RtORB の現バージョンの動作について

産総研　原　功

1.　RtORB において CORBA サーバントを作成する基本的な流れ

ここでは、RtORB における COBRA オブジェクトをどのような手順で作成するかの基本的な流れを説明します。「CORBA サーバント」という言葉が出てきましたが、サーバント（Servant）とは、サーバー機能とクライアント機能を同時に持つようなオブジェクトのことを指します。

以下では、サーバー側のプログラムとクライアント側のクライアントの基本的な流れを説明します。サーバーにもクライアントにもなる CORBA オブジェクトを使うプログラムを実装する場合には、サーバー側のプログラムと同じになります。

1.1.　サーバー側のプログラムの流れ

RtORB において、サーバー側のプログラムは、基本的に下記のような処理を行います。

ORB の初期化：

最初に下記の関数を実行し、RtORB の初期化を行います。

```
orb = (CORBA_ORB)CORBA_ORB_init(&argc, argv, 0, &env);
```

CORBA_ORB_init 関数の返り値は orb ですが、この時大域変数の The_ORB、The_RootPOA がセットされます。

RootPOA の取得：

本来であれば、CORBA_ORB_resolve_initial_references 関数を実行して RootPOA の CORBA オブジェクトを取得し、narrow で RootPOA を取得します。現在は、CORB_ORB_init の時に、大域変数 The_RootPOA にセットされようになっていますので、明示的に処理する必要はありません。

CORBA サーバントの生成：

次に、サーバントを生成します。サーバントの生成する関数は、IDL コンパイラで生成されているはずです。例えば、サーバントのクラス名が Echo だとすると

```
Echo myEcho = (Echo)impl_Echo__create(poa, &env);
catchDefaultException(&env);
```

のように impl_(クラス名)__create という関数が定義されています。またこの例では、catchDefaultException 関数による処理が直後に実行されていますが、この関数は Exception イベントハンドラです。（C 言語では例外の発生はできないため CORBA_Envitonmet という構造体をオペレーション呼び出し時に渡し、正常終了したかどうかの情報が渡されます。）

CORBA サーバントのネームサービスへの登録：

生成した CORBA サーバントを NameServer に登録します。

POAManager の生成：

本来は、POA の状態を管理するのは、POAManager の役割。そのため、POAManager を生成します。現在は POAManager と POA は同一オブジェクトになっており、かつ、POA は、RootPOA しか許さないようになっているので、

```
PortableServer_POA__get_the_POAManager(poa, &env);
```

の結果は、The_RootPOA を返しているだけになります。

POA の活性化：

POAManager を介して POA を有効化します。

ORB の実行：

CORBA_ORB_run 関数を実行して、CORBA のイベントループを起動させます。（ここはほぼ無限ループが動作しており、リモートオブジェクトからのリクエスト処理をします。）

ORB の終了：

CORBA_ORB_run 関数で実行されているイベントループの終了後処理をします。すべての CORBA サーバント、POA、ORB を解放します。

1.2. クライアント側のプログラムの流れ

RtORB におけるクライアントのプログラムは、基本的に下記の手順で処理されます。

ORB の初期化：

サーバー側のプログラムと同様に、最初に CORBA_ORB_init 関数により ORB の初期化を行います。

CORBA オブジェクトの取得：

（IOR が既知の場合）

CORBA_ORB_string_to_object 関数を使って、登録されているオブジェクト参照（IOR）からオブジェクトに変換します。

この関数の返り値は CORBA_Object になっていますので、CORBA_Object__narrow 関数を使ってそのデータ型を特殊化します。（特殊化とは、CORBA_Object 型のデータをその小クラスである任意のデータ型に変更することです。キャスティングに近いですが、そのデータの本来の型チェックはしています）

（ネームサーバーから IOR を取得する場合）

CORBA_ORB_resolve_initial_references 関数を使って NameServer の IOR を取得します。

次に、対象となる CORBA オブジェクト用の CosNaming_Name を生成します。CosNamig_Name は、ネーミングツリーを表すデータです。

最後に、CosNaming_NamingContext_resolve 関数を実行して、対象の CORBA オブジェクト参照を取得し、オブジェクトに変換します。

リモートメッソドの実行：

上記で、取得した CORBA オブジェクトに対してオペレーションのリモート呼び出しを実行する

ORB の終了と破棄：

CORBA_ORB_shutdown 関数で ORB を停止させ、CORBA_ORB_destroy 関数で ORB を破棄します。

1.3. OpenRTM-aist 内での RtORB の動作

OpenRTM-aist では、RT コンポーネントは原則としてサーバントとして存在しています。RT コンポーネントや ORB 等の管理は、RTC_Manager で行っており、CORBA 関連の初期化については RTC_Manager_initORB 関数内で実行されています。

RTC_Manager_initORB 関数では、下記のように処理されます。

【RTC_Manager_initORB 関数】

1. CORB_ORB_init 関数を実行し RtORB を初期化します。この関数への引数は Manager オブジェクトの argc、argv の２つのスロット変数と ORB_id(=NULL)、そして CORBA_Environmet の変数です。argc,argv は、コンポーネント起動時にコマンドラインから渡される引数データです。実行結果は、Manager オブジェクトの m_pORB スロット変数にセットされます。

```
mgr->m_pORB = CORBA_ORB_init(&mgr->argc, mgr->argv, 0, &env);
```

2. Manager オブジェクトの m_pPOA スロット変数に The_RootPOA をセットします。The_RootPOA は、前述の CORB_ORB_init 関数により大域変数にセットされています。

```
mgr->m_pPOA = The_RootPOA;
```

3. PortableServer_POA__get_the_POAManager 関数を実行し POAManager を生成します。そして、Manager オブジェクトの m_pPOAManager スロット変数に POAManager をセットします。現在は POA と POAManager を統合しましたので、PortableServer_POA__get_the_POAManager 関数は The_RootPOA を返すだけの関数です。

```
mgr->m_pPOAManager=PortableServer_POA__get_the_POAManager(mgr->m_pPOA, &env);
```

4. PortableServer_POAManager_activate 関数を実行し POA の活性化します。

```
PortableServer_POAManager_activate(mgr->m_pPOAManager, &env);
```

2. ORB の初期化と RootPOA の取得

次に、RtORB で ORB を初期化している部分について述べます。RtORB では、他の CORBA の実装となるべく同じ API を使えるように実装をすすめました。RtORB が既存の CORBA との互換性を無視して（と言ってもデータ通信はできますが）実装すれば、初期化ルーチンは、例えば、Init_RtORB という関数を作ってしまえば良いと思っています。

以下の関数群は、omniORB や MICO、ORBit2 を使っていた方々のために作った関数であることを補足しておきます。

2.1. ORB の初期化

【関数】

```
CORBA_ORB    CORBA_ORB_init(int *argc, char **argv, char orb_id, CORBA_Environment *env)
```

【動作】

この関数は ORB の初期化を行い、生成した ORB を返します。この関数内で、コマンド引数の処理を行い MICO や omniORB と同じような引数（といっても限定していますが）でのオプション処理をすることができます。

【処理の流れ】

1. RtORB_calloc 関数で、ORB のオブジェクト（構造体）の領域確保し初期化します。
2. 引数(argc, argv)の処理をします。現在サポートしているオプションは、ORBServerId、ORBid、ORBListenEndpoints、ORBNoProprietaryActivation、ORBInitRef、ORBDefaultInitRef、ORBServerPort の 7 つです。また、それぞれのオプションにしたがって ORB 内の cfg 変数を更新します。

   但し、現在のところ ORBInitRef、setDefaultInitRef、ORBServerPort 以外の実装部分はなく、

他のオプションを指定しても、メッセージが出力されるだけである。

3. orb->cfg.init_ref が存在する場合（ORBInitRef でオブジェクトリファレンスをセットした場合）、一覧を表示します。
4. CORBA_Ref_find_url 関数で初期定義に NameService の URL があるかどうかを確認します。もし、何も定義されていない場合には、エラーメッセージを出力して終了します。この処理に関しては、何も定義されていない場合には、localhost:2809 を調べるようにした方がよいと思います。今後、変更する可能性があります。これは、OpenRTM-aist では、起動時に NameService の CORBA オブジェクトがなくても起動し、定期的にネームサーバーに CORBA オブジェクトの登録を実行しているためです。

```
CORBA_Ref_find_url(orb->cfg.init_ref, "NameService");
```

5. Get_IP_Address 関数で動作させるコンピュータの IP アドレスを ORB の hostname 変数にセットします。（通常は、これが EndPoint になりますので、変数名を変更する可能性があります）

```
orb->hostname = (char *)Get_IP_Address();
```

6. PortableServer_POA_new 関数で RootPOA の生成します。

```
PortableServer_POA_new("RootPOA", orb->cfg.port);
```

7. RootPOA の status 変数を POA_HOLDING にセットします。また、この RootPOA を The_RootPOA という大域変数にセットします。
8. The_ORB に生成した ORB オブジェクトをセットし The_ORB を返します

2.2. POA の生成

【関数】

```
PortableServer_POA PortableServer_POA_new(char *id, unsigned short port)
```

【動作】

POA の生成と初期化を行い、生成した POA 構造体を返します。

【処理の流れ】

1. RtORB_calloc 関数により POA のオブジェクト（構造体）の領域確保し初期化を行うます。
2. createHashtable 関数により CORBA サーバント管理用のハッシュ表を生成し、object_map 変数にセットします。

```
poa->object_map = createHashtable(64);
```

3. GIOP_Connection__create 関数により GIOP のサーバー用コネクション構造体を生成します。

```
poa->_server = GIOP_Connection__create();
```

4. GIOP_Connectio__open 関数により GIOP のサーバー用ソケットを生成します。このソケットは、他の CORBA オブジェクトからの IIOP 経由のオペレーションを受信するポートになります。

```
GIOP_Connection__open(poa->_server,
```

5. poa->requests を NULL で初期化します。この変数は、IIOP 経由のオペレーションを一旦ためておくための List になります。
6. set_SockeProfile_arg 関数により GIOP のサーバー用コネクション構造体の引数を表す変数に poa をセットします。RtORB では、IIOP のリクエストを処理する関数(PortableServer_execute_request 関数)は、POA 構造体が渡されることを前提としているためです。

```
set_SockProfile_arg(poa->_server->sock, poa);
```

7. POA の id 変数に引数で与えられた id をセットします。

```
poa->id = (char *)RtORB_strdup(id, "PortableServer_POA_new");
```

2.3. RootPOA の取得

ORB の初期化処理が成功していれば、The_RootPOA という大域変数に RootPOA がセットされていますのでそれを使用します。

3. CORBA サーバントの生成

CORAB サーバントを生成する関数は、IDL ファイル(echo.idl とする)を IDL コンパイラでコンパイルすると echo-skelimpl.c というファイルに生成されています。例えば、サーバントのクラス名が Echo だとすると下記のようになります。

```
CORBA_Echo
 impl_Echo__create(PortableServer_POA poa, CORBA_Environment *ev)
 {
    CORBA_Echo retval;
    impl_POA_Echo *newservant;
    PortableServer_ObjectId objid;

    newservant = (impl_POA_Echo *)RtORB_calloc(1, sizeof(impl_POA_Echo)," create...");
    newservant->servant.vepv = &impl_Echo_vepv;
    newservant->poa = poa;
    POA_Echo__init((PortableServer_Servant)newservant, ev);
     /* Before servant is going to be activated all
      * private attributes must be initialized.   */

     /* ------ init private attributes here ------ */
     /* ------ ---------- end ------------- ------ */

    objid = PortableServer_POA_activate_object(poa, newservant, ev);
    RtORB_free(objid, " objid");
    retval = PortableServer_POA_servant_to_reference(poa, newservant, ev);

    return retval;
 }
```

この関数は、Echo がスロット変数を何も持っていなければ、このまま定義で問題ありません。もし、独自のスロット変数を定義した場合には、この関数内に初期化処理を追加してください。
従って、Echo サーバントを生成するには、下記のように記述します。

```
Echo myEcho = (Echo)impl_Echo__create(poa, &env);
```

3.1. 通常の CORBA サーバントの生成手順

前述した CORBA サーバント Echo について、詳しく見ていきます。RtORB 内での CORBA サーバントの生成プロセスは、ほぼすべて下記の通りです。

1. RtORB_calloc 関数で CORBA サーバントの構造体を生成し newservant という変数にセットします。このとき構造体の初期化もします。
2. newservant の内部変数の servant.vepv に&impl_Echo_vepv をセットします。impl_Echo_vepv は、Echo CORBA サーバントのメソッド関数に関する定義が格納された構造体です。

   ```
   newservant->servant.vepv = &impl_Echo_vepv;
   ```

3. newservant の内部変数の poa に引数で与えられた poa をセットします。この処理は、poa が複数あることを前提にされた為です。既に、RtORB では、RootPOA しか使わないという前提をおいていますので、この処理は将来なくす予定です。

   ```
   newservant->poa = poa;
   ```

4. POA_Echo__init 関数を実行して、CORBA サーバントの初期化を行います。この関数は、echo-skels.c の中に自動生成されています。

   ```
   POA_Echo__init((PortableServer_Servant)newservant, ev);
   ```

5. その他のユーザ定義のメンバーの初期化処理をここで行います。

   ```
   /* ------ init private attributes here ------ */
   /* ------ ---------- end ------------- ------ */
   ```

6. PortableServer_POA_activate_object 関数によって CORBA サーバントを活性化し、ObjectId を返します。RtORB では、この ObjectId を object_key(サーバント生成時に自動的に付けられる ID)にしています。

   ```
   objid = PortableServer_POA_activate_object(poa, newservant, ev);
   ```

7. objid は、今後使わないのでを解放します。(この処理は、必要ないはずですが IDL コンパイラが自動生成しています。次期バージョンでは削除する予定です)

   ```
   RtORB_free(objid, " objid");
   ```

8. PortableServer_POA_servant_to_reference 関数を実行して CORBA サーバントを CORBA オブジェクトに変換します。

   ```
   retval = PortableServer_POA_servant_to_reference(poa, newservant, ev);
   ```

9. 上記で変換した CORBA オブジェクトを返り値として返します。

3.1.1. CORBA サーバントの初期化（POA_Echo__init）

POA_Echo__init 関数は echo-skels.c に定義されており、IDL コンパイラによって自動生成されています。通常は、下記のように ClassInfo の設定とサーバント構造体の初期化し、RtORB_POA_Object を生成しています。

```
void POA_Echo__init(PortableServer_Servant servant, CORBA_Environment *env)
{
  static PortableServer_ClassInfo class_info = {
```

```
        NULL,
        (void*(*)())&get_skel_small_Echo,
        "IDL:Echo:1.0",
        &Echo__classid,
        &Echo__imp
    };
    PortableServer_ServantBase__init (((PortableServer_ServantBase *)servant), env);
    RtORB_POA_Object__create (&class_info, servant, NULL, NULL);
}
```

これから分かるように、サーバントの初期化は、次の手順で行われます。

1. 最初に、PortableServer_ClassInfo を定義します。これは、repositoy_id や実装関数、クラス ID の定義等が含まれている構造体です。
2. PortableServer_ServantBase__init 関数を呼び出し、すべてのサーバントに共通な初期化を行います。
3. 最後に RtORB_POA_Object__create 関数を呼び出し、RtORB_POA_Object 構造体を生成します。この RtORB_POA_Object は、POA の object_map に登録されているオブジェクトです。

また、RtORB では、PortableServer_ClassInfo、RtORB_POA_Object を下記のように定義しています。

```
typedef struct {
    void (*dummy)();              // ダミー関数、初期化関数を入れる予定だった
    void *(*impl_finder)();       // メソッド名にしたがって、実装の関数ポインタを返すような関数
    const char *class_name;   // クラスの ID（IDL ファイル名から?）
    uint32_t *class_id;           // クラス ID の整数値のはずだが、現在すべて 0 にセットされる。
    CORBA_Class_Impl *class_impl;   //クラスの実装情報、Typecode,クラス名、メッソド構造体、クラスのリポジトリ ID など
}PortableServer_ClassInfo;

typedef struct{
    struct {
        void *_private;
        void *_vepv;
    } impl_serv;                          // サーバントの情報（ServantBase と同じ）
    PortableServer_POA poa;               //POA の情報
    PortableServer_Servant servant;     //サーバントへのポインタ？ ここまので定義は、impl_POA_ServantBase と同じ。
    CORBA_Object obj;                     // CORBA オブジェクトへのポインタ
    void *_reserve;                       // 予備
}RtORB_POA_Object;
```

この他にサーバントを表す構造体として下記のような定義があります。

```
typedef void *PortableServer_Servant;

typedef struct {
   void *_private;
   PortableServer_ServantBase__vepv *vepv;
} PortableServer_ServantBase;

typedef struct{
   PortableServer_ServantBase servant;
   PortableServer_POA poa;
   void *_private;
}impl_POA_ServantBase;
```

例えば Echo の場合には、IDL コンパイラより下記のように生成されています。

```
typedef struct {
   void *_private;
   POA_Echo__vepv *vepv;
} POA_Echo;

typedef struct {
    POA_Echo servant;
    PortableServer_POA poa;

    /* ------ add private attributes here ------ */
    /* ------ ---------- end ------------ ------ */
} impl_POA_Echo;
```

これらのことから、RtORB_POA_Object 構造体は、impl_POA_Echo、 impl_POA_ServantBase、PortableServer_ServantBase にキャストすることができるようになっていることがわかります。

3.1.1.1. PortableServer_ServantBase__init 関数

この関数 CORBA サーバントの初期化の時に呼ばれ、すべてのサーバントに共通な初期化関数です。引数で与えられたオブジェクトに対して最低限必要なメソッド finalize、default_POA、is_a、non_existent、add_ref、remove_ref を調査し、セットされていなければデフォルトの関数ポインタをセットします。

3.1.1.2. RtORB_POA_Object__create 関数

この関数 CORBA サーバントの初期化の時に呼ばれ、RtORB_POA_Object 構造体の生成と初期化を行います。引数で与えられた PortableServer_ClassInfo、PortableServer_Servant から RtORB_POA_Object

を生成します。この関数における処理の流れは下記の通りです。

1. 返り値となる poa_obj を RtORB_alloc 関数で領域確保し、memset ですべて 0 に初期化する。
2. 引数で与えられた PortableServer_ClassInfo の値を内部変数の impl_serv._private にセットします。

```
poa_obj->impl_serv._private = info;
```

3. new_CORBA_Object 関数で、新しい CORBA オブジェクトの構造体を生成し、内部変数の obj にセットします。

```
poa_obj->obj = (CORBA_Object)new_CORBA_Object(NULL);
```

4. GIOP_Connection__create 関数で GIOP の接続（新規サーバーSocket）を生成し、CORBA オブジェクト obj の内部変数の connection にセットします。

```
poa_obj->obj->connection = GIOP_Connection__create();
```

5. CORBA オブジェクト obj の内部変数の_ur を初期化します。

```
poa_obj->obj->_url = NULL;
```

6. CORBA オブジェクト obj の内部変数の num_urls を初期化します。

```
poa_obj->obj->num_urls = 0;
```

7. CORBA オブジェクト obj の内部変数の impl にクラスの実装情報をセットします。

   poa_obj->obj->impl = info->class_impl;

8. クラス名をコピーし、CORBA オブジェクト obj の内部変数の typedId にセットします。

```
poa_obj->obj->typedId = RtORB_strdup(info->class_name, "RtORB_POA_Object__create");
```

9. poa_obj の内部変数 servant と CORBA オブジェクト obj 内部変数 servant に引数で与えられた PortableServer_Servant にセットします。

```
poa_obj->servant = servant;
poa_obj->obj->servant = servant;
```

10. 引数で与えられた PortableServer_Servant の_private に生成した RtORB_POA_Object をセットします。

```
sb->_private = poa_obj;
```

3.1.2. PortableServer_POA_activate_object 関数

この関数は、引数で与えられた PortableServer_Servant を活性化します。先ず、CORBA オブジェクトを会えるために、引数で与えられた PortableServer_Servant から RtORB_POA_Object を抽出します。PortableServer_ServantBase は、PortableServer_Servant をキャストするだけで良いですし、RtORB_POA_Object は、その内部変数である_private にセットされているはずですので、この_private の中に格納されたポインタを RtORB_POA_Object にキャストします。
すると RtORB_POA_Object の obj に CORBA オブジェクトがセットされているはずですので、の内部変数である object_key を変数 ObjectId にコピーします。その後、register_PortableServer_Servant 関数を呼び出し、サーバントを POA に登録後 ObjectId を返り値として返します。

3.1.2.1. register_PortableServer_Servant 関数

この関数は、サーバントの活性化を行うときに呼ばれる関数です。引数で与えられた PortableServer_POA の object_map に、PortableServer_Servant の RtORB_POA_Object を登録します。この関数の処理流れは、

下記の通りです。

1. 最初に、引数で与えられた PortableServer_Servant から CORBA オブジェクトを取得し obj にセットします。(本来なら sb、sb->_private が NULL 出ないかチェックすべきです)

```
obj = ((RtORB_POA_Object *)sb->_private)->obj;
```

2. 同じく RtORB_POA_Object も取得して poa_obj にセットしておきます。(ここも、本来なら sb が NULL 出ないかチェックすべきです。上と順番を入れ替えるべきですね)

```
poa_obj = (RtORB_POA_Object *)sb->_private;
```

3. IOR を格納用のバッファ ior を確保し、初期化します。現在のところバッファは 1024bytes と固定値にしています。
4. registerItem 関数で、POA の object_map に、RtORB_POA_Object を登録します。

```
registerItem(poa->object_map, obj->object_key, poa_obj);
```

5. 次に、createIOR 関数でサーバントの IOR を生成し ior にセットします。ここの動作は、POA の Socket でリクエストを受けるとすれば、POA 生成時に typeId と object_key 以外は、固定になりますので別の関数で IOR を生成した方が効率的ですだと思います。次期バージョンで修正予定です。

```
createIOR(ior,poa->_server->sock,(char *)obj->typedId,(char *)obj->object_key);
```

6. ior コピーして CORBA オブジェクトにセットしておきます。これは、CORBA オブジェクトからオブジェクト参照を作るときに参照される文字列です。しかし、上記のことを考えると予め作っておかなくても良いかもしれません。その分メモリ消費量を下げる事ができますので。今後どうするかは、実際に実装したアプリケーション依存になるかと思います。

```
obj->_ior_string = (unsigned char *)RtORB_strdup(ior, "register_PortableServer_Servant");
```

7. ior はローカル変数でしたのでを開放します。

3.1.3. PortableServer_POA_servant_to_reference 関数

この関数は、引数で与えられた PortableServer_Servant から CORBA オブジェクトを得てそれを返します。

```
sb = (PortableServer_ServantBase *)servant;
poa_obj = (RtORB_POA_Object *)sb->_private;
return poa_obj->obj;
```

この関数でも servant, sb が NULL でないかをチェックする必要がありますね。

4. CORBA サーバントのネームサービスへの登録

CORBA サーバントのネームサービスへの登録する場合には、NameServer の rootContext に名前付きで登録する方法と rootContext に独自の NamingContext を生成して登録するという 2 つの方法があります。

4.1. rootContext に登録する場合

1. CORBA_ORB_resolve_initial_references 関数により NameServer の rootContext を取得します。

```
CORBA_ORB_resolve_initial_references(orb, "NameService", env);
```

2. ネームサービスに登録する名前のための CosNaming_Name 構造体を CosNaming_Name__alloc 関数でメモリ領域を確保します。CosNaming_Name__alloc 関数は、CosNaming.idl を IDL コンパイラによって自動生成されたコードです。CosNaming.h に記載されています。

```
objectName = CosNaming_Name__alloc();
```

3. ネームサービスに登録する名前のための配列用バッファを CosNaming_Name_allocbuf 関数で領域確保します。ここでは、rootContext に登録しますので、引数は 1 となっています。階層的にネーミングコンテキストを生成して、名前を登録したい場合には、その階層数が引数になります。但し、下記で使用する CosNaming_NamingContext_bind 関数は、階層的な CosNaming_Name を引数として渡すと拡張されたネームサーバーでなければ例外が発生しますので注意が必要です。

```
objectName->_buffer = CosNaming_Name_allocbuf(1);
```

4. 前述の配列バッファの長さを設定します。ここでは、1 となっています。

```
objectName->_length = objectName->_maximum = 1;
```

5. 次に、登録する名前を設定します。この例では、ID=Echo, Kind=Object にしています。OpenRTM-aist の場合には、kind=rtc としています。

```
objectName->_buffer[0].id = "Echo";
objectName->_buffer[0].kind = "Object";
```

6. CosNaming_NamingContext_bind 関数を実行し CORBA オブジェクトをネームサーバーに登録します。この時、既に同じ名前で他の CORBA オブジェクト登録済みの場合には、ex_CosNaming_NamingContext_AlreadyBound の例外が発生しますので、catchException 関数でその例外をハンドリングすべきです。

```
CosNaming_NamingContext_bind(rootContext, objectName, obj, env);
```

7. ex_CosNaming_NamingContext_AlreadyBound が発生した場合には、CosNaming_NamingContext_rebind 関数を用いて上書きします。

4.2. 独自の NamingContext を生成して登録する場合

1. CORBA_ORB_resolve_initial_references 関数により ORB に登録されている NameServer の rootContext を取得します。

```
CORBA_ORB_resolve_initial_references(orb, "NameService", env);
```

2. ネーミングサーバーに独自の NamingContext を作成するための CosNaming_Name 構造体を CosNaming_Name__alloc 関数で領域確保します。

```
contextName= CosNaming_Name__alloc();
```

3. NamingContext の実体を格納するための配列バッファを CosNaming_Name_allocbuf 関数で領域確保します。この例では、引数として 1 を渡しています。

```
contextName->_buffer = CosNaming_Name_allocbuf(1);
```

4. 前述の NamingContext の実体を格納するための配列バッファの長さをセットします。この例では、1 になります。

```
contextName->_length =contextName->_maximum = 1;
```

5. 次に NamingContext の値を代入します。この例では、ID=test. King=my_context にしています。OpenRTM-aist では、NamingContext の Kind を host_cxt としています。

```
contextName->_buffer[0].id = "test";
contextName->_buffer[0].kind = "my_context";
```

6. CosNaming_NamingContext_bind_new_context 関数で新規 NamingContext をネーミングサーバーに登録します。この時、既に同じ NamingContext がネーミングサーバーに登録済みの場合

には、ex_CosNaming_NamingContext_AlreadyBound が発生します。その場合は、catchException 関数でハンドリングし、CosNaming_NamingContext_resolve 関数でネームサーバー上の NamingContext を取得します。

7. その後は、上記と同様な手続き必要なだけ NamingContext の階層を作成して、rootContext に名前を登録する場合の処理と同じ手続でネームサーバーに CORBA オブジェクトを登録します。

上記の場合には、ex_CosNaming_NamingContext_AlreadyBound が発生する可能性があるが、NamingContextExt というが拡張機能の実装がネームサーバーに実装されてれば、NamingContext の配列を 2 以上でセットし、CosNaming_NamingContext_rebind 関数を使って強制的に上書き登録をすることができるようになっています。OpenRTM-aist では、この拡張機能を用いて、rebind を常に行っています。そのため、RtORB の NameService では、この拡張機能を実装しています。

5. POAManager の生成

本来は、POA の状態を管理するのは、POAManager の役割です。通常は、POA を引数とする PortableServer_POA__get_the_POAManager 関数によって POAManager を獲得するのですが、現在の RtORB では POAManager と POA は同一オブジェクトになっており、かつ、POA は、RootPOA しか許さないようになっています。そのため、PortableServer_POA__get_the_POAManager 関数による実行結果は、The_RootPOA と同じになっています。

6. POA の活性化

POA を活性化する場合には、POAManager を介して行います。C++では、POAManager の activate オペレーションで行いますが、RtORB では、PortableServer_POAManager_activate 関数を用いて行います。

```
PortableServer_POAManager_activate(poa_manager, &env);
```

しかしながら、RtORB の現在の実装では POAManager と POA は同一オブジェクトと定義していますので、この関数は、PortableServer_POA_activate 関数を内部で呼んでいるのみです。

6.1. PortableServer_POA_activate 関数

この関数は、引数で与えられた PortableServer_POA の状態をチェックし、対応する Soket 記述子にリクエストキューを生成します。また、PortableServer_enqueue_request 関数を set_SockProfile 関数で、大域変数で struct sockport_profile SockProfile の内部変数である command_proc に登録します。

また、POA を活性化するには、その状態が POA_HOLDING である必要があります。

6.2. set_SockProfile 関数

この関数は、引数で与えられた Socket 記述子に対応した sockport_prpofile 構造体 SockProfile に、type、connection_proc、disconnect_proc、command_proc の各コールバック関数をセットします。

ここで、type は、SOCK_CLOSED, SOCK_SERVER, SOCK_SERVICE, SOCK_CLIENT のどれかになります。CORBA サーバントの場合には、SOCK_SERVER がセットされるようになっています。

6.3. POA の状態

RtORB では、POA は、下記の 5 つの状態を持つように定義しています。

- POA_HOLDING：　POA が生成された状態
- POA_ACTIVE：　　POA が活性化している状態
- POA_INACTIVE：POA が非活性の状態
- POA_DISCARDING：POA が終了する前段階
- POA_TERMINATE：POA が終了している状態

7. CORBA オブジェクトの取得

7.1. IOR 又は ObjectURL が既知の場合

リモート CORBA オブジェクトに対するオペレーションを実行するには、クライアント側では CORBA オブジェクト参照の生成を行います。リモート CORBA オブジェクトに対するオペレーションを実施するには、その所在とオブジェクト ID を含んでいる IOR または ObjectURL が既知でなければいけません。IOR とは、Interoperable Object Reference の略であり、接頭辞 IOR: の後に 16 進数の文字列が並ぶ文字列です。また ObjectURL は、Object Uniform Resource Locator のことであり、インターネット上のリソースを特定するための文字列です。この ObjectURL の表記は、インターネットのアドレスを表した URL と似たようなものであり、RtORB ではこの ObjectURL として Corbaloc と呼ばれる表示形式をサポートしています。

上述の IOR または ObjectURL を CORBA_ORB_string_to_object 関数に与えることでリモート CORBA オブジェクト参照を生成することができ、そのオブジェクト参照を介してサーバーにある CORBA オブジェクトにアクセスすることができます。

7.2. IOR について

IOR には、TypeID と 1 つ以上のプロファイルから構成されています。プロファイルには、使用する GIOP のバージョン、IP アドレス、ポート番号等が含まれています。

ちなみに、RtORB では、IOR は、createIOR 関数で生成することができ、

```
"IOR:" + byte_order(octet) +TypedID(string) +    Number of TaggedProfiles = 1 (long)
        + IOP_TAG_INTERNET_IOP(long) + Size of TaggedProfile(long)+ TaggedProfile
```

となっています。

ここで、TaggedProfile は、

```
byte_order(octet) + version.major(octet) + version.minor(octet) +IP Address(string)
      + PortNo. (short) + object_key(string) + Number of TaggedComponent(octet) + TaggedComponent
```

となっており、カプセル化されたバイト列になっています。

RtORB では、TaggedComponent を、

```
ORB_TYPE(long) +ORB_TYPE data(sequence) + CODE_SETS(long) + CODE_SET data (sequence)
```

として生成しています。

IIOP Complete という書籍では、IOR の例として下記のようなものがあげられています。

```
IOR:000000000000000e49444c3a48656c6c6f3a312e300000000000000100000000000000003a000100000000000f3134302e31383
82e31382e32313900201389000000000001a4f422f49442b4e554d0049444c3a48656c6c6f3a312e30003000
```

この文字を整形してみると、下記のようになります。

```
IOR:
00 00 00 00 00 00 00 0e 49 44 4c 3a 48 65 6c 6c
6f 3a 31 2e 30 00 00 00 00 00 00 00 01 00 00 00 00
```

```
00 00 00 3a 00 01 00 00 00 00 00 0f 31 34 30 2e
31 38 38 2e 31 38 2e 32 31 39 00 20 13 89 00 00
00 00 00 1a 4f 42 2f 49 44 2b 4e 55 4d 00 49 44
4c 3a 48 65 6c 6c 6f 3a 31 2e 30 00 30 00
```

すると、この例の IOR には下記の情報が記載されていることがわかります。

```
byte_order = 0
TypedID = IDL:Hello:1.0 (0x49,0x44,0x4c,0x3a,0x48,0x65,0x6c,0x6c,0x6f,0x3a,0x31,0x2e,0x30,0x00), len=0x0e
Number of TaggedProfiles = 1 (0x00000001)
IOP_TAG_INTERNET_IOP = 0 (0x00000000)
Size of TaggedProfile = 58 (0x3a)
bye_order = 0
version.major=1
version.minor=0
IP Address = 140.188.18.219 (0x31,0x34,0x30,0x2e,0x31,0x38,0x38,0x2e,0x31,0x38,0x2e,0x32,0x31,0x39,0x00),
len=0x0f
PortNo. = 5001 (0x1389)
object_key = OB/ID+NUM IDL:Hello:1.0 0 (0x4f,0x42,0x2f,0x49,0x44,0x2b,0x4e,0x55,0x4d,0x00,0x49,0x44,
0x4c,0x3a,0x48,0x65,0x6c,0x6c,0x6f,0x3a,0x31,0x2e,0x30,0x00,0x30,0x00), len=0x1a
```

すなわち、この IOR は、140.188.18.219 のマシン上にポート番号 5001 という Socket ポートで "OB/ID+NUM IDL:Hello:1.0 0" という名前のオブジェクトで"IDL:Hello:1.0"のインスタンスであるということになります。また、ここで示されているコンピュータが、GIOP1.0 のメッセージを受付けビッグエンディアンで符号化を実施するコンピュータであることもわかります。

ちなみに、CORBA のネーミングサーバーに登録される情報というのは、この形式の文字列と名前のペアになります。

7.3. ネームサーバーから IOR を取得する場合

また、リモート CORBA オブジェクの参照を生成するには、CORBA のネーミングサーバーを用いる方法もあります。RtORB では、CosName の Stub もライブラリで提供していますので、ネーミングサーバーの IOR または ObjectURL が既知の場合には、以下のような手続きでリモート CORBA オブジェクの参照を得ることができます。

1. CORBA_ORB_resolve_initial_references 関数等を使ってネーミングサーバーの IOR を取得する。
2. 対象となるリモート CORBA オブジェクトを表す CosNaming_Name を生成する。
3. CosNaming_NamingContext_resolve 関数で対象のリモート CORBA オブジェクトを取得する。

8. ORB の実行と IIOP の低レベル処理

次に、RtORB におけるリモート CORBA オペレーションがどのように実行されているかを見てみます。RtORB では、外部からの CORBA オペレーションは、CORBA_ORB_run 関数を実行することにより CORBA のイベントループを起動されます。

8.1. CORBA_ORB_run 関数

この関数では、POA_main_loop 関数を呼んでいるのみである。

```
POA_main_loop( The_RootPOA );
```

8.2. POA_main_loop 関数

この関数では、server_loop 関数を呼んでいるのみであり、引数には、select システムコールでの最大待ち時間、外部からのリクエストを処理する関数、RootPOA をとります。

```
server_loop(10000.0, PortableServer_execute_request, rootPOA);
```

8.3. server_loop 関数

この関数は、RtORB のもっとも低レベルの通信レイアでのイベントループになります。
この関数の処理の流れは、下記の通りです。

1. init_socket_servers 関数を用いて、GIOP で使用する fd_set 構造体を初期化します。

   ```
   init_socket_servers();
   ```

2. 次に、select システムコールで使用する time_out 構造体に引数で与えられた msec の時間をセットします。
3. そして、以下の処理に対する繰返しループを実行します。
   i. time_out の時間、または GIOP リクエストの到達まで待ち、リクエストに対する処理を実行します。

      ```
      select_socket_servers(time_out);
      ```

   ii. 第２引数の関数ポインタが NULL でなければ、第２引数の関数を第３引数のポインタを引数にして関数を実行します。すなわち、PortableServer_execute_request(The_RootPOA) が実行されて、イベントキューに溜まったメッセージに応じて、リクエスト処理を行います。

      ```
      (*idle)(arg);
      ```

8.4. init_socket_servers 関数

この関数は、GIOP で使用する fd_set 構造体を初期化します。この関数の処理の流れは下記の通りです。

1. get_Servers_from_SockProfile 関数によって、sockport_profile の type が SOCK_SERVER または SOCK_SERVICE のものを抽出します。

   ```
   get_Servers_from_SockProfile(sockServers, &nSock, 0);
   ```

2. FD_ZERO マクロによって、 select システムコールで使用する fd_set 構造体を初期化します。

   ```
   FD_ZERO(&main_socket_bits);
   ```

3. sockport_profile の type が SOCK_SERVER のものを fd_set 構造体にセットします。

8.5. get_Servers_from_SockProfile 関数（要 修正）

この関数は、init_socket_servers 関数の中で呼ばれ、sockport_profile の type が SOCK_SERVER または SOCK_SERVICE のものを抽出します。SockProfile は、予め大域変数として定義された sockport_profile 構造体です。また、FD_SETSIZE は、OS ごとに定義されている最大の fd_set の数と定義されています。

8.6. select_socket_servers 関数

この関数は、select_socket_ports 関数を呼び出しているのみです。

```
select_socket_ports( &main_socket_bits, time_out);
```

8.7. select_socket_ports 関数

この関数は、select システムコールで、外部からの GIOP リクエストを待ち、接続要求の場合には GIOP 用の socket を生成し、そうでなければ予め SockProfile の内部変数 command_proc に設定されているコールバック関数を実行します。また、command_proc が NULL または実行時にエラーが発生した場合には、その接続を閉じます。この関数の処理の流れは下記の通りです。

1. select システムコールで使う fd_set socks を FD_ZERO 関数で初期化します。
2. 引数で与えられた select システムコールで入力待ちになるソケット記述子のセット fd_set sockbits を、先ほど初期化した socks にコピーします。
3. 引数で与えられた time_out 構造体をコピーします。このコピーが select システムコールに渡されます。
4. select システムコールを呼び出し、GIOP 経由の接続要求またはメッセージ到着のイベントを待ちます。
5. select システムコールで、イベントが来た socket 記述子すべてに対して以下の処理を行います。
   I. active_port に 0 をセットし、イベントが来た socket 記述子を取得します。
   II. SockProfile 配列から active_port に対応する command_proc の関数ポインタを取得しておきます。
   III. SockProfile 配列から active_port に対応する type を取得し、それに応じて次の処理を行う。
      ① type が SOCK_SERVER の時：
         - これは、新規の接続要求である。そのため accept_connection 関数で GIOP 通信用の socket を生成する。この時 active_port に対応する connection_proc がセットされていれば、active_port と arg を引数として関数を実行する。この connection_proc には、POA の活性化時に、PortableServer_enqueue_request 関数がセットされているはずです。
         - accept が成功すれば、新規に生成された socket 記述子に対応する SockProfile に対して copy_sockport_profile 関数でコールバック関数等をコピーし、type を SOCK_SERVICE にセットします。さらに、その socket 記述子を FD_SET 関数で sockbits にセットします。
         - 次に、FD_CLR で socks から active_port を除去し、イベント待ちの socket 記述子の数を 1 つ減じます。
      ② type が SOCK_CLIENT または、SOCK_SERVICE の時：
         - active_port に対応する command_proc の関数を GIOP_ConnectionHandler h={CONNECTION_TYPE_SOCKET, active_port} を引数として実行します。この時処理が失敗すれば、コネクションが切れているということになりますので active_port に対応する disconnect_proc がセットされていればそれを実行します。そして、active_port を閉じ、sockbits から active_port をクリアするとともに type を SOCK_CLOSED に変更します。
         - 次に、FD_CLR で socks から active_port を除去し、イベント待ちの socket 記述子の数を 1 つ減じます。
      ③ 上記以外の時：
         - エラーメッセージを表示し、イベント待ちの socket 記述子の数を 1 つ減じます。

8.8. PortableServer_execute_request 関数

この関数は、GIOP メセージを処理します。この関数が CORBA のリモートメソッド呼び出しにおけるメインの関数になるといえます。この関数に引数で渡されるポインタは POA であるはずです。
また GIOP のリクエストは、PortableServer_enqueue_request 関数により、GIOP メッセージ到着時に poa->requests にストックされていますので、そのリクエストキューに対して

```
GIOP_execute_request(poa, poa->requests);
```

を実行します。

8.9. GIOP_execute_request 関数

この関数は、GIOP メセージをすべて処理します。この関数の処理の流れは下記の通りです。

1. 引数であたえられた PtrList *lst（poa->requests）の各要素に対して以下の処理を実行します。
   I. リクエストキューからその要素の中身を取り出し、request にセットします。
   ```
   request = (GIOP_Request_Item)current_request->item;
   ```
   II. 取り出した要素 request の中の GIOP メッセージを GIOP メッセージヘッダー型の head にキャストします。
   ```
   header = (GIOP_MessageHeader *)request->buf;
   ```
   III. また、request の中の GIOP メッセージを buf にセットします。
   ```
   buf = request->buf;
   ```
   IV. request の中のコネクションハンドラを変数 h にセットします。
   ```
   h = request->connh;
   ```
   V. 変数 version に GIOP メッセージのバージョンをセットします。
   ```
   version = header->minor;
   ```
   VI. GIOP メッセージのバイトオーダーのフラグ（header->flags & 0x01）をチェックして変数 byte_order にセットします。byte_order が異なる場合には、GIOP ヘッダーに記載されたメッセージサイズを変換します。
   VII. 次に、送られてきたメッセージのタイプ（header->message_type）に応じて下記の処理を行います。

- header->message_type == GIOP_Request の時：このメッセージは、通常の CORBA オペレーションコールですので、下記の動作をします。
  - ✧ RtORB がスレッドをサポートしてコンパイルされている場合には、該当リクエストのコピーを作って RunThread 関数で新規スレッドを生成し GIOP_Request_perform 関数を実行します。
  - ✧ スレッドをサポートしていない場合には、GIOP_Request_perform 関数を実行し、この関数の終了を待ちます。
- header->message_type == GIOP_Reply の時：これは、返信が来たときの処理であり GIOP_Reply_perform 関数を実行します。ただし、BidrectionalIIOP でない限りこの部分には、到達しません。OpenRTM-aist で使用する場合には、到達しない処理になります。

- header->message_type == GIOP_CancelRequest の時：これは、外部オブジェクトからリクエス

トの中断を知らせる関数です。GIOP_CancelRequest_perform 関数を実行します。この部分は、通常、Fragment メッセージの送信の中断や現在の処理の実行の中断の時に使いますが、RtORB では、リクエストの実行中はこのメッセージのハンドリングはできませんし、Fragment の処理も実装していませんので、この部分に入ることはありません。

- header->message_type == GIOP_LocateRequest の時：これは、CORBA オブジェクトの所在を確認するときに入ります。GIOP_LocationRequest_perform 関数を実行します。
- header->message_type が GIOP_LocateReply の時：これは、上記の GIOP_LocateRequest に対する返答があったときにこの部分に入り、GIOP_LocationReply_perform 関数を実行します。これは、GIOP_Reply の時と同様に BidirectionalIIOP の時しか入りませんので、RtORB では未到達になります。
- header->message_type が GIOP_CloseConnection の時：これは、GIOP のコネクションを閉じる時にコールされますが、RtORB では何もしません。
- header->message_type が GIOP_MessageError の時： これは、GIOP メッセージが処理できないときの処理です。GIOP のバージョンミスマッチなどの時に返されます。本来は、GIOP メッセージのバージョンを下げて再送等を実装するのですが、RtORB でリクエストを出すときには、GIOP1.0 にしていますので、再送機能は実装していません。そのため、"Message Error occured¥n"のメッセージを表示するだけになっています。
- header->message_type が GIOP_Fragment の時：これは、GIOP メッセージが分割して送られてくるときに入ります。Java などの実装では、通信バッファが非常に小さいために、GIOP1.1 以上のメッセージに対応する場合には、この部分に入ります。しかし RtORB では、Fragment に関する処理は未実装になっていますので、"FragmentMessage have not implemented.....¥n" のメッセージを表示するのみです。この部分に入らないように、IOR を GIOP1.0 のみに対応したり、GIOP_Request 時に、GIOP1.0 でなければメッセージエラーを返すような処理を追加したりする必要があります。
- 上記以外の場合：基本的に、この部分には到達しないはずですが、 "Invalid message type ([header->message_type])¥n"というメッセージを表示することにしています。

VIII. 1 つの GIOP メッセージに対する処理が終わったので内部変数 release に 1 をセットします。

```
current_request->released = 1;
```

IX. リクエストキュー内の次の要素を current_request にセットします。

```
current_request = current_request->next;
```

2. すべてのリクエストに対する処理が終了したら（current_requste == NULL の場合）、PtrList_remove_released_items 関数で処理が終わったリクエストを削除します。

9. CORBA オブジェクト間の通信時の振舞

次に、RtORB における CORBA オブジェクト間の通信の処理について述べていきます。下記は、クライアントの CORBA オブジェクトからサーバーの CORBA オブジェクトのオペレーションを呼び出す手順です。

- **リモート CORBA オブジェクトの取得**

先ず、クライアント側では、リモート CORBA オブジェクトへの参照の生成を行います。CORBA オブジェクトの参照の生成は、**「7.CORBA オブジェクトの取得」**で述べたように、直接 IOR を与えるかネ

ーミングサービスに問い合わせてオブジェクト参照（スタブオブジェクト）を生成します。

- **リモート CORBA オブジェクトのメソッド呼び出し**

上記の手続きでリモート CORBA オブジェクトの参照　C_Obje　が生成できたとすると、次には、リモート CORBA オブジェクトのメソッドの呼び出しを行います。

通常、C++等のオブジェクト指向の言語では、メソッド呼び出しは下記の形式で行います。

```
C_Obj.method1(arg1, arg2)
```

しかし C 言語は、基本的にオブジェクト指向の言語ではないため、上記のように書くのではなくオブジェクトを引数に付け加えた関数で表す必要があります。そこで RtORB では、クラスメソッドは、IDL コンパイラにより下記のような関数にマッピングされるようになっています。

```
クラス名_メソッド名
```

上記の２つの手続きを行えば、CORBA オペレーションは可能なのですが、より割り易くするために単純な例を考えてみます。

例として、下記のような interface が echo.idl というファイルに定義されているとします。

```
interface Echo {
  string echoString(in string mesg);
  string echoString2(in string mesg, out string res);
  string echoString3(inout string mesg);
};
```

RtORB では、この例の場合、リモート CORBA オブジェクトの参照は”Echo”という型になります。リモート CORBA オブジェクトの参照が変数 myEcho にセットされているとすると、echoString というのメソッド呼び出しは、下記のように記述します。

```
Echo_echoString(myEcho, "hello" , &env);
```

ここで env は、 CORBA_Environment という型の構造体であり、Exception などの情報が入ることになります。この関数は、echo-common.c に定義されており、IDL コンパイラにより自動生成され、その内容は、下記のようになります。

```
CORBA_string
Echo_echoString(CORBA_Echo _obj, const CORBA_char * mesg, CORBA_Environment *ev)
{
  CORBA_string _ORBIT_retval;
  void * _args[1];
  _args[0] = (void *)&mesg;
  invokeMethod (_obj, &Echo__imp.methods[0], (void **)&_ORBIT_retval, _args, ev );
  return _ORBIT_retval;
}
```

このようにクライアント側では、リモート CORBA へのオペレーションは、invokeMethod 関数を実行することで実現されています。

9.1. invokeMethod 関数

この関数は、リモート CORBA オブジェクトに対してメソッドの呼び出しを行います。この関数は、rtorb.c に記述されており、次のような処理を行います。

先ず、CORBA_ORB_find_object 関数にて、リモート CORBA オブジェクトが The_RootPOA に含まれていないかどうかをチェックします。もし、The_RootPOA のオブジェクトマップ（ハッシュ表）になければ、invokeMethod_via_GIOP 関数を呼び出し、IIOP 経由（TCP ソケットストリーム経由）でメソッドの実行のリクエストを行います。逆に、リモート CORBA オブジェクトが The_RootPOA に含まれていた場合には、実装の関数を検索し、その関数を実行します。すなわち、同一プロセス内の CORBA オブジェクトに対するメソッド呼び出しは、単なる関数呼び出しになりますので、とても高速に動作させることが可能です。

9.1.1. invokeMethod_via_GIOP 関数

この関数は、リモート CORBA オブジェクトが外部プロセスに存在している場合に呼ばれます。この関数は、rtorb.c に記述されており、次のような処理を行います。

1. リモート CORBA オブジェクトの参照情報(obj->connection)に基づき IIOP の通信用 TCP ソケットを生成します。

   ```
   h = make_client_connection(obj->connection);
   ```

2. obj->_url[0].location_flags をチェックして、以前呼び出しを行ったかどうかをチェックします。もし今回初めて呼び出しを行う場合には、confirmLocation 関数を実行してリモート CORBA オブジェクトが存在するかどうかを確認します。確認できたら、obj->_url[0].location_flags を 1 にセットします。
3. GIOP リクエスト、リプライメッセージを作成するためのバッファ arg_buf、req_buf、buf の領域を確保します。ここで、arg_buf は引数の符号化、req_buf は、GIOP リクエストメッセージの符号化（arg_buf を含む）、buf はリクエストの結果を格納するためのものです。
4. Marshal_Args 関数を実行し、CORBA オペレーションの引数を符号化します。

   ```
   len = Marshal_Args(arg_buf, args, method->in_argc, method->in_argv);
   ```

5. createRequest 関数により、符号化された引数を含めリモート CORBA オブジェクトへ送る GIOP リクエストメッセージを生成します。

   ```
   len = createRequest(req_buf, 1, obj->_url[0].object_key, obj->_url[0].object_key_len,method->name, strlen(method->name)+1, arg_buf, len, 2 );
   ```

6. GIOP_ConnectionHandler_send 関数で、リモート CORBA オブジェクトへ上で生成した GIOP リクエストメッセージを送信します。
7. receiveMessage 関数で、リモートメソッドの実行結果を受信します。RtORB の IIOP では、ブロッキング IO を使っていますので、receiveMessage 関数ではリモート CORBA オブジェクトからの返信があるまで待つことになります。戻り値が 0 よし小さければ SystemException が発生したことを意味するので、エラーメッセージの表示を行い、バッファ arg_buf、req_buf、buf を廃棄して終了します。
8. header.flags をチェックして、バイトオーダーを確認しリプライメッセージのサイズを取得します。
9. 次に、リプライメッセージを処理するために actionReply 関数を呼び出します。（buf から CORBA_Sequence_Octet 型の reply_body の抽出を行っています？）
10. env->_major をチェックし、正常終了であれば、deMarshal_Arguments 関数を実行します。この

関数により CORBA オペレーションの返り値等を復号化することができます。

11. CORBA_USER_EXCEPTION が発生していれば、メッセージを出力します。
12. 最後に、IIOP 通信に使用したバッファ等（reply_body、h、arg_buf、req_buf、buf）を廃棄して終了します。

9.1.1.1. make_client_connection 関数

この関数は、invokeMethod_via_GIOP 関数の中で呼び出され、IIOP の通信のための TCP ソケットストリームを生成します。この関数は、connection.c で定義されており、以下の処理を行います。
先ず、GIOP_Connection *conn を引数とします。この要素変数である conn->sock をチェックして 0 であった場合に、make_client_socket_port 関数を用いて IIOP 通信用の TCP socket を生成します。その後、GIOP_ConnectionHandler 構造体を生成して、それを返します。
この時、GIOP_ConnectionHandler 構造体を malloc 関数にて領域確保を実施しているので、受け取った側で開放作業を忘れないようにしなければいけません。
本来は、GIOP_ConnectionHandler を生成する必要はないと思いますので、次期バージョンでは改修予定です。

9.1.1.2. confirmLocation 関数

この関数は、リモート CORBA オブジェクトの場所を確認するものです。giop.c に定義されており、(GIOP_ConnectionHandler *h, CORBA_URL *ior)を引数としてとして持ち、以下の処理を実施します。

1. GIOP メッセージ用のバッファ buf を生成し、正常に生成されていれば、その内容を 0 でクリアします。
2. createLocateRequest 関数で ior の CORBA オブジェクトに対する LocateRequest を生成します。
3. GIOP_ConnectionHandler_send 関数で、LocateRequest を送信します。
4. buf を受信に使うためにメモリ内容を 0 でクリアします。
5. receiveMessage 関数で CORBA オブジェクトからの返信を待ちます。受信エラーがあればエラーメッセージを出力し−1 を返します。
6. new_CORBA_Sequence_Octet 関数で、GIOP メッセージ処理用の CORBA_Sequence_Octet を生成します。
7. newLocateReplyHeader 関数で LocateReply のヘッダー処理用メモリ領域 locate_reply_header の領域を確保します。
8. deMarshalLocateReply 関数で、LocateReply を復号化します。
9. locate_reply_header->locate_status を返り値 result にセットします。
10. delete_CORBA_Sequence_Octet 関数で GIOP メッセージ処理用の CORBA_Sequence_Octet を解放します。
11. RtORB_free 関数で、locate_reply_header と buf を解放します。
12. result を返り値として、終了します。

9.1.1.3. Marshal_Args 関数

この関数は、リモート呼び出しリクエストメッセージのためにオペレーションの引数を符号化（marshaling）

します。この関数は、rtorb.c に定義されており、引数として(octet *buf, void **argv, int i_args, CORBA_IArg *i_argv)をとります。
この関数では、CORBA_IArg 構造体の内容に応じて argv を marshal_by_typecode 関数を使って符号化し、buf に保存します。RtORB では、CORBA_IArg 構造体を下記のように定義しています。

```
typedef struct CORBA_IArg {
    CORBA_TypeCode tc;
    unsigned char   io;
    char *name;
}CORBA_IArg;
```

この構造体の内部変数である type には、CORBA_I_ARG_IN、CORBA_I_ARG_INOUT、CORBA_I_ARG_OUT のいずれかが入ります。CORBA オペレーションにおいて、クライアントからサーバーへ送信すべき引数は、CORBA_I_ARG_IN、CORBA_I_ARG_INOUT である引数のみです。

9.1.1.4. createRequest 関数

この関数は、CORBA オペレーションを呼び出すための GIOP リクエストメッセージを生成します。giop.c で定義されており、以下の処理を実施します。

1. newRequestHeader 関数で、GIOP_RequestHeader の領域を確保し初期化します。
2. GIOP_Create_MessageHeader 関数で、GIOP_MessageHeader を生成します。このとき、符号化用のバイトオーダーbyte_order と GIOP リクエストメッセージのマイナーバージョン version を設定しまう。RtORB では、Fragment メッセージを実装していませんので version=0 とする方がよいと思います。
3. version にしたがって、GIOP_RequestHeader の設定を行います。GIOP_RequestHeader は、version の値によって下記のような処理を行います。
   - version < 2 の時：(GIOP1.0 または GIOP1.1 の場合)
     - header->_1_0.request_id に next_request_id 関数で、request_id を設定します。
     - header->_1_0.response_expected に引数で与えられた response をセットします。RtORB では、oneway をサポートしませんので常に 1 が入ることを期待しています。
     - set_CORBA_Sequence_Octet 関数で、header->_1_0.object_key に引数で与えられている object_key をセットします。ここは、strdup 関数などでコピーの方がいいかもしれません。
     - set_CORBA_Sequence_Octet 関数で、header->_1_0.operation に引数で与えられている operation をセットします。ここも、strdup 関数などでコピーの方がいいかもしれません。
   - -version >= 2 の時：(GIOP1.2 または GIOP1.3 の場合)
     - header->_1_2.request_id に next_request_id 関数で、request_id を設定します。
     - header->_1_2.response_flangs に引数で与えられている response をセットします。RtORB では、oneway をサポートしませんので常に 1 が入ることを期待しています。
     - header->_1_2.target_type に GIOP_KeyAddr(=0)をセットします。
     - set_CORBA_Sequence_Octet 関数で、header->_1_2.target.object_key に引数で与えられている object_key をセットします。ここも、strdup 関数などでコピーの方がいいかもしれません。
     - set_CORBA_Sequence_Octet 関数で、header->_1_2.operation に引数で与えられている operation をセットします。ここも、strdup 関数などでコピーの方がいいかもしれません。

4. MarshalRequest 関数で header、args を符号化し、buf に格納します。返り値は、buf のサイズになりますので、その値を len にセットします。
5. MsgHeader->message_size に len - SIZEOF_GIOP_HEADER をセットします。これは、GIOP のメッセージボディのサイズになります。
6. memcpy 関数で、MsgHeader を buf にコピーします。
7. deleteRequestHeader 関数で header を破棄します。
8. RtORB_free 関数で MsgHeader を解放します。

9.1.1.5. GIOP_ConnectionHandler_send 関数

この関数は、リモート CORBA オブジェクトに対して GIOP メッセージを送信します。giop.c に記述されており、(GIOP_ConnectionHandler *h, char *buf, int32_t len)を引数として呼び出され、次の処理をします。
h->type == CONNECTION_TYPE_SOCKET の時には、writeBytes 関数を用いて、指定されたソケットストリーム（h->sock）に GIOP メッセージである buf を送信します。ここで、GIOP_ConnectionHandler の内部変数である type は、IIOP では必ず CONNECTION_TYPE_SOCKET になっているはずです。この処理は、以前共有メモリベースの通信を実装したときに使っていたものなのですが、現在では IIOP のみを正式サポートにしていますので、この条件分岐は必要なくなります。今後の仕様変更で修正されます。
また、writeBytes 関数は、指定されたソケットストリームに GIOP メッセージの書き込みを行っているが、この時、GIOP ヘッダー部分とメッセージボディの部分の２つに分けて処理しています。（本来は、必要ないかもしれませんが念のための処理です。）

9.1.1.6. receiveMessage 関数

この関数は、リモート CORBA オブジェクトからの返事（リプライ）を受信するためのものです。RtORB では、ブロッキング IO を使っていますので、read システムコールを呼び出すとリモートオブジェクトの処理の終了を待つことになります。この関数は、giop.c に記述されており、(GIOP_ConnectionHandler *h, GIOP_MessageHeader *header, octet *buf, int maxMsg)を引数として呼び出されます。
この関数の処理は、下記の通りです。

1. GIOP_ConnectionHandler_recv 関数で、GIOP ヘッダー部分を受信します。このヘッダーは buf に保存されます。この時受信エラーが発生すれば、－1 を返して終了します。
2. 次に、変数 header に buf の内容を memcpy 関数でコピーします。Header は、GIOP メッセージヘッダーを格納するための構造体です。
3. header->flags をチェックし、(header->flags & 0x01) == 0 であれば、header->message_size=ntohl(header->message_size)を実行して、header->message_size の内容を正しくします。この処理は、RtORB がビッグエンディアンであることが前提になっているので正確な処理ではありません。本来は、自分の符号化のバイトオーダーと異なっていれば、データバイト列を逆順にするという処理が必要です。次期バージョンでは修正します。
4. 次に、GIOP メッセージボディのサイズを msgLen にセットします。
```
msgLen = header->message_size;
```
5. msgLen > maxMsg であれば、バッファオーバーフローとなるので、エラーメッセージを出力し、－1 を返して終了します。
6. GIOP_ConnectionHandler_recv 関数で残りのメッセージを受信します。この時メッセージは、

(buf+SIZEOF_GIOP_HEADER)、すなわち変数 buf の先頭から SIZEOF_GIOP_HEADER の位置に保存します。GIOP_ConnectionHandler_recv 関数実行時ににエラーがあれば、エラーメッセージを出力し、−1 を返して終了します。

7. GIOP_ConnectionHandler_recv 関数が正常終了であれば、 header->message_size を返り値として終了します。

9.1.1.7. actionReply 関数

この関数は、GIOP メッセージの ReplyHeader 解釈して、第 1 段階の復号処理を行います。rtorb.c に定義されており、(octet *buf, CORBA_Sequence_Octet *body, CORBA_Class_Method *method, CORBA_Environment *env)を引数として呼び出されます。この関数の処理手順は下記の通りです。

1. memcpy 関数で、GIOP_MessageHeader header 構造体に返信メッセージ buf からコピーします。
2. newReplyHeader 関数で GIOP_ReplyHeader 用のデータ領域を確保し、reply_header にセットします。
3. deMarshalReply 関数で、buf から reply_header、body を抽出します。
4. header.version.minor < 2 の場合には、返信用ヘッダーは reply_header->_1_0 になりますので、その内部変数の reply_status を reply_status にセットします。

   ```
   reply_status = reply_header->_1_0.reply_status;
   ```

   それ以外の場合には、返信用ヘッダーは reply_header->_1_2 になりますので、その内部変数の reply_status を reply_status にセットします。

   ```
   reply_status = reply_header->_1_2.reply_status;
   ```

5. 次に、符号化時のバイトオーダーをあらわす header.flag をチェックし、(header.flags & 0x01) == 0 であれば、header->message_size=ntohl(header->message_size)として header->message_size の内容を変換する。それ以外であれば、 byte_order = 1 にセットする。この処理は、RtORB の実行 OS がビッグエンディアンであることが前提になっているので修正の必要があります。本来は、自分の符号化時のバイトオーダーと異なっていれば、データバイト列を逆順にするという処理が必要になります。
6. env->_major に reply_status をセットし、reply_status の値に応じて下記の処理をします。
   - reply_status が GIOP_NO_EXCEPTION の場合：正常終了なので何もしません。
   - reply_status が GIOP_USER_EXCEPTION の場合：あとで処理するので何もしません。
   - reply_status が GIOP_SYSTEM_EXCEPTION の場合：エラーメッセージを出力します。
   - reply_status が GIOP_LOCATION_FORWARD の場合：GIOP メッセージと"Location forward¥n"を出力します。
   - reply_status が GIOP_LOCATION_FORWARD_PERM の場合：GIOP メッセージと"Location forward¥n"を出力します。
   - reply_status が GIOP_NEEDS_ADDRESSING_MODE とその他の場合:GIOP メッセージを出力します。
7. deleteReplyHeader 関数で reply_header を破棄します。

9.1.1.8. deMarshal_Arguments 関数

この関数は、リモート CORBA オペレーションの返り値を deMarshal_Result 関数で復号化し、必要があれ

ば、引数の復号化を実行します。第２段階の復号処理になります。この関数は、giop-marshal.c で定義されており、 (void **retval, void **args, octet *buf, CORBA_Class_Method *method, int order)を引数として呼び出されます。この関数内での処理は、下記のように実施されます。

1. retval が NULL でない場合には、deMarshal_Result 関数でリモートメソッド呼び出しの結果を復号化します。
2. リモートメソッドの引数で CORBA_I_ARG_OUT または、CORBA_I_ARG_INOUT のものがあれば、返り値に続いて GIOP メッセージ内で符号化されていますので、それを demarshal_by_typecode 関数で復号化します。

9.1.1.8.1. deMarshal_Result 関数

この関数は、リモート CORBA オペレーションの返り値を、その型に応じて demarshal_by_typecode 関数を用いて複合化します。この関数は、giop-marshal.c で定義されており、(void **retval, CORBA_TypeCode tc, octet *buf, int *size, int order)を引数として呼び出されます。この関数の処理手順は下記の通りです。

1. SKIP_ALIAS(tc)マクロ関数で tc->kind が tk_alias であれば、そのメンバーの TypeCode に置き換えます。
2. tc->kind をチェックして、その値に応じて次の処理をします。
   - tc->kind が tk_union の場合：CORBA_TypeCode_is_fixed_size(tc)が真であれば、demarshal_by_typecode 関数で復号化します。この部分にはバグがあります。現時点で union は原則使われていないので問題が出ていませんが、書きなおす必要があります。
   - tc->kind が tk_struct、tk_sequence、tk_any の場合：RtORB_typecode_alloc 関数で結果を入れるメモリ領域 val を確保して、demarshal_by_typecode 関数で復号化します。そして、*retval = val で retval をセットします。
   - その他の場合：demarshal_by_typecode 関数で結果を復号化します。

この関数は、invokeServant 関数の修正に伴い、再度変更する必要があります。精査してみないとわかりませんが、もしかするとこの関数自体が必要ないかもしれません。その時は、invokeServant の修正、deMarshal_Arguments 関数の修正も行う必要があります。

9.2. リモート CORBA オブジェクトのメソッドの実行処理

次に、リモート CORBA オブジェクトのメッソド呼び出し実行時に、サーバー側の実行処理について述べます。「8.ORB の実行と IIOP の低レベルの処理」のところでも述べましたが、サーバー側では CORBA_ORB_run 関数を実行することで、IIOP 用の socket 記述子からのデータ送受信処理のための無限ループが実行されています。低レベルのイベント処理に関しては、そちらを参照して下さい。

リモート CORBA オブジェクトのメッソド呼び出し実行が起こった場合には、基本的に下記のような処理が進みます。

1. まず、リモート CORBA オブジェクトのメッソド呼び出しがクライアント側で起こった場合には、サーバー側に接続要求が発生します。それに応じて、IIOP 通信用の socket 記述子が accept_connection 関数で生成され、通信用 socket が確立します。
2. 次に、クライアントからの GIOP メッセージは、PortableServer_enqueue_request 関数により

サーバオブジェジェクトの所属する poa の request queue にリストとして入れられます。

3. すべてのクライアントからの GIOP メッセージの受信終了後、PortableServer_execute_request 関数により、request queue にリストされた CORBA リクエストメッセージに対して、リモート CORBA オブジェクトのオペレーションが順次実行されていきます。

9.2.1. PortableServer_enqueue_request 関数

この関数は、GIOP のリクエストメッセージを poa の request queue に追加していきます。RtORB では、poa は、The_RootPOA のみとしていきますので、今後 API を変更する可能性があります。
また、この関数は、PortableServer_POA_activate 関数内で呼ばれており、下記のように socket の enqueue コマンドとして登録されています。

```
set_SockProfile(poa->_server->sock, SOCK_SERVER, NULL, NULL, PortableServer_enqueue_request);
```

この関数での処理は、下記のようになっています。

1. GIOP_ConnectionHandler_get_arg 関数で POA を取得します。(The_RootPOA のみを使用する場合には不必要になります)
2. MutexLock マクロで Mutex をロックします。このマクロは、Thread 対応版のみ有効になるように設定されています。
3. GIOP_enqueue_request 関数で GIOP リクエストメッセージをメッセージキューに入れます。
   poa->requests = (PtrList *)GIOP_enqueue_request(h, poa->requests);
4. MuteUnLock マクロで手順 2 でロックした Mutex を解放します。このマクロも Thread 対応版のみ有効になるように設定されています。
5. poa->requests == -1（メッセージ読み込み失敗）ならば、-1 を返し、そうでなければ 1 を返します。

9.2.1.1. GIOP_enqueue_request 関数

この関数は、PortableServer_enqueue_request 関数から呼ばれ、GIOP のリクエストメッセージを poa の requst queue（引数で与えられています）に追加します。この関数での処理は下記の手順で実行されます。

1. GIOP リクエストメッセージを読み込むためのバッファ(buf)の領域を確保します。
2. receiveMessage 関数を呼び出し、GIOP リクエストメッセージを読み込みます。
3. Request 構造体のバッファを確保し、ConnectionHandler 格納用の領域確保、GIOP リクエストメッセージの領域確保を行います。
4. Request に ConnectionHandler と GIOP リクエストメッセージをコピーします。
5. poa の request queue（リスト）に Request を追加します。
6. GIOP リクエストメッセージを格納していた buf を破棄します。
7. poa の request queue を返して終了します。

9.2.2. PortableServer_execute_request 関数

この関数は、GIOP リクエストメッセージを処理します。この関数が CORBA のリモートメソッド呼び出しにおけるメインの関数になると言えます。詳細は、「8.8. PortableServer_execute_request 関数」を参照

して下さい。簡単に言うと、この関数から GIOP_excute_request 関数が呼ばれます。この GIOP_excute_request 関数では、受信した GIOP メッセージの種類に応じて、GIOP_Request_perform 関数、GIOP_Reply_perform 関数、GIOP_CancelRequest_perform 関数、GIOP_LocationRequest_perform 関数、GIOP_LocationReply_perform 関数を実行します。

9.2.2.1. GIOP_Request_perform 関数

この関数は、受信した GIOP メッセージが Request であった場合に GIOP_excute_request 関数から呼び出され、リモート CORBA オペレーションのリクエストの処理を行います。この関数は、giop.c で定義されており、(GIOP_ConnectionHandler *h, octet *buf, PortableServer_POA poa, GIOP_MessageHeader *header, CORBA_Sequence_Octet *body, CORBA_Environment *env)を引数として呼び出されます。この関数の処理は、下記のように実行されます。

1. まず、GIOP のヘッダーから符号化のバイトオーダーを抽出します。(header->flags & 0x01) != 0 であれば、buf のバイトオーダーをビッグエンディアンにセットします。
2. newRequestHeader 関数で、GIOP_RequestHeader の格納領域 request_header を確保します。
3. deMarshalRequest 関数で、GIOP メッセージヘッダーを復号化します。
4. invokeServant 関数を呼び出し、リモート CORBA オペレーションを実行します。結果は、GIOP_ReplyBody *reply_body にセットされます。
5. reply_Message 関数で、処理結果をクライアントに送ります。
6. deleteReplyBody 関数で、reply_body を破棄します。
7. deleteRequestHeader 関数で、request_header を破棄します。

9.2.2.1.1. invokeServant 関数

この関数は、リモート CORBA オペレーションの実体です。この関数は、giop.c で定義されており、(PortableServer_POA poa, GIOP_RequestHeader *header, CORBA_Environment *env, octet *arg_buf, int version, int order)を引数として呼び出されます。この関数の処理の流れは次のようになっています。

1. まず、RtORB_alloc 関数で GIOP_ReplyBody 構造体を格納する領域 reply を確保します。
2. reply を次のように初期化します。

```
reply->status = GIOP_NO_EXCEPTION ;
reply->body = NULL;     reply->body_size = 0;
```

3. 引数の poa が NULL でなければ、tbl = poa->object_map としてサーバント格納テーブルをセットします。poa が NULL であれば、デフォルト ORB の管理下にある格納テーブル _ORB_->_object_table を tbl にセットします。(次のバージョンでは、poa は The_RootPOA のみになるので、この処理は、tbl = The_RootPOA->object_map; に書きなおされます。)
4. 送信された GIOP メッセージのマイナーバージョン version をチェックし、obj_key と function を次のようにセットします。

■ version > 1 の場合：(GIOP1.2 以上の場合)

```
obj_key = (char *)header->_1_2.target.object_key._buffer;
function = (char *)header->_1_2.operation._buffer;
```

- 上記以外の場合：（GIOP1.0 または GIOP1.1 の場合）

```
obj_key = (char *)header->_1_0.object_key._buffer;
function = (char *)header->_1_0.operation._buffer;
```

5. obj_key が NULL の時はネーミングサービスとみなしますので、 obj_key に"NameService"という文字列をセットします。
6. getValue 関数により、obj_key に対応する RtORB_POA_Object をオブジェクト管理テーブルから取得します。

```
poa_obj = (RtORB_POA_Object *)getValue(tbl, obj_key);
```

7. poa_obj が NULL､すなわち obj_key に対応する RtORB_POA_Object が見つからない場合には、reply に GIOP_SYSTEM_EXCEPTION をセットし、"Unknown Object" を reply->exception.exception_id._buffer にセットし、reply を返して終了します。
8. poa_obj を取得できれていれば、poa_obj->_private は、PortableServer_ClassInfo ですので、これをキャスティングして info にセットします。
9. また、poa_obj->servant は、PortableServer_ServantBase にキャストして、PortableServer_ServantBase *sb にセットしておきます。
10. info->impl_finder には、実装関数を検索するための関数ポインタが代入されているはずですので、下記を実行して、実装関数の関数ポインタを得ます。

```
call_impl_func = (impl_func_type)(*info->impl_finder)(&sb->_private, function, &m_data, &impl_method );
```

11. call_impl_func が NULL の場合は、下記の処理を実行します。
    - function が_is_a の場合には、sb->vepv[0]->is_a を調べて、ここに関数ポインタが設定されているのであれば、引数を復号化して、(sb->vepv[0]->is_a)(&sb->_private, id , env);を実行し実行結果を reply にセットし、reply を返して終了します。
    - function が_non_existent の場合には、sb->vepv[0]->non_existent を調べて、ここに関数ポインタが設定されているのであれば、(sb->vepv[0]->non_existent)(&sb->_private, env);を実行し実行結果を reply にセットし、reply を返して終了します。
    - 上記以外であれば、リモートメソッドは実装されていないので reply に GIOP_SYSTEM_EXCEPTION をセットし、reply を返して終了します。
12. 次に、deMarshal_Arg 関数で、実装関数のに渡す引数を復号化し argv にセットします。
13. 実行結果を入れる result に Result_alloc で確保した領域のポインタをセットします。
14. (*call_impl_func)関数を実行し、実装関数を呼び出します。

```
(*call_impl_func)(sb, result, m_data, argv, env, impl_method);
```

15. env->_major をチェックすることで、実装関数が正常終了したかどうかわかるため、次の処理を行います。

- env->_major が CORBA_NO_EXCEPTION の場合：Marshal_Reply_Arguments 関数で結果を符号化します。

```
Marshal_Reply_Arguments(reply, (void**)result, argv, m_data);
```

- env->_major が CORBA_USER_EXCEPTION の場合：
    1. reply_buf を RtORB_alloc 関数で確保し、初期化します。
    2. 次に、marshalString 関数で env->_repo_id を符号化します。
    3. marshal_by_typecode 関数で env->_params を符号化します。

4. reply->body_size に符号化した情報の長さをセットします
5. reply->body を RtORB_alloc 関数で確保します。
6. memcpy 関数で reply->body に reply_buf をコピーします
7. reply->status に GIOP_USER_EXCEPTION をセットし、RtORB_free 関数で reply_buf を破棄します。

- env->_major が CORBA_SYSTEM_EXCEPTION の場合：reply に GIOP_SYSTEM_EXCEPTION をセットし、その他の必要な情報をセットします。
- env->_major が上記以外の場合：これはありえないはずですので、何もしません。

16. RtORB_Arguments_free 関数で argv のポインタの内容を破棄します。
17. RtORB_Result__free 関数で result を破棄します。
18. RtORB_free 関数で argv を破棄します。

9.2.2.2. GIOP_Reply_perform 関数

この関数は、リモート CORBA オペレーションの実行結果（GIOP リクエスメッセージの返事）を処理します。この関数は、giop.c に定義されており、(GIOP_ConnectionHandler *h, octet *buf, PortableServer_POA poa, GIOP_MessageHeader *header, CORBA_Sequence_Octet *body, CORBA_Environment *env)を引数として呼び出されます。この関数も GIOP_excute_request 関数から呼び出されますが、BiderectionalIIOP をサポートしない場合には、必要ではありません。RtORB の現在の実装は、呼び出されることはないと思います。OpenRTM-aist では使わないので削除予定です。

1. newReplyHeader 関数で、GIOP_ReplyHeader の格納領域 reply_header を確保します。
2. deMarshalReply 関数で、GIOP のメッセージから GIOP_ReplyHeader と ReplyBody を復号化します。
3. header->version.minor < 2 の場合：reply_status = reply_header->_1_0.reply_status とし、それ以外であれば、reply_status = reply_header->_1_2.reply_status として、実行結果の状態をセットします。
4. reply_status の値に応じて下記の処理を実行します。
   - reply_status が GIOP_NO_EXCEPTION の場合：正常終了なので何もしません。
   - reply_status が GIOP_USER_EXCEPTION の場合： "User Exception occured!!!¥n"を出力します。
   - reply_status が GIOP_SYSTEM_EXCEPTION の場合："System Exception occured in GIOP_Reply_perform¥n"を出力します。
   - reply_status が GIOP_LOCATION_FORWARD の場合：GIOP メッセージと"Location forward¥n"を出力します。
   - reply_status が GIOP_LOCATION_FORWARD_PERM の場合：GIOP メッセージと"Location forward¥n"を出力します。
   - reply_status が GIOP_NEEDS_ADDRESSING_MODE とその他の場合:GIOP メッセージを出力します。
5. deleteReplyHeader 関数で reply_header を破棄します。

9.2.2.3. GIOP_CancelRequest_perform 関数

この関数は、CancelRequest メッセージが送られてきた場合に実行されます。この関数は、giop.c で定義されており、(GIOP_ConnectionHandler *h, octet *buf, PortableServer_POA poa, GIOP_MessageHeader *header, CORBA_Sequence_Octet *body, CORBA_Environment *env) を引数として呼び出されます。このメッセージを受信すると、対応する request_id のメッセージのキャンセル処理をします。GIOP_excute_request 関数から呼ばれる場合には、既に呼び出された CORBA オペレーションの中断処理または、Fragment メッセージの送信中止の時に使われます。しかし、RtORB では、Thread 対応でない場合には、CORBA オペレーション時には、その終了までブロックされますので、実質的にこのメッセージの受け取り処理はできません。クライアントからリクエストが Fragment メッセージの場合には、意味がありますが、前述したように RtORB では Fragment メッセージをサポートしていませんので、意味がありません。そのため、将来的には削除する予定です。これらを踏まえて、現在実装されているこの関数の処理を見ていきます。

1. まず、newCancelRequestHeader 関数で、GIOP_CancelRequestHeader の格納領域 cancel_request_header を確保します。
2. deMarshalCancelRequest 関数で、cancel_request_header を復号化します。
3. "Call Cancel %d¥n"のメッセージを stderr へ出力します。Fragment メッセージをサポートする場合には、ここで request queue にある該当メッセージの削除を行い、Thread でオペレーションを実行中の場合には、Thread を強制終了させるコードを挿入することになります。
4. RtORB_free 関数で、cancel_request_header を破棄します。

9.2.2.4. GIOP_LocationRequest_perform 関数

この関数は、リモート CORBA オペレーションの実体の一つであり、GIOP_excute_request 関数から呼ばれます。リモート CORBA オブジェクトが正しいオブジェクトかどうかの問い合わせに対する処理を実施します。この関数は、giop.c で定義されており、(GIOP_ConnectionHandler *h, octet *buf, PortableServer_POA poa, GIOP_MessageHeader *header, CORBA_Sequence_Octet *body, CORBA_Environment *env) を引数として呼び出されます。この関数の処理の流れは次のようになっています。

1. newLocateRequestHeader 関数で、GIOP_LocateRequestHeader 構造体のの格納領域を確保して変数 locate_request_header にセットります。
2. deMarshalLocateRequest 関数で、locate_request_header を復号化します。
3. reply_locateMessage 関数を呼び出して、GIOP リクエストの処理を実行します。
4. deleteLocateRequestHeader 関数で、locate_request_header を破棄します。

9.2.2.4.1. reply_locateMessage 関数

この関数は、GIOP_LocationRequest_perform 関数の実体になります。giop.c で定義されており、(PortableServer_POA poa, GIOP_ConnectionHandler *h, GIOP_LocateRequestHeader *locate_request_header, int version)を引数として呼び出されます。この関数での処理は次のようになっています。

1. RtORB_alloc 関数で、返信用の buf を確保し、初期化します（0 でクリアします)。
2. version を確認して、header.request_id、object_key を次のようにセットします。
   - version < 2 の場合：(GIOP1.0 または GIOP1.1 の場合)

```
header.request_id = locate_request_header->_1_0.request_id;
object_key = (char *)locate_request_header->_1_0.object_key._buffer;
```

   - 上記以外の場合：(GIOP1.2 以上の場合)

```
header.request_id = locate_request_header->_1_2.request_id;
object_key = (char *)locate_request_header->_1_2.target.object_key._buffer;
```

3. find_object 関数で、object_key のオブジェクトのエントリを確認します。
4. MarshalLocateReply 関数で、find_object 関数の結果を符号化します。
5. GIOP_ConnectionHandler_send 関数で結果を送信します。
6. RtORB_free 関数で buf を破棄します。

9.2.2.5. GIOP_LocationReply_perform 関数

この関数は、LocationRequest の GIOP メッセージを送信後、その返事を処理するためのものです。giop.c で定義されており、(GIOP_ConnectionHandler *h, octet *buf, PortableServer_POA poa, GIOP_MessageHeader *header, CORBA_Sequence_Octet *body, CORBA_Environment *env) を引数として呼び出されます。この関数も GIOP_excute_request 関数から呼び出されますが、その時は、BidirectionalIIOP をサポートしているときのみ有効です。本来は、クライアント側の処理になります。
現在の実装では、OpenRTM-aist を動作させる場合には、この関数は呼び出されておりませんので、将来的には削除擦る可能性があります。以上を踏まえて、この関数での処理を見ていきます。

1. 先ず、newLocateReplyHeader 関数で GIOP_LocateReplyHeader の格納領域を確保し変数 locate_reply_header にそのポインタをセットします。
2. deMarshalLocateReply 関数で、locate_reply_header を復号化します。
3. locate_reply_header->locate_status の値の応じて下記の処理を行います。

- GIOP_UNKNOWN_OBJECT の場合："Unknown Object¥n"と表示します。
- GIOP_OBJECT_HERE の場合：何もしません。
- GIOP_OBJECT_FORWARD、GIOP_OBJECT_FORWARD_PERM の場合："Object forward¥n"と表示して、GIOP メッセージを表示します。本来は、ここで再送処理または、場所の確認をすべきだと思います。
- GIOP_LOC_SYSTEM_EXCEPTION の場合："Locate System Exception¥n"と表示して、GIOP メッセージを表示します。
- GIOP_LOC_NEEDS_ADDRESSING_MODE の場合："Not Implemented¥n"と表示して、GIOP メッセージを表示します。
- 上記以外の場合：locate_reply_header->locate_status を表示します。

4. RtORB_free 関数で、locate_reply_header を破棄します。

10. データの符号化と復号化

データの符号化と復号化は、何故必要なのでしょうか？
それは、２つのプロセス間で通信する場合に、どんなデータを送るかを予め決めたおかないと正しいデータの送受信が出来ないので、決まったルールでどんなコンピュータでも扱える形にしましょうということです。これは、全く同じコンピュータ間で、相手がどのような形のデータ構造を要求しているのかが分かっていれば、あまり問題にならないのですが、CORBA のような分散オブジェクトを実装する場合には、予めデータ構造を決めておくことは、その利用を制限することになりますのでよくありません。そのため、通常のプログラミングで扱うデータ構造を効率よく表現する必要がありますが、一般に、いろいろな OS や CPU 上で動くプログラムには、内部表現が統一されていません。

例えば、mips 系の CPU では、ビッグエンディアンで数字を表現しますが、Intel x86 系の CPU では、リトルエンディアンで数字を表現しています。このエンディアン（バイトオーダーともいう）というのは、多バイトのデータをメモリ上に配置する方式ことで、4 バイトで表現される 100 という整数があった場合に、ビッグエンディアンの CPU では、

```
0x00, 0x00, 0x00, 0x64
```

となっているのに対し、リトルエンディアンでは、

```
0x64, 0x00, 0x00, 0x00
```

と表現されています。このような違いを吸収し、どのような OS 上の CORBA オブジェクトとも通信できるように、すべてのデータをバイト列に変換して送るということをしています。（エンディアンについては、Wikipedia を見てください。詳しく書いてあります。）

このために必要なのが、符号化処理(marshaling)と復号化処理(unmarshaling)です。
この符号化に関する定義はすでに、「CORBA の概要」で述べましたので、そちらを参照して下さい。

以下では、RtORB で実装している marshaling と unmarshaling について述べていきます。

10.1. TypeCode

符号化と復号化の処理の説明の前に、CORBA で扱うデータ型について述べます。CORBA で扱うデータ型をどのように表現し、他の CORBA オブジェクトと共通認識を持つために、TypeCode と呼ばれるものが定義されています。CORBA でのリモート呼び出しで、データ交換を行うときに、そのデータ表現がどうなっているかを表したものが TypCode です。TypeCode 型については、「CORBA の概要」でも述べていますので、RtORB での構造については省略します。

TypeCode には、RepositoryID と呼ばれる文字列が定義されており、この文字列によって一意に決定されます。例えば、CORBA ネーミングサービスの CosNaming における Istring という型は、下記のように定義されております。

```
module CosNaming {
   typedef string Istring;
};
```

この RepositoryID は、

```
IDL::omg.org/CosNaming/Istring:1.0
```

となっており、データ型の種類を示す"kind" は、"tk_alias"(=21)になっています。また、CORBA の基本データ型の long の場合には、RepositoryID が

IDL::omg.org/CORBA/long:1.0

となっています。もちろん、このデータ型のの"kind"は、"tk_long"(= 3)となっています。
この"kind"と TypeCode の中の定義（構造体や配列のように要素ある場合は、そのメンバーの TypeCode が再帰的に、TypeCode に定義されています）によって符号化と復号化を行います。

10.2. RtORB に実装された符号化処理(marshaling)

RtORB では、データの符号化処理は、marshal_by_typecode 関数で行われます。この関数は、TypeCode を引数として符号化処理を行いますので、構造体や配列などは、再帰呼び出しを行うことになります。
そのため RtORB を安全なコードで書く場合には、再帰呼び出しの部分を修正しなければいけないかもしれません。

marshal_by_typecode 関数は、giop-marshal.c に記述されています。
この関数は、(octet *buf, void *argv, CORBA_TypeCode tc, int *current)を引数として呼び出され、次のような処理を行っています。
buf は、符号化された結果を入れるバッファ、argv は、符号化するデータへのポインタ、tc は、符号化するデータの TypeCode、current は、符号化したデータを buf に入れるときのオフセットになります。

1. まず、tc を調べて NULL であれば、0 を返し終了します。
2. CORBA_TypeCode_add_dynamic を実行して tc を dynamic_tcs という大域変数で指定されているハッシュ表に登録しています。この処理は、動的な TypeCode をサポートするための実験的なコードですので、現在は必要ありません。
3. 次に、tc が tk_alias であれば、スキップして実体の TypeCode に変換します。これは、SKIP_ALISA というマクロで行っており、このマクロは、corba.h に記載されています。
4. 次に tc->kind にしたがって下記のようにデータの符号化を行っていきます。

- **tk_null, tk_void の場合：**この場合には、何も符号化しませんので、*current を返して終了します。
- **tk_char, tk_octet, tk_boolean の場合：**1 バイトデータですので、argv を char*にキャストし、marshalOctet 関数で buf に内容をコピーして current を 1 つ進めます。
- **tk_short の場合：**argv を short*にキャストし、marshalShort 関数で buf にコピーして、current を 2 つ進めます。
- **tk_ushort の場合：**argv を unsigned short*にキャストし、marshalShort 関数で buf にコピーして、current を 2 つ進めます。
- **tk_long の場合：**argv を int32_t *にキャストし、marshalLong 関数で buf にコピーして、current を 4 つ進めます。
- **tk_ulong の場合：**argv を uint32_t*にキャストし、marshalLong 関数で buf にコピーして、current を 4 つ進めます。
- **tk_float の場合：**argv を float*にキャストし、marshalFloat 関数で buf にコピーして、current を 4 つ進めます。
- **tk_double の場合：**argv を double*にキャストし、marshalDouble 関数で buf にコピーして、current を 8 つ進めます。
- **tk_string の場合：**argv を char**にキャストし、marshalString 関数で buf にコピーして、current を 4+文字列の長さ分進めます。

- tk_sequence の場合:
  - size_of_typecode(tc->member_type[0], F_DEMARSHAL)を行い sequence の要素のデータの長さを skip にセットします。この時 size_of_typecode 関数には、F_DEMARSHAL を 2 番目の引数で与えますが、これは、コンピュータ内部の Native のデータ長をとるという指定です。また cpos=0 にセットしておきます。
  - argv を CORBA_SequenceBase* にキャストし、sb という変数にセットします。CORBA_SequenceBase 型は、RtORB 内ですべての Sequence 型が共通部分を表す型です。
  - marshalLong 関数で符号化するデータの配列の長さ（sb->_length）を符号化します。
  - align_of_typecode 関数で、Sequence の要素の型の Alignment を求めて align_base にセットします。この関数も F_DEMARSHAL を 2 番目の引数で与えますが、これは、コンピュータ内部の Native のデータでの Alignment を求めています。
  - 次に、Sequence の長さの分だけ、以下の処理を繰返します。
    - align_base のアライメントにあうように cpos を進めます。
    - 符号化するためのデータの先頭ポインタを_buffer にセットします。

      ```
      _buffer =(char *)sb->_buffer + cpos;
      ```
    - marshal_by_typecode 関数を再帰呼び出しで、この_buffer を要素の TypeCode で符号化します。
    - cpos を skip だけ進めます。
- tk_exception, tk_struct の場合:
  - 符号化してデータ格納のためのオフセットを更新します。ここで、align_of_typecode 関数の第 2 引数で F_MARSHAL が与えられていますが、これによって CORBA のデータ構造におけるアライメントを得ることができます。この時、cpos=0 にセットしておきます。

    ```
    Address_Alignment(current, align_of_typecode(tc,F_MARSHAL));
    ```
  - strcut のメンバーの数だけ以下の処理を行います。
    - i 番目のメンバーの TypeCode を ctc にセットします。

      ```
      ctc = tc->member_type[i];
      ```
    - cpos を ctc のアライメントに合うように進めます。この時のアライメントは、コンピュータ内部の Native のデータアライメントになります。

      ```
      Address_Alignment(&cpos, align_of_typecode(ctc,F_DEMARSHAL));
      ```
    - 符号化するデータの先頭ポインタを_buffer にセットします。

      ```
      _buffer = (void **)((char *)argv + cpos);
      ```
    - marshal_by_typecode 関数を再帰呼び出しし、ctc の TypCode で、_buffer を先頭ポインタにもつデータを符号化します。
    - 符号化したデータ長分 cpos を進めます。

      ```
      cpos = cpos + size_of_typecode(ctc, F_DEMARSHAL);
      ```
- tk_union の場合:
  - 符号化してデータ格納のためのオフセットを更新します。これは、struct の時と同様の処理です。この時、というふうに、args と discrim に符号化を行うデータの先頭ポインタをセットします。

    ```
    Address_Alignment(current, align_of_typecode(tc,F_MARSHAL));
    ```

```
void **args = (void **)argv;
void **discrim = (void **)argv;
```

- marshal_by_typecode 関数を再帰呼出し、tc->discriminator の TypeCode で args を符号化します。
- idx に符号化するデータが、union のどのデータであるかのインデックスを idx にセットします。

```
idx = get_union_switch(tc->discriminator, discrim);
```

- tc->discriminator で表されている union のデータインデックスのデータ長を n にセットします。

```
n = size_of_typecode(tc->discriminator, F_DEMARSHAL);
```

- Address_Alignment 関数で、汎用ポインタのサイズをアライメントとして n を調整します。これは、RtORB では、union を構造体として表現しているためです。（もしかしたら、このルーチンは間違っている可能性がありますが、OpenRTM-aist を使用する上で、union の部分は使っていませんので影響ありません）
- n だけ符号化するデータの先頭ポインタを進めます。

```
args = (void **)((char *)args + n);
```

- idx が tc->default_index でなかった場合に次の処理をします。（ここでは、idx==tc->default_index 時の処理がないので、正常動作しない可能性が大きいです。後日書きなおしますが、多分この部分を削除でよさそうです。）
- 次に、tc->member_count 分だけ以下の処理を繰返します。
  - i 番目のメンバーの TypeCode を ctc にセットします。

    ```
    ctc = tc->member_type[i];
    ```

  - もし、tc->member_label[i] == idx　であった場合に、marshal_by_typecode 関数を再帰呼び出しをして、データを符号化します。
  - i 番目のコンピュータ内部のデータ長をセットします。

    ```
    n = size_of_typecode(ctc, F_DEMARSHAL);
    ```

  - n を汎用ポインタの長さのアライメントに合うように進めます。

    ```
    Address_Alignment(&n, sizeof(void *));
    ```

  - n だけ符号化するデータの先頭ポインタを進めます。

    ```
    args = (void **)((char *)args + n);
    ```

- tk_objref の場合：
  - 符号化するデータを CORBA_Object 型にキャストします。

    ```
    CORBA_Object obj = *(CORBA_Object *)argv;
    ```

  - 4 のアライメントを実行します。

    ```
    Address_Alignment(current, 4);
    ```

  - obj が NULL である場合には、下記の処理をします。
    - marshalLong 関数で 1 を buf にコピーします。
    - marshalOctet 関数で'¥0'を buf にコピーします。
    - marshalLong 関数で 0 をコピーします。（ここの処理がちょっと怪しい）
  - obj が NULL 出ない場合には、次の処理を行います。

- ◆ obj->_ior_string が NULL のときは、CORBA_Object__to_string 関数で obj->_ior_string をセットします。
- ◆ ior にオブジェクト参照をセットします。

```
ior = (char *)obj->_ior_string
```

- ◆ ior は、文字列になっているので、これを String2Octet 関数でバイト列に変換します。

```
ior_byte = (char *)String2Octet(ior + 12);
```

- ■ バイト列に変換した ior_byte の長さを len = strlen(ior+12) / 2 で len にセットします。
  - ◆ memcpy 関数で ior_byte をその長さ分 buf にコピーします。
  - ◆ RtORB_free 関数で ior_byte を破棄します。
  - ◆ *current を len だけ進めます。

```
*current += len;
```

- ■ **tk_any の場合:**
  - ■ 符号化するデータを CORBA_any*にキャストします。

```
CORBA_any *any = (CORBA_any *)argv;
```

  - ■ CORBA_any の実 TypeCode を tc_にセットします。

```
CORBA_TypeCode tc_ = any->_type;
```

  - ■ tc_ が NULL のときは、tc_を tk_null の TypeCode にします。
  - ■ アライメント調整をします。

```
Address_Alignment(current, 4);
```

  - ■ marshal_typecode 関数で TypeCode を符号化する。
  - ■ CORBA_any_get_encoded 関数で CORAB_any のエンコードされたデータを変数 v にセットします。これは、CORBA_any の実装では、strcut や sequence の複合型であった場合には、符号化された形でデータを保持していることを前提としているためです。

```
char *v = CORBA_any_get_encoded(any, &len);
```

  - ■ v が NULL でなければ、すでに符号化されているはずなので、memcpy 関数で buf にコピーします。
  - ■ v が NULL の場合には、string を含む基本データ型であるので、CORBA_any_get_value 関数でデータの先頭ポインタを args にセットし、marshal_by_typecode 関数の再帰呼び出しでデータを符号化します。
- ■ **上記以外の場合：** RtORB ではサポートしていないので、エラーメッセージを出します。

10.3. RtORB に実装された復号化処理(unmarshaling)

RtORB では、データの復号化処理は、demarshal_by_typecode 関数で行われます。これも符号化の処理と同じように TypeCode を引数として処理を行いますので、再帰呼び出しを行いつつ処理を進めることになります。

基本的には、marshal_by_typecode 関数と符号化されたバイトオーダーを考慮しながらの逆処理を行うことになります。