動的適応可能な分散システムアーキテクチャ

孫 静涛[1] 佐藤 一郎[1]

概要：近年、インターネットの普及と IoT(Internet of Things) や M2M(Machine to Machine) 技術の発展とともない、動的に適応性可能な分散システムの研究が大きく注目されている。しかし、既存の分散システムはいずれも固定なアーキテクチャであり、頻繁に変化しているシステムのニーズとアプリケーションのリクワイアメントに応じで、分散システムは最適に対応しているとは限らない。そこで、本提案手法の鍵となるアイディアは、コンピュータ上に実行中のソフトウェアの実行位置を別のコンピュータに移動することで、分散システムのアーキテクチャは Client/Server 方式または Peer-to-Peer 方式に動的に変更させる点になる。本稿は本提案の分散システムアーキテクチャの設計方針と実装方式について、説明する。

Dynamic Adaptation for Distributed System Architecture

SUN JINGTAO[1] SATO ICHIRO[1]

1. はじめに

近年、インターネットの普及と IoT や M2M 技術と伴い、分散システムに関わる研究がますます注目が高まっている。しかし、既存の分散システムはいずれも固定なアーキテクチャであり、頻繁に変化しているシステムのニーズやアプリケーションのリクワイヤメントに応じで、分散システムは最適に対応しいるとは限らない。例えば、既存分散システムにノードを参加・脱離するには、システムは一時的に中止する必要がる。また、ネットワークは頻繁的に接続・切断する場合にはシステムの利用者にサービスを提供できなくなることがある。さらに、データにアクセスが集中・分散するときに、コンピュータは障害で止まってしまうことがある。

従来の分散システムの研究 [15] [16] では、コンピュータ同士がお互いに自律的に協調しながらシステムの動的適応性を実現している。典型的な例題としては、ORB（Object Request Broker）メカニズムが挙げられる。ORB はクライアント側のオブジェクトを自律的に優位なサーバを選択し、そのサーバのオブジェクトのメソッドを動的に呼び出すような仕組みで、システムの動的適応性を実現している。しかし、従来研究では次のような課題を抱えている。

- コンピュータ上のプロセスの連携方式しか変えていない。そのため、コンピュータにプロセスを動的に追加・削除することができない。
- コンピュータ同士は相互に受送信していなかったやプロセスにタスクを割り当ていなかった場合には、リモートコンピュータのプロセスはずっと続いてしまい、コンピュータのリソースは無駄に使われてしまう状況が多い。
- コンピュータに割り当てているプロセスは静的なものものため、他のコンピュータに移動・実行ができない。

そこで、本研究では、ローカルコンピュータ上に実行しているソフトウェアの実行位置をリモートコンピュータに移動させ、新しい動的適応可能な分散システムアーキテクチャことを提案する。従来の研究手法と比べ、本提案では、コンピュータ間の受送信回数が減らせる。また、コンピュータのリソースの有効利用率を向上させることが可能になる。さらに、分散システムにノードの追加・削除を動的に対応できるメリットがる。本論文は、本提案方式のシステム構成や実装方法及び応用事例について、説明する。

本論文では、以下のセクションから構成されている。第 1 章では、本研究の背景を紹介する。第 2 章では、動的適応システムのリクワイアメントを説明し、本研究のアプローチを紹介する。第 3 章では、動的適応システムの設計方法を具体的に示す。第 4 章では、本ミドルウェアの実装

[1] 国立情報学研究所・総合研究大学院大学
NII, Hitotsubashi, Chiyoda-ku, Tokyo 101–8430, Japan

方法についてを詳解する。第 5 章では、本提案方式を用い、応用事例を説明する。第 6 章では、適応システムに関する関連研究について紹介する。最後に第 8 章で、本稿をまとめる。

2. 動的適応システムの概要

本章では、提案する動的適応システムの課題・目標とシステムのリクワイアメント及び提案方法について、述べる。

2.1 課題と目標

既存の分散システムでは、Client-Server 方式や Peer-to-Peer 方式などのネットワークアーキテクチャを用いて構築されている。しかし、分散システムアーキテクチャはいずれも固定の方式であり、既存の分散システムにノードの追加・削除を動的に対応できない。また、プロセスはコンピュータ上で動き続けるため、コンピュータのリソースが無駄に使用される。さらに、ネットの不安定や超負荷による通信レイテンシー問題とコンピュータの故障による処理の中断などの問題が挙げられる。

そのために、既存の分散システムアーキテクチャはこれらの変化に応じて、動的に適応できる能力を必要としている。そこで、本研究では、従来の分散システムアーキテクチャのように Client/Server 方式または Peer-to-Peer 方式のどちらに特化するではなく、両方式のメリットを持っている新しい動的適応可能な分散システムアーキテクチャの実現を目標にする。

2.2 リクワイアメント

既存分散システムに動的適応性を導入するためには、本研究は次のようなシステムの構築要件を考えている。

- コンピュータの障害問題。
  例えば、データセンター内のコンピュータは障害を発生した場合、既存の分散システムは利用者にサービスを提供することができなくなる。そのため、現代の分散システムでは、障害が起きていても、利用者にサービスを中断せずに、提供することを求めれている。
- ネットワークのレイテンシー問題。
  ネットワークのレイテンシーは物理的な制限であるが、レイテンシーが高ければ高いほど、ユーザに大きな影響を与える。例えば、対話型アプリケーションやビデオ通話アプリケーションなどである。ほとんどの既存の分散システムは、タイムアウトメカニズムを実装されてている。しかし、ネットワークレイテンシーはタイムアウト時間より長かった場合には、利用者はアプリケーションの信頼性を疑う可能性を生じる。そのため、現在の分散システムでは、レイテンシーをいかにも最小化にすることが求めれている。
- Input/Output 問題。
  データの読み書きは分散システムにとって、処理時間は非常に長く、減らすべきものである。例えば、システムの利用者はフロントエンドサーバーにデータの読み込み指示を送信した場合、フロントエンドサーバーは要求されたファイルの中身を一行ずつ読み、利用者に返すという仕組みになっている。そのため、現代の分散システムでは、I/O 処理時間を最小化することを必要としている。
- ビックサイズ問題。
  近年、分散システム上保存しているデータは多くなっているとともに、データサイズも大きくなっている傾向がある。データ量が小さい場合には、ネットワーク通信を介して、受信するコンピュータに移動し、処理を依頼するのがいい場合があるが、データサイズが大きくなれば、データそのものの圧縮時間や転送時間が非常に時間がかかるものである。現在分散システムでは、データを移動しなく、データの近く処理できるようなメカニズムが求めれている。
- アプリケーション設計・開発の難しさ。
  アプリケーションの開発要件は年々に難しくなっている傾向があり、プログラムの設計・開発を挑むアプリケーション開発者は難しい問題を直面しつつある。例えば、実行中のアプリケーションは他のコンピュータに移動・実行したい要件に対しては、ソフトウェアレベルでプログラムを設計・開発するが困難である。そのため、現代の分散システムはこのような開発要件をアーキテクチャレベル実現することが期待されている。

上記のようにリクワイアメントを説明したが、いずれも分散システムは動的に様々なリクワイアメントを対応できるように適応メカニズムを期待している。

2.3 アプローチ

本節では、本研究では、Client/Server 方式または Peer-to-Peer 方式の優位な方式に, 動的かつ自律的に双方向に変化できるアーキテクチャを説明する。図 1 に示す。

Client/Server 方式は機能を分離することを目的として提案されたネットワークアーキテクチャであり、Peer-to-Peer 方式は, 対等関係を持つコンピュータ同士はお互いに情報の提供と情報の共有を同時に実行できるネットワークアーキテクチャである。両方式は、システムの構成はもちろん、利用状況によって、 向き不向きの場合がある。また、 どちらかのアーキテクチャには環境変化に応じて、適応できないである.

そこで、本研究では、コンピュータの一部の機能を別のコンピュータに移動するによって、動的にアーキテクチャを転換させる。そうすると、システムの利用状況が変わっ

た時に、適応なシステムアーキテクチャでシステムの利用者にサービスを継続的に提供できるようになる。例えば、Peer-to-Peer 方式おいては、特定の Peer にデータ提供を頻繁に求められる場合、その Peer はデータ提供機能だけにして、処理機能を別の Peer に移動すれば、この Peer は Client/Server 方式のサーバに相当する。逆に、サーバ側が実行している一部の機能をクライアントに移動し、このクライアントはサーバ処理を一部を肩代わりする。従って、このクライアントは処理機能と提供機能を持つわけであり、Peer-to-Peer 方式の中の Peer に相当することも言える。

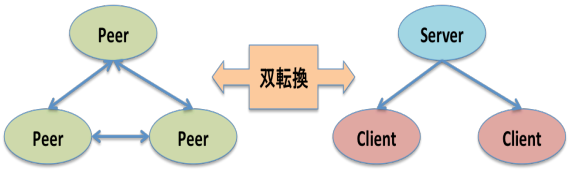

図 1　双転換アーキテクチャ

本研究では、鍵となるアイディアを以下に示す。

- 両方式のメリットを持ち、そして、環境変化に応じて、動的に適応できる分散システムアーキテクチャを実現したい。
- システムの利用状況が変化するときに、システムアーキテクチャは動的に変更して、利用者のリクワイアメントをスマートに対応したい。
- ノードの参加・脱離は、システムアーキテクチャレベルで動的に対応したい。
- ネットワークレイテンシーが高ければ、ソフトウェアの移動により、コンピュータ同士の通信回数を減らしたい。
- データサイズが大きければ、ソフトウェアの移動により、データの近くて、データを処理する。
- アプリケーション開発者のプログラムの設計・開発負担を下げたい。分散システムは状況に応じで、自律的に処理していきたい。

3.　動的適応システムの設計

本章では、提案するシステムアーキテクチャが変えられる動的に適応可能な新しい分散システムアーキテクチャの全体構成と各モジュールの構成とモジュール間での通信方式及び適応システム全体動作について、説明する。

3.1　システムの全体構成

本研究では、図 2 に示しているように JVM(Java Virtual Machine) の上にランタイムシステムモジュールを構築し、さらにその上にルール管理モジュールとアプリケーションモジュールを構築する。

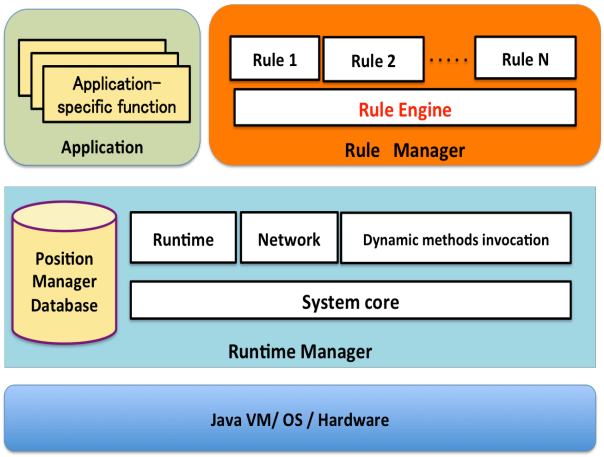

図 2　動的適応システムアーキテクチャ

3.2　モジュールの設計

本節はこの 3 つのモジュールの内部設計について、説明する。

ルール管理モジュールは、ルールエンジンを主に設計している。本研究では、システムの利用者が作った適応ルールに基づき、分散システムアーキテクチャを動的に変更させるように実装している。そのため、ルールモジュールはシステムの利用者が外部で書いていた適応ルールを管理しつつ、各ルールの内容に応じで、コンピュータ上に実行しているソフトウェアの移動先を決めておく。

ランタイムシステムモジュールは、アプリケーションモジュールの総合管理とランタイムシステムの情報管理とインターネットプロトコルの実装と動的メッソド呼び出す機能といった 4 つの基本機能として構成されている。

アプリケーションモジュールは、個々単独なアプリケーションである。

3.3　モジュール間での通信機構

モジュール間での通信方式については、次のように構成されている。

(1) ランタイムシステムモジュールは各独立しているアプリケーションを起動させ、これらのアプリケーションの一元化管理を行っている。例えば、各アプリケーションの移動やアプリケーションの状態管理とメッソドの呼び出しなどがある。

(2) ルール管理モジュールは、システムの利用者が作った適応ルールを管理する。また、それらの適応ルールをローディングし、ルールデータベース内に事前に用意していたルールとフェッチする。フェッチした結果 (移動先を示した IP アドレスとポート番号) はランタイムシステムモジュールに引き渡す。

(3) ランタイムモジュールはルール管理モジュールからの結果を実行させ、その結果を示した移動先に、実行しているソフトウェアを状態を持ちつつ、目的地のコン

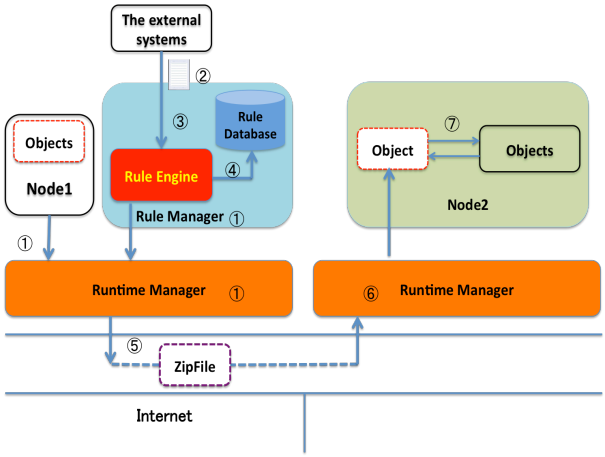

図 3 適応システム全体動作

ピュータに移動する。

3.4 適応システム全体動作

本節では、動的適応システムの全体動作について、図 3 で説明する。

- Step 1: ランタイムシステムを起動させてから、その上に、ルールマネージャーと個々のアプリケーションを動かす。
- Step 2: システムの利用者はアプリケーションの個性を考えつつ、適応ルールを作る。
- Step 3: ルールマネージャーはシステムの利用者が作った適応ルールをローディングする。
- Step 4: システムの利用者が作ったルールはルールデータベース内の既存ルールとフェッチし、その結果はランタイムシステムにお知らせする。ここでの結果というのは、アプリケーションの移動先を示している情報（IP Address と Port Number）のことである。
- Step 5: 移動先情報を含まれている結果はランタイムシステムに届いたら、移動するソフトウェアのクローンを作る。さらに、TCP/IP プロトコルを介して、クラスファイルとクローンを圧縮して、目的地のコンピュータに転送する。
- Step 6: 目的地のコンピュータは圧縮しているファイルを解凍する。
- Step 7: 動的に、届いているソフトウェアは自身のクラスファイルからメッソド呼び出す行為を実行する。

上記のステップを繰り返すことによって、分散システムのアーキテクチャを Client/Server 方法また Peer − to − Peer 方法に動的に変更し、様々なシステムの利用状況を適応する。

4. 実装

本章では、提案する適応システムのランタイムシステムとルールマネージャーの実装方法について説明する。実装に際しては、表 1 の実装環境を使用した。

表 1 実装環境

	System Information
CUP	2 Ghz Intel Core i7
OS	OS X Mavericks v10.9.2
Memory	8GB
Language	java 1.7.0.45

4.1 Rule Manager

この節では、本研究で提案する適応ルールの定義及びルールマネージャーの実行方法について、述べる。

4.1.1 ルール

本システムが扱う適応ルールというのはシステムの利用者側がアプリケーションのリクエストに従い、定義した規則のことである。その規則によって、ルールエンジンがアプリケーションの実行位置を決める。この適応ルールの中身としては、ユーザが定義する使用条件とその条件で起きる行動である。前者は、様々なシステムの要素とネットワークの状況などを含む。例えば、プロセッサの機能、ネットワークの接続状況、リソースの有効利用率、アプリケーションの固有条件などである。後者は、前者の条件に従って、ソフトウェアの移動位置を決める。

現在の実装では、シェルターミナルで「Go」「Move」のようなコマンドを作り、システムを動的に適応させることにしている。今後は、コマンドを適応システムから分離し、新しい適応言語として実装予定である。システムの利用者は適応言語を用い、数行ベースで適応ルールを書けば、分散システムが動的適応すると考えている。

4.1.2 ルールマネージャー

- ルールマネージャーを起動する。システムの利用者が作ったシェルコマンドをロードし、その中の要素を読み出す。
- 利用者が定義したルールの要素は事前にルールデータデータの中で保存しているルールの要素とフェッチする。フェッチした結果としては、次の 3 つの状況となる。
  (1) ルールデータデータ内のルールと利用者が定義したルールと一致すれば、そのルールの行動を起こす。言い換えると、そのルールに記録されているアプリケーションの実行位置ランタイムシステムにお知らせする。
  (2) ルールデータデータ内のルールと利用者が定義したルールと一致していなかったら、システムの利用者がその実行位置を決めて、ランタイムシステムにお知らせする。また、利用者が定義したルールをルールデータベースに追加し、次回からユー

ザが定義していた同じルールが来た時に、ルールエンジンはすぐ対応出来る対策にしておく。

(3) ルールデータデータ内のルールと利用者が定義したルールと一致していないかつ利用者は実行位置を決めたくない場合には、ルールエンジンはデフォルトの実行位置をランタイムシステムにお知らせし、従来の通りでシステムを動かす。

現在の実装では、適応性ルールの競合を検出するためのメカニズムをサポートしていませんが、次回の論文では、我々は設計する予定のルールを定義するための適応言語を実装した後に、ルール競合メカニズムの構築を実装する予定である。

4.2 Runtime Manager

ランタイムマネージャーを実装する際には、ランタイムマネージャーの API は表 2 のように設計している。

ランタイムマネージャーはライフサイクル機能を持ち、ソフトウェアの実行位置を管理している。また、本提案方式では、ソフトウェアのクラスファイルだけ移動するではなく、ソフトウェアのオブジェクトを直列化した後に、状態を持って目的地のコンピュータに移動することができる。そのために、ソフトウェアのオブジェクトはクラスファイルからメッソドを動的に呼び出せば、移動する前と同じ状態を持っているソフトウェアの姿が再現できる。本システムの実装では Java のリフレクションメカニズム、直列化・反直列化とスレッドプールを利用している。

4.2.1 マイグレーション

本節では、オブジェクトマイグレーションの実装方法について、述べる。図 4 示す。

- Step 1:ランタイムマネージャーを起動させる。この際には、起動するランタイムシステムの状況やランタイムシステム上においてあるソフトウェアの状況などをハッシュマップに記録する。また、移動するオブジェクトのクラスファイルの圧縮作業を行う。
- Step 2:ルールマネージャーから移動指示を受けた後に、ランタイムシステムは移動するオブジェクトのクローンを作る。クローンを使用する理由としては、クローンは移動元のコンピュータから移動先のコンピュータへ移動する間に、他のコンピュータと移動元のコンピュータとやり取りを行われていても、プロセスが止まらなく、処理が続くことができるからである。オブジェクトのクローンを作るためには、Java の Cloneable インターフェースを実装している。
- Step 3:直列化されたクローンオブジェクトを生成したら、クローンに唯一 ID 番号を付けてクラスファイルと一緒に、目的地のコンピュータに送る。本研究では、ID 管理しやすくためには、UUID クラスを使用していなく、ネットワークの IP アドレス、ポート番号、

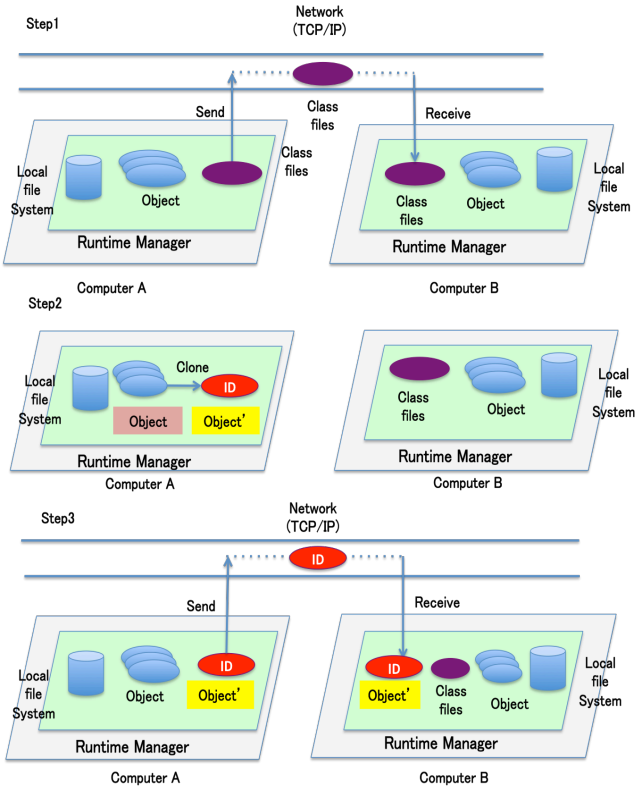

図 4 ランタイムマネージャー

システムの現在時間及び乱数を用いて、UUID よりさらに、自由性が高い３２ビットの唯一 ID を作ることにしている。

4.2.2 動的メッソド呼び出し

本節では、動的メッソド呼び出の実装について、述べる。

- 目的地側のコンピュータに圧縮されているファイルを届いたら、そのファイルを解凍する。
- ユーザが定義しているクラスローダーから移動元から届いていたクラスファイルをローディングする。このクラスローダーはシステムクラスローダーを継承して生成したものである。
- 目的地のコンピュータにソフトウェアのクローンが届いたら、Java のリフレクションメカニズムを使用して目的地側のコンピュータに呼び出したいフィールド、メッソド、コンストラクタなどを値を設定する。
- 目的地にある元オブジェクトは移動元からきたソフトウェアのクローンが動的に呼び出され、移動元と全く同じ状態を持つソフトウェアを蘇る。
- ソフトウェアは移動先で継続的にシステムの利用者からのリクエストを処理する。また、システムの利用状況を変化したら、このソフトウェアは移動元にリスポンス返していく手順である。

本提案では、上記の方式を繰り返すことで、システムは利用状況が変わったときに、ソフトウェアの実行位置を移動することによって、システムアーキテクチャレベルで動

表 2 RuntimeSystem API

名前	説明
public void init()	初期化
public void start(ObjectID id,Object app)	ランタイムシステムの起動。
public void go(RuleInfo result, Object[] objects)	アプリケーションの移動
public Method invoke(Object methodName)	動的メッソド呼び出しを
public void release(Object[] objects)	アプリケーション状態の書き込み
public Object[] acquire()	アプリケーション状態の読み込み
public void terminate()	ランタイムシステムの終了

的適応性を実現している。

4.3 研究方法の利点と欠点

本提案手法は、次のようなメリットがあると考えている。

- 分散システム上のコンピュータがクラッシュしたとしても、クラッシュしたコンピュータの一部のプログラムは他のコンピュータに継続的に処理することできる。
- ネットワークが切れたり、接したりするの場合には、コンピュータの一部のプログラムは、通信している向こう側のコンピュータに移動することにより、処理が続けられる一方で、通信回数が減らせる。
- クライアントのニーズが変わった場合には、サーバ側は動的に適応できるが、サーバ側のニーズにより、クライアント側にも動的に適応できる。言わば、双方向に適応可能な分散システムになる。
- 移動するプログラムのサイズが小さいため、通信コストが少なくなり、システム全体的な運営コストが下がられる。
- 本ミドルウェアは CORBA(Common Object Request Broker Architecture) を基づいて設計しているため、コンピュータ同士の間のデータ通信を簡単に隠すことができる。
- 本ミドルウェアは RMI(Remote Method Invocation) と違い、オブジェクトのマイグレーションをサポートしている。

当然、本ミドルウェアの実装方式により、オブジェクトの直列化・反直列化とプログラムの再配置といった問題の最小化処理が残っている。また、アプリケーションの移動によるセキュリティ問題の解決案につては、議論していく必要がある。

5. 応用事例

本章では、３つの応用事例を用いて、本提案手法応用について、説明する.

5.1 コンテンツ配信サービス

コンテンツ配信サービスでは、配信側となるサーバと受信側となるクライアントから構成されている。そのため、PC やスマートフォンや タブレットなどの電子端末のモニターサイズに応じて、フレームレートサイズの自己合わせなどのようなシステムリクワイアメントは配信側のサーバに送信したと想定する。

現代のコンテンツ配信サービスでは、フレームレートのような最適化処理は配信側のサーバ側で実行している。しかし、配信側のサーバのタスクを増えるため、サーバは負荷状況となるケースが少なくない。一方で、クライアント端末は高性能化が進んでおり、計算リソースは余っていることが多くなっている中で、本提案手法を用いけば、フレームレートサイズを変化するプログラムを受信側のクライアントに移動すれば、クライアント側は移動してきたプログラム実行し、サーバ側から届いているコンテンツは自己合わせすれば、サーバの処理負担が減らせるとともに、サーバ台数は減らせるかもしれないし、クライアント端末が余ったいるリソースを有効に使用できるメリットがある。

5.1.1 コンピュータの障害防止

例えば、分散ファイルシステム [17] [18] [19] の事例のように、

- ある特定の時間帯で、システムの利用者はフロントエンドサーバに大量な同じファイルにアクセスするリクエストを送った場合には、フロントエンドサーバ側の処理は普段より一時的に超負荷状態になると予測できる。その際に、フロントエンドサーバ側は一部のプログラムをバックエンドサーバに送り、その処理がバックエンドサーバ側に分散して処理することが可能になる。そうすると、フロントエンドサーバの負荷状態が緩和でき、サーバ台数を増やすより、システムの構築コストと減少させることが可能になる。
- また、データサイズを変換するリクエストはシステムの利用者からフロントエンドサーバに送った場合には、まず、バックエンドサーバが要求されているデータをフロントエンドサーバに送り、フロントエンドサーバはそのデータを加工してからユーザに返すという流れになるが、リクエストの大量発生やネットワークの不安定や転送時間が長い場合には、データの加工はバックエンドサーバで処理すべき、フロントエンドサーバはデータ加工するプログラムをバックエンドサーバに

送り、処理した結果だけもらえば、いいと考えている。本提案方式を利用すれば、このような問題は分散システムアーキテクチャが自律的に解決してくれるメッソドがある。

5.1.2 ネットワーク状況

ネットワークが輻輳している場合、それに対処する方法として、通信ノードは通信回数を減らす方法とデータ量を減らす方法の二つがある。

前者に関しては、 例えば二つのノードにおいて通信が頻繁に行われる場合、 片方のノードの処理の一部をソフトウェア移動により、 もう片方のノードに移動させることで、 ノード間通信をノード内通信に変えることで輻輳を減らせるメッソドがる。Unix 系システムの grep コマンドは、まさかこのように事例である。ファイルを一行毎にパターンマッチングし、 対象となるファイルが別のコンピュータにあると、 ファイルを一行毎に読み込んでその各行内容を通信する必要がある。そのため、本提案方式を用いて、 grep 相当のプログラムをサーバに移動させれば、通信量を激減することができる.

後者に関しては、例えば、 データを送信前に圧縮してから送信し、 受信側で再びも元のデータに戻す手法では、通信量が減らせるが、しかし、コンテンツ種類に応じで、圧縮方式が選べられないデメリットがある。そのため、本研究手法を用いて、圧縮プログラムとを送信側のコンピュータに配置し、解凍プログラムは受信側のコンピュータに配置すれば、コンテンツ種類に応じで、適切な圧縮アルゴリズを選ぶこともでき、さらに、コンピュータの通信量が減らせる。

6. 関連研究

既存分散システムに適応性を実現するために、パラメータの変更やソフトウェアの書き換えなどの研究が提案されている。例えば、Qiang Huo チームや Epifani Ilenia 氏など [1] [2] [3] では、パラメータの変更による、システムが動的に変更できると提案している。しかし、これらの提案手法では、適応可能範囲が限定されている問題がある。また、MIT の Koza 氏では、遺伝アルゴリズム [4][5][6] を活用し、適応性の実現を提案している。しかし、遺伝アルゴリズムでは、ソフトウェアを書き換えは事前予測不可能であるし、コンピュータのリソースを膨大に使われてしまう問題がある。Koza 氏と違い、Dowling Jim チームや Rainbow チームでは、ソフトウェアの書き換 [7] [8] [9] [10] を事前に予測して適応性を実現しているが、ソフトウェアを書き換えする前に、予測ためのデータの計算量と時間がかかってしまうし、書き換えた後に、データの一貫性の問題と分散システムの信頼性に関わる問題は生じてしまう。

従って、本提案方式では、分散システムに動的適応性を実現するために、モバイルエージェント技術を用いて、動的適応性を実現した。現在のモバイルエージェントシステムに関わる研究 [11] [12] [13] [14] では、リモートコンピュータに移動したプログラムは移動先のコンピュータにあるプログラムを継続的に処理を続くような研究が面になっているため、動的適応性にはサポートしていないである。しかし、本提案手法では、ソフトウェアの実行位置を他のコンピュータに移動し、分散システムアーキテクチャを Clinet/Server または Peer − to − Peer 方法に動的かつ双方向に変更することは可能した。

7. おわりに

本稿では, 既存分散システムのシステム概要、提案した動的適応可能にするミドルウェアの設計方法、実現手法及び応用事例について概説した。 本提案の動的適応システムは、システムの利用状況に応じて、システムが Client/Server 方式または Peer-to-Peer 方式のどちらの有利の方式を選び、分散システムアーキテクチャを動的に変更できる新たな分散システムアーキテクチャを提案した。本提案手法では、Client/Server 方式または Peer-to-Peer 方式のメリットを取り、様々なチャレンジをもたらしてきた。

今後は, 我々が提案した動的適応可能にするミドルウェアの有効性を検証するために、複数のアプリケーションを作り、本研究で提案したミドルウェアを用いて、評価実験を行う予定がある。 また、本提案手法では、センサーネットワークやサイバーフィジカルシステムにも有効な方法と考え、センターネットワークシステムとサイバーフィジカルシステムへの応用も試していく予定である。

参考文献

[1] Lee, Chin-Hui, and Qiang Huo. "On adaptive decision rules and decision parameter adaptation for automatic speech recognition." Proceedings of the IEEE 88.8 (2000): 1241-1269.

[2] Zaharie, Daniela. "Control of population diversity and adaptation in differential evolution algorithms." Proc. of MENDEL. Vol. 9. 2003.

[3] Epifani, Ilenia, et al. "Model evolution by run-time parameter adaptation." Software Engineering, 2009. ICSE 2009. IEEE 31st International Conference on. IEEE, 2009.

[4] Koza, John R., and Rice, James P. Process for Problem Solving Using Spontaneously Emergent Self-Replicating and Self-Improving Entities. United States Patent No. 5,390,282. Filed June 16, 1992. Issued February 14, 1995.

[5] Koza, John R. Non-Linear Genetic Algorithms for Solving Problems. Japanese Patent Application No. 126512/89 in 1989. Japanese Patent 2,818,802. Issued August 28, 1998.

[6] Koza, John R. "Introduction to genetic programming tutorial: from the basics to human-competitive results." Proceedings of the 12th annual conference companion on Genetic and evolutionary computation. ACM, 2010.

[7] Dowling, Jim, et al. "Using reflection to support dynamic adaptation of system software: A case study driven eval-

uation." Reflection and Software Engineering. Springer Berlin Heidelberg, 2000. 169-188.

[8] Lopes, Leandro, et al. "Distributed Requirements Specification: Minimizing the Effect of Geographic Dispersion." ICEIS (3). 2004.

[9] Cheng, Shang-Wen. Rainbow: cost-effective software architecture-based self-adaptation. ProQuest, 2008.

[10] Satoh, Ichiro. "Self-organizing software components in distributed systems." Architecture of Computing Systems-ARCS 2007. Springer Berlin Heidelberg, 2007. 185-198.

[11] Germanakos, Panagiotis, Constantinos Mourlas, and George Samaras. "A mobile agent approach for ubiquitous and personalized eHealth information systems." Proceedings of the Workshop on' Personalization for e-Health' of the 10th International Conference on User Modeling (UM 2005). 2005.

[12] Fok, Chien-Liang, Gruia-Catalin Roman, and Chenyang Lu. "Agilla: A mobile agent middleware for self-adaptive wireless sensor networks." ACM Transactions on Autonomous and Adaptive Systems (TAAS) 4.3 (2009): 16.

[13] Weyns, Danny, and Michael Georgeff. "Self-adaptation using multiagent systems." Software, IEEE 27.1 (2010): 86-91.

[14] Aversa, Rocco, et al. "Cloud agency: A mobile agent based cloud system." Complex, Intelligent and Software Intensive Systems (CISIS), 2010 International Conference on. IEEE, 2010.

[15] Truyen, Eddy, Bo Norregaard Jorgensen, and Wouter Joosen. "Customization of component-based Object Request Brokers through dynamic reconfiguration." Technology of Object-Oriented Languages, 2000. TOOLS 33. Proceedings. 33rd International Conference on. IEEE, 2000.

[16] Yau, Stephen S., et al. "Reconfigurable context-sensitive middleware for pervasive computing." IEEE Pervasive Computing 1.3 (2002): 33-40.

[17] 孫　静涛, 佐藤　一郎. 動的適応可能な分散ファイルシステムの提案. 第 126 回 OS 研究発表会 (SWOPP2013), 情報処理学会. 2013 年 7 月.

[18] Ghemawat, Sanjay, Howard Gobioff, and Shun-Tak Leung. "The Google file system." ACM SIGOPS Operating Systems Review. Vol. 37. No. 5. ACM, 2003.

[19] Shvachko, Konstantin, et al. "The hadoop distributed file system." Mass Storage Systems and Technologies (MSST), 2010 IEEE 26th Symposium on. IEEE, 2010.