記　号　処　理 71－2
（１９９３．１１．１９）

パッケージシステムの分散環境への拡張

泉　達也　　友永佳津子

日立東北ソフトウェア　　日立製作所 ソフトウェア開発本部

ネットワーク上に分散した言語処理系間でRPC(Remote Procedure Call)を行う方法として、パッケージシステムを拡張した分散パッケージを提案する。従来、ネットワークアプリケーションの記述には他言語インタフェースによるプログラムインタフェースを記述しなければならず、アプリケーションプログラムの抽象性・保守性が低下していた。

ネットワーク分散への対応は、Common Lisp などの処理系で備えているパッケージシステムに分散資源を指定する記法を追加することで実現した。さらに、論理型プログラミング言語Prologへの分散パッケージの適用について考察した。

分散パッケージにより、ネットワークに依存したプログラミングインタフェースの記述を減らし、アプリケーションプログラムの記述性を向上させることが可能となる。

Extention of Package System for Distributed Processors

TATSUYA IZUMI* and KAZUKO TOMONAGA**

* Hitachi Tohoku Software, Ltd.　　2-6-10, Honcho, Aoba-Ku, Sendai, 980 Japan
**Software Development Center, Hitachi, Ltd.　549-6, Shinano-cho, Totsuka-ku, Yokohama, 244 Japan

This paper presents 'distributed package', an extention of package(module) system for remote procedure calls (RPC) between distributed language processor. The concept of network-distributed package is addition of network-related annotation into existing package syntax that is included such as Common Lisp. We also have considered Prolog-based distributed package.

To introduce distributed package into current symbolic programming language, we can decrease the complexity of network programming interface drastically.

1.はじめに

論理型プログラミング言語Prologは、ホーン節をベースにした宣言的なプログラミングができる。そのため知識処理分野だけではなく、グループウェアなどの高度な分散アプリケーションを抽象性を保ったまま記述できるスクリプト言語としての素地を備えている。

例えば、電子メールを媒介にプログラムを実行させるActive Mailや、複数のクライアント(=ユーザ)から予定をassertさせ、サーバ側で制約解決を行い会議日程を決める分散スケジューラのような分散アプリケーションが考えられる。これらのアプリケーションでは、動的な知識や構成情報の変更を伴うため、LISPやPrologをエンジンとすることによりフレキシビリティを高めることが可能となる。

記号処理言語で分散アプリケーションの記述をしようとする際に、現状ではネットワークインタフェースのプログラミングに多大な労力を要している。現在多くの処理系では、Cをはじめとする他言語インタフェースにより

・ダイレクトなソケットインタフェース

・プログラムを文字列(string)として送受信

などの方法で行われている。

我々は本来の記号処理言語の利点である抽象性を生かした記述手段の提供を検討した結果、パッケージの概念を拡張した分散パッケージにより実現できる見通しを得たので報告する。

我々は現在DEC-10 Prolog準拠でパッケージ機能を追加したPROLOG E2に分散パッケージを実装し、その評価を行っている。

2. パッケージ

(1)パッケージ

大規模プログラムでのシンボルの衝突を回避する手段として、Common Lisp[1]ではパッケージシステムを用いてこの問題を解決している。

全てのシンボルは名前空間であるパッケージに属し、通常は自身のパッケージ(カレントパッケージ)内のシンボルへのみアクセスが可能である。

パッケージ外のシンボルへのアクセスを行うためにはパッケージ修飾子 (<パッケージ名>+":") を用いて指定を行う。例として、

```
(bar X)
```

は自パッケージ内の関数 bar の呼び出しを、

```
(foo:bar X)
```

はパッケージ foo 内の関数 bar の呼び出しを示す。

パッケージの概念を導入することにより、プログラムに透過性を持たせることが可能となる。また、パッケージを単位として、機能的にまとまったプログラムの部品化・抽象化が可能となる。

(2)Prologのパッケージシステム

PrologでもCommon Lisp同様のパッケージシステムを構築している処理系がある。ISO Prolog[2]でもモジュール(Modules)として、パッケージ機構が提案されている。

```
% パッケージの定義
:- define_package(foo).
:- package(foo).
bar(X) :-
        baz(X).
baz(1).
:- package_end.
```

図1 PROLOG E2のパッケージ機能

図1にPROLOG E2のパッケージシステムの概要を示す。

define_package/1 述語でパッケージを生成し、package/1 宣言でパッケージの開始を宣言する。package_end/0 宣言までの間に記述された bar/1, baz/1 の各述語がパッケージ foo 内に定義される。

パッケージ foo 内の述語を他のパッケージから実行、参照(パッケージ間呼び出し, inter-package call)する場合は、中置演算子 : を用いて

```
?- foo : bar(X).
```

のように指定する。

3.分散パッケージ

(1)RPC(Remote Procedure Call)

分散処理のプログラミングインタフェースとしては、RPCがよく知られている。

RPCは単一プロセス内の関数呼出をネットワーク拡張したもので、手続き(関数)単位にコントロールの受け渡しを行う。すなわち、RPCのプログラミングインタフェースは通常のプロシジャコールとほぼ同じである。

OSF/NCA RPC[3] では、ターゲットとなるリモートプロシジャのバインディングを行い、その結果得られるハンドルを用いてプロシジャの遠隔呼び出しを実行する。

我々が今回目標とするのはRPCベースの言語処理系の結合である。図2で示すように、複数の処理系をサーバとクライアントとして位置付け、クライアントからサーバ(中の関数又は述語)をRPC的に実行する。

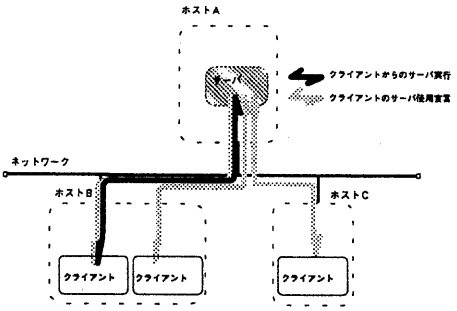

図2 Prologの分散処理系結合

(2)分散パッケージによる名前空間の結合

本節では、2.(2) をベースにした分散処理系のパッケージ拡張を検討する。

分離した名前空間という点では、同一プロセス内のパッケージと別プロセスとして動作する処理系のパッケージを等価と考えることができる。

我々は、分散処理系に対応したパッケージ機能として、複数の分散処理系(インタプリタ)で、相手先(サーバ)パッケージの仮想パッケージを呼出側(クライアント)に用意し、通常のパッケージ間呼び出しの延長として実現する方式を検討した。

以下、分散処理系への拡張を行ったパッケージ(機能)を分散パッケージと称する。

また、サーバで定義するプログラムの実体があるパッケージをサーバパッケージ、クライアント側で定義する仮想パッケージをクライアントパッケージと呼ぶ。

(3)分散パッケージの構成

分散パッケージを構成するサーバパッケージとクライアントパッケージを図3のように定義する。

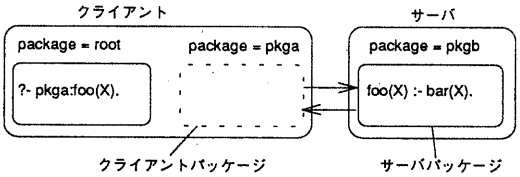

図3 分散パッケージ

・サーバパッケージ

サーバパッケージは実際の実行主体となる、実体を持つパッケージである。通常(スタンドアロン)のパッケージに対しサーバパッケージの宣言を行うことにより、サーバパッケージとしての使用が可能となる。

・クライアントパッケージ

クライアントパッケージは実体を持たず、サーバパッケージの結合情報のみを持つ。この例では、pkgaとpkgbが結合されている。クライアントPrologで、クライアントパッケージへのパッケージ間呼び出しが発生すると、対応するサーバパッケージへ呼び出しが渡され、実行される。

図4 にPrologでのサーバ側の定義例を示す。本例では、ホスト名 hosta 上で稼動する Prolog のサーバパッケージ foo を定義し、サーバパッケージとしての使用宣言を行っている(define_server_package/1)。

```
%% host=hosta (server)
:- define_package(foo).
:- package(foo).
bar(X) :-
        baz(X).
baz(1).
:- package_end.
:- define_server_package(foo).
```

図4 サーバパッケージの定義

図5にクライアントパッケージの定義例を示す。define_client_package/1 宣言によりクライアントパッケージ rfoo を作成し、server_package/2 宣言でサーバパッケージ fooとの対応付けを行う。

```
%% host=hostb (client)
:- define_client_package(rfoo).
:- server_package(host(hosta),
                  package(foo)).
```

図5 クライアントパッケージの定義

(4)分散パッケージの実行

クライアントパッケージ rfoo への呼び出しが発生すると、制御はサーバパッケージ foo の存在するホスト hosta のインタプリタへ転送され、サーバパッケージ内で実行される。実行後、結果及び制御はクライアントに転送され、クライアントパッケージの呼出元へ戻される。

(5)パッケージ内シンボルの隠蔽

大規模プログラムの抽象性を保つため、或は第三者の利用によるプログラム参照の有効範囲を制限するためには、パッケージ内の特定のシンボルを隠蔽または公開する機構が必要となる。

Common Lispでは、この解決手段としてexport 関数により、パッケージ内のシンボルを外部から参照可能とするか否かを制御する。exportをPrologで記述した例を図6 に示す。

```
%% host=hosta (server)
:- define_package(foo).
:- package(foo).
:- export bar/1.
bar(X) :-
        baz(X).
baz(1).
:- package_end.
```

図6 分散パッケージへの export の導入

図6 で定義したパッケージ内の述語を他から実行した場合、export宣言された bar/1 の呼び出しは

foo:bar(X). → succ, X=1

となるが、宣言の対象とならない baz/1 の呼び出しは失敗する。

foo:baz(X). → fail

export の機構を分散パッケージに組み込むことにより、よりマクロなレベルでのプログラムの結合を可能にする。

(6)入出力の扱い

サーバパッケージの実行は呼び出し元とは異なるインタプリタで行われるため、入出力をどのように取り扱うかが課題となる。透過性を第一に考えた場合、一個のインタプリタが動作しているように見えるようにするためには入出力も呼び出し元に戻す方法も考えられる。

一方、実際のアプリケーションで、プログラムがサーバで実行されることを意識することによりファイルなどのサーバ側資源を共有することが可能となり、このメリットは大きい。従って、入出力に関しては以下の方針とした。

・入出力はサーバ側のリソースを使用する。
図7の bar/1 中の write 述語による出力は、サーバ側に行 われるため、呼び出したクライアントには表示されない。

また、図7,8 の例では、異なるユーザがサーバのファイルリソースを共有する可能性があ

ることを考慮して、クライアント毎にファイル操作が発生する述語の実行権限を設定できるようにしている。

・サーバパッケージとなるインタプリタでのエラーメッセージはクライアント側に転送する。図8 の baz/2 の実行により発生したエラー (type_error) はクライアント側に転送され、表示される。

```
%% host=hosta (server)
:- define_package(foo).
:- package(foo).
bar(X) :-
        write(X).
baz(X,Y) :-
        Y is X * X.
:- package_end.
:- define_server_package(foo).
```

図7 入出力のあるプログラム

```
?- rfoo:bar('this is sideeffect output').
yes
?- rfoo:baz(a, Y).
type_error.
no
```

図8 入出力のあるプログラムの実行

4. Prologでの分散パッケージの実現

本節では、Prologで分散パッケージを実装する際に留意すべき問題について検討する。

(1)ユニフィケーション

Prologの実行はユニフィケーションという項の構造のパターンマッチが伴う。

```
remote:bar(fact_1, fact_a).
  → succ
remote:baz(X, Y, Z).
  → X = a, Y = b, Z = c
```

図9 分散パッケージでのユニフィケーション

図9 の例で示す通り、従来の(単独の)インタプリタの実行と同様に baz/3 内の変数はサーバ側でユニファイされ、その結果を受け取ることになる。

(2)バックトラック

バックトラックをサーバパッケージで行う際にも同一の処理系内での実行と同様のバックトラッキングの動作を保証する必要がある。

クライアントで図10 の問い合わせを行った際に、(a)(b)で変数のバックトラックが正しく行われなければらない。

```
remote:bar(X) ,  abc(X,Y,Z) , remote:baz(X,Z).
      (a)                          (b)
```

図10 節にまたがるパッケージ呼び出し

本項に関連して、複数のクライアントから実行が行われる際の排他期間の設定も問題となる。

図10(a)の bar/1 呼び出し終了後、他のクライアントに実行させると logical update view が成立しない恐れがある。そのため、(a)の節呼び出し成功後でも、(a)を含む節が exit または fail するまでコントロールを解放せず、現クライアントのコール状態を保存する必要が生じる。

実際に複数のクライアントが同時にサーバをアクセスする場合に、コントロールを解放するタイミングの設定は極めて困難であるため、タイムアウトやバックトラック回数の制限などの対応を併せて行うことが必要となる。

(3)アクセスコントロール

本項では、複数のクライアントからのプログラム実行による、サーバのデータベース更新への対処を検討する。

Prologの場合はassert, retract 系述語によりプログラムの書き換えが発生するため、複数のクライアントからの実行によるデータベースの変更に対する制限を行う機構が必要となる。

例えば、ユーザ毎の
・項単位あるいはパッケージ単位のプログラムの書き換え
・ファイルなどの外部入出力の利用許可

などのアクセスを制御する機能が必要となる。

```
%% host=hosta (server)
:- define_package(foo).
:- package(foo).
:- access(izumi,  heap(write)).
:- access(mori, heap(read_only)).
bar(X) :-
        baz(X).
baz(1).
poi(X) :-
        assert(poi2(X)).
:- package_end.
```

図11 アクセスコントロールの導入

図11の例で、分散パッケージ foo での実行について、
・クライアントユーザが izumi の場合は、パッケージ内の項の追加(assert) / 削除(retract) ができる。
・ユーザ mori はデータベースの更新が禁止されるため、poi/1 の実行でエラーとなる。

(4)動作環境の相違

分散パッケージ間の実行は、複数の独立した処理系で実行されるので、以下のような動作環境の相違による実行結果の違いが発生することが考えられる。

・演算子(オペレータ)
・システムフラグ(文字セット, エラー処理)
・読み込み表(readtable)

これらの機能についても、(3)と同様に変更権限をクライアント毎に設定することにより対処できる。

(5)デバッグ

クライアントパッケージは仮想的なものであるため、クライアント側でのデバッグ対象とするか否か、検討の余地がある。
今回の検討では、分散パッケージのプログラムのデバッグは実体のあるものだけを対象とすることで、分散パッケージの定義されたインタプリタ間の動作の切り分けが明確になると考え、以下のような方針をとった。

・クライアントの実行に関して、サーバ側の述語の実行はデバッグの対象外とする。すなわち、システム述語と同様にサーバ側の述語は「ブラックボックス的に」実行される。

・サーバパッケージのデバッグを可能とする。クライアントからの calling context に依存したデバッグが可能となる。

(6)メッセージ送信型の実行

メッセージ送信型の実行方法をサポートすることにより、同期実行を必要としない、より柔軟な分散処理の記述が可能となる。

```
sample(X) :-
        send(rfoo, remote_clause(X)),
        another_clause(X).
```

図12 メッセージ送信形式の導入

図12の例では、send/2 述語の実行は remote_clause(X) をサーバに転送後、直ちに成功する。
another_clause(X) と remote_clause(X) はほぼ並列に実行される。

5. おわりに

これまで述べたように、分散パッケージを導入することにより、従来のパッケージベースのプログラムをほとんど変更なしに分散処理アプリケーションに適用できることを示してきた。

本論文で示した分散パッケージはネットワークプロトコルへの依存を極力除いている。本論文の記述例では、ホスト名による分散パッケージの特定を行っている。これは現在の大多数のプラットフォームで適用できるものと考えられる。さらにネームサービスなどの機構を利用すれば、より柔軟に資源を対応づけることが可能となる。

同時に、Prolog を例として、分散パッケージの処理系実装に際しては幾つかの留意すべき点があることも示した。

分散パッケージにより、記号処理言語の本来の記述性の良さを生かしたネットワークアプリケーションプロトタイプの作成が可能となったと考える。

参考文献

1) Guy L.Steele Jr. : COMMON LISP THE LANGUAGE Second Edition, Digital Press (1990) [井田昌幸(監訳) : COMMON LISP 第2版, 共立出版(1991)]
2) ISO/IEC JTX1 SC22 WG17 : ISO Prolog Part 2(Modules) - Working Draft 4.1 (1993)
3) Tom Lyons: Network Computing System Tutorial (1991) [平田 真(訳) : ネットワーク・コンピューティング・システム・チュートリアル, トッパン(1992)]