pymta Documentation

Release 0.5dev

Felix Schwarz

Nov 12, 2020

Contents

pymta is a library to build a custom SMTP server in Python. This is useful if you want to...

- test mail-sending code against a real SMTP server even in your unit tests.
- build a custom SMTP server with non-standard behavior without reimplementing the whole SMTP protocol.
- have a low-volume SMTP server which can be easily extended using Python.

CHAPTER 1

Goals of pymta

The main goal of pymta is to provide a basic SMTP server for unit tests. It must be easy to inject custom behavior (policy checks) for every SMTP command. Furthermore the library should come with an extensive set of tests to ensure that does the right thing(tm).

Eventually I plan to build a highly customizable SMTP server which can be easily hacked (just for the fun of it).

CHAPTER 2

Development Status

Currently (06/2009, version 0.4) the library only implements basic SMTP with very few extensions (e.g. PLAIN authentication). However, as far as I know, it is the only MTA written in Python that implements a process-based strategy for connection handling which is an advantage because many libraries - including most Python DB API implementations - can not be used in an asynchronous environment and you can use your CPUs to their fullest extent. And last but not least pymta comes with many unit tests and good, comprehensive documentation.

'Advanced' features which are necessary for any decent MTA like TLS and pipelining are not yet implemented. Currently pymta is used only in the unit tests for TurboMail. Therefore it should be considered as beta software.

CHAPTER 3

Related Projects

There are some other SMTP server implementations in Python available which you may want to use if you need a proven implementation right now:

- Python's smtpd
- Twisted Mail
- tmda-ofmipd
- Son Of Sam Email Server
- smtps.py

Python's **smtpd** is a module which is included in the standard distribution of Python for a long time. Though it implements only a *very* basic feature set this module is used as a basis for many smaller SMTP server implementations. In the beginning I used this module for my unit tests too but quite soon I had to realize that the code is old and messy and it is nearly impossible to implement a custom behavior (e.g. reject certain recipients). pymta evolved out of smtpd after multiple refactorings based on the idea to use a simple finite state machine (initially repoze.workflow, now including a custom one).

Twisted Mail is probably the most featureful SMTP server implementation in Python available right now. It uses the twisted framework which can be either a huge advantage or disadvantage, depending on your point of view. It can use TLS via OpenSSL (using the Twisted infrastructure). When I started out with the naïve idea of just extending Python's smtpd a bit, I dismissed Twisted Mail because it seemed to be quite hard to implement some custom behavior without writing too much code.

tmda-ofmipd is another implementation which is based on Python's smtpd. It is distributed only as part of a larger Python application which makes it harder to use if you just need a plain Python SMTP server. Furthermore the code was not cleaned up so it may be a bit hard to understand but it supports TLS (using tlslite).

Son Of Sam Email Server implements an SMTP server (based Python's smtpd too) but focuses on delivery and user info lookup. There are no changes to Python's smtpd so the server does not support any kind of recipient verification.

smtps.py is a really simple, single-threaded SMTP server rewritten from scratch with a quite clean design (compared to Python's smtpd) although it only implements the absolute minimum of SMTP and many things like the command parsing are just hard-coded. On the other hand, the server's behavior can be changed by implementing a custom strategy class. Trac included an extended version of smtps in its test suite.

CHAPTER 4

Installation and Setup

pymta is just a Python library which uses setuptools so it does not require a special setup. To serve multiple connections in parallel, pymta uses the multiprocessing module which was added to the standard library in Python 2.6 (there are backports for Python 2.4 and 2.5). Furthermore you need to install pycerberus.

pymta supports Python 2.7 and Python 3.4+.

4.1 multiprocessing

The multiprocessing module hides most the operating system differences when it comes to multiple processes. The module is included in Python 2.6 but it is available standalone via pypi:

```
easy_install multiprocessing
```

If multiprocessing is not installed, pymta will fall back to single-threaded execution automatically (therefore multiprocessing is no hard requirement in the egg file).

CHAPTER 5

Architectural Overview

pytma uses multiple processes to handle more than one connection at the same time. In order to do this in a platform-independent manner, it utilizes the multiprocessing module.

The basic SMTP program flow is determined by two state machines: One for the SMTP command parsing mode (single-line commands or data) in the SMTPCommandParser and another much bigger state machine in the SMTPSession to control the correct order of commands sent by the SMTP client.

The main idea of pymta was to make it easy adding custom behavior which is considered configuration for 'real' SMTP servers like Exim. The 'pymta.api' module contains classes which define interfaces for customizations. These interfaces are part of the public API so I try to keep them stable in future releases. Use IMTAPolicy to add restrictions on certain SMTP commands (check recipient addresses, scan the message's content for spam before accepting it) and IAuthenticator to authenticate SMTP clients (check username and password). With an IMessageDeliverer you can specify what to do with received messages.

5.1 Problems with asynchronous architectures

The two most important SMTP implementations in Python (smtpd and Twisted Mail) both use an asynchronous architecture so they can serve multiple connections at the same time without the need to start multiple processes or threads. Because of this they can avoid the increased overall complexity due to locking issues and can save some resources (creating a process may be costly).

However there are some drawbacks with the asynchronous approach:

- SMTP servers are not necessarily I/O bound. Some operations like spam scanning or other message checks may eat quite a lot of CPU. With Python you need to use multiple processes if you really want to utilize multiple CPUs due to the Global Interpreter Lock.
- All libraries must be able to deal with the asynchronous pattern otherwise you risk to block all connections at the same time. Many programmers are not familiar with this pattern so most libraries do not support this. This is especially true for most of Python's DB api implementations which is why Twisted implemented its own asynchronous DB layer. Unfortunately by using this layer you have to use plain SQL, because the most popular ORMs like SQLAlchemy do not support their layer.

Given these conditions IMHO it looks like a bad design choice to use an asynchronous architecture for a SMTP server library which should be easily hackable to handle even uncommon cases.

CHAPTER 6

Components

pymta consists of several main components (classes) which may be important to know.

6.1 PythonMTA

The PythonMTA is the main server component which listens on a certain port for new connections. There should be only one instance of this object. When a new connection is received, the PythonMTA spawns WorkerProcess (if you have the multiprocessing module installed) which triggers a SMTPCommand parser that handles all the SMTP communitcation. When a message was submitted successfully, the new_message_accepted() method of your IMessageDeliverer will be called so it is in charge of actually doing something with the message.

You can instantiate a new server like that:

```
from pymta import PythonMTA, BlackholeDeliverer

if __name__ == '__main__':
    # SMTP server will listen on localhost/port 8025
    server = PythonMTA('localhost', 8025, BlackholeDeliverer())
    server.serve_forever()
```

Interface

class pymta.**PythonMTA**(*local_address*, *bind_port*, *deliverer_class*, *policy_class=None*, *authenticator_class=None*)

Create a new MTA which listens for new connections afterwards. local_address is a string containing either the IP oder the DNS host name of the interface on which PythonMTA should listen. deliverer_class, policy_class and authenticator_class are callables which can be used to add custom behavior. Please note that they must be picklable if you use forked worker processes (default). Every new connection gets their own instance of policy_class and authenticator_class so these classes don't have to be thread-safe. If you omit the policy, all syntactically valid SMTP commands are accepted. If there is no authenticator specified, authentication will not be available.

shutdown_server(*timeout_seconds=None*)

This method notifies the server that it should stop listening for new messages and shut down itself. If

timeout_seconds was given, the method will block for this many seconds at most.

6.2 Policies

class pymta.api.**IMTAPolicy**

Policies can change with behavior of an MTA dynamically (e.g. don't allow relaying unless the client is located within the trusted company network, enable authentication only for some connections). In established MTAs like Exim and Postfix it's a very important task for every system administrator to configure the message acceptance policies which are normally part of the configuration file.

A policy does not change the SMTP implementation itself (the state machine) but can send out custom replies to the client. A policy doesn't have to care if the commands were given in the correct order (the state machines will take care of that). The only thing is that the message object passed into many policy methods does not contain all data at certain stages (e.g. accept_mail_from can not access the recipients list because that was not submitted yet).

'IMTAPolicy' provides a very permissive policy (all commands are accepted) from which you can derive custom policies. Its methods are usually named 'accept_<SMTP command name>'.

Every method in the 'IMTAPolicy' interface can return either a single boolean value (True/False) or a tuple. A boolean value specifies if the command should be accepted. The caller is responsible for sending the actual default replies.

Alternatively a policy can choose to return a tuple to have more control over the reply sent to the client: (decision, (reply code, reply message)). The decision is the boolean known from the last paragraph. The reply code is an integer which should a be a valid SMTP code. reply message is either a basestring with a custom message or an iterable of basestrings (in case a a multi-line reply is sent).

Last but not least a PolicyDecision can be returned which embodies the decision as well as (optionally) a custom reply. The reply has the same format as described in the paragraph before. The PolicyDecision can ask the server to close the connection unconditionally after or even before sending the response to the client (in the latter case no response will be sent).

accept_auth_login(*username*, *message*)

Decides if AUTH LOGIN should be allowed for this client. Please note that username and password are not verified before, the authenticator will check them after the policy allowed this command.

The method must not return a response by itself in case it accepts the AUTH LOGIN command!

accept_auth_plain(*username*, *password*, *message*)

Decides if AUTH plain should be allowed for this client. Please note that username and password are not verified before, the authenticator will check them after the policy allowed this command.

The method must not return a response by itself in case it accepts the AUTH PLAIN command!

accept_data(*message*)

Decides if we allow the client to start a message transfer (the actual message contents will be transferred after this method allowed it).

accept_ehlo(*ehlo_string*, *message*)

Decides if the EHLO command with the given helo_name should be accepted.

accept_from(*sender*, *message*)

Decides if the sender of this message (MAIL FROM) should be accepted.

accept_helo(*helo_string*, *message*)

Decides if the HELO command with the given helo_name should be accepted.

accept_msgdata(*msgdata*, *message*)
This method actually matches no real SMTP command. It is called after a message was transferred completely and this is the last check before the SMTP server takes the responsibility of transferring it to the recipients.

accept_new_connection(*peer*)
This method is called directly after a new connection is received. The policy can decide if the given peer is allowed to connect to the SMTP server. If it declines, the connection will be closed immediately.

accept_rcpt_to(*new_recipient*, *message*)
Decides if recipient of this message (RCPT TO) should be accepted. If a message should be delivered to multiple recipients this method is called for every recipient.

ehlo_lines(*peer*)
Return an iterable for SMTP extensions to advertise after EHLO. By default support for SMTP SIZE extension will be announced if you set a max message size.

max_message_size(*peer*)
Return the maximum size (in bytes) for messages from this peer. When this method returns an integer, there pymta will check the actual message size after the message was received (before the accept_msgdata method is called) and will respond with the appropriate error message if necessary. If you return None, no size limit will be enforced by pymta (however you can always reject a message using accept_msgdata().

Here is a short example how you can implement a custom behavior that checks the HELO command given by the client:

```
def accept_helo(self, helo_string, message):
    # pymta will return the default error message for the given command if
    # you just return False
    return False

    # This will send out a '553 Bad helo string' and the command is
    # rejected. pymta won't send any additional reply because you did that
    # already.
    return (False, (553, 'Bad helo string'))

    # This is basically the same as above but now it will trigger a
    # multi-line SMTP response:
    # 553-Bad helo string
    # 553 Evil IP
    return (False, (553, ('Bad helo string', 'Evil IP')))
```

6.3 Authenticators

class pymta.api.**IAuthenticator**
Authenticators check if the user's credentials are actually correct. This may involve some checking against external subsystems (e.g. a database or a LDAP directory).

authenticate(*username*, *password*, *peer*)
This method is called after the client issued an AUTH PLAIN command and must return a boolean value (True/False).

6.4 Deliverers

class pymta.api.**IMessageDeliverer**

Deliverers take care of the message routing/delivery after a message was accepted (e.g. put it in a mailbox file, forward it to another server, ...).

new_message_accepted(*msg*)

This method is called when a new message was accepted by the server. Now the MTA is then in charge of delivering the message to the specified recipients. Please note that you can not reject the message anymore at this stage (if there are problems you must generate a non-delivery report aka bounce).

There will be one deliverer instance per client connection so this method may does not have to be thread-safe. However this method may get called multiple times when the client transmits more than one message for the same connection.

6.5 Message

The Message is a data object contains all information about a message sent by a client. This includes not only the actual RFC822 message contents but also information about the SMTP envelope, the peer and the helo string used. The information is filled as the client sends some commands so not all information may be available at any time (e.g. the msg_data not available before the client actually sent the RFC822 message).

6.6 Peer

The Peer is another data object which contains the remote host ip address and the remote port.

6.7 SMTPSession

This class actually implements the most complicated part of the SMTP state machine and is responsible for calling the policy. If you want to extend the functionality or need to implement some custom behavior which is beyond what you can do using Policies, check this class.

The SMTP state machine is quite strict currently but I consider this a feature and not something I'll try to improve in the near future.

6.8 Unit Test Utility Classes

pymta was created to ease testing SMTP communication without the need to set up an external SMTP server. While writing tests for other applications I created some utility classes which are probably helpful in your tests as well...

6.9 Example SMTP server application

In the examples directory you find a pymta-based implementation of a debugging server that behaves like Python's DebuggingServer: All received messages will be printed to STDOUT. Hopefully it can serve as a short reference how to write very simple pymta-based servers too.

6.10 Speed

If you want to use pymta for a real SMTP server, you should not be concerned too much about speed. If you go really for a high-volume setup with several million messages per day and hundreds of simultaneous connections, you should tune one of the well-known SMTP servers like Exim, Postfix or sendmail to get the maximum performance. However, I measured theoretical peak performance using Postal 0.70 to give you some theoretical figures.

Environment and benchmark settings:

- System: Fedora 10 with an AMD x2 4200 (2.2 GHz), Python 2.5
- pymta: version 0.3, DebuggingServer with NullDeliverer and no policy.
- postal: 4 threads, no SSL connections, one message per connection (defaults)

With that configuration I got something between 1540-2270 messages per minute (median 1879 messages) which is actually quite low. Many real SMTP servers would deliver something between 5,000-10,000 messages per minute in a comparable setting[1]. During my measurements the system load was barely noticable (below 5%) so I guess most of the time is lost waiting for locks. Using a really fast IPC mechanism or a custom PythonMTA implementation that uses the os.fork would probably increase the throughput by quite easily.

6.11 License Overview

pymta itself is licensed under the very liberal MIT license (see COPYING.txt in the source archive) so there are virtually no restrictions where you can integrate the code.

However, pymta depends on some (few) other packages which come with different licenses. In order to ease license auditing, I'll list the other licenses here (no guarantees though, check yourself before you trust):

- Python uses the Python Software Foundation License 2 which is a BSD-style license.
- The multiprocessing uses a 3-clause BSD license.
- pycerberus uses the MIT license, just like pymta.

I believe that all licenses are GPL compatible and do not require you to publish your code if you don't like to.

[1] However, as soon you add some more complicated database queries or spam and virus checks to that, the real throughput will decrease dramatically (even if the scanning takes only 0.1 seconds per message you won't exceed 600 messages per minute). In real setups the bare SMTP speed does not matter that much.

Index