Evy Documentation

Release 0.1

<evy contributors>

January 07, 2015

Contents

Code talks! This is a simple web crawler that fetches a bunch of urls concurrently:

```
urls = ["http://www.google.com/intl/en_ALL/images/logo.gif",
     "https://wiki.secondlife.com/w/images/secondlife.jpg",
     "http://us.i1.yimg.com/us.yimg.com/i/ww/beta/y3.gif"]

import evy
from evy.patched import urllib2

def fetch(url):
  return urllib2.urlopen(url).read()

pool = evy.GreenPool()
for body in pool.imap(fetch, urls):
  print "got body", len(body)
```

CHAPTER 1

Contents

1.1 Basic Usage

If it's your first time to Evy, you may find the illuminated examples in the *Design Patterns* document to be a good starting point.

Evy is built around the concept of green threads (i.e. coroutines, we use the terms interchangeably) that are launched to do network-related work. Green threads differ from normal threads in two main ways:

- Green threads are so cheap they are nearly free. You do not have to conserve green threads like you would normal threads. In general, there will be at least one green thread per network connection.
- Green threads cooperatively yield to each other instead of preemptively being scheduled. The major advantage from this behavior is that shared data structures don't need locks, because only if a yield is explicitly called can another green thread have access to the data structure. It is also possible to inspect primitives such as queues to see if they have any pending data.

1.2 Primary API

The design goal for Evy's API is simplicity and readability. You should be able to read its code and understand what it's doing. Fewer lines of code are preferred over excessively clever implementations. Like Python itself, there should be one, and only one obvious way to do it in Evy!

Though Evy has many modules, much of the most-used stuff is accessible simply by doing `import evy`. Here's a quick summary of the functionality available in the `evy` module, with links to more verbose documentation on each.

1.2.1 Greenthread Spawn

`evy.`**`spawn`**(*func*, **args*, ***kw*)
: This launches a greenthread to call *func*. Spawning off multiple greenthreads gets work done in parallel. The return value from `spawn` is a `greenthread.GreenThread` object, which can be used to retrieve the return value of *func*. See `spawn` for more details.

`evy.`**`spawn_n`**(*func*, **args*, ***kw*)
: The same as `spawn()`, but it's not possible to know how the function terminated (i.e. no return value or exceptions). This makes execution faster. See `spawn_n` for more details.

`evy.`**`spawn_after`**(*seconds*, *func*, **args*, ***kw*)
: Spawns *func* after *seconds* have elapsed; a delayed version of `spawn()`. To abort the spawn and prevent

func from being called, call `GreenThread.cancel()` on the return value of `spawn_after()`. See `spawn_after` for more details.

1.2.2 Greenthread Control

`evy.`**`sleep`**`(`*seconds=0*`)`
Suspends the current greenthread and allows others a chance to process. See `sleep` for more details.

class `evy.`**`GreenPool`**
Pools control concurrency. It's very common in applications to want to consume only a finite amount of memory, or to restrict the amount of connections that one part of the code holds open so as to leave more for the rest, or to behave consistently in the face of unpredictable input data. GreenPools provide this control. See `GreenPool` for more on how to use these.

class `evy.`**`GreenPile`**
GreenPile objects represent chunks of work. In essence a GreenPile is an iterator that can be stuffed with work, and the results read out later. See `GreenPile` for more details.

class `evy.`**`Queue`**
Queues are a fundamental construct for communicating data between execution units. Evy's Queue class is used to communicate between greenthreads, and provides a bunch of useful features for doing that. See `Queue` for more details.

class `evy.`**`Timeout`**
This class is a way to add timeouts to anything. It raises *exception* in the current greenthread after *timeout* seconds. When *exception* is omitted or `None`, the Timeout instance itself is raised.

Timeout objects are context managers, and so can be used in with statements. See `Timeout` for more details.

1.2.3 Patching Functions

`evy.`**`import_patched`**`(`*modulename*, **additional_modules*, ***kw_additional_modules*`)`
Imports a module in a way that ensures that the module uses "green" versions of the standard library modules, so that everything works nonblockingly. The only required argument is the name of the module to be imported. For more information see *Import Green*.

`evy.`**`monkey_patch`**`(`*all=True*, *os=False*, *select=False*, *socket=False*, *thread=False*, *time=False*`)`
Globally patches certain system modules to be greenthread-friendly. The keyword arguments afford some control over which modules are patched. If *all* is True, then all modules are patched regardless of the other arguments. If it's False, then the rest of the keyword arguments control patching of specific subsections of the standard library. Most patch the single module of the same name (os, time, select). The exceptions are socket, which also patches the ssl module if present; and thread, which patches thread, threading, and Queue. It's safe to call monkey_patch multiple times. For more information see *Monkeypatching the Standard Library*.

1.2.4 Network Convenience Functions

These are the basic primitives of Evy; there are a lot more out there in the other Evy modules; check out the *Module Reference*.

1.3 Design Patterns

There are a bunch of basic patterns that Evy usage falls into. Here are a few examples that show their basic structure.

1.3.1 Client Pattern

The canonical client-side example is a web crawler. This use case is given a list of urls and wants to retrieve their bodies for later processing. Here is a very simple example:

```
urls = ["http://www.google.com/intl/en_ALL/images/logo.gif",
       "https://wiki.secondlife.com/w/images/secondlife.jpg",
       "http://us.i1.yimg.com/us.yimg.com/i/ww/beta/y3.gif"]

import evy
from evy.patched import urllib2

def fetch(url):
    return urllib2.urlopen(url).read()

pool = evy.GreenPool()
for body in pool.imap(fetch, urls):
    print "got body", len(body)
```

There is a slightly more complex version of this in the *web crawler example*. Here's a tour of the interesting lines in this crawler.

`from evy.patched import urllib2` is how you import a cooperatively-yielding version of urllib2. It is the same in all respects to the standard version, except that it uses green sockets for its communication. This is an example of the *Import Green* pattern.

`pool = evy.GreenPool()` constructs a `GreenPool` of a thousand green threads. Using a pool is good practice because it provides an upper limit on the amount of work that this crawler will be doing simultaneously, which comes in handy when the input data changes dramatically.

`for body in pool.imap(fetch, urls):` iterates over the results of calling the fetch function in parallel. `imap` makes the function calls in parallel, and the results are returned in the order that they were executed.

The key aspect of the client pattern is that it involves collecting the results of each function call; the fact that each fetch is done concurrently is essentially an invisible optimization. Note also that imap is memory-bounded and won't consume gigabytes of memory if the list of urls grows to the tens of thousands (yes, we had that problem in production once!).

1.3.2 Server Pattern

Here's a simple server-side example, a simple echo server:

```
import evy

def handle(client):
    while True:
        c = client.recv(1)
        if not c: break
        client.sendall(c)

server = evy.listen(('0.0.0.0', 6000))
pool = evy.GreenPool(10000)
while True:
    new_sock, address = server.accept()
    pool.spawn_n(handle, new_sock)
```

The file *echo server example* contains a somewhat more robust and complex version of this example.

`server = evy.listen(('0.0.0.0', 6000))` uses a convenience function to create a listening socket.

`pool = evy.GreenPool(10000)` creates a pool of green threads that could handle ten thousand clients.

`pool.spawn_n(handle, new_sock)` launches a green thread to handle the new client. The accept loop doesn't care about the return value of the `handle` function, so it uses `spawn_n`, instead of `spawn`.

The difference between the server and the client patterns boils down to the fact that the server has a `while` loop calling `accept()` repeatedly, and that it hands off the client socket completely to the handle() method, rather than collecting the results.

1.3.3 Dispatch Pattern

One common use case that Linden Lab runs into all the time is a "dispatch" design pattern. This is a server that is also a client of some other services. Proxies, aggregators, job workers, and so on are all terms that apply here. This is the use case that the `GreenPile` was designed for.

Here's a somewhat contrived example: a server that receives POSTs from clients that contain a list of urls of RSS feeds. The server fetches all the feeds concurrently and responds with a list of their titles to the client. It's easy to imagine it doing something more complex than this, and this could be easily modified to become a Reader-style application:

```
import evy

feedparser = evy.import_patched('feedparser')

pool = evy.GreenPool()

def fetch_title(url):
    d = feedparser.parse(url)
    return d.feed.get('title', '')

def app(environ, start_response):
    pile = evy.GreenPile(pool)
    for url in environ['wsgi.input'].readlines():
        pile.spawn(fetch_title, url)
    titles = '\n'.join(pile)
    start_response('200 OK', [('Content-type', 'text/plain')])
    return [titles]
```

The full version of this example is in the *Feed Scraper*, which includes code to start the WSGI server on a particular port.

This example uses a global (gasp) `GreenPool` to control concurrency. If we didn't have a global limit on the number of outgoing requests, then a client could cause the server to open tens of thousands of concurrent connections to external servers, thereby getting feedscraper's IP banned, or various other accidental-or-on-purpose bad behavior. The pool isn't a complete DoS protection, but it's the bare minimum.

The interesting lines are in the app function:

```
1  pile = evy.GreenPile(pool)
2  for url in environ['wsgi.input'].readlines():
3      pile.spawn(fetch_title, url)
4  titles = '\n'.join(pile)
```

Note that in line 1, the Pile is constructed using the global pool as its argument. That ties the Pile's concurrency to the global's. If there are already 1000 concurrent fetches from other clients of feedscraper, this one will block until some of those complete. Limitations are good!

Line 3 is just a spawn, but note that we don't store any return value from it. This is because the return value is kept in the Pile itself. This becomes evident in the next line...

Line 4 is where we use the fact that the Pile is an iterator. Each element in the iterator is one of the return values from the fetch_title function, which are strings. We can use a normal Python idiom (`join()`) to concatenate these incrementally as they happen.

1.4 Greening The World

One of the challenges of writing a library like Evy is that the built-in networking libraries don't natively support the sort of cooperative yielding that we need. What we must do instead is patch standard library modules in certain key places so that they do cooperatively yield. We've in the past considered doing this automatically upon importing Evy, but have decided against that course of action because it is un-Pythonic to change the behavior of module A simply by importing module B.

Therefore, the application using Evy must explicitly green the world for itself, using one or both of the convenient methods provided.

1.4.1 Import Green

The first way of greening an application is to import networking-related libraries from the `evy.green` package. It contains libraries that have the same interfaces as common standard ones, but they are modified to behave well with green threads. Using this method is a good engineering practice, because the true dependencies are apparent in every file:

```
from evy.patched import socket
from evy.patched import threading
from evy.patched import asyncore
```

This works best if every library can be imported green in this manner. If `evy.green` lacks a module (for example, non-python-standard modules), then `import_patched()` function can come to the rescue. It is a replacement for the builtin import statement that greens any module on import.

`evy.patcher.`**`import_patched`**(*module_name*, **additional_modules*, ***kw_additional_modules*)

Imports a module in a greened manner, so that the module's use of networking libraries like socket will use Evy's green versions instead. The only required argument is the name of the module to be imported:

```
import evy
httplib2 = evy.import_patched('httplib2')
```

Under the hood, it works by temporarily swapping out the "normal" versions of the libraries in sys.modules for an evy.green equivalent. When the import of the to-be-patched module completes, the state of sys.modules is restored. Therefore, if the patched module contains the statement 'import socket', import_patched will have it reference evy.green.socket. One weakness of this approach is that it doesn't work for late binding (i.e. imports that happen during runtime). Late binding of imports is fortunately rarely done (it's slow and against PEP-8), so in most cases import_patched will work just fine.

One other aspect of import_patched is the ability to specify exactly which modules are patched. Doing so may provide a slight performance benefit since only the needed modules are imported, whereas import_patched with no arguments imports a bunch of modules in case they're needed. The *additional_modules* and *kw_additional_modules* arguments are both sequences of name/module pairs. Either or both can be used:

```
from evy.patched import socket
from evy.patched import SocketServer
BaseHTTPServer = evy.import_patched('BaseHTTPServer',
                        ('socket', socket),
                        ('SocketServer', SocketServer))
```

```
BaseHTTPServer = evy.import_patched('BaseHTTPServer',
                        socket=socket, SocketServer=SocketServer)
```

1.4.2 Monkeypatching the Standard Library

The other way of greening an application is simply to monkeypatch the standard library. This has the disadvantage of appearing quite magical, but the advantage of avoiding the late-binding problem.

`evy.patcher.`**`monkey_patch`**`(`*os=None*, *select=None*, *socket=None*, *thread=None*, *time=None*, *psycopg=None*`)`

This function monkeypatches the key system modules by replacing their key elements with green equivalents. If no arguments are specified, everything is patched:

```
import evy
evy.monkey_patch()
```

The keyword arguments afford some control over which modules are patched, in case that's important. Most patch the single module of the same name (e.g. time=True means that the time module is patched [time.sleep is patched by evy.sleep]). The exceptions to this rule are *socket*, which also patches the `ssl` module if present; and *thread*, which patches `thread`, `threading`, and `Queue`.

Here's an example of using monkey_patch to patch only a few modules:

```
import evy
evy.monkey_patch(socket=True, select=True)
```

It is important to call `monkey_patch()` as early in the lifetime of the application as possible. Try to do it as one of the first lines in the main module. The reason for this is that sometimes there is a class that inherits from a class that needs to be greened – e.g. a class that inherits from socket.socket – and inheritance is done at import time, so therefore the monkeypatching should happen before the derived class is defined. It's safe to call monkey_patch multiple times.

The psycopg monkeypatching relies on Daniele Varrazzo's green psycopg2 branch; see the announcement for more information.

`evy.patcher.`**`is_monkey_patched`**`(`*module*`)`

Returns whether or not the specified module is currently monkeypatched. *module* can either be the module itself or the module's name.

Based entirely off the name of the module, so if you import a module some other way than with the import keyword (including `import_patched()`), is_monkey_patched might not be correct about that particular module.

1.5 Examples

Here are a bunch of small example programs that use Evy. All of these examples can be found in the `examples` directory of a source copy of Evy.

1.5.1 Web Crawler

`examples/webcrawler.py`

```python
#! /usr/bin/env python
"""
This is a simple web "crawler" that fetches a bunch of urls using a pool to
control the number of outbound connections. It has as many simultaneously open
connections as coroutines in the pool.

The prints in the body of the fetch function are there to demonstrate that the
requests are truly made in parallel.
"""

urls = ["http://www.google.com/intl/en_ALL/images/logo.gif",
        "https://wiki.secondlife.com/w/images/secondlife.jpg",
        "http://us.i1.yimg.com/us.yimg.com/i/ww/beta/y3.gif"]

import evy
from evy.patched import urllib2

def fetch (url):
    print "opening", url
    body = urllib2.urlopen(url).read()
    print "done with", url
    return url, body

pool = evy.GreenPool(200)
for url, body in pool.imap(fetch, urls):
    print "got body from", url, "of length", len(body)
```

1.5.2 WSGI Server

examples/wsgi.py

```python
"""This is a simple example of running a wsgi application with evy.
For a more fully-featured server which supports multiple processes,
multiple threads, and graceful code reloading, see:

http://pypi.python.org/pypi/Spawning/
"""

import evy
from evy.web import wsgi

def hello_world (env, start_response):
    if env['PATH_INFO'] != '/':
        start_response('404 Not Found', [('Content-Type', 'text/plain')])
        return ['Not Found\r\n']
    start_response('200 OK', [('Content-Type', 'text/plain')])
    return ['Hello, World!\r\n']

wsgi.server(evy.listen(('', 8090)), hello_world)
```

1.5.3 Echo Server

examples/echoserver.py

```
#! /usr/bin/env python
"""\
Simple server that listens on port 6000 and echos back every input to
the client.  To try out the server, start it up by running this file.

Connect to it with:
  telnet localhost 6000

You terminate your connection by terminating telnet (typically Ctrl-]
and then 'quit')
"""

import evy

def handle (fd):
    print "client connected"
    while True:
        # pass through every non-eof line
        x = fd.readline()
        if not x: break
        fd.write(x)
        fd.flush()
        print "echoed", x,
    print "client disconnected"

print "server socket listening on port 6000"
server = evy.listen(('0.0.0.0', 6000))
pool = evy.GreenPool()
while True:
    try:
        new_sock, address = server.accept()
        print "accepted", address
        pool.spawn_n(handle, new_sock.makefile('rw'))
    except (SystemExit, KeyboardInterrupt):
        break
```

1.5.4 Socket Connect

examples/connect.py

```
"""Spawn multiple workers and collect their results.

Demonstrates how to use the evy.patched.socket module.
"""
import evy
from evy.patched import socket

def geturl (url):
    c = socket.socket()
    ip = socket.gethostbyname(url)
    c.connect((ip, 80))
    print '%s connected' % url
    c.sendall('GET /\r\n\r\n')
    return c.recv(1024)

urls = ['www.google.com', 'www.yandex.ru', 'www.python.org']
pile = evy.GreenPile()
```

```python
for x in urls:
    pile.spawn(geturl, x)

# note that the pile acts as a collection of return values from the functions
# if any exceptions are raised by the function they'll get raised here
for url, result in zip(urls, pile):
    print '%s: %s' % (url, repr(result)[:50])
```

1.5.5 Multi-User Chat Server

`examples/chat_server.py`

This is a little different from the echo server, in that it broadcasts the messages to all participants, not just the sender.

```python
import evy
from evy.patched import socket

PORT = 3001
participants = set()

def read_chat_forever (writer, reader):
    line = reader.readline()
    while line:
        print "Chat:", line.strip()
        for p in participants:
            try:
                if p is not writer: # Don't echo
                    p.write(line)
                    p.flush()
            except socket.error, e:
                # ignore broken pipes, they just mean the participant
                # closed its connection already
                if e[0] != 32:
                    raise
        line = reader.readline()
    participants.remove(writer)
    print "Participant left chat."

try:
    print "ChatServer starting up on port %s" % PORT
    server = evy.listen(('0.0.0.0', PORT))
    while True:
        new_connection, address = server.accept()
        print "Participant joined chat."
        new_writer = new_connection.makefile('w')
        participants.add(new_writer)
        evy.spawn_n(read_chat_forever,
                    new_writer,
                    new_connection.makefile('r'))
except (KeyboardInterrupt, SystemExit):
    print "ChatServer exiting."
```

1.5.6 Feed Scraper

`examples/feedscraper.py`

This example requires Feedparser to be installed or on the PYTHONPATH.

```
"""A simple web server that accepts POSTS containing a list of feed urls,
and returns the titles of those feeds.
"""
import evy

feedparser = evy.import_patched('feedparser')

# the pool provides a safety limit on our concurrency
pool = evy.GreenPool()

def fetch_title (url):
    d = feedparser.parse(url)
    return d.feed.get('title', '')


def app (environ, start_response):
    if environ['REQUEST_METHOD'] != 'POST':
        start_response('403 Forbidden', [])
        return []

    # the pile collects the result of a concurrent operation -- in this case,
    # the collection of feed titles
    pile = evy.GreenPile(pool)
    for line in environ['wsgi.input'].readlines():
        url = line.strip()
        if url:
            pile.spawn(fetch_title, url)
        # since the pile is an iterator over the results,
    # you can use it in all sorts of great Pythonic ways
    titles = '\n'.join(pile)
    start_response('200 OK', [('Content-type', 'text/plain')])
    return [titles]


if __name__ == '__main__':
    from evy import wsgi

    wsgi.server(evy.listen(('localhost', 9010)), app)
```

1.5.7 Port Forwarder

examples/forwarder.py

```
""" This is an incredibly simple port forwarder from port 7000 to 22 on
localhost.  It calls a callback function when the socket is closed, to
demonstrate one way that you could start to do interesting things by
starting from a simple framework like this.
"""

import evy

def closed_callback ():
    print "called back"


def forward (source, dest, cb = lambda: None):
    """Forwards bytes unidirectionally from source to dest"""
    while True:
        d = source.recv(32384)
        if d == '':
            cb()
            break
        dest.sendall(d)

listener = evy.listen(('localhost', 7000))
while True:
    client, addr = listener.accept()
    server = evy.connect(('localhost', 22))
    # two unidirectional forwarders make a bidirectional one
    evy.spawn_n(forward, client, server, closed_callback)
    evy.spawn_n(forward, server, client)
```

1.5.8 Recursive Web Crawler

`examples/recursive_crawler.py`

This is an example recursive web crawler that fetches linked pages from a seed url.

```
"""This is a recursive web crawler.  Don't go pointing this at random sites;
it doesn't respect robots.txt and it is pretty brutal about how quickly it
fetches pages.

The code for this is very short; this is perhaps a good indication
that this is making the most effective use of the primitves at hand.
The fetch function does all the work of making http requests,
searching for new urls, and dispatching new fetches.  The GreenPool
acts as sort of a job coordinator (and concurrency controller of
course).
"""
from __future__ import with_statement

from evy.patched import urllib2
import evy
import re

# http://daringfireball.net/2009/11/liberal_regex_for_matching_urls
url_regex = re.compile(r'\b(([\w-]+://?|www[.])[^\s()<>]+(?:\([\w\d]+\)|([^[:punct:]\s]|/)))')


def fetch (url, seen, pool):
    """Fetch a url, stick any found urls into the seen set, and
    dispatch any new ones to the pool."""
    print "fetching", url
    data = ''
    with evy.Timeout(5, False):
        data = urllib2.urlopen(url).read()
    for url_match in url_regex.finditer(data):
        new_url = url_match.group(0)
        # only send requests to evy.net so as not to destroy the internet
        if new_url not in seen and 'evy.net' in new_url:
            seen.add(new_url)
            # while this seems stack-recursive, it's actually not:
            # spawned greenthreads start their own stacks
            pool.spawn_n(fetch, new_url, seen, pool)


def crawl (start_url):
    """Recursively crawl starting from *start_url*.  Returns a set of
    urls that were found."""
    pool = evy.GreenPool()
    seen = set()
    fetch(start_url, seen, pool)
    pool.waitall()
    return seen

seen = crawl("http://evy.net")
print "I saw these urls:"
print "\n".join(seen)
```

1.5.9 Producer Consumer Web Crawler

`examples/producer_consumer.py`

This is an example implementation of the producer/consumer pattern as well as being identical in functionality to the recursive web crawler.

```
"""This is a recursive web crawler.  Don't go pointing this at random sites;
it doesn't respect robots.txt and it is pretty brutal about how quickly it
fetches pages.

This is a kind of "producer/consumer" example; the fetch function produces
jobs, and the GreenPool itself is the consumer, farming out work concurrently.
It's easier to write it this way rather than writing a standard consumer loop;
GreenPool handles any exceptions raised and arranges so that there's a set
number of "workers", so you don't have to write that tedious management code
yourself.
"""
from __future__ import with_statement

from evy.patched import urllib2
import evy
import re

# http://daringfireball.net/2009/11/liberal_regex_for_matching_urls
url_regex = re.compile(r'\b(([\w-]+://?|www[.])[^\s()<>]+(?:\([\w\d]+\)|([^[:punct:]\s]|/)))')


def fetch (url, outq):
    """Fetch a url and push any urls found into a queue."""
    print "fetching", url
    data = ''
    with evy.Timeout(5, False):
        data = urllib2.urlopen(url).read()
    for url_match in url_regex.finditer(data):
        new_url = url_match.group(0)
        outq.put(new_url)


def producer (start_url):
    """Recursively crawl starting from *start_url*.  Returns a set of
    urls that were found."""
    pool = evy.GreenPool()
    seen = set()
    q = evy.Queue()
    q.put(start_url)
    # keep looping if there are new urls, or workers that may produce more urls
    while True:
        while not q.empty():
            url = q.get()
            # limit requests to evy.net so we don't crash all over the internet
            if url not in seen and 'evy.net' in url:
                seen.add(url)
                pool.spawn_n(fetch, url, q)
        pool.waitall()
        if q.empty():
            break

    return seen


seen = producer("http://evy.net")
print "I saw these urls:"
print "\n".join(seen)
```

1.5.10 Websocket Server Example

`examples/websocket.py`

This exercises some of the features of the websocket server implementation.

```
import evy

from evy.web import wsgi, websocket

# demo app
import os
import random

@websocket.WebSocketWSGI
def handle (ws):
    """  This is the websocket handler function.  Note that we
    can dispatch based on path in here, too."""
    if ws.path == '/echo':
        while True:
            m = ws.wait()
            if m is None:
                break
            ws.send(m)

    elif ws.path == '/data':
        for i in xrange(10000):
            ws.send("0 %s %s\n" % (i, random.random()))
            evy.sleep(0.1)


def dispatch (environ, start_response):
    """ This resolves to the web page or the websocket depending on
    the path."""
    if environ['PATH_INFO'] == '/data':
        return handle(environ, start_response)
    else:
        start_response('200 OK', [('content-type', 'text/html')])
        return [open(os.path.join(
            os.path.dirname(__file__),
            'websocket.html')).read()]

if __name__ == "__main__":
    # run an example app from the command line
    listener = evy.listen(('127.0.0.1', 7000))
    print "\nVisit http://localhost:7000/ in your websocket-capable browser.\n"
    wsgi.server(listener, dispatch)
```

1.5.11 Websocket Multi-User Chat Example

`examples/websocket_chat.py`

This is a mashup of the websocket example and the multi-user chat example, showing how you can do the same sorts of things with websockets that you can do with regular sockets.

```
import os

import evy
from evy.web import wsgi, websocket

PORT = 7000

participants = set()

@websocket.WebSocketWSGI
def handle (ws):
    participants.add(ws)
    try:
        while True:
            m = ws.wait()
            if m is None:
                break
            for p in participants:
                p.send(m)
    finally:
        participants.remove(ws)


def dispatch (environ, start_response):
    """Resolves to the web page or the websocket depending on the path."""
    if environ['PATH_INFO'] == '/chat':
        return handle(environ, start_response)
    else:
        start_response('200 OK', [('content-type', 'text/html')])
        html_path = os.path.join(os.path.dirname(__file__), 'websocket_chat.html')
        return [open(html_path).read() % {'port': PORT}]

if __name__ == "__main__":
    # run an example app from the command line
    listener = evy.listen(('127.0.0.1', PORT))
    print "\nVisit http://localhost:7000/ in your websocket-capable browser.\n"
    wsgi.server(listener, dispatch)
```

1.6 Using SSL With Evy

Evy makes it easy to use non-blocking SSL sockets. If you're using Python 2.6 or later, you're all set, evy wraps the built-in ssl module. If on Python 2.5 or 2.4, you have to install pyOpenSSL to use evy.

In either case, the `patched` modules handle SSL sockets transparently, just like their standard counterparts. As an example, `evy.patched.urllib2` can be used to fetch https urls in as non-blocking a fashion as you please:

```
from evy.patched import urllib2
from evy.green.threads import spawn
bodies = [spawn(urllib2.urlopen, url)
     for url in ("https://secondlife.com","https://google.com")]
for b in bodies:
    print b.wait().read()
```

1.6.1 With Python 2.6

To use ssl sockets directly in Python 2.6, use `evy.patched.ssl`, which is a non-blocking wrapper around the standard Python `ssl` module, and which has the same interface. See the standard documentation for instructions on use.

1.6.2 PyOpenSSL

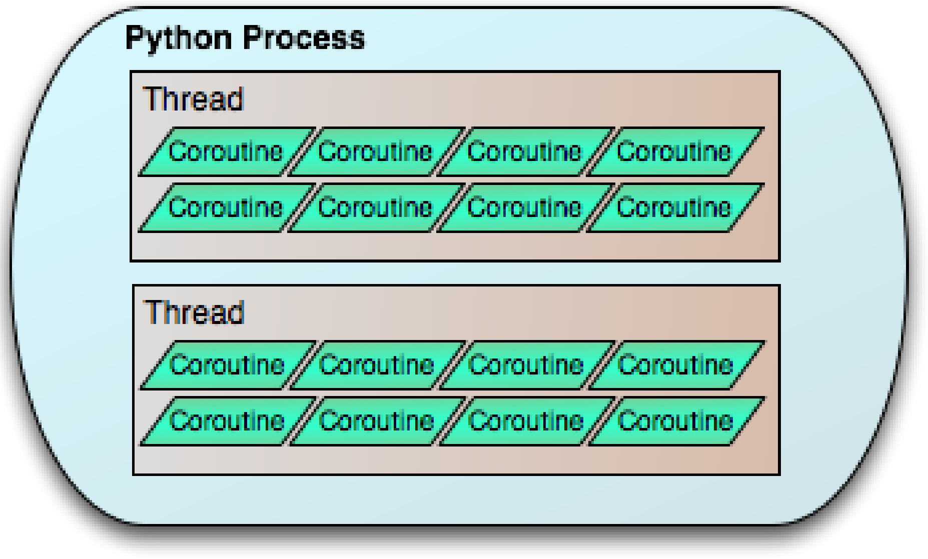
`evy.patched.OpenSSL` has exactly the same interface as pyOpenSSL (docs), and works in all versions of Python. This module is much more powerful than `socket.ssl()`, and may have some advantages over `ssl`, depending on your needs.

Here's an example of a server:

```
from evy.patched import socket
from evy.patched.OpenSSL import SSL

# insecure context, only for example purposes
context = SSL.Context(SSL.SSLv23_METHOD)
context.set_verify(SSL.VERIFY_NONE, lambda *x: True))

# create underlying green socket and wrap it in ssl
sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
connection = SSL.Connection(context, sock)

# configure as server
connection.set_accept_state()
connection.bind(('127.0.0.1', 80443))
connection.listen(50)

# accept one client connection then close up shop
client_conn, addr = connection.accept()
print client_conn.read(100)
client_conn.shutdown()
client_conn.close()
connection.close()
```

1.7 Threads

Evy is thread-safe and can be used in conjunction with normal Python threads. The way this works is that coroutines are confined to their 'parent' Python thread. It's like each thread contains its own little world of coroutines that can switch between themselves but not between coroutines in other threads.



You can only communicate cross-thread using the "real" thread primitives and pipes. Fortunately, there's little reason to use threads for concurrency when you're already using coroutines.

The vast majority of the times you'll want to use threads are to wrap some operation that is not "green", such as a C library that uses its own OS calls to do socket operations. The `tpool` module is provided to make these uses simpler.

The optional *pyevent hub* is not compatible with threads.

1.7.1 Tpool - Simple thread pool

The simplest thing to do with `tpool` is to `execute()` a function with it. The function will be run in a random thread in the pool, while the calling coroutine blocks on its completion:

```
>>> import thread
>>> from evy import tpool
>>> def my_func(starting_ident):
...     print "running in new thread:", starting_ident != thread.get_ident()
...
>>> tpool.execute(my_func, thread.get_ident())
running in new thread: True
```

By default there are 20 threads in the pool, but you can configure this by setting the environment variable `EVY_THREADPOOL_SIZE` to the desired pool size before importing tpool.

1.8 Zeromq

1.8.1 What is ØMQ?

"A ØMQ socket is what you get when you take a normal TCP socket, inject it with a mix of radioactive isotopes stolen from a secret Soviet atomic research project, bombard it with 1950-era cosmic rays, and put it into the hands of a drug-addled comic book author with a badly-disguised fetish for bulging muscles clad in spandex."

Key differences to conventional sockets Generally speaking, conventional sockets present a synchronous interface to either connection-oriented reliable byte streams (SOCK_STREAM), or connection-less unreliable datagrams (SOCK_DGRAM). In comparison, 0MQ sockets present an abstraction of an asynchronous message queue, with the exact queueing semantics depending on the socket type in use. Where conventional sockets transfer streams of bytes or discrete datagrams, 0MQ sockets transfer discrete messages.

0MQ sockets being asynchronous means that the timings of the physical connection setup and teardown, reconnect and effective delivery are transparent to the user and organized by 0MQ itself. Further, messages may be queued in the event that a peer is unavailable to receive them.

Conventional sockets allow only strict one-to-one (two peers), many-to-one (many clients, one server), or in some cases one-to-many (multicast) relationships. With the exception of ZMQ::PAIR, 0MQ sockets may be connected to multiple endpoints using connect(), while simultaneously accepting incoming connections from multiple endpoints bound to the socket using bind(), thus allowing many-to-many relationships.

1.8.2 API documentation

ØMQ support is provided in the `evy.patched.zmq` module

1.9 Understanding Evy Hubs

A hub forms the basis of Evy's event loop, which dispatches I/O events and schedules greenthreads. It is the existence of the hub that promotes coroutines (which can be tricky to program with) into greenthreads (which are easy).

Evy has only one hub implementation, the libuv one, and when you start using it.

1.9.1 How the Hubs Work

The hub has a main greenlet, MAINLOOP. When one of the running coroutines needs to do some I/O, it registers a listener with the hub (so that the hub knows when to wake it up again), and then switches to MAINLOOP (via `get_hub().switch()`). If there are other coroutines that are ready to run, MAINLOOP switches to them, and when they complete or need to do more I/O, they switch back to the MAINLOOP. In this manner, MAINLOOP ensures that every coroutine gets scheduled when it has some work to do.

MAINLOOP is launched only when the first I/O operation happens, and it is not the same greenlet that __main__ is running in. This lazy launching is why it's not necessary to explicitly call a dispatch() method like other frameworks, which in turn means that code can start using Evy without needing to be substantially restructured.

1.9.2 More Hub-Related Functions

1.10 Testing Evy

Evy is tested using Nose. To run tests, simply install nose, and then, in the evy tree, do:

```
$ python setup.py test
```

If you want access to all the nose plugins via command line, you can run:

```
$ python setup.py nosetests
```

Lastly, you can just use nose directly if you want:

```
$ nosetests
```

That's it! The output from running nose is the same as unittest's output, if the entire directory was one big test file.

Many tests are skipped based on environmental factors. These are printed as S's during execution, and in the summary printed after the tests run it will tell you how many were skipped.

1.10.1 Doctests

To run the doctests included in many of the evy modules, use this command:

```
$ nosetests --with-doctest evy/*.py
```

Currently there are 16 doctests.

1.10.2 Standard Library Tests

Evy provides for the ability to test itself with the standard Python networking tests. This verifies that the libraries it wraps work at least as well as the standard ones do. The directory tests/stdlib contains a bunch of stubs that import the standard lib tests from your system and run them. If you do not have any tests in your python distribution, they'll simply fail to import.

There's a convenience module called all.py designed to handle the impedance mismatch between Nose and the standard tests:

```
$ nosetests tests/stdlib/all.py
```

That will run all the tests, though the output will be a little weird because it will look like Nose is running about 20 tests, each of which consists of a bunch of sub-tests. Not all test modules are present in all versions of Python, so there will be an occasional printout of "Not importing %s, it doesn't exist in this installation/version of Python".

If you see "Ran 0 tests in 0.001s", it means that your Python installation lacks its own tests. This is usually the case for Linux distributions. One way to get the missing tests is to download a source tarball (of the same version you have installed on your system!) and copy its Lib/test directory into the correct place on your PYTHONPATH.

1.10.3 Writing Tests

What follows are some notes on writing tests, in no particular order.

The filename convention when writing a test for module *foo* is to name the test *test_foo.py*. We don't yet have a convention for tests that are of finer granularity, but a sensible one might be *test_foo_class.py*.

If you are writing a test that involves a client connecting to a spawned server, it is best to not use a hardcoded port because that makes it harder to parallelize tests. Instead bind the server to 0, and then look up its port when connecting the client, like this:

```
from evy.io.convenience import listener, connect
server_sock = listener(('127.0.0.1', 0))
client_sock = connect(('localhost', server_sock.getsockname()[1]))
```

1.10.4 Coverage

Coverage.py is an awesome tool for evaluating how much code was exercised by unit tests. Nose supports it if both are installed, so it's easy to generate coverage reports for evy. Here's how:

```
nosetests --with-coverage --cover-package=evy
```

After running the tests to completion, this will emit a huge wodge of module names and line numbers. For some reason, the `--cover-inclusive` option breaks everything rather than serving its purpose of limiting the coverage to the local files, so don't use that.

The html option is quite useful because it generates nicely-formatted HTML that are much easier to read than line-number soup. Here's a command that generates the annotation, dumping the html files into a directory called "cover":

```
coverage html -d cover --omit='tempmod,<console>,tests'
```

(`tempmod` and `console` are omitted because they gets thrown away at the completion of their unit tests and coverage.py isn't smart enough to detect this.)

1.11 Environment Variables

Evy's behavior can be controlled by a few environment variables. These are only for the advanced user.

EVY_THREADPOOL_SIZE

The size of the threadpool in `tpool`. This is an environment variable because tpool constructs its pool on first use, so any control of the pool size needs to happen before then.

1.12 Module Reference

1.12.1 `backdoor` – Python interactive interpreter within a running process

The backdoor module is convenient for inspecting the state of a long-running process. It supplies the normal Python interactive interpreter in a way that does not block the normal operation of the application. This can be useful for debugging, performance tuning, or simply learning about how things behave in situ.

In the application, spawn a greenthread running backdoor_server on a listening socket:

```
evy.spawn(backdoor.backdoor_server, evy.listen(('localhost', 3000)))
```

When this is running, the backdoor is accessible via telnet to the specified port.

```
$ telnet localhost 3000
Python 2.6.2 (r262:71600, Apr 16 2009, 09:17:39)
[GCC 4.0.1 (Apple Computer, Inc. build 5250)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> import myapp
>>> dir(myapp)
['__all__', '__doc__', '__name__', 'myfunc']
>>>
```

The backdoor cooperatively yields to the rest of the application between commands, so on a running server continuously serving requests, you can observe the internal state changing between interpreter commands.

1.12.2 `corolocal` – Coroutine local storage

1.12.3 `debug` – Debugging tools for Evy

The debug module contains utilities and functions for better debugging Evy-powered applications.

`evy.tools.debug.`**`spew`**`(`*trace_names=None*, *show_values=False*`)`
Install a trace hook which writes incredibly detailed logs about what code is being executed to stdout.

`evy.tools.debug.`**`unspew`**`()`
Remove the trace hook installed by spew.

`evy.tools.debug.`**`format_hub_listeners`**`()`
Returns a formatted string of the current listeners on the current hub. This can be useful in determining what's going on in the event system, especially when used in conjunction with `hub_listener_stacks()`.

`evy.tools.debug.`**`format_hub_timers`**`()`
Returns a formatted string of the current timers on the current hub. This can be useful in determining what's going on in the event system, especially when used in conjunction with `hub_timer_stacks()`.

`evy.tools.debug.`**`hub_listener_stacks`**`(`*state=False*`)`
Toggles whether or not the hub records the stack when clients register listeners on file descriptors. This can be useful when trying to figure out what the hub is up to at any given moment. To inspect the stacks of the current listeners, call `format_hub_listeners()` at critical junctures in the application logic.

`evy.tools.debug.`**`hub_exceptions`**`(`*state=True*`)`
Toggles whether the hub prints exceptions that are raised from its timers. This can be useful to see how greenthreads are terminating.

`evy.tools.debug.`**`tpool_exceptions`**`(`*state=False*`)`
Toggles whether tpool itself prints exceptions that are raised from functions that are executed in it, in addition to raising them like it normally does.

`evy.tools.debug.`**`hub_timer_stacks`**`(`*state=False*`)`
Toggles whether or not the hub records the stack when timers are set. To inspect the stacks of the current timers, call `format_hub_timers()` at critical junctures in the application logic.

`evy.tools.debug.`**`hub_blocking_detection`**`(`*state=False*, *resolution=1*`)`
Toggles whether Evy makes an effort to detect blocking behavior in an application.

It does this by telling the kernel to raise a SIGALARM after a short timeout, and clearing the timeout every time the hub greenlet is resumed. Therefore, any code that runs for a long time without yielding to the hub will get interrupted by the blocking detector (don't use it in production!).

The *resolution* argument governs how long the SIGALARM timeout waits in seconds. If on Python 2.6 or later, the implementation uses `signal.setitimer()` and can be specified as a floating-point value. On 2.5 or earlier, 1 second is the minimum. The shorter the resolution, the greater the chance of false positives.

1.12.4 `db_pool` – DBAPI 2 database connection pooling

The db_pool module is useful for managing database connections. It provides three primary benefits: cooperative yielding during database operations, concurrency limiting to a database host, and connection reuse. db_pool is intended to be database-agnostic, compatible with any DB-API 2.0 database module.

It has currently been tested and used with both MySQLdb and psycopg2.

A ConnectionPool object represents a pool of connections open to a particular database. The arguments to the constructor include the database-software-specific module, the host name, and the credentials required for authentication. After construction, the ConnectionPool object decides when to create and sever connections with the target database.

```
>>> import MySQLdb
>>> cp = ConnectionPool(MySQLdb, host='localhost', user='root', passwd='')
```

Once you have this pool object, you connect to the database by calling `get()` on it:

```
>>> conn = cp.get()
```

This call may either create a new connection, or reuse an existing open connection, depending on whether it has one open already or not. You can then use the connection object as normal. When done, you must return the connection to the pool:

```
>>> conn = cp.get()
>>> try:
...     result = conn.cursor().execute('SELECT NOW()')
... finally:
...     cp.put(conn)
```

After you've returned a connection object to the pool, it becomes useless and will raise exceptions if any of its methods are called.

Constructor Arguments

In addition to the database credentials, there are a bunch of keyword constructor arguments to the ConnectionPool that are useful.

- min_size, max_size : The normal Pool arguments. max_size is the most

important constructor argument – it determines the number of concurrent connections can be open to the destination database. min_size is not very useful. * max_idle : Connections are only allowed to remain unused in the pool for a limited amount of time. An asynchronous timer periodically wakes up and closes any connections in the pool that have been idle for longer than they are supposed to be. Without this parameter, the pool would tend to have a 'high-water mark', where the number of connections open at a given time corresponds to the peak historical demand. This number only has effect on the connections in the pool itself – if you take a connection out of the pool, you can hold on to it for as long as you want. If this is set to 0, every connection is closed upon its return to the pool. * max_age : The lifespan of a connection. This works much like max_idle, but the timer is measured from the connection's creation time, and is tracked throughout the connection's life. This means that if you take a connection out of the pool and hold on to it for some lengthy operation that exceeds max_age, upon putting the connection back in to the pool, it will be closed. Like max_idle, max_age will not close connections that are taken out of the pool, and, if set to 0, will cause every connection to be closed when put back in the pool. * connect_timeout : How long to wait before raising an exception on connect(). If the database module's connect() method takes too long, it raises a ConnectTimeout exception from the get() method on the pool.

DatabaseConnector

If you want to connect to multiple databases easily (and who doesn't), the DatabaseConnector is for you. It's a pool of pools, containing a ConnectionPool for every host you connect to.

The constructor arguments are:

- module : database module, e.g. MySQLdb. This is simply passed through to the ConnectionPool.
- credentials : A dictionary, or dictionary-alike, mapping hostname to connection-argument-dictionary. This is used for the constructors of the ConnectionPool objects. Example:

```
>>> dc = DatabaseConnector(MySQLdb,
...      {'db.internal.example.com': {'user': 'internal', 'passwd': 's33kr1t'},
...       'localhost': {'user': 'root', 'passwd': ''}})
```

If the credentials contain a host named 'default', then the value for 'default' is used whenever trying to connect to a host that has no explicit entry in the database. This is useful if there is some pool of hosts that share arguments.

- conn_pool : The connection pool class to use. Defaults to db_pool.ConnectionPool.

The rest of the arguments to the DatabaseConnector constructor are passed on to the ConnectionPool.

Caveat: The DatabaseConnector is a bit unfinished, it only suits a subset of use cases.

1.12.5 `event` – Cross-greenthread primitive

1.12.6 `greenpool` – Green Thread Pools

1.12.7 `greenthread` – Green Thread Implementation

1.12.8 `hubs` - The events hubs

1.12.9 `pools` - Generic pools of resources

1.12.10 `queue` – Queue class

1.12.11 `semaphore` – Semaphore classes

1.12.12 `timeout` – Universal Timeouts

class `evy.timeout.`**`Timeout`**

Raises *exception* in the current greenthread after *timeout* seconds:

```
timeout = Timeout(seconds, exception)
try:
    ... # execution here is limited by timeout
finally:
    timeout.cancel()
```

When *exception* is omitted or `None`, the `Timeout` instance itself is raised:

```
>>> Timeout(0.1)
>>> evy.sleep(0.2)
Traceback (most recent call last):
 ...
Timeout: 0.1 seconds
Traceback (most recent call last):
 ...
Timeout: 0.1 seconds
```

In Python 2.5 and newer, you can use the `with` statement for additional convenience:

```
with Timeout(seconds, exception) as timeout:
    pass # ... code block ...
```

This is equivalent to the try/finally block in the first example.

There is an additional feature when using the `with` statement: if *exception* is `False`, the timeout is still raised, but the with statement suppresses it, so the code outside the with-block won't see it:

```
data = None
with Timeout(5, False):
    data = mysock.makefile().readline()
if data is None:
    ... # 5 seconds passed without reading a line
else:
    ... # a line was read within 5 seconds
```

As a very special case, if *seconds* is None, the timer is not scheduled, and is only useful if you're planning to raise it directly.

There are two Timeout caveats to be aware of:

- If the code block in the try/finally or with-block never cooperatively yields, the timeout cannot be raised. In Evy, this should rarely be a problem, but be aware that you cannot time out CPU-only operations with this class.
- If the code block catches and doesn't re-raise `BaseException` (for example, with `except:`), then it will catch the Timeout exception, and might not abort as intended.

When catching timeouts, keep in mind that the one you catch may not be the one you have set; if you going to silence a timeout, always check that it's the same instance that you set:

```
timeout = Timeout(1)
try:
    ...
except Timeout, t:
    if t is not timeout:
        raise # not my timeout
```

`evy.timeout.`**`with_timeout`**`(`*seconds*, *function*, **args*, ***kwds*`)`

Wrap a call to some (yielding) function with a timeout; if the called function fails to return before the timeout, cancel it and return a flag value.

Parameters

- **seconds** (*int or float*) – seconds before timeout occurs
- **func** – the callable to execute with a timeout; it must cooperatively yield, or else the timeout will not be able to trigger
- ***args** – positional arguments to pass to *func*
- ****kwds** – keyword arguments to pass to *func*
- **timeout_value** – value to return if timeout occurs (by default raises `Timeout`)

Return type Value returned by *func* if *func* returns before *seconds*, else *timeout_value* if provided, else raises `Timeout`.

Raises Timeout if *func* times out and no `timeout_value` has been provided.

Exception Any exception raised by *func*

Example:

```
data = with_timeout(30, urllib2.open, 'http://www.google.com/', timeout_value="")
```

Here *data* is either the result of the `get()` call, or the empty string if it took too long to return. Any exception raised by the `get()` call is passed through to the caller.

1.12.13 `websocket` – Websocket Server

This module provides a simple way to create a websocket server. It works with a few tweaks in the `wsgi` module that allow websockets to coexist with other WSGI applications.

To create a websocket server, simply decorate a handler method with `WebSocketWSGI` and use it as a wsgi application:

```
from evy.web import wsgi, websocket
import evy

@websocket.WebSocketWSGI
def hello_world(ws):
    ws.send("hello world")

wsgi.server(evy.listen((’’, 8090)), hello_world)
```

You can find a slightly more elaborate version of this code in the file `examples/websocket.py`.

As of version 0.9.13, evy.websocket supports SSL websockets; all that’s necessary is to use an *SSL wsgi server*.

Note: The web socket spec is still under development, and it will be necessary to change the way that this module works in response to spec changes.

1.12.14 `wsgi` – WSGI server

The wsgi module provides a simple and easy way to start an event-driven WSGI server. This can serve as an embedded web server in an application, or as the basis for a more full-featured web server package. One such package is Spawning.

To launch a wsgi server, simply create a socket and call `evy.wsgi.server()` with it:

```
from evy.web import wsgi
import evy

def hello_world(env, start_response):
    start_response(’200 OK’, [(’Content-Type’, ’text/plain’)])
    return [’Hello, World!\r\n’]

wsgi.server(evy.listen((’’, 8090)), hello_world)
```

You can find a slightly more elaborate version of this code in the file `examples/wsgi.py`.

SSL

Creating a secure server is only slightly more involved than the base example. All that’s needed is to pass an SSL-wrapped socket to the `server()` method:

```
wsgi.server(evy.wrap_ssl(evy.listen((’’, 8090)),
                              certfile=’cert.crt’,
                              keyfile=’private.key’,
                              server_side=True),
            hello_world)
```

Applications can detect whether they are inside a secure server by the value of the `env[’wsgi.url_scheme’]` environment variable.

Non-Standard Extension to Support Post Hooks

Evy's WSGI server supports a non-standard extension to the WSGI specification where `env['evy.posthooks']` contains an array of *post hooks* that will be called after fully sending a response. Each post hook is a tuple of `(func, args, kwargs)` and the *func* will be called with the WSGI environment dictionary, followed by the *args* and then the *kwargs* in the post hook.

For example:

```
from evy.web import wsgi
import evy

def hook(env, arg1, arg2, kwarg3=None, kwarg4=None):
    print 'Hook called: %s %s %s %s %s' % (env, arg1, arg2, kwarg3, kwarg4)

def hello_world(env, start_response):
    env['evy.posthooks'].append(
        (hook, ('arg1', 'arg2'), {'kwarg3': 3, 'kwarg4': 4}))
    start_response('200 OK', [('Content-Type', 'text/plain')])
    return ['Hello, World!\r\n']

wsgi.server(evy.listen(('', 8090)), hello_world)
```

The above code will print the WSGI environment and the other passed function arguments for every request processed.

Post hooks are useful when code needs to be executed after a response has been fully sent to the client (or when the client disconnects early). One example is for more accurate logging of bandwidth used, as client disconnects use less bandwidth than the actual Content-Length.

1.12.15 `evy.patched.zmq` – ØMQ support

1.12.16 `zmq` – The pyzmq ØMQ python bindings

`pyzmq` [1] Is a python binding to the C++ ØMQ [2] library written in Cython [3]. The following is auto generated `pyzmq's` from documentation.

1.13 Authors

1.13.1 Evy

Alvaro Saurin <alvaro.saurin@gmail.com>

1.13.2 From Eventlet and others

- Ryan Williams, rdw on Freenode, breath@alum.mit.edu
- Bob Ippolito
- Donovan Preston
- AG Projects

[1] http://github.com/zeromq/pyzmq
[2] http://www.zeromq.com
[3] http://www.cython.org

- Chris AtLee
- R. Tyler Ballance
- Denis Bilenko
- Mike Barton
- Patrick Carlisle
- Ben Ford
- Andrew Godwin
- Brantley Harris
- Gregory Holt
- Joe Malicki
- Chet Murthy
- Eugene Oden
- radix
- Scott Robinson
- Tavis Rudd
- Sergey Shepelev
- Chuck Thier
- Nick V
- Daniele Varrazzo
- Ryan Williams
- John Beisley
- Tess Chu
- Nat Goodspeed
- Dave Kaprielian
- Kartic Krishnamurthy
- Bryan O'Sullivan
- Kent Quirk
- Ryan Williams

1.14 History

Eventlet began life as Donovan Preston was talking to Bob Ippolito about coroutine-based non-blocking networking frameworks in Python. Most non-blocking frameworks require you to run the "main loop" in order to perform all network operations, but Donovan wondered if a library written using a trampolining style could get away with transparently running the main loop any time i/o was required, stopping the main loop once no more i/o was scheduled. Bob spent a few days during PyCon 2006 writing a proof-of-concept. He named it eventlet, after the coroutine implementation it used, greenlet. Donovan began using eveventlet as a light-weight network library for his spare-time project Pavel, and also began writing some unittests.

- http://svn.red-bean.com/bob/eventlet/trunk/

When Donovan started at Linden Lab in May of 2006, he added evy as an svn external in the `indra/lib/python directory`, to be a dependency of the yet-to-be-named backbone project (at the time, it was named restserv). However, including Eventlet as an svn external meant that any time the externally hosted project had hosting issues, Linden developers were not able to perform svn updates. Thus, the Eventlet source was imported into the linden source tree at the same location, and became a fork.

Bob Ippolito has ceased working on Eventlet and has stated his desire for Linden to take it's fork forward to the open source world as "the" Eventlet.

1.15 License

Evy is made available under the terms of the open source MIT license

CHAPTER 2

Indices and tables

- *genindex*
- *modindex*
- *search*

Python Module Index

Index