マルチメディア通信と分散処理 52－33
（1991. 9. 27）

分散システムにおけるソフトウェア統合環境

宮本衛市 原田康徳 渡辺慎哉
北海道大学工学部情報工学科

あらまし 分散システム上のハードウェアおよびソフトウェア資源を利用して、大規模なシステムが構築するためには、きわめて複雑で、注意深いプログラミングを必要とする。我々は非均質で分散された資源をオブジェクトとして統一的に捉えることにより、安全で柔軟なプログラミング環境（Kamui 環境）を開発している。そこでは、プログラマはネットワークを意識することなく、ネットワーク内にある様々なハードウェアおよびソフトウェア資源を利用することができることを目指しており、したがってソフトウェアの再利用性も高められるものと考えている。本論文では、このような環境を構築するための、いくつかの基本的な枠組について述べる。

Integrated Software Environment in Distributed Systems

Eiichi MIYAMOTO, Yasunori HARADA, Shin-ya WATANABE
Department of Information Engineering,
Faculty of Engineering,
Hokkaido University
Kita 13 Nishi 8, Kita-ku,
Sapporo 060, JAPAN

Abstract It needs very complex and careful programming to develop large-scale systems, utilizing hardware and software resources in a distributed system. We are now developing a safe and flexible programming environment (called Kamui Environment), considering heterogeneous and distributed resources as objects. In the environment, we intend programmers to be able to utilize various hardware and software resources in the network, and enhance the reusability of software. In this paper, we describe some frameworks to establish such an environment.

1 はじめに

様々な計算機がネットワークを介して接続され、各計算機が所有するハードウェアあるいはソフトウェア資源を駆使した分散システムが構築されるようになってきた。例えば、共有プリンタ、共有ファイルシステム、電子メイルシステム、電子フォーラムなどはすでに実用化されている。最近は、距離的あるいは時間的に離れた人間同士が、計算機ネットワークを介して共同作業を行なう環境（ＣＳＣＷ）の研究・開発が盛んであり、対人間の問題も重要であるが、システムの大規模化に伴い、対計算機ネットワークに関するソフトウェア開発のプラットフォームの構築も緊急の課題となっている。

一般に、計算機ネットワーク上の計算機上には必ずしも同一ではないＯＳあるいはプログラミング言語が設置されており、そのため各マシーンで開発されたソフトウェアの互換性は必ずしも保証されず、ソフトウェアの再利用性を損なっている。また、各計算機上で開発されたソフトウェア間で協調動作をするような場合の支援環境は、まだ十分とはいえない。例えば、ユーザインタフェースマシーンで採取した音声データを認識し、その自然言語処理をLispマシーンで行ない、それを基にファイルマシーン上のデータベースを検索し、その結果を３次元画像としてユーザインタフェースマシーン上に表示するような一連の処理は、現在のハードウェア能力で十分実現可能であるが、これに必要なソフトウェアを計算機ネットワーク上で実現するためには、きわめて複雑なプログラミングを必要とする。さらに、上で挙げたようなソフトウェアは、それらがシステムの一部になることを意識せずに作成されており、そのようなソフトウェアをできるだけ自然に統合することのできるソフトウェア基盤は不可欠である。

我々は非均質で分散された資源をオブジェクトとして統一的に捉えることにより、安全で柔軟なプログラミング環境（Kamui環境）を開発している[1]。そこでは、プログラマはネットワークを意識することなく、ネットワーク内にある様々なハードウェアおよびソフトウェア資源を利用することができることを目指しており、したがってソフトウェアの再利用性も高められるものと考えている。

本論文は、このような環境を構築するためのいくつかの基本的な枠組について述べたものである。以下、２章では環境構築の基本となるオブジェクトモデルについて、３章ではオブジェクトが受信するメッセージの到着順序を制御する機構について、４章では、開放分散システムについて述べる。

2 オブジェクトモデル

Kamui環境では、ソフトウェア資源をすべてオブジェクトとして捉える。しかも、できるだけ多くの資源を取り込むために、単純なモデルを採用する。すなわち、ここで考えているオブジェクトの基本動作は、非同期メッセージを受け付け、それをキューに入れ、オブジェクトはキューから順次メッセージを取り出して、対応する処理を行なう。計算モデルとしてはActorモデルであり、並列計算モデルKamui88の流れを汲んでいる。もちろん、より複雑なオブジェクトモデルの参加を拒んでいるわけではなく、Kamui環境に参加できるための最低限の機能を要求しているのであり、これを満足しているオブジェクトをすべてKamuiオブジェクトと呼ぶ。しかし、非均質で分散されたオブジェクトを上記のようなほとんど制約のないモデルのみで記述したのでは、正しいオブジェクトの振る舞いを記述するために、プログラマにかえって意味的な負担をかけることになり、一方処理系も動的なチェックに持ち込まれるため、過重な負担を強いられることになる。そこで、我々はオブジェクトの記述モデルに型の概念を導入する。

2.1 オブジェクトの型機構

Kamui環境のねらいはネットワーク内のソフトウェア資源の部品化であり、それら資源の接続可能性を高めることである。そのためには、オブジェクトの仕様と実現を分離することであり、その仕様を表現する枠組みとして型を考える。型はネットワーク内のすべてのオブジェクトから見えなければならず、したがってオブジェクトの記述言語には依存しないものでなければならない。オブジェクトの型に関する多くの提案は、型をクラスとして扱うものであり、そこではクラス間の継承関係が重要となってくる。しかし、異言語間で継承を考えることは不可能であるから、クラスをネットワーク内でトランスペアラントな仕様の枠組みとして捉えることはできない。

仕様として型を考えるときのもう一つの問題点は、オブジェクトの接続可能性に関することである。部

品化が有効に行なわれるようになると、大部分のプログラミングは既存の部品を接続することで用が足りてしまう。ユーザインタフェースにおけるツールキットなどはその良い例であり、ボタンやメニューなどの部品を適当な値に初期化した後、別のウインドウなどの部品として張り付けられる。このような部品間の接続が安全に行なわれるためには、メッセージを受け付ける側のオブジェクトの型ばかりでなく、メッセージを送り出す側のオブジェクトが、どのようなメッセージを送り出すのかが仕様上明示されている必要がある [2]。

以上のことから、われわれはオブジェクトのクラスを型のベースとはせず、メッセージの集合をベースに型を考える。大規模なシステムになると、メッセージ名が大局的に一つの意味に限定することは困難になってきて、同一名が多重の意味で使われるようになるのは避けられなくなる。そこで、メッセージ名の同一性を保つことのできる範囲を限定し、これをビューと名付ける。ビューはいくつかのメッセージからなり、オブジェクトがいくつかの仕事をこなすのであれば、それに対応したビューをもつことになる。これを多重ビューと呼び、各ビューにはネットワーク内でユニークな名前を付与するものとする。

さて、あるオブジェクトがいくつかのビューを持つのであれば、メッセージを発信するオブジェクト側でどのビューに関与しているかを明示させれば、発信されるメッセージが発信側および受信側でその整合性を確認することができる。そこで、オブジェクトの型に受信の型と発信の型を区別して考え、オブジェクトのモデルに取り込むものとする。前者はオブジェクトが受け付けるビューをすべて網羅したものになるのに対し、後者はオブジェクトがどのビューでメッセージを発信するのかを仕様で宣言するものである。

2.2 オブジェクト記述言語

上記のモデルを組み込んだプログラミング言語として、Kamui-C を開発した [3]。これは C++をベースにして、C++へプリコンパイルするときに Kamui 環境のオブジェクト群から情報を得て、C++のプログラムを生成する。次いで、C++コンパイラでコンパイルされ、Kamui-C の実行時ライブラリとリンクされて実行可能なプログラムとなる。以下に、Kamui-C のプログラム例を示す。

```
KamuiC
include "class.h"
defprotocol pCounter {
  send set(int aValue);
  send up();
  send down();
  int get();}
defprotocol pDraw {
  send move(int aValue);
  send circle(int aValue);}
defclass cCounter
        [pCounter, ~pSignal signal] {
  int value;}
defmethods cCounter pCounter {
  send set(int aValue) {value = aValue}
  send up() {value++; if (value % 10 ==0)
                    {signal.sig();};}
  send down() {value--;}
  int get() {return value;}}
```

冒頭の行は以下が KamuiC で記述されていることを示し、次の行で KamuiC ファイルを挿入している。defprotocol でビュー（言語上ではプロトコルと呼んでいる）を宣言しており、環境のプロトコルデータベースとの比較、更新、登録などが行なわれる。send は非同期メッセージの受理を意味し、その他の型（ここでは int）は同期メッセージを意味し、その戻り値の型を指す。defclass はオブジェクトのクラスを宣言し、このクラスがプロトコル pCounter を受理することを示し、同時にインスタンス変数 draw を通じてプロトコル pDraw でメッセージを発信することを宣言している。defmethods はプロトコル単位でメソッドを定義するものとし、ここではクラス cCounter のプロトコル PCounter のメソッドを記述している。

上記のプログラムに、さらに次のプログラムを追加したとしよう。

```
defprotocol pClock {
  send reset();
  send up();}
defclass cClockCounter
        [pClock] cCounter {}
defmethods cClockCounter pClock {
  send reset() {pCounter_set(0);}
  send up() {pCounter_up();}}
```

ここで、クラス cClockCounter はクラス cCounter を継承しているので、2つのビューを持つことになる。メソッドは新しく宣言したプロトコルに関するものだけを記述し、その中ではプロトコル pCounter のメソッドを再び呼んでいる。

オブジェクトの生成は次のようにして行なう。

```
KamuiCMain {
  gDB.Add("clock-counter",
          cClockCounter new);}
```

KamuiCMain は C++の main() に相当し、オブジェクトを公開したいときには、データベースオブジェクト gDB に名前と対でオブジェクトを登録する。したがって、このオブジェクトを別のプログラムからは、次のように呼び出すことができる。

```
KamuiCMain {
  pClock clock;
  clock = gDB.Get("clock-counter");
  clock.reset();
  clock.up();
  pCounter(clock).down();}
```

ここで、pCounter(clock) はオブジェクト clock をメッセージプロトコル pCounter で使うことを意味し、オブジェクトがそのプロトコルを知っているかどうかの検査が動的に行なわれる。

Kamui-Lisp は Lisp 上のオブジェクト記述言語である。前述の例を Kamui-Lisp で記述すると次のようになる。

```
(defKamuiClass cCounter (pCounter)
  (value))
(defKamuiMethods (cCounter pCounter)
  (set (v) (setq value v))
  (up () (incf value))
  (down () (decf value))
  (get () value))
```

3 分散オブジェクト指向モデル

Kamui 環境では、分散されたオブジェクトは受理したメッセージを並行に処理していく。一般に、メッセージの発信と受信にはネットワークに依存した時間遅れが生じ、同一オブジェクト間であっても、発信されたメッセージが発信順で受信されるとは限らない。それゆえ、メッセージの到着順を制御するためには、オブジェクト間で同期をとる手続きを記述しなければならず、これはプログラミングを複雑にしてプログラマに大きな負担をかけることになることのみならず、不必要に同期メカニズムを導入して、並行処理の能率を下げる事態を生じかねない。そこで、ここではオブジェクトから発信される複数のメッセージの到着順序を宣言的に表現する分散オブジェクトモデルを提案する [4]。このモデルは Actor 形式をベースにして、メッセージの標的となるオブジェクトにおける到着順序を半順序関係を基づいて発信側で宣言するもので、その効果は次々と伝搬されるものとするので、複雑に絡み合ったメッセージの到着順序を構造的に表現することが可能である。

3.1 メッセージ群の構造化

複数のメッセージの到着順を構造化するためには、それらの間の半順序関係を表現できなければならない。そこで、メッセージの関係を逐次関係と並列関係をプリミティブな関係とし、これをもとにメッセージの到着順を構造化する。メッセージを $m_1 \ldots m_n$ とすると、逐次関係は

$$(seq\ m_1 \ldots m_n)$$

と表現される。これは、これらのメッセージが送信された場合、この順序で到着すべきこと（処理されるべきこと）を宣言したものである。また、並列関係は

$$(par\ m_1 \ldots m_n)$$

と表現され、これらの到着順には順序関係がなく、したがって受信順で処理して構わないことを意味している。より複雑な関係は上の2つの関係を組み合わせて、例えば次のように表現する。

$$(seq\ m_1\ (par\ m_2\ m_3)\ m_4)$$

そこで、メッセージを送信するとき、到着順が保持されなければならない標的オブジェクトを指定して、オブジェクトのメソッド内での送信手続きを Lisp 風に表現すると次のようになる。

$$(send - with - arraival - order$$
$$: at\ a_d$$
$$: structure\ (seq\ m_1\ (par\ m_2\ m_3)\ m_4)$$

ここで、2つの生産者オブジェクトが交互に1つの消費者オブジェクトにメッセージを送る例を示す。

```
(defunc producer (another ...)
 (behavior
  :name 'initialize
  :args '(prod)
  :body '(setq another prod))
 (behavior
  :name 'proceed
  :continuation 'consumer
  :body
   '(let ((data ( ... )))
     (send-with-arrival-order
      :at consumer
      :structure
       (seq (message
             :target consumer
             :function 'put
             :args (list data))
            (message
             :target another
             :function 'proceed
             :continuation consumer)))))
```

生産者オブジェクトはメッセージ initialize により、もう一方の生産者オブジェクトを知り、メッセージ proceed によりデータを送信する。このとき、1つ目のメッセージは消費者オブジェクトへ直接送られ、2つ目のメッセージはもう一方の生産者オブジェクトに、consumer を continuation として送られる。順序保存は consumer に対して指定されているので、1つ目のメッセージは2つ目のメッセージに起因するメッセージよりも consumer には先に到着する。消費者オブジェクトは受け取ったメッセージ put をただ処理すればよい。

3.2 到着順序の保存メカニズム

上で述べたモデルを実現するための、到着順序の保存メカニズムについて述べる。これは基本的にはメッセージに順序に関する情報を付加し、標的となるオブジェクトでメッセージの順序を認識できるようにしようとするものである。したがって、メカニズムは、到着順序の宣言に基づいてメッセージに順序情報を付加する部分と、受け取ったメッセージの順序情報を基に、メッセージの待機と処理を決める部分である。

まず、メッセージを次の3つ組 (A,T,f) で構成する。ここで、

A: 標的オブジェクトの集合

T: 当該メッセージの前に処理されるべきメッセージの識別子の半順序集合に基づくリスト

f: 当該メッセージの本体

メッセージの到着順序は、一般的には半順序となり、したがって受け取る側でメッセージを送信元で意図した半順序に構成できればよい。そのため、T は必要なメッセージのコンテクストを伝える内容となる。

4 開放分散システム

計算機ネットワークが展開していくにつれ、分散環境が大規模化し、これを一つの分散システムとして捉えることが困難になってきた。すなわち、システム全体にわたってグローバルなオブジェクトの存在を仮定することは困難になってきて、未知のオブジェクトが存在し、それゆえ異なったオブジェクトに重複した命名をするなどの事態を避けられなくなってきた。そこで、システム全体をグローバルに考える代りに、グローバルに考えられる領域を設定し、この範囲を越えて他の領域へアクセスする手段を用意するようなモデルを考える。これをドメインポートモデルと呼び、名前の有効範囲であるドメインと、隣接したドメインを接続するポートによって、開放分散システムを把握しようとするものである [5]。

4.1 ドメインポートモデル

ドメインは名前がユニークに保たれるシステムの領域の最小単位である。その中では名前は集中的に管理され、そのユニークさが保証される。ドメインの大きさは任意であって、空間的および時間的に管理が容易な範囲に限定される。ドメイン間には階層を考えず、ポートによって網状に接続されるものとする。ポートには各ドメインでの名前が付けられており、ポートを指すことによって隣接するドメインにある名前に言及できる。さらに離れたドメインにある資源へは、そこへたどり着くドメインのリンク、すなわち各ドメインのポートを接続してアクセスする。このように、他のドメインにある資源に対しては、そこへ至るドメイン間の経路、すなわちポートの接続パスによってその名前とする。したがって、各資源の名前は呼ばれるドメインによって異なり、ま

た同じドメインからでも接続するドメインの経路が異なれば、異なった名前となる。図1に示すシステムでは、ドメインAからみて、ドメインDにある資源dは[13.14.4]のほか、[13.12.9.4]として参照することができる。ドメインAからドメインDにある資源dに[13.14.4]という名前でメッセージを送信すると、ドメインBで[14.4]と名前が変わり、ドメインDで[4]となって所定の資源に届けられる。ポートはこのような名前の付け替えの機能を持っているものとする。

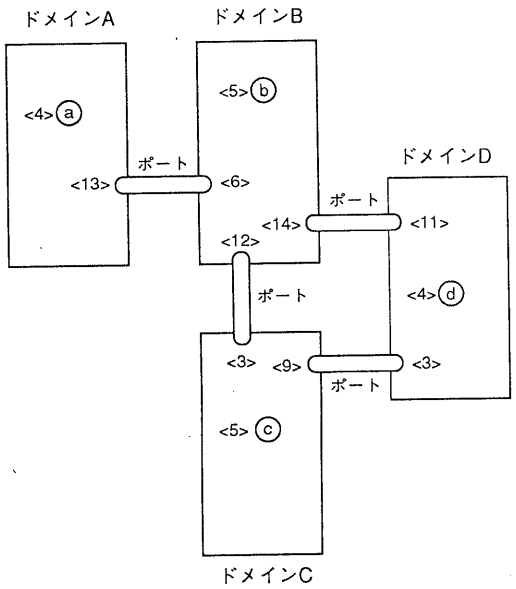

図 1: 分散システムのドメイン分割

4.2 開放分散系の問題点

ドメイン間にループがあれば、他のドメインにある資源を呼ぶとき、複数の名前（経路）が考えられるが、その中で最も短い名前を選ぶほうが能率がよい。メッセージを送るさいに通過するドメインが最も少ないからである。しかし、ドメイン間の接続情報自身、開放分散の性質があり、絶えず変化していて、十分な情報は行き渡っていないと考えるべきである。それ故、開放分散系のユーザは、とにかく必要とする資源を呼ぶことができれば、その名前の最適化は系自身が行うことが望ましい。ループを構成するドメインのポートには、経路が最適になるような置き換え規則を持たせることで解決できよう。一度、名前の最適化が行われば、それを送信元へ伝えれば、以後は最適な名前で呼ぶことができる。

名前の自動的な最適化機構は、オブジェクトマイグレーションや、ドメインまたはポートの変更にも動的に対応することができる。すなわち、変更された経路情報を適当なポートの書き換え規則として与えておけば、そのポートにたどり着きさえすれば変更された情報は補うことができ、最適化されて送信元へ教えることができるからである。

5 おわりに

本論文では、分散システムにおけるソフトウェア環境を構築するためのいくつかの問題点について言及したが、もちろん問題点は他にも山積している。なかでも、今後分散システムが大きくなっていくにつれ、開放分散としてシステムを捉えることが肝要になってくると考えられ、今後大いに研究を進めていくべき方向と考えている。

参考文献

[1] 原田、宮本：送出メッセージの型宣言機能に基づいたオブジェクトの部品化について、 日本ソフトウェア科学会第7回大会論文集、1990, pp.249-252.

[2] 原田、浜田、渡辺、宮本：分散型環境 Kamui について、情報処理学会研究報告、vol.90-SYM-57-5, 1990.

[3] 原田、渡辺、三谷、宮本：イベントグループを導入した並行分散型オブジェクト指向言語 Kamui-C について、日本ソフトウェア科学会第6回大会論文集、1989, pp.413-416.

[4] 渡辺、宮本：メッセージ群の構造化に基づく分散オブジェクト指向モデル、日本ソフトウェア科学会第7回大会論文集、1990, pp.85-88.

[5] 原田、浜田、渡辺、宮本：開放分散に対応した Kamui 環境、情報処理学会研究報告、vol.90-PL-27-14, 1990.