記 号 処 理 64－6
（1992. 3. 16）

Common Lispのプロセス間通信のための一機構

古坂孝史
情報処理振興事業協会技術センター
井田昌之
青山学院大学
情報科学研究センター研究教育開発室

現在の Common Lisp(以降 CL と記す) の仕様では、同一マシン上もしくは他の計算機上のプロセスと通信するプロセス間通信に関連する機能が定義されていない。そこで、いくつかの代表的な既存処理系上でC言語で記述されたアプリケーションとのプロセス間通信を行なう共通の機構を示す。また、構築したモデルに基づいた設計や実装を示す。実装したデータ転送処理の性能向上の選択点について述べ、最後に、このモデルに基づいて設計したアプリケーションの例を示す。

A COMMON PROVISION FOR PROCESS COMMUNICATION AMONG COMMON LISP IMPLEMENTATIONS

Takashi Kosaka
Information-technology Promotion Agency, Japan (IPA)
Software Technology Center
Shuwashibakoen 3-chome BLDG., 6F
3-1-38 Shibakoen, Minato-ku, Tokyo 105, Japan
kosaka@stc.ipa.go.jp
Masayuki Ida
Computer Science Research Lab., Information Science Research Center,
Aoyama Gakuin University
4-4-25 Shibuya,Shibuya-ku,Tokyo,150 Japan
ida@csrl.aoyama.ac.jp

Common Lisp specification has no process communication features to communicate with other processes. This paper shows that a process communication model of Common Lisp on several implementations. And this paper describes a design and an implementation of the model. Also this paper shows a selection point of view to send a data. At last, this paper shows a sample system using the model.

1 はじめに

分散システムは、プロセス間通信のデータ転送処理の速度がシステムの性能を決定する一要因になる。プロセス間通信のデータ転送処理の処理速度は、プロセス間通信の実装方法や転送するデータオブジェクト、転送処理の実装により異なる。

現在の Common Lisp(以降 CL と記す) の仕様では、同一マシン上もしくは他の計算機上のプロセスと通信するプロセス間通信に関連する機能が定義されていない。そのためプロセス間通信を利用するアプリケーションは、処理系依存の関数や機能等を利用してプロセス間通信やデータの転送処理を実現する必要がある。その結果、プロセス間通信を利用したデータの送受信の方法は同一ではない。実現方法によってはデータの送受信の効率が悪くなる可能性がある。

そこで、本稿ではいくつかの代表的な既存処理系3つを選び C 言語で記述されたアプリケーションとのプロセス間通信を行なう共通のモデルに基づく機構を示す。また、3つの既存処理系 (Allegro、Lucid、Symbolics) 上でそのモデルに基づく設計や実装を示す。実装したデータ転送処理の性能向上の選択点について述べ、最後に、モデルに基づき設計されたアプリケーションの例を示す。

2 プロセス間通信のモデルの設定

本稿では、CL で記述したアプリケーションと UNIX/C で記述されたアプリケーションとのプロセス間通信を考える。そのため、UNIX/C でのプロセス間通信のモデルを示し、そのモデルを参考にして CL を意識したモデルを構築する。

2.1 UNIX/C によるプロセス間通信のモデル

UNIX では、ファイル、デバイス、プロセス等の計算機資源に対する実際の入出力はバイト単位に行う。この制限に従って、UNIX/C で記述したアプリケーションでは、プロセス間通信による実際の転送をバイト単位に行なう。

C 言語で記述したアプリケーション中のデータオブジェクトは、内部表現が処理系やハードウェア環境により異なる。また C 言語では、データオブジェクトの内部表現は一般にバイト単位で表現できる。これらのことから、C 言語のデータオブジェクトをバイト単位のデータに変換した場合、処理系やハードウェア依存になる。従って、C 言語で記述したアプリケーション中のデータオブジェクトをバイト単位に変換した結果は同一でない。そのため、バイト単位に変換したアプリケーション内のデータオブジェクトは、プロセス間通信を用いて送受信しても正確に伝わらない。そこで、プロセス間通信のチャネル上でバイト単位に表現された抽象的なデータ構造を置き、アプリケーションで用いているデータオブジェクトから抽象的なデータ構造へ変換するライブラリを置けばよい。

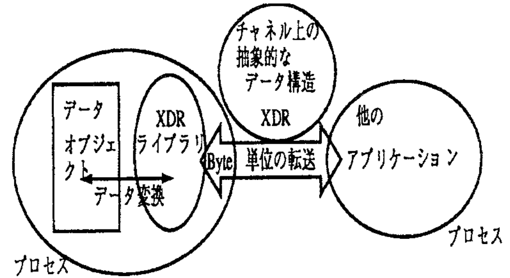

図 1: XDR を用いたプロセス間通信のモデル

UNIX/C の TCP/IP を利用したプロセス間通信の場合、チャネル上の抽象的なデータ構造は、XDR[1] を用いる方法がよく利用されている。また、変換するライブラリに XDR ライブラリがある。XDR を用いたプロセス間通信のモデルを図 1に示す。XDR を用いることで、転送に対してデータ型をある程度保持できる。

2.2 XDR を意識した CL によるプロセス間通信のモデルの設定

CL には、stream データ型がある。本稿で設定するプロセス間通信のモデルでは、stream データ型を利用する。stream データ型による実際の転送は、利用者レベルでのデータ型即ち character 型、integer 型、若しくは、それぞれの副型のデータオブジェクトにより行なわれる。

streamデータ型を用いたプロセス間通信で利用者レベルのデータオブジェクトを送信しても相手のプロセスに正確に伝わらない可能性がある。例えば、character型のデータオブジェクト内の非標準文字オブジェクトを考える。その非標準文字オブジェクトを引数とする時のchar-code関数で返す値は、処理系依存である。また、integer型のデータオブジェクトによる数値表現の範囲も処理系依存である。

データオブジェクトを正確に転送するためには、XDRと同様にプロセス間通信のチャネル上に抽象的なデータ構造を置き、CLのデータオブジェクトから抽象的なデータ構造へ変換するライブラリを置けばよい。更に本稿では、CLとUNIX/Cとの間のプロセス間通信を考えているので、抽象的なデータ構造は、バイト単位のデータ構造でなければならない。

streamデータ型を用いたプロセス間通信のモデルを図1を参考にして図2に示す。

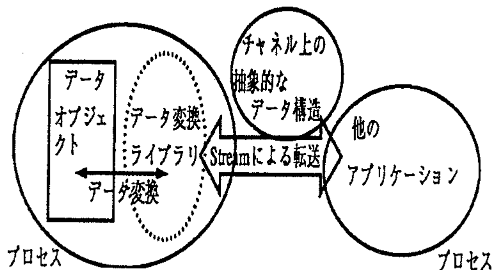

図 2: Streamデータ型を用いたプロセス間通信のモデル

図1と図2からから分かるように本稿が想定しているプロセス間通信のモデルは、C言語のプロセス間通信のモデルとほぼ同一である。データ転送に対する相違点は、図1では、バイト単位の転送を行ない、図2では、streamデータ型を用いた転送が上げられる。更に図2のチャネル上の抽象的なデータ構造へ変換する機構は、データ型が処理系依存でない型を対象とする。そのため、XDRで表現しているデータ型は、そのまま利用できない。例えば、XDRには文字列というデータ型を定義している。CLの非標準文字オブジェクトも含む文字列は、処理系依存の表現になるので、単純の文字列というデータ型では、相手のプロセスに正確にデータが伝わらない。従って、XDRそのままでは、利用できない。

3 Streamデータ型を用いたプロセス間通信モデルのためのID付きデータ構造の設計

CLの仕様では、利用者のためのデータオブジェクトの型が多く定義されている。更に、利用者はクラスや構造体を用いてデータ構造を定義することにより、新たなデータオブジェクトを生成できる。しかし、CLのcoerce仕様では、これら全てのデータオブジェクトをバイト単位のデータオブジェクトに変換できない。そのため、限られたデータオブジェクトだけが、バイト単位のデータオブジェクトに変換できる。例えば、そのデータオブジェクトは、整数型と標準文字型である。

これらのデータオブジェクトでプロセス間通信を行うことは、抽象度の低い利用である。そこで、整数型と標準文字型のデータオブジェクトの要素をもつ抽象的なデータ構造を定義する。

その定義は、CLOSのクラスや構造体等を利用するアプリケーション依存で良い。また、それぞれの要素をアクセスする仕組みがあれば、それらのデータオブジェクトを参照したり、更新することができる。便宜的にこのデータ構造をプロセス間通信データ構造と呼ぶ。例として図3に3つの要素を持つプロセス間通信データ構造をCLOSのクラス定義を用いて示す。(以降、プロセス間通信データ構造は、クラス定義されている場合を想定して記述する。) 図3に示すプロセス間通信データ構造クラスの例では、slot1からslot3に整数型か標準文字型のデータオブジェクトをセットする。スロットにセットするデータオブジェクトは、アプリケーションにより異なる。

プロセス間通信データ構造クラスは、一つのインスタンスを生成する度に決定するユニークなIDを持っている。そのIDを保持するスロットは、図3中のdata-idである。例えば、IDの決定方法に

```
(defclass process-communication-data1 ()
  ((data-id :accessor data-id)
   (slot1   :accessor slot1)
   (slot2   :accessor slot2)
   (slot3   :accessor slot3)))
```

図 3: プロセス間通信データ構造の例

ついてはカウンタ等を利用してアプリケーション依存の方法で定義すれば良い。更に、IDとインスタンスを管理する機構を設定する。この機構の実現例としては、テーブルや2分岐データ構造等を利用すれば良い。この機構を置くことによりIDから対応するインスタンスを求めることもインスタンスからIDを見つけることもできるようになる。

CLから送られてくるデータを受信する他のプロセスは、CL側のプロセスで定義したプロセス間通信データ構造と同じスロット数をもつデータ構造を定義する。また、そのデータ構造で生成したデータオブジェクトは、CLの場合と同様にユニークなIDがある。CL側のプロセス間通信データ構造のIDと受信するプロセス側のデータ構造のIDは、一対一に対応する。

プロセス間通信データ構造を用いたIDとスロット値の転送例を図 4に示す。

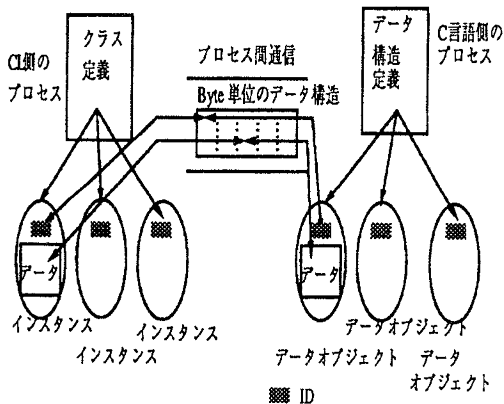

図 4: IDによるスロット値の送受信

図 4に示すように、プロセス間通信データ構造のインスタンスをCL側のプロセスからプロセス間通信を利用して仮想的に他のプロセスへ送信する。その場合、それぞれのスロット値を取り出してIDと共にバイト単位のデータ構造に変換して送信する。また、受けてのC言語側のプロセスでは、バイト単位のデータ構造からIDとデータを取り出しIDに一致するデータオブジェクトを参照し、対応するスロットにデータを設定する。

同様にCL側のプロセスでデータを受信する場合、C言語側のプロセスはIDと共にスロット値を送信する。CL側のプロセスでは、バイト単位のデータ構造からIDとデータを取り出す。そして、IDに一致するインスタンスを見つけだして対応するスロットへデータをセットする。

4 個別実装技術

4.1 チャネルの実装

前章では、プロセス間通信データ構造の設計を行なった。この設計に基づいて実装するためには、プロセス間通信のチャネルを生成する必要がある。そこで、3つの既存処理系の処理系依存の関数や機能等を利用してプロセス間通信のチャネル設定を行なう。(以下、処理系 A、B、Cと記す)

Aは、プロセス間通信の設定機能を持っているLISPマシンの処理系である。この設定機能を利用することによりプロセス間通信のチャネルは、streamデータ型となる (図 5)。

```
(defvar *stream*
  (TCP:OPEN-TCP-STREAM "SS2" 6750 nil
        :CHARACTERS nil))
```

図 5: プロセス間通信のチャネルの設定
(LISPマシンの処理系の場合)

BとCは、UNIXワークステーション上の処理系である。BとCは、他言語インタフェースを利用してC言語でプロセス間通信のチャネルの設定を行なう (図 6)。図 6中に示す関数get_inet_domainは、UNIXのソケットを利用してプロセス間通信のチャネルを生成する。この関数を利用すると、プロセス間通信のチャネルは、UNIXのファイルディスクリプタとなる。

```
int sock_fd; /* 広域変数 */
int get_inet_domain(host,no)
   char *host;
   int no;
{
  int sock;
  struct sockaddr_in addr;
  struct hostent *hp;
  bzero((char *)&addr, sizeof(addr));
  if ((hp = gethostbyname(host))==NULL)
      return -1;
  addr.sin_family = hp->h_addrtype;
  bcopy(*hp->h_addr_list,&addr.sin_addr,
                          hp->h_length);
  addr.sin_port = htons(6750+no);
  if((sock=socket(PF_INET,SOCK_STREAM,0))<0)
      return -1;
  if (connect(sock,&addr,sizeof(addr))<0)
      return -1;
  sock_fd = sock;  /* 広域変数に記憶 */
  return sock; /* ＦＤを返す */
}
```

図6: プロセス間通信のチャネルの設定
(UNIX上のCL処理系の場合)

更にBとCの処理系では、UNIXのファイルディスクリプタをstreamデータ型に変換する機能もある。C言語で定義した関数をLISP関数と等価にする初期設定やファイルディスクリプタをstreamデータ型に変換する処理を図7に示す。

図7に示すように処理系Bは、get_inet_domainで求まるファイルディスクリプタをstreamデータ型に変換する関数make-lisp-streamを用いている。一方、処理系CはDavid Grayが提案したCLOSによるstreamデータ型の拡張機能を利用する。そのため、streamを継承する新しいクラスを定義する。更に、write-byteやforce-outputは、stream-write-byteやstream-force-outputの総称関数を呼び出す仕組みになっているので、それらのメソッドを利用者が記述する。

A、B、Cの処理系では、定義した*stream*にバッファがある。そのバッファは、一定量のデータが溜るとその内容をプロセス間通信のチャネルに転送し、クリアする。

```
#+:B ;;C言語で記述した関数の初期化
(def-foreign-function (get_inet_domain
  (:return-type :fixnum))
  (host :string) (no :fixnum))
#+:C ;;C言語で記述した関数の初期化
(ff:defforeign 'get_inet_domain
  :arguments '(string fixnum)
  :return-type :fixnum)
#+:B ;; チャネルをstreamに変換
(defvar *stream* (make-lisp-stream
  :OUTPUT-HANDLE (get_inet_domain "SS2" 0)
  :element-TYPE '(unsigned-byte 8)))
#+:C ;; プロセス間通信のクラスの定義
(defclass socket-stream
     (FUNDAMENTAL-BINARY-OUTPUT-STREAM)
 ((fd :initarg :fd :initform -1)
  (counter :initarg :couter :initform 0)
  (packet-data :initarg :packet-data
    :initform (make-array 1024
    :element-type '(unsigned-byte 8)))))
#+:C ;; streamを生成
(defvar *stream*
   (make-instance 'socket-stream
   :element-type 'integer
   :fd (get_inet_domain "SS2" 0)))
```

図7: ファイルディスクリプタのstream化と
初期設定 (UNIX上CLの場合)

4.2 データ転送の実装

4.1で示した実装により、プロセス間通信のチャネルを生成する。次にそのチャネルにデータを転送する処理の実装を考える。プロセス間通信を設定したチャネルは、streamデータ型としてCLの環境下で取り扱う。従って、データの転送には、CLの仕様にある入出力関数を利用する。転送するデータがバイト単位であるため、転送にはwrite-byte関数を使う。

あるデータオブジェクトをバイト単位のデータに変換すると、そのバイト長は1バイトとは限らない。そのため、一定長のバイト列を用意する。そ

のバイト列にデータオブジェクトをバイト単位に変換して挿入する。

既存処理系の A、B、C には、バイト列をまとめて転送する処理系依存の機能がある。この機能を利用するとバイト列をまとめて転送できる。図 8 に処理系 A、B、C のデータを転送する処理を示す。

```
#+:C ;write-byte 関数に対応させる
(defmethod stream-write-byte
 ((stream socket-stream) (data integer))
 (with-slots (counter packet-data) stream
  (setf (aref packet-data counter) data)
  (incf counter)
  (when (> counter 1024)
    (write_buffer packet-data 1024)
    (setf counter 0))))
#+:C ;force-output 関数に対応させる
(defmethod STREAM-FORCE-OUTPUT
 ((stream socket-stream))
 (with-slots  (counter packet-data) stream
  (write_buffer packet-data counter)
  (setf counter 0)))
;;; データ送信処理
(defun send-data1 (string) ;;; ケース 1
 (let ((no (set-packet string)))
   (real-send *data* no)))
(defun send-data2 (string) ;;; ケース 2
 (let ((no (set-packet string)))
   (write_buffer *data* no)))
;;;write-byte を利用
(defun real-send (data len)
 (dotimes (i len)
   (write-byte (aref data i) *stream*))
 (force-output *stream*))
#+:A ;write_buffer 関数の LISP バージョン
(defun write_buffer (data len)
 (SEND *stream* :STRING-OUT data 0 len)
  (force-output *stream*))
```

図 8: 本モデルによるデータ転送

図 8 のデータ転送処理は、一定長の文字列の転送を想定している。その処理として、send-data1 と send-data2 の関数を定義した。send-data1 は、バイト列に変換したデータを 1 バイト単位に stream へ出力し、出力終了後に stream に溜まったデータを送信する。一方、send-data2 は、バイト列に変換したデータを直ちに送信する。

send-data1 と send-data2 で共有する関数 set-packet は、文字列オブジェクトをデータ変換してバイト列のデータオブジェクト*data*に挿入する。また、その関数のリターン値はバイト数である。

図 8 中の stream-write-byte と stream-force-output は、利用者が定義したストリームクラスを引数特定子とするメソッドである。stream-write-byte は、与えられた数値を stream のバッファに挿入する。また、stream-force-output は、そのバッファの内容をまとめて転送する。

図 8 中の send-data1 の中で利用する real-send は、バイト列から 1 バイト単位に数値を取り出し定義した*stream*に write-byte で出力する。全ての出力が終了した後、force-output を行う。

send-data2 は、まとめたデータを転送する write_buffer 関数を呼び出している。処理系 A では、write_buffer を記述するのにその処理系依存の SEND を利用した。一方、処理系 B と C では、C 言語で記述した。従って、図 8 には示していないが他言語インタフェースを利用して write_buffer の初期化を行なう必要がある。C 言語で記述した write_buffer を図 9 に示す。

```
write_buffer(data,byte_len)
unsigned char *data;
int byte_len;
{
    write(sock_fd,data,byte_len);
}
```

図 9: まとめてデータを転送する関数 write_buffer(UNIX 上の CL の場合)

図 9 中の sock_fd は、図 6 に示すプログラムで求まったソケットのファイルディスクリプタである。図 9 で示すように、まとめてデータを転送する UNIX の write 関数で記述した。

5 性能向上のための選択点

David Gray が提案した stream の拡張機能を全ての処理系が持っていないので、2 つの転送処理方法の選択が生じる。その選択は、1)CL の仕様で可能な限り転送処理を記述方法。2) 別機能を利用して転送処理を記述方法。である。

そこで、前章の図 8 で示した send-data1 と send-data2 によるデータ転送の性能比較を行なう。send-data1 は、1) の CL の仕様で可能な限り転送処理を記述した場合である。また、send-data2 は、2) の別機能を利用して転送処理を記述をした場合である。

このデータ転送の性能比較を行なうために、サーバ/クライアントモデルを用いる。CL で記述したプログラムをクライアントプロセス、また、転送されてるデータを受け取る C 言語で記述した処理をサーバプロセスとする。

サーバプロセスは、プロセス間通信のチャネルが設定されるまで停止しいる。そして、一度設定され後、クライアントプロセスからデータが到着するまで処理を停止し、データが到着したら内容を表示する。

転送処理を測定する関数 send-data1 と関数 send-data2 を CL の time マクロを利用して実行時間を測定した。なお、この測定はサーバプロセスにデータが到着するまでの時間ではなく、転送処理に要する時間である。それぞれの関数を 10 回実行し実行時間の平均を算出した。実行時間の結果を表 1に示す。

表 1: プロセス間通信を利用したデータの転送処理時間の比

処理系名	実験関数	実行時間の比
処理系 A	send-data1	1.41
	send-data2	1
処理系 B	send-data1	1.45
	send-data2	1
処理系 C	send-data1	2.00
	send-data2	1

表 1中の実行時間の比は、send-data2 の実行時間を使って正規化した値である。そのため、実行時間の比は、send-data2 の処理時間の倍数を示している。表 1より、プロセス間通信のデータの転送は、可能な限り CL の仕様で記述する方法より別機能を利用して記述した方が良いといえる。

6 設定したモデルでのプロセス間通信の例 (*YyonX* の例)

6.1 プロセス間通信に用いるデータ構造 (描画リージョンとテリトリ)

YyonX は、サーバ/クライアントモデルに基づいた CL 用の分散ウィンドウツールキット [2] である。クライアント部分 (*Yy*-Client) は、CLOS により記述されたアプリケーションやウィンドウの管理機能を含んでいる。サーバ部分 (*Yy-server*) は、X server と直接通信を行い X Window System の扱うことのできる資源を利用したウィンドウ利用環境を提供する [3]。*Yy-server* は、C 言語で記述されている。*Yy*-Client と *Yy-server* との間には、プロセス間通信をするためのチャネルがある。

Yy-Client は、ウィンドウ自身やウィンドウを管理するための概念をクラスで定義している [4]。これらの定義したクラスの多くは、矩形概念のクラスを継承している。更に、それらのクラスのなかには、線や円と言った描画プリミティブを表示するクラスもある。このクラスを描画リージョンと呼ぶ。

YyonX でのウィンドウは、様々な構成要素を持っている。ウィンドウやこれらの構成要素は、描画リージョンを継承している。ウィンドウの構成要素には、タイトルバー、スクロールバーやフレームなどがある。ウィンドウの構成要素は描画プリミティブを表示できる。

Yy-Client での、プロセス間通信データ構造は、描画リージョンクラスである。そのため、描画リージョンのインスタンスと ID を管理する機構として 2 分岐によるデータ構造を定義した。

Yy-server は、描画プリミティブが表示できる矩形の領域をテリトリと呼ばれる概念で管理している。*YyonX* においてテリトリは、X Window

System で提供しているウィンドウに対応したデータ構造を持っている。*Yy*-Client の描画リージョンと同様にテリトリは、プロセス間通信データ構造を含むデータ構造である。そのためテリトリとID を管理する機構を設けた。

YyonX では、描画リージョンのインスタンスのもつ ID がテリトリのもつ ID と一対一の関係にあり、その値は整数値である。

6.2 バイト単位のデータ構造にならないデータオブジェクトの転送方法

例えば、*Yy*-Client の矩形概念を表現したクラスは、仮想的に 6 つのスロットがある。それぞれのスロットは、矩形の大きさや位置を表現する。*YyonX* では、位置や大きさを表現するために整数値を利用する。

描画リージョンを継承しているクラスは、さまざまなデータオブジェクトをスロット値として定義している。同様に、*Yy-server* でのテリトリを表現するデータ構造は、矩形を表現する構造以外にさまざまなデータオブジェクトを持っている。

Yy-Client での位置や大きさを表現しているデータは整数値であるので、バイト単位のデータに変換できる。しかし、それら以外のスロット値はバイト単位のデータに変換できない。そこで、それらのデータオブジェクトをプロセス間通信で利用できるデータに変換する機構を置いた。

バイト単位のデータオブジェクトに変換できないデータオブジェクトのプロセス間通信は、まずそのデータオブジェクトと同じ意味のデータ構造や機能を *Yy-server* で用意する。そして、ある特定の局面で転送するバイト列は、転送できないデータオブジェクトであるという取り決めを *Yy*-Client と *Yy-server* 間で行なう。

YyonX では、*Yy*-Client と *Yy-server* との間で *Yy*-protcol と呼ばれるデータ転送の取り決めがある。*Yy*-protcol は、一意の取り決めをおよそ 55 種類用意している。

7 おわりに

CL アプリケーションのプロセス間通信のモデルを示した。そして、そのモデルに基づく CL と C 言語で記述したアプリケーション間のプロセス間通信の設計を行ない、その例を示した。

また、実存する複数の処理系に対するプロセス間通信のためのチャネルの設定やデータの転送方法の実装を示した。この方法は、ここで取り上げた処理系以外でも充分利用可能と思われる。実装した転送方法では、CL の仕様で可能な限り記述する方法と別機能を利用して記述する方法とがある。これらの方法を実装してその実行性能を調べた。その結果、別機能を利用して記述する方法とが良いと言うことが分かった。

今後の課題として、別プロセス上の CL アプリケーションとの間のプロセス間通信の設計や実装がある。また、stream に関連する仕様とその作成技術の共通の仕組みを検討する必要がある。

謝辞

田中啓介、太田幸雄の両氏 (青山学院大学) 及び *Yy* プロジェクトメンバーとの議論は、有意義であった。感謝の意を表します。

参考文献

[1] Network Working Group :XDR:External Data Representation Standard, rfc1014, Jun. 1987

[2] Masayuki Ida, et.al. : An Overview of *Yy* and *YyonX* , CSRL Technical Report 90-004 Aoyama Gakuin Univ., feb. 25, 1991

[3] 古坂孝史、他. : *YyonX* における *Yy*-WS の設計, 情報処理学会第 40 回全国大会, mar. 1991

[4] 田中啓介、他 : *YyonX* における *Yy-server* の設計, 情報処理学会第 40 回全国大会, mar. 1991

[5] bit 別冊 Douglas Comer 著/村井純、他訳 :TCP/IP によるネットワーク構築, 共立出版, jul. 1990