2024 機械情報夏学期 ソフトウェア第二

担当：岡田 慧 (k-okada@jsk.t.u-tokyo.ac.jp)

12. メッセージキュー・RPC・分散オブジェクト通信

1　分散オブジェクト通信

分散オブジェクト通信とは，ネットワーク上の異なるコンピュータ上に置かれたオブジェクトに対して，メンバ関数などの機能を呼び出し，その結果を得るようにする技術である．

実際には,「機能」自体を提供するサーバ計算機上のオブジェクトと，これを利用したいクライアント計算機上のオブジェクトの間で通信を行う．クライアント計算機側のオブジェクトはサーバオブジェクトの代理 (Proxy) オブジェクトと呼ばれ，ユーザは代理オブジェクトに対してメンバ関数を呼び出すだけでよく，分散オブジェクト通信のシステム（またはライブラリ）が，クライアントからサーバにメッセージを送り，また，返り値をメッセージとして受け取る．

これにより，異なる計算機上にあるオブジェクトをあたかも同一計算機上にあるかのようにプログラムを書くことができるオブジェクトの位置透過性だけでなく，異なるプログラミング言語で実装されたオブジェクトを呼び出すことを可能にするオブジェクトの実装言語独立性を実現できる．

1.1　ORB (Object Request Broaker)

ORB（分散オブジェクト間通信機構）はクライアントオブジェクトとサーバーオブジェクトとの間のメッセージ通信を仲介するためのミドルウェアを指す言葉であり，その重要な機能はシリアライズ/デシリアライズである．これはプロセス内のデータ構造を共通形式であるバイト列や XML フォーマットに変換し転送し，転送先ではその共通形式であるバイト列をプロセスのデータ構造に復元する機構であり，変換することをシリアライズ (直列化,serialize)，または Marchalling(マーシャリング）とよび，復元することをデシリアライズ (直列化復元,deserialize)，または Unmarchalling(アンマーシャリング）と呼ぶ[1][2]．ORB を介することで，オブジェクトの位置透過性と，実装言語独立性を実現できる．

1.1.1　オブジェクトの位置透過性

オブジェクトの位置透過性とは，あるオブジェクトが他のオブジェクトにメッセージを送信する場合に，相手オブジェクトがローカルマシン上で稼動しているか，リモートマシン上で稼動しているかを意識する必要がなく通信が可能であることを指す概念である．

[1] オブジェクトを直列化してファイルなどの永久記憶装置に保存することを永続化（Persistence）と呼ぶ．

[2] 同じシリアライズという言葉はマルチスレッドプログラミングの文脈では，複数のスレッドからアクセスを受ける資源が，それぞれのスレッドが順番に資源を利用できるよう調整する逐次化機能のことを指す．

クライアントからサーバへの通信要求はまずクライアントマシン上の ORB に行われ，ORB はローカルマシン上でサーバオブジェクトが存在すればそこにおくり，そうでなければサーバオブジェクトが存在するリモートマシン上の ORB に対して要求を送る．

1.1.2 オブジェクトの実装言語独立性

一方，オブジェクトの実装言語独立性とは，各言語で異なるデータ形式の違いを ORB により吸収し，実装言語の異なるオブジェクト間での通信が可能であることを指す概念である．オブジェクトから ORB への通信では，各言語固有のデータ形式を ORB の規格で定義された共通のデータ形式に変換し，ORB から各言語オブジェクトにデータが送られるときは，共通データ形式から言語固有のデータ形式に変換することで，ORB 間の通信が常に共通のデータ形式で行われるようにする．

1.2 インターフェース記述言語 (IDL:Interface Definition Language)

IDL とはソフトウェアコンポーネント[3]間のインターフェースを記述するプログラム言語であり，オブジェクトのもつ関数（メソッド）の名前や，その関数の引数・戻り値の型などの定義を行う．定義された IDL は IDL コンパイラを用いて各言語に対応したコードを生成し利用する．

IDL を用いることでプログラミング言語に非依存の記述が可能なため，例えば C と Java で書かれたコンポーネント間の通信が可能になる．

1.3 スタブとスケルトン

IDL コンパイラによって生成されたコードにはスタブとスケルトンが含まれる．

スタブとはクライアント側から利用され，クライアントのオブジェクトか見た場合，サーバ側にあるオブジェクトをあたかもローカルマシン上にあるかに見せかけるプロキシ（代理）オブジェクトの役割を果たす．スタブのインスタンスはクライアントプログラムは起動時に生成され，スタブはサーバオブジェクトのインターフェースを持つため，クライアントプログラムの中ではスタブに対してメソッドを送信し，スタブの中では ORB を介してサーバ側でメソッドを呼び出しの要求を行う．

スケルトンとはサーバオブジェクトのインターフェースをクライアントから利用できるよう公開したものであり，IDL で定義されたインターフェースを持っている．クライアント側からのメソッド呼び出し要求に地足宛て，サーバ側の ORB ではサーバオブジェクトを管理するオブジェクトアダプタ（OA）がスタブと対応付けられているスケルトンを探し出す．

また，サーバの実装はスケルトンにあるインターフェースを実装したクラスを作成することにより実現する．

[3]大雑把に言えばプロセスのこと．細かく言うとサービスを提供するシステムの構成要素であり，複数使用可能，コンテキスト非依存，他コンポーネントとの接続可能，カプセル化（外部インターフェース以外の隠蔽），独立配布，バージョン管理可能，という特徴を持ったもの．

2　メッセージキュー

メッセージキューは非同期通信プロトコルの一種である．同期通信プロトコルとはリクエストに対してレスポンスを返すものである．一方，非同期通信プロトコルはリクエストに対して必ずもレスポンスを返すとは限らないものである．

非同期通信プロトコルでは，メッセージフィルタ (Topic-based Message Filtering) という機能を有するものが多い．これは，全てのメッセージを受信するのではなくて，必要なものだけを選択し，受信する仕組みである．

トピックベースのメッセージフィルタ (Topic-based Message Filtering) では，送信側が特定の"トピック"やチャンネルにメッセージを送信する．受信側は，自ら指定したトピックに送信されたメッセージを受信する．この方式では送信側が，どのメッセージを届けるかを指定している．

もう一つの方法として，コンテンツベースのメッセージフィルタ (Content-based Message Filtering) がある．これは受信側が，指定した制約にマッチするメッセージだけを受け取るものであり，この方式では受信側がどのメッセージを受け取るか指定している．

2.1　MQTT

ZeroMQ は出版 (Publish)-購読 (Subscribe) 型の配信モデルをサポートする軽量メッセージングライブラリであり，Facebook Messenger も内部で利用している．出版購読型の配信モデルは，送信側を出版と呼び，受信側を購読と呼ぶ．出版側は誰が購読するかは指定せず，ネットワーク上にデータを送出する．すなわち購読者に関する知識を持たない．一方，購読側は，ネットワーク上に流れるメッセージはすべて受信する[4]．すなわち出版側の情報を持たない．

MQTT はメッセージングブローカ (MQ サーバ) を必要とし，オープンソスのものでは，Mosquitto や RabbitMQ などと呼ばれるプログラムが存在する．

2.1.1　ブローカ (MQ サーバ) の起動 (RabbitMQ)

RabbitMQ サーバは以下のようにしてインストールする．

```
$ sudo apt-get install rabbitmq-server
```

最初に一回起動し，サーバ上に MQTT のプラグインをインストールし，これらのプラグインを有効にするために MQ サーバプログラムを再起動する．

```
sudo service rabbitmq-server start
sudo rabbitmq-plugins enable rabbitmq_mqtt
sudo rabbitmq-plugins enable rabbitmq_management
sudo service rabbitmq-server restart
```

起動に成功していれば，ブラウザから `http://localhost:15672` で管理画面にアクセス出来る．ユーザ，パスワードはそれぞれ `guest` である．

[4] もちろんメッセージフィルタにより興味のあるクラスのメッセージだけを受信する

2.1.2 ブローカ (MQ サーバ) の起動 (Mosquitto)

Mosquitto サーバは以下のようにしてインストールする.

```
$ sudo apt install mosquitto mosquitto-clients
```

MQ サーバはどちらか一つだけで良いので，すでに RabbitMQ サーバをインストールしている場合は，`sudo service rabbitmq-server stop` として，RabbitMQ サーバを停止し Mosquitto を利用するようにする.

```
$ sudo service mosquitto restart
```

これで MQ サーバが立ち上がり MQTT プログラムを利用できるようになる.

2.1.3 MQTT コマンドラインツールの活用

MQTT はトピックベースのメッセージフィルタ (Topic-based Message Filtering) を採用してる.
`mosquitto_sub`，`mosquitto_pub` というコマンドラインツールで出版配信を確認してみよう.

```
$ mosquitto_sub -t topic
```

とすると，`-t` オプションで指定された `topic` というトピックに送信されたメッセージだけを購読している.
ここで，別のターミナルから

```
$ mosquitto_pub -t topic -m hello
```

とすると，`topic` トピックに `hello` というメッセージを送信するため，先程のコマンドで受信出来ていることがわかる.
一方,

```
$ mosquitto_sub -t hello
```

とすると，`hello` トピックを購読するため，メッセージは受信していない．ここで,

```
$ mosquitto_pub -t hello -m world
```

とすると，`hello` トピックにメッセージを送信し，先程のコマンドで受信出来ていることを確認できる.

2.1.4 Publish-subscribe 型の通信プログラム例

ここでは，Publish-subscribe 型の通信プログラム例を見ていこう.
Publish(出版），Subscribe(購読）のプログラムは以下のようになる.
以下の出版プログラムでは,

```
    client.publish(topic, ss)
```

として topic トピックにメッセージ ss を送信している．

また，購読プログラムでは，

```
client.subscribe(topic)
```

としてメッセージフィルタを指定している．このプログラムでは，デフォルトは*を指定している．これは正規表現として扱われ，全てのトピックを受信している．

```
#!/usr/bin/python

import paho.mqtt.client as mqtt
import time, sys

# set topic name
topic = "topic"
if len(sys.argv) > 1:
    topic = sys.argv[1]

# Initialize mqtt client
client = mqtt.Client()

# Connect to broaker
client.connect("localhost",
               port=1883,
               keepalive=60)

#  Do 10 publish
for request in range(10):
    ss = "Hello World! {} on {}".format(
        request, topic)
    print("Sending: {}".format(ss))
    client.publish(topic, ss)

    time.sleep(1)
```

```
#!/usr/bin/python

import paho.mqtt.client as mqtt
import time, sys

def on_message(client, userdata, msg):
    print("Received message [{}] on \
topic [{}]".format(msg.payload, msg.topic))

# set topic-filter name
topic = "*"
if len(sys.argv) > 1:
    topic = sys.argv[1]

# Initialize mqtt client
client = mqtt.Client()

# Connect to broaker
client.connect("localhost",
               port=1883,
               keepalive=60)

# Set callback function
client.on_message = on_message

# Topic-based filter
client.subscribe(topic)

# event loop
client.loop_forever()
```

Python のライブラリは以下のようにしてインストールする．

```
$ sudo apt install python3-paho-mqtt
```

2.2 ZeroMQ

ZeroMQ はメッセージングライブラリである．その特徴は中心的なメッセージングブローカを有さない点にある．このような仕組みを有する物をブローカレスと呼ぶこともある．

ZeroMQ 自体は，遠隔手続き呼び出しやタスクが分散した状態を想定した同期通信である Request-reply, データが分散している状態を想定した非同期通信である Publish-subscribe，並列計算においてタスクを分岐させたり収拾するのに用いる非同期通信である Push-pull (pipeline) [5]などの通信パターンを提供している．

2.2.1 Publish-subscribe 型の通信プログラム例

ここでは，Publish-subscribe 型の通信プログラム例を見ていこう．
Publish(出版），Subscribe(購読）のプログラムは以下のようになる．

```
#!/usr/bin/env python
#
# Publisher program
#   Sends "Hello World" to subscriber

import time
import zmq

print("Connecting to 5555")
context = zmq.Context()
socket = context.socket(zmq.PUB)
socket.connect("tcp://localhost:5555")

#  Do 10 publish
for request in range(10):
    ss = "Hello World! {}".format(request)
    print("Sending: {}".format(ss))
    socket.send(ss.encode('utf-8'))

    time.sleep(1)
```

```
#!/usr/bin/env python
#
# Subscriber program
#   Read message from publisher

import zmq

context = zmq.Context()

print("Wait for Connecting")
socket = context.socket(zmq.SUB)
socket.bind("tcp://*:5555")

# set filter option "" means pass through
socket.setsockopt(zmq.SUBSCRIBE, "".encode('utf-8'))

while True:
    #  Wait for next request from client
    message = socket.recv()
    print("Received: {}".format(message))
```

ZeroMQ はコンテンツベースのメッセージフィルタリングフィルタリング機構を提供している．
以下のプログラムでは""の部分を，"Hello"とすると，Hello という文字列から始まるメッセージのみを受信する．これを別の単語にすると，購読しない事が確認できる．

```
    socket.setsockopt(zmq.SUBSCRIBE, "".encode('utf-8'))
```

ZoroMQ は PUB/SUB とは別にサーバ（bind を実行するプログラム），クライアント (connect を実行するプログラム）が存在する．サーバは 1 つしか立てることが出来ず，クライアントは複数立ち上げることが出来る．この例はクライアントが出版側になっているが，サーバ側を出版側，クライアントを購読側，複数の購読プロセスを立ち上げることも可能である．
Python のライブラリは以下のようにしてインストールする．

```
$ sudo apt install python-zmq
```

C++のサンプルプログラムを以下に示す．

[5]Publish-subscribe は接続していないときのメッセージは捨てられる，一方，pipeline では接続していない時は送信や受信を待つ

```
#include <zmqpp/zmqpp.hpp>
#include <sstream>

int main(int argc, char *argv[]) {
  // initialize the 0MQ context
  zmqpp::context context;

  // generate a publish socket
  zmqpp::socket socket (context, zmqpp::socket_type::publish);

  // open the connection
  std::cout << "Connecting to 5555" << std::endl;
  socket.connect("tcp://localhost:5555");

  int request_nbr;
  for (request_nbr = 0; request_nbr != 10; request_nbr++) {
    // compose a message from a string and a number
    std::stringstream ss;
    ss << "Hello World " << request_nbr;
    std::cout << "Sending: " << ss.str() << std::endl;

    // send a message
    zmqpp::message message = ss.str();
    socket.send(message);

    // wait for next loop
    std::this_thread::sleep_for(std::chrono::seconds(1));
  }
}
```

```
#include <zmqpp/zmqpp.hpp>

int main(int argc, char *argv[]) {
  // initialize the 0MQ context
  zmqpp::context context;

  // generate a subscribe socket
  zmqpp::socket socket (context, zmqpp::socket_type::subscribe);

  // bind to the socket
  std::cout << "Wait for Connecting" << std::endl;
  socket.bind("tcp://*:5555");

  // set filter option "" means pass through
  socket.subscribe("");

  while (1) {
    // receive the message
    zmqpp::message message;
    // decompose the message
    socket.receive(message);
    std::string text;
    message >> text;
    std::cout << "Received: " << text << std::endl;
  }
}
```

C++のライブラリは以下のようにしてインストールする.

```
$ sudo apt install libzmqpp-dev
```

2.2.2 Request, reply 型の通信プログラム例

ここでは，同期型通信である Requst-reply 型の通信プログラム例を見ていこう．

```
#!/usr/bin/env python
#

import time
import zmq
import sys

# initialie request argument
i1 = 4
i2 = 7
print(sys.argv, len(sys.argv))
if len(sys.argv) > 1:
    i1 = int(sys.argv[1])
if len(sys.argv) > 2:
    i2 = int(sys.argv[2])
request = {'i1': i1, 'i2': i2}

print("Connecting to 5555")
context = zmq.Context()
socket = context.socket(zmq.REQ)
socket.connect("tcp://localhost:5555")

print("Sending: {}".format(request))
socket.send_json(request)

response = socket.recv_json()
print("Received: {}".format(response))
print("Answer is {}".format(response['i3']))
```

```
#!/usr/bin/env python
#
# Subscriber program
#   Read message from publisher

import zmq
import json

context = zmq.Context()

print("Wait for Connecting")
socket = context.socket(zmq.REP)
socket.bind("tcp://*:5555")

while True:
    #  Wait for next request from client
    message = socket.recv_json()
    print("Received: {}".format(message))
    reply = {'i3': message['i1']+message['i2']}
    print("Returns: {}".format(reply))
    socket.send_json(reply)
```

ここでは，ディクショナリ型のデータを JSON(JavaScript Object Notation) に変換し，通信を行っている．これは，データ構造を文字列で表すものであり，シリアライズ技術の一つと言える．JSON はもともと JavaScript 様に作られたフォーマットであるが，現在は広く一般的に使われている．

例えば，zmqpp ライブラリ (`http://zeromq.github.io/zmqpp/classzmqpp_1_1socket.html`) には JSON 形式でデータを送受信する機能はないが，受け取った文字列情報から JSON データに

復元することができる．これにより，位置透過性を実現することが出来る．

```
#include <zmqpp/zmqpp.hpp>
#include <jsoncpp/json/json.h>
#include <sstream>

int main(int argc, char *argv[]) {
  // initialize the OMQ context
  zmqpp::context context;

  // generate a publish socket
  zmqpp::socket socket (context, zmqpp::socket_type::request);

  // open the connection
  std::cout << "Connecting to 5555" << std::endl;
  socket.connect("tcp://localhost:5555");

  // create request message
  Json::Value root;
  root["i1"] = 4;
  root["i2"] = 7;

  // compose a message from JSON object
  std::stringstream ss;
  ss << root;
  printf("Sending: %s\n", ss.str().c_str());

  // send a request
  zmqpp::message request = ss.str();
  socket.send(request);

  // receive the response
  zmqpp::message message;
  // decompose the message
  socket.receive(message);
  std::string text;
  message >> text;

  printf("Received: %s\n", text.c_str());
  Json::Reader reader;
  Json::Value response;
  reader.parse(text, response);
  printf("Answer is %d\n", response["i3"].asInt());
}
```

3 XML-RPC

XML-RPC とは RPC プロトコルの一種であり、符号化に XML を採用し、転送機構に HTTP を採用している[6]。

RPC(remote procedure call) とは，あるプログラムから別のプログラムのサブルーチンを実行することであり、遠隔手続き呼び出しと呼ばれる。またオブジェクト指向における RPC は Remote Invocatoin あるいは、Remote Method Invocatoin と呼ぶことがある。CORBA も RPC の実装の一つといえる。

XML(Extensible Markup Language) とは、文章をコンピュータで読み込み可能にする符号化の規則であり、拡張可能なマーク付け言語と訳され、任意のデータ型をテキストとして表現するための機構になっている。実際に数多くの XML を利用した言語が開発されており、その中には RSS や Atom といった Web 技術や OpenOffice や GoogleDoc のデータフォーマット[7]、あるいはロボットの幾何モデル表現[8]等に広く利用されている。

XML は Markup と Content からなり、Markup は<と>で囲まれる。<と>で囲まれたものには start-tags `<tag>`、end-tags `</tag>`、empty-element tag `<tag/>`がある。tag の部分はに任意の文字列をいれることができ、これを tag と呼ぶ。Markup は tag の他に name と value の組からなる atribute を含めることができる。

例えば`<img src="madonna.jpg" alt='Foligno Madonna, by Raphael'/>` では、src と alt の attribute があり、`<step number="3">Connect A to B.</step>` は、number という attribute があり、その値が 3 となっている。また、tag で囲まれた `Connect A to B.` を element と呼ぶ。

HTTP(HyperText Transfer Protocol) とは，Web ブラウザと Web サーバの間でコンテンツの送受信に用いるリクエスト-レスポンス型のプロトコルである。リクエストには URL とメソッドがあり、指定された URL のリソースを取り出す GET[9]、クライアントからサーバにデータを送信する POST、画像のアップロード等指定した URL にリソースを保存する PUT、ヘッダ情報のみをリクエストする HEAD が代表的なものである。これ以外に URL のリソースを削除する DELETE、サーバがサポートしている HTTP のバージョンを調査する OPTION、サーバまでのネットワーク経路をチェックする TRACE、暗号化したメッセージを転送する CONNECT が HTTP/1.1 で定義されている。

XML-RPC の特徴の一つに `Introspection`(イントロスペクション) がある[10]．これは XML-RPC の標準仕様には含まれていないものの，ほとんどすべてのライブラリ実装に含まれている．これはサーバ側で提供するメンバ関数や，その呼び出し引数，返り値の情報をクライアントから取得できる機能であり，これにより，CORBA のようにインターフェース記述言語 (IDL) ファイルを定義し，これをコンパイルしてサーバ，クライアントプログラムで読み込み実行する必要はなく，クライアントはその実行時にサーバの提供する共通インターフェースに対応するクラスやメンバ関数を生成している．

---

[6]似たような技術に SOAP がある。SOAP(Simple Object Access Protocol) は XML-RPC を拡張したものであり、envelove/header/body 等のあたらしいタグが導入されている。また、最近では HTTP の GET/POST/DELETE/PUT を使いリソースを操作するという WebAPI の仕様を指す REST(Representational State Transfer) という仕組みが流行している。

[7]OpenOffice で作成した文書 (odt) や表 (ods) は XML データを zip で固めたものである。拡張子を.zip に変更し unzip すると、中身をテキストファイルで取り出すことができる。

[8]http://openrave.programmingvision.com/index.php/Started:Formats

[9]telnet www.mech.t.u-tokyo.ac.jp 80 として GET / HTTP/1.0（リターン 2 回を押す）とすると結果を見ることができる。

[10]http://xmlrpc-c.sourceforge.net/introspection.html

3.1 XML-RPC の文法

XML-RPC の呼び出しは XML の<methodCall>タグで指定する。<methodCall>タグは<methodName>タグを持ち、そのエレメントに文字列でメソッド名を指定する。引数は<params>タグ内の<param>タグで指定する。それぞれの<param>タグには<value>というタグを持ち引数を指定する。引数の型は<i4>(or <int>), <boolean>, <string>, <double>, <dateTime.iso8601>, <base64> が利用できる。以下は examples.getStateName を引数 41 で呼び出している。

```
<?xml version="1.0"?>
<methodCall>
   <methodName>examples.getStateName</methodName>
   <params>
      <param>
         <value><i4>41</i4></value>
      </param>
   </params>
</methodCall>
```

3.2 XML-RPC の例

XML-RPC のサンプルサーバが http://gggeek.altervista.org/sw/xmlrpc/demo/server/server.php[11] に立ち上がっている.

実際に XML-RPC を使った通信は以下のようになる. まず以下のようにサーバと通信を開始し、

```
$ telnet gggeek.altervista.org  80
Trying 104.21.22.124...
Connected to gggeek.altervista.org.cdn.cloudflare.net.
Escape character is '^]'.
```

キーボードから以下のように入力する[12]。

```
POST /sw/xmlrpc/demo/server/server.php HTTP/1.0
User-Agent: Frontier/5.1.2 (WinNT)
Host: gggeek.altervista.org
Content-Type: text/xml
Content-length: 117

<?xml version="1.0"?>
<methodCall>
   <methodName>system.listMethods</methodName>
   <params></params>
</methodCall>
```

こうすると以下のような結果が変えってくるはずである (ヘッダを除いている)。

[11] ここにブラウザでアクセスしてもエラーになるだけである

[12] 厳密には送信する文字列の正しい長さを Content-length に入れる必要がある。カット＆ペーストをするとうまくいかない場合がある。

```
<methodResponse>
<params>
<param>
<value><array>
<data>
<value><string>examples.getStateName</string></value>
<value><string>examples.sortByAge</string></value>
<value><string>examples.addtwo</string></value>
<value><string>examples.addtwodouble</string></value>
...
</methodResponse>
```

これでシステムが提供する、メソッド名一覧を得ることができる。ここから、examples.addtwo というメソッドを利用してみる。そのためには，`system.methodSignature` というメソッド利用することで，`examples.addtwo` メソッドの引数と返り値を知ることが出来る.

```
POST /sw/xmlrpc/demo/server/server.php HTTP/1.0
User-Agent: Frontier/5.1.2 (WinNT)
Host: gggeek.altervista.org
Content-Type: text/xml
Content-length: 192

<?xml version="1.0"?>
<methodCall>
   <methodName>system.methodSignature</methodName>
   <params>
     <param><value><string>examples.addtwo</string></value></param>
   </params>
</methodCall>
```

```
<methodResponse>
<params>
<param>
<value><array>
<data>
<value><array>
<data>
<value><string>int</string></value>
<value><string>int</string></value>
<value><string>int</string></value>
</data>
</array></value>
</data>
</array></value>
</param>
</params>
</methodResponse>
```

これを参考して，以下を送信する。

```
POST /sw/xmlrpc/demo/server/server.php HTTP/1.0
User-Agent: Frontier/5.1.2 (WinNT)
Host: gggeek.altervista.org
Content-Type: text/xml
Content-length: 209

<?xml version="1.0"?>
<methodCall>
   <methodName>examples.addtwo</methodName>
   <params>
     <param><value><i4>4</i4></value></param>
     <param><value><i4>7</i4></value></param>
   </params>
</methodCall>
```

すると

```
<methodResponse>
<params>
<param>
<value><int>11</int></value>
</param>
</params>
</methodResponse>
```

のような結果を得ることができる。

3.3 XML-RPC クライアントライブラリ

実際には各言語毎に利用しやすいツールやライブラリが存在し、それを活用するとよい

3.3.1 コマンドラインツール

コマンドラインからは以下のように呼び出すことが出来る.

```
$ xmlrpc http://gggeek.altervista.org/sw/xmlrpc/demo/server/server.php system.listMethods
$ xmlrpc http://gggeek.altervista.org/sw/xmlrpc/demo/server/server.php
   system.methodSignature examples.addtwo
$ xmlrpc http://gggeek.altervista.org/sw/xmlrpc/demo/server/server.php examples.addtwo i/4 i/7
```

xmlrpc コマンドは以下のようにしてインストールする

```
$ sudo apt-get install libxmlrpc-core-c3-dev
```

3.3.2 Python

Python3 からは以下のように呼び出すことが出来る.

```
import xmlrpc.client
server = xmlrpc.client.ServerProxy('http://gggeek.altervista.org/sw/xmlrpc/demo/server/server.php')
result = server.examples.addtwo(4,7)
```

また，Python2 からは以下のように呼び出すことが出来る．

```
$ python
>>> import xmlrpclib
>>> server = xmlrpclib.Server('http://gggeek.altervista.org/sw/xmlrpc/demo/server/server.php')
>>> result = server.examples.addtwo(4,7)
>>> print result
```

3.3.3 C++

C++では，libxmlrpc を用いて以下のようなプログラムを作ることが出来る．

```
#include "SystemProxy.h"

int main (int argc, char **argv) {
  XmlRpcClient::Initialize("example", "0.1");
  XmlRpcClient server ("http://gggeek.altervista.org/sw/xmlrpc/demo/server/server.php");

  SystemProxy proxy = SystemProxy(server);

  int ret = proxy.addtwo(4, 7);

  printf("%d\n", ret);
}
```

SystemProxy は代理オブジェクトであり，xml-rpc-api2cpp という，XML-RPC のイントロスペックション API をサポートしたサーバから，C++の代理オブジェクトクラスを生成するツールであり，

```
xml-rpc-api2cpp http://gggeek.altervista.org/sw/xmlrpc/demo/server/server.php examples SystemProxy
```

として実行する．

ツール，ライブラリは以下のようにしてインストールする．

```
$ sudo apt install libxmlrpc-c++8-dev xmlrpc-api-utils
```

4 CORBA

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 800 以上の IT ベンダ，組織，個人が参加する組織である 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ためのミドルウェアである．

複数のプラットフォーム (Windows, Linux) 上での様々なプログラミング言語 (C, C++, Java, Python) のミドルウェアが提供されており，複数の環境や言語が混在した分散アプリケーションを構築することができるが，国際規格団体 (OMG) で定義された規格である点が最大の特徴であり，これまでに 20 年以上に渡って大規模なエンタープライズ・プロジェクトで利用されてきている．

CORBA では全てのオブジェクトは CORBA ネームサーバと呼ばれる ORB を介してのみ通信を行い，オブジェクト間で直接通信を行うことはない．

4.1 GIOP, IIOP, POA

GIOP(General Inter-ORB Protocol) とは OMG が管理する ORB 間の抽象プロトコルの標準規格であり，IIOP(Internet Inter-ORB Protocol) とは GIOP の TCP/IP 上での実装である．GIOP では，IDL のデータ型を ORB 間の通信データ用のバイト列にマッピングする規程を CDR(The Common Data Representation)，リモートオブジェクトへの参照の形式の規定である IOR(Interportable Object Reference)，リモートメソッド呼び出し，呼び出しへの応答メッセージ等のオブジェクト間で通信されるメッセージ，を定義する．

POA(Portable Object Adapter) はオブジェクトアダプタの仕様であり，サーバオブジェクトを管理する機能を提供するものである．

4.2 CORBA C 言語サンプルプログラム

4.2.1 プログラムのコンパイルと実行

```
1   module EchoApp {
2     interface Echo {
3       string echoString(in string input);
4     };
5   };
```

を IDL の簡単なサンプルとして，echo.idl としたとき，

```
$ omniidl -bcxx echo.idl
```

として IDL compiler によりコンパイルし，オブジェクトの実装と CORBA ORB 間の変換を行うためのソースコードを生成する．

このとき，omniidl コマンドがみつからない場合は

```
$ sudo apt-get install omniidl omniidl-python libomniorb4-dev omniorb-nameserver
```

のようにしてインストールする．生成されたコードは，スタブコード（クライアント側で利用され

るプログラム），スケルトンコード（サーバ側で利用されるプログラム），共通コード（クライアント，サーバの両方で使われるプログラム）が含まれている．

echo.idl を使ったサーバクライアントプログラムとして echo-server.cpp echo-client.cpp を紹介する．

```
$ g++ -o echo-server echo-server.cpp echo_impl.cpp echoSK.cc `pkg-config omniORB4 --cflags --libs`
$ g++ -o echo-client echo-client.cpp echoSK.cc `pkg-config omniORB4 --cflags --libs`
```

のようにしてプログラムをコンパイルする．echo-server.cpp, echo-client.cpp, echo_impl.cpp, echo_impl.h は次頁以降を記載してある．それ以外のファイルは idl ファイルから生成されるはずである．最後にターミナルで

```
$ ./echo-server  -ORBInitRef NameService=corbaloc:iiop:127.0.0.1:2809/NameService
```

としてサーバを実行し，別のターミナルで

```
$ ./echo-client  -ORBInitRef NameService=corbaloc:iiop:127.0.0.1:2809/NameService
```

としてクライアントを実行する．引数をつけることでどのマシンの ORB を利用するか，という情報を与えることが出来る．2809 は ORB のデフォルトのポート番号である．

4.2.2 トラブルシューティング

もし， `Caught CORBA exception: system exception` または `failed binding of service IDL:omg.org/CORBA/TRANSIENT:1.0` と表示され，うまくいかない場合は，まずはプログラム実行時の引数の打ち間違いの可能性を疑い，再度確認しよう．

その上で同じエラーが出る場合は， `/var/lib/omniorb/`以下のファイルを消して，ネームサーバを再起動すると良い．これらの作業には管理者権限が必要なので sudo コマンドを使い、何度も注意しながら，`sudo rm /var/lib/omniorb/omninames*`とする．

その上で，`sudo /etc/init.d/omniorb4-nameserver restart` としてネームサーバを再起動し改めてプログラムを立ち上げてみよう．

4.2.3 C 言語サンプルプログラムの解説

`echo.idl` では `POA_EchoApp::Echo` というクラスを継承した `echoApp_impl` クラスのメンバ関数として `echoString()` を定義している．これに対応する C++言語の実装として，以下のように `echo_impl.h`，`echo_impl.cpp` を記述する．

echo_impl.h:

```
1   #ifndef __ECHO_IMPL_H__
2   #define __ECHO_IMPL_H__
3   #include "echo.hh"
4
5   class EchoApp_impl : public POA_EchoApp::Echo
6   {
7   public:
8     virtual char * echoString(const char * message);
9   };
10
11  #endif
```

echo_impl.cpp:

```
1   #include "echo_impl.h"
2
3   #include <iostream>
4
5   using namespace std;
6
7   char * EchoApp_impl::echoString(const char * message) {
8     cout << "C++ (omniORB) server: " << message << endl;
9     char * server = CORBA::string_alloc(42);
10    strncpy(server, "Message from C++ (omniORB) server", 42);
11    return server;
12  }
13
```

ここでは，サーバのプログラムの 21 行目の `EchoApp_impl * service = new EchoApp_impl;` の部分でサーバ側のオブジェクトとして，このオブジェクトを生成している点を理解しよう．

また，クライアント側のプログラムは，51 行目の `service_server = EchoApp::Echo::_narrow(obj);` として，6 行目のサーバの代理オブジェクトである `EchoApp::Echo_ptr service_server;` を初期化し，66 行目の `string result = service_server->echoString("Hello from C++");` で代理オブジェクト経由でメンバ関数 (`echoString`) を呼び出している．

クライアントプログラムだけ見て，`EchoApp::Echo_ptr service_server;` を，`echoString()` メンバ関数を有するオブジェクトとみなせば，通常のメンバ関数呼び出として，文字列を引数とし，返り値の文字列を表示しているプログラムとして理解できる．

echo-server.cpp:

```
1  // https://xennis.org/wiki/CORBA#Server
2
3  #include "echo.hh"
4  #include "echo_impl.h"
5  #include <iostream>
6
7  /** Server name, clients needs to know this name */
8  #define SERVER_NAME             "Example"
9  using namespace std;
10
11 int main(int argc, char ** argv)
12 {
13   try {
14     CORBA::ORB_ptr orb = CORBA::ORB_init(argc, argv);
15     CORBA::Object_var poa_obj = orb->resolve_initial_references("RootPOA");
16     PortableServer::POA_var poa = PortableServer::POA::_narrow(poa_obj);
17     PortableServer::POAManager_var manager = poa->the_POAManager();
18     //--------------------------------------------------------------------
19     // Create service
20     //--------------------------------------------------------------------
21     EchoApp_impl * service = new EchoApp_impl;
22     try {
23       CORBA::Object_var ns_obj = orb->resolve_initial_references("NameService");
24       if (!CORBA::is_nil(ns_obj)) {
25         CosNaming::NamingContext_ptr nc = CosNaming::NamingContext::_narrow(ns_obj);
26         CosNaming::Name context_name;
27         context_name.length(1);
28         context_name[0].id = CORBA::string_dup("context");
29         context_name[0].kind = CORBA::string_dup("");
30         CosNaming::NamingContext_var tc;
31         try{
32           tc = nc->bind_new_context(context_name);
33         }catch(CosNaming::NamingContext::AlreadyBound& ex) {
34           CORBA::Object_var obj;
35           obj = nc->resolve(context_name);
36           tc = CosNaming::NamingContext::_narrow(obj);
37           if( CORBA::is_nil(tc) ) {
38             cerr << "Failed to narrow naming context." << endl;
39             return 0;
40           }
41         }
42         CosNaming::Name object_name;
43         object_name.length(1);
44         object_name[0].id = CORBA::string_dup(SERVER_NAME);
45         object_name[0].kind = CORBA::string_dup("");
46         try{
47           tc->bind(object_name, service->_this());
48         }catch(CosNaming::NamingContext::AlreadyBound& ex) {
49           cerr << "Example binding already existed -- rebound" << endl;
50           tc->rebind(object_name, service->_this());
51         }
52         cout << argv[0] << " C++ (omniORB) server '" << SERVER_NAME << "' is running .." << endl;
53       }
54     } catch (CosNaming::NamingContext::NotFound &) {
55       cerr << "Caught CORBA exception: not found" << endl;
56     } catch (CosNaming::NamingContext::InvalidName &) {
57       cerr << "Caught CORBA exception: invalid name" << endl;
58     } catch (CosNaming::NamingContext::CannotProceed &) {
59       cerr << "Caught CORBA exception: cannot proceed" << endl;
60     }
61     manager->activate();
62     orb->run();
63     delete service;
64     orb->destroy();
65   } catch (CORBA::UNKNOWN) {
66     cerr << "Caught CORBA exception: unknown exception" << endl;
67   } catch (CORBA::SystemException &) {
68     cerr << "Caught CORBA exception: system exception" << endl;
69   }
70 }
```

echo-client.cpp:

```
// http://omniorb.sourceforge.net/omni41/omniORB/omniORB002.html#toc13
// https://xennis.org/wiki/CORBA_-_Advanced_example_with_server-client_in_Java_and_C%2B%2B#C.2B.2B_non-interactive_client
#include "echo.hh"
#include <iostream>

EchoApp::Echo_ptr service_server;
using namespace std;

/** Name is defined in the server */
#define SERVER_NAME             "Example"

int main(int argc, char ** argv)
{
  try {
    //-------------------------------------------------------------------------
    // Initialize ORB object.
    //-------------------------------------------------------------------------
    CORBA::ORB_ptr orb = CORBA::ORB_init(argc, argv);
    //-------------------------------------------------------------------------
    // Resolve service
    //-------------------------------------------------------------------------
    service_server = 0;
    try {
      //-----------------------------------------------------------------------
      // Bind ORB object to name service object.
      // (Reference to Name service root context.)
      //-----------------------------------------------------------------------
      CORBA::Object_var ns_obj = orb->resolve_initial_references("NameService");
      if (!CORBA::is_nil(ns_obj)) {
        //---------------------------------------------------------------------
        // Bind ORB object to name service object.
        // (Reference to Name service root context.)
        //---------------------------------------------------------------------
        CosNaming::NamingContext_ptr nc = CosNaming::NamingContext::_narrow(ns_obj);
        //---------------------------------------------------------------------
        // The "name text" put forth by CORBA server in name service.
        // This same name ("MyServerName") is used by the CORBA server when
        // binding to the name server (CosNaming::Name).
        //---------------------------------------------------------------------
        CosNaming::Name name;
        name.length(2);
        name[0].id = CORBA::string_dup("context");
        name[0].kind = CORBA::string_dup("");
        name[1].id = CORBA::string_dup(SERVER_NAME);
        name[1].kind = CORBA::string_dup("");
        //---------------------------------------------------------------------
        // Resolve "name text" identifier to an object reference.
        //---------------------------------------------------------------------
        CORBA::Object_ptr obj = nc->resolve(name);
        if (!CORBA::is_nil(obj)) {
          service_server = EchoApp::Echo::_narrow(obj);
        }
      }
    } catch (CosNaming::NamingContext::NotFound &) {
      cerr << "Caught corba not found" << endl;
    } catch (CosNaming::NamingContext::InvalidName &) {
      cerr << "Caught corba invalid name" << endl;
    } catch (CosNaming::NamingContext::CannotProceed &) {
      cerr << "Caught corba cannot proceed" << endl;
    }
    //-------------------------------------------------------------------------
    // Do stuff
    //-------------------------------------------------------------------------
    if (!CORBA::is_nil(service_server)) {
      cout << "EchoApp client is running ..." << endl;
      string result = service_server->echoString("Hello from C++");
      cerr << "The object said " << result << endl;
    }
    //-------------------------------------------------------------------------
    // Destroy OBR
    //-------------------------------------------------------------------------
    orb->destroy();
  } catch (CORBA::UNKNOWN) {
    cerr << "Caught CORBA exception: unknown exception" << endl;
  }
}

```

4.3 CORBA Python 言語サンプルプログラム

echo.idl から Python 用のプログラムを作ってみよう．IDL コンパイルならびにクライアント，サーバプログラムのコンパイルは

```
$ omniidl -bpython echo.idl
```

として行う．これで，EchoApp, EchoApp___POA という２つのディレクトリと，echo_idl.py というファイルができるはずである．このとき，omniidl: Could not import back-end 'python' と表示されこれらのファイルが生成されない場合は，

```
$ sudo apt-get install omniidl-python
```

のようにして omniidl の Python バックエンドをインストールする．

サンプルプログラムの実行は２つターミナルを開き，それぞれのターミナルで以下のプログラムを実行する．実行はサーバのプログラムを先に行うこと．

```
$ python echo-server.py -ORBInitRef NameService=corbaloc:iiop:127.0.0.1:2809/NameService
$ python echo-cleint.py -ORBInitRef NameService=corbaloc:iiop:127.0.0.1:2809/NameService
```

また実行時に `ImportError: No module named omniORB` と表示される場合は

```
$ sudo apt-get install python-omniorb
```

として Python 用の CORBA ライブラリをインストールしたのち，再度サンプルプログラムを実行すること．

サーバープログラムの 9 行目で以下の `EchoApp__POA.Echo` クラスを継承しサーバ側のメンバ関数を実装している．

```
# Define an implementation of the Echo interface
class Echo_i (EchoApp__POA.Echo):
    def echoString(self, mesg):
        print "echoString() called with message:", mesg
        return mesg + ", Copy"
```

またクライアント側では，30 行目の

```
    eo = obj._narrow(EchoApp.Echo)
```

で EchoApp.Echo の代理オブジェクトに紐付け，38 行目で呼び出しを行っている．

`obj` を，`echoString()` メンバ関数を有するオブジェクトとみなせば，通常のメンバ関数呼び出として，文字列を引数とし，返り値の文字列を表示しているプログラムとして理解できる．

```
    message = "Hello from Python"
    result = obj.echoString(message)
```

echo-server.py:

```
#!/usr/bin/env python

# http://omniorb.sourceforge.net/omnipy3/omniORBpy/omniORBpy002.html
import sys
from omniORB import CORBA, PortableServer
import CosNaming, EchoApp, EchoApp__POA

# Define an implementation of the Echo interface
class Echo_i (EchoApp__POA.Echo):
    def echoString(self, mesg):
        print("echoString() called with message:", mesg)
        return mesg + ", Copy"

# Initialise the ORB and find the root POA
orb = CORBA.ORB_init(sys.argv, CORBA.ORB_ID)
poa = orb.resolve_initial_references("RootPOA")

# Create an instance of Echo_i and an Echo object reference
ei = Echo_i()
eo = ei._this()

# Obtain a reference to the root naming context
obj         = orb.resolve_initial_references("NameService")
rootContext = obj._narrow(CosNaming.NamingContext)

if rootContext is None:
    print("Failed to narrow the root naming context")
    sys.exit(1)

# Bind a context named "test.my_context" to the root context
name = [CosNaming.NameComponent("context", "")]
try:
    testContext = rootContext.bind_new_context(name)
    print("New test context bound")

except CosNaming.NamingContext.AlreadyBound as ex:
    print("Test context already exists")
    obj = rootContext.resolve(name)
    testContext = obj._narrow(CosNaming.NamingContext)
    if testContext is None:
        print("test.mycontext exists but is not a NamingContext")
        sys.exit(1)

# # Bind the Echo object to the test context
name = [CosNaming.NameComponent("Example", "")]
try:
    testContext.bind(name, eo)
    print("New Example object bound")

except CosNaming.NamingContext.AlreadyBound:
    testContext.rebind(name, eo)
    print("Example binding already existed -- rebound")

# Activate the POA
poaManager = poa._get_the_POAManager()
poaManager.activate()

# Block for ever (or until the ORB is shut down)
orb.run()
```

echo-client.py:

```
#!/usr/bin/env python

# http://omniorb.sourceforge.net/omnipy3/omniORBpy/omniORBpy002.html
import sys
from omniORB import CORBA
import CosNaming, EchoApp

# Initialise the ORB
orb = CORBA.ORB_init(sys.argv, CORBA.ORB_ID)

# Obtain a reference to the root naming context
obj         = orb.resolve_initial_references("NameService")
rootContext = obj._narrow(CosNaming.NamingContext)

if rootContext is None:
    print("Failed to narrow the root naming context")
    sys.exit(1)

# Resolve the name "test.my_context/ExampleEcho.Object"
name = [CosNaming.NameComponent("context", ""),
        CosNaming.NameComponent("Example", "")]
try:
    obj = rootContext.resolve(name)

except CosNaming.NamingContext.NotFound as ex:
    print("Name not found")
    sys.exit(1)

# Narrow the object to an Example::Echo
eo = obj._narrow(EchoApp.Echo)

if eo is None:
    print("Object reference is not an Example::EchoApp")
    sys.exit(1)

# Invoke the echoString operation
message = "Hello from Python"
result = obj.echoString(message)

print("I said '{}'. The object said '{}'.".format(message,result))

```

5 宿題

提出先：ITC-LMS を用いて提出すること

提出内容：以下の問題の実行結果の画面をキャプチャしファイル名は「問題番号.png」とし，また講義中にでてきたキーワードについて知らなかったもの, 興味のあるものを調べ「学籍番号.txt」としててアップロードすること．テキストファイルはワードファイルなどだと確認出来ないことがあるため，emacs/vi 等のテキストエディタを使って書こう．プログラムが長くなりキャプチャ画面に入り切らなくなってきたらプログラムファイルと実行結果を「問題番号.txt」にまとめてアップロードしてよい．
画像で提出する場合は，各自のマシンの Mac アドレスが分かるようにすること．例えば画面中に ifconfig というコマンドを打ち込んだターミナルを表示すればよい．
ITC-LMS にアップロードする際には講義・宿題の感想を必ずコメントに記すこと．また授業中に質問した者はその旨を記すこと．質問は成績評価時の加点対象となる．

キーワード：分散オブジェクト通信，RPC におけるイントロスペクション，出版購読型モデルにおけるメッセージのフィルタリング，JSON と YAML

1. CORBA のシステム構成図を描いてみよ．図の中に，クライアント，サーバ，ORB，スタブ，スケルトン，IDL，直列化といった単語が入るようにすること．

2. `http://yubin.senmon.net/service/xmlrpc/` で，XML-RPC による，郵便番号のチェックや住所を取得するサービスが稼働している．ここにアクセスし，郵便番号 113-8656 に対する情報を取得するプログラムを作成せよ．

   ヒント：コマンドラインからは以下のようにして情報を取得できる．文字コードを正しく表示するには，パイプで `ascii2uni -a 7` につなげればよい．

   ```
   xmlrpc http://yubin.senmon.net/service/xmlrpc/ yubin.fetchAddressByPostcode 1138656
   ```