snakeMQ Documentation

Release 1.6

David Siroky

Nov 26, 2019

Contents

Generated on Nov 26, 2019.

Contents:

CHAPTER 1

Changelog

1.6 (2018-07-03)

- fix missing method call flow

1.5 (2018-06-11)

- fix signal debug output

1.4 (2017-10-07)

- fix pip build

1.3 (2017-06-07)

- fix memory leak in Messaging

1.2 (2014-03-28)

- Python support versions 3.3, 3.4
- Messaging.on_message_drop() callback
- listener - allow choosing any available port number
- logging to any file instead of just stdout
- fix shebangs

1.1 (2013-04-30)

- simple bandwidth throttle
- maintain call order in callbacks
- RPC - req_id logging on server side
- more SSL examples
- persistent queue examples
- fix unittests missing asserts
- fix failed SSL handshake cleanup

- fix PyPy sqlite storage VACUUM
- fix handle_close() on SSL handshake failure
- fix SocketPoll socket to file descriptor conversion
- fix TestBell.test_bell_pipe nonblocking failure

1.0 (2012-05-18)

- fixed documentation

0.5.10 (2011-12-20)

- messaging
  - new on_message_sent() callback
  - fix on_packet_sent() callback
- fix packeter queuing
- RPC
  - API change - as_signal() moved to RemoteMethod
  - timeout for regular calls
  - custom pickler
  - exception traceback as a string

0.5.9 (2011-09-12)

- winpoll.py renamed to poll.py and substitutes epoll() when missing
- fix failing on a short connection which is closed by OS immediately after creation
- fix buffer/memory python compatibility issues
- fix ttl=None/infinite TTL

0.5.8 (2011-08-19)

- keep-alive - ping/pong to test if the peer connection is alive
- fix MSW poll bell nonblocking issue
- fix message without previous identification
- fix link connect ENETUNREACH

0.5.7 (2011-07-29)

- fix multiple connecting peers with the same identification

0.5.6 (2011-07-01)

- SQLAlchemy storage
- MongoDB queue ordering fix

0.5.5 (2011-06-09)

- fix MS Windows SSL
- messaging example with SSL
- fix Python 3 syntax
- updated documentation, RPC

- RPC module moved from package snakemq.contrib to snakemq
- message TTL can be set to None (infinity)
- Link.send() does not return amount of sent bytes, moved to Link.on_ready_to_send
- removed snakemq.init()
- internal refactorizations

0.5.4 (2011-05-22)

- SSL
- MongoDB storage

0.5.3 (2011-05-18

- Python3 adaptation

0.5.2 (2011-05-15)

- all callbacks can be bound to more then 1 callable
- more examples

0.5.1 (2011-05-12)

- fix poll and bell for MS Windows
- fix RPC example

0.5 (2011-05-04)

- initial release

CHAPTER 2

Tutorial

2.1 Terminology

listener Somebody must listen for incoming connections. Similar to TCP server.

connector Similar to TCP client.

peer Program running a snakeMQ stack.

2.2 Connection

The stack can have multiple connectors and listeners. Once the connection is made between a connector and a listener then there is internally no difference on both sides. You can send and receive on both sides.

It does not matter who is listening and who is connecting (but obviously there must be a pair connector-listener). Who will be who depends on the network topology and the application design. E.g. if peer A is somewhere on the internet and peer B is behind NAT then A must have a listener and B must have a connector.

2.3 Simple messaging peers

Import modules:

```
import snakemq.link
import snakemq.packeter
import snakemq.messaging
import snakemq.message
```

Build stack:

```
my_link = snakemq.link.Link()
my_packeter = snakemq.packeter.Packeter(my_link)
my_messaging = snakemq.messaging.Messaging(MY_IDENT, "", my_packeter)
```

where `MY_IDENT` is the peer identifier e.g. `"bob"` or `"alice"`. Second parameter is *domain* used for routing - currently unused.

Since the link is symmetrical it does not matter who is connecting and who is listening. Every link can have arbitrary count of listeners and connectors:

- Bob:

```
my_link.add_listener(("", 4000))  # listen on all interfaces and on port 4000
my_link.add_connector(("localhost", 4001))
```

- Alice:

```
my_link.add_connector(("localhost", 4000))
my_link.add_connector(("localhost", 4001))
```

- Jack:

```
my_link.add_connector(("localhost", 4000))
my_link.add_listener(("", 4001))
```

After that connections are defined from everybody to each other.

Run link loop (it drives the whole stack):

```
my_link.loop()
```

Bob wants to send a message to Alice:

```
# drop after 600 seconds if the message can't be delivered
message = snakemq.message.Message(b"hello", ttl=600)
my_messaging.send_message("alice", message)
```

The message is inserted into a queue and it will be delivered as soon as possible. The message is also transient so it will be lost if the process quits.

Note: Sending message to itself (e.g. `bob_messaging.send_message("bob", message)`) will not work. It will be queued but it will be eventually delivered only to another peer with the same identifier (peers with identical identifier are not desired).

Receiving callback:

```
def on_recv(conn, ident, message):
    print(ident, message)

my_messaging.on_message_recv.add(on_recv)
```

2.4 Persistent queues

If you want the messaging system to persist undelivered messages (messages will survive process shutdowns, they will be loaded on start) then use a storage and mark messages as persistent:

```
from snakemq.storage.sqlite import SqliteQueuesStorage
from snakemq.message import FLAG_PERSISTENT

storage = SqliteQueuesStorage("storage.db")
my_messaging = snakemq.messaging.Messaging(MY_IDENT, "", my_packeter, storage)

message = snakemq.message.Message(b"hello", ttl=600, flags=FLAG_PERSISTENT)
```

SnakeMQ supports various storage types but SQLite is recommended for its speed and availability as a default library module.

Note: Persistent are only **outgoing** messages. Once it is delivered it is up to the other side to make sure that the message will not be lost.

It does not matter if the sending side is a connector or a listener.

2.5 Logging

If you want to see what is going on inside:

```
import logging
import snakemq

snakemq.init_logging()
logger = logging.getLogger("snakemq")
logger.setLevel(logging.DEBUG)
```

2.6 SSL context

To make the link secure add *SSLConfig*:

```
import ssl

sslcfg = snakemq.link.SSLConfig("testpeer.key", "testpeer.crt",
                                ca_certs="testroot.crt",
                                cert_reqs=ssl.CERT_REQUIRED)

# peer A
my_link.add_listener(("", 4000), ssl_config=sslcfg)

# peer B
my_link.add_connector(("localhost", 4000), ssl_config=sslcfg)
```

2.6.1 Get peer's certificate

To get the peer's certificate use method *getpeercert()*. For example your link's `on_connect()` might look like:

```
def on_connect(conn):
    sock = slink.get_socket_by_conn(conn)
    print sock.getpeercert()
```

See `examples/ssl_*.py`.

2.7 Remote Procedure Call

SnakeMQ's RPC implementation has a huge advantage - you don't need to take care of connectivity/reconnections. Register your objects and call their methods whenever it is needed. Since the messaging is symmetrical then both peers can act as server and client at the same time.

Two kinds of calls:

- *Regular call with response* - calling will be blocking until the remote side connects and returns result. Remote exceptions can be propagated as well. If the connection is broken during the call then the client will attempt to perform the call again until it gets any result. This may lead to starvation on the client side (TODO).
- *Signal call without response* - calling is not blocking and returns `None`. You can set TTL of the signal.

Call kinds can't be combined. If a method is marked as a signal then it can be called only as a signal.

Build stack for messaging and add:

```
import snakemq.rpc

# following class is needed to route messages to RPC
rh = snakemq.messaging.ReceiveHook(my_messaging)
```

Server:

```
class MyClass(object):
    def get_fo(self):
        return "fo value"

    @snakemq.rpc.as_signal  # mark method as a signal
    def mysignal(self):
        print("signal")

srpc = snakemq.rpc.RpcServer(rh)
srpc.register_object(MyClass(), "myinstance")
my_link.loop()
```

Client:

```
crpc = snakemq.rpc.RpcClient(rh)
proxy = crpc.get_proxy(REMOTE_IDENT, "myinstance")
proxy.mysignal.as_signal(10)  # 10 seconds TTL
my_link.loop()

# in a different thread:
proxy.mysignal()  # not blocking
proxy.get_fo()  # blocks until server responds
```

2.7.1 Exceptions

Propagation of remote exceptions is turned on by default. It can be disabled on the server side:

```
srpc.transfer_exceptions = False
```

If the exception is transfered and raised on the client side then it has local traceback. Remote traceback is stored in attribute `exception.__remote_traceback__`.

2.8 Bandwidth throttling

Very simple bandwidth throttling per connection. Place a throttle between link and packeter:

```
import snakemq.throttle

my_link = snakemq.link.Link()
my_throttle = snakemq.throttle.Throttle(my_link, 10000) # ~10 kB/s
my_packeter = snakemq.packeter.Packeter(my_throttle)
```

CHAPTER 3

API

3.1 Link

class snakemq.link.**SSLConfig**(*keyfile=None*, *certfile=None*, *cert_reqs=0*, *ssl_version=2*, *ca_certs=None*)

Container for SSL configuration.

__init__(*keyfile=None*, *certfile=None*, *cert_reqs=0*, *ssl_version=2*, *ca_certs=None*)

See ssl.wrap_socket

class snakemq.link.**LinkSocket**(*sock=None*, *ssl_config=None*, *remote_peer=None*)

getpeercert(*binary_form=False*)

See python documentation - ssl.SSLSocket.getpeercert()

Returns peer's SSL certificate if available or None

class snakemq.link.**Link**

Just a bare wire stream communication. Keeper of opened (TCP) connections. **Not thread-safe** but you can synchronize with the loop using *wakeup_poll()* and *on_loop_pass*.

add_connector(*address*, *reconnect_interval=None*, *ssl_config=None*)

This will not create an immediate connection. It just adds a connector to the pool.

Parameters

- **address** – remote address
- **reconnect_interval** – reconnect interval in seconds

Returns connector address (use it for deletion)

add_listener(*address*, *ssl_config=None*)

Adds listener to the pool. This method is not blocking. Run only once.

Returns listener address (use it for deletion)

cleanup()
Close all sockets and remove all connectors and listeners.

del_connector(*address*)
Delete connector.

del_listener(*address*)
Delete listener.

loop(*poll_timeout=0.2*, *count=None*, *runtime=None*)
Start the communication loop.

Parameters

- **poll_timeout** – in seconds, should be less then the minimal reconnect time
- **count** – count of poll events (not timeouts) or None
- **runtime** – max time of running loop in seconds (also depends on the poll timeout) or None

send(*conn_id*, *data*)
After calling *send* wait for *on_ready_to_send* before sending next data.

This operation is non-blocking, data might be lost if you close connection before proper delivery. Always wait for *on_ready_to_send* to have confirmation about successful send and information about amount of sent data.

Do not feed this method with large bulks of data in MS Windows. It sometimes blocks for a little time even in non-blocking mode.

Optimal data size is 16k-64k.

stop()
Interrupt the loop. It doesn't perform a cleanup.

wakeup_poll()
Thread-safe.

3.1.1 Callbacks

Link.**on_recv = None**
`func(conn_id, data)`

Link.**on_ready_to_send = None**
`func(conn_id, last_send_size)`, last send was successful

Link.**on_loop_pass = None**
`func()`, called after poll is processed

3.2 Packeter

class snakemq.packeter.**Packeter**(*link*)

__init__(*link*)

Parameters **link** – *Link*

send_packet(*conn_id*, *buf*)
Queue data to be sent over the link.

Returns packet id

3.2.1 Callbacks

```
Packeter.on_connect = None
    func(conn_id)

Packeter.on_disconnect = None
    func(conn_id)

Packeter.on_packet_send = None

Packeter.on_packet_recv = None
    func(conn_id, packet)
```

3.3 Messaging

class snakemq.messaging.**Messaging**(*identifier*, *domain*, *packeter*, *queues_storage=None*)

__init__(*identifier*, *domain*, *packeter*, *queues_storage=None*)

Parameters

- **identifier** – peer identifier
- **domain** – currently unused
- **packeter** – *Packeter*
- **queues_storage** – QueuesStorageBase

send_message(*ident*, *message*)
Thread safe.

Parameters

- **ident** – destination address
- **message** – *Message*

3.3.1 Callbacks

```
Messaging.on_connect = None
    func(conn_id, ident)

Messaging.on_disconnect = None
    func(conn_id, ident)

Messaging.on_message_recv = None
    func(conn_id, ident, message)

Messaging.on_message_sent = None
    func(conn_id, ident, message_uuid)
```

Messaging.**on_message_drop = None**
`func(ident, message_uuid)` Called when the message is dropped from the queue due to the TTL timeout.

3.3.2 Keep-alive

Messaging.**keepalive_interval = None**
time to ping, in seconds (None = no keepalive)

Messaging.**keepalive_wait = None**
wait for pong, in seconds

3.3.3 Message

class snakemq.message.**Message**(*data*, *ttl=0*, *flags=0*, *uuid=None*)

__init__(*data*, *ttl=0*, *flags=0*, *uuid=None*)

Parameters

- **data** – (bytes) payload
- **ttl** – messaging TTL in seconds (integer or float), None is infinity
- **flags** – combination of FLAG_*
- **uuid** – (bytes) unique message identifier (implicitly generated)

snakemq.message.**FLAG_PERSISTENT = 1**
store to a persistent storage

3.4 Miscellaneous

3.4.1 Logging

snakemq.**init_logging**(*stream=None*)
Initialize logging (to standard output by default).

Parameters **stream** – for logging.StreamHandler

Index

Symbols

A

C

D

F

G

I

K

L

M

O

P

S

W