pynng Documentation

Release 0.1.0

Cody Piersall

Jan 07, 2020

Contents

pynng is Python bindings to Nanomsg Next Generation (nng). It provides a nice Pythonic interface to the nng library. The goal is that pynng's interface feels natural enough to use that you don't think of it as a wrapper, while still exposing the power of the underlying library. It is installable with pip on all major platforms (Linux, Windows, macOS). It has first class support for Trio and `asyncio`, in addition to being able to be used synchronously.

nng is an implementation of the Scalability Protocols; it is the spiritual successor to ZeroMQ. There are a couple of distributed systems problems that the scalability protocols aim to solve:

1. There are a few communication patterns that are implemented over and over and over and over and over again. The wheel is continuously reinvented, but no implementations are compatible with each other.
2. Not only is the wheel continuosly reinvented, it is reinvented for every combination of *transport* and *protocol*. A *transport* is how data gets from one place to another; things like TCP/IP, HTTP, Unix sockets, carrier pigeons. A *protocol* is the way that both sides have agreed to communicate with each other (some protocols are ad-hoc, and some are more formal).

The scalability protocols are the basic tools you need to build a distributed system. The following **protocols** are available:

- **pair** - simple one-to-one communication. (*`Pair0`*, *`Pair1`*.)
- **request/response** - I ask, you answer. (*`Req0`*, *`Rep0`*)
- **pub/sub** - subscribers are notified of topics they are interested in. (*`Pub0`*, *`Sub0`*)
- **pipeline**, aka **push/pull** - load balancing. (*`Push0`*, *`Pull0`*)
- **survey** - query the state of multiple applications. (*`Surveyor0`*, *`Respondent0`*)
- **bus** - messages are sent to all connected sockets (*`Bus0`*)

The following **transports** are available:

- **inproc**: communication within a single process.
- **ipc**: communication across processes on a single machine.
- **tcp**: communication over networks via tcp.
- **ws**: communication over networks with websockets. (Probably only useful if one end is on a browser.)
- **tls+tcp**: Encrypted TLS communication over networks.
- **carrier pigeons**: communication via World War 1-style carrier pigeons. The latency is pretty high on this one.

These protocols are language-agnostic, and implementations exist for many languages.

This library is available under the MIT License and the source is available on GitHub.

If you need two processes to talk to each other—either locally or remotely—you should be using the scalability protocols. You never need to open another plain socket again.

Okay, that was a little hyperbolic. But give pynng a chance; you might like it.

CHAPTER 1

Installing pynng

On Linux, Windows, and macOS, a quick

```
pip3 install pynng
```

should do the trick. pynng works on Python 3.5+.

CHAPTER 2

Getting Started

2.1 Pynng's core functionality

At the heart of pynng is the *pynng.Socket*. It takes no positional arguments, and all keyword arguments are optional. It is the Python version of nng_socket.

2.1.1 The Socket

Note: You should never instantiate a *pynng.Socket* directly. Rather, you should instantiate one of the *subclasses*.

class pynng.**Socket**(*, *listen=None*, *dial=None*, ***kwargs*)

Open a socket with one of the scalability protocols. This should not be instantiated directly; instead, one of its subclasses should be used. There is one subclass per protocol. The available protocols are:

- *Pair0*
- *Pair1*
- *Req0* / *Rep0*
- *Pub0* / *Sub0*
- *Push0* / *Pull0*
- *Surveyor0* / *Respondent0*
- *Bus0*

The socket initializer receives no positional arguments. It accepts the following keyword arguments, with the same meaning as the *attributes* described below: `recv_timeout`, `send_timeout`, `recv_buffer_size`, `send_buffer_size`, `reconnect_time_min`, `reconnect_time_max`, and `name`

To talk to another socket, you have to either *dial()* its address, or *listen()* for connections. Then you can *send()* to send data to the remote sockets or *recv()* to receive data from the remote sockets. Asynchronous

versions are available as well, as *asend()* and *arecv()*. The supported event loops are asyncio and Trio. You must ensure that you close() the socket when you are finished with it. Sockets can also be used as a context manager; this is the preferred way to use them when possible.

Sockets have the following attributes. Generally, you should set these attributes before *listen()*-ing or *dial()*-ing, or by passing them in as keyword arguments when creating the *Socket*:

- **recv_timeout** (int): Receive timeout, in ms. If a socket takes longer than the specified time, raises a pynng.exceptions.Timeout. Corresponds to library option NNG_OPT_RECVTIMEO
- **send_timeout** (int): Send timeout, in ms. If the message cannot be queued in the specified time, raises a pynng.exceptions.Timeout. Corresponds to library option NNG_OPT_SENDTIMEO.
- **recv_max_size** (int): The largest size of a message to receive. Messages larger than this size will be silently dropped. A size of 0 indicates unlimited size. The default size is 1 MB.
- **recv_buffer_size** (int): The number of messages that the socket will buffer on receive. Corresponds to NNG_OPT_RECVBUF.
- **send_buffer_size** (int): The number of messages that the socket will buffer on send. Corresponds to NNG_OPT_SENDBUF.
- **name** (str): The socket name. Corresponds to NNG_OPT_SOCKNAME. This is useful for debugging purposes.
- **raw** (bool): A boolean, indicating whether the socket is raw or cooked. Returns True if the socket is raw, else False. This property is read-only. Corresponds to library option NNG_OPT_RAW. For more information see nng's documentation. Note that currently, pynng does not support raw mode sockets, but we intend to in the future:
- **protocol** (int): Read-only option which returns the 16-bit number of the socket's protocol.
- **protocol_name** (str): Read-only option which returns the name of the socket's protocol.
- **peer** (int): Returns the peer protocol id for the socket.
- **local_address**: The SockAddr representing the local address. Corresponds to NNG_OPT_LOCADDR.
- **reconnect_time_min** (int): The minimum time to wait before attempting reconnects, in ms. Corresponds to NNG_OPT_RECONNMINT. This can also be overridden on the dialers.
- **reconnect_time_max** (int): The maximum time to wait before attempting reconnects, in ms. Corresponds to NNG_OPT_RECONNMAXT. If this is non-zero, then the time between successive connection attempts will start at the value of reconnect_time_min, and grow exponentially, until it reaches this value. This option can be set on the socket, or on the dialers associated with the socket.
- **recv_fd** (int): The receive file descriptor associated with the socket. This is suitable to be passed into poll functions like select.poll() or select.select(). That is the only thing this file descriptor is good for; do not attempt to read from or write to it. The file descriptor will be marked as **readable** whenever it can receive data without blocking. Corresponds to NNG_OPT_RECVFD.
- **send_fd** (int): The sending file descriptor associated with the socket. This is suitable to be passed into poll functions like select.poll() or select.select(). That is the only thing this file descriptor is good for; do not attempt to read from or write to it. The file descriptor will be marked as **readable** whenever it can send data without blocking. Corresponds to NNG_OPT_SENDFD.

> **Note:** When used in `select.poll()` or `select.select()`, `recv_fd` and `send_fd` are both marked as **readable** when they can receive or send data without blocking. So the upshot is that for `select.select()` they should be passed in as the *rlist* and for `select.poll.register()` the *eventmask* should be `POLLIN`.

- **tls_config** (*TLSConfig*): The TLS configuration for this socket. This option is only valid if the socket is using the TLS transport. See *TLSConfig* for information about the TLS configuration. Corresponds to `NNG_OPT_TLS_CONFIG`. This option is write-only.

await arecv()
The asynchronous version of *recv()*

await arecv_msg()
Asynchronously receive the *Message* `msg` on the socket.

await asend(*data*)
Asynchronous version of *send()*.

dial(*address*, *, *block=None*)
Dial the specified address.

Parameters

- **address** – The address to dial.
- **block** – Whether to block or not. There are three possible values this can take:
  1. If `True`, a blocking dial is attempted. If it fails for any reason, the dial fails and an exception is raised.
  2. If `False`, a non-blocking dial is started. The dial is retried periodically in the background until it is successful.
  3. (**Default behavior**): If `None`, a blocking dial is first attempted. If it fails an exception is logged (using the Python logging module), then a non-blocking dial is done.

listen(*address*, *flags=0*)
Listen at specified address.

`listener` and `flags` usually do not need to be given.

new_context()
Return a new *Context* for this socket.

recv(*block=True*)
Receive data on the socket. If the request times out the exception `pynng.Timeout` is raised. If the socket cannot perform that operation (e.g., a *Pub0*, which can only *send()*), the exception `pynng.NotSupported` is raised.

Parameters **block** – If block is True (the default), the function will not return until the operation is completed or times out. If block is False, the function will return data immediately. If no data is ready on the socket, the function will raise `pynng.TryAgain`.

recv_msg(*block=True*)
Receive a *Message* on the socket.

send(*data*)
Sends `data` (either `bytes` or `bytearray`) on socket.

Feel free to peruse the examples online, or ask in the gitter channel.

Available Protocols

class pynng.**Pair0**(***kwargs*)

A socket for bidirectional, one-to-one communication, with a single partner. The Python version of nng_pair0.

This is the most basic type of socket. It accepts the same keyword arguments as *Socket* and also has the same *attributes*.

This demonstrates the synchronous API:

```
from pynng import Pair0
address = 'tcp://127.0.0.1:13131'
# in real code you should also pass recv_timeout and/or send_timeout
with Pair0(listen=address) as s0, Pair0(dial=address) as s1:
    s0.send(b'hello s1')
    print(s1.recv())  # prints b'hello s1'
    s1.send(b'hi old buddy s0, great to see ya')
    print(s0.recv())  # prints b'hi old buddy s0, great to see ya
```

This demonstrates the asynchronous API using Trio. Remember that `asyncio` is also supported.

```
from trio import run
from pynng import Pair0


async def send_and_recv():
    address = 'tcp://127.0.0.1:13131'
    # in real code you should also pass recv_timeout and/or send_timeout
    with Pair0(listen=address) as s0, Pair0(dial=address) as s1:
        await s0.asend(b'hello s1')
        print(await s1.arecv())  # prints b'hello s1'
        await s1.asend(b'hi old buddy s0, great to see ya')
        print(await s0.arecv())  # prints b'hi old buddy s0, great to see ya

run(send_and_recv)
```

class pynng.**Pair1**(*, *polyamorous=None*, ***kwargs*)

A socket for bidirectional communication with potentially many partners. The Python version of nng_pair1.

It accepts the same keyword arguments as *Socket* and also has the same *attributes*. It also has one extra keyword-only argument, `polyamorous`, which must be set to `True` to connect with more than one peer.

Note: If you want to connect to multiple peers you **must** pass `polyamorous=True` when you create your socket.

To get the benefits of polyamory, you need to use the methods that work with *Message* objects: *Socket.recv_msg()* and *Socket.arecv_msg()* for receiving, and *Pipe.send()* and *Pipe.asend()* for sending.

Here is an example of the synchronous API, where a single listener connects to multiple peers. This is more complex than the *Pair0* case, because it requires to use the *Pipe* and *Message* interfaces.

```
from pynng import Pair1

address = 'tcp://127.0.0.1:12343'
with Pair1(listen=address, polyamorous=True) as s0, \
        Pair1(dial=address, polyamorous=True) as s1, \
```

(continues on next page)

(continued from previous page)

```
        Pair1(dial=address, polyamorous=True) as s2:
    s1.send(b'hello from s1')
    s2.send(b'hello from s2')
    msg1 = s0.recv_msg()
    msg2 = s0.recv_msg()
    print(msg1.bytes)  # prints b'hello from s1'
    print(msg2.bytes)  # prints b'hello from s2'
    msg1.pipe.send(b'hey s1')
    msg2.pipe.send(b'hey s2')
    print(s2.recv())  # prints b'hey s2'
    print(s1.recv())  # prints b'hey s1'
```

And here is an example using the async API, using Trio.

```
from pynng import Pair1
import trio


async def polyamorous_send_and_recv():
    address = 'tcp://127.0.0.1:12343'
    with Pair1(listen=address, polyamorous=True) as s0, \
            Pair1(dial=address, polyamorous=True) as s1, \
            Pair1(dial=address, polyamorous=True) as s2:
        await s1.asend(b'hello from s1')
        await s2.asend(b'hello from s2')
        msg1 = await s0.arecv_msg()
        msg2 = await s0.arecv_msg()
        print(msg1.bytes)  # prints b'hello from s1'
        print(msg2.bytes)  # prints b'hello from s2'
        await msg1.pipe.asend(b'hey s1')
        await msg2.pipe.asend(b'hey s2')
        print(await s2.arecv())  # prints b'hey s2'
        print(await s1.arecv())  # prints b'hey s1'

trio.run(polyamorous_send_and_recv)
```

class pynng.**Req0**(*, *resend_time=None*, ***kwargs*)

A req0 socket.

The Python version of nng_req. It accepts the same keyword arguments as *Socket* and also has the same *attributes*. It also has one extra keyword-argument: `resend_time`. `resend_time` corresponds to `NNG_OPT_REQ_RESENDTIME`

A *Req0* socket is paired with a *Rep0* socket and together they implement normal request/response behavior. the req socket *send()s* a request, the rep socket *recv()s* it, the rep socket `send()s` a response, and the req socket *recv()s* it.

If a req socket attempts to do a *recv()* without first doing a *send()*, a `pynng.BadState` exception is raised.

A *Req0* socket supports opening multiple *Contexts* by calling *new_context()*. In this way a req socket can have multiple outstanding requests to a single rep socket. Without opening a *Context*, the socket can only have a single outstanding request at a time.

Here is an example demonstrating the request/response pattern.

```
from pynng import Req0, Rep0
```

(continues on next page)

(continued from previous page)

```
address = 'tcp://127.0.0.1:13131'

with Rep0(listen=address) as rep, Req0(dial=address) as req:
    req.send(b'random.random()')
    question = rep.recv()
    answer = b'4'  # guaranteed to be random
    rep.send(answer)
    print(req.recv())  # prints b'4'
```

class pynng.**Rep0**(***kwargs*)

A rep0 socket.

The Python version of nng_rep. It accepts the same keyword arguments as *Socket* and also has the same *attributes*.

A *Rep0* socket along with a *Req0* socket implement the request/response pattern: the req socket *send()s* a request, the rep socket *recv()s* it, the rep socket send()s a response, and the req socket *recv()s* it.

A *Rep0* socket supports opening multiple *Contexts* by calling *new_context()*. In this way a rep socket can service multiple requests at the same time. Without opening a *Context*, the rep socket can only service a single request at a time.

See the documentation for *Req0* for an example.

class pynng.**Pub0**(***kwargs*)

A pub0 socket.

The Python version of nng_pub. It accepts the same keyword arguments as *Socket* and also has the same *attributes*. A *Pub0* socket calls *send()*, the data is published to all connected *subscribers*.

Attempting to *recv()* with a Pub0 socket will raise a pynng.NotSupported exception.

See docs for *Sub0* for an example.

class pynng.**Sub0**(***kwargs*)

A sub0 socket.

The Python version of nng_sub. It accepts the same keyword arguments as *Socket* and also has the same *attributes*. It also has one additional keyword argument: topics. If topics is given, it must be either a str, bytes, or an iterable of str and bytes.

A subscriber must *subscribe()* to specific topics, and only messages that match the topic will be received. A subscriber can subscribe to as many topics as you want it to.

A match is determined if the message starts with one of the subscribed topics. So if the subscribing socket is subscribed to the topic b'hel', then the messages b'hel', b'help him and b'hello' would match, but the message b'hexagon' would not. Subscribing to an empty string (b'') means that all messages will match. If a sub socket is not subscribed to any topics, no messages will be receieved.

Note: pub/sub is a "best effort" transport; if you have a very high volume of messages be prepared for some messages to be silently dropped.

Attempting to *send()* with a Sub0 socket will raise a pynng.NotSupported exception.

The following example demonstrates a basic usage of pub/sub:

```
import time
from pynng import Pub0, Sub0, Timeout
```

(continues on next page)

(continued from previous page)

```
address = 'tcp://127.0.0.1:31313'
with Pub0(listen=address) as pub, \
        Sub0(dial=address, recv_timeout=100) as sub0, \
        Sub0(dial=address, recv_timeout=100) as sub1, \
        Sub0(dial=address, recv_timeout=100) as sub2, \
        Sub0(dial=address, recv_timeout=100) as sub3:

    sub0.subscribe(b'wolf')
    sub1.subscribe(b'puppy')
    # The empty string matches everything!
    sub2.subscribe(b'')
    # we're going to send two messages before receiving anything, and this is
    # the only socket that needs to receive both messages.
    sub2.recv_buffer_size = 2
    # sub3 is not subscribed to anything
    # make sure everyone is connected
    time.sleep(0.05)

    pub.send(b'puppy: that is a cute dog')
    pub.send(b'wolf: that is a big dog')

    print(sub0.recv())  # prints b'wolf...' since that is the matching message
    print(sub1.recv())  # prints b'puppy...' since that is the matching message

    # sub2 will receive all messages (since empty string matches everything)
    print(sub2.recv())  # prints b'puppy...' since it was sent first
    print(sub2.recv())  # prints b'wolf...' since it was sent second

    try:
        sub3.recv()
        assert False, 'never gets here since sub3 is not subscribed'
    except Timeout:
        print('got a Timeout since sub3 had no subscriptions')
```

subscribe(*topic*)

Subscribe to the specified topic.

Topics are matched by looking at the first bytes of any received message.

Note: If you pass a `str` as the `topic`, it will be automatically encoded with `str.encode()`. If this is not the desired behavior, just pass `bytes` in as the topic.

---

unsubscribe(*topic*)

Unsubscribe to the specified topic.

Note: If you pass a `str` as the `topic`, it will be automatically encoded with `str.encode()`. If this is not the desired behavior, just pass `bytes` in as the topic.

---

class pynng.**Push0**(***kwargs*)

A push0 socket.

The Python version of nng_push. It accepts the same keyword arguments as *Socket* and also has the same *attributes*.

A *Push0* socket is the pushing end of a data pipeline. Data sent from a push socket will be sent to a *single* connected *Pull0* socket. This can be useful for distributing work to multiple nodes, for example. Attempting to call *recv()* on a Push0 socket will raise a `pynng.NotSupported` exception.

Here is an example of two *Pull0* sockets connected to a *Push0* socket.

```
import time

from pynng import Push0, Pull0, Timeout

addr = 'tcp://127.0.0.1:31313'
with Push0(listen=addr) as push, \
        Pull0(dial=addr, recv_timeout=100) as pull0, \
        Pull0(dial=addr, recv_timeout=100) as pull1:
    pass
    # give some time to connect
    time.sleep(0.01)
    push.send(b'hi some node')
    push.send(b'hi some other node')
    print(pull0.recv())  # prints b'hi some node'
    print(pull1.recv())  # prints b'hi some other node'
    try:
        pull0.recv()
        assert False, "Cannot get here, since messages are sent round robin"
    except Timeout:
        pass
```

class pynng.**Pull0**(***kwargs*)

A pull0 socket.

The Python version of nng_pull. It accepts the same keyword arguments as *Socket* and also has the same *attributes*.

A *Pull0* is the receiving end of a data pipeline. It needs to be paired with a *Push0* socket. Attempting to *send()* with a Pull0 socket will raise a `pynng.NotSupported` exception.

See *Push0* for an example of push/pull in action.

class pynng.**Surveyor0**(***kwargs*)

A surveyor0 socket.

The Python version of nng_surveyor. It accepts the same keyword arguments as *Socket* and also has the same *attributes*. It has one additional attribute: `survey_time`. `survey_time` sets the amount of time a survey lasts.

Surveyor0 sockets work with *Respondent0* sockets in the survey pattern. In this pattern, a *surveyor* sends a message, and gives all *respondents* a chance to chime in. The amount of time a survey is valid is set by the attribute `survey_time`. `survey_time` is the time of a survey in milliseconds.

Here is an example:

```
from pynng import Surveyor0, Respondent0, Timeout

import time
```

(continues on next page)

(continued from previous page)

```
address = 'tcp://127.0.0.1:13131'

with Surveyor0(listen=address) as surveyor, \
        Respondent0(dial=address) as responder1, \
        Respondent0(dial=address) as responder2:
    # give time for connections to happen
    time.sleep(0.1)
    surveyor.survey_time = 500
    surveyor.send(b'who wants to party?')
    # usually these would be in another thread or process, ya know?
    responder1.recv()
    responder2.recv()
    responder1.send(b'me me me!!!')
    responder2.send(b'I need to sit this one out.')

    # accept responses until the survey is finished.
    while True:
        try:
            response = surveyor.recv()
            if response == b'me me me!!!':
                print('all right, someone is ready to party!')
            elif response == b'I need to sit this one out.':
                print('Too bad, someone is not ready to party.')
        except Timeout:
            print('survey is OVER!  Time for bed.')
            break
```

class pynng.**Respondent0**(***kwargs*)

A respondent0 socket.

The Python version of nng_respondent. It accepts the same keyword arguments as *Socket* and also has the same *attributes*. It accepts no additional arguments and has no other attributes

Surveyor0 sockets work with *Respondent0* sockets in the survey pattern. In this pattern, a *surveyor* sends a message, and gives all *respondents* a chance to chime in. The amount of time a survey is valid is set by the attribute `survey_time`. `survey_time` is the time of a survey in milliseconds.

See *Surveyor0* docs for an example.

class pynng.**Bus0**(***kwargs*)

A bus0 socket. The Python version of nng_bus.

It accepts the same keyword arguments as *Socket* and also has the same *attributes*.

A *Bus0* socket sends a message to all directly connected peers. This enables creating mesh networks. Note that messages are only sent to *directly* connected peers. You must explicitly connect all nodes with the *listen()* and corresponding *listen()* calls.

Here is a demonstration of using the bus protocol:

```
import time

from pynng import Bus0, Timeout

address = 'tcp://127.0.0.1:13131'
```

(continues on next page)

(continued from previous page)

```
with Bus0(listen=address, recv_timeout=100) as s0, \
        Bus0(dial=address, recv_timeout=100) as s1, \
        Bus0(dial=address, recv_timeout=100) as s2:
    # let all connections be established
    time.sleep(0.05)
    s0.send(b'hello buddies')
    s1.recv()  # prints b'hello buddies'
    s2.recv()  # prints b'hello buddies'
    s1.send(b'hi s0')
    print(s0.recv())  # prints b'hi s0'
    # s2 is not directly connected to s1.
    try:
        s2.recv()
        assert False, "this is never reached"
    except Timeout:
        print('s2 is not connected directly to s1!')
```

2.1.2 Pipe

class pynng.**Pipe**(...)

A "pipe" is a single connection between two endpoints. This is the Python version of nng_pipe.

There is no public constructor for a Pipe; they are automatically added to the underlying socket whenever the pipe is created.

await asend(*data*)

Asynchronously send bytes from this *Pipe*.

send(*data*)

Synchronously send bytes from this *Pipe*. This method automatically creates a *Message*, associates with this pipe, and sends it with this pipe's associated *Socket*.

2.1.3 Context

class pynng.**Context**(...)

This is the Python version of nng_context. The way to create a *Context* is by calling *Socket.new_context()*. Contexts are valid for *Req0* and *Rep0* sockets; other protocols do not support contexts.

Once you have a context, you just call *send()* and *recv()* or the async equivalents as you would on a socket.

A "context" keeps track of a protocol's state for stateful protocols (like REQ/REP). A context allows the same *Socket* to be used for multiple operations at the same time. For an example of the problem that contexts are solving, see this snippet, **which does not use contexts**, and does terrible things:

```
# start a socket to service requests.
# HEY THIS IS EXAMPLE BAD CODE, SO DON'T TRY TO USE IT
# in fact it's so bad it causes a panic in nng right now (2019/02/09):
# see https://github.com/nanomsg/nng/issues/871
import pynng
import threading

def service_reqs(s):
    while True:
        data = s.recv()
```

(continues on next page)

(continued from previous page)

```
        s.send(b"I've got your response right here, pal!")


threads = []
with pynng.Rep0(listen='tcp://127.0.0.1:12345') as s:
    for _ in range(10):
        t = threading.Thread(target=service_reqs, args=[s], daemon=True)
        t.start()
        threads.append(t)

    for thread in threads:
        thread.join()
```

Contexts allow multiplexing a socket in a way that is safe. It removes one of the biggest use cases for needing to use raw sockets.

Contexts cannot be instantiated directly; instead, create a *Socket*, and call the *new_context()* method.

await arecv()
Asynchronously receive data using this context.

await arecv_msg()
Asynchronously receive a *Message* on the context.

await asend(*data*)
Asynchronously send data using this context.

close()
Close this context.

recv()
Synchronously receive data on this context.

recv_msg()
Synchronously receive a *Message* using this context.

send(*data*)
Synchronously send data on the context.

2.1.4 Message

class pynng.**Message**(*data*)
Python interface for nng_msg. Using the *Message* interface gives more control over aspects of sending the message. In particular, you can tell which *Pipe* a message came from on receive, and you can direct which *Pipe* a message will be sent from on send.

In normal usage, you would not create a *Message* directly. Instead you would receive a message using *Socket.recv_msg()*, and send a message (implicitly) by using *Pipe.send()*.

Since the main purpose of creating a *Message* is to send it using a specific *Pipe*, it is usually more convenient to just use the *Pipe.send()* or *Pipe.asend()* method directly.

Messages in pynng are immutable; this is to prevent data corruption.

> **Warning:** Access to the message's underlying data buffer can be accessed with the `_buffer` attribute. However, care must be taken not to send a message while a reference to the buffer is still alive; if the buffer is used after a message is sent, a segfault or data corruption may (read: will) result.

2.1.5 Dialer

class pynng.**Dialer**(...)
The Python version of nng_dialer. A *Dialer* is returned whenever *Socket.dial()* is called. A list of active dialers can be accessed via Socket.dialers.

A *Dialer* is associated with a single *Socket*. The associated socket can be accessed via the socket attribute. There is no public constructor for creating a *Dialer*

close()
Close the dialer.

2.1.6 Listener

class pynng.**Listener**(...)
The Python version of nng_listener. A *Listener* is returned whenever *Socket.listen()* is called. A list of active listeners can be accessed via Socket.listeners.

A *Listener* is associated with a single *Socket*. The associated socket can be accessed via the socket attribute. There is no public constructor for creating a *Listener*.

close()
Close the listener.

2.1.7 TLSConfig

Sockets can make use of the TLS transport on top of TCP by specifying an address similar to how tcp is specified. The following are examples of valid TLS addresses:

- "tls+tcp:127.0.0.1:1313", listening on TCP port 1313 on localhost.
- "tls+tcp4:127.0.0.1:1313", explicitly requesting IPv4 for TCP port 1313 on localhost.
- "tls+tcp6://[::1]:4433", explicitly requesting IPv6 for IPv6 localhost on port 4433.

class pynng.**TLSConfig**(...)
TLS Configuration object. This object is used to configure sockets that are using the TLS transport.

Parameters

- **mode** – Must be TLSConfig.MODE_CLIENT or TLSConfig.MODE_SERVER. Corresponds to nng's mode argument in nng_tls_config_alloc.
- **server_name** (*str*) – When configuring a client, server_name is used to compare the identity of the server's certificate. Corresponds to nng_tls_config_server_name.
- **ca_string** (*str*) – Set certificate authority with a string. Corresponds to nng_tls_config_ca_chain
- **own_key_string** (*str*) – When passed with own_cert_string, is used to set own certificate. Corresponds to nng_tls_config_own_cert.
- **own_cert_string** (*str*) – When passed with own_key_string, is used to set own certificate. Corresponds to nng_tls_config_own_cert.
- **auth_mode** – Set the authentication mode of the connection. Corresponds to nng_tls_config_auth_mode.
- **ca_files** (*str or list[str]*) – ca files to use for the TLS connection. Corresponds to nng_tls_config_ca_file.

- **cert_key_file** (*str*) – Corresponds to `nng_tls_config_cert_key_file`.
- **passwd** (*str*) – Password used for configuring certificates.

Check the TLS tests for usage examples.

2.2 Exceptions in pynng

pynng translates all of NNG error codes into Python Exceptions. The root exception of the hierarchy is the `NNGException`; `NNGException` inherits from Exception, and all other exceptions defined in this library inherit from `NNGException`.

The following table describes all the exceptions defined by pynng. The first column is the name of the exception in pynng (defined in `pynng.exceptions`), the second is the nng error code (defined in `nng.h`), and the third is a description of the exception.

pynng Exception	nng error code	Description
`Interrupted`	`NNG_EINTR`	The call was interrupted; if this happens, Python may throw a Keyboar
`NoMemory`	`NNG_ENOMEM`	Not enough memory to complete the operation.
`InvalidOperation`	`NNG_EINVAL`	An invalid operation was requested on the resource.
`Busy`	`NNG_EBUSY`	
`Timeout`	`NNG_ETIMEDOUT`	The operation timed out. Some operations cannot time out; an example
`ConnectionRefused`	`NNG_ECONNREFUSED`	The remote socket refused a connection.
`Closed`	`NNG_ECLOSED`	The resource was already closed and cannot complete the requested op
`TryAgain`	`NNG_EAGAIN`	The requested operation would block, but non-blocking mode was requ
`NotSupported`	`NNG_ENOTSUP`	The operation is not supported on the socket. For example, attempting
`AddressInUse`	`NNG_EADDRINUSE`	The requested address is already in use and cannot be bound to. This h
`BadState`	`NNG_ESTATE`	An operation was attempted in a bad state; for example, attempting to
`NoEntry`	`NNG_ENOENT`	The requested resource does not exist.
`ProtocolError`	`NNG_EPROTO`	
`DestinationUnreachable`	`NNG_EUNREACHABLE`	Could not reach the destination.
`AddressInvalid`	`NNG_EADDRINVAL`	An invalid address was specified. For example, attempting to listen on
`PermissionDenied`	`NNG_EPERM`	You did not have permission to do the requested operation.
`MessageTooLarge`	`NNG_EMSGSiZE`	
`ConnectionReset`	`NNG_ECONNRESET`	
`ConnectionAborted`	`NNG_ECONNABORTED`	
`Canceled`	`NNG_ECANCELED`	
`OutOfFiles`	`NNG_ENOFILES`	
`OutOfSpace`	`NNG_ENOSPC`	
`AlreadyExists`	`NNG_EEXIST`	
`ReadOnly`	`NNG_EREADONLY`	
`WriteOnly`	`NNG_EWRITEONLY`	
`CryptoError`	`NNG_ECRYPTO`	
`AuthenticationError`	`NNG_EPEERAUTH`	
`NoArgument`	`NNG_ENOARG`	
`Ambiguous`	`NNG_EAMBIGUOUS`	
`BadType`	`NNG_EBADTYPE`	
`Internal`	`NNG_EINTERNAL`	

CHAPTER 3

Indices and tables

- genindex
- modindex
- search

Index