マルチメディア通信と分散処理 66－5
（1994．7．7）

分散ソフトウェア構築のための DOL/C++ クラス ライブラリ

鈴木寿郎　　中澤修
(株) 沖テクノシステムズラボラトリ　(株) 沖電気工業

分散ソフトウェア構築のための DOL/C++ クラス ライブラリ の概要と実装について論ずる。このライブラリは，オブジェクト指向言語 C++ を使ったプロセス間通信のための，抽象化された，柔軟かつ簡単なインタフェースを提供する。

通信は，汎用データ型で表現された値を，各通信者オブジェクトが，遠隔手続き呼び出しと通信サーバ・プロセスを介して送信することで行われる。広報通信を行うこと，遠隔通信者を動的に参照することも容易である。

The DOL/C++ Class Library for Distributed Programming

Hisao Suzuki　　Osamu Nakazawa
Oki Technosystems Laboratory　Oki Electric Industry

We developed the DOL/C++ class library for distributed programming. It provides a high-level, simple and flexible interface for inter-process communications with object-oriented language C++ in distributed environment. The methodology and some implementation issues are described.

For communicating processes (possibly on different machines), each 'correspondent' object in each process sends a message to another. The message contains a text of a general data type, and is carried by the communication-server process using RPCs. Multicast communications and dynamic referencing to remote correspondents (i.e., correspondents in other processes) are also facilitated.

1 はじめに

複数の計算機上の複数のプロセスによる協同的な処理を実現し得る環境が，ごく一般的になりつつある。このような分散処理を実現するソフトウェアを構築する典型的な方法には，次の三つがある [1]。

1. 既存のプログラミング言語からオペレーティング・システムのシステム・コールやライブラリを呼び出す
2. 既存のプログラミング言語と RPC(遠隔手続き呼び出し) 用スタブ生成器を組み合わせて用いる
3. 分散プログラミングを可能とするように設計された新たな言語を用いる

第1，第2の方法は，多くの開発環境で利用可能であり，広く使われている。しかし第1の方法には，普通，その低水準なアプリケーション・インタフェースによって，プログラムの生産性や可読性が低くなるという問題がある。第2の方法は，より高水準なプログラミングを可能としているが，普通，通信相手の決定方法や交信形式が強く固定されており，あらかじめ想定されている形態と異なる通信の実現は必ずしも容易ではない。また，スタブ生成用の特殊な言語と，その既存の言語との関係を，ともに習得しなければならない。一方，第3の方法は，プログラムの移植性，処理系の入手性，そして有用なライブラリ資産の利用可能性などの問題が解決される限りにおいて，理想的な解決を提供し得る。

プログラミング言語 C を上位互換的に拡張した C++ [2] は，実装の詳細を「クラス」という利用者定義型に閉じ込め，それを組み込み型と同様に扱うことを可能にしている。つまり，クラスを定義することによって事実上，言語を拡張することができる。したがって，第1ないし第2の方法によって，第3の方法を近似的に実現することが可能である。

Unix ワークステーション上で著者らが開発している DOL(Distributed Objective Language)/C++ クラス・ライブラリ [3] は，このひとつの試みであり，汎用性のための柔軟さを保ちつつ，プロセス間通信のプログラミングを自明で容易なものにすることを目指している。

本稿では，このライブラリが実現するクラスとそのシステムについて述べる。§2 でプログラマから見たシステム像について概説する。§3, §4, §5 で各クラスとその実装について議論する。最後に §6 でまとめと応用，今後の課題を示す。

2 システムの概要

分散ソフトウェアを構築するときの主要な問題のひとつは，通信端点の設定/通信相手の検索/通信路の確立である。DOL/C++ クラス・ライブラリ (以下，本ライブラリ) では，「通信者」クラスと「伝達者」クラスを定義することによってこの問題の解決を容易化した。また，通信データの符号化/復号化を1種類の汎用データ型に閉じ込めることで，これに関する問題を簡単化した。本ライブラリはプロセス間通信を ONC RPC [4] によって実現しているが，その詳細はプログラマから隠されている。

本ライブラリを利用するプログラムのプロセスの集合は，全体で通信系を構成する。この通信系は，異なるマシン間にまたがることができる。

2.1 通信者

通信系内のプロセスは，「通信者」(correspondent) と呼ばれる通信端点を設けることができる。プロセス間通信は，基本的に，通信者から通信者への非同期的なメッセージ通信として実現される。

ひとつのプロセスで複数の通信者を設けることもできる。プロセス fork 時には，親プロセスから子プロセスへひとつの通信者を譲渡することもできる。

通信者には「名前付き通信者」と「無名通信者」がある。通信系は，名前付き通信者の平坦な名前空間を構成する。名前には，各プロセスの位置や属性とは無関係に任意の文字列を用いることができる。

ある名前の通信者は，通信系全体で一時にひとつしかない。複数のプロセスが同じ名前の通信者を設けようとしたとき，遅れたプロセスは通信者オブジェクトとして無効値を得る。したがって名前付き通信者は一種のセマフォとしても役立つ。

2.2 伝達者

通信者どうしの通信は「一時に一対一」の通信である。「一時に一対多」の通信を実現するため，通信者によって「伝達者」(messenger) を「雇う」ことができる。通信者から伝達者あてにメッセージを送ると，伝達者は，その時点で「雇主」である (複数の) 通信者 (正確には，通信者が属するそれぞれのプロ

セス) にそのメッセージを分配する[1]。つまり伝達者は，受信者としての通信者の集合を表す。名前付き通信者と同様，伝達者も名前で参照される。

伝達者がメッセージを受け取ったとき，雇主がひとつも無ければ (つまりその受信者の集合が空ならば)，雇主が現れるまでメッセージは伝達者のもとに留め置かれる。これは通信相手の始動を待ち合わせる手段として役立つ。

2.3 通信メッセージ

通信メッセージは，送信者，受信者，発行番号，返答番号，本文からなる。送信者と発行番号の組は，メッセージを一意に指定する。

返答番号が $n(\neq 0)$ のとき，そのメッセージは，過去に受信者が発行番号 n で出したメッセージに対する返答であると解釈される。本ライブラリは，返答メッセージ送受信のための通信者メンバ関数を用意している。メッセージの送受は基本的に非同期的であるが，返答番号を利用して RPC などの同期的交信を容易に模倣することができる。

2.4 例

下記は，伝達者を利用した随時参加脱退可能な多ホスト多人数会話プログラムの簡単な例である [3]。これを実行中の端末から 1 行を入力すると，同じくこれを実行中の他の各端末に，その 1 行を本文とするメッセージが表示される。

```
#include <stdio.h>
#include <d_Corr.ch>

char s[200], *r;
// 鍵盤入力…実行が終了したならば (r==s || r==NULL)
void foo() {r = fgets(s, 200, stdin);}

int main()
{
  d_Message m;
  d_Messenger a("m1"); // 伝達者を構築—無名通信者
  for (;;) {           // a.corr() も雇主として構築
    r = s + 1;
    // 鍵盤入力 s とメッセージ m を同時に待つ
    if (a.fetches(m, foo)) {
      printf("%s-%d:%X:%s",    // m を表示
        m.sender().host(), // 送信者のホスト名
        m.sender().uid(),  // 送信者のユーザ ID
        m.issueNumber(),   // 送信者が付けた発行番号
        (char *)m.text()); // 本文 (文字列へキャスト)
    }
    if (r == NULL) break;
    if (r == s && s[0] != '\0') {
      a.carries(s); // 通信者 a.corr() から伝達者 a へ
    }               // d_Data(s) を本文とする
  }                 // メッセージを送信
  return 0;
}
```

[1] 現実世界とは逆にメッセージを受け取る側が伝達者を「雇う」ことに注意

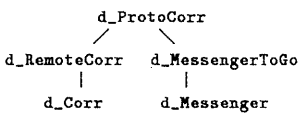
名前 "m1" で伝達者 a を作るとき，それに随伴する通信者 a.corr() が同時に暗黙のうちに作られ，そのプロセスでの a の雇主となる。

a.fetches(m, foo) は，伝達者 a あてのメッセージがキューに着信するのを待って，それを m に代入する。ただし待ち合わせの間は，手続き foo() を実行する。

a.carries(s) は，a が s を本文とするメッセージを運ぶことを意味する。実際には a.corr() が a へメッセージを送信する。引数 s は，言語の型変換規則によるコンストラクタの暗黙の適用によって汎用データ型 d_Data のオブジェクトに変換され (より正確には，s から一時オブジェクトが構築され)，それがメッセージ本文を構成する。a は，送信者 a.corr() 以外の a の雇主が属しているプロセスすべてにメッセージを伝達する。

3 通信者類

通信者クラス d_Corr と伝達者クラス d_Messenger は，次のようなクラス階層を構成している。これらのクラスを総称して「通信者類」と呼ぶ。



d_RemoteCorr は，他プロセスの持つ通信者のクラスである。d_MessengerToGo は，送信先としてだけ使うことのできる伝達者のクラスである。

各クラスのオブジェクトの実体はどれもひとつの識別番号 (と仮想関数表へのポインタ) である。通信者の識別番号はホスト・マシン番号 $\alpha(\neq 0)$，固有番号 β，ユーザ ID γ の三つのビット・フィールドからなる。伝達者の識別番号はフィールド β,γ にまたがった固有番号である。通信者と区別するため $\alpha = 0$ である。ただし $\alpha = \beta = \gamma = 0$ は，通信者も伝達者も表さない無効値の番号である。d_ProtoCorr は，等価性述語 == など，識別番号に対する共通の関数を提供する基底クラスである。

実質的な型と静的な型とは実行時必ずしも一致しない。例えば，通信メッセージの送/受信者の型は d_RemoteCorr であるが，伝達者を仲立ちにしたメッセージ m の受信者 `m.receiver()` は，伝達者の識別番号を持ち，`d_ProtoCorr::isMessenger()` が成り立つ。したがって本ライブラリでは通信者類の各クラスを互いに変換可能にしている。

4 通信者類と通信サーバ

各マシン，各プロセスにまたがる通信系での，通信者と伝達者の管理とメッセージの伝送は，実際には主従2層の通信サーバ・プロセスが実現している。

従サーバである私書箱プロセス POB (post office box) は，各ホストの各ユーザごとに，その各プロセスに属する通信者を管理する。そして，その通信者にあてられたメッセージを預る私書箱として働く。

主サーバ・プロセス PM (postmaster) は通信系に一つだけあり，系内すべての POB を管理し，通信者と伝達者それぞれの名前表を維持する。また伝達者あてメッセージの分配と留め置きとを行う。

4.1 通信者/伝達者の管理

無効値でない d_Corr オブジェクト c は，それを構築したプロセスに属しており `c.isMine()` が成り立つ。無効値でない d_Messenger オブジェクト a は，それに随伴して構築された d_Corr オブジェクト `a.corr()` (`.isMine()`) に雇われており，`a.isHired()` が成り立つ。

ライブラリは，こういった c や a に対し，その構築時に POB に届け出るとともに，発行番号や雇主などの情報を記述した表を内部に設ける。この記述表は参照カウンタによって管理される。プロセス内で対応する識別番号の d_Corr ないし d_Messenger オブジェクトが無くなって，記述表内の参照カウンタの値が 0 になったとき，ライブラリは，この表を破棄するとともに，オブジェクトの消滅を POB に届け出る。届出が名前に関係する場合，POB は更に PM に連絡/照会して名前表を管理/参照する。

```
d_RemoteCorr R("a quick brown fox");
if (R.isValid()) { ····
```

のように，よそのプロセスの通信者を名前で参照するときも，POB を介して PM への照会が行われる。

当然の制約として，メッセージの送信者は送信時 `isMine()` が成り立っていなくてはならない。また，プロセスがメッセージを受信するには，`isMine()` が成り立つ通信者をひとつ以上設ける必要がある。

4.2 メッセージの伝送

通信者対通信者の送信では，送信者のプロセスから受信者の POB へ直接メッセージが送信される。一方，伝達者を仲立ちにした送信では，メッセージは，自分の POB を介していったん PM に伝送され，そこから他の各 POB に分配される。

メッセージは私書箱プロセス POB のキューに格納される。受信者のプロセス ρ が POB に着信通知を頼んでいた場合には，POB は ρ にシグナルを送る。そうでない場合には ρ が受け取りに来るのをただ待つ (プログラマが本ライブラリとは無関係の用途でシグナルをハンドルすることに干渉しないよう，この着信通知シグナルのデフォルト値には，無害な `SIGCONT` が選ばれている)。

例えば ρ が，自分に属する通信者 c_i あてのメッセージを受け取る時，ρ は POB 内にある ρ 用のキューの内容をすべて受け取る。そしてそれを自身の内部キューに追加格納する。そして受信者欄が c_i (ないし，もしあれば c_i が雇っている伝達者 `c_i.messenger()`) であるメッセージを，この内部キューから検索する。

5 メッセージ本文

通信メッセージの本文をどう扱うか，外部表現とプロセス内部の表現の間の符号化/復号化をどう実現するかは，通信端点の取り扱いとともに，分散ソフトウェア構築の主要な問題である。本ライブラリでは，単純化のため本文の型として汎用のデータ型 d_Data 1種類だけを用意し，そこにすべての符号化/復号化処理を閉じ込めることにした。

d_Data は，通信メッセージの本文を構成する自己記述的な汎用データ型のクラスであり，整数，浮動小数点数，文字列へのポインタ，それら一対の組，通信者類，整数ベクタ (一次元配列) へのポインタ，浮動小数点数ベクタへのポインタ，及び d_Data ベクタへのポインタを表現することができる。何も値を有していない状態 (Nil) を表現することもできる。どの種類の値かを判別するため，d_Data 値内部には型タグが設けられている。ベクタへのポインタを表すときは，そのベクタの長さも d_Data 値内部に

保持される。

d_Data が通信者類を表現できることは，伝達者の存在とあわせて，通信相手の動的な各種の決定方法の実現にとって有用である。

i860 と SPARC など CPU 間のエンディアン等の違いは，ONC RPC が用いる通信用外部表現 XDR と d_Data との自動的な相互変換が吸収する。ライブラリ内部でのこの相互変換に型タグが使われる。

5.1 ベクタの共有と参照カウンタ

言語固有のガーベジ・コレクタを持たない C++ では，文字列を含めベクタの記憶領域の管理方法が，アプリケーション・インタフェースの使いやすさを左右する。本ライブラリは，d_Data 値が指すベクタを，通信者/伝達者と同じく参照カウンタを用いて管理する。

ヒープにベクタを確保する時，その直前の番地の領域を余分に確保し，そこに参照カウンタを設ける。このようなベクタは，代入演算などに際し，ベクタ自体はコピーされない。ポインタだけがコピーされるとともに，参照カウンタの値つまり参照数が増やされる。d_Data 値の構築や代入，破壊が行われるごとに，対応する d_Data メンバ関数は，その d_Data 値が指すベクタの参照数を増減させる。そのベクタがどの d_Data 値からも参照されなくなって参照数が 0 になった時，そのベクタは解放される。

なお，このように管理されるベクタに対しては，内容交換演算の使用が効率上，有効である [5]。

5.2 借用値と共有値

```
d_Data S = "hello, world";
```

のように初期値として d_Data とは無関係に構築された配列を用いる場合，そこには参照カウンタを設ける余地は無い。またそれがヒープ上にあるとも限らない。d_Data とは別の起源を持つこのような「借用値」と，d_Data 値どうしが参照カウンタを使って共同管理し共有する「共有値」との 2 種類のどちらであるかは，d_Data 値の型タグ欄に設けた 1 ビットのフラグで区別する。借用値には d_Data での記憶管理は行わない (もうひとつの，より自明な解決方法として，このような配列を無条件にヒープ上に複製することも考えられた。しかし，これは非効率的であるばかりでなく，配列値は参照渡しで，という C/C++ の伝統にも—したがって C/C++ プログラマの直感にも—反している)。

5.3 深い複写

借用値のコピーとして共有値を作るには，陽に深い複写で構築するか又は深い複写での代入を行う。

```
d_Data B(A, d_DeepCopy);
C.becomes(A, d_DeepCopy);
```

通常のコピー・コンストラクタや代入演算子で行われる複写と，指定子 d_DeepCopy [2]付きの関数で行われる深い複写の違いを下記にまとめる。

複写元の値	通常の複写	深い複写
借用値	借用値	新造共有値 参照数=1
共有値	共有値 参照数++	新造共有値 参照数=1

新造ベクタの構築を陽に行うこともできる。A が要素数 3 の d_Data ベクタを指すとき，共有値としてヒープ上に A のコピー B を作るひとつの方法は，次のように B を構築することである。深い複写の場合とは異なり，B の各要素が指すベクタは A との共有になる。[3]

```
d_Data B(d_Array, A[0], A[1], A[2]);
```

5.4 利用者定義の型変換

d_Data が表現する型の多くは，利用者定義変換として働くコンストラクタとキャスト演算子の暗黙の適用によって，透過的な相互変換が可能である。したがって多くの場合 d_Data という型を意識する必要はない (なお残念ながら，文字列を除き C++ の配列からは長さの情報が得られないため，それらからの d_Data の構築は透過的ではない)。

```
int v[10];
d_Data A = d_Data(d_Vec, 10, v);
d_Data B = 1.2; // d_Data(double)
int * u = A;    // operator int *()
double r = B;   // operator double()
```

プログラマが独自に定義したデータ構造については，典型的には，下記の例に示すような方法で d_Data との相互変換関数を定義することができる。これは，クラスの継承関係とメンバの包含関係をそのまま d_Data ベクタの木構造に翻訳する方法である。

[2] d_DeepCopy は実際は型 d_DeepCopyType の唯一の値である
[3] 多重定義により A[i] は ((d_Data *)A)[i] と解釈される

クラス D が，B と T と int から構成されているとする。

```
class D: B {
   T m1;
   int m2;
   // 以下 メンバ関数宣言
};
```

キャスト演算子 operator d_Data() const が B と T に対しそれぞれ定義されているとき，D から d_Data へのキャスト演算子は

```
D::operator d_Data() const {
  return d_Data(d_Array,
    B::operator d_Data(), m1, m2);
}
```

と定義できる。結果の値は参照カウンタで自動管理されるから，この変換関数は，全くブラック・ボックスとして扱うことが可能である。

同様に，const d_Data & を引数とするコンストラクタが B と T に対しそれぞれ定義されているとき，d_Data から D を構築するコンストラクタは

```
D::D(const d_Data &A)
: B(A[0]), m1(A[1]), m2(A[2]) {}
```

と定義できる。

この方法の限界は，ポインタによる木構造以外のグラフ構造，特に循環路のあるリスト構造を変換できないことである。また，関数へのポインタに対しても自明な変換方法はない。これらには，記号表を渡すなど ad hoc な相互変換の取り決めが必要である。

6 おわりに

標準入出力を使った 10 行ほどの簡単なプログラムを socket ライブラリ [4] を使って分散化すると数十行の大きさになってしまったこと，RPC を使ったサーバとクライアントをコルーチン的に動作させようとすると不条理なまでに低水準な技巧が必要となったこと，本ライブラリの設計と製作においては，この二つの経験が動機の一部をなしていた。こういった問題に対し本ライブラリはひとつの解答を示せたと思う。通信端点や通信データの取り扱いの容易さと柔軟性が，本ライブラリの提供する主要な利点である。

今までに，本ライブラリを用いた応用として，メンバ関数呼び出しとしての RPC を実現するための基底クラス d_Process と，その派生クラス開発ツールが作成されている [6]。

また，本ライブラリによる通信系への GNU Emacs [7] インタフェースも実現されている。これは，本ライブラリを使った通信代行プログラムを C++ で記述し，それを Emacs のサブプロセスとして実行することで通信機能を実装している。

現在，GUI のため本ライブラリと Tcl/Tk [8] を協同させるクラスが試作されている (本ライブラリの通信者は単なる通信端点であり，この点に限れば，通信者がメッセージ着信に対し自律的に振舞う ROME [9] よりも，低水準であるといえる。一方，ROME はプログラム全体の構成を強く固定しているように思われる。通信ライブラリが固定的枠組を用意しなくても，通信者の自律性は，継承等により少数のコードで与え得ることが，d_Process やこのクラスの試作で確認されつつある)。

ライブラリに関しての今後の課題としては，実装の効率化や頑健化などが挙げられる (特に，本ライブラリは，通信の実装部に ONC RPC を使用することで，開発コストを節約している一方，いくつか冗長な処理を内部に抱えている。例えば，ONC RPC の XDR ルーチンが生成した受信メッセージ本文は XDR の解放ルーチンで解放しなければならないから，受信時にいったん深く複写することで，それを d_Data クラスの管理下においている)。

参考文献

[1] 加藤，大堀，村上，益田: 高階遠隔手続き呼出しに基づいた分散 C 言語について，コンピュータソフトウェア，Vol.9, No.3, 1992, pp.65-82.

[2] Stroustrup, B.: *The C++ Programming Language*, Addison-Wesley, 2nd edition, 1991.

[3] 鈴木，中澤，佐藤: 分散ソフトウェア開発用ツールキット—ライブラリ—，情処 48 全大, 6D-6, 1994.

[4] UNIX Software Operation: *UNIX SYSTEM V RELEASE 4 Programmer's Guide: Networking Interfaces*, Prentice-Hall, 1990.

[5] Baker, H. G.: Lively Linear Lisp—'Look Ma, No Garbage!', *SIGPLAN Notices*, Vol.27, No.8, 1992, pp.89-98.

[6] 佐藤，鈴木，中澤: 分散ソフトウェア開発用ツールキット—環境—，情処 48 全大, 6D-7, 1994.

[7] Lewis, B., LaLiberte, D. and Stallman, R.: *GNU Emacs Lisp Reference Manual*, Free Software Foundation, 1990.

[8] Ousterhout, J. K.: *Tcl and the Tk Toolkit*, Addison-Wesley, 1993.

[9] Burleigh, S.: ROME: Distributing C++ Object Systems, *IEEE Parallel & Distributed Technology*, May 1993, pp.21-32.