

# YAMI Python Module Reference

Copyright © 2001-2008 Maciej Sobczak

## Contents

<b>1</b>	<b>Introduction</b>	<b>2</b>
<b>2</b>	<b>Parameters</b>	<b>2</b>
<b>3</b>	<b>Agent</b>	<b>3</b>
3.1	Policies . . . . .	3
3.1.1	Examples . . . . .	5
3.2	New messages . . . . .	5
3.3	Incoming messages . . . . .	6
3.4	General API . . . . .	8
3.5	Additional functions . . . . .	11

## 1 Introduction

The *YAMI Python Module* is a set of classes provided for the Python interpreter and supporting the *YAMI* communication concepts. The module is in fact a wrapper for the *YAMI Core Library*.

The *YAMI Python Module* consists of two files:

- `yamipyc.so` – this is a shared library that should be loaded into the Python interpreter; the module defines and implements the *YAMI* classes and functions (on Windows it is `yamipyc.dll`),
- `YAMI.py` – this file implements the *shadow definitions* and provides the object-based interface to the definitions in the first file

It is perfectly possible to use the bare-bone interface from the `yamipyc.so` file, but the object-based interface is designed to make the use of *YAMI* as easy as possible. Therefore, only object-based interface is documented. For reference considering the other interface, see the source code.

Almost every function in the module can result in an error, which is set as a native Python exception.

## 2 Parameters

The concept of *Parameter Set* is implemented with the use of Python lists. There is no separate object that needs to be created before the message is sent. Also, the native Python lists are used after receiving the message or the response.

The list of parameters is typed, for example:

```
['hello', 123, 3.14159]
```

The list above contains three values: *string* value `hello`, *integer* value `123` and *double* (floating point) value `3.14159`.

There are two parameter types that cannot be easily represented with the native Python type: *byte* and *binary*.

The byte type can be wrapped in the `Byte` object, for example:

```
>>> b = Byte(5)
>>> b.getValue()
5
>>> list = [1, 3.14, Byte(5)]
>>> list[0]
1
>>> list[1]
3.14
>>> list[2].getValue()
5
>>>
```

The above example creates a single `Byte` object, initialized with the value 5. The value can be read with the `getValue` method. Later, the code creates the list containing one integer, one double and one byte object.

The binary type can be wrapped in the `Binary` object, for example:

```
>>> b = Binary(5)
>>> b.setAt(0, 100)
>>> b.getAt(0)
100
>>> b.set('abcdefgh', 5)
>>> b.getAt(2)
99
>>> b.get()
'abcde'
>>>
```

The code above creates a single `Binary` object of size 5. It sets the first byte (index 0) to the value 100 and reads it. Later, it sets the binary data to the first 5 characters of the given string. The third byte (index 2, pointing to the character 'c') has value 99 and the whole binary block can be read as string, giving the value 'abcde'.

Every function that accepts the list of parameters can also accept the empty list [], the list can be omitted, which is equivalent to the empty list.

### 3 Agent

*Agent* is responsible for routing, sending and receiving messages over the network. Conceptually, it is similar to the ORB in CORBA systems.

The *Agent* object is created in (almost) every program that uses the *YAMI* library, no matter if the given component acts as a *client* or as a *server* in the distributed system.

The multi-threading abilities (and other things that can vary between different *Agents*) of the *Agent* object greatly depend on the parameters that are provided during its creation – these parameters are called *Policies*.

#### 3.1 Policies

From the programmer's perspective, the *Policies* object is a structure containing the following fields:

**agentlevel** The level of the *YAMI* specification that this *Agent* will conform to. The *Agent* will not send nor receive messages from incompatible levels. Default: 2.

**objqmaxlength** The maximum length of the queue that is kept for each registered object. If the queue is full with respect to this parameter and there

is new incoming message for the given object, it will be rejected with `eOverflow` notification sent to the remote site (to the *Agent* that sent the message). Default: 10.

**objqmaxsize** The maximum size of the same queue (in bytes). Default value: 1048576.

**senders** The number of threads used for sending. If set to non-zero value, the sender thread(s) take care of all the packets that need to be sent to the remote *Agents*. If set to zero, no sender thread is created and the sending is performed directly in the context of requesting thread. Default: 1.

**sqmaxlength** The maximum length of queue for sender thread. Default: 256.

**sqmaxsize** The maximum size of the same queue. Default: 1048576.

**connpoolsize** The size of the connection pool (applies both to sending and receiving connections). Default: 10.

**sendtries** The number of times the sender module should try to send the packet before giving up (and reporting an error), if there is a network problem. Default: 5.

**hassocket** The flag stating if the *Agent* should create the listening socket. If set to zero, no receiving will be possible (this option can be useful for lightweight *Agents* that are created for one-way messaging). Default: 1.

**reuseaddr** The flag stating if the *Agent's* listening socket should be created with the `SO_REUSEADDR` option (when the flag has non-zero value). This policy can be useful for *Agents* that are frequently created and destroyed on the same port. Default: 1.

**haswaker** The flag stating if the waker module is needed. The waker module requires a separate thread for its own and in some lightweight configurations is not necessary. Default: 1.

**allowduplex** The flag stating if the *Agent* should accept incoming duplex connections. Accepting duplex connections has influence on the amount of network resources that can be potentially consumed by the server *Agent*. Default: 1.

**maxdplxconns** The number of incoming duplex connections that are kept in a pool. If the pool is full with respect to this parameter, no new incoming duplex connection will be accepted. Default: 100.

**receiveridle** The maximum idle time for the receiver thread, in milliseconds. This time is the maximum delay between sending the duplex request and including the socket in the set checked for incoming packets (when this is the first duplex request to the given address). Note that very short idle times may implicate more CPU resources used by the receiver thread. Default: 500.

Normally, the *Agent* can be created without any *Policies* – in this case it is created with the default values (the default values depend on the respective defaults in the Core Library – they can be changed by appropriate changes in the core headers and recompiling the Python module).

### 3.1.1 Examples

```
>>> p = Policies()
```

The code above creates new *Policies* object and assigns it to the `p` variable.

```
>>> p.sendtries
5
>>> p.sendtries = 10
>>> p.sendtries
10
>>>
```

The code above reads the `sendtries` value (5 by default), later sets this value to 10 and reads it again.

## 3.2 New messages

The functionality of the message sent to the remote object is encapsulated by the `Message` class.

The only way to create the instances of this class is to call `send` method of the `Agent` class.

The following methods are available in the `Message` class:

### `setTimeout(timeout)`

Sets the timeout on the message sent to the remote object. The `Agent` provides its own alarming subsystem for those objects, which want to protect themselves against crashes or errors in the remote objects. If there is no response from the remote object during the `timeout` seconds, the local `Agent` will set the `eTimedOut` status on the message, which allows the program to wake up, if it was waiting for the response after calling the `wait` method.

If the `wait` method was not called, the `setTimeout` still works – it sets the `eTimeout` on the message object if there is no packet coming from the remote object or the remote `Agent` in the meantime.

### `getStatus()`

Retrieves the message's status. The possible values are:

**ePosted** The message was posted to the queue owned by the sender thread.

- ePending** The message was successfully sent over the network.
- eReplied** There is a reply available for the given message.
- eRejected** The message was rejected by the remote object (but was received).
- eUnknownObj** The message was rejected by the remote *Agent*, because no object was registered with a matching name.
- eOverflow** The message was rejected by the remote *Agent*, because of the overflow in the remote object's queue.
- eNetError** The message was not successfully sent over the network.
- eTimedOut** The message was *timed out* by the waker module.
- eRejectByAgent** The message was rejected by the destination *Agent*. This may happen if the message with the parameter set of Level2 was sent to the *Agent* running on Level1.

#### **wait()**

This method *blocks* the calling thread until the status of the message changes to one of the status codes: **eReplied**, **eRejected**, **eUnknownObj**, **eOverflow**, **eNetError**, **eTimedOut** or **eRejectByAgent**. When this happens for any reason, the calling thread wakes up and should check for the message's status. Clients calling this function should consider also **setTimeout**, because without it, the **wait** can block *forever* (which can happen when the remote object crashes in the middle of processing).

#### **getResponse()**

Retrieves the parameters returned by the remote object as a list (possibly empty). If the status of the message is not **eReplied**, the exception will be set.

### **3.3 Incoming messages**

The functionality of the message received from the remote site is encapsulated by the **IncomingMsg** class.

The instances of this class are created by the *Agent* and for the client code provide access to the message parameters and to some essential operations like replying to the message.

The only way to create the object of this class is to call **getIncoming** method of the **Agent** object.

The following methods are available in the **Message** class:

#### **getMsgName()**

Gets the name of the message.

#### **getObjectName()**

Returns the name of the object to which this message was sent. This method makes sense only for default objects, but can be called by any other object as well (then, it returns the same name as the name given to the `getIncoming` method of the `Agent` object).

**getParameters()**

Returns the (possibly empty) list of parameters sent with the message.

**getLevel()**

Retrieves the level of *YAMI* specification, to which the sending object complies. It can help to choose the best reply, according to the abilities of the remote (sending) object.

**getSourceAddr()**

Retrieves the address of originating *Agent*. This address is preserved with message forwarding, which means that it relates to the *Agent* that originated the message, not to the one that forwarded it.

**getSourcePort()**

Retrieves the port of originating *Agent*, see above.

**getEphemericalAddr()**

Retrieves the address of connecting socket. This address might be different than the source address in case of forwarded message. The intent of this function is to allow programs to send outgoing messages to remote agents via duplex channels initiated already by those remote agents.

**getEphemericalPort()**

Retrieves the (ephemeric) port number of the connecting socket, see above for details and intent.

**reject()**

Marks the incoming message as rejected. No further processing (for example sending reply) will be possible.

**reply()****reply(parameters)**

Sends the reply, together with the (possibly empty) list of parameters (the first form replies without any parameters and is equivalent to the empty list) to the remote object.

**forward(domainname, objectname, msgname)**

**forward(domainname, objectname, msgname, params)**

**forwardAddr(address, port, level, objectname, msgname)**

**forwardAddr(address, port, level, objectname, msgname, params)**

Forwards the incoming message to the object designated by **objectname** to the domain registered as **domainname**. The forwarded message has the same *from* address, so any action performed on the remote side (like rejecting or replying) is visible to the originating object (the one, that has sent the message in the first place). There is no limit on the length of this forwarding chain. The application is allowed to completely change the name of the message as well as its parameter list. This strategy allows to perform some preprocessing before actually forwarding the message.

The “Addr” versions allow to forward the message to the destination with explicitly given address, bypassing the *Agent*’s internal address book.

### 3.4 General API

The functionality of the *YAMI* message broker is encapsulated by the **Agent** class.

The following methods are available in the **Agent** class:

**Agent()**

**Agent(port)**

**Agent(port, policies)**

The constructor.

The listening socket will be created on the given port for messages coming from remote sites.

If no *Policies* object is provided, the default values are used.

Using 0 as the port number or calling the first form of this constructor has the effect of starting the *Agent* object with the listening socket on the port assigned by the operating system.

**domainRegister(name, address, port, level)**

**domainRegisterEx(name, address, port, level, options)**

Registers a new domain in the Agent. The **name** is an arbitrary name (visible only to the local Agent), the **address** is the network address (either in the *xxx.xxx.xxx.xxx* form or as a human readable *comp.company.com*) of the remote domain. The port and level of the remote domain should comply to the actual values. If the domain with a given name is already registered, the exception will be thrown.

The **options** parameter is a const value: either **eNone** (for simplex connections, this is the default when the first form is used) or **eFixedDuplex**

(for duplex connections). Note that this parameter is ignored when the message is *forwarded* to the given domain – forwards are always performed in the simplex mode.

#### **domainUnregister(name)**

Unregisters the given domain.

#### **objectRegister(name)**

Registers new object in the local Agent. The registration includes creation of the incoming message queue, which means that incoming messages are accepted and queued immediately after this function completes. The **name** is arbitrary name and will be used by the remote objects to send messages to this object. If the special 'YAMI\_ANY\_OBJECT' name is used, it denotes the default object that retrieves all messages sent to otherwise unknown objects.

#### **objectUnregister(name)**

Unregisters the object. This means, that all waiting messages for this object (and all new that will be received) will be discarded with the *unknown object* reply message.

#### **send(domainname, objectname, messagename)**

#### **send(domainname, objectname, messagename, params)**

#### **sendAddr(address, port, level, objectname, messagename)**

#### **sendAddr(address, port, level, objectname, messagename, params)**

#### **sendAddrEx(addr, port, level, objname, msgname, options)**

#### **sendAddrEx(addr, port, level, objname, msgname, options, pars)**

The **domainname** should be a name of the already registered domain or a special name, 'YAMI\_THIS\_DOMAIN', denoting the local domain (then, the short-cut routing will be performed – this is only a hint to the Agent, because the Agent can figure it out by itself, if the domain is registered with the same address and port as the local Agent). The **objectname** should be a name of the object registered at the remote Agent (or the remote Agent should have default object set up). The **messagename** is a name of the new message and will be visible to the remote object. The **params** should be a (possibly empty) list of parameters that will be sent together with the message. Depending on the levels of the local and the remote Agents, there are restrictions on the types of parameters that can be put in this list. This method returns a new object of **Message** class (and it is the only way to create the objects of that class), which can be used for later reference, for example to retrieve the status of the message and its return

parameters, if any. The “Addr” versions allow to send the message to the destination with explicitly given address, bypassing the *Agent*’s internal address book.

The `options` parameter is a const value: either `eNone` (for simplex connections, this is the default when the first form is used) or `eFixedDuplex` (for duplex connections).

**sendOneWay(domainname, objectname, msgname)**

**sendOneWay(domainname, objectname, msgname, params)**

**sendOneWayAddr(addr, port, level, objname, msgname)**

**sendOneWayAddr(addr, port, level, objname, msgname, params)**

**sendOneWayAddrEx(addr, port, level, objname, msgname, options)**

**sendOneWayAddrEx(addr, port, lvl, objn, msgn, options, params)**

As the `send` method, but the `Message` object is not created. It is not possible to retrieve any information about the message later. Note: it does not make any sense for the remote site to reply or to reject the message sent with this method. The “Addr” versions allow to send the message to the destination with explicitly given address, bypassing the *Agent*’s internal address book.

**getIncoming(objectname)**

**getIncoming(objectname, wait)**

The `name` should be a name of already registered object. If there is no message waiting for a given object, then `None` is returned. If the second form is used, the `wait` parameter (if non-zero) tells the *Agent* to freeze the calling thread until some message arrives.

**getIncomingCount(objectname)**

This function gets the current length of the incoming message queue associated with the given object.

**getLocalAddress()**

Returns the local address of the *Agent* object (it does not need to have a listening socket) in the standard form `xxx.xxx.xxx.xxx`. It can be useful for objects that need to pass their location to remote objects.

**getLocalPort()**

Retrieves the port number that is actually used by the listening socket.

If the *Agent* was created without the listening socket, this function throws an exception.

It can be useful for objects that need to pass their location to remote objects.

#### **getImplLimit(limitname)**

Retrieves the implementation limit, as defined in the `yamilimits.h` header file.

### **3.5 Additional functions**

#### **easySend(address, port, level, objname, msgname)**

#### **easySend(address, port, level, objname, msgname, parameters)**

Creates an *ad-hoc* lightweight *Agent* and sends a *one-way* message to the given domain and object. The *Agent* created for sending the message is immediately destroyed, so this function is provided for convenience where messages are sent rarely or where the performance of communication infrastructure is of no concern. For purposes where many messages are sent to remote objects, it is preferred to create full *Agent* object once and use it for sending many messages.