CORBA の概要

（独）産業技術総合研究所　知能システム研究部門
原　　功

1. はじめに

今更ながら CORBA の説明は、多くのサイトで紹介されているのでいうまでもないと思いますが、Wikipedia では

> Common Object Request Broker Architecture（コモン オブジェクト リクエスト ブローカー アーキテクチャー、略称 CORBA）とは、Object Management Group(OMG)が定義した標準規格であり、様々なコンピュータ上で様々なプログラミング言語で書かれたソフトウェアコンポーネントの相互利用を可能 にするものである。

と記載されていました。これではあまりイメージがわかないのですが、要するに、**CORBA を使うと別のコンピュータで動いているアプリケーションと簡単に通信ができて、あたかも自分のプログラムではライブラリを利用するように外部アプリケーションの操作ができてしまう**というものです。

この機能は、UNIX ではよく使われる TCP/IP ソケットを使った通信ポートを開いて、プロセス間でのデータの受け渡しを CORBA ライブラリで隠蔽しているだけとも言えます。そのため、自分で Socket システムコールを使って、データの送受信の方法を考えてプログラムを書いていくよりも、簡単で安全であることが期待できると思います。

しかし、なぜか CORBA と聞くと多くの人が「CORBA は重くて…」というのをよく耳にするのでしょうか？ これは、CORBA の実装方法と利用方法にあると思います。CORBA では、負荷分散機能も実装している場合があり、そのような実装を使えば、自分で通信プログラムを書くよりもはるかに効率的で柔軟なプログラミングが出来るはずなのです。CORBA のライブラリによっては、自分の目的よりもはるかに高機能で実装されているので、そのフットプリントが大きくなり、本当に単純なデータ通信しかしない場合には、重く感じる場合もあるかもしれません。ただし、自分のプログラムを多機能にしていくのであれば、CORBA のような標準化された分散オブジェクトのライブラリを使った方がよい場合もありますので、よく吟味した方がよいと思います。

この文書では、CORBA で最低限の機能を提供できるように実装された RtORB の動作を基に、CORBA の基本的な動作ついて解説をしたいと思います。

2. CORBA の基本的な動作

CORBA は、TCP/IP ソケットを介して他のアプリケーションと通信していると述べましたが、ORB(Object Request Broker)の部分を簡単に描くと図 1 のようになっています。この図で、クライアントである Stub というオブジェクトに、実装されているメソッドを実行させたい場合には、通常のオブジェクトへのオペレーションと同じように

インスタンス名.メソッド名（引数 1，引数 2，引数 3,…）

の形でプログラムに記述すれば、ORB の部分を通じて、自分のプロセス内にオブジェクトの実体があれば、それを呼び出し、他のプロセス（他のコンピュータ上も含む）にあれば、Stub オブジェクト（クライアン

ト側にある CORBA オブジェクト）内に書かれている URL にしたがって TCP ソケットの生成と接続し、オペレーションに必要な引数等のデータを送信、リモートオブジェクトでオペレーションの実行を行い、その結果をもらうという動作をします。

この引数や結果を送受信するときには、双方のプロセス間で決まった形のメッセージ形式を使わないといけませんので、それを規定しているのが GIOP(General Inter-Orb Protocol)です。また、IIOP(Internet Inter-Orb Protocol)は、TCP/IP ソケットを使った通信路で GIOP を動かしているものを言います。CORBA の仕様では、GIOP は実際の通信層とは独立に定義されており、いくつかの条件を満たしたネットワーク層を使えばよいということが述べられています。また、オブジェクト間のデータ通信を行うときに様々なデータをバイト列に符号化するために CDR（Common Data Representation）が使われています。CORBA 間の通信にバイト列を使うのは、様々な CPU、OS に対しても透過的にアクセスし得るメッセージ形式を実現したいからです。CDR による符号化は、実は 32 ビットの UNIX OS で使われるデータ構造に非常に近くなっています。そのため、データ構造によっては、データの符号化・復号化の処理に非常に時間がかかる場合があります。

CORBA を使用する場合に、送受信するデータ構造をどうするかをよく考えて実装すると比較的高速な通信ができるようになります。

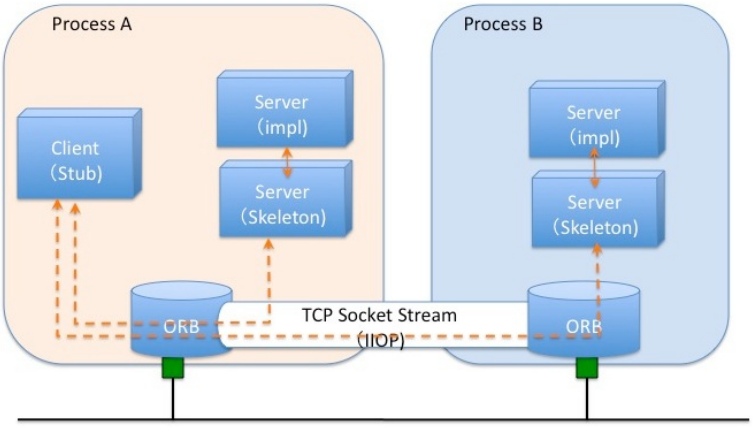

図 1 CORBA の基本的な構造

RtORB では、自分のプロセス内の CORBA オブジェクトを ORB 内の管理テーブルで管理しており、内部オブジェクトに対するオペレーションの場合には、実装関数を直接呼び出し、データの符号化・復号化の処理が省略できるようになっています。外部プロセスに対するオペレーションの場合には、IIOP を使って対象 CORBA オブジェクトと通信するようになっています。RtORB における基本的な構造を図 2 に示します。各 CORBA オブジェクトにおける IIOP の通信は、個々にスレットに割り当てられるのではなく、ORB のイベントループにより制御されています。

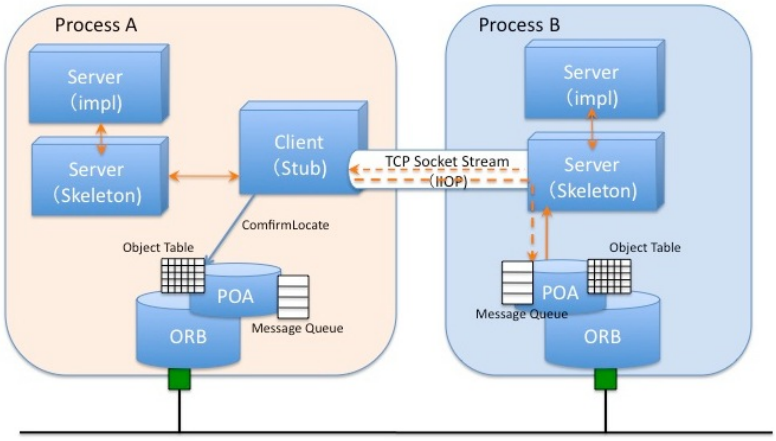

図 2 RtORB の基本的な構造

このような実装は、厳密に言えば本来の分散オブジェクトではありません。この実装では、負荷分散機能を実装できていないからです。本来、分散オブジェクトの目的の１つに動的な負荷分散機能も視野に入れられていると思います。図１のように実装していれば、ORB の部分で、負荷の少ない CORBA オブジェクトに処理を依頼するといったことも実現できると思います。RtORB では、OpenRTM-aist の動作に必要な部分を最低限で実装したいという思いがあり、このような実装になっています。

3. スケルトンとスタブ

図１では、クライアントとなるオブジェクトに Stub、サーバーとなるオブジェクトに Skeleton と描かれています。これらは、それぞれ次のような意味があります。

Stub（スタブ）:

スタブとは、クライアントアプリケーションからみた時に、サーバーのオブジェクトがあたかも、自分のところ で動作しているように見せる代理（Proxy）オブジェクトです。

Skeleton（スケルトン）:

スケルトンとは、 サーバー側の実装オブジェクト（図では Impl）を外部のスタブオブジェクトから見えるように公開されたインターフェースオブジェクトです。

これらのスタブとスケルトンが、協調して動作することで、外部プロセス内のオブジェクトがあたかも自分のところで動いているように見せることができます。また、内部のオブジェクトであっても、これらの２つのオブジェクト経由でアクセスすることで、どこに実装があっても同じようにアクセスすることが可能に なっています。

これらのオブジェクトは、IDL (Interface Definition Language)と呼ばれるインターフェース記述言語で記述されたファイルから、各 CORBA ライブラリに対応した IDL コンパイラによって生成されるプログラムに記述されています。サーバー側は、スケルトンのプラグラムファイル、クライアント側はスタブのプログラムファイルをコンパイルしリンクすることで利用することができます。

IDL ファイルには、CORBA の通信で使うためのデータ型の定義、CORBA オブジェクトのクラス定義等が含まれ、IDL コンパイラによって、スタブ ファイル、スケルトンファイル、各ヘッダーファイルを出力します。これらのファイルには、ユーザ定義のデータ型、通信時の符号化、復号化（marshaling/unmarshaling）をするプログラムのコードが含まれています。そのために、IDL コンパイラ

によっては、非常に大きなファイルが出力されます。RtORB では、C 言語と C++言語との両方に対応するプログラムコードが含まれますので、IDL ファイルに比べて非常に大きなファイルが生成されます。また、IDL コンパイラによっては、実装クラスの雛形ファイルである、Skeleton Impl ファイルも出力するものもあります。

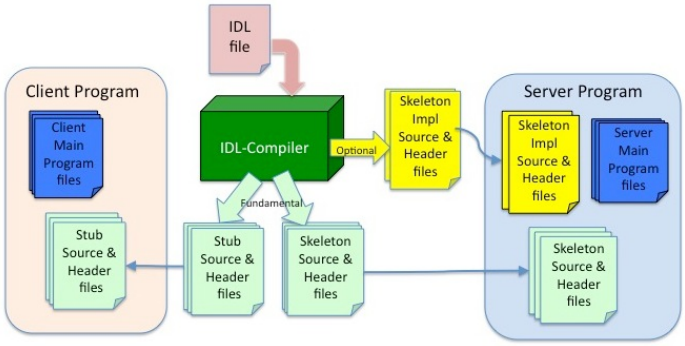

図 3 IDL コンパイラの概要

RtORB では、現在のところ ORBit2（http://projects.gnome.org/ORBit2/）の IDL コンパイラを改変して利用しています。そのため、IDL ファイルをコンパイルすると下記のファイルが出力されます。

- Stub と Skeleton の共通ヘッダーファイル（C,C++共通）
- Stub 用のプログラムファイル（-stubs.c）（C,C++共通）
- Skeleton 用のプログラムファイル（-skels.c）（C,C++共通）
- Stub と Skeleton の共通プログラムファイル（-common.c）（C,C++共通）
- Skeleton Impl 用のテンプレートファイル（-skelimpl.c）（C 言語では、これに実装を加えて、C++では、これと別に実装ファイルが必要）オプション指定で出力する。

4. GIOP と IIOP について

GIOP と IIOP に関しては、書籍や Web でも非常にたくさんの情報があります。詳細はそちらを参照していただきたいのですが、簡単に言えば、GIOP は、CORBA でリモート呼び出しを行うときのプロトコルであり、IIOP は、GIOP を TCP/IP 上に実装したものということができます。RtORB では、IIOP のみを実装しています。

4.1. GIOP の定義

OMG の定義では、GIOP の定義は以下の 3 つの項目があります。

- **The Common Data Representation (CDR)の定義**
  OMG IDL で定義されたデータ型を ORB 間で送受信するための低レベルの通信用表現へのマッピング。引数や返値を送受信するための各データをバイト列に符号化／復号化（marshaling/unmarshaling） するための定義。
- **GIOP Message Formats**
  CORBA クライアントと CORBA サーバー間で情報の受け渡しを行うためのメッセージ様式。これには、8 つの形式（GIOP1.0 では 7 つの形式）のメッセージがあり、その中の 2 つはリモートプロ

シージャコールで必須のメッセージの読み書き(read/write)であ る。 それ以外は、メッセージのコントロールや最適化や信頼性を向上させるための補助メッセージ形式である。

- GIOP Transport Assumptions
  GIOP メッセージの通信路がどのような機能をもつべきか、また、メッセージの送受信時における順序に関する規定したもの。

です。これらの三つに加えて、IIOP では、

- Internet IOP Message Transport
  GIOP メッセージを送受信するための TCP/IP 通信路をどのように開設し、メッセージを送受信するかの規定。

があります。また、The Interoperable Object Reference (IOR) というリモートオブジェクトへの参照の形式についてもここで規定されている。IOR は、マジックナンバー（”GIOP”という文字列)、バイトオーダー。タグ付きのプロファイル、プロトコルのバージョン、サーバーのアドレスとポート番号、リモートオブジェクトを識別するオブジェクトキー、コンポーネントのプロファイルを含んだバイト列になっている。要は、実装されているオブジェクトへのアクセスポート（IP アドレス とポート番号）オブジェクトキーなどの CORBA の実装などの情報が、バイト列で表現されたものになります。そのため、これがわかればリモートアクセスをすることができることになる。また、CORBA の NameServer に登録したり、入手したりするのはこの IOR であり、CDR 時のオブジェクト参照でもあります。

GIOP と IIOP は、全く別に規定されているものではなく、むしろ GIOP の TCP/IP 上の実装として IIOP は存在しています。現在では、CORBA の新しい仕様には、マルチキャスト上に GIOP を実装した MIOP の規定も追加されています。

4.2. GIOP Message Format

前述のように、GIOP メッセージには 8 つのメッセージ形式があります。これらのメッセージによって、オブジェクト間のオペレーションの要求を実施したり、オブジェクトが実装されている場所を探したり、通信経路を設定したりすることができます。GIOP のメッセージは、”GIOP"の文字列から始まる 12 バイトのヘッダーとメッセージ本体からなっており、メッセージ本体は、CDR で符号化された各形式のメッセージヘッダーと本体からなっています。GIOP ヘッダーは、マジックナンバー（”GIOP"の文字列)、バージョン情報、 CDR で符号化処理の時のバイトオーダー、メッセージのタイプ、メッセージ本体の長さ（GIOP ヘッダーを含まない)からなっています。この GIOP ヘッダーは固定長になっているので、通信時には GIOP ヘッダーを最初に受信し、その後リモート呼び出しを行うメソッド名や引数が含まれたメッセージを受信し、オペレーション処理後に応答を返すという動作を行います。

RtORB では、GIOP_execute_request 関数で GIOP のメッセージに対応する処理を行っています。また、RtORB で実装している GIOP メッセージヘッダーを下記に示します。

```
typedef struct {
        octet major;
        octet minor;
} GIOP_Version;
typedef struct{
        char magic[4];
        GIOP_Version version;
        octet flags;       /// GIOP1.0 --> byte_code
        octet message_type;
        unsigned long message_size;
  } GIOP_MessageHeader;
```

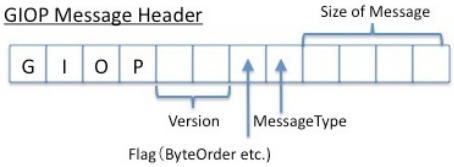

図 4 GIOP Message Header

ここで GIOP の本体のメッセージサイズは、flag に示されたバイトオーダーによって書かれているために、自身のバイトオーダーと異なる場合には、そのバイト列を逆順に並べ替える必要があります。例えば、符号化されたときのバイトオーダーがビッグエンディアンであり、その長さが 1012 であった場合には、0x00,0x00,0x03,0xf4 がセットされていることとなり、このバイト列をそのままリトルエンディアンで解釈すると 4093837312 となってしまうためです。

GIOP1.0 では、flag はメッセージの符号化のバイトオーダーのみを表していましたが、GIOP1.1 以降では、flag の最下位のビットがバイトオーダーを表しており（0 であればビッグエンディアン、 1 であればリトルエンディアン）、その上のビットは、分割されたメッセージが後に続いているかどうか（0 であればメッセージの最後のパート、 1 であれば分割されたメッセージがあとに続いている）を表しています。また、GIOP のメッセージの種類は下記のように定義されており、それぞれのメッセジージ対応する処理を実装していなければいけません。

```
enum GIOP_MsgType{
    GIOP_Request,         // 0  Client
    GIOP_Reply,           // 1  Server
    GIOP_CancelRequest,   // 2  Client
    GIOP_LocateRequest,   // 3  Client
    GIOP_LocateReply,     // 4  Server
    GIOP_CloseConnection, // 5  Servar(or Client, from GIOP1.2)
    GIOP_MessageError,    // 6  Client or Server
    GIOP_Fragment,        // 7  Client or Server, from GIOP1.1
    GIOP_MsgType_END      // 8
};
```

上記のように、GIOP にはいくつかのバージョンが存在し、拡張され続けられています。そのため、GIOP1.2 等を実装する場合には、それ以前のバージョンである GIOP1.0, GIOP1.1 も動作するように実装することが義務付けられています。そのため、GIOP1.0 さえ実装しておけば、最新の CORBA とも通信することができます。

4.2.1. Request Message

Request Message は、CORBA のリモートプロシージャコールの基本となる一つで、クライアントからサーバーへのオペレーションの実行要求を送る場合に使われます。すなわち、原則として、Request Message の符号化はクライアント、復号化はサーバーで行うことになります。原則というのは、GIOP1.2 から Bidirectional IIOP(双方向の送受信)ができるようになっており、コールバック関数を起動するときには逆になるからです。

また、Request Message は、下記のような構造になっています。

12-bytes GIOP Header	Variable-length GIOP Request Header	Variable-length GIOP Request Body

4.2.1.1. Request Message Header

Request Message Header は、CORBA の仕様書では、下記のように定義されています。

```
struct RequestHeader_1_0 { // Renamed from RequestHeader
  IOP::ServiceContextList service_context;
  unsigned long  request_id;
   boolean response_expected;
  IOP::ObjectKey  object_key;
   string operation;
  CORBA::OctetSeq requesting_principal;
};
typedef octet RequestReserved[3];
struct RequestHeader_1_1 {
  IOP::ServiceContextList service_context;
  unsigned long  request_id;
   boolean  response_expected;
   RequestReserved reserved; // Added in GIOP 1.1
  IOP::ObjectKey object_key;
  string operation;
  CORBA::OctetSeq requesting_principal;
};
// GIOP 1.2, 1.3
typedef short AddressingDisposition;
const short KeyAddr = 0;
const short ProfileAddr = 1;
const short ReferenceAddr = 2;
  struct IORAddressingInfo {
  unsigned long selected_profile_index;
  IOP::IOR     ior;
};
union TargetAddress switch (AddressingDisposition) {
  case KeyAddr:        IOP::ObjectKey object_key;
  case ProfileAddr:    IOP::TaggedProfile profile;
  case ReferenceAddr:  IORAddressingInfo ior;
};
```

```
struct RequestHeader_1_2 {
   unsigned long   request_id;
  octet  response_flags;
  RequestReserved  reserved; // Added in GIOP1.1
  TaggedAddress   target;
  string  operation;
  IOP::ServiceContextList service_context;
  // rquesting_principal not in GIOP 1.2 and 1.3
};
typedef RequestHeader_1_2 RequestHeader_1_3;
```

上記の構造からわかるように、Request Message Header は、GIOP1.2 以降に大きく変更されています。また、各構造体の各要素は次のような意味をもっています。

request_id：

この ID は Reply Message の時のヘッダーに付与される ID と同じものであり、クライアントが返信メッセージを同定するために使うために生成しているものです。この ID は、同一コミュニケーション中に対して再利用してはいけません。

response_flags：

この要素は、メッセージが同期型なのかどうかを表しています。この値が 0x00 の時には、oneway のオペレーションのように返信を要求しないようなオペレーションの要求になります。0x01 の場合は、メッセージ本体なしの返信をするという意味であり、このメッセージの Reply Message には、Reply Body は必要ありません。また、0x03 の時には、返信を期待するという意味であり、通常の同期型リモートプロシジャーコールと同等になります。この 0x03 は、 GIOP1.0,1.1 での response_expected == TURE(1)と同じ意味であり、0x00 が response_expected == FALSE(0)と同じ意味になるので注意した方が良いです。

reserved：

この要素は、3 バイトの長さで、通常 0 が代入されています。将来の仕様変更等に対応するための部分であり、現時点では使われていません。

object_key：

この要素は、GIOP1.0,1.1 でリモートプロシジャーコールをするオブジェクトを表す ID になる。通常は、サーバー内で対象オブジェクトを一意に同定するための文字列になります。

target：

この要素は、GIOP1.2 以降でリモートプロシジャーコールをするオブジェクトを表すものです。object_key がサーバー内で使っている ID に対して、この要素は 3 通りの表現方法を提供しています。この要素の型は、共用体になっており、**KeyAddr** の時は、GIOP1.0,1.1 と同じようにサーバー内での ID を表し、**ProfileAddr** の時には、IOR と同値の TaggedProfile 型の要素になります。**IORAddressingInfo** の場合には、完全な IOR となり IORAddressingInfo 型の要素となります。この要素の selected_profile_index は、ior の何番目の Profile を用いるかを表しており、最初の要素

の場合には 0 になります。

operation：

この要素は、リモートプロシージャコールの時のメソッドに対応する文字列になります。通常、オブジェクトの attribute の内容を得たり、変更したりする場合には、 _get_<attribute>, _set_<attribute>をそれぞれ使用します。その他で言えば、GIOP1.1 以前の仕様で実装された CORBA の中には_non_existent のオペレーションが_not_existent になっている場合があります。これは以前の資料で綴りミスであったために起こっており、以前の CORBA との完全な互換性を確保する場合には、_non_existent,_not_existent の両方のメソッドを実装した方がよいと思います。

service_context：

これは ServiceContext を表しており、クライアントからサーバーに対して渡されます。通常のこの要素では、文字列のコーディングやセッション情報を渡されるようになっています。（ちなみに、RtORB では、何も送らないようにしていますし、もし渡されても何も処理していません。）

requesting_principal：

これは、BOA で呼び出しを行うクライアントを同定するために付け加えていたものであり、GIOP1.2 以降は省略されています。これは、ServiceContext での OMG Security Service で同様なことができるためです。

4.2.1.2. Request Message Body

GIOP1.0,1.1 では、CDR を使って符号化された Request Message Body はヘッダーに続いて格納されるが、GIOP1.2 以降では Request Message Body は、8 octet の Alignment になっています。Request Message Body は、IDL で規定されている in と inout のすべてのパラメータと必要であれば、Context 疑似オブジェクトを CDR で符号化したものが含まれています。通常の平易なリクエストでは、Context 疑似オブジェクトは含まれません。

4.2.2. Reply Message

Reply Message は CORBA のリモートプロシージャコールの基本となる一つであり、Request Message に対する応答処理です。そのため、サーバーからクライアントへ実行結果を返送する場合に使われます。したがって、Request Message とは逆に、原則として Reply Message の符号化はサーバー、復号化はクライアントで行うことになります。このメッセージが発行されるのは、Request Message で response_flags または request_expected が 0x00 でない場合になります。

Reply Message は、基本的に下記のような構造になっています。

12-bytes GIOP Header	Variable-length GIOP Reply Header	Variable-length GIOP Reply Body

4.2.2.1. Reply Message Header

Reply Message Header は、CORBA の仕様書では下記のように定義されています。

```
#if MAX_GIOP_MINOR_VERSION < 2
 // GIOP 1.0 and 1.1
 enum ReplyStatusType_1_0 { // Renamed from ReplyStatusType
   NO_EXCEPTION,
   USER_EXCEPTION,
    SYSTEM_EXCEPTION,
   LOCATION_FORWARD
  };
  // GIOP 1.0
 struct ReplyHeader_1_0 { // Renamed from ReplyHeader
   IOP::ServiceContextList service_context;
   unsigned long request_id;
   ReplyStatusType_1_0 reply_status;
  };
 // GIOP 1.1
 typedef ReplyHeader_1_0 ReplyHeader_1_1;  // Same Header contents for 1.0 and 1.1
#endif // MAX_GIOP_VERSION_NUMBER
#if MAX_GIOP_MINOR_VERSION >= 2  // GIOP 1.2, 1.3
 enum ReplyStatusType_1_2 {
   NO_EXCEPTION,
   USER_EXCEPTION,
   SYSTEM_EXCEPTION,
   LOCATION_FORWARD,
   LOCATION_FORWARD_PERM,  // new value for 1.2
   NEEDS_ADDRESSING_MODE  // new value for 1.2
 };
 struct ReplyHeader_1_2 {
   unsigned long request_id;
   ReplyStatusType_1_2  reply_status;
   IOP::ServiceContextList  service_context;
 };
 typedef ReplyHeader_1_2 ReplyHeader_1_3;
#endif // MAX GIOP VERSION NUMBER
```

また、各要素は次のように定義されています。

request_id：

これは Request Message の **request_id** に対応するもので、同じ値になります。

reply_status：

これはリクエスト終了時の状態であり、正常終了なのか例外が発生しているかを示しています。この値が **NO_EXCEPTION** と等しければ正常終了を表しており、返り値等を含んだ Reply Message

Body を持つことになります。それ以外であれば、例外の内容、再実行のためのサーバーのオブジェクト参照などが含まれます。

service_context：

これはサーバーからクライアントに送られるサービスデータです。このヘッダーに続いて Reply Message Body が格納されます。

4.2.2.2. Reply Message Body

GIOP1.0,1.1 では、Reply Message Header に続いて、CDR で符号化された Reply Message Body が格納されます。GIOP1.2 以降では、Reply Message Body も 8-octet Boundary の Alignment をとることが規定されています。また、Reply Message Header には reply_status に応じて下記のようなデータが含まれています。

- NO_EXCEPTION の場合：

  この場合はリモートプロシジャーコールの正常終了を意味し、返り値と IDL で規定した inout,out の順に CDR で符号化されたものが格納されています。

- USER_EXCEPTION または SYSTEM_EXCEPTION の場合:

  この場合には、リモートプロシジャーコールの時に例外が発生したことを意味し、例外情報が含まれます。SYSTEM_EXCEPTION のときには、下記の構造体のデータが含まれますが、USER_EXCEPTION の場合には、IDL で規定されていた Exception 構造体になります。

```
struct SystemExceptionReplyBody {
   string  exception_id;
   unsigned long minor_code_value;
   unsigned long completion_status;
 };
```

  この構造体で minor_code_value の上位 20 ビットは、Vendor Minor Codeset ID(VMCID)であり、下位 12 ビットには minor code が格納されます。VMCID はベンダーに対する ID であり、OMG で一意にきめられたもので、4096 までの minor code を使用することになります。VMCID が 0 の場合には、予約されていない任意のベンダーとして定義されていますが、発行ベンダーがわからなくなるため、正式にはその仕様は奨励されていません。

- LOCATION_FORWARD の場合:

  この場合には IOR がその内容に含まれます。ここで示された IOR は、同じリクエストをこのオブジェクトに対して再送することが期待されています。

- LOCATION_FORWARD_PERM の場合:

  これは GIOP1.2 以降に追加されました。これは LOCATION_FORWARD に似ていますが、リクエストはここで返された IOR も同時に有効ですが、将来的にここで返された IOR を使うことを奨励すると言うことを意味しています。

- NEED_ADDRESSING_MODE の場合:

  これも、GIOP1.2 以降に追加されました。この返信には GIOP::AddressingDisposition 型のデータが 含まれており、クライアントはこれをもとにしたリクエストを再送することが期待されています。

4.2.3. CancelRequest Message

このメッセージは、GIOP1.0, 1.1 ではクライアントからサーバーに対してのみ送られるものでありましたが、GIOP1.2 以降からは双方向で送られることになりました。CaneclRequest Message は、Request や LocateRequest を中止したいとき（relply を受けたくないとき）に送信します。

基本的な構造は下記のようになっており、GIOP Header と CancelRequest Header からなるメッセージです。

12-bytes GIOP Header	4-bytes GIOP CancelRequest Header

```
struct CancelRequestHeader {
    unsigned long  request_id;
};
```

4.2.4. LocateRequest Message

このメッセージは、クライアントからサーバーに対して送られるメッセージであり、対象オブジェクトがそこに存在するかどうかを確認することができます。このメッセージを使わずに Request Message を直接発行しても受けるオブジェクトが存在しなければ、代替 IOR を送信するようにはなっているのですが、その場合には送り手であるクライアントに再送機能が実装されている必要があります。従って、再送機能を実装していない CORBA を利用する場合には、まず、このメッセージで対象オブジェクトの所在を確認して Request Message を発行する必要があります。

LocateRequest Message は下記の構造を持っており、object_key または target が LocationRequest Header に埋め込まれるのみで、Reqgest Message や Reply Message のようにメッセージ本体はありません。また、LocateRequest Message Header の構造体の要素は、Request Message Header に記載したものと同じ意味を持っています。

12-bytes GIOP Header	Variable-length GIOP LocationRequest Header

```
// GIOP 1.0
 struct LocateRequestHeader_1_0 {// Renamed LocationRequestHeader
   unsigned long   request_id;
   IOP::ObjectKey  object_key;
 };
   // GIOP 1.1
 typedef LocateRequestHeader_1_0 LocateRequestHeader_1_1;
 // Same Header contents for 1.0 and 1.1
// GIOP 1.2, 1.3
 struct LocateRequestHeader_1_2 {
   unsigned long         request_id;
   TargetAddress         target;
 };
 typedef LocateRequestHeader_1_2 LocateRequestHeader_1_3;
```

4.2.5. LocateReply Message

このメッセージは、前述の LocateRequest Messsage の返信として発行されます。そのためサーバーからクライアントに対してのみ発行されるものです。
また、このメッセージの構造は下記のようになっています。

12-bytes GIOP Header	8-bytes GIOP LocationReply Header	Variable-length GIOP LocationReply Body

LocateReply Message Header は、仕様書に下記のように記載されています。これでわかるように、LocateReply Message Header には、LocateRequest Message で使った request_id と LocateStatus の ID が格納されています。

```
#if MAX_GIOP_MINOR_VERSION < 2
  // GIOP 1.0 and 1.1
  enum LocateStatusType_1_0 {// Renamed from LocateStatusType
    UNKNOWN_OBJECT,
    OBJECT_HERE,
    OBJECT_FORWARD
  };
  // GIOP 1.0
  struct LocateReplyHeader_1_0 {// Renamed from LocateReplyHeader
   unsigned long        request_id;
   LocateStatusType_1_0 locate_status;
  };
  // GIOP 1.1
  typedef LocateReplyHeader_1_0 LocateReplyHeader_1_1;
 // same Header contents for 1.0 and 1.1
#else
 // GIOP 1.2, 1.3
  enum LocateStatusType_1_2 {
    UNKNOWN_OBJECT,
    OBJECT_HERE,
    OBJECT_FORWARD,
    OBJECT_FORWARD_PERM,      // new value for GIOP 1.2
    LOC_SYSTEM_EXCEPTION,     // new value for GIOP 1.2
    LOC_NEEDS_ADDRESSING_MODE  // new value for GIOP 1.2
  };
  struct LocateReplyHeader_1_2 {
   unsigned long        request_id;
   LocateStatusType_1_2 locate_status;
  };
  typedef LocateReplyHeader_1_2 LocateReplyHeader_1_3;
#endif // MAX_GIOP_VERSION_NUMBER
```

また、ここで記載されている LocateStatus は、次のような意味をもっています。

- **UNKNOWN_OBJECT の場合**:
  これは指定されたオブジェクトに関して、サーバーがわからない場合に発行されます。このとき LocateReply Message Body の中身はありません。
- **OBJECT_HERE の場合**:
  これはサーバー上に指定されたオブジェクトが存在していることを示しています。このときも LocateReply Message Body の中身はありません。

- OBJECR_FORWARD または OBJECT_FORWARD_PERM の場合:
  この場合には LocateReply Message Body が存在します。その内容は、対象となるオブジェクトの IOR となり、Reply Message の時の OBJECR_FORWARD または OBJECT_FORWARD_PERM の場合と同じものが入ります。
- LOC_SYSTEM_EXCEPTION の場合:
  この場合には LocateRequest Message の実行に対して例外が発生したことを示しています。このときも LocateReply Message Body が存在し、その内容は GIOP::SystemExceptionReplyBody になります。
- LOC_NEEDS_ADDRESSING_MODE の場合:
  この場合にも LocateReply Message Body が存在し、その内容は対象となるオブジェクトの所在を GIOP::AddressingDisposition で表現されたものになります。具体的には、 Reply Message の NEED_ADDRESSING_MODE 時と同じものが入ることになります。

4.2.6. CloseConnection Message

CloseConnection Message は、GIOP1.1 以前のバージョンではサーバーのみが使用するメッセージであり、通信路を遮断するためにクライアントへ送るメッセージです。このメッセージは GIOP Header のみからなっています。GIOP 1.2 以降のバージョンでは、双方向の通信をサポートするために、サーバーとクライアントの両方で使用するように変更されました。ただし、RtORB では現在のところまだ実装は行っていません。

4.2.7. MessageError Message

MessageError Message は、全ての GIOP メッセージの応答として発行することができます。例えば、バージョン番号やメッセージタイプが受け入れられなかったり、ヘッダーの不備（マジックナンバーがことなるなど）があったりした場合に発行されるメッセージです。このメッセージも CloseConnection Message と同様に GIOP ヘッダーのみならなるメッセージです。ただし、RtORB では現在のところまだ実装は行っていません。

4.2.8. Fragment Message

Fragment Message は、GIOP1.1 から付け加えられた機能です。これは CORBA の送受信バッファが限られている場合に、GIOP メッセージを分割して送って良いように決められたものです。この Fragment を使う場合は、通常、すべてのメッセージの受信後に開始される復号処理を、Fragment の到着順に始めることが可能になります。これによって、より省資源の計算機上でも CORBA を実装できるようになっています。但し、逐次的に復号化処理を行うかどうかは、CORBA の実装に依存しており、RtORB では現在のところまだ実装は行っていません。そのため、Fragment Message を発行するようなクライアントおよびサーバーに対しては、正常な動作は期待できません。

GIOP1.1 では、Request と Reply の２つのメッセージについて Fragment メッセージが許可されていました。この場合、最初のメッセージは通常の Requst または Reply メッセージであり、その GIOP Message Header の flag の fragment bit(下から２番目のビット)が TRUE(1)になっています。そして、それに続く Fragment メッセージで、最後の Fragment を除いては最初のメッセージと同様に flag の fragment bit が TRUE(1)になっています。すなわち、最後のメッセージのみ flag の fragment bit を FALSE(0)にすること

で、Fragment メッセージの終端を表しています。通常は、受信側はすべてのメッセージを受け取った後に、Request または Reply の処理を開始することになります。

また、途中で CancelRequest Message を送信側に送れば、それ以降の Fragment Message は送信されません。さらに、8 バイト以下の基本データは分割することはできないことになっています。

GIOP1.1 では、それぞれの Fragment メッセージ中のデータの Alignment は、全体の（すべて結合されたときの）Alignment ではなく、Fragment Massage ごとの Aligment を行っていることに注意してください。これは、送信側が全体の Alignment を気にせずに符号化しているということを表しています。

GIOP1.2 以降では、この Request と Reply の他に LocateRequest と LocateReply の Fragment も可能になりました。そして、そのメッセージの構造は下記のように規定されています。

12-bytes GIOP Header	4-bytes GIOP Fragment Header	Variable-length GIOP Fragment Body

また、FragmentHeader は、仕様書には下記のように記載されている。

```
struct FragmentHeader_1_2 {
  unsigned long request_id;
};
typedef FragmentHeader_1_2 FragmentHeader_1_3;
```

このように、GIOP1.2 以降では、FragmentHeader が GIOP Message Header と Fragmanet Message の間に挿入されてるようになりました。この FragmentHeader の request_id は、Fragment の最初のデータの request_id がセットされており、各 Fragment Message がどのメッセージのものであるのかを同定することが可能になりました。さらに、GIOP1.2 以降では、Fragment Message の長さを、最後の Fragment Maessage 以外は必ず 8 バイトの倍数になるように定められました。これによって、GIOP1.1 の時のようにメッセージデータを復元場合に Alignment の調整をする必要がなくなりました。（CDR のすべての Alignment の最小公倍数が 8 であるため、必ず Alignment がそろいます）

上記のことは、GIOP1.1 の場合のみ、Fragment メッセージを復号化するにはオリジナルの request_id が不明になり、Alignment も逐次処理を進めなければならないことを意味しています。そのため GIOP1.1 に置ける Fragment Message の処理には、少し手間がかかることになります。

Java のソースコードを見てみると、GIOP1.1 で Fragment のサポートがされていないのは、このためなのでしょうか？（GIOP1.1 で Fragment Message を送ると SystemError になるように実装されているようです。）

5. CommonData Representation (CDR)

CDR は、OMG の GIOP の仕様で規定されているもので、データの表現情報の一つといえます。この CDR の表現を用いて、リモートオブジェクトへのデータの受け渡しや結果の受信を行っています。この CDR は、すべてのデータを CPU や OS に依存せず送受信に使うための符号化と複合化（marshaling と unmarshaling）し、バイト列にするために定められたものです。

RtORB では、CDR に関する関数は giop-marshal.c の中に記述しています。

5.1. データ型

データの符号化と復号化の説明の前に、CORBA で使用するデータ型について述べます。CORBA で扱うデータ型には、char や int のようなプリミティブ型と struct や union,sequence などのように IDL で設定するコンストラクト型に大別されます。この他にはデータ型そのものを表す TypeCode や CORBA オブジェクトの参照、すべての型を代入できる any 型などがあり、このような型を擬似オブジェクト型と呼んでいます。プリミティブ型は、各 CORBA の実装で予め決められているデータ表現であり、バイト列に符号化する時に置くことができる場所が決められています。これは Alignment と呼ばれており、データの符号化、復号化のプログラムを書くときに気をつけなければいけません。例えば、char の Alignment は 1 であり、double 型の Alignment は 8 になっています。この両者を char, double の順で符号化する場合には、下図のように、char が先頭にあればその後ろの 7 バイトは、何も入らず 9 バイト目から double の値を入れる必要があります。（通常、Alignment とデータ長は同じであるが、CDR では、Alignment は 8 が最大値となっています。

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
char								double							

従って、CORBA で扱うデータ長と Alignment は、基本的に 32 ビットの UnixOS のデータ構造であると考えてほぼ間違いありません。（「ほぼ」と述べたのは、Sequence 型や String 型などには、データ本体の前にデータ長を表す 32 ビット整数を入れたり、struct 型ではそれぞれのデータ型を順に符号化したりするので、厳密にはコンピュータ上のデータ構造とは異なります。）そのため、LongDouble 型のデータでも Alignment は 16 ではなく 8 に なっています。したがって、最近の 64 ビットのマシン上で CORBA を実装するには、データの符号化と複合化処理を実装する場合には注意する必要があります。

5.1.1. プリミティブ型

プリミティブ型は、CORBA で扱う基本的なデータ型です。各データ型のデータ長、Alignment は、下図のように決められています。 また、各データ型の RtORB へのマッピングも示します。

TYPE	データ長	Alignment	RtORB におけるマッピング
char	1	1	Char
wchar	2	1, 2 or 4 for GIOP 1.1 1 for GIOP 1.2 and lator	unsigned short
octet	1	1	Char
short	2	2	Short
unsigned short	2	2	unsigned short
long	4	4	Int
unsigned long	4	4	unsigned int
long long	8	8	なし

unsigned long long	8	8	なし
float	4	4	Float
double	8	8	Double
long double	16	8	なし
boolean	1	1	Char

上記のことから、例えば、char と long の２つのデータをバイト列に符号化する場合には、つぎのようになります。

- char,long の順：1 バイト目に char, 5 バイト目から 8 バイト目に long のデータが格納されます。

0	1	2	3	4	5	6	7
c				Long			

- long,char の順：1 バイト目に long, 5 バイト目に char のデータが格納されます。

0	1	2	3	4
long				C

このようにデータ型をどの順番で符号化かによって、CDR 後のバイト列のデータ長が変わってきます。

5.1.2. コンストラクト型

コンストラクタ型のデータ型では、複数のデータ型の複合型になります。そのため、特に決まった Alignment はなく、構成要素のデータ型によって決まります。すなわち、符号化する際にどのような順でデータに並べられているかによって決まります。コンストラクト型の各データ型の Alignment は、その要素のデータ型の Alignment に従うと思っていればよいと思います。

5.1.2.1. String 型と Wstring 型

String 型と Wstring 型は文字列を表すデータ型であり、C 言語では、それぞれ"char*"と”wchar_t *"にマッピングされています。RtORB では、Wstring 型を"unsigned short *"にマッピングしていますが、これは仮対応ですので正しい動作をするとは限りません。

String 型のデータをバイト列への符号化する場合には、**文字列の長さ+文字列**となります。**文字列の長さ**および**文字列**には、終端記号である**¥0** が含まれるので注意する必要があります（すなわち、strlen で得られる長さに１を足したもの)。このことから、String 型では Alignment は long と同じ ”4” になります。Wstring 型の場合には少し複雑です。そのため、RtORB には、現在のところ該当するプログラムコードが実装されておらず、エラーとなります。CORBA における文字列の扱いは、仕様書の Cord Set Conversion に詳しく書いてありますので、詳細を知りたい場合にはそちらを参照してください。

簡単に言えば、文字列には２種類あり、バイト単位で扱えるもの(byte-oriented)とそうでないもの(non-byte-oriented)があります。バイト単位で扱えるものには、single-byte の ASCII,ISAO 8859-1,EBCDIC 等と muiti-byte の UTF-8,eucJP,Shift-JIS, JIS 等があり、これらのコードを扱うだけで

あれば String 型のサポートで Cord Set Conversion は、何もしないように実装すれば大丈夫です。バイト単位で扱えない文字列には、ISO 10646 USC-2(Unicode)や UTF-16 などの Wstring 型があります。この文字列を取り扱う場合には、クライアントとサーバー間で Cord Set Conversion をどうするかを事前に決めておく必要があります。さらに、文字列の取り扱いについては GIOP の 1.2 以降でその仕様が変更されているために、すべての文字コードに対する処理を実装するには非常にコストがかかります。そのため RtORB では、使用頻度とアプリケーションを考えると、無理に実装する必要ないかと思っています。どうしても必要な場合は、UTF-16 に対する対応ぐらいは考えてもよいと思っています。

5.1.2.2. Struct 型

Struuct 型は、C 言語のマッピングでも構造体になります。すなわち構造体の各データ型をそのまま置き換えるといえます。しかしながら、Alignment は、CDR の符号化の場合には、要素の一番はじめの要素の Alignment に従います。これは、コンピュータ内の実データの Alignment とは異なります。通常、コンピュータ内での構造体の場合には、Alignment は、各要素の Alignement の最大値となります。例えば、char 型と double 型からなる下記の構造体を考えてみます。

```
struct sample{
   char c;
   double d;
};
```

この struct sample のデータ構造は、

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
char								double							

となります。このデータ型と long 型の 2 つのデータを符号化する場合に、struct sample, long の順で符号化すると次のようになります。

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
char								double								long			

このように、struct sample のデータは、16 バイト必要であることがわかります。これとは逆に、long, struct sample の順の場合には、次のようになります。

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
long				char				double							

このように、struct sample のデータは、12 バイトしか必要ではありません。すなわち、struct 型の場合には直前のデータ型に応じて、そのデータ型のデータを格納するために必要なデータ長が変化することになります。一方、通常のコンピュータでは、上記の例では、strcut sample のデータ長は 16 バイトになり、Alignment は、8 になります。

5.1.2.3. Union 型

Union 型は、C 言語の標準マッピングではそのまま union となっています。当初は、RtORB でも union 型へマッピングしていましたが、 OpenRTM-aist で C++をサポートしながら使用するために、下記の例に示すように格納されたデータ型の種類を表す index と構成要素を各要素とした構造体を使用しています。

これは、C++のラッパーを作成するときに「コンストラクタをもつ構造体に共用体を内部に持つこと」が禁止されているためです。
例えば、IDL で下記のような Union 型が定義されたとします。

```
enum NumericType {
   SHORT_TYPE,
    LONG_TYPE,
    FLOAT_TYPE,
    DOUBLE_TYPE
};
union Numeric switch (NumericType){
     case SHORT_TYPE: short short_value;
     case LONG_TYPE: long long_value;
     case FLOAT_TYPE: float float_value;
     case DOUBLE_TYPE: double double_value;
};
```

RtORB の場合にこの定義は、IDL コンパイラにより次のように展開されます。

```
enum {
   SHORT_TYPE,
   LONG_TYPE,
   FLOAT_TYPE,
   DOUBLE_TYPE
};
typedef int NumericType;
typedef Numeric_type Numeric;
struct Numeric_type{
    NumericType __d;
    struct{
      CORBA_short short_value;
      CORBA_long long_value;
      CORBA_float float_value;
      COBRA_double double_value;
   } _u;
};
```

このように Union 型の場合には、構造体として扱われるようになります。
また、Union 型を符号化する場合には、**現在値が入っている型を表す「index(__d)＋データ(_u.xx)」**になるように定義されています。例えば上記の例で、この共用体に CORBA_long 型のデータはセットされていた場合には、CORBA_long 型を表す int 型の“２”と_u.long_value にセットされているデータを符号化することになる。RtORB では、現在値が入っている型を表す index は、enum 型であることを前提にし

ていますので（ほとんどの場合はこれで対応できます)、Union 型の Alignment は、4 となります。

5.1.2.4. Sequence 型

Sequence 型はいわゆる可変長配列であり、C++での vector クラスとよく似ています。例えば、次のような Sequence 型を考えます。

```
struct NmaeValue{
   string name;
   any value;
 };
 typedef seqnece<NameValue> NVList;
```

RtORB では、この定義の Sequence 型は、IDL コンパイラによって下記のように変換されます。

```
typedef struct NameValue_type NameValue;
 struct NameValue_type{
   CORBA_string name;
   CORBA_any value;
 };
   typedef struct CORBA_sequence_NameValue_type{
   CORBA_unsigned_long  _maximum, _length;
   NameValue *_buffer;
   CORBA_unsigned_long _release;
 } CORBA_sequence_NameValue;
 typedef CORBA_sequence_NameValue  NVList;
```

この例のように、Sequence 型のデータは、配列の最大長、配列の長さ、データを格納するデータ配列へのポインタで構成されています。

Seqeunce 型を CDR で符号化する場合には、**Sequence の長さ＋構成されたデータの列**の形式で行われます。これはある意味 String 型を汎用化したものと考えて良いと思います。そのため、Sequence 型の Alignment は、配列長を表す CORBA_long 型の Alignement と同じ”4”になります。また配列の中身を Alignment は、格納データそれぞれの Alignment に従って順次符号化処理を進めることになります。

5.1.2.5. Enum 型

Enum 型は、CORBA では同じ enum にマッピングされます。C 言語の場合には、CORBA_long 型と同じ uint32_t にマッピングされています。そのため、Enum 型の符号化は CORBA_ulong の場合と全く同じになります。

5.1.3. 疑似オブジェクト型

擬似オブジェクト型は、CORBA における”型”を表す TypeCode 型、すべてのデータ型を代入することができる Any 型、例外を表す Exception 型、CORBA オブジェクト参照を表す ObjectReference 型などがあります。また、この他に Principal や Context などもあります。

5.1.3.1. TypeCode 型

TypeCode 型は、CORBA で使用する”データ型”を表したものです。RtORB では、TypeCode 型を下記のような構造体で表しています。この定義は、corba-defs.h に記載されています。

```
typedef struct CORBA_TypeCode_struct{
  void *  parents;
  CORBA_TCKind  kind;
  CORBA_RepositoryId  repository_id;
  CORBA_Identifier  identifier;
  unsigned long  member_count; /* struct, union, enum, value, exception, event */
  CORBA_Identifier  *member_name; /* struct, union, enum, value, exception, event */
  struct CORBA_TypeCode_struct  **member_type; /*struct, union, value, exception, event
*/
  long   *member_label;  /* union */
  struct CORBA_TypeCode_struct  *discriminator; /* union */
  long  default_index;  /* union */
  unsigned long  length;  /* string, wstring, sequence, array */
  struct CORBA_TypeCode_struct  *content_type; /* sequence, array, value_box, alias */
  unsigned short  fixed_digits;  /* fixed */
  short  fixed_scale;    /* fixed */
  CORBA_Visibility  *member_visibility;  /* value, event */
  CORBA_ValueModifier  type_modifier;   /* value, event */
  struct CORBA_TypeCode_struct  *concrete_base_type;   /* value, event */
  short  size;  /* Not used */
  short  alignment;  /* Not used */
}CORBA_TypeCode_struct;
```

この TypeCode の定義自体は、ORBit2 で用いられているものとほぼ同じです。その理由は、RtORB の IDL コンパイラが ORBit2 で用いられてものを基にしているためです。RtORB では、IDL コンパイラで出力されるファイルに独自部分（C++のラッパー等）が付加されるようになっています。また、上記の定義からわかるように、TypeCode の一意性は kind, repository_id, identifier の要素によって決まります。

TypeCode の符号化については、その型によって変わります。先ず kind のみの符号化で済むのは、tk_null, tk_void, tk_short, tk_long, tk_ushort,tk_ulong, tk_float, tk_double, tk_boolean, tk_char, tk_octet,tk_any, tk_TypeCode, tk_Principal, tk_longlong, tk_ulonglong, tk_longdouble といったプリミティブ型と擬似オブジェクト型の一部です。また、kind+(long)0 で符号化されるのは，tk_string, tk_wstring になります。それ以外の TypeCode は、それぞれ符号化の方法が異なっていますので詳細は仕様書を参照して下さい。

5.1.3.2. Any 型

Any 型は、C 言語で言えば汎用ポインタ(void *)に相当するものです。RtORB では、Any 型にセットさ

れるデータの型を表す TypeCode とデータ値を表す汎用ポインタと基本データ型の共用体から構成されています。この Any 型の定義は、corba-defs.h に下記のように記載されています。

```
typedef union  {
    CORBA_boolean val_bool;
    CORBA_octet val_octet;
    CORBA_char  val_char;
    CORBA_unsigned_long val_ulong;
    char * val_str;
    struct CORBA_Object_struct * val_obj;
    void * val_except;     /* tk_struct */
    struct val_t {
      int   len;
      char * data;
    } val_encoded;
} CORBA_any_val;

typedef struct CORBA_any {
    struct CORBA_TypeCode_struct *_type;
    CORBA_any_val *_val;
    CORBA_unsigned_long _release;
} CORBA any ;
```

ここで示している表現は、厳密には完全ではありません。しかし、OpenRTM-aist で用いるには、十分な機能を果たしています。それは、OpenRTM-aist では、現在 Any 型を非常に限定された場所でしか使用していないためです。
Any 型の符号化では、まずそのデータの TypeCode が符号化され、それに続いてデータが符号化されます。

5.1.3.3. Exception 型

Exception 型は、CORBA の関数呼び出し時に発生する例外を表します。RtORB では、Exception は、それを一意にあらわる文字列をもつ構造体として表現されています。この型の符号化・復号化はほぼ Struct 型と同様に動作しますので詳細は省略します。

5.1.3.4. ObjectReference 型

ObjectReference 型は、CORBA オブジェクトを表すための型で、オブジェクト参照と同じです。オブジェクト参照は、The Interoperable Object Reference (IOR)のことであり、マジックナンバー、バイトオーダー。タグ付きのプロファイル、プロトコルのバージョン、サーバーのアドレスとポート番号、リモートオブジェクトを識別するオブジェクトキー、コンポーネントのプロファイルを含んだ印字可能な文字列になっています。

オブジェクト参照は文字列ですが、その符号化処理は通常の文字列の場合と異なり長さを符号化しません。これは、オブジェクト参照のバイト列が、カプセル化されたものになっているからであり、その情報

のみで復号可能だからです。
ちなみに、CORBA オブジェクトのリファレンス表現でよく出てくる　IOR:.....　という表現は、”IOR:” 以降の文字列をバイト列に変換して復号化すればよいことになります。つまり　”01000000”　という文字列は、　01000000　というバイト列にすればよいのです。

5.2. データの符号化（marshaling）

データの符号化は、CORBA オブジェクトのメソッド呼出の時、クライアントとサーバーの双方でデータの受け渡しをするときに行われます。CORBA では、すべてのデータはバイト列に符号化され、上述のように符号化するデータの型によって Alignment が予め決められていることに注意してください。また、データの符号化は、通常、符号化するコンピュータ上のデータ形式で行われますので、メッセージの送受信時には、ヘッダー等に符号化したときのバイトオーダー情報（ビッグエンディアンかリトルエンディアン）が付加されます。

5.3. データの復号化（unmarshaling）

データの復号化は、上述のバイト列で送られてきたデータを元のデータの形に復元することです。送られてきたデータは、通常、送信元のデータ形式で符号化されているために、ヘッダー等に付加されたバイトオーダーフラグに基づき復号化が行われます。但し、カプセル化されたデータに関しては、その中に符号化時のバイトオーダー情報が挿入されていますので、その方式にしたがって復号化します。

6. CORBA を実装するには…

ここまで、CORBA の概要について、述べてきました。では、CORBA を自分で実装するには、どうしたらよいでしょうか？ 上で述べたように、CORBA というのは、IIOP のみを考えると TCP/IP ソケットストリームを使っての GIOP メッセージの交換であり、そのメッセージの 内容に応じて、クライアントやサーバーの振舞うものであると言えます。

そのため、従来のクライアントーサーバー型のプログラム（Web サーバーなど）に GIOP メッセージの解釈ルーチン、データの符号化と復号化、オブジェクトの管理などを加えれば互換性のあるライブラリできそうです。

RtORB では、この考えに基づき、クライアントーサーバーのプログラムにオブジェクト管理の部分や、GIOP メッセージ､データの処理部分を実装しています｡クライアントーサーバーのプログラムに関しては、いろんな実装があると思いますが、以前、事情通ロボットの研究で使っていたライブラリを基本に実装を積み重ねています。そのため、極力スレッドを使わない実装をしていますが、そのためにリモート呼び出しに制限が付いています｡(Pthread が実装されている OS がでは､この制限をなくすこともできますが…)

私の場合には、たまたま C 言語を選択して実装を行いましたが、基本的な考え方はクライアントーサーバープログラミングの応用ですので、クライアントーサーバーのネットワークプログラミングを書くことができる言語であれば、どの言語でも実装できると思います。