

推薦論文

適応型分散オブジェクト指向環境の実現

加藤 健士[†] 小田 謙太郎[†] 吉田 隆一[†]

分散オブジェクト指向環境では様々な計算機が接続されている。このような環境ではネットワーク構成の変化、また電源異常などの実行環境の動的変化が必ず発生する。本研究ではこのような変化に適応できる能力を個別のオブジェクトに与えた適応型分散オブジェクト指向環境の実現を目指す。本論文では実行環境の変化の検出と、その変化に適応するための戦略の提供を行う枠組みについて述べる。この枠組みでは、各オブジェクトが戦略を立てるオブジェクトを持ち、環境の変化が事象としてそのオブジェクトに通知される。また、我々が開発している分散オブジェクト指向環境「Juice」に対しての実装を行った。これによって計算負荷が高い場合や電池残量が減少した場合などの状況に対して適応するために、オブジェクトの移送などを行い計算を続行することが可能になった。

Implementation of the Adaptable Distributed Object-Oriented Environment

TAKESHI KATO,[†] KENTARO ODA[†] and TAKAICHI YOSHIDA[†]

In a distributed object-oriented environment, various computers are connected in order to coordinate actions. In such an environment, dynamic changes of the runtime environment surely arise (e.g., changes of network topology, abnormality of power source). Our research aims at implementing the Adaptable Distributed Object-Oriented Environment, in which each object has the ability to adapt dynamic changes of environment. In this paper, we describe a framework that detects changes of the runtime environment and provides strategies for adaptation such changes. Within the user-defined object, an object designed to provide this strategy has been added. Then, the detected change of the environment is passed to it as an event. Moreover, we implement it for our Distributed Object-Oriented Environment "Juice". With this implementation, the system is able to continue computation through object migration to respond to situations like high computation load, low battery etc.

1. はじめに

現在、Object Management Group の CORBA¹⁾ などの分散オブジェクト技術を用いたシステムが普及している。このような分散オブジェクト技術を用いたシステムでは様々なアーキテクチャを用いた計算機がネットワークに接続され、通信を行いながら計算が進められる。このようなシステムではソフトウェアの更新や、ネットワーク構成の変化などのシステムを構成する計算機の変化が発生する。また計算負荷の変動や、電池の消耗による電源異常などの変化も発生する。分散システムでは、このような実行環境の動的な変化に対して適応する機能が必要となる。

たとえば、ノート型計算機の電池の消耗などの電源

異常が発生した場合には、他のホストへオブジェクトを移送することで、計算を続行することが可能になる。本論文では実行環境の動的な変化に対して適応していく能力を持った分散オブジェクト指向環境を適応型分散オブジェクト指向環境と呼ぶ。本論文は、適応型分散オブジェクト指向環境の実現を目指して、アプリケーションを構成する各オブジェクトに環境変化に適応する能力を持たせることを目的とする。

環境変化への適応能力を持ったオブジェクトを実現するモデルとして、flying-object モデル²⁾を用い、分散オブジェクト指向環境「Juice」³⁾に実行環境の動的な変化に適応するための機能を実装した。Juice は我々が開発している分散オブジェクト指向環境で、Java のコードへ変換するためのトランスレータと、分散計算

[†]九州工業大学情報工学部
Department of Artificial Intelligence, Kyushu Institute of Technology

本論文の内容は 2001 年 3 月の火の国情報シンポジウム 2001 にて報告され、火の国情報シンポジウム 2001 プログラム委員長により情報処理学会論文誌への掲載が推薦された論文である。

に必要な機能を提供するランタイムライブラリから構成されている。また、実装に Java 言語を用いているので、ハードウェアの違いなどは Java VM によって吸収される。そこで、ランタイムライブラリに環境の変化に適応するための機能を追加した。

本論文では、まず 2 章で分散オブジェクト指向環境 Juice について述べ、3 章で適応型分散オブジェクト指向環境の実現方針、4 章でその実装について述べる。また、5 章で本実装での評価を行い、6 章で関連研究との比較、今後の課題について述べる。最後に 7 章でまとめを行う。

2. 分散オブジェクト指向環境 Juice

Juice は我々が開発を行っている分散オブジェクト指向環境である。ここでは Juice について説明する。

2.1 Juice の特徴

分散オブジェクト技術としては Object Management Group の CORBA、また Java ベースでは電子技術総合研究所の HORB⁴⁾ や、Sun Microsystems 社の Java RMI⁵⁾ などがある。これらの技術を利用することで、プログラムは Socket などを用いた複雑なネットワークプログラミングをせずに、リモートホスト上のオブジェクトに対してアクセスすることができる。しかし、これらの技術を用いてもリモートホスト上のオブジェクトと通常のオブジェクトの間には違いがある。たとえば、HORB の場合にはリモートホスト上の Server オブジェクトにアクセスするためには Server_Proxy という異なるクラスである代理オブジェクトを用いなければならない。そのためにプログラムはこうした代理オブジェクトと通常のオブジェクトの違いなどを意識してプログラミングをしなければならない。

Juice では図 1 のような仮想的な空間を提供する。Juice ではプロキシオブジェクトなどの代理オブジェクトと通常のオブジェクトとの違いをユーザに見えないようにシステム側で隠蔽している。そのため、プログラムは代理オブジェクトと、通常のオブジェクトとの違いなどを意識せずにプログラムを記述することができる。

また、Juice では分散計算の特性を活かすためにメッセージ送信の並行記述が実現されている。これには並行メッセージ送信と、非同期メッセージ送信が用意されている。

2.2 Juice のオブジェクトモデル

Juice でのオブジェクトは flying-object というモデルを用いて実装されている。flying-object は user-

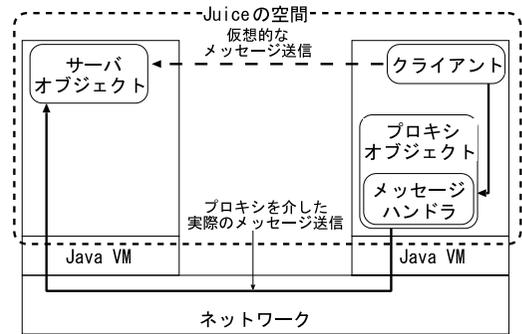


図 1 Juice の提供する空間

Fig. 1 The space that provided by Juice.

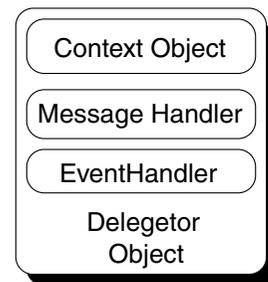


図 2 flying-object のモデル

Fig. 2 The model of flying-object.

defined なオブジェクトであること、flying-object 自身も言語が提供するオブジェクトとなら変わらないという点で first-class オブジェクトであること、オブジェクトの振舞いを動的に変更できることがあげられる。

flying-object は図 2 のようにオブジェクトに必要な機能を実現する 4 つのオブジェクトから構成される。flying-object を構成する 4 つのオブジェクトはそれぞれ以下の機能を有する。

- **DelegatorObject** : 他の構成オブジェクトを包む“皮”であり、全体として 1 つのオブジェクトに見せる。すなわち、プログラマからは DelegatorObject のみが見える。DelegatorObject はプログラマが定義したオブジェクトと同じ型に見えるため、プログラマは flying-object であることを意識する必要がない。flying-object に渡されるメッセージはすべて DelegatorObject が受け取り、MessageHandler や EventHandler に送る。
- **MessageHandler** : DelegatorObject から送られてきたメッセージを受け取り、リモートホスト上のオブジェクトや ContextObject にメッセージを送るなどの処理を行う。
- **EventHandler** : 実行環境の変化を Event とい

う形で受け取り、それに対しての戦略を提供する。戦略はどの Event に対して、どのように適応するかを指示する。たとえば、電源の異常などを Event として受け取り、オブジェクトの移送などを行う。

- **ContextObject** : オブジェクトの状態や、メソッドなどを持つユーザが定義したオブジェクトである。実際のメソッドの処理は ContextObject で行う。

本論文では適応型分散オブジェクト指向環境の実現を目指し、EventHandler を実装することで環境の変化に適応する機能を実現した。

2.3 Juice の実装

上記のモデルを実現するため、Juice を実装した。Juice は Java への変換を行うトランスレータと、ランタイムライブラリから構成される。ランタイムライブラリには、前節の MessageHandler や EventHandler のコードが含まれ、分散計算を行うために必要な機能などが実装されている。また、トランスレータは、ソースコード変換によりユーザが定義したクラスのソースコードファイルを入力とし、Java のコードとして出力する。トランスレータの出力コードには、MessageHandler や EventHandler のインスタンス化を依頼するコードや、分散計算を行うための MessageHandler へのメソッド呼び出しのコードなどが埋め込まれる。またユーザの定義したクラスと同一の型のオブジェクトを DelegationObject としてインスタンス化するコードも生成する³⁾。

3. 適応型分散オブジェクト指向環境の実現方針

3.1 基本モデル

適応型分散オブジェクト指向環境を実現するために、本論文では計算負荷の変動などの環境の変化が Event という形で、オブジェクトに対して通知されるようなモデルを考える。

環境の変化を検出するために Monitor オブジェクトを導入する。しかし、Java VM によって実行環境の情報が隠蔽されること、また Event の通知先のオブジェクトの管理などを考えると Monitor のみで実装することは難しい。そこで Observer と Monitor レジストリを導入した。

ある Event に対してオブジェクトが適応のために選択する行動を、ここでは戦略と呼ぶ。上述のように選択された戦略に対し、実際に適応するためにとる詳細な行動の定義をここでは戦術と呼ぶ。たとえば、CPU

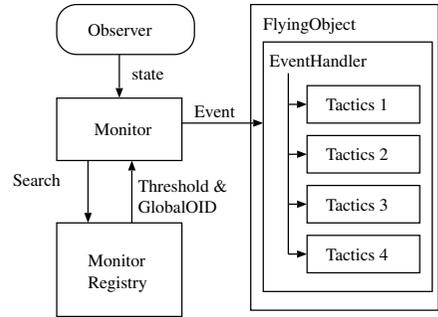


図 3 基本モデル

Fig. 3 The basic model.

の負荷が高くなった場合、戦略として他のホストへのオブジェクトの移送や、ディスクへのスワップアウトなどが考えられる。戦術は、どのようにオブジェクトを他のホストへ移送するかなどの詳細な定義である。

環境の変化の検出からそれに対応する処理のモデルを図 3 に示す。図 3 での各オブジェクトやプログラムの機能は以下のとおりである。

- **Observer** : 実行環境の変化を検出するためのプログラムである。Java VM とは別のプロセスとして動作している。Observer が検出した実行環境の状態は state として Monitor に送られる。
- **Monitor** : Observer から state を受け取り、Event として各オブジェクトに対して送るオブジェクトである。
- **Monitor Registry** : Monitor が各オブジェクトに Event を送る際に参照するテーブルである。これは、Event の送り先となるオブジェクトの ID である GlobalOID、Event に対して適応を行うかを判断するための閾値 Thresholdなどを管理する。
- **EventHandler** : Event に対してどのように適応すべきかの戦略を提供するオブジェクトである。
- **Tactics** : Event に対して、上記の戦略に従って実際にとる行動である戦術を記述したオブジェクトである。このオブジェクトは EventHandler 内に存在する。

以下、これらの各項目について詳細に説明する。

3.2 環境変化の検出

3.2.1 Observer

Observer は環境の変化を検出する。Java VM により実際の実行環境が隠蔽されているため、Monitor から見た抽象的な環境として Observer を用意する。ここでは Observer を別プログラムとして作成し、Java VM とは別のプロセスとして動作させ、プロセス間通

信により Monitor との通信を実現することとした。

実行環境の変化を検出するためには native method を用いる方法も考えられる。しかし、native method を用いて実行環境の変化を検出する方法では、動的な機能の追加、更新を行うたびに、新たな native method をロードし、Java VM 自体を再起動する必要がある。そのため、Observer を別プロセスとする方式を採用した。

3.2.2 Monitor レジストリ

Monitor は検出した環境の変化を Event の形でオブジェクトに対して送る。そのため Monitor はオブジェクトへの参照など、オブジェクトに関する情報を知らなければならないので、Monitor はそれらの情報を管理する Monitor レジストリを持つようにした。Monitor レジストリに格納される情報は、外部ファイルとして提供される。また Monitor は、Monitor レジストリを外部に持つことにより、複数の Monitor から情報を共有できるようにした。

Monitor がどのオブジェクトに対して Event を送るのかについて幾つかのパターンが考えられる。まず、すべてのオブジェクトに対してすべての Event を送る方法がある。次に、Event のマスクを用意して、Event を受け取ると宣言したオブジェクトに対してのみ Event を送信する方法などである。

EventHandler は戦略を提供するものなので、本来ならすべての環境変化を受け取り、適応すべき Event の選択や閾値との比較などの作業は EventHandler で行うべきである。しかし、この手法をとる場合はすべての環境変化が、すべてのオブジェクトに対して送られることになり、コストを考えた場合に現実的ではない。そこで本研究では Event のマスクの情報や閾値に関する情報もレジストリに保存し、適応すべき Event の選択や閾値との比較などは Monitor が Monitor レジストリに依頼して実行することとした。

3.2.3 Monitor オブジェクト

Monitor オブジェクトは Observer から実行環境の状態を受け取り、環境の変化を Event として各オブジェクトに送信する。この際、Monitor オブジェクトは、Monitor レジストリに依頼して、適応すべき Event の選択を行い、各オブジェクトに対して実際に Event を送信すべきか否かの判断を行う。

3.3 EventHandler

オブジェクトに送られてきた Event に対して、環境変化への適応の戦略を選択し、実行するのが EventHandler である。EventHandler では Event の種類によって、環境に適応するための処理を実行する。

3.3.1 オブジェクトと EventHandler の関連づけ
送られてくる Event に対してどのような戦略を選択するかは、オブジェクトの性質に依存する。たとえば、File といった OS に直接依存するオブジェクトの EventHandler では直列化ができないため、オブジェクトの移送といった戦略をとることができない。そのため EventHandler をどのようにオブジェクトに関連づけるのかという問題がある。

オブジェクトと EventHandler の関連づけとしては以下のようなパターンが考えられる。

- オブジェクトごとに指定：オブジェクトの生成時に EventHandler を指定する。この場合には、各オブジェクトごとに細かい指定ができるが、Juice 言語自体に EventHandler を指定するための構文の追加が必要になる。また、Juice 言語を使って作られたアプリケーションを利用する管理者から EventHandler を指定することが困難になる。
- クラス単位で指定：クラス名と EventHandler の対応表を記述し、オブジェクトを生成する際にこの表を参照しながら EventHandler を生成する。上記の 2 つについて考えた場合に、オブジェクト単位で指定するほうが flying-object モデルには適している。しかし、言語の拡張が必要になりコストがかかる。クラス単位での指定ではオブジェクトごとほどに細かい制御ができないが、アプリケーションを使う管理者からでも指定を行うことが可能になる。そのため、クラス単位で指定したほうが使いやすいのではないかと考え、クラス単位で指定するようにした。

3.3.2 戦略の記述

クラス単位での EventHandler の指定方法としては、Juice 言語の拡張によりクラス定義部分に記述する方法と、別のファイルにクラスと EventHandler の対応を記述する方法がある。本研究ではアプリケーションの管理者からでも戦略が記述できるように、別のファイルで記述する方法を選択した。以後、EventHandler を記述するファイルを EventHandler 定義ファイルと呼ぶ。

EventHandler 定義ファイルではクラス名と EventHandler の対応、EventHandler が提供する戦略のリストなどが宣言的に記述される。さらに、Event と閾値、またそれに対する戦略も EventHandler 定義ファイルに宣言的に記述される。

3.3.3 戦術の記述

Tactics オブジェクトが戦術を提供する。

戦術の記述方法としてはユーザに戦術を記述させる方法と、ランタイムライブラリで提供する方法の 2 つ

が考えられる。しかし、アプリケーションの管理者に戦術を記述させることは現実的ではない。したがって戦術はランタイムライブラリの実装の段階で用意するようにした。もちろん、アプリケーションのプログラマが Tactics オブジェクトを実装し、戦術を定義することも可能である。

4. 適応型分散オブジェクト指向環境の実装

前章までで適応型分散オブジェクト指向環境を実現するための枠組みと、設計について述べた。ここでは実際に行った実装について述べる。

4.1 実行環境の変化の検出

4.1.1 Observer の実装

本研究では実行環境の変化を検出するために Observer を導入した。本実装では Linux 用の Observer の実装を行った。これは proc ファイルシステムを利用することで実行環境の計算負荷の変化や、ノート型小型計算機における電池の状態などの情報を容易に知ることができるからである。本研究では Linux 用の Observer のみを実装したが FreeBSD などにおいても kmem などを利用することでこれらの情報を取り出すことができ、容易に実装することが可能である。

この Linux 用の Observer では以下の実行環境の変化を state として検出し、Monitor へ出力する。

- CPU の負荷
- メモリの空き容量
- ネットワークの使用率
- ロードアベレージ
- 電池の残量

Observer は Monitor オブジェクトによって子プロセスとして実行される。Monitor オブジェクトと Observer の通信は実装を容易にするために標準入出力を用いて行うようにした。

4.1.2 Monitor レジストリの実装

Monitor レジストリはハッシュテーブルを用いて実装した。ハッシュテーブルには Event の送信先となるオブジェクトへの参照と、閾値などの Event マスクのデータが記述されている。Event マスクのデータは、Event ID と閾値のデータのペアの列を Vector オブジェクトとして表現している。

閾値データ

閾値データは EventThreshold クラスのサブクラスとして定義されている。EventThreshold クラスは Event オブジェクトの値と閾値の比較を行うためのメソッドを持つ。

4.1.3 Monitor オブジェクトの実装

前節で述べたように、Monitor オブジェクトは子プロセスとして環境の変化を検出する Observer を生成する。次に、Observer からの出力を解析し Event に変換する処理を行っている。ここでは Event と Monitor での処理について述べる。

Event オブジェクト

Monitor が検出した実行環境の変化は Event オブジェクトとしてオブジェクトに送られる。この実行環境の変化をデータとして表現するために Event クラスを定義した。Monitor が検出するすべての変化はこの Event クラスのサブクラスとして表現される。

Monitor の実行

Monitor オブジェクトでは、Observer から送られてきた state を Event オブジェクトに変換する。次に Event オブジェクトを使って Monitor レジストリを探索し、Event を受け取るオブジェクトを決定する。Event のマスクや閾値との比較などの処理は Monitor レジストリで行われる。

Event を送信する際の処理

Monitor オブジェクトは、Observer から state が送られてきた場合には以下の処理を行う。

- (1) state から Event オブジェクトを生成する。
- (2) Event オブジェクトを引数として Monitor レジストリの lookup() メソッドを実行する。lookup() メソッドでは以下の処理を行う。
 - (a) 渡された Event オブジェクトの Event ID をキーとしてテーブルをひく。
 - (b) 取り出された閾値データのメソッドによって Event オブジェクトの値と閾値の比較を行う。
 - (c) 閾値との比較でオブジェクトへ Event を送ることが決定した場合に、テーブルから得られたオブジェクトへの参照を Vector オブジェクトに格納する。
 - (d) Vector オブジェクトを Monitor に返す。
- (3) Monitor レジストリから受け取った Vector オブジェクトに格納されているオブジェクトすべてに対して Event オブジェクトを送る。

4.2 EventHandler 定義ファイル

EventHandler が提供する戦略などは前章で述べた EventHandler 定義ファイルに記述される。本研究では実装のコストと可読性などを考え、XML によって EventHandler 定義ファイルを記述することにした。図 4 に XML による戦略の定義例を示す。この例ではロードアベレージが高くなった場合にオブジェクトの

```

<BindingList>
  <DefaultBinding>
    <ClassName>    <!-- 対応づけるクラス名 -->
      Juice.DirectoryServer
    </ClassName>
    <EventHandler> <!-- 戦略定義 -->
      <Strategy>   <!-- 戦略の 1 つを定義 -->
        <ClassName> <!-- 使用する戦術 -->
          MigrationTactics
        </ClassName>
        <Threshold>
          <Event>  <!-- 監視する Event -->
            LoadAverageEvent
          </Event>
          <Value> 3.0 </Value> <!-- 閾値 -->
        </Threshold>
      </Strategy>
    </EventHandler>
  </DefaultBinding>
</BindingList>

```

図 4 EventHandler 定義ファイルの記述

Fig. 4 Description of an EventHandler definition file.

移送を行うという戦略を記述している。また、その戦略を Juice.DirectoryServer というクラスに対応づけている。

4.3 EventHandler の生成

EventHandler の生成処理は 3 つの処理に分けることができる。最初の段階は XML で記述されたクラス名と EventHandler の対応や戦略の定義を解析する処理になる。次にクラス名に一致した EventHandler を生成する処理、最後に Monitor レジストリに必要な Event マスクを登録するなどの初期化処理になる。以下にそれぞれの処理の実装について述べる。

4.3.1 EventHandler の記述のパーズ

前節で述べたように戦略は XML によって EventHandler 定義ファイルに記述される。そのため、EventHandler を生成するためにはまず XML のパーズを行わなければならない。本実装では Juice で書かれたアプリケーションが起動されたときに EventHandler 定義ファイルのパーズが行われることとした。

4.3.2 EventHandler の生成

戦略を提供する EventHandler は EventHandler クラスとして定義した。EventHandler では XML で書かれた EventHandler 定義ファイルをパーズした結果

表 1 実験に用いた環境
Table 1 The testing environment.

CPU	Celeron 500 MHz
RAM	384 MB
OS	Linux 2.4.12
Ethernet	10 Base-T
JDK	Java 2 Platform, SE v1.4.0

を基に、Tactics オブジェクトを生成し、Event ID をキーとしてハッシュテーブルに格納する。Event を受け取ると、このハッシュテーブルに基づいて戦術が実行される。

4.3.3 EventHandler の初期化

Monitor レジストリに対して Event マスクや、閾値の登録をすることは EventHandler の仕事になる。このような Monitor レジストリへの情報の登録などの処理を、ここでは EventHandler の初期化と表現している。

EventHandler の初期化では以下のような処理を行う。

- (1) XML で書かれた EventHandler 定義ファイルのデータを取り出す。
- (2) Monitor レジストリに Event マスクと閾値を登録
 - (a) Strategy を 1 つ取り出す。
 - (b) Tactics オブジェクトを生成する。
 - (c) 閾値を Strategy から取り出して、Monitor レジストリに登録。
 - (d) 上の処理をすべての Strategy に対して繰り返す。

5. 評価

5.1 実験

ここで提案したモデルと、本実装の性能の評価を行うために、適応動作に関してどの程度の時間を要するのかを実験により測定した。実験に用いた環境を表 1 に示す。

まず、実際の適応を行う処理にどの程度の時間がかかるかを測定した。適応の条件と、その条件が成立した場合の適応戦略に関して、以下の 2 つの場合に関して実験を行った。

- (1) ロードアベレージが 1.0 を超えた場合にオブジェクトの移送を行う。
- (2) 物理的なメモリの空き容量が 20 MB 以下になった場合にディスクへスワップさせる。

すなわち、実際の適応を行う処理に必要な時間として、オブジェクトを別の計算機へ移送する戦術を実行

表 2 戦術実行に要した時間

Table 2 Elapsed time for invocation of tactics.

tactics	elapsed time (ms)
migrate to other servers	160
swap out to disk	41

表 3 Event 通知のオーバーヘッド

Table 3 Overhead for event notification.

Num of objects	elapsed time (μ s)
1	4.3
10	18
100	230
1,000	5,700

するのに要する時間と、オブジェクトをディスクへスワップする戦術を実行するのに要する時間の測定を行った。測定結果を表 2 に示す。

オブジェクトの移送に要する時間の測定ではオブジェクトが他の計算機へ移動した直後に現在の計算機へ戻るようにし、オブジェクトが計算機間を往復するのに要した時間を測定した。実際のオブジェクト移送の戦術では他の計算機への移送のみなので、表に示す結果は得られた時間の 1/2 である。また、オブジェクトのディスクへのスワップでは実際にオブジェクトをファイルに書きこむ際に要した時間の測定を行った。

我々のモデルでは、適応戦略は個々のオブジェクトに対して与えることができるので、ここにあげた戦術実行に必要な時間は、個々のオブジェクトに対して適応戦略を決定する際の目安となる。たとえば、寿命の長いサーバオブジェクトの場合は、一時的に応答性が低下するとしても、平均的な応答性を考えると移送による時間消費は問題にならない。また、一般的に環境の変化はそう頻繁に起こるものではないため、戦術の実行に要する時間は問題にならない程度の時間だと考える。

次に、環境変化が検出されてからオブジェクトに通知されるまでの時間の測定を行った。この実験ではつねに Event を発生し続けるような Monitor を作成し、戦術が 100 万回 Event を受け取るまでに要した時間を測定した。また、Event 通知に要する時間は Event の通知先となるオブジェクトの数などに依存する。そこで Event の通知先のオブジェクトの数を 1, 10, 100 個と増やした場合の時間の測定も行った。その結果を表 3 に示す。この表は、Event の通知先のオブジェクトの数に対する、Event 発生 1 回あたりの通知時間を示す。

Event の通知に要する時間は通知先のオブジェクトの数が増えるにしたがって増大していく。本実装では

Event はすべてのオブジェクトに通知されるのではなく、Monitor によってフィルタされ、Event に興味を持つオブジェクトに対してしか通知されず、Event 通知におけるオーバーヘッドを減らすことができるようになっており、その効果がこの表より見てとれる。すなわち、Event を通知するべきオブジェクトを絞ることにより、効率が良いイベント通知が実現できたと言える。

また、たとえ 1,000 個のオブジェクトに対して Event を通知することを考えても約 6 ms のオーバーヘッドであり、戦術の実行に要する時間を考えた場合に十分に小さい。

最後に今回の実験で確認していない電池の残量などの環境変化についても、Observer を実装する際に正しく環境の変化を検出できることを確認している。よって、ノート型計算機における電池の状態などについても適応できる。

5.2 応用例

本研究では適応を行うための戦術として他サーバへのオブジェクトの移送、またディスクへのオブジェクトのスワップアウトを実装した。また環境の変化として CPU の使用率や電源残量などのハードウェアの情報を検出する Monitor を実装し、実際に環境に対して適応できることを確認した。このような環境への適応を行えるシステムの利用することで以下のような例が実現できると考えられる。

- 使用できるメモリの量に応じて時間的計算量を優先したアルゴリズムから、空間的計算量を優先したアルゴリズムへ切り替える。
- バグの修正などを行った新しいバージョンのプログラムへ実行中に切り替える。
- 通信先に応じて通信路の暗号化などを行う。
- 環境の信頼度が低い場合にレプリカを作成する。

6. 考察

6.1 関連/類似研究

実行環境の動的な変化に対して適応するための言語として LEAD++⁶⁾ がある。LEAD++ ではメソッドの集合から、実行環境の条件に合ったメソッドを選び出し、実行するという形で動的な適応が実現されている。環境への適応をどのレベルでとらえるかという違いが存在するが、本研究でのアプローチではユーザコードと適応のためのコードが分離されているという

ノート型計算機で実行されている場合や、無線 LAN を用いて接続されている場合など

利点がある。

また、OSでの適応の例として Tiger⁷⁾ がある。Tiger ではスレッドのスケジューリングやオブジェクトの永続化などをメタオブジェクトとして表現し、必要なメタオブジェクトの選択によって実行時の振舞いを目的に合わせて適応させることが可能になっている。何に対して適応させるかが違うが、Tiger でも適応のためのコードはユーザコードに埋め込まれる。

動的なソフトウェアの再構成を行う技術としては ASX⁸⁾ がある。ASX では dynamic link やオブジェクトどうしの組合せの変更などによる適応が可能なフレームワークが提案されているが、基本的に管理者や管理ツールといったものからの指示によって適応が行われるという点で、本研究とは異なる。

また、エージェントでの適応、拡張を自動的に行うための研究として Flage⁹⁾ があげられる。Flage ではエージェントが移動する際に新しいメソッド定義を得ることで拡張などが行われる。環境に合わせるための適応ではなく、必要なメソッドを得るために環境を移動していくという点で本研究とは異なる。

環境変化の検出を行うための研究としては Environment Server¹⁰⁾ や CM1¹¹⁾ などがあげられる。Environment Server では様々な環境の情報を同じインタフェースを用いて操作できるようにすることで、取り扱う環境情報の追加などの際のコストを減らすというアプローチが取られている。Environment Server ではオブジェクト間の結合を変更する手法で環境への適応が行われている。本研究でのアプローチでは Event オブジェクトによって環境変化の情報を同じインタフェースで扱える。また、適応のコードが分離されているという利点が存在する。

CM1 では実行環境の情報を抽象化して、アプリケーションが望むレベルの情報を取り出すためのフレームワークが提案されている。この環境情報を抽象化する、加工するという手法は情報の正規化などを行ううえで重要だと思われる。

6.2 今後の課題

6.2.1 実行環境の状態の正規化

早急に解決すべき問題として実行環境の情報の正規化がある。本研究で実装した Observer では検出した実行環境の情報をそのまま Monitor へ渡すようになっている。しかし、たとえば CPU や、ネットワークの使用率などを考えた場合、使用しているハードウェアなどの違いによって、その値の評価が異なるはずである。高速な CPU と、それよりも遅い CPU があつた場合に、同じ使用率であっても後者の CPU 上のオブ

ジェクトは移送したほうがよい場合などが考えられる。

このようなハードウェアの違いに対応するために、情報の正規化を行う必要がある。また、将来的にどのようなハードウェアが出現するか予測することは困難なため、このような正規化のための情報はプログラムとは別の形で記述する必要がある。

そこで Observer によって検出された環境の情報から必要なレベルの情報へ変換するためのフィルタとなるオブジェクトを導入し、このフィルタの動作を決定するためのパラメタを XML による戦略記述ファイルに記述することを考えている。これによって正規化を行うことが可能になり、また必要なパラメタをプログラムから分離することが可能になる。また、フィルタを導入することで後述する Event の複合なども可能になると思われる。

6.2.2 Event の複合

現在の実装では XML による戦略の記述において Event のマスクを提供する方法だけである。Event マスクとして指定されている変化のいずれかが発生した時点で Event が送られる OR による指定しか提供していない。

しかし Event の種類や、それに対する戦術によっては複数の環境の情報を元に行うほうがよいと思われるものも存在する。たとえば、ネットワークの負荷が高く、かつ CPU の負荷が高い場合にはオブジェクトの移送という戦術をとるよりも、ディスクへのスワップといった戦術のほうが効果的である。

このような複数の Event の組合せを指定できるようにすることは、戦略の種類を広げ、より多くの状況に適応できるようになると考えられる。そのために、この Event の組合せの実現も解決しなければならない問題である。

7. おわりに

本研究では Juice 言語を拡張し、電源の異常などの環境の変化が発生しても、その変化に対してオブジェクト単位で適応できる適応型分散オブジェクト指向環境を実現した。本論文では、その設計や実装について述べた。

適応型分散オブジェクト指向環境を実現するために XML による戦略の提供と、Monitor による環境の監視というモデルを導入した。これにより、実行環境の変化に対して適応することが可能になり、また XML による戦略の記述によりアプリケーションの管理者が環境にあわせた戦略を記述することが可能になった。

本実装では Linux のみの実装を行ったが、他の環境、

たとえば FreeBSD などでも Observer は容易に実装できる。また、何も出力しない Observer を実装することで、アプリケーションは環境変化への適応のない最低限の動作が可能であるので、たとえ Observer の実装が困難な OS 上であってもアプリケーションの実行を妨げるものではない。

参 考 文 献

- 1) OMG: *The Common Object Request Broker Architecture Specification 2.2* (1998).
- 2) Oda, K., Tazuneki, S. and Yoshida, T.: The Flying Object for an Open Distributed Environment, *ICON-15*, pp.87–92 (2001).
- 3) 小田謙太郎, 和田智仁, 吉田隆一: ソースコード変換を用いた新しい分散オブジェクトアーキテクチャの提案, 情報処理学会シンポジウムシリーズ, Vol.99, No.18, pp.128–132 (1999).
- 4) Hirano, S.: *HORB Home Page*, <http://horb.etl.go.jp/>.
- 5) Sun Microsystems: *Java Remote Method Invocation Specification 1.4* (1997).
- 6) Amano, N. and Watanabe, T.: LEAD++: An Object-Oriented Reflective Language for Dynamically Adaptable Software, *OOPSLA '98 WORKSHOP*, No.13, pp.91–95 (1998).
- 7) Zimmermann, C. and Cahill, V.: It's Your Choice — On the Design and Implementation of a Flexible Metalevel Architecture, *Proc. Int. Conf. on Configurable Distributed Systems*, IEEE, Annapolis, Maryland (1996).
- 8) Schmidt, D. and Suda, T.: An Object Oriented Framework for Dynamically Configuring Extensible Distributed Communication Systems, *IEE/BCS Distributed Systems Engineering Journal (Special Issue on Configurable Distributed Systems)*, Vol.2, pp.280–293 (1994).
- 9) Kumeno, F., Ohsuga, A. and Honiden, S.: Autonomous Adaptation by Mobile Agent and Thesaurus, *IEICE TANS. INF. & SYST.*, Vol.E83-D, No.4, pp.679–690 (2000).
- 10) Nakajima, T., Aizu, H., Kobayashi, M. and Shimamoto, K.: Environment Server: A System Support for Adaptive Distributed Applications, *Lecture Notes in Computer Science*, Vol.1368, pp.142–157 (1998).
- 11) Tateoka, T., Sanuhara, H., Teraoka, F. and Tada, Y.: CM1: Communication Monitor for Applications Adaptive to Execution Environments, *IEICE TANS. INF. & SYST.*, Vol.E83-

D, No.5, pp.1020–1027 (2000).

(平成 13 年 9 月 14 日受付)

(平成 14 年 11 月 5 日採録)

推 薦 文

ネットワークの分散オブジェクト環境で、バッテリー残量の変化などによる危険回避や CPU の使用率による計算負荷変動などに対応し、戦略によってオブジェクト移送を行う点を評価し、推薦に値すると認めた。

(火の国情報シンポジウム 2001 プログラム委員長 岡田 直之)



加藤 健士 (学生会員)

2001 年九州工業大学情報工学部知能情報工学科卒業。現在、同大学大学院情報工学研究科博士前期課程在学中。分散オブジェクト技術、適応型分散オブジェクト指向環境等に

興味を持つ。



小田謙太郎

1999 年九州工業大学情報工学部知能情報工学科卒業。2001 年同大学大学院情報工学研究科博士前期課程情報科学専攻修了。現在、同大学院博士後期課程在学中。分散オブジェクト技術、適応型分散オブジェクトシステム等に興味を持つ。

興味を持つ。



吉田 隆一 (正会員)

1982 年慶應義塾大学工学部電気工学科卒業。1987 年同大学大学院工学研究科博士後期課程電気工学専攻修了。工学博士。同年九州工業大学情報工学部知能情報工学科助手。

1990 年同助教授。1993 年から翌年にかけてオレゴン科学技術大学院大学において客員研究員。2002 年九州工業大学大学院情報工学研究科情報創成工学専攻教授。オブジェクト指向計算、分散計算に興味を持つ。日本ソフトウェア科学会、人工知能学会、IEEE、ACM 各会員。