

# YAMI Python Module Tutorial

Copyright © 2001-2008 Maciej Sobczak

---

## Contents

<b>1</b>	<b>Introduction</b>	<b>2</b>
<b>2</b>	<b>Echo</b>	<b>2</b>
2.1	Easy client . . . . .	2
2.2	Easy client with data . . . . .	3
2.3	Full blown client . . . . .	4
2.4	Server . . . . .	5
2.5	Asynchronous vs. synchronous . . . . .	5
2.6	Error handling . . . . .	7
<b>3</b>	<b>Calculator</b>	<b>7</b>
3.1	Client . . . . .	7
3.2	Server . . . . .	9

## 1 Introduction

If you already know the *YAMI Core Library* and you have read its tutorial, you will quickly notice that this text is almost exactly the same. The only substantial change is the source code. This apparent author's *laziness* is to show you that the *YAMI Python Module* is in fact very thin wrapper around the core library. Do not hesitate to run the examples from both tutorials in pairs to see that *YAMI* provides abstracts that are language-independent and that the connectivity between programs written in different languages is really achieved.

In this tutorial, there are two assumptions:

- Every server runs on the port 12340. This is arbitrary choice.
- Every client runs on the same machine as the server (this is for simplicity only – if you have access to the network, try the examples running the processes on different machines), and if it requires its own listening socket, its port is assigned by the operating system.

## 2 Echo

As a first example, you will write your own *echo* server and client. The echo server will wait for messages from clients and will print them on the screen. Apart from that, the server will also accept a special *shutdown* message. There will be no data sent back to clients.

### 2.1 Easy client

First, you will learn how to write simple client. In most distributed systems it is easier to write clients than servers.

The first client in this tutorial sends a message to the server that tells it to shut down. Here it is (file `shutdown.py`):

```
1 from YAMI import *
2
3 easySend('127.0.0.1', 12340, 2, 'echo', 'shutdown')
```

Yes, it is *that* easy.

The first line loads the library. It assumes that the library is available from the directory where the example code is placed.

The line with call to `easySend` alone is enough to send a message to the server. There are five parameters used in this command:

- `127.0.0.1` – the server's address. It is a string value and can have the form *comp.company.com* as well.
- `12340` – the server's port

- 2 – the level of the message sent. In this example it can be 1 as well (which means “only strings, please”). In most cases you want it to be Level2, unless you are sending a message to the component that understands only strings.
- `echo` – the name of the destination object. In *YAMI*, messages are sent to objects which means that there can be many destinations in the same server process.
- `shutdown` – the message name. Later you will see that this message causes the server to shut down.

Note that the `easySend` command is overloaded and that there is another version of this command that accepts additional parameter. We will use it later. Interesting? So let’s pass some data with the message.

## 2.2 Easy client with data

In the previous section, the message had only a name. In many cases it is enough, because it allows the server to make decisions about what action to perform, but there are situations where it would be nice to pass some data together with the message. In *YAMI*, there are two levels of messages with regard to the data that can be sent:

- Level1 – this level allows only strings and wide strings. It was designed for the components written in languages that do not understand other data types.
- Level2 – this level allows strings, wide strings, integers, doubles, bytes and arbitrary binary blocks.

Both levels are accessible in *YAMI Python Module*.

There can be many values of different types that can go with the same message at once. All the values are stored in the so-called *Parameter Set*, which is basically a list of values of chosen type.

The client that causes the server to print “Hello, YAMI!” message on the screen can look like here (file `hello.py`):

```
1 from YAMI import *
2
3 easySend('127.0.0.1', 12340, 2, 'echo', 'print',\
4         ['Hello, YAMI!'])
```

Yes, it is *that* easy.

In this example, we append the list of parameters to the message. The list has only one parameter of the string value “Hello, YAMI!”. Note also that the message name has changed.

### 2.3 Full blown client

The examples written so far have one thing in common: they send only one message to the server and quit. There is nothing that prevents you from sending millions of messages using the `easySend` function, but in terms of performance it will not be a good idea. The `easySend` function creates a *YAMI Agent* that is responsible for actually sending a message and destroys it after that. If you want to send many messages, it is a better idea to follow the scheme:

1. Create the *Agent* object.
2. Send the message with the help of the *Agent*. Repeat this step as many times as you wish.

In this section you will write the client according to this scheme. The client itself will be smarter than before, too – it will read the lines of text from its standard input and send each line as a separate message to the *echo* server. The code is below (file `printall.py`):

```

1 from YAMI import *
2
3 agent = Agent()
4 agent.domainRegister('echoserver', '127.0.0.1', 12340, 2)
5
6 try:
7     while 1:
8         line = raw_input()
9         agent.sendOneWay('echoserver', 'echo', \
10             'print', [line])
11 except EOFError:
12     pass
13
14 del agent

```

The *Agent* is created with its own listening port assigned by the operating system. This *Agent* does not use this port, since no information is sent back to the client. Please read about *Policies* to learn how to create the *Agent* object without the listening socket.

After the *Agent* is created, the remote domain is registered in it, like in *White Pages Book*. The address and port of destination *Agent* (the one created by the server) is remembered under the name `echoserver` for later use (the name in the address book is arbitrary).

The program performs a loop where it reads lines of text from standard input. Each line is then sent as a parameter with the message to the server.

Note that the `sendOneWay` method allows to send the message without any response from the server. The responses will be used in later examples.

## 2.4 Server

Finally, we will write the server (file `server.py`):

```

1 from YAMI import *
2
3 agent = Agent(12340)
4 agent.objectRegister('echo')
5
6 print 'server started'
7
8 while 1:
9     im = agent.getIncoming('echo', 1)
10
11     msgname = im.getMsgName()
12     if msgname == 'shutdown':
13         print 'received the shutdown message'
14         del im
15         break
16     else:
17         params = im.getParameters()
18         print params[0]
19
20     del im
21
22 del agent

```

After the *Agent* is created, the object is registered in it as a potential target of messages. If any message comes with the object name that is equal to the name of the registered object, the *Agent* will store the message in a queue, one queue for one object. Later, the server program goes into a loop that asks the *Agent* if there are any messages for the `echo` object and processes them. In this example, the server will block waiting for new message if the queue is empty – the last parameter to `getIncoming` method is non-zero.

When the message is finally retrieved, the server checks its name. If it is `shutdown`, it breaks the loop and terminates. Otherwise (which means that the message name is `print`; in real system you would also take some steps if the message has unknown or unacceptable name), the server gets the parameter list from the message and prints its first (indexed from 0) element on the standard output.

## 2.5 Asynchronous vs. synchronous

*YAMI* sends and processes messages asynchronously by default. This means two things:

- When you “send” the message, it is only put into the sending queue. There is a separate sender thread in the *Agent* that is responsible for actually

sending messages. The advantage? You can put many messages into the queue and go back to your work without waiting for them to be physically sent.

- When the message arrives at the remote *Agent*, it is put into the queue associated with each registered object and waits there for processing. The advantage? The message can be quickly received and stored while you are busy with long computations or processing previous messages.

The asynchronous sending and receiving is not always good, however. Let's say that you run the command:

```
$ python printall.py
```

*(I assume some Unix environment here)*

and later type some lines of text by hand. You are not really fast with typing, so the messages (each message contains one line of text) are actually sent to the server before you get your finger off the keyboard. But try this little experiment:

```
$ python printall.py < printall.py
```

which should send as many messages to the server as there are lines in the `printall.py` file. After reaching end-of-file, the process terminates... which can happen before the sender thread sends anything! Or you will see only few of the lines on the server side. There are three solutions to this problem:

- Put some delay at the end of the client process, but before the *Agent* is destroyed, so that the sender thread will have a chance to send all messages from its queue. Do not tell anybody.
- Rewrite the client and server so that the server will send back some response after each message. Wait for this response before the client sends the next message. Do not tell anybody that the excessive network traffic is your fault.
- Create the *Agent* in the client process *without* the separate sender thread. In such configuration, every message will be sent immediately, without storing in any queue – it will be synchronous. This way you will insure that each line is physically sent to the remote end (to the server) before next line is read from the standard input. Show your code to everybody.

Similar issues arise on the server side. Sadly, synchronous message retrieval is not possible in this version of *YAMI Python Module*. The only solution is to increase the queue size.

Look at the `printsync.py` and `servbig.py` files to see how to do it. If you compile them, you can try the test:

```
$ python servbig.py
```

in one console and (probably on different computer in the network, but you should change the address used in the client source code) in the other:

```
$ python printsync.py < somelongfile
```

Enjoy.

## 2.6 Error handling

Many different things can go wrong in the distributed system. Some of them will be your fault, some not. Almost every *YAMI* function and method can throw an exception. In the example programs presented so far there was no exception management – for simplicity and to avoid cluttering the source code. In real programs, you will for sure want to check for and manage the exceptions.

## 3 Calculator

In this second example, you will write the client-server pair where the client asks for something and the server *responds* to the messages, sending some data back to client. This is probably the most common way of developing distributed systems.

The calculator example consists of:

- The server that accepts messages: `add`, `sub`, `mul` and `div`, each with two integer parameters. Each message will be replied to, with the appropriate integer result.
- The client that sends messages asking the server to perform some computation.

I'm sure that after this example you will be able to write distributed systems of arbitrary complexity.

### 3.1 Client

Let's start with easy things. The client (file `calccclient.py`) looks like here:

```
1 from YAMI import *
2
3 serverhost = '127.0.0.1'
4 serverport = 12340
5 domainname = 'somDomain'
6 objectname = 'calculator'
7
8 agent = Agent()
9 agent.domainRegister(domainname, serverhost, serverport, 2)
10
```

```
11 msg = agent.send(domainname, objectname, 'add', [100, 20])
12
13 msg.wait()
14
15 status = msg.getStatus()
16
17 if status == eReplied:
18     retpar = msg.getResponse()
19     result = retpar[0]
20     print 'the result is', result
21 elif status == eRejected:
22     print 'the last message was rejected'
23 else:
24     print 'no correct reply'
25
26 del msg
27 del agent
```

The main difference to the previous example is that the `send` method is used instead of `sendOneWay`. The `msg` variable encapsulates the message token. It allows you to retrieve some information about the message sent to the remote objects. It also allows you to *synchronize* the client activity with the server – the `wait` method allows the client to wait for some change in the message's status.

As you can see, the message is sent with two integer parameters in the list of parameters (100 and 20). The message is sent asynchronously, so that you can continue your job without waiting for response and ask for it later. In this example there is nothing interesting to do anyway, so we decide to *wait* (with the call to `wait`) until something interesting happens to the message token. The process wakes up when (for example) the response arrives. The `status` of the message is examined to find out if there is a real reply or maybe some network error or something else. If there is a reply, we just retrieve the returning list of parameters and print its only (first) parameter.

The Calculator example shows also that there are two contexts where the list of parameters can be used:

- When the client sends a message to the server. The list of parameters can be supplied at the client-side and retrieved at the server-side.
- When the server replies to the message. The server can supply a list of parameters and the client can extract it after receiving the response.

These two contexts allow to send data in both directions.

Easy? So take also a look at the `calcclient2.py` file and play with it – it is an interactive calculator console (there is very simple code used for string parsing – remember to put spaces around operators, like in `'2 + 3'`).

There is one thing to remember, though. The `wait` method is your friend, but do not trust him. It may happen that the server crashes in the middle of

the computations. Then – the client does not receive any notification and the status of the message is always `ePending`. If you call the `wait` method, you are in troubles (the other solution, sometimes reasonable, is to periodically ask the message token if the response arrived – the client can decide by himself that something went wrong after, say, 10th try). To help you with this problem, the *Agent* object provides the *waker* service. The `calcclient2.py` file shows how to use it.

### 3.2 Server

The server is presented below (file `calcserver.py`):

```
1 from YAMI import *
2
3 serverport = 12340
4 objectname = 'calculator'
5
6 print 'starting the server'
7
8 agent = Agent(serverport)
9 agent.objectRegister(objectname)
10
11 print 'waiting for messages...'
12
13 while 1:
14     incoming = agent.getIncoming(objectname, 1)
15     msgname = incoming.getMsgName()
16
17     print 'I have received the message:', msgname
18
19     if msgname != 'add' and msgname != 'sub' and \
20        msgname != 'mul' and msgname != 'div':
21         print 'unknown name - rejecting'
22         incoming.reject()
23         continue
24
25     params = incoming.getParameters()
26     val1 = params[0]
27     val2 = params[1]
28
29     if msgname == 'add':
30         result = val1 + val2
31     elif msgname == 'sub':
32         result = val1 - val2
33     elif msgname == 'mul':
34         result = val1 * val2
```

```
35     else:
36         if val2 == 0:
37             print 'dividing by 0 not allowed'
38             print 'rejecting'
39             incoming.reject()
40             continue
41         else:
42             result = val1 / val2
43
44     incoming.reply([result])
45
46     del incoming
47
48 del agent
```

This server has one design quirk: the server does not perform any interaction with the user and there is no way to remotely shut the server down. Locally, the administrator can just kill the server process and bless the operating system that will hopefully garbage-collect all the resources used by the process.

I do not advocate this style of writing servers. The two clean solutions that I recommend are:

- Provide additional shut-down message, like in the Echo example. Just break the message loop when you receive the *shutdown* message.
- Provide (register) additional shut-down *object* and receive message also for this second object. The first message that will be received for this special shut-down object will tell you that it is time to terminate the server in a clean way.

Enjoy! And, of course, tell your friends about *YAMI*.