APNs client Documentation

Release 0.2 beta

Sardar Yumatov

Jul 20, 2017

Contents

Python client for Apple Push Notification service (APNs).

Check the client with similar interface for Google Cloud Messaging.

CHAPTER 1

Requirements

- six - Python 2 and 3 compatibility library.
- pyOpenSSL - OpenSSL wrapper. Required by standard networking back-end.

Standard library has support for SSL transport. However, it is impossible to use it with certificates provided as a string. We store certificates in database, because we handle different apps on many Celery worker machines. A dirty solution would be to create temporary files, but it is insecure and slow. So, we have decided to use a better OpenSSL wrapper and `pyOpenSSL` was the easiest to handle. `pyOpenSSL` is loaded on demand by standard networking back-end. If you use your own back-end, based on some other SSL implementation, then you don't have to install `pyOpenSSL`.

CHAPTER 2

Alternatives

There are many alternatives available. We have started with pyapns and APNSWrapper. This library differs in the following design decisions:

- *Support certificates from strings*. We do not distribute certificate files on worker machines, they fetch it from the database when needed. This approach simplifies deployment, upgrades and maintenance.
- *Keep connections persistent*. An SSL handshaking round is slow. Once connection is established, it should remain open for at least few minutes, waiting for the next batch.
- *Support enhanced format*. Apple developers have designed a notoriously bad push protocol. They have upgraded it to enhanced version, which makes it possible to detect which messages in the batch have failed.
- *Clean pythonic API*. No need for lots of classes, long lists of exceptions etc.
- *Do not hard-code validation, let APNs fail*. This decision makes library a little bit more future proof.

CHAPTER 3

Changelog

v0.2 Networking layer became pluggable, making `gevent` based implementations possible. Everything is refactored, such that IO, multi-threading and SSL are now loaded and used on demand, allowing you to cleanly override any part of the client. The API is largely backward compatible. IO related configuration is moved to transport layer and exception handling is a bit more verbose. The client is using standard logging to send fine grained debug messages.

v0.1 First simple implementation, hardwired with raw sockets and `pyOpenSSL`. It does not work in `gevent` or any other *green* environment.

CHAPTER 4

Support

APNs client was created by Sardar Yumatov, contact me if you find any bugs or need help. Contact Getlogic if you need a full-featured push notification service for all popular platforms. You can view outstanding issues on the APNs Bitbucket page.

Contents:

Getting Started

You will need Apple's developer account. Then you have to obtain your provider's certificate. The certificate must be in `PEM` format. You may keep the private key with your certificate or in a separate file. Read the APNs manual to understand its architecture and all implications. This library hides most of the complex mechanics behind APNs protocol, but some aspects, such as constructing the payload or interpreting the error codes is left to you.

Usage

If you don't mind to SSL handshake with APNs each time you send a batch of messages, then use `Session.new_connection()`. Otherwise, create a new session and keep reference to it to prevent it being garbage collected. Example:

```
from apnsclient import *

# For feedback or non-intensive messaging
con = Session().new_connection("feedback_sandbox", cert_file="sandbox.pem")

# Persistent connection for intensive messaging.
# Keep reference to session instance in some class static/global variable,
# otherwise it will be garbage collected and all connections will be closed.
session = Session()
con = session.get_connection("push_sandbox", cert_file="sandbox.pem")
```

The connections you obtain from `Session` are lazy and will be really established once you actually use it. Example of sending a message:

```
# New message to 3 devices. You app will show badge 10 over app's icon.
message = Message(["my", "device", "tokens"], alert="My message", badge=10)

# Send the message.
srv = APNs(con)
try:
    res = srv.send(message)
except:
    print "Can't connect to APNs, looks like network is down"
else:
    # Check failures. Check codes in APNs reference docs.
    for token, reason in res.failed.items():
        code, errmsg = reason
        # according to APNs protocol the token reported here
        # is garbage (invalid or empty), stop using and remove it.
        print "Device failed: {0}, reason: {1}".format(token, errmsg)

    # Check failures not related to devices.
    for code, errmsg in res.errors:
        print "Error: {}".format(errmsg)

    # Check if there are tokens that can be retried
    if res.needs_retry():
        # repeat with retry_message or reschedule your task
        retry_message = res.retry()
```

APNs protocol is notoriously badly designed. It wasn't possible to detect which message has been failed in a batch. Since last update, APNs supports enhanced message format with possibility to detect *first failed message*. On detected failure the `retry()` method will build a message with the rest of device tokens, that you can retry. Unlike GCM, you may retry it right away without any delay.

If you don't like to keep your cached connections open for too long, then close them regularly. Example:

```
import datetime

# For how long may connections stay open unused
delta = datetime.timedelta(minutes=5)

# Close all connections that have not been used in the last delta time.
# You may call this method at the end of your task or in a spearate periodic
# task. If you use threads, you may call it in a spearate maintenance
# thread.
session.outdate(delta)

# Shutdown session if you want to close all unused cached connections.
# This call is equivalent to Session.outdate() with zero delta time.
session.shutdown()
```

You have to regularly check feedback service for any invalid tokens. Schedule it on some kind of periodic task. Any reported token should be removed from your database, unless you know the token has been re-registered again. Example:

```
# feedback needs no persistent connections.
con = Session().new_connection("feedback_sandbox", cert_file="sandbox.pem")
srv = APNs(con)
```

```
try:
    # on any IO failure after successfull connection this generator
    # will simply stop iterating. you will pick the rest of the tokens
    # during next feedback session.
    for token, when in service.feedback():
        # every time a devices sends you a token, you should store
        # {token: given_token, last_update: datetime.datetime.now()}
        last_update = get_last_update_of_token(token)

        if last_update < when:
            # the token wasn't updated after the failure has
            # been reported, so the token is invalid and you should
            # stop sending messages to it.
            remove_token(token)
except:
    print "Can't connect to APNs, looks like network is down"
```

The `APNs.feedback()` may fail with IO errors, in this case the feedback generator will simply end without any warning. Don't worry, you will just fetch the rest of the feedback later. We follow here *let if fail* principle for much simpler API.

Extending the client

The client is designed to be as generic as possible. Everything related to transport layer is put together in pluggable back-ends. The default back-end is `apnsclient.backends.stdio`. It is using raw python sockets for networking, `select()` for IO blocking and `pyOpenSSL` as SSL tunnel. The default back-end will probably fail to work in `gevent` environment because `pyOpenSSL` works with POSIX file descriptors directly.

You can write your own back-end that will work in your preferred, probably *green*, environment. Therefore you have to write the following classes.

`your.module.Certificate` SSL certificate instance using SSL library of your choice. Extends from `apnsclient.certificate.BaseCertificate`. The class should implement the following methods:

- `load_context()` - actual certificate loading from file or string.
- `dump_certificate()` - dump certificate for equality check.
- `dump_digest()` - dump certificate digest, such as `sha1` or `md5`.

`your.module.Connection` SSL tunnel using IO and SSL library of your choice. Extends from `apnsclient.backends.BaseConnection`. The class should implement the following methods:

- `reset()` - flush read and write buffers before new transmission.
- `peek()` - non-blocking read, returns bytes directly available in the read buffer.
- `read()` - blocking read.
- `write()` - blocking write.
- `close()` - close underlying connection.
- `closed()` - reports connection state.

`your.module.Backend` Factory class for certificates, thread locking and networking connections. Extends from `apnsclient.backends.BaseBackend`. The class should implement the following methods:

- `get_certificate()` - load certificate and returns your custom wrapper.

- `create_lock()` - create `theading.Lock` like semaphore.
- `get_new_connection()` - open ready to use SSL connection.

The main logic behind each of these classes is easy. The certificate is used as a key in the connection pool, so it should support `__eq__` (equality) operation. The equality check can be performed by comparing whole certificate dump or just the digest if you don't like the idea to hold sensitive data in python's memory for long. The connection implements basic IO operations. The connection can be cached in the pool, so it is possible that some stale data from a previous session will slip into the next session. The remedy is to flush read an write buffers using `Connection.reset()` before sending a new message. The back-end instance acts as a factory for your certificates, locks and connections. The locking is dependent on your environment, you don't have to monkey patch `threading` module therefore.

It is a good idea to look at the source code of the standard back-end `apnsclient.backends.stdio` and elaborate from that. Your back-end can be supplied to the `Session` using fully qualified module name, as a class or as an initialized instance. If you supply your back-end using module name, then the name of your back-end class must be `Backend`.

If you hit any trouble or if you think your back-end is worth sharing with the rest of the world, then contact me Sardar Yumatov or make an issue/pull-request on APNs Bitbucket page.

apnsclient Package

Apple Push Notification service client. Only public API is documented here to limit visual clutter. Refer to the sources if you want to extend this library. Check *Getting Started* for usage examples.

`apnsclient` Package

class `apnsclient.transport.`**`Session`**(*pool='apnsclient.backends.stdio'*, *connect_timeout=10*, *write_buffer_size=2048*, *write_timeout=20*, *read_buffer_size=2048*, *read_timeout=20*, *read_tail_timeout=3*, ***pool_options*)

The front-end to the underlying connection pool. The purpose of this class is to hide the transport implementation that is being used for networking. Default implementation uses built-in python sockets and `select` for asynchronous IO.

Arguments

- pool (str, type or object): networking layer implementation.
- connect_timeout (float): timeout for new connections.
- write_buffer_size (int): chunk size for sending the message.
- write_timeout (float): maximum time to send single chunk in seconds.
- read_buffer_size (int): feedback buffer size for reading.
- read_timeout (float): timeout for reading single feedback block.
- read_tail_timeout (float): timeout for reading status frame after message is sent.
- pool_options (kwargs): passed as-is to the pool class on instantiation.

`get_connection`(*address='push_sanbox'*, *certificate=None*, ***cert_params*)

Obtain cached connection to APNs.

Session caches connection descriptors, that remain open after use. Caching saves SSL handshaking time. Handshaking is lazy, it will be performed on first message send.

You can provide APNs address as `(hostname, port)` tuple or as one of the strings:

•push_sanbox – `("gateway.sandbox.push.apple.com", 2195)`, the default.

•push_production – `("gateway.push.apple.com", 2195)`

•feedback_sandbox – `("feedback.sandbox.push.apple.com", 2196)`

•feedback_production – `("feedback.push.apple.com", 2196)`

Arguments

- address (str or tuple): target address.
- certificate (`BaseCertificate`): provider's certificate instance.
- cert_params (kwargs): `BaseCertificate` arguments, used if `certificate` instance is not given.

new_connection(*address='feedback_sandbox'*, *certificate=None*, ***cert_params*)

Obtain new connection to APNs. This method will not re-use existing connection from the pool. The connection will be closed after use.

Unlike *get_connection()* this method does not cache the connection. Use it to fetch feedback from APNs and then close when you are done.

Arguments

- address (str or tuple): target address.
- certificate (`BaseCertificate`): provider's certificate instance.
- cert_params (kwargs): `BaseCertificate` arguments, used if `certificate` instance is not given.

outdate(*delta*)

Close open unused connections in the pool that are left untouched for more than `delta` time.

You may call this method in a separate thread or run it in some periodic task. If you don't, then all connections will remain open until session is shut down. It might be an issue if you care about your open server connections.

Arguments delta (`timedelta`): maximum age of unused connection.

shutdown()

Shutdown all connections in the pool. This method does will not close connections being use at the calling time.

class `apnsclient.apns.`**APNs**(*connection*)

APNs client.

Arguments

- connection (`Connection`): the connection to talk to.

feedback()

Fetch feedback from APNs.

The method returns generator of `(token, datetime)` pairs, denoting the timestamp when APNs has detected the device token is not available anymore, probably because application was uninstalled. You have to stop sending notifications to that device token unless it has been re-registered since reported timestamp.

Unlike sending the message, you should fetch the feedback using non-cached connection. Once whole feedback has been read, this method will automatically close the connection.

Note: If the client fails to connect to APNs, probably because your network is down, then this method will raise the related exception. However, if connection is successfully established, but later on the IO fails, then this method will simply stop iterating. The rest of the failed tokens will be delivered during the next feedback session.

Example:

```
session = Session()
# get non-cached connection, free from possible garbage
con = session.new_connection("feedback_production", cert_string=db_
↪certificate)
service = APNs(con)
try:
    # on any IO failure after successfull connection this generator
    # will simply stop iterating. you will pick the rest of the tokens
    # during next feedback session.
    for token, when in service.feedback():
        # every time a devices sends you a token, you should store
        # {token: given_token, last_update: datetime.datetime.now()}
        last_update = get_last_update_of_token(token)

        if last_update < when:
            # the token wasn't updated after the failure has
            # been reported, so the token is invalid and you should
            # stop sending messages to it.
            remove_token(token)
except:
    print "Check your network, I could not connect to APNs"
```

Returns generator over `(binary, datetime)`

send(*message*)

Send the message.

The method will block until the whole message is sent. The method returns *Result* object, which you can examine for possible errors and retry attempts.

Note: If the client fails to connect to APNs, probably because your network is down, then this method will raise the related exception. However, if connection is successfully established, but later on the IO fails, then this method will prepare a retry message with the rest of the failed tokens.

Example:

```
# if you use cached connections, then store this session instance
# somewhere global, such that it will not be garbage collected
# after message is sent.
session = Session()
# get a cached connection, avoiding unnecessary SSL handshake
con = session.get_connection("push_production", cert_string=db_certificate)
message = Message(["token 1", "token 2"], alert="Message")
service = APNs(con)
try:
    result = service.send(message)
except:
```

```
    print "Check your network, I could not connect to APNs"
else:
    for token, (reason, explanation) in result.failed.items():
        delete_token(token) # stop using that token

    for reason, explanation in result.errors:
        pass # handle generic errors

    if result.needs_retry():
        # extract failed tokens as new message
        message = message.retry()
        # re-schedule task with the new message after some delay
```

Returns *Result* object with operation results.

class apnsclient.apns.**Message**(*tokens*, *alert=None*, *badge=None*, *sound=None*, *content_available=None*, *expiry=None*, *payload=None*, *priority=10*, *extra=None*, ***extra_kwargs*)

The push notification to one or more device tokens.

Read more about the payload.

Note: In order to stay future compatible this class doesn't transform provided arguments in any way. It is your responsibility to provide correct values and ensure the payload does not exceed the limit of 256 bytes. You can also generate whole payload yourself and provide it via `payload` argument. The payload will be parsed to init default fields like alert and badge. However if parsing fails, then these standard fields will become unavailable. If raw payload is provided, then other data fields like alert or sound are not allowed.

Arguments

- tokens (str or list): set of device tokens where to the message will be sent.
- alert (str or dict): the message; read APNs manual for recognized dict keys.
- badge (int or str): badge number over the application icon or special value such as "increment".
- sound (str): sound file to play on arrival.
- content_available (int): set to 1 to indicate new content is available.
- expiry (int, datetime or timedelta): timestamp when message will expire.
- **payload (dict or str): JSON-compatible dictionary with the** complete message payload. If supplied, it is given instead of all the other, more specific parameters.
- priority (int): priority of the message, defaults to 10
- extra (dict): extra payload key-value pairs.
- extra_kwargs (kwargs): extra payload key-value paris, will be merged with `extra`.

__getstate__()

Returns `dict` with `__init__` arguments.

If you use `pickle`, then simply pickle/unpickle the message object. If you use something else, like JSON, then:

```
# obtain state dict from message
state = message.__getstate__()
# send/store the state
# recover state and restore message
message_copy = Message(**state)
```

Note: The message keeps `expiry` internally as a timestamp (integer). So, if values of all other arguments are JSON serializable, then the returned state must be JSON serializable. If you get `TypeError` when you instantiate `Message` from JSON recovered state, then make sure the keys are `str`, not `unicode`.

Returns *kwargs* for *Message* constructor.

tokens
List target device tokens.

class apnsclient.apns.**Result**(*message*, *failure=None*)
Result of send operation.

errors
Returns list of `(reason, explanation)` pairs denoting severe errors, not related to failed tokens. The reason is an integer code as described in APNs tutorial.

The following codes are considered to be errors:

- `(1, "Processing error")`
- `(3, "Missing topic")`
- `(4, "Missing payload")`
- `(6, "Invalid topic size")`
- `(7, "Invalid payload size")`
- `(None, "Unknown")`, usually some kind of IO failure.

failed
Reports failed tokens as `{token : (reason, explanation)}` mapping.

Current APNs protocols bails out on first failed device token, so the returned dict will contain at most 1 entry. Future extensions may upgrade to multiple failures in a batch. The reason is the integer code as described in APNs tutorial.

The following codes are considered to be token failures:

- `(2, "Missing device token")`
- `(5, "Invalid token size")`
- `(8, "Invalid token")`

needs_retry()
Returns True if there are tokens that should be retried.

Note: In most cases if `needs_retry` is true, then the reason of incomplete batch is to be found in `errors` and `failed` properties. However, Apple added recently a special code *10 - Shutdown*, which indicates server went into a maintenance mode before the batch completed. This response is not really an error, so the before mentioned properties will be empty, while `needs_retry` will be true.

retry()

Returns *Message* with device tokens that can be retried.

Current APNs protocol bails out on first failure, so any device token after the failure should be retried. If failure was related to the token, then it will appear in *failed* set and will be in most cases skipped by the retry message.

CHAPTER 5

Indices and tables

- genindex
- modindex
- search

Python Module Index

a

Index

Symbols

A

E

F

G

M

N

O

R

S

T