PostTroll Documentation

Release v1.2.2

Pytroll crew

November 10, 2016

Contents

PostTroll is a message system for pytroll.

A typical use is for event-driven production chains, using messages for notifications.

To get the software, take a look on github.

CHAPTER 1

Use Example

The main use of this library is the *posttroll.message.Message*, *posttroll.subscriber.Subscribe* and *posttroll.publisher.Publish* classes, but the *nameserver* script is also necessary. The *nameserver* scripts allows to register data publishers and then for the subscribers to find them. Here is the usage of the *nameserver* script:

```
usage: nameserver [-h] [-d {start,stop,status,restart}] [-l LOG] [-v]

optional arguments:
  -h, --help            show this help message and exit
  -d {start,stop,status,restart}, --daemon {start,stop,status,restart}
                        Run as a daemon
  -l LOG, --log LOG     File to log to (defaults to stdout)
  -v, --verbose         print debug messages too
  --no-multicast        disable address broadcasting via multicasting
```

So, after starting the nameserver, making two processes communicate is fairly easy. Here is an example of publishing code:

```
from posttroll.publisher import Publish
    from posttroll.message import Message
    import time

    try:
        with Publish("a_service", 9000) as pub:
            counter = 0
            while True:
                counter += 1
                message = Message("/counter", "info", str(counter))
                print "publishing", message
                pub.send(str(message))
                time.sleep(3)
    except KeyboardInterrupt:
        print "terminating publisher..."
```

And the subscribing code:

```
from posttroll.subscriber import Subscribe

    with Subscribe("a_service", "counter",) as sub:
        for msg in sub.recv():
            print msg
```

If you do not want to broadcast addresses via multicasting to nameservers in your network, you can start the nameserver

with the argument *–no-multicast*. Doing that, you have to specify the nameserver(s) explicitly in the publishing code:

```
from posttroll.publisher import Publish
    from posttroll.message import Message
    import time

    try:
        with Publish("a_service", 9000, nameservers=['localhost']) as pub:
            counter = 0
            while True:
                counter += 1
                message = Message("/counter", "info", str(counter))
                print "publishing", message
                pub.send(str(message))
                time.sleep(3)
    except KeyboardInterrupt:
        print "terminating publisher..."
```

See also:

posttroll.publisher.Publish and *posttroll.subscriber.Subscribe*

CHAPTER 2

Converting from older posttroll versions

Migrating from older versions of posttroll (pre v0.2), so some adaptations have to be made. Instead of *data types*, the services now have *aliases*. So, for the publishing, the following call:

```
with Publish("a_service", ["data_type1", "data_type2"], 9000) as pub:
```

would translate into:

```
with Publish("a_service", 9000, ["data_type1", "data_type2"]) as pub:
```

On the subscriber side, the following:

```
with Subscribe("data_type1") as sub:
```

would have to be changed to:

```
with Subscribe("a_service") as sub:
```

Note that the behaviour is changed: all the messages comming from the publisher *a_service* would be iterated over, including messages that have another data type than the one you want. This is why there is now the possibility to add a subject filter directly inside the *posttroll.subscriber.Subscribe* call:

```
with Subscribe("a_service", "data_type1") as sub:
```

This means that the subjects of the messages you are interested in should start with “data_type1” though...

CHAPTER 3

API

3.1 Publisher

The publisher module gives high-level tools to publish messages on a port.

class posttroll.publisher.**NoisyPublisher**(*name*, *port=0*, *aliases=None*, *broadcast_interval=2*, *nameservers=None*)

Same as a Publisher, but with broadcasting of its own name and address.

Setting the *name* to a meaningful value is import since it will be searchable in the nameserver. The *port* is to be provided as an int, and setting to 0 means it will be set to a random free port. *aliases* is a list of alternative names for the process. *broadcast_interval*, in seconds (2 by default) says how often the current name and address should be broadcasted. If *nameservers* is non-empty, multicasting will be deactivated and the publisher registers on these nameservers only

send(*msg*)

Send a *msg*.

start()

Start the publisher.

stop()

Stop the publisher.

class posttroll.publisher.**Publish**(*name*, *port=0*, *aliases=None*, *broadcast_interval=2*, *nameservers=None*)

The publishing context.

Broadcasts also the *name*, *port*, and optional *aliases* (using *posttroll.message_broadcaster.MessageBroadcaste*

See *NoisyPublisher* for more information on the arguments.

Example on how to use the *Publish* context:

```
from posttroll.publisher import Publish
from posttroll.message import Message
import time

try:
    with Publish("my_service", 9000) as pub:
        counter = 0
        while True:
            counter += 1
            message = Message("/counter", "info", str(counter))
            print "publishing", message
```

```
                pub.send(str(message))
                time.sleep(3)
    except KeyboardInterrupt:
        print "terminating publisher..."
```

class posttroll.publisher.**Publisher**(*address*, *name=''*)
The publisher class.

address is the current address of the Publisher, e.g.:

```
tcp://localhost:1234
```

Setting the port to 0 means that a random free port will be chosen for you.

name is simply the name of the publisher.

An example on how to use the *Publisher*:

```
from posttroll.publisher import Publisher, get_own_ip
from posttroll.message import Message
import time

pub_address = "tcp://" + str(get_own_ip()) + ":9000"
pub = Publisher(pub_address)

try:
    counter = 0
    while True:
        counter += 1
        message = Message("/counter", "info", str(counter))
        pub.send(str(message))
        time.sleep(3)
except KeyboardInterrupt:
    print "terminating publisher..."
    pub.stop()
```

heartbeat(*min_interval=0*)
Send a heartbeat ... but only if *min_interval* seconds has passed since last beat.

send(*msg*)
Send the given message.

stop()
Stop the publisher.

posttroll.publisher.**get_own_ip**()
Get the host's ip number.

3.2 Subscriber

Simple library to subscribe to messages.

class posttroll.subscriber.**NSSubscriber**(*services=None*, *topics='pytroll:/'*, *addr_listener=False*, *addresses=None*, *timeout=10*, *translate=False*, *nameserver='localhost'*)
Automatically subscribe to *services* (requesting addresses from the nameserver. If *topics* are specified, filter the messages through the beginning of the subject. *addr_listener* allows to add new services on the fly as they appear on the network. Additional *addresses* to subscribe to can be specified, and address translation can be

performed if *translate* is set to True (False by default). The *timeout* here is specified in seconds. The *nameserver* tells which host should be used for nameserver requests, defaulting to "localhost".

Note: 'services = None', means no services, and 'services =""' means all services.

start()
Start the subscriber.

stop()
Stop the subscriber.

class posttroll.subscriber.**Subscribe**(*services=None*, *topics='pytroll:/'*, *addr_listener=False*, *addresses=None*, *timeout=10*, *translate=False*, *nameserver='localhost'*)
Subscriber context. See *NSSubscriber* for initialization parameters.

Example:

```
from posttroll.subscriber import Subscribe

with Subscribe("a_service", "my_topic",) as sub:
    for msg in sub.recv():
        print msg
```

class posttroll.subscriber.**Subscriber**(*addresses*, *topics=''*, *message_filter=None*, *translate=False*)
Subscribes to *addresses* for *topics*, and perform address translation of *translate* is true. The function *message_filter* can be used to discriminate some messages on the subscriber side. *topics* on the other hand performs filtering on the publishing side (from zeromq 3).

Example:

```
from posttroll.subscriber import Subscriber, get_pub_address

addr = get_pub_address(service, timeout=2)
sub = Subscriber([addr], 'my_topic')
try:
    for msg in sub(timeout=2):
        print "Consumer got", msg

except KeyboardInterrupt:
    print "terminating consumer..."
    sub.close()
```

add(*address*, *topics=None*)
Add *address* to the subscribing list for *topics*.

It topics is None we will subscibe to already specified topics.

add_hook_pull(*address*, *callback*)
Same as above, but with a PULL socket. (e.g good for pushed 'inproc' messages from another thread).

add_hook_sub(*address*, *topics*, *callback*)
Specify a *callback* in the same stream (thread) as the main receive loop. The callback will be called with the received messages from the specified subscription.

Good for operations, which is required to be done in the same thread as the main recieve loop (e.q operations on the underlying sockets).

addresses
Get the addresses

close()
Close the subscriber: stop it and close the local subscribers.

recv(*timeout=None*)
Receive, optionally with *timeout* in seconds.

remove(*address*)
Remove *address* from the subscribing list for *topics*.

stop()
Stop the subscriber.

subscribers
Get the subscribers

update(*addresses*)
Updating with a set of addresses.

3.3 Messages

A Message goes like: <subject> <type> <sender> <timestamp> <version> [mime-type data]

```
Message('/DC/juhu', 'info', 'jhuuuu !!!')
```

will be encoded as (at the right time and by the right user at the right host):

```
pytroll://DC/juhu info henry@prodsat 2010-12-01T12:21:11.123456 v1.01 application/json "jhuuuu !!!"
```

Note: It's not optimized for BIG messages.

class posttroll.message.**Message**(*subject=''*, *atype=''*, *data=''*, *binary=False*, *rawstr=None*)
A Message.

- Has to be initialized with a *rawstr* (encoded message to decode) OR
- Has to be initialized with a *subject*, *type* and optionally *data*, in which case:
  - It will add add few extra attributes.
  - It will make a Message pickleable.

static decode(*rawstr*)
Decode a raw string into a Message.

encode()
Encode a Message to a raw string.

head
Return header of a message (a message without the data part).

host
Try to return a host from a sender.

user
Try to return a user from a sender.

exception posttroll.message.**MessageError**
This modules exceptions.

posttroll.message.**datetime_decoder**(*dct*)
Decode datetimes to python objects.

posttroll.message.**datetime_encoder**(*obj*)
Encodes datetimes into iso format.

posttroll.message.**is_valid_data**(*obj*)
Check if data is JSON serializable.

posttroll.message.**is_valid_sender**(*obj*)
Currently we only check for empty strings.

posttroll.message.**is_valid_subject**(*obj*)
Currently we only check for empty strings.

posttroll.message.**is_valid_type**(*obj*)
Currently we only check for empty strings.

3.4 Address receiver

Receive broadcasted addresses in a standard pytroll Message: /<server-name>/address info ... host:port

class posttroll.address_receiver.**AddressReceiver**(*max_age=datetime.timedelta(0, 600)*, *port=None*, *do_heartbeat=True*, *multicast_enabled=True*)
General thread to receive broadcast addresses.

get(*name=''*)
Get the address(es).

is_running()
Check if the receiver is alive.

start()
Start the receiver.

stop()
Stop the receiver.

posttroll.address_receiver.**getaddress**
alias of *AddressReceiver*

3.5 Name server

Manage other's subscriptions.

Default port is 5557, if $NAMESERVER_PORT is not defined.

class posttroll.ns.**NameServer**(*max_age=datetime.timedelta(0, 600)*, *multicast_enabled=True*)
The name server.

run(**args*)
Run the listener and answer to requests.

stop()
Stop the name server.

exception posttroll.ns.**TimeoutError**
A timeout.

posttroll.ns.**get_active_address**(*name*, *arec*)
Get the addresses of the active modules for a given publisher *name*.

posttroll.ns.**get_pub_address**(*name*, *timeout=10*, *nameserver='localhost'*)
Get the address of the publisher for a given publisher *name* from the nameserver on *nameserver* (localhost by default).

posttroll.ns.**get_pub_addresses**(*names=None*, *timeout=10*, *nameserver='localhost'*)
Get the address of the publisher for a given list of publisher *names* from the nameserver on *nameserver* (localhost by default).

3.6 Multicasting

3.6.1 Context

class posttroll.message_broadcaster.**MessageBroadcaster**(*msg*, *port*, *interval*, *designated_receivers=None*)
Class to broadcast stuff.

If *interval* is 0 or negative, no broadcasting is done.

is_running()
Are we running.

start()
Start the broadcasting.

stop()
Stop the broadcasting.

class posttroll.message_broadcaster.**AddressBroadcaster**(*name*, *address*, *interval*, *nameservers*)
Class to broadcast stuff.

posttroll.message_broadcaster.**sendaddress**
alias of *AddressBroadcaster*

3.6.2 Multicast code

Send/receive UDP multicast packets. Requires that your OS kernel supports IP multicast.

This is based on python-examples Demo/sockets/mcast.py

class posttroll.bbmcast.**MulticastSender**(*port*, *mcgroup='225.0.0.212'*)
Multicast sender on *port* and *mcgroup*.

close()
Close the sender.

class posttroll.bbmcast.**MulticastReceiver**(*port*, *mcgroup='225.0.0.212'*)
Multicast receiver on *port* for an *mcgroup*.

BUFSIZE = 1024

close()
Close the receiver.

settimeout(*tout=None*)
A timeout will throw a 'socket.timeout'.

posttroll.bbmcast.**mcast_sender**(*mcgroup='225.0.0.212'*)
Non-object interface for sending multicast messages.

posttroll.bbmcast.**mcast_receiver**(*port*, *mcgroup='225.0.0.212'*)
Open a UDP socket, bind it to a port and select a multicast group.

posttroll.bbmcast.**SocketTimeout**
alias of timeout

3.7 Connections

3.8 Misc

posttroll.**renew_context**()

posttroll.**strp_isoformat**(*strg*)
Decode an ISO formatted string to a datetime object. Allow a time-string without microseconds.

We handle input like: 2011-11-14T12:51:25.123456

CHAPTER 4

Indices and tables

- genindex
- modindex
- search

p

S

T

U