システムソフトウエアと
オペレーティング・システム 60−15
マルチメディア通信と分散処理 61−15
（１９９３．７．９）

# 通信プロトコルをデータ構造で抽象化する拡張C言語 ＲＳＣ

岡野 裕之

日本アイ・ビー・エム (株) 東京基礎研究所

**あらまし** 本論文は，データ構造によって通信プロトコルを抽象化することを特徴に持つ，データ共有型のプロセス間通信を行うための言語仕様RSC を示し，共有メモリがない環境における RSC の実行環境の実現について述べる．RSC の設計では，データを共有するプロセス間の同期を，データへのアクセスだけで実現するために，データへのアクセスを検出してサーバ側の手続きを呼び出す方式を提案する．RSC はこの方式に基づき，共有するデータの定義そのものが通信プロトコルの仕様とみなせる言語設計となっている．RSC の実現は，トランスレータによって，共有データへのアクセスをスタブ関数への呼び出しに変換することによって行った．

# RSC : A C-language Extension for Abstracting Communication Protocols with Data Structures

Hiroyuki Okano

okanoh@trl.ibm.co.jp
IBM Research, Tokyo Research Laboratory
1623-14, Shimotsuruma, Yamato, Kanagawa, 242 JAPAN

**Abstract.** This paper describes an extended C-language, which we developed, for writing client-server style inter-process communication with data sharing. The language shall be referred to as RSC or Remote Sharing for C. Processes can share data as well as pointers by using RSC, so complex data can be passed directly from a server to clients without converting the data format to communication packets. We propose a method of inter-process synchronization, which is triggerred only by data accesses. By using our method, definitions of shared data format can be read as specifications of communication protocols, so that human misreadings of protocol descriptions will be significantly reduced.

# 1 はじめに

従来，クライアント/サーバ型のプロセス間通信を行う場合，ソケットなどのパケット通信や関数で抽象化されたRPCが用いられてきた．しかし，これらの手段ではポインタを直接受け渡すことができないために[1]，クライアント/サーバ間でやりとりされるデータが複雑なデータ構造をしている場合，実際のデータをパケット用のデータに変換する手間と，その逆を行う手間が必要となる．実際，プロセス内で扱われるデータは，複数の構造体がポインタでリンクされるような構造をしていることが多く，ソケット，RPCを使う際の負担は少なくない．

一方，最近になって分散共有メモリの技術が確立され，ハードウェアでそれを実現するシステムも出てきた[2][3]．また，64ビットの広大なアドレス空間を持つプロセッサが市場に出てきて[4]，複数のプロセスが同一のアドレス空間上でメモリを共有することが現実的になってきた．

これらの背景を考え合わせると，クライアント/サーバのプロセス間でメモリを共有し，複雑なデータ構造を直接受け渡すことができるプロセス間通信手段へのニーズが大きく，またそれが実現可能になったと言える．このような状況への対応として，ポインタを受け渡すことができるようにRPCを拡張することが考えられるが，データへのアクセスだけでプロセス間通信やプロセス間の同期を行うという，今までになかったパラダイムも考えられる．

筆者は以上のような認識に基づいて，C/C++の言語仕様を拡張し，データ共有型プロセス間通信用の言語仕様RSC (Remote Sharing for C) を設計した．また，共有メモリのない環境において，トランスレータとスタブジェネレータによってRSCの実行環境を実現した．本論文では，RSCの設計について述べた後，RSCの概要および言語仕様を示す．さらに，共有メモリのない環境で行ったRSCの実現について述べる．

# 2 設計

筆者は以前，日英機械翻訳システムのクライアント・プログラムの開発に携ったことがある[5]．この時に，クライアント/サーバ間で複雑なデータ構造を受け渡す必要があったことから，サーバ上のデータをそのままの形でクライアントに共有させるプロセス間通信手段の重要性を認識し，RSCを設計・実現した．本章では，同期手段と通信プロトコルの抽象化について，これらの設計を述べる．

## 2.1 同期手段

RSCの設計において最も注意したのが，プロセス間の同期方法である．複数のプロセスの間でデータを共有する場合，それが共有メモリで実現されているかパケット通信を用いて仮想的に実現されているかにかかわらず，また，プロトコルをデータ構造だけで抽象化するか関数で抽象化するかにかかわらず，同期の問題は最も考慮されるべき点である．

同期には，

1) 一方のプロセスがデータを初期化するまで他方のプロセスのそれへの参照を待たせる．

2) 一方のプロセスがデータを参照し終わるまで他方のプロセスがそのデータを変更しないようにする．

の2つの種類があり，これらを何らかの同期手段で実現する必要がある．

1つの方法として，セマフォなどの同期プリミティブやRPCなどを用いて，明示的に同期を記述することが考えられる．しかし，データ共有とは別の概念である同期をユーザに意識させると，プログラミング上の負担になり，バグが発生しやすいことも予想できる．そこで，データ共有の概念に融合した形でのプロセス間同期モデルを考えた．

このモデルでは，通信に参加するプロセスは，主にデータを参照する1つ以上のクライアントと，主にデータを生成する1つのサーバに分かれる．また，クライアントの処理が主導的に行われるのに対し，サーバの処理はクライアントからの要求にしたがって従属的に行われる．

まず，種類1) の同期を次のように実現する．

- ある共有データに対しサーバ上の手続きを関連づけておく (図1)．
- データが初期化される前にクライアントがそれを参照すると，システムがその参照の直前でクライアントをブロックする．
- システムはサーバの手続きを呼び出してデータを初期化する．
- クライアントの参照先を初期化されたデータに合わせて，クライアントの実行を再開する．

種類2) の同期はそのまま実現せず，データを複製すること，および，種類1) の同期を利用することで実現する．RSCで記述するプロセス間通信は，上記のようにクライアント/サーバ型を想定し，サーバが生成する (あるいは保持している) データをクライアントが参照するという処理を主な対象とする．そこでは，クライアントがサーバにデータを要求してそ

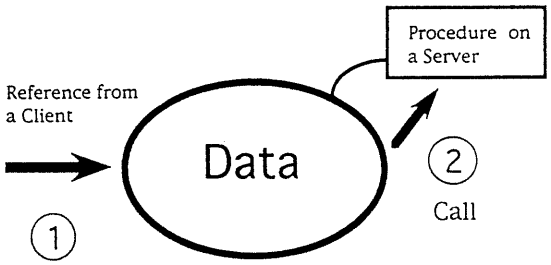


図 1: フック関数による同期

れへの参照をはじめるタイミングを，プロトコルとして明確にできるはずである．そこで，次のようにプログラミングすることで，同期の問題を解決する．

- サーバは次のデータをクライアントが要求するまで，現在共有中のデータを変更しない．
- クライアントの要求にしたがって次のデータを用意する際，前のデータを破棄せずに残しておく．

データの破棄 (解放) は，プログラムで明示的に行わず，ガベージコレクタが行う．トランスレータを用いてプログラムを変換し，共有されるポインタの評価式をスタブ関数への呼び出しに展開すれば，共有中のデータをすべて把握できるので，ガベージコレクタの実現は難しくない．

### 2.2 通信プロトコルの抽象化

通信を扱うプログラムでは，プログラマがプロトコルを誤って解釈するために起こるバグ，および，プロトコルの仕様書と実際のプログラムの仕様が食い違っているために起こるバグが多く発生する．これを軽減する方策の 1 つとして，プロトコルの仕様を簡単にして誤解を防ぐことがある．これに対し RSC を用いると，サーバ上のデータをそのままの形で受け渡すことができ，通信用の中間データ形式を設ける必要がないため，プロトコルの仕様が簡単になるという利点が得られる．

もう 1 つの方策として，プロトコルの仕様をそのままプログラムの仕様に反映することが考えられる．これを実現するため，前節で示した，データへのアクセスだけでデータ通信と処理の同期を行うという方式を発展し，データ構造だけで通信プロトコルを記述できる言語仕様を設計した．この設計でのプロトコルの仕様記述は，C 言語で言う構造体の定義のようなデータ構造の定義だけであり，その中でサーバ側の手続きとの関連づけなど，すべての情報を指定する．したがって，プログラマは，プロトコルの仕様であるデータ構造定義を読んで理解し，その定義をそのままプログラムに取り込んでデータをアクセスすればよい．

ソケットなどのパケット通信と RPC を例にとって，RSC の設計と比較してみる．まずパケット通信では，コマンドとデータがパケットの中に混在しており，バイト・ストリームで抽象化されていると言える．RPC に関しては，**関数呼び出し**によって抽象化されている．これに対し RSC は，**データ構造**によって抽象化され，プロトコルの仕様記述をそのままプログラムから参照できる．

つまり，RSC のデータ構造による通信プロトコルの抽象化は，

- 通信プロトコルの仕様がプログラムに融合しやすい．

という特徴があり，このために，

- 文書上の仕様と実際のプログラムの仕様が食い違うことがない．

という利点が得られる．これはシステムを開発する上で重要な点である．

## 3 RSC の概要

RSC の言語仕様は C/C++のスーパーセットで，次のような拡張が加えられている．

- クライアント/サーバ間で共有される構造体を宣言する rscdef 指定子が加えられている．
- 共有される構造体のメンバにフック関数を指示できる．
- 演算子として rsctop が加えられている．

ここでは概要を説明するために，例をあげてクライアント/サーバの動作を示す．

```
typedef  struct      {
         char*       fullname;
         char        telephone[20];
         Image*      face;
         }           PersonalInfo;

rscdef   struct      {
         char*       name = callup_hook();
         PersonalInfo*     info;
         }           CallupService;
```

これは，名前から個人情報を検索してクライアントに提供するサーバのインタフェースの例である．2 つ目の構造体 CallupService に対して rscdef 指定子が書かれている．これは，この構造体をサーバからクライアントに提供することを指定する．1 対のクライアント/サーバ間で rscdef を指定できる構造体は 1

つだけで，これを rscdef 構造体と呼ぶ．しかし，クライアント/サーバ間で受け渡される構造体が 1 つだけという制限はなく，rscdef 構造体のメンバとしてリンクされる構造体も共有の対象となる．この例では，PersonalInfo および Image がリンクされており，これらもクライアント/サーバ間で共有される．rscdef 構造体から参照される構造体を，rscdef 構造体を含めて，rscdef 関連構造体と呼ぶ．

CallupService の構造体宣言の中に，callup_hook() という関数が書かれている．これは，サーバ側に存在するフック関数と呼ばれる関数で，クライアント側の name へのアクセスによって呼び出される．クライアント側のプログラムがこのフック関数を直接呼び出すことはできない．つまり，RSC で記述すると，クライアント/サーバ間のインタフェースがデータ構造だけで抽象化され，手続きの呼び出しや同期がデータへのアクセスだけで行われる．

次に，クライアント/サーバのそれぞれがどのような動作をするのかを示す．まず，サーバは，次のような処理で最初に共有する rscdef 構造体を指定する．

```
CallupService     top;
top.name          = (void*)0;
top.info          = (void*)0;
rsctop            &top;
```

演算子 rsctop は，サーバ側で評価 (実行) された場合，最初に共有する rscdef 構造体を指定する．

これに対しクライアントは，次のような処理でサーバを呼び出す．

```
CallupService      *callup;
rsctop     callup;
callup —〉 info     = (void*)0;
callup —〉 name     = "foo";
if (!callup —〉 info)
     printf("not found\n");
else
     printf("%s %s\n",
          callup —〉 info —〉 fullname,
          callup —〉 info —〉 telephone);
```

この例では，サーバからの戻り値になる callup —〉 info を 0 で初期化しておき，callup —〉 name に検索したいキーを代入している．この代入によって，サーバ側のフック関数に文字列 "foo" が渡され，フック関数はこの呼び出しの中でデータを検索し，結果を info に代入する．したがって，クライアントが callup —〉 info を参照した時には，その値としてサーバ側のフック関数が設定した値が取り出される．

以上のように，RSC では，構造体のポインタ・メンバへの代入，ポインタ・メンバの参照によって，クライアント/サーバ間の非同期なプロセス間通信を行う．非同期というのは，つまり，callup —〉 name への代入がフック関数の処理を待たずに完了し，フック関数の終了待ちは callup —〉 info を参照した時点で行われるということである．さらに詳しい RSC の仕様は，次の章で述べる．

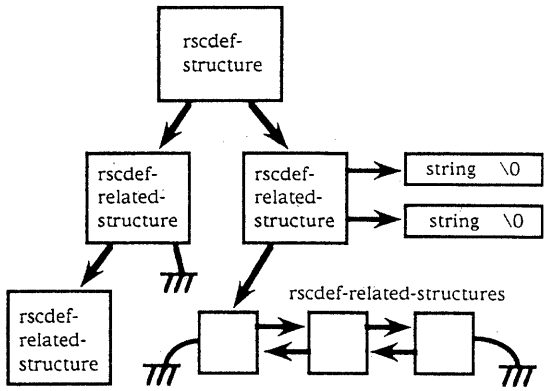


図 2: 共有されるデータ

# 4 RSC の言語仕様

この章では，RSC の言語仕様の内，C/C++の仕様として含まれていない，つまり，RSC として拡張された部分の仕様について述べる．まず，拡張された指定子，演算子などの前に，語彙として rscdef 構造体と rscdef 関連構造体を定義する．

## rscdef 構造体

rscdef 構造体とは，クライアント/サーバ間で共有される，ポインタで互いにリンクされた一連の構造体の内，頂点にあたる構造体のことである．rscdef 構造体からポインタでたどれる構造体および文字列は，すべて共有の対象となる (図 2)．

## rscdef 関連構造体

rscdef 関連構造体とは，rscdef 構造体も含めて，rscdef 構造体からポインタでたどれる構造体の集合を指す．rscdef 関連構造体には，次に示すデータ型のメンバを定義できる．

- int, long などの非構造型，および，それらの配列．
- char へのポインタ．(ただし，ポインタの先は文字列であること．)
- 構造体へのポインタ．(ただし，間接数は 1 であること．ポインタで指される構造体は，rscdef 関連構造体として扱われる．)

- 構造型およびその配列.(ただし，構造型のメンバとしてポインタが含まれていないこと．つまり，ポインタを含む構造型を入れ子にできない．)

## rscspec 指定子

rscspec 指定子は，rscdef 指定子によって指定される rscdef 構造体に属性を与え，どのような文脈，および，環境で使用されるかを指定する．

これは，extern "C" { } と似た文法を持ち，次のように記述される．

```
rscspec  <rsctype> | <rschost>
         [,<rsctype> | <rschost>]... {
         ...
}
```

ここで，<rsctype> は "shared" か "nonshared" のどちらかで，rsctype 属性を指定する．<rschost> は "client" か "server" のどちらかで，rschost 属性を指定する．"client", "server" はそれぞれ，クライアントの文脈，サーバの文脈であることを指定する．

通常，rscdef 構造体の型定義を 1 つのヘッダファイルに書いておき，次のように記述する．

```
rscspec "nonshared","client" {
        #include <utypes.h>
}
rscspec "nonshared","server" {
        #include <utypes.h>
}
```

rsctype 属性，および，rschost 属性のデフォルトの値は不定であり，これらの属性が不定の状態で rscdef 指定子による rscdef 構造体の宣言を行うことはできない．(トランスレータへのオプションで，これらのデフォルト値を指定することができる．)

## rscdef 指定子

rscdef 指定子は，構造体の typedef 名を定義すると同時に，その構造体が rscdef 構造体であることを宣言する．これは，typedef キーワードと同様の文法を持ち，次のように記述される．

```
rscdef   struct   [<identifier>]    {
         [<member> [ = <hook-function>
                     ( [<parameter-list>] ) ] ; ]
         ...
         }        <typedef-name>;
```

rscdef 指定子が記述される時点において，rscspec 指定子などによって，rsctype 属性，rschost 属性が確定していなければならない．この宣言によって，一連の rscdef 関連構造体が確定し，以降に記述される式において rscdef 関連構造体を型に持つポインタが出現すると，RSC のトランスレータはその式に適当な変換を加える．

rscdef 関連構造体の定義には，struct 指定子による通常の構造体定義と異なり，メンバに対してフック関数を指定することができる．フック関数を指定できるメンバは，char*型か構造体へのポインタ型であり，次の例のように記述される．

```
rscdef   struct   {
  char* foo       = foo_hook ( 123, "foo" );
  char* foo2      = foo_hook ( 456, "foo" );
  Info* bar       = bar_hook ( "bar" );
  }     RscDef;
```

フック関数はサーバ側にある関数で，クライアントがこれらのメンバ(フック・メンバ)に参照，代入するごとに呼び出される．ただし，実際には参照のたびに毎回呼び出されるわけではなく，すでにクライアント/サーバ間で有効なアドレスが共有されている場合には，呼び出しは起こらない．

上の例におけるフック関数の，サーバ・プログラムにおける実際の定義では，次のような引数を宣言する．(ただし，引数の識別子は別のものでもよい．)

```
char*  foo_hook ( Rscdef* base, char* addr,
                  int clnt_to_serv, int a, char* s );
Info*  bar_hook ( Rscdef* base, Info* addr,
                  int clnt_to_serv, char* s );
```

第 1 引数の base には，評価されたフック・メンバのベース・ポインタが渡される．この例では，ベース・ポインタの型は RscDef*になる．

第 2 引数の addr には，評価されたフック・メンバの値が渡される．フック・メンバへの代入による呼び出しの場合，共有メモリのある環境においては，クライアントが代入したポインタがそのまま渡される．共有メモリのない環境においては，クライアントが代入したポインタの先のデータがサーバ側のヒープ領域にコピーされ，それへのポインタが渡される．フック・メンバの参照による呼び出しの場合，addr に渡されるデータは，それまでサーバ上にあったフック・メンバのデータがそのまま渡される．

第 3 引数の clnt_to_serv には 1 か 0 が渡され，クライアントのフック・メンバへのアクセスが，参照であれば 0，代入であれば 1 になる．

第 4 引数以降には，構造体の定義におけるフック関数への引数がそのまま渡される．引数が指定されていない場合は，第 4 引数以降は渡されない．第 4 引数を指定すると，例にある foo と foo2 のように，2 つ以上のメンバに同じフック関数を指定する場合に便利であり，フック関数の中からどのメンバでの呼び出しであるかを判断できるようになる．

### rsctop 演算子

rsctop 演算子は，次に示す単項演算子と同じ優先順位を持つ．

++ -- - + ! ~ & * sizeof new delete

オペランドとして rscdef 構造体へのポインタを 1 つ取り，rscdef 構造体以外の関連構造体を指定することはできない．この演算の値，および，演算の動作は，rscdef 構造体の rschost 属性が，クライアントの文脈になっているかサーバの文脈になっているかによって異なるが，いずれの文脈でも，rsctop 演算子の評価式を左辺値にすることはできない．

クライアントの文脈で rsctop 演算子が評価されると，サーバから共有された rsctop 構造体へのポインタをオペランドに代入し，演算の値としてそのアドレスをとる．したがって，rsctop 演算子を評価した式の型は，オペランドの型と同じとなる．サーバがヌル・ポインタを共有した場合は，ヌル・ポインタが値となる．共有メモリのない環境においては，サーバから共有された rscdef 構造体をクライアント側のヒープ領域にコピーし，それへのポインタがオペランドに代入され，演算を評価した値となる．2 度目以降の rsctop 演算子の評価では，オペランドに 1 度目と同じ値を代入し，演算を評価した値も 1 度目と同じになる．

サーバの文脈で rsctop 演算子が評価されると，オペランドのポインタで指される rscdef 構造体をクライアントとの間で共有し，クライアントが rsctop を評価した時にそれを渡す．演算を評価した値は，クライアントとの共有が成功した場合に 0 となり，失敗した場合に -1 となる．つまり，rsctop 演算子を評価した式の型は int である．共有メモリのない環境においては，クライアントに rscdef 構造体の実体をコピーした後，クライアント側のスタブ関数から送られるコマンドを受け取って実行するサービス・ループに入り，この演算が終了するのは，クライアントとの接続が切れた後となる．

### フック・メンバへのアクセス

rscdef 関連構造体のメンバの内，フック関数が指定されたポインタ型のメンバをフック・メンバと呼ぶ．フック・メンバへのクライアントにおける参照に関して，RSC は次のような仕様となっている．

- フック・メンバが初期化されていない場合，サーバ上のフック関数を呼び出し，初期化が完了したデータに対して実際の参照が行われる．
- 初期化されているフック・メンバへの参照ではフック関数を呼び出さない．

また，クライアントにおける代入については次のような仕様である．

- 代入される値にかかわらずサーバ側のフック関数を呼び出す．
- フック・メンバへ 0 を代入することで，そのフック・メンバは初期化されていない状態となる．
- 代入によって上書きされたポインタは，他から参照されていなければガベージコレクションの対象となる．

## 5 実現

共有メモリのない環境において，下位レイヤとしてパケット通信を使って，RSC の実行環境を実現した．本実現では，プロセス間で共有されるデータの実体をパケットで受け渡すが，ポインタだけを送るように変更することで共有メモリのある環境にも適応できる．下位レイヤのインタフェースには依存しない設計になっているので，どの通信手段を用いるかは任意である．

実現方式は，トランスレータによって，rscdef 関連構造体のポインタ・メンバへのアクセスをスタブ関数への呼び出しに変換するというものであり，トランスレータとスタブジェネレータの規模は C++で約 11,500 行である．RSC の仕様ではヒープ上のデータの解放をガベージコレクタが行うことになっているが，この部分はまだ実現できていない．したがって，現時点ではユーザの責任で free() を呼び出し，ポインタを解放している．

本章では，変換の概要と下位レイヤとのインタフェースを示したあと，スタブ関数が具体的にどのような処理を行うかを式の種類ごとに示す．

### 5.1 式の変換の概要

トランスレータはソースプログラムを解析し，型定義や関数のプロトタイプ宣言など，型と識別子に関する情報を抽出する．式を見つけると，一度逆ポーランド列に直し，どの項が左辺値になるかというフラグを立てた後，式を生成して出力ファイルに埋め込む．その際，rscdef 関連構造体の中のポインタ・メンバへのアクセスを見つけると，次の例のようにスタブ関数への呼び出しに変換する．

_rsc_MemType_BaseType ( base, 5 )

これは，右辺に base —〉 mem があり，base が構造体 BaseType へのポインタ，mem が構造体 MemType へのポインタである場合の変換の例である．参照用のスタブ関数はこのように，ベースの型とメンバの型の組ごとに 1 つ作られる．第 2 引数は，mem が

BaseType の中の，何番目のポインタ・メンバかという情報である．

共有メモリのない環境では，後述するように，初期化されていない状態を (void*)1 で表す．したがって，base のようにベース・ポインタが副作用のないポインタの場合は，実際には，

```
(base —〉 mem == (void*)1?
    _rsc_MemType_BaseType ( base, 5 ):
    base —〉 mem)
```

のように展開される．

## 5.2 下位レイヤとのインタフェース

生成されるスタブ関数には，ユーザが提供する通信関数への呼び出しが埋め込まれる．通信関数の名前は任意であり，スタブジェネレータへのオプションとしてこれを指定する．通信関数の引数は，複数のプロセスとの通信が可能なように考慮し，次のような定義としている．

```
int   <read-func>  ( char* typedef_name,
                     char* buffer, int size );
int   <write-func> ( char* typedef_name,
                     char* buffer, int size );
```

第1引数の typedef_name には，通信に関連する rscdef 構造体の typedef 名が渡され，例えばその typedef 名が CallupService なら "CallupService" が渡される．クライアントあるいはサーバのプログラムの中で，1種類だけの rscdef 構造体を扱う場合は，この第1引数を調べる必要はない．2種類以上の rscdef 構造体を扱う場合，それぞれの rscdef 構造体に対応する通信セッションは別々のものにする必要があるので，typedef_name を使ってこれを判断する．

第2引数の buffer，第3引数の size は，システムコールの read/write のそれと同様に，通信バッファを示す．

## 5.3 スタブ関数における処理

以下では，式の種類ごとに，実際にスタブ関数がどのような処理を行うかを示す．

### クライアントでの rsctop の評価

1. サーバから rsctop 構造体を受け取り，クライアント側のヒープ領域に保存する．
2. 受け取った構造体のメンバの内，ポインタ型を持つメンバの値をすべて (void*)1 に変換する．
3. ヒープ上の rscdef 構造体へのポインタをオペランドに代入する．
4. このアドレスを式の値とする．

### サーバでの rsctop の評価

1. オペランドの rscdef 構造体をクライアントに送信する．
2. クライアントのスタブ関数からの要求を処理するサービス・ループに入る．
3. 通信セッションが終了すると，通信関数の戻り値によって，正常終了なら0，異常終了なら -1 を式の値とする．

### クライアントでの rscdef 関連構造体のポインタ・メンバの参照

これは例えば，base —〉 mem のような式の評価である．クライアント側には，base に対応するサーバ側のデータのアドレスが保持されている．このアドレスと共に，mem が何番目のポインタ・メンバかという情報をサーバに伝えることで，サーバは対応するポインタのアドレスを特定する．

1. メンバの値が (void*)1 でなければ，それを式の値にする．(goto 10)
2. メンバの値が (void*)1 であれば，サーバに対応するデータの送信を要求する．
3. サーバは，そのメンバがフック・メンバの場合，フック関数を呼び出し，その戻り値を対応するメンバに代入する．
4. サーバは，対応するメンバのデータをクライアントに送信する．
5. サーバから送られたデータを受け取り，ヌル・ポインタでなければクライアント側のヒープ領域に保存する．
6. 受け取ったデータが以前に受け取ったデータの場合，以前のデータのアドレスを式の値にする．(goto 10)
7. 受け取ったデータが構造体の場合，ポインタ型を持つメンバの値をすべて (void*)1 に変換する．
8. 評価中のポインタ・メンバにヒープ上のデータのアドレスを代入する．
9. 処理6のために，アドレスの対応表に受け取ったデータに関する情報を登録する．
10. このアドレスを式の値とする．

処理1,2によって，クライアント側で有効なデータをヒープ上に保持した後は，何度その式を参照してもその値が式の値となり，サーバとの通信は起こらない．サーバとの通信を強制的に起こしたい場合は，参照の前にあらかじめ0を代入しておく．

上記のアドレスの対応表とは，サーバ側でのアドレスと，クライアント側のローカル・アドレスの対応を管理するもので，処理6のように多重に参照されているアドレスを見つける処理だけでなく，ガベージコレクタの実現にも利用できるものである．

### サーバでの rscdef 関連構造体のポインタ・メンバの参照

サーバでのポインタの参照に関しては何の変換も行われない．したがって，参照されたメンバの値そのままが式の値となる．

### クライアントでの rscdef 関連構造体のポインタ・メンバへの代入

これは例えば，base 一〉 mem が左辺値に現れる式の評価である．クライアントはまず，右辺値のポインタの指す内容をサーバに送信する．サーバ側のスタブ関数は，受け取ったデータが構造体の場合，その中のポインタ・メンバをすべて0に初期化するので，ユーザがその先のデータをたどることはできない．つまり，クライアントからサーバへのデータの共有では，「ポインタの先のポインタ」は扱えないという仕様になっている．

1. 右辺の値がヌル・ポインタの場合は，メンバに(void*)1を代入する．
2. サーバに，右辺のポインタが指す先のデータを送信する．ヌル・ポインタの場合は，ヌル・ポインタだけを送信する．
3. サーバはこのデータを受け取り，ヌル・ポインタでなければヒープ領域に保存し，対応するメンバにそのアドレスを代入する．
4. 受け取ったデータが構造体の場合，ポインタ型を持つメンバの値をすべて(void*)0に変換する．
5. サーバは，このメンバがフック・メンバの場合，フック関数を呼び出す．

### サーバでの rscdef 関連構造体のポインタ・メンバへの代入

サーバでのポインタの代入に関しては何の変換も行われない．したがって，右辺の値がそのままメンバに代入され，式の値となる．

## 6 おわりに

本論文は，データ構造によって通信プロトコルを抽象化することを特徴に持つ，データ共有型のプロセス間通信を行うための言語仕様RSCを示し，共有メモリがない環境におけるRSCの実行環境の実現について述べた．RSCのようなプロセス間通信手段は，複雑なデータを扱うクライアント/サーバ間の通信に有効であり，分散共有メモリが広まりつつある現状も反映している．今後分散共有メモリが普及するとすれば，共有メモリのない従来の環境との互換性が問題になるはずであり，その点でも，RSCによってプロセス間通信を記述しておく意義は大きいと考える．

現在は実行環境を実現した段階であり，RSCの応用はこれからの課題である．今後は，実際のアプリケーションの記述を通じて，言語仕様を見直していきたい．また，今回の実現は共有メモリのない環境で行ったが，RSCの言語仕様は共有メモリのある環境での実現も考慮して設計してある．将来的には分散共有メモリを使った実現も行いたい．

**謝　辞**

本研究に関して有用な助言を下さった，慶応大学 斎藤信男教授，徳田英幸助教授をはじめ，SFC コンソーシアム・オープン情報ベースの参加メンバーの方々に感謝いたします．

## 参考文献

[1] Sun Microsystems: "RPC: Remote Procedure Call Protocol Specification Version 2 (RFC 1057)," in Internet Network Working Group Request for Comments, No. 1057, Network Information Center, SRI International, Jun. 1988.

[2] B. Nitzberg, and V. Lo: "Distributed Shared Memory: A Survey of Issues and Algorithms," Computer, Vol. 24, No. 8, Aug. 1991, pp. 52-60.

[3] A. Forin, J. Barrera, M. Young, and R. Rashid: "Design, Implementation, and Performance Evaluation of a Distributed Shared Memory Server for Mach," CMU-CS-88-165, Aug. 1988.

[4] MIPS Computer Systems Inc.: "MIPS R4000 Microprocessor User's Manual," 1991.

[5] 岡野:"日英機械翻訳システム JETS における翻訳エディタ," 情報処理学会第42回(平成3年前期)全国大会予稿集, 3-39, Mar. 1991.