プロセス間通信（サーバー・クライアント)について

同じコンピュータで動いている複数のプロセス(実行中のプログラムのこと)、もしくは異なるコンピュータで動いている複数のプロセスの間の通信（情報の受け渡し）のための方法

ここでは、一つのプロセスをサーバー、他のプロセスをクライアントとする方法について述べる。

サーバーは文字通り、クライアントからの要求を受けて何らかの仕事をする「受け」のプロセス、クライアントはサーバーに対して要求を出してサーバーに仕事をさせるプロセス。例をあげれば、ウェブページの閲覧を提供しているプロセスがサーバー、それにアクセスするブラウザがクライアント。

1. socket という仕組みを使う。

サーバーはクライアントからの接続要求を受け付けた後、そのクライアントから送られてきたメッセージをほぼそのまま返す。

以下では『ソケット』という概念が重要---これは 2 つのプロセス(サーバーとクライアント)をつなぐパイプ（管）のようなものと考えるとよいだろう。以下ではソケットを作り、それをサーバーとクライアントそれぞれに結びつけている(bind している)

サーバーの例

```
#! /usr/bin/python
# -*- cidubg:utf-8 -*-
# server1.py
import socket        # socket モジュールをインポート
from time import ctime                    # 現在の時刻を文字列に変換

host = '127.0.0.1'  # このプログラムを実行する PC が対象
port = 9876         # 相手の IP アドレスと使用するポート番号を指定
BufSize = 1024     # 受信できる文字列のサイズ

#socket を作る:。
# AF_INET: IPv4 インターネット・プロトコルの使用
# SOCK_STREAM: TCP/IP を用いた STREAM 型のソケットの使用
serversock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

# 作成したソケットを サーバー(host:port で指定)に関連付けする
serversock.bind((host,port))

# 受け付ける client の数を 1 とする
serversock.listen(1)

print 'Waiting for connections...'
#   接続要求を受け取り、クライアントの IP アドレスとポート番号を記録
clientsock, client_address = serversock.accept()

# 無限ループ
while True:
      rcvmsg = clientsock.recv(BufSize)   #クライアントからの文字列(1KB)受信
      print 'Received -> %s' % (rcvmsg)   #受信したメッセージを表示
      if rcvmsg == '':              # 受信した文字列が空(=接続断)ならループを終了
            break
      print 'Type message...'
      clientsock.sendall('[%s] %s'%(ctime(), rcvmsg))      # 文字列を送信

clientsock.close()  # ソケットを閉じる
```

以下はクライアント:の例:

```
#! /usr/bin/python
# -*- cidubg:utf-8 -*-
# client1.py
import socket
host = '127.0.0.1'              # サーバーの IP アドレスとポート番号
port = 9876
BufSize = 1024    # 受信できる文字列のサイズ

# サーバーと通信するためのソケット作成
clientsock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
clientsock.connect((host,port))

while True:                     # 無限ループ
    c_msg = raw_input('> ')         # プロンプトを出して入力待ち
    if c_msg == '':                 # 入力なしなら終了
      break
    #
    clientsock.send(c_msg)          # 入力文字列をサーバーに送る
    rcvmsg = clientsock.recv(BufSize)       # サーバーから受信
    print 'Received -> %s' % (rcvmsg)       # それを表示
    if rcvmsg == '':                # 空文字列ならば終了
      break
clientsock.close()              # ソケットを閉じる
```

演習 1. 自分の PC でまず server1.py を実行する。次に『別なコマンドプロンプト」(もしくは Terminal)上で client1.py を実行する。client1.py のもとでいろいろな文字列を入力してみる(その都度 Enter キーを押すのをわすれないこと)。最後に、「コントロールキーを押したまま z を押す」(これをコントロール Z,もしくは Ctrl-Z とか^Z と表す)。どうなるだろうか？またそれはなぜだろうか？
(ヒント: コマンドプロンプト上では Ctrl-Z は end-of-file を意味します。Linux では Ctrl-D です。なお、WIndows では Ctrl-Z は『やり直し／もとに戻す」ためのショートカット)

演習 2. 自分の PC でまず server1.py を実行する。次に『別なコマンドプロンプト」(もしくは Terminal)上で client1.py を実行する。さらに、『別なコマンドプロンプト」(もしくは Terminal)上で client1.py を実行する。(つまり client1.py を２つ別々に実行させる)
それぞれの client1.py のもとでいろいろな文字列を入力してみる(その都度 Enter キーを押すのをわすれないこと)。どうなるだろうか？

なお、今のプログラムで、127.0.0.1 (localhost) をサーバープロセスが実行されている PC の IP アドレス(例えば、192.168.41.111 など)にすると、「別な PC で実行されているサーバーとクライアントの間の通信ができる」

2. 接続できるクライアント数を増やす

接続できるクライアントは（プログラムから予想できるように)、1 個に限られていた。また、その一つが接続を切るとサーバーも終了するものであった（プログラムを読んで、そこの仕組みを理解しておこう)。

ここでは接続できるクライアント数を増やすことを考えよう。書き換えるのは server1.py である。まず、一つのクライアントだけに制限していた以下の行を

次のようにする：

```
# 受け付ける client の数を 1 とする
serversock.listen(1)
```

次のようにする：

```
# 受け付ける client の数を 5 とする
serversock.listen(5)
```

また、クライアント接続のコードを新たに while True: の中に入れ、ひとつのクライアントが接続を断ったら、その接続を閉じ(close)てから、新たなクライアントに接続するようにする:

```
# 無限ループ
while True:
    #   接続要求を受け取り、クライアントの IP アドレスとポート番号を記録
    clientsock, client_address = serversock.accept()
    print '...connected from:', client_address
    while True:
      rcvmsg = clientsock.recv(BufSize)   #クライアントからの文字列(1KB)受信
      print 'Received -> %s' % (rcvmsg)   #受信したメッセージを表示
      if rcvmsg == '':       # 受信した文字列が空(=接続断)ならループを終了
          break
      print 'Type message...'
      clientsock.sendall('[%s] %s'%(ctime(), rcvmsg))          # 文字列を送信
    clientsock.close()                  # クライアントとの接続を切る
```

以上の改変をしたものが server2.py である。

演習 3. 自分の PC でまず server2.py を実行する。次に別々のコマンドプロンプト」（もしくは Terminal)上で client1.py を 1 つずつ実行する。それぞれの client1.py のもとでいろいろな文字列を入力してみる(その都度 Enter キーを押すのをわすれないこと)。どうなるだろうか？どれに反応があるだろう

か。またそれはなぜか。さらに反応があったクライアントプログラムを Ctrl-Z を入力して接続を断ったら、何が起こるだろうか？残ったクライアントでいろいろな入力をしてみよう。どのような反応があるだろうか。

演習 4. クライアントから「滅びの言葉」（例えば balse）の入力があったら、サーバーが停止するようにさせたい。どこをどのようにすればこれが実現できるだろうか。

3. 複数のクライアントから同時に接続できるようにする

前節の server2.py では、演習 3 で見たように、同時に複数のクライアントに接続することはできない。しかし、チャットプログラムのように、複数の人が同じサーバーにアクセスして情報交換できるようにすることが望ましい場合もある。

つまり、サーバーは、クライアントからのアクセス要求があればそれらに接続するためのソケットを「それぞれ」作り、「それぞれ」からの情報を受信しすることができてほしい（チャットプログラムならばさらに、受信したメッセージをクライアント全てに送信するここが必要だろう）。このような用途ために、いわば『分身の術』(クライアントからの接続要求をうけ、メッセージを受信し、データを送信するという『プログラム』を複製する)を可能にする仕組みが色々用意されている。その一つの方法は「スレッド(thread)」と呼ばれるものである。

もっと簡単な方法は、Python の acyncore モジュールの dispatcher を使う方法である。Acyncore とはソケットを使った非同期通信のためのモジュール（スレッドは使っていない）で、通信が勝手なタイミング（非同期的）で行われる通信を扱うものである。基本的な使い方は、次ページの server3.py プログラムに見られるように、dispatcher クラスのインスタンスを作り、acyncore.loop を呼び出すことである。

最後の方にある server = EchoServer(HOST,PORT) が、dispacher クラスのインスタンスを作って、変数 server にセットしている。そして最後にある acyncore.loop によって、通信があるかどうおか監視が行われ、通信があった場合に EchoServer クラス(これ自体は asyncore.dispatcher クラスのサブクラス)で定義された handle_accept 関数の中身が実行される。

dispacher クラス(またそのサブクラスである EchoSerer クラス)の__init__メソッドは、インスタンスを作るもので、そこではソケットを作り、HOST:PORT に bind している。それに対し、handle_accept() メソッドは、listen 中のソケットで read イベントが発生した時に何をするかを記述したものである。accept メソッドにより接続を受け入れる。結果が None の場合は接続が行われなかったことを意味する。接続が行われた時は、クライアントのアドレスとソケットを返すので、このプログラムでは EchoHanler クラスにより、受信したメッセージを(send メソッドによりそのままクライアントに送り返すようにしている。

演習 5. このプログラムが 192.168.41.111 上で動いている。自分の PC を sirai-lab のネットワークに接続し、クライアントのプログラムを適切に修正して、192.168,41,111 上で動いている server3.py と通信してみよ。

```
#! /usr/bin/python
# server3.py

import asyncore, socket

HOST = '127.0.0.1'
PORT = 9876
BufSize = 1024
NoOfClients = 5

class EchoHandler(asyncore.dispatcher_with_send):
    def handle_read(self):
        data = self.recv(BufSize)
        if data:
            self.send(data)

class EchoServer(asyncore.dispatcher):
    def __init__(self, host, port):
        asyncore.dispatcher.__init__(self)
        self.create_socket(socket.AF_INET, socket.SOCK_STREAM)
        self.set_reuse_addr()
        self.bind((HOST,PORT))
        self.listen(NoOfClients)

    def handle_accept(self):
        pair = self.accept()
        if pair is None:
            pass
        else:
            sock, addr = pair
            print('Incoming connection from %s' % repr(addr))
            handler = EchoHandler(sock)

server = EchoServer(HOST,PORT)
asyncore.loop()
```

4. 音声認識システム Julius

Julius は-module オプションにより「モジュールモード」で起動する．モジュールモードでは，起動後，クライアントからの TCP/IP 接続待ちとなる．デフォルトのポート番号は 10500 であるが，ポート番号を 12345 にするには、-module 12345 とすればよい．

クライアントから接続を受けると，Julius は音声認識可能な状態となり、音声入力に対して音声認識を行い，クライアントへ認識結果を送信する．また，音声入力の開始・終了などのイベントも随時送出する．クライアントからは，認識の一時停止や再開といった動作制御，認識用文法の追加や削除などが行える．

クライアントとの接続は一対一を想定しており，複数クライアントの同時接続には対応していない．クライアントとのネットワークソケットが切断されると，Julius は認識を停止して，次のクライアントが接続するまで待ち状態となる．

クライアントから Julius にソケット経由でコマンドを送ることで, Julius を制御することができる．コマンドは，コマンドの文字列を末尾に改行コード"¥n" をつけて送信する．そのコマンドが引数を必要とする場合は，そのコマンド文字列の次の行として，引数を送る．

次の rapiro.py は、モジュールモードで動かした Julius をサーバーとして、音声認識による出力を受け取り、その出力をシリアル通信により Rapiro ロボットに送るプログラムである:

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-
import socket, serial
host = 'localhost'
port = 10500
clientsock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
clientsock.connect((host, port))
com = serial.Serial('/dev/ttyAMA0',57600,timeout=10)

while True:
        recv_data = clientsock.recv(512)
        com.write(recv_data)
```