amphora Documentation

Release 0.1

Vladimir Lagunov

May 29, 2013

CONTENTS

Amphora is a Python library for asynchronous remote procedure executing via AMQP protocol. Amphora actively uses gevent and requires for monkey-patching Python sockets with gevent.

Amphora works with Python 2.6 and 2.7 and was tested with RabbitMQ 2.7, 3.0 and 3.1.

Key features of Amphora:

- Reliability. You will never lose any request or response accidentally even if you suddenly lose network connection for a while.
- Designed for using with gevent and greenlets (similar with Erlang actors).
- RPC server can subscribe and unsubscribe from different request queues on the fly. Limiting subscriptions by maximum requests and/or by timeout.

Contents:

CHAPTER

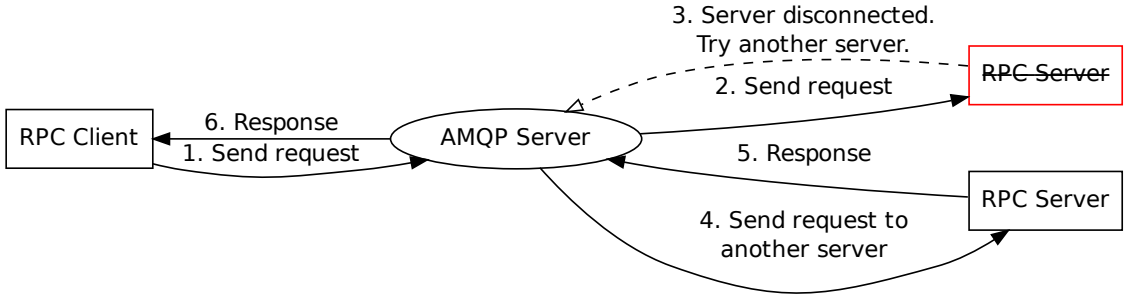
ONE

ABOUT RELIABILITY

Amphora allows you to shut down some components of your system for short maintenance. The system holds its current state and waits until all components start to work again. But it is not possible to save all messages in every fantastic or real case and there are some restrictions about.

1.1 Shutting down the server

If you want to immediately shut down one server then all requests will be requeued by AMQP server and sent to another server.



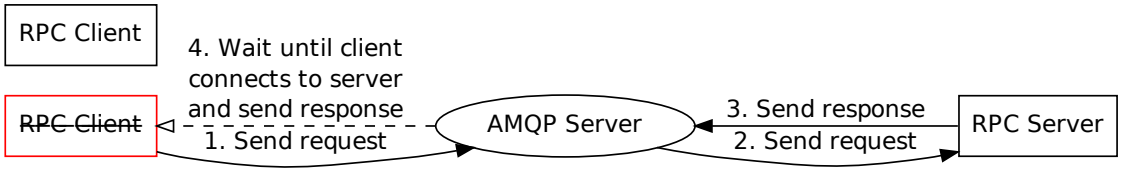
When there is only one server, client will wait until the server start up.

Note: If you terminate the server when it handles requests, then those request will be executed again when server will start up. Be careful with side effects of request handlers.

1.2 Shutting down the client

When your client loses network connection with AMQP server, client will receive responses when network fixes. But if you manually shut down your client application, your responses will be lost. It is like to terminate application when it stores records in database (without transactions): records will be created in the database but client will not know about results of this operation because client is dead.

Keep in mind that remote function call result can be delivered only to client that sent those request.



1.3 Shutting down the AMQP server

Main feature is that you will not lose your requests and responses even if you send *kill -9* to AMQP server. While AMQP server is down, both client and server holds outgoing messages. Client and server will authomatically reconnects to AMQP server when it starts.

> **Warning:** When you restart AMQP server, you will not lose requests but each pending request can be executed twice.

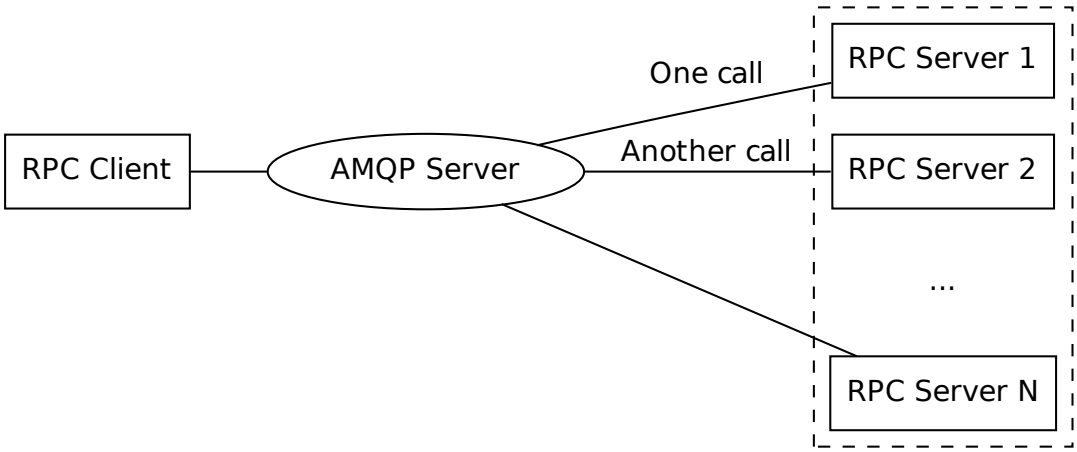
CHAPTER

TWO

TUTORIAL

2.1 Simple usage

Simplest case - distributed RPC. Client sends request that will be handled by one of several servers.



Everything you need is to create a server instance and create a client instance. When amphora connects to AMQP server it creates all needed queues and exchanges.

Let's start with server.

Amphora allows to have multiple RPC services in one AMQP virtual host. You should specify namespace name for each your service. In our example we will do simple arithmetic operations at RPC servers. So name our namespace *"math"*.

```
#!/usr/bin/env python
from amphora import AmqpRpcServer
server = AmqpRpcServer('math')
server.serve(nowait=False)
```

Now you can visit RabbitMQ management web-interface and you'll see that amphora created exchanges *rpc_math_request* and *rpc_math_response*. Also was created queue *rpc_math_request* and bound to exchange with the same name.

Add some functions to RPC server:

```
#!/usr/bin/env python
from math import sqrt
from amphora import AmqpRpcServer
server = AmqpRpcServer('math')

@server.add_function
def sum_numbers(*args):
    return sum(args)

@server.add_function
def hypotenuse(cathetus1, cathetus2):
    return sqrt(cathetus1 ** 2 + cathetus2 ** 2)

server.serve(nowait=False)
```

That's all, our server is ready! You can start as many server processes as you want.

Now we will write simple client. We need to specify one namespace both in server and client.

```
#!/usr/bin/env python
from amphora import AmqpRpcClient
client = AmqpRpcClient('math')

import gevent; gevent.sleep(1)  # Let the amphora create all queues
```

Look again at RabbitMQ management. There was created another queue that looks like `rpc_math_somemagicstring_response`. It bound with `rpc_math_response` with routing key `somemagicstring`.

Now we can do requests:

```
#!/usr/bin/env python
from amphora import AmqpRpcClient
client = AmqpRpcClient('math')

import gevent; gevent.sleep(1)  # Let the amphora create all queues

print client.call.sum_numbers(1, 2, 3, 4)  # Prints 10
print client.call.hypotenuse(3, 4)  # Prints 5.0
```

Amphora allows you to handle remote exceptions:

```
#!/usr/bin/env python
from amphora import AmqpRpcClient
client = AmqpRpcClient('math')

import gevent; gevent.sleep(1)

print client.call.hypotenuse("foo", "bar")
# Traceback:
#   ...
# RemoteException: TypeError: unsupported operand type(s) for ** or pow(): 'str' and 'int'
```

2.2 Asynchronous calls

Often you don't want to wait until remote call finished. Maybe you want to make parallel requests or you don't bother about remote call result.

Lets create server function that can work several seconds:

```
from urllib2 import build_opener, URLError
from amphora import AmqpRpcServer

server = AmqpRpcServer('example')

@server.add_function
def check_is_down(site):
    opener = build_opener()
    try:
        opener.open(site)
    except URLError:
        return "Site {0} is down".format(site)
    return "Site {0} is up".format(site)

server.serve(nowait=False)
```

Client implementation:

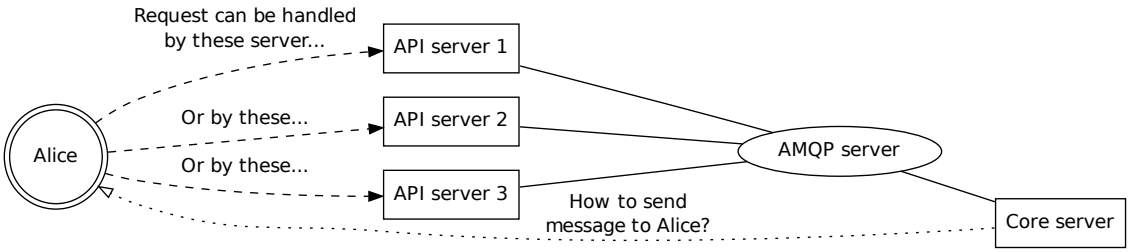
```
from amphora import AmqpRpcClient

client = AmqpRpcClient('example')

# All of them executed asynchronously
results = [client.defer(site) for site in (
    'http://google.com', 'http://yandex.com', 'http://example.com',
    'http://thereisnosuchsitename.com')]

# Now you can do some stuff
do_another_work()

# Time to check results
for result in results:
    print result.get()
```

2.3 Different request queues

More complex example. You have distributed web service that observes multiple users. One part of service (*"API"*) communicates with user via WebSockets or long polling. Another part of service (*"Core"*) contains all busines logics but does not interact with user directly.

"Core" wants to immediately show the user Alice some message or wants from user to fill form.

You have many *"API"* servers and each user can be observed only by one *"API"* server at time. *"Core"* does not know which of *"API"* servers interacts with user Alice and which interacts with Bob.



AmqpRpcServer can create request queues on demand and remove them when they becomes needless.

"API" at startup creates RPC server. When user requests new messages then *"API"* executes method `amphora.AmqpRpcServer.receive_from_queue()` and waits until some event occurs.

Meanwhile when RPC server get request to send message to user it triggers event and user request wakes up.

```
# views.py
import gevent
from handlers import server, events

def user_get_message(request):
    username = request.user.username
    try:
        server.receive_from_queue(
            username, max_calls=1, timeout=10, block=True)
    except gevent.Timeout:
        return {'ok': True, 'message': None}
    message = events.pop(username)
    return {'ok': True, 'message': message}


# handlers.py
from amphora import AmqpRpcServer

server = AmqpRpcServer("messages")
events = {}

@server.add_function
def user_show_message(username, message):
    events[username] = message

server.serve()
```

If queue does not exist, `amphora.AmqpRpcServer.receive_from_queue()` will silently wait until it creates.

Before sending messages client should create request queue for user:

```
client.create_new_request_queue("Alice")
```

When *"Core"* wants to send message to user it just calls:

```
client.defer(routing_key="Alice").show_message(
     "Alice", "Hello, Alice!")
```

Or you can use `amphora.AmqpRpcClient.tune_function()` that will determine routing key:

```
@client.tune_function('show_message')
def message_tune_function(args, kwargs):
    if args:
        return args[0]
    return kwargs.get('username')

client.defer.show_message("Alice", "Goodbye, Alice!") # Routing key "Alice"
client.call.show_message(username="Bob", message="Goodbye, Bob!")  # Routing key "Bob"
```

2.4 Broadcasting requests

If you want to execute each request in all server instances, then use `AmqpRpcFanoutServer` and `AmqpRpcFanoutClient`.

Handler function in fanout can ignore request by raising `amphora.IgnoreRequest`.

Current implementation can't get all results from each server. If you execute `amphora.AmqpRpcFanoutClient.call()` and each server will return the result then first delivered response will be returned and other responses will be lost.

CHAPTER

THREE

API

3.1 Server API

class amphora.**AmqpRpcServer**(*service_name*, *serializer=<class 'amphora.serializer.RpcJsonSerializer'>*, ***kwargs*)

Server class for AMQP RPC.

Parameters

- **service_name** (*str*) – Namespace of your RPC service.
- **serializer** (AbstractRpcSerializer) – Class for generating AMQP messages. JSON messages generated by default.
- **amqp_host** (*str*) – IP address or hostname of AMQP server. By default 127.0.0.1.
- **amqp_port** (*int*) – TCP port of AMQP server. By default 5672.
- **amqp_user** (*str*) – Username for authentication in AMQP. By default guest.
- **amqp_password** (*str*) – Password for authentication in AMQP. By default guest.
- **amqp_vhost** – Virtual host for authentication in AMQP. By default /.

add_function(*func*, *name=None*)

Add function for handling remote call. Can be used as common function or as decorator.

Parameters

- **func** (*callable*) – Remote call handler.
- **name** (*str*) – Remote call handler name. If omited then call names the same as handler name.

Example:

```
# math_server.py
from gevent import monkey; monkey.patch_socket()
from amphora import AmqpRpcServer

server = AmqpRpcServer('simplemath')

def add(x, y):
    return x + y
server.add_function(add)

def substract(x, y):
    return x - y
```

```
server.add_function(substract, name="sub")

@server.add_function
def multiply(x, y):
    return x * y

@server.add_function('div')
def divide(x, y):
    return x / y

server.serve(nowait=False)


# math_client.py
from gevent import monkey; monkey.patch_socket()
from amphora import AmqpRpcClient

client = AmqpRpcClient('simplemath')
# TODO: must work without __import__('gevent').sleep(1)
print client.call.add(2, 3)  # 5
print client.call.sub(10, 3)  # 7
print client.call.multiply(5, 5)  # 25
print client.call.div(64, 16)  # 4
```

Added function can raise `amphora.IgnoreRequest` for re-sending this request to another server.

receive_from_queue(*queue*, *max_calls=None*, *timeout=None*, *block=True*)

Receive new calls from specified queue.

`amphora.AmqpRpcServer` authomatically starts receiving from queue `rpc_{service_name}_request`, but you may enable receiving calls from several queues.

Parameters

- **queue** (*str*) – AMQP queue name with requests.
- **max_calls** (*int*) – Maximum call count. By default unlimited.
- **timeout** (*float*) – Stop receiving from queue after `timeout` seconds.
- **block** (*bool*) – Block current greenlet until maximum calls reached or timed out.

Raises

- **gevent.Timeout** – When timed out and maximum calls was not reached.
- **ValueError** – If queue not found.

If you set `max_calls` then *AmqpRpcServer* handles only specified calls. After reaching maximum *AmqpRpcServer* cancel consuming from queue but does not close channel until all handling messages acked or rejected. Same rules applies for `timeout`. You can specify both parameters at once.

When requests delivered too frequently, AMQP server can send some request messages just after *basic_cancel* call. These messages will be rejected with requeuing. In order to avoid infinite resending messages that can cause denial of service, AmqpRpcServer can reject messages with one second delay.

serve(*nowait=True*)

Connect to AMQP and start working.

Parameters **nowait** (*bool*) – Should current greenlet being blocked until server stop? By default don't block.

prepare_stop()
Reject any new requests. Use this method in couple with `stop()` when you want to do "warm shutdown".

stop()
Stop instance and disconnect from AMQP server. All unsent messages will be hold until you'll call `serve()` again.

stop_publisher()
Stop publishing messages. Consuming continues working (if it was not stopped earlier).

stop_consumer()
Stop consuming messages. Publishing continues working (if it was not stopped earlier).

class amphora.**AmqpRpcFanoutServer**(*service_name*, *serializer=<class 'amphora.serializer.RpcJsonSerializer'>*, ***kwargs*)
Fanout server class for AMQP RPC.

Fanout exchanges in AMQP used for broadcasting. If you execute remote function via `AmqpRpcFanoutClient` then request will be handled by every connected `AmqpRpcFanoutServer`.

One difference from `AmqpRpcServer` it that added functions can raise `amphora.IgnoreRequest`. When function raises it, server sends "ack" to request message but does not send any result or exception back to client.

3.2 Client API

class amphora.**AmqpRpcClient**(*service_name*, *uuid=None*, *serializer=<class 'amphora.serializer.RpcJsonSerializer'>*, *timeout=60*, *autostart=True*, ***kwargs*)
Client class for AMQP RPC.

Parameters

- **service_name** (*str*) – Namespace of your RPC service.
- **uuid** (*str*) – Unique identifier for current client instance. If omited then will be generated random id.
- **serializer** (`AbstractRpcSerializer`) – Class for generating AMQP messages. JSON messages generated by default.
- **timeout** (*float*) – Default timeout for `call`
- **amqp_host** (*str*) – IP address or hostname of AMQP server. By default `127.0.0.1`.
- **amqp_port** (*int*) – TCP port of AMQP server. By default 5672.
- **amqp_user** (*str*) – Username for authentication in AMQP. By default `guest`.
- **amqp_password** (*str*) – Password for authentication in AMQP. By default `guest`.
- **amqp_vhost** – Virtual host for authentication in AMQP. By default `/`.

defer
Call remote procedure without blocking current greenlet.

Parameters

- **function_name** (*str*) – You can specify function name directly as string.
- **routing_key** (*str*) – Routing key for remote call.
- **wait_publish** (*bool*) – Wait until request message published.

Returns Helper object for calling remote procedures.

Return type amphora.PrettyCaller

When helper object called, it returns gevent.event.AsyncResult instance. When remote function completes returned value stores into *AsyncResult*.

You can specify routing key for this call in two ways:

- By specifying argument *routing_key*.
- Usung method tune_function().

Examples:

```
client = amphora.AmqpRpcClient("example")

# Remote call of function with name "send_email"
async_result = client.defer.send_email('root@example.com', "Hello!")
print async_result.get()

# Remote call of function with name "show_message"
# with custom routing key
client.create_new_request_queue("user12345")
client.defer(routing_key="user12345").show_message(
     "Hello, user12345!")

# Remote call of functions with names "send_message.sms"
client.defer.send_message.sms("+12345678901", "Message")

# You can use helper objects like any normal python objects.
# This code calls "send_message.email.text"
# and "send_message.email.html"
email_sender = client.defer.send_message.email
email_sender.text("root@example.com", "Message")
email_sender.html("user@example.com", "Message")

# If you don't want to generate function name with helper
for format in ("html", "text"):
    client.defer(function_name="send_message.email." + format)(
        "root@example.com", "Hello!")
```

call

Call remote procedure with blocking current greenlet.

Parameters

- **function_name** (*str*) – You can specify function name directly as string.
- **timeout** (*float*) – Timeout for waiting for result of remote call. If not specified then used timeout specified in constructor.
- **routing_key** (*str*) – Routing key for remote call.

Raises

- **amphora.RemoteException** – When remote function raises exception.
- **amphora.WrongRequest** – When server can't parse request. For example when you trying to call function not registered in server.
- **amphora.NoResult** – When call timed out.

Returns Helper object for calling remote procedures.

Return type amphora.PrettyCaller

When helper object called, it waits until remote function completes and returns result of remote function.

Look for examples and explanations into `defer` documentation.

`create_new_request_queue`(*queue*, *routing_key=None*, *nowait=False*)
Asynchronously create new queue and bind it to request exchange.

Parameters

- **queue** (*str*) – Queue label that will be used for generating queue name.
- **routing_key** (*str*) – Routing key for binding to request exchange.
- **nowait** (*bool*) – Should block current greenlet until queue created and bound? If True then don't block.

If routing key not specified then routing key will be the same as queue name.

Then name of the queue passes to template `rpc_{queue}_request`.

For example, if you call `create_new_request_queue('test')` then creates queue `rpc_test_request` and binds to request exchange via routing key `test`.

`remove_request_queue`(*queue*, *nowait=False*)
Asynchronously removes queue.

Parameters

- **queue** (*str*) – Queue label that will be used for generating queue name.
- **nowait** (*bool*) – Should block current greenlet until queue deleted? If True then don't block.

The name of the queue generates like in `create_new_request_queue()`.

> **Warning:** If you call `remove_request_queue()` and then immediately call `create_new_request_queue()` then queue will be deleted but may not be created.

`tune_function`(*func_name*)
Set the function that will calculate queue and routing key for specified function by passed args.

Your tuning function should return dict with two keys: `"queue"` and `"routing_key"`. (`queue` is deprecated).

Example:

```
from amphora import AmqpRpcClient

def calculate(args, kwargs):  # Note, no * or **
    user_id = str(kwargs['user_id'])
    return {'queue': user_id, 'routing_key': user_id}

client = AmqpRpcClient('test')
client.tune_function('show_message')(calculate)

# Will be called with routing key "12345"
client.defer.show_message("Hello!", user_id=12345)
```

`serve`(*nowait=True*)
Connect to AMQP and start working.

Parameters **nowait** (*bool*) – Should current greenlet being blocked until server stop? By default don't block.

stop()
Stop instance and disconnect from AMQP server. All unsent messages will be hold until you'll call `serve()` again.

stop_publisher()
Stop publishing messages. Consuming continues working (if it was not stopped earlier).

stop_consumer()
Stop consuming messages. Publishing continues working (if it was not stopped earlier).

class amphora.**AmqpRpcFanoutClient**(*service_name*, *uuid=None*, *serializer=<class 'amphora.serializer.RpcJsonSerializer'>*, *timeout=60*, *autostart=True*, ***kwargs*)
Client for `AmqpRpcFanoutServer`. API is the same as for `AmqpRpcClient`.

3.3 Exceptions

exception amphora.**NoResult**
Raises by `amphora.AmqpRpcClient.call` when request is timed out.

exception amphora.**WrongRequest**
Raises by `amphora.AmqpRpcClient.call` or by `gevent.AsyncResult.get()` returned by `amphora.AmqpRpcClient.defer` when server can't parse request. For example, when you try to call function not registered in server.

exception amphora.**WrongResult**
Raises by `amphora.AmqpRpcClient.call` or by `gevent.AsyncResult.get()` returned by `amphora.AmqpRpcClient.defer` when server can't parse response. Normally you will never get this exception.

exception amphora.**RemoteException**(*error_type*, *args*)
Raises by `amphora.AmqpRpcClient.call` or by `gevent.AsyncResult.get()` returned by `amphora.AmqpRpcClient.defer` when remote function raises exception.

Parameters

- **error_type** (*str*) – Class name of remote exception.
- **args** (*tuple*) – Arguments passed to remote exception.

Example:

```
# server.py
@server.add_function
def divide(x, y):
    if x < 0:
        raise ValueError('x', x)
    elif y < 0:
        raise ValueError('y', y)
    else:
        return x / y

# client.py
from amphora import RemoteException
client.call.divide(6, 3)  # Prints "2"

try:
   client.call.divide(-5, 5)
except RemoteException as exc:
```

```
        print exc.error_type  # Prints "ValueError"
        print exc.args  # Prints '("x", -5)'

    try:
        client.call.divide(10, 0)
    except RemoteException as exc:
        print exc.error_type  # Prints "ZeroDivisionError"
```

exception amphora.**IgnoreRequest**
Reject request when raised in AmqpRpcServer or ignore it when raised in AmqpRpcFanoutServer.

CHAPTER

FOUR

INDICES AND TABLES

- *genindex*