントリをpe_tableから削除する。RAFTは$RPlist$のアップデートの通知を受信することによってRAFT基本アルゴリズムを起動する。Ready状態のRP_rは通常のRAFTプロセスと同様に起動され、リカバリが行われる。

クラッシュ障害に対処するRAFT実行時システムの動作に関するアルゴリズムを以下に示す。

```
ListenEvents = detect_crash

(* クラッシュしたPE上で実行されていたRAFTプロセスを得る *)
let findRP_r crash_pe =
  getRPbyPE (pe_table.lookup_pe crash_pe)

(* RAFTプロセスの状態をReadyに戻す関数 *)
let makereadyRP (m:M) = ({m_d ; Ready} : RP)

let CrashRecover crash_pe =
  (* リカバリを行うRAFTプロセス *)
  let rp_r = findRP_r crash_pe in
  RPlist.replace rplist (makereadyRP rp_r) ;
  RPlist.updatenotify
```

図 5.7: クラッシュ障害への対処のアルゴリズム

5.6.2 バリュー障害

バリュー障害モデルを仮定している際に、複数のレプリカによる同一の関数の出力または分解結果が一致しない場合、APRは他のレプリカに対して該当する関数の再計算の要求を行う。APRにおけるバリュー障害にへの対処は、計算木の部分木の単位で行われた。つまり、障害検出はモジュールの出力属性の不一致によっ

てのみ行われるため、属性合成時に検出された障害は、そのモジュールが属性を相続する時点まで遡ってリカバリを行う必要があった。

一方 RAFT では、APR の論理的なモジュールという単位をより細粒度の RAFT プロセスという単位に分割して出力を複数のレプリカで共有するため、分解プロセス RP_d および合成プロセス RP_s のそれぞれの段階において障害の検出が行われる。

バリュー障害の検出はレプリカ間の同一 RAFT プロセスにおける出力結果 (分解結果および合成属性) を比較することによって行われる。これは全てのレプリカにおける出力結果の保存を行う操作を以下のように定義することによって実装される。

$$\text{出力結果の保存} \stackrel{def}{=} \text{(出力結果の安定記憶への書き込み; 比較)}$$

障害が発生したと疑われる RAFT プロセスを実行したレプリカでは、障害の伝播を防ぐ必要がある。このため、計算木において障害が発生した RAFT プロセスに該当する部分木に関する新たな計算は起動せず、$Wlist$ 上で当該部分木よりも後にある (つまり障害との依存関係のない)Ready 状態のモジュールを実行するスレッドを起動する必要がある。この状態は、他のレプリカからの再計算の結果を受け取って障害の要因が確定し、妥当性が確かめられるか、もしくはリカバリが完了するまで続く。

障害検出後からリカバリが完了するまでの状態においては、計算木中の障害が発生した部分の計算だけ実行が遅れるため、計算完了までの時間が長くなる可能性がある。つまり他のレプリカにおいてリカバリのために行なわれる再計算が完了してリカバリが完了するまでは当該部分木の計算を開始することができないため、可能な限り短い時間でリカバリのための再計算を行うことが必要である。

このため RAFT は、バリュー障害が発生した場合は以下に示すアルゴリズムに従って再計算の起動を行なう。

アルゴリズム 5 (バリュー障害による再計算起動要求) あるレプリカ R_i がある RAFT プロセス RP_j の出力属性もしくは分解結果 out_{RP_j} を保存する際にバリュー障害を検出したとする。R_i 中の RAFT 実行時システムは全てのレプリカ $R_1, \ldots, R_n$(n は総レプリカ数) 中で RP_j の計算を行なっていないレプリカの中で pe_table 中の S_{cpu}

の合計が最も高いレプリカ R_r を選択する。R_i の実行時システムはグループマルチキャスト[7]によって障害の検知と R_r への RP_j のリカバリ要求を送信する。

□

ここで、RP_j の計算を行っていないレプリカは、n-バリュー障害の仮定より必ず存在する。

```
let rj = r_id
let rpname = nameof RP

let vf_found rj rpname =
  (* 障害を疑うRAFTプロセスに該当する
   * APRにおける部分木を取り出す *)
  let st = APR.get_st (rp2m rpname) in (* subtree *)
  let vf_mlist = List.flatten st in
  (* 該当する部分木の計算を一時停止するためにマークする *)
  let mark_vf rplist vf_mlist =
    match rplist with
      [] -> ()
    | x ->
      if (List.mem rp2m(hd x) vf_mlist) then
        RPlist.update rplist {rpname ; makerecoverRP}
      else ();
      mark_vf (tl rplist) vf_mlist
  in

  mark_vf rplist vf_mlist;

  (* バリュー障害suspectメッセージを送信 *)
  let m = {Vf_suspect; rpname} in
  Cast (m)
```

図 5.8: バリュー障害検出アルゴリズム

アルゴリズム 5 に示した再計算の起動を行うため、RAFT 基本アルゴリズムに対して以下の拡張を行なう。

[7]グループ通信に関する詳細は第 6 章で述べる。

アルゴリズム 6 (再計算起動要求に対する操作) 再計算起動要求を受けた R_r 上の RAFT は、要求される部分木に対する計算をリカバリであることを示す特別な属性 $recover$ を付加して $Wlist$ の先頭に挿入する。$Wlist$ 中で recover 属性を持つプロセスに対し RAFT は高い実行優先度 Pri_r を与えて直ちに起動する。

□

ここでは直ちにリカバリのための RAFT プロセスを起動する必要があるため、RAFT プロセス起動の条件の一つである、対象 PE における $(RPl = \phi)$ の条件は考慮されない。

アルゴリズム 5 とアルゴリズム 6 によってバリュー障害時の再計算が実行される。再計算の実行には、RAFT は pe_table 中のデータである RPl、すなわち既に実行を行なっているプロセスは考慮せずに直ちに Pri_r の優先度で計算の起動を行なう。またこのプロセスは、利用可能資源の中で最も高速なプロセッサを持つ PE、つまり S_{cpu} が最大の PE に対して割り当てる。

上記のアルゴリズムでは、有限であるプロセスの実行優先度を障害の数だけ仮定しなければならない。しかし現実的にはバリュー障害はリカバリが完了するまでの間にたかだか数回 (実行優先度の数未満) しか発生しないという仮定を行なうことが可能であるため、有限の実行優先度をリカバリプロセスに付加することは妥当であるとする。

5.7 計算資源の変化への対応

長時間に渡る計算実行中の利用可能資源は変化は、障害の発生以外に計算資源の削除要求や新たな計算資源の追加要求によって発生する。さらに計算資源の負荷が PE 上で行われる何らかの活動 (アプリケーションの計算とは依存関係のない活動) によって大きくなり、オペレーティングシステムによって計算アプリケーションに割り当てられる資源に変化が発生することがある。これら 3 つの資源変化に対する要求への RAFT のオペレーションについて述べる。

5.7.1 資源削除

ユーザからの計算資源の削除要求を受け取るとRAFT実行時システムはまずはじめに、RAFTプロセスへの該当計算資源の割り当てをそれ以降は行わないようにする。このためRAFTシステムはpe_table中の削除要求のあったPE(PE_dとする)の情報を特別な値、

$$PF(t) = \perp$$

に書き換える。その後、PE_dに既に割り当てられて実行中のRAFTプロセスの完了の後、つまりPE_dのRPlリストが[]になると、PE_dのエントリをpe_tableから削除することでシステムから削除し、ユーザに通知する。

即時に資源削除が必要な場合は、該当資源を即時に削除し、資源削除を障害として扱う必要がある。RAFTにおけるクラッシュ障害への対処は局所的かつ実行時システム内の処理のみを必要とするものであり、資源削除を単一のPEのクラッシュ障害として扱うことは可能である。しかし利用可能資源の削除を障害として扱う場合、当該資源上で実行しているプロセスのリカバリが必要になることはいうまでもなく、計算終了までの時間に影響をおよぼす。

5.7.2 資源追加

新たな計算資源の追加に対して、RAFTはその資源を$RPl = []$であるPEとしてpe_tableに追加する。この時点で$RPlist$に起動可能(Ready)のプロセスが存在すれば、RAFTは直ちに資源割り当てと起動を行う必要がある。このためRAFT実行時システムは、新たな計算資源の追加が行われる度にRAFT基本アルゴリズムの起動を行う。

ユーザによってシステムに追加された計算資源は、該当資源からコミュニケーションコストが最も少ない場所に存在するレプリカのpe_tableに登録することが妥当である。なぜならAPRでは、計算の進行が早いレプリカによって実行されたRAFTプロセスの計算結果を使用することによって、全ての計算が完了するまでの時間を短縮することができるからである。このため計算資源の追加によって得られる計算速度の向上は、その資源がどのレプリカに追加されたのかに関わらず、全てのレプリカの計算完了までの時間の短縮につながる。

資源削除/追加に関して、実装においてはグループコミュニケーションによるメンバーシップの変更 (メンバーの削除/追加) を含むが、これらに関しては第 6 章において詳しく述べる。

5.7.3 資源の利用率の変化

本システムで対象としている実行環境 (第 6.1 節) では、全ての計算資源を常に本システムで実行する計算アプリケーションが占有できるという仮定はおいていない。つまり長時間にわたる計算アプリケーションを実行中の一部の PE においては、計算途中でその PE に対してユーザから対話的に他のプロセスの実行を要求される状況もある。

このような状況においては、実行中の RAFT プロセスに対してオペレーティングシステムから割り当てられる CPU 時間が減少し、その結果として RAFT プロセスの実行速度が低下し、一部のモジュールの出力が著しく遅れる可能性がある。このため RAFT は実行する計算アプリケーションの CPU 時間占有率に関して定期的に調査を行ない、低い占有率が一定時間以上継続した場合に exception を発生し、これをパフォーマンス障害として扱う。

第 6 章

実装方法

本章では RAFT アルゴリズムを用いた APR の実装方法について述べる。実装例としては以下のシステムを用いる。

- オペレーティングシステム：Linux 2.2.14
- 関数型プログラミング言語：Objective Caml 3.01
- グループコミュニケーションシステム：Ensemble System 1.00

APR は関数型の計算パラダイムに基づき、計算実行単位間の障害発生に関する独立性の高さを利用して耐故障性を提供している。このことから本実装では、計算を行う実体間の入出力の受け渡しと制御メッセージは全てネットワークを介したメッセージ通信によって行われ、分散共有メモリを利用しないことを前提とする。

本章ではまずはじめに提案するシステム構成の概要を述べる。続いて関数型プログラミング言語 Objective Caml とグループコミュニケーションレイヤ Ensemble System の紹介を行う。本章の後半では、システムの構成要素同士の通信に重点をおいて、実装の詳細について述べる。

6.1 対象アプリケーションと実行環境

6.1.1 対象とするアプリケーション

本研究で対象とするアプリケーションは、大規模で実行が長時間に及ぶ計算アプリケーションである。耐故障性の保証には必ずコストが必要であることから、短時間で実行が完了するアプリケーションに対して耐故障性を持たせるためにユーザがコストを払うことは少ない。

逆に耐故障性を高めることにより、計算の実行が長時間に及ぶアプリケーションに対しても様々な計算資源の利用が可能になる。本研究では以下に示すような一般的な環境に存在する多くの計算資源を、耐故障性を高めることにより大規模な計算アプリケーションで使用する。

6.1.2 実行環境

本システムがターゲットとしている実行環境は図 6.1 に示すような環境である。この環境は粗結合マルチプロセッサ環境、とくに NOW(Network Of Workstations) と呼ばれ、多数の PC やワークステーションが通常の、または高速のネットワークを介して接続されている環境である。

個々の計算要素は 1 つまたはそれ以上の CPU を含み、ローカルにメモリを持つ。計算に用いる全ての PE は仮定する障害に対処するために必要な数のレプリカとしてグループ分けされる。全ての PE はイーサーネットなどの通常のネットワークを介して接続され、1 つのレプリカを形成する。

レプリカ内には 1 つの安定記憶 (Stable Storage) が存在し、妥当性が確認されたデータの保存など、必要な場合に利用される。あるレプリカと他のレプリカは、障害発生に関する独立性を高くするために、ネットワーク上で遠い場所に配置されることがある。

本実装では、以下の仮定をおく。

- 1 つのレプリカの構成要素は、LAN の内部に閉じている
- 1 つのレプリカ中には安定記憶が 1 つ存在する

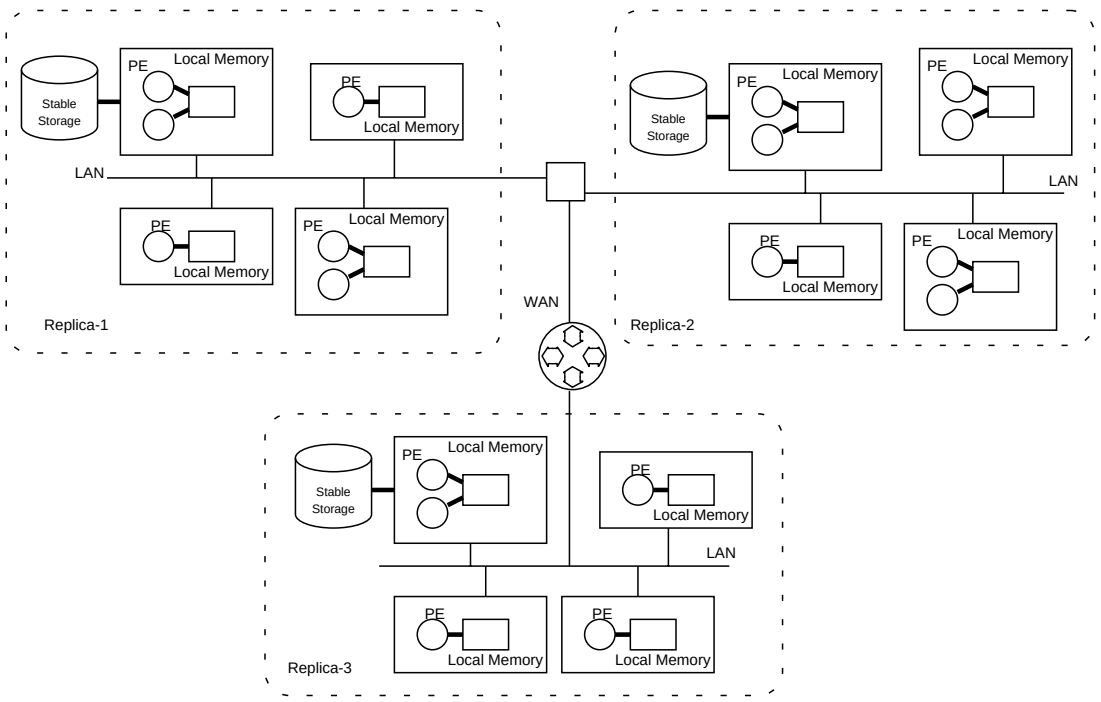

図 6.1: 実行環境

- レプリカ同士は LAN 内または WAN によって接続されている

6.2 Objective Caml System

Objective Caml System[Ler00](以下 OCaml) は関数型言語 ML の一種であり、その実装は 1984 年からフランスの INRIA で開発されている。本実装では以下に示すような理由により OCaml を使用する。

関数型言語の利点

OCaml は関数型プログラミング言語の一種である。OCaml は関数型プログラミング言語の一般的な特徴である参照透過性、高階関数、多相型などの特徴を持つ。FTAG と APR が耐故障性を提供する上で、参照透過性は欠かすことのできない性質である。また FTAG においては関数は親の関数分解の計算結果として得られる値である。モジュール間を流れるメッセージは多相型として定義され、属性、制御メッセージなどを流すことができる。

OCaml が関数型の言語であることから、FTAG 言語で記述された APR アプリケーションをプログラムコンバータ (後述) によって実行プログラムに変換することが容易である。

マルチプラットフォームサポート

OCaml はバイトコードインタプリタとネイティブコンパイラの双方を提供している。OCaml は現在 UNIX,Windows,MacOS などのオペレーティングシステムに移植されている。このため、バイトコードにコンパイルされたアプリケーションは基本的にこれらの全てのオペレーティングシステム上で動作する。この特徴は、分散環境に存在する多くのアイドルな計算資源を利用するために重要である。

データマーシャラ

OCaml は基本ライブラリ内にデータマーシャラを持ち、ユーザ定義の型を持つデータを容易に送受信することができる。これによってユーザ定義の型を持つデータを透過的に安全にネットワーク越しに送受信することができる[1]。

低レベルプリミティブのサポート

OCaml のライブラリは低レベルのオペレーションを Unix システムコールライブラリなどによって数多くサポートする。これは RAFT 資源管理システムによる計算資源の状態の観測や、通信レイヤの実装に有益である。

これらの理由から本実装では、計算機言語として Objective Caml System を使用する。

[1]実際には受信側で型を与える必要があるため、メッセージタイプはライブラリで定義してコンパイル時にリンクすることによって一貫性を保っている。

6.3 Ensemble System

Ensemble System[Hay98](以下 Ensemble) は、Cornell 大学で開発されたグループコミュニケーションシステムである。Ensemble は第 2 章で述べたグループコミュニケーションの一般的な性質に加えて、以下のような特徴を持つ。

プロトコルの柔軟性

Ensemble はグループや通信に関して保証する性質毎に、プロトコルレイヤを部品として提供する。ユーザは Ensemble が提供する様々なのプロトコルレイヤの中から、アプリケーションにおいて保証したい性質に応じて取捨選択を行ない、プロトコルスタックを組み立てることができる[2]。具体的にはプログラマは通信に関して保証したいプロパティの一覧を配列型のデータとして記述するだけでよく、Ensemble System がこれに必要なプロトコルレイヤを選択し、プロトコルスタックを構築する。その後ライブラリをリンクすることによって、必要な機能だけを持ったプロトコルスタックを利用することができる。

ポータビリティ

Ensemble システムは、ほとんどの Unix システムと WindowsNT 上で動作が確認されている。配布は基本的にソースコードで行なわれ、Ensemble システムのソースから make したライブラリをアプリケーションにリンクする方法でユーザは使用する。ライブラリは OCaml のバイトコードライブラリ (libens.cma) とネイティブコードライブラリ (libens.cmxa) の双方で提供される。

OCaml インターフェース

Ensemble システムの実装の大部分は OCaml を用いて行なわれている。このため OCaml からのプログラミングインターフェースが豊富である[3]。

---

[2]現バージョンである The Ensemble System version 1.00 で、34 のプロトコルレイヤが実装されている

[3]OCaml 以外に、C/C++, Java のインターフェースを持つ

6.4 ソフトウェアの構成

ソフトウェアの構成の概観を図6.2に示す。図6.2では図を見やすくするために一つのレプリカのみに注目している。図中の太線矢印は、実行中のコンポーネント間の通信を表している。細線矢印は計算実行開始前のAPRアプリケーションプログラムの入力および実行可能ファイルの配布を表す。

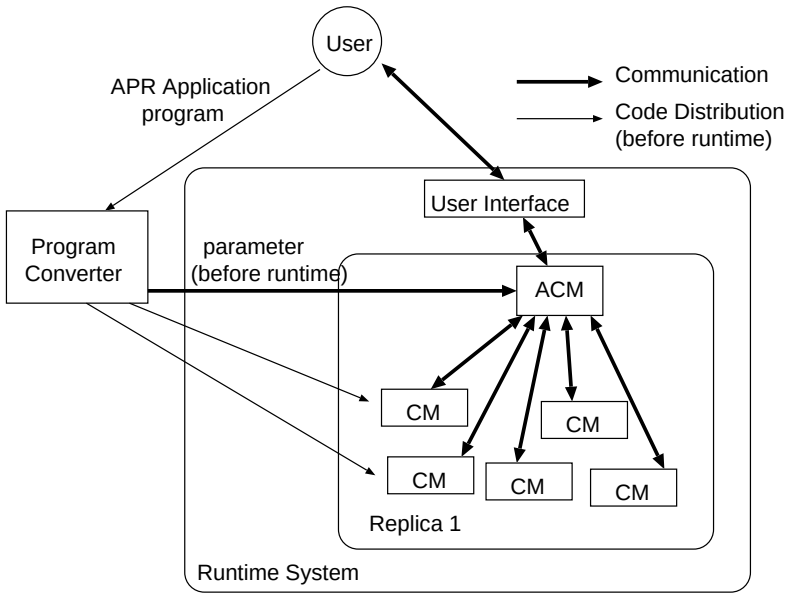

図 6.2: システムの構成

システムに対する入力は以下のものがある。

- FTAG言語で記述されたAPRアプリケーションプログラム
- ルートモジュールへの入力属性
- 利用可能環境に関する情報
- (システム管理に関するコマンド)

一方システムの出力は

- ルートモジュールの出力属性
- (システムの状態に関する情報)

以下に個々の構成要素の機能について述べる。

プログラムコンバータ

ユーザからの入力である APR アプリケーションプログラムのコードの文法チェックを行い、このコードを OCaml コードに変換する。さらに OCaml コードを次節で示す各種実行時ライブラリとリンクして、実行可能ファイルである変換されたモジュール (CM: Converted Module) を生成する。実行時システムの中心となる ACM(後述) に対しては、以下に示すユーザから与えられる起動時の情報を送信する。

- 利用可能資源に関する情報
- 仮定する障害モデル

変換されたモジュール (CM: Converted Module)

変換されたモジュール (以下 CM) はプログラムコンバータによって出力されたアプリケーションプログラム本体の実行可能ファイルである。全ての CM は APR アプリケーションの入力プログラムとしてユーザによって記述された全ての関数の定義を含んでいる。CM は計算実行開始前に全ての利用可能な PE1 つに対して 1 つ配布されて起動される

起動された CM のプロセスは、2 つのスレッドとして動作を行う。

第 1 のスレッドは**コミュニケーションスレッド**であり、グループコミュニケーションのエンドポイントとしての機能を継続して実行する。CM のコミュニケーションスレッドは、識別子 $pe -_i d$ を内部属性として保持し、具体的には以下の操作を行う。

- 起動要求の受信
- 属性の送受信
- 資源情報の送信

第 2 のスレッドは**アプリケーションスレッド**であり、RAFT プロセスとして起動要求があったときのみ動作する。アプリケーションスレッドは RAFT プロセスのコードを内部に持ち、以下に示す入出力を行う RAFT プロセスとして動作する。

- RAFT 分解プロセス

  (Mname * in_attr) → (Cname * Dlist)

- RAFT 合成プロセス

  (Mname * in_attr * Cname * c_attr) → out_attr

つまり RAFT プロセスの起動要求はコミュニケーションスレッドに対して行われ、RAFT プロセスの計算はアプリケーションスレッドによって行われる。アプリケーションスレッドの計算が完了すると、コミュニケーションスレッドにより出力属性が送信される。

APR 計算マネージャ(ACM: APR Computation Manager)

APR 計算マネージャ(以下 ACM) は APR および RAFT の機能を実装した実行時システム本体である。実際には次節で述べる FTAG,APR,RAFT,Ensemble の各種ライブラリを用いて実行時システムのほぼ全ての機能を提供する一つの実行可能ファイルである。ACM は計算実行開始前に全てのレプリカにおけるリーダ PE(第 6.6 節) 上に転送されて起動される。

起動された ACM では、2 つのスレッドが動作する。

第 1 のスレッドは**コミュニケーションスレッド**であり、グループコミュニケーションにおけるエンドポイントとして継続して動作する。ACM のコミュニケーションスレッドは、以下の操作を行う。

- 起動要求の送信
- レプリカ内の CM との属性の送受信
- 他レプリカとの分解結果および出力属性結果の送受信

第 2 のスレッドは**マネージメントスレッド**であり、実行時システムの機能を実装しているスレッドであり、継続して以下の動作を行う。

- APR タスクスケジューリング
- RAFT プロセス起動管理

- 安定記憶の管理

ユーザインターフェース

ユーザーインターフェースは実行前にはユーザからの APR アプリケーションの入力と、ルートモジュールの入力属性の入力に使用される。計算実行中はシステムの状態の監視、PE の削除要求の入力、PE の追加要求の入力のみに用いるのみである

ユーザーインターフェースは、計算実行中には実行時システムとアプリケーションの実行に必要な情報を送受信することは無い[4]。このためユーザインターフェースは、システムから接続を解除したり再接続をしたりすることが可能である。

6.5 安定記憶

レプリカ内に必ず 1 つ存在する ACM は、安定記憶 (図 6.3) を保持する。

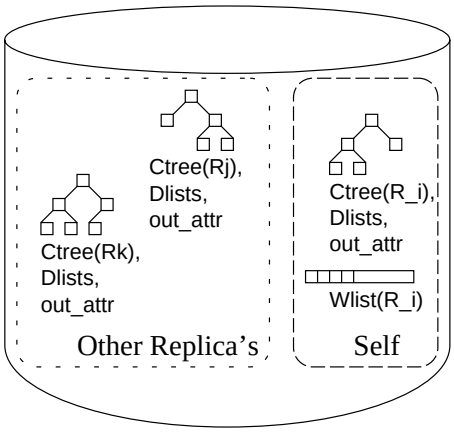

図 6.3: ACM 内の安定記憶

レプリカ R_i の安定記憶には以下の情報が保存される。

- レプリカ R_i の

[4]動作している資源管理システムに対してユーザが任意の時刻にジョブを投入するという、いわゆるインタラクティブなシステムのユーザインターフェースではない。

計算木構造 $Ctree$

関数分解リスト $Dlist_{M\_x}$

出力属性 $out\_attr_{M\_x}$

起動要求リスト $Wlist, RPlist$

- レプリカ R_i 以外の全てのレプリカの

  計算木構造 Ctree

  関数分解リスト $Dlist$

  出力属性 out_attr

全てのレプリカがこれらの情報を自レプリカ内の安定記憶に保持することにより、他のレプリカの属性の参照が必要になった場合においても、通信の発生を抑制できる。

6.6 通信の実装

システム内で行なわれる通信を図 6.4 に示すように 2 種類に大別する。

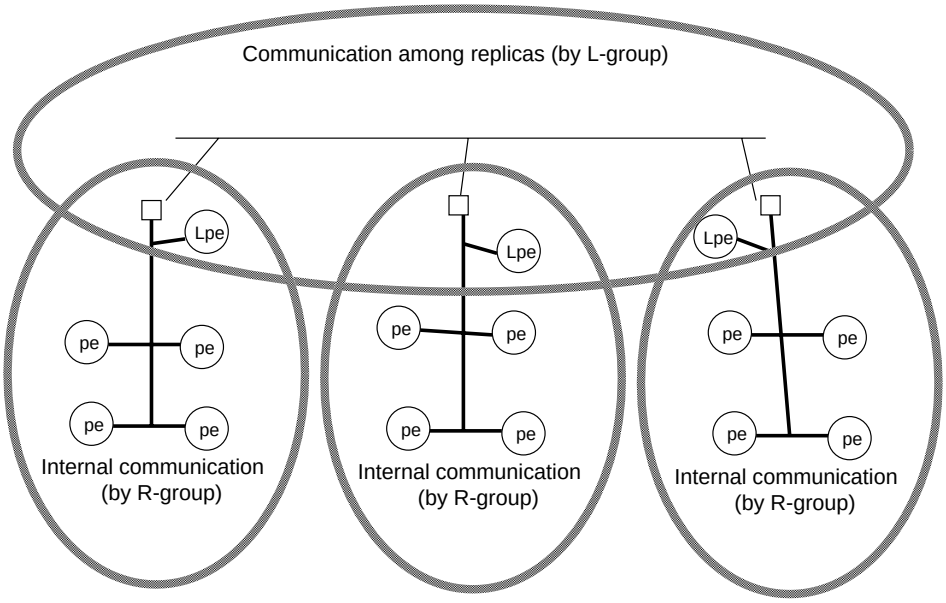

図 6.4: グループの階層

第 1 の通信を**レプリカ内通信**と呼ぶ。これは同一レプリカ内に存在する ACM と複数の CM プロセスの間の通信である。ACM, CM とも、コンポーネント内で実行しているコミュニケーションスレッドが通信を行う。

第 2 の通信を**レプリカ間通信**と呼ぶ。これは複数のレプリカ間における通信である。

以降の節でに述べるように、レプリカ内通信とレプリカ間通信は多くの部分で異なる性質を要求されるため、本実装ではグループの概念を階層化した通信アーキテクチャを採用する。

システム構成要素の起動とグループの形成に関して図 6.5 に示す。

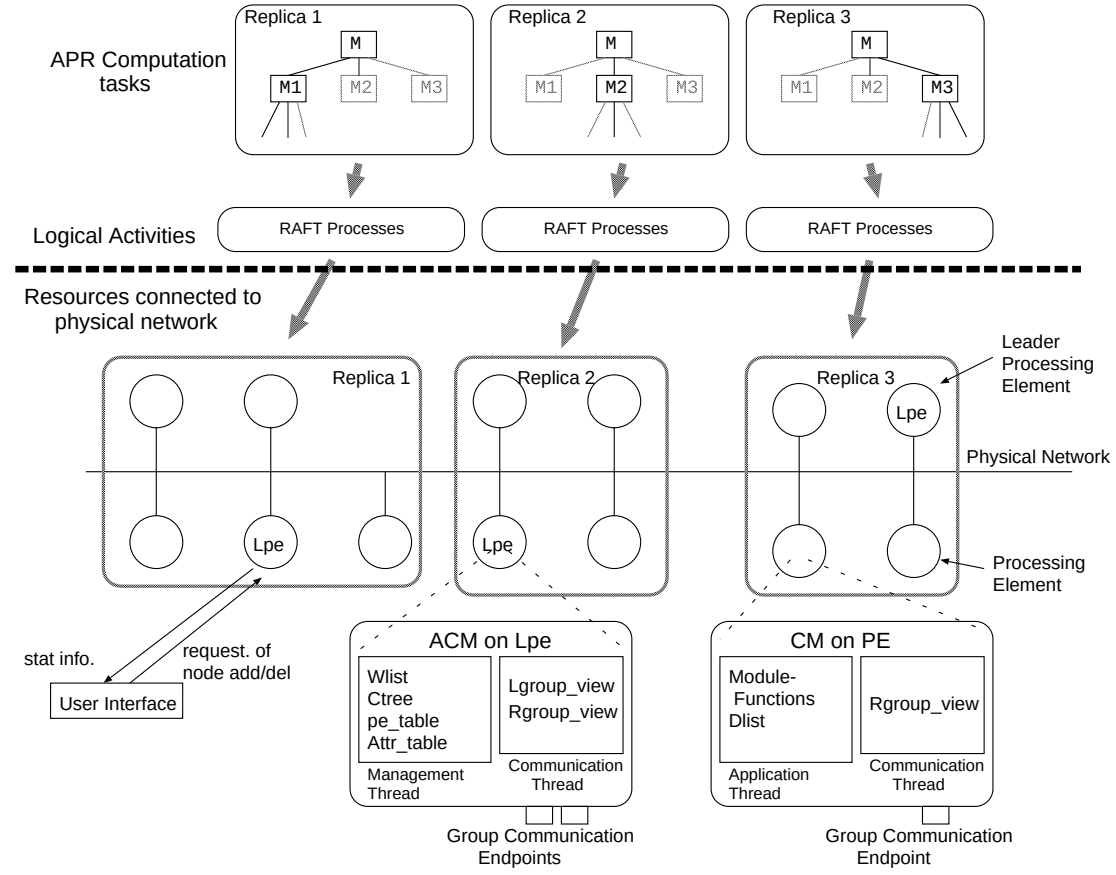

図 6.5: RAFT プロセスおよびスレッドの起動とグループの形成

実行開始時点で利用可能な全ての計算資源において CM が起動される。CM 内で起動されたコミュニケーションスレッドは、同一レプリカ内にある全ての CM のコミュニケーションスレッドで R-Group を形成する。

出力属性の送受信はグループによる通信 (Cast) ではなく、CM のコミュニケーションスレッドと実行時システムの間の 1 対 1 の通信 (Send) によって行われる。実行時システムは R-Group への Cast によって、以下に示すいくつかの制御メッセージの送受信を行なう。

- 資源状態の問い合わせ
- ACMS すなわち計算ポリシーの変更通知 (アルゴリズム 2)

グループ内の一つのメンバーをリーダー PE(Lpe) と呼び、Lpe においては実行時システム ACM が動作する。レプリカ数と同じ数だけ起動する Lpe 間ではさらにリーダーグループ L-Group を形成する。

L-Group における通信がレプリカ間通信である。レプリカ間通信は主にグループ内ブロードキャストである Cast を用いて実装される。以下にレプリカ内通信とレプリカ間通信の各々に関して詳細に述べる。

6.6.1 レプリカ内通信

論理的には関数型の計算モデルにおいては、属性は計算木中の親モジュールとサブモジュールの間で受け渡しが行なわれる。しかし APR および RAFT では出力属性および関数分解結果の他のレプリカへの送信を行なうために、CM において計算された全ての出力はレプリカ内の実行時システム ACM に送信され、ACM 内の Attr_table(図 6.5) に格納される。この通信は R-group に属する CM と ACM の Send オペレーションによって実装される。

関数の出力属性以外にも、ACM から CM への計算起動操作や CM から ACM への計算完了通知などがある。以下に全てのレプリカ内通信を列挙する。

- メンバーシップの管理 (R-group)
- 起動関数名[5]と入力属性 (ACM→CM)
- 関数名と出力属性または関数分解結果 (CM→ACM)

[5] リカバリ時にはリカバリ属性を含む。

- PF 属性のアップデート (CM→ACM)
- PF 属性[6]のリクエスト (ACM→R-group)

レプリカ内通信においては、ほとんどの通信が Send オペレーションによる 1 対 1 通信である。Cast 通信は、メンバーシップの管理とユーザインターフェースからのイベントの配送のみに用いる。

PF 属性のアップデートは基本的に、定義 8 で述べた通り CM から行なわれる[7]。

ACM と CM の間の通信に用いられるメッセージの具体的なデータ型およびオペレーションは第 6.8 節に定義する。

APR および RAFT では基本的には通信は正しく行われることを暗に仮定している[8]。このためこれらの通信はリライアブルである必要がある。つまり、送信したメッセージが必ず受信され、すでに送信されたメッセージのみが受信されなければならない。あるいは通信が正しく行われなかった場合には必ずその障害を検出して対処する必要がある。本実装ではグループコミュニケーションシステムを使うことによってリライアブルな通信を実装する。

6.6.2 レプリカ間通信

レプリカ内通信では主に Send オペレーションによる 1 対 1 通信が主体であったが、レプリカ間通信では Cast オペレーションによるグループ内ブロードキャストを多く用いる。

レプリカ間通信は以下の状況で用いられる。

- メンバーシップの管理 (L-group)
- 分解結果および出力属性の配布 (ACM→L-group)
- バリュー障害検出メッセージ (ACM→L-group)

[6]各 CM が内部属性として保持する計算資源の状態を示す変数。第 5.4 節参照

[7]PF 属性のリクエストメッセージは、ユーザインターフェースからの状態取得要求があった場合のみ起動される。

[8]実装のレベルにおいてはリンク障害を考慮している。リンク障害については第 6.7.2 節で述べる。

- バリュー障害リカバリ要求の送信 (ACM→ACM)
- バリュー障害発生時の $\mathrm{Total}(S_{cpu})$ の送信 (ACM→ACM)

上記で前者 3 つはグループに対する通信であり、後者 2 つの通信は 1 対 1 通信である。

レプリカ内通信と同様、ここでもリライアブルな通信を行う必要があるため、グループコミュニケーションの `Send` および `Cast` を用いた実装を行う。

L-group 内の通信は遅延の大きな通信を含む場合があるため、L-group のコミュニケーションスレッドで定義するタイムアウトは起動時には大きく設定され、L-group 構成直後に調整される。タイムアウトの機構を含めたリンク障害の扱いに関しては次節において詳しく述べる。

6.7 障害の扱い

6.7.1 レプリカ内における障害検出

疎結合分散環境環境においては、ローカルエリアのネットワークの内部におけるノードのクラッシュ障害の検出は、watchdog タイマによる定期的な返信要求に対する返信が一定時間以上起こらないことによって検出するという方法が一般的に受け入れられている [Jal94]。第 2.3 節で述べたようにこの障害検出は、誤って障害を検出してしまう可能性のある障害検出システムの存在を認めることにより実現される。

本実装においてもこれと同様の仮定を行なう。レプリカ内にこにおいて、コミュニケーションスレッド間で一定時間以上ハートビートによって動作が確認できない PE はクラッシュ障害を起こしたとする。

レプリカ内における PE のクラッシュ障害の検出はグループコミュニケーションレイヤを用いることによって実装する。障害の検出は主に Ensemble ライブラリが提供する `Suspect` プロトコルレイヤによって行なわれる。`Suspect` プロトコルレイヤは定期的に全てのグループメンバーに対して $suspect\_sweep$ 時間毎にハートビートメッセージを送信する。$suspect\_max\_idle$ 時間が経過してリプライを受信できないメンバーが存在した場合はグループコーディネータにより $block\_view$ コー

ルバックメソッドが全てのメンバーで起動され、全てのメンバーは送信を停止する。その後コーディネータからの $install\_view$ コールバックメソッドの呼び出しにより、新たな $view$ が全てのメンバー内で設定される。

$install\_view$ によって Ensemble は全てのメンバーに対して新たな $rank$(個々のメンバーの識別子) を与える。ACM 上のコミュニケーションスレッドはこの $install\_view$ コールバックメソッドが起動すると、ACM が管理するプロセスの識別子である pe_id とグループメンバの識別子である $rank$ の対応付けを管理するテーブル `member_table` を更新する。

view change が発生すると ACM は直ちに `member_table` のバックアップである `member_table.0` を作成し、新たな rank と pe_table の対応付けに `member_table` をアップデートする。ACM は `member_table.0` を用いて障害が発生した PE を特定し、APR への再実行の要求と pe_table からの当該エントリの削除を行なう。

バリュー障害の検出とリカバリに関しては、第 5.6 節で述べた方法によって、全て ACM 実行時システム内で行なわれる。

6.7.2 レプリカ間通信におけるリンク障害の扱い

APR 複製技術はクラッシュとバリューの 2 種類の障害に対処するための技術であるが、実際の疎結合分散環境においては、広域ネットワークにおいて過渡的な障害が発生しないという仮定が受け入れられる場面は非常に少ない。このため本実装では過渡的なリンク障害に対処するための補助的な手段をユーザに提供する。

レプリカ間通信は多くの場合、広域で過渡的な障害が発生しやすいネットワークを越えて行なわれることが多く、さらにそのネットワーク上の通信の遅延は時刻によって変化する。また、レプリカ全体の障害は計算の進行に著しく影響するため、特に「障害発見までに必要となる時間の短縮」よりも「障害の正しい発見」が重要である。このためレプリカ間通信では、時刻によって変化する通信遅延の考慮とリンク障害モデルの仮定の双方が必須である。

この問題に実装レベルでできる限り対処するために、レプリカ間通信を行なう ACM のコミュニケーションスレッドは、グループメンバーへの通信のレイテンシに関して定期的な調査を行なう。これにより `Suspect` プロトコルレイヤの

$suspect\_max\_idle$ パラメータをこの遅延 $LT_{\mathsf{L-group}}$ の関数として定義し、定期的に更新する。

$$suspect\_max\_idle(t) = f_{suspect}(LT_{\mathsf{L-group}})$$

ここで $LT_{\mathsf{L-group}} \neq \infty$
関数 $f_{suspect}$ はレプリカが使用する広域ネットワークの性質を考慮し、障害の検出の正しさと障害検出に必要な時間のトレードオフを考慮して、使用する環境毎に慎重に決定する必要がある。

あるレプリカの実行時システムを実行しているPEがクラッシュしてview_changeが起きたときは、基本的にはレプリカ内におけるPEのクラッシュの場合と同様に扱う。それに加え、リンク障害が比較的起こりやすいレプリカ間の通信においては、リンクが正常状態に復帰してグループの再構成が行われる場合がある。この場合もレプリカ内の時と同様、ACMは、rankと pe_table の対応付けを更新する。同時に、再構成によってマージしたレプリカを含む全てのレプリカは、自レプリカによって計算を行った全てのデータに関してL-groupにマルチキャストを行う。これによりグループの再構成の後における安定記憶内部のデータの一貫性を保つ。

6.8 実行時システムを構成するライブラリ群

ACM, CMは多くの共有するデータ型を持ち、使用するインターフェースを相互に提供する。本システムはデータ型、メッセージ、インターフェースに注目し、各々の機能を以下に示す複数のライブラリとして分離して記述する。

- FTAG計算木管理ライブラリ
- FTAGメッセージライブラリ
- APR実行制御ライブラリ
- APR耐故障ライブラリ
- RAFT資源情報収集ライブラリ

- RAFT 資源割り当てライブラリ
- Ensemble グループコミュニケーションライブラリ

CM, ACM およびインターフェースはこれらのライブラリをリンクした結果として得られる実行可能ファイルである。以下に個々のライブラリが提供する機能を示す。

FTAG 計算木管理ライブラリ

FTAG 計算木管理ライブラリは、FTAG で定義されている計算中の計算木の構造である Ctree データ構造を持ち、Ctree に対する各種の操作、すなわち関数分解によって発生する計算木の成長や、障害の発生によって発生する部分木の削除を行うインターフェースを提供する。関数分解結果は本ライブラリが提供する $Dlist$ 内に格納する。FTAG 計算木管理ライブラリは ACM、APR 実行制御ライブラリによって使用される。

FTAG メッセージライブラリ

FTAG 計算モデルに示されるモジュール間の属性の送受信に必要なメッセージ型の定義、およびメッセージの送受信および送受信に必要なデータのマーシャリングを行うための操作を提供する。実際にはこのメッセージには、FTAG における属性の送受信以外の制御メッセージも含む。Ftagmsg 型は以下のように定義される。

```
type ftagmsg =
{
        mtype:        mtype_t ;       (* type of ftagmsg *)
        fname:        fname_t ;       (* module name in the tree *)
        attr:         attr_t list ;   (* a list of attributes *)
        optarg:       optarg_t        (* optional *)
}
```

メッセージの作成、メッセージからの特定のフィールドのとりだしなどは以下のオペレーションによって実装される。

```
val make_msg : char -> string -> attr_t list -> ftagmsg
val get_mtype : ftagmsg -> mtype_t
val get_fname : ftagmsg -> fname_t
val get_attr : ftagmsg -> attr_t list
val set_mtype : ftagmsg -> mtype_t -> ftagmsg
val set_fname : ftagmsg -> fname_t -> ftagmsg
val set_attr : ftagmsg -> attr_t list -> ftagmsg
val add_attr : ftagmsg -> attr_t list -> ftagmsg
val set_opt : ftagmsg -> optarg_t -> ftagmsg
val get_opt : ftagmsg -> optarg_t

val printout : ftagmsg -> unit
val printoutm : string -> unit
```

メッセージをネットワークを介して通信するために、本システムではEnsembleグループコミュニケーションレイヤを用いる。本ライブラリは関数間を送受信する任意のデータをEnsembleで扱えることができるように変換するために、以下に示すデータマーシャリングの操作を提供する[9]。

```
val unmarshal   : string -> ftagmsg
val marshal : ftagmsg -> string
```

アプリケーションスレッド中の任意のデータ型[10]は、上記関数によってマーシャリングされ、Ensembleのメッセージとして送信される。

FTAGメッセージライブラリは、CM, ACMによって使用される。

APR実行制御ライブラリ

APRの実行順序に関するアルゴリズムを実装する。実行順序を決定するために本ライブラリは、計算木に関する情報をFTAG計算木管理ライブラリが提供するオペレーションによって獲得する。内部には$Wlist$を保持し、後述するRAFTライブラリのオペレーションを呼び出す。

[9] Ensembleシステムは OCamlの pervasive ライブラリ (標準で読み込まれるライブラリ) で提供されている型のデータを安全に送受信することができる

[10] Ftagmsg ライブラリで定義されているもの

APR 耐故障ライブラリ

APR 耐故障ライブラリは全てのレプリカから受信した出力属性を保存する安定記憶 (Stable Storage) を実装し、$attr\_lists$ を保持する。r_list_n はシステム起動時に安定記憶内にレプリカの数 n と同数作成され、出力属性が書き込まれる。

また本ライブラリは仮定する障害モデルに関する情報を保持する。バリュー障害仮定時は、r_lists_i 内に属性を書き込む際に、妥当性試験を行なう。妥当性試験により障害の発生が確認された場合は、L-group の通信により、他レプリカへのリカバリ起動要求操作を実装する。本ライブラリは ACM によって利用される。

RAFT 資源情報収集ライブラリ

本ライブラリはオペレーティングシステムから PF の構成に必要な情報を取り出す。本ライブラリの実装は本システムの実装中で唯一、オペレーティングシステムに依存する部分である。現在実装を行なっている linux-2.2.14 においては、/proc ファイルシステムを用いてほとんどの情報を獲得している。本ライブラリは CM によって用いられる。

RAFT 資源割り当てライブラリ

RAFT 資源割り当てライブラリは、データ構造 pe_table を管理する。本ライブラリは大別して 2 つの機能を提供する。第 1 に、ACM が利用可能資源に関する情報を CM から収集し、管理するための操作を提供する。本ライブラリにはオペレーティングシステムに依存するコードは含まれない。pe_table の持つ情報のアップデートは、CM からの資源状態変化通知の受信によって行なわれる。

第 2 に、本ライブラリは APR 実行順序制御ライブラリからの起動要求リクエストに対して、対応する RAFT プロセスのスレッドを選びだし、利用計算資源の決定と起動を行う。

本ライブラリは CM,ACM によって用いられる。

プロセス実行制御ライブラリ

アプリケーション RAFT プロセスの起動と終了に関する操作を定義する。RAFT プロセスは実際には CM のアプリケーションスレッドの起動という形で実装される。本ライブラリは CM によって用いられる。

Ensemble グループコミュニケーションライブラリ

Ensemble System は、グループコミュニケーションのためのオペレーションを提供するライブラリを提供している。

Ensemble System が提供するライブラリを用いて、FTAG メッセージライブラリによってマーシャリングされたデータの送受信を実装する。Cast(m) オペレーションは、グループに属している全てのメンバーに対してメッセージ m をキャストする。Send(d, m) オペレーションによって、グループ内のメンバー d に対して、メッセージ m をユニキャストする。

また ACM と CM のコミュニケーションスレッドが実装する Ensemble のコールバックメソッドの定義もこのライブラリに含まれる。

本ライブラリは、CM, ACM のコミュニケーションスレッドによって使用される。

6.9 評価方法

ここまでで、RAFT 資源管理システムおよび APR システムに関する実装を具体的に提案した。この実装の性能は、多くの点でアプリケーションに依存する部分が考えられる。定量的な評価を行なう項目としては、以下に示すような事柄が考えられる。

6.9.1 資源利用の公平性

RAFT は関数型計算における関数分解の段階から属性合成の段階まで、一貫して資源割り利用に関する制御を行うアルゴリズムである。また関数という論理的な実体を RAFT プロセスというより細粒度の単位に分割しているため、負荷分散は比較的行われやすくなっているはずである。

このことを検証するため、レプリカ中の PE の負荷の時刻的推移に関して検証を行なう。

6.9.2 リカバリ時間

バリュー障害からのリカバリ時間に関して、優先度を導入した RAFT のリカバリと通常のリカバリ方式の各々に関して、リカバリ完了までの時間をを比較する。つまり優先度 Pri_r の付加をせず、$RPl = \phi$ の条件を通常通り適用した場合のリカバリに要する時間を調べ、本論文で述べた手法と比較する。

6.9.3 プロセス起動コストに関する評価

非常に簡単な属性関係式を持ち、しかも再帰呼び出しを繰り返すようなアプリケーションを入力する。つまりアプリケーションの計算時間に対して再帰回数が非常に多いアプリケーションを入力して、入力プログラムにはアサーションを挿入し、計算の粒度制御の有効性を確かめる。また与えるアプリケーションプログラムで再帰計算の基底段階を変更することによってどの程度の性能改善が得られるかを評価する。

6.9.4 資源状態観測時期に関する評価

資源状態を表すPFは、一定時間T_{pf}毎にCMが測定と算出を行ない、$Thr_{pf}\%$以上変化した場合にACMに資源情報を送信する。T_{pf}, Thr_{pf}を変化させたときの、このアップデートに必要な計算およびコミュニケーションのコストを測定し、いくつかの場合における資源割り当ての公平性について議論する。

6.9.5 PE数に関するスケーラビリティ

レプリカ内のPE数に関する、実行時システムの計算が消費するCPU時間、コミュニケーションコストについて検証する。また、アプリケーションによって定義される木構造の形とPE数を変化させ、資源の利用率およびスケーラビリティを評価する。

第 7章

まとめと今後の課題

本論文ではまずはじめに、APR 計算起動アルゴリズムの定式化を行ない、これによって APR の論理的な動作とそれらの活動が消費する資源に関する分析を詳細に行なった。この分析結果を基に、APR のタスク起動要求に対する資源割り当てアルゴリズム RAFT を提案した。

RAFT ではまずはじめに、APR の論理的なタスクを分散環境における実際のプロセスに対してマッピングを行なうために、より細粒度の RAFT プロセス (RAFT 分解プロセス、合成プロセス) を定義した。

次に RAFT 資源管理システムを提案した。RAFT 資源管理システムは、分散環境に存在する利用可能計算資源の情報を管理すると同時に、APR の計算起動アルゴリズムによって生成される RAFT プロセスの計算資源に対して割り当てを行うものであった。RAFT は、APR が耐故障性を保証するために用いている特徴、すなわち参照透過性とそれにより得られる非常に自然なロールバックポイントという論理的な特徴を、実装においても保証することを確認した。

また APR 実行時システムの具体的な実装方法を提案した。実装ではグループコミュニケーションを用いることによって、レプリカ内のクラッシュ障害への対処、レプリカ間における出力属性の共有によるバリュー障害への対処を実現するアーキテクチャを提案した。

これらの結果、耐故障性の提供と並列計算による計算時間の短縮を行なう複製技術である APR の分散環境における実装の持つ特徴が明らかになり、実装のための指針が明確になった。

また本研究において、計算アプリケーションのプログラミングから分散環境における実装までの全体像を明らかにした。これにより、「プログラマは関数型のプログラミング言語であるFTAGによってアプリケーションを記述するだけで、耐故障性と並列計算による計算時間の短縮、そして安価な計算資源の利用可能性という3つの利益を得ることができる。」ということを、FTAG言語による定義から実装までの全てのレベルにおいて明確にした。

疎結合分散環境において多くの計算資源を利用して大規模な計算を行うという意味において、関連研究としてGrid Computingと呼ばれる分野がある[IFT01]。代表的な実装としては、Globusプロジェクトにおける GUSTOがある[FK99]。GUSTOは、ネットワークの物理層から資源管理、メッセージ送受信のためのプロトコルに至って、それぞれにおける既存の技術をローカルサービスとして位置づけ、これらよりも1段階高いレイヤでシステム全体をコーディネートする。この手法は、それぞれのローカルエリアのサイトにおいて既存の実装を利用することができるという面で魅力的である。しかしながら本研究において提案したRAFTシステムのように、アプリケーションの持つ性質を利用することによって、プログラマに対して透過的に耐故障性の提供や並列計算の実現を行うことが難しい。

APR複製技術はそのアイデアが提案されてからまだまもない技術であり、現在のところアプリケーションへの適用例が存在しない。実際の科学技術計算などの長時間にわたるアプリケーションを入力として実行し、定量的な評価および改良を行なうことが今後の課題である。

謝辞

本研究を行なうに当たり終始御指導を賜わった片山卓也教授に深謝致します。研究を進めるにあたりそれぞれの分野から有益な助言を下さった、北陸先端科学技術大学院大学の日比野靖教授、篠田陽一教授、東京工業大学の渡部卓雄助教授、早稲田大学の中島達夫助教授に感謝致します。

また、日頃から有益な御助言をいただき多面に渡って励ましていただいた、東京工業大学の鈴木正人助教授、片山研究室 Adel Cherif 助手に感謝致します。

ソフトウェア基礎講座の権藤克彦助教授、伊藤恵助手、青木利晃助手、ならびに片山・権藤研究室の諸兄には、ゼミはもちろん、日常の雑談においても非常に広い範囲に渡り情報交換や議論の相手をして頂いた。ここに感謝致します。

最後に、私を支えてくれた妻京子に感謝します。

参考文献

[AD76] Peter A. Alsberg and John D. Day. A principle for resilient sharing of distributed resources. In *In Proceedings of the Second International Conference on Software Engineering*, pages 627–644, 1976.

[AMSM92] D. Agarwal, P. Melliar-Smith, and L. Moser. Totem: A protocol for message ordering in a wide-area network, 1992.

[Avi85] A. Avizienis. The N-Version approach to fault-tolerant software. *IEEE Transactions on Software Engineering*, SE-11(12):1491–1501, 1985.

[Bir86] K. Birman. Isis: A system for fault tolerant distributed computing, 1986.

[Bir93] Kenneth P. Birman. The process group approach to reliable distributed computing. *Communications of the ACM*, 36(12):37–53, 1993.

[BM93] Ozalp Babaoglu and Keith Marzullo. Consistent global states of distributed systems: Fundamental concepts and mechanisms. In Sape Mullender, editor, *Distributed Systems*, chapter 4, pages 55–96. ACM Press, New York, second edition, 1993.

[Che98] Adel Cherif. *Replication For Fault-Tolerant Software Using a Functional and Attribute Grammar Based Computation Model.* PhD thesis, Japan Advanced Institute of Science and Technology, March 1998.

[CK98] A. Cherif and T. Katayama. Replica management for fault tolerant systems. *IEEE Micro*, 18(5):54–65, 1998.

[CT96] Tushar Deepak Chandra and Sam Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2):225–267, March 1996.

[CZ85] David R. Cheriton and Willy Zwaenepoel. Distributed process groups in the V kernel. *ACM Transactions on Computer Systems*, 3(2):77–107, May 1985.

[DM96] Danny Dolev and Dalia Malki. The Transis approach to high availability cluster communication. *Communications of the ACM*, 39(4):64–70, April 1996.

[FK99] Ian Foster and Carl Kesselman. The Globus Project: A status report. In *Proceedings of Heterogeneous Computing Workshop*, pages 4–18, 1999.

[FLP85] Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374–382, April 1985.

[F.S90] F.Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys*, 22(4):299–319, Dec 1990.

[Hay98] Mark G. Hayden. *The Ensemble System*. PhD thesis, Cornell University, Graduate School of Cornell University, Mar 1998.

[HT93] Vassos Hadzilacos and Sam Toueg. Fault-tolerant broadcasts and related problems. In Sape Mullender, editor, *Distributed Systems*, chapter 5, pages 97–145. ACM Press, New York, second edition, 1993.

[IFT01] Carl Kesselman Ian Foster and Steven Tuecke. The Anatomy of the Grid : Enabling scalable virtual organizations. In *to be appeared in the Proceedings of the First IEEE/ACM Intl. Symposium on Cluster Computing and the Grid – CCGrid 2001*, 2001.

[Jal94] Pankaj Jalote. *Fault Tolerance in Distributed Systems.* Prentice Hall, 1994.

[J.H99] J.H.Reppy. *Concurrent Programming in ML.* Cambridge University Press, 1999.

[Knu73] D. E. Knuth. *The Art of Computer Programming. Volume 3: Sorting and Searching.* Addison-Wesley, 1973.

[Kui89] Matthijs F. Kuiper. *Parallel Attribute Evaluation.* PhD thesis, University of Utrecht, Padualaan 14, P.O. Box 80.089, Utrecht, The Netherlands, November 1989.

[Lam78] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7), July 1978.

[Lap92] Jean-Claude Laprie, editor. *Dependability: Basic Concepts and Terminology*, pages 210–245. IFIP WG 10.4 Dependable Computing and Fault Tolerance. Springer-Verlag Wien New York, 1992.

[Ler00] Xavier Leroy. *The Objective Caml System release 3.00.* INRIA, France, April 2000. http://caml.inria.fr/ocaml/.

[LK00] Sandeep Lodha and Ajay Kshemkalyani. A fair distributed mutual exclusion algorithm. *IEEE Trans. on Parallel and Distributed Systems*, 11(6), June 2000.

[Lyu95] Michael R. Lyu, editor. *preface*, page xi. Trends in Software. John Wiley & Sons Ltd., August 1995.

[PvE93] Rinus Plasmeijer and Marko van Eekelen. *Functional Programming and Parallel Graph Rewriting.* Addison-Wesley Publishing, June 1993.

[Ran75] B. Randell. System structure for software fault tolerance. *IEEE Transactions on Software Engineering*, SE-1(2):220–232, Jun 1975.

[Sch93] Fred B. Schneider. What good are models and what models are good? In Sape Mullender, editor, *Distributed Systems*, chapter 2, pages 17–26. ACM Press, New York, second edition, 1993.

[Ser99] Jocelyn Serot. Explicit parallelism. In Kevin Hammond and Greg Michaelson, editors, *Research Directions in Parallel Functional Programming*, chapter 18, pages 379–396. Springer-Verlag, Berlin, 1999.

[SKS94] M. Suzuki, T. Katayama, and R. D. Schlichting. Implementing fault tolerance with an attribute and function based model. In *Proceedings of the Twenty-forth Annual Internatiocal Symposium on Fault-Tolerant Computing*, pages 244–253, Austin, Texas, June 1994.

[VFN95] author Victor F. Nicola. Checkpointing and the modeling of program execution time. In Michael R. Lyu, editor, *Software Fault Tolerance*, Trends in Software, chapter 7. John Wiley & Sons Ltd., August 1995.

[vRBM96] Robbert van Renesse, Ken Birman, and Silvano Maffeis. HORUS: A flexible group communication system. *Communications of the ACM*, 39(4):76–83, April 1996.

本研究に関する発表論文

[1] 豊島真澄, 鈴木正人, 片山卓也: “耐故障性ソフトウェアのための計算モデル FTAG の実装と評価”, 信学技法 Vol.98, No.23 pp.39-46, 1998 年 4 月.

[2] Masumi Toyoshima, Masato Suzuki, Takuya Katayama: “Using a functional language for designing fault tolerant parallel and distributed software”, Proceedings of 4th Intl. Conference on Information Systems, Analysis and Synthesis (ISAS'98), pp.249-256, July 1998, Orlando, FL, USA.

[3] 豊島真澄, 鈴木正人, 片山卓也: “疎結合分散環境におけるプロセッサ割り当てに関する考察”, 信学技法 Vol.98, No.488 pp.17-23, 1998 年 12 月.

[4] Masumi Toyoshima, Adel Cherif, Masato Suzuki, Takuya Katayama: “Design and Implementation of Fault Tolerant Parallel Software in a Distributed Environment using a Functional Language”, In Proceedings of the IEEE Workshop on Fault Tolerant Parallel and Distributed Systems (FTPDS'99), pp.153-163, April 1999, Puerto-Rico, USA.

[5] 豊島真澄、アデルシェリフ、鈴木正人、片山卓也: “疎結合分散環境における耐故障ソフトウェアの通信の設計”, 信学技法 Vol.99, No.345 pp.25-32, 1999 年 10 月.

[6] Masumi Toyoshima, Adel Cherif, Takuya Katayama: “Improving the Efficiency of Replication for Highly Reliable Systems”, FastAbstracts Proceedings of 10th International Symposium on Software Reliability Engineering (ISSRE'99), pp.27–28, Nov.1999, Boca Raton, FL, USA.

[7] Masumi Toyoshima, Adel Cherif, Masato Suzuki and Takuya Katayama: "Implementing Fault Tolerant Software in Distributed Environment", Chapter 15th in book "Dependable Network Computing", pp341–357, Dimiter R. Avreskey(Ed.), Cluwer Academic Publishers, Jan.2000.

[8] Adel Cherif, Masumi Toyoshima, Takuya Katayama: "Adaptive Computation Management for Fault Tolerance", in Proceedings of 6th Intl. Conference on Information Systems, Analysis and Synthesis, July 2000