Eventlet

Eventlet is an easy to use networking library written in Python. Eventlet is capable of supporting a large number of sockets per process by using nonblocking I/O, cooperatively multiplexing them through a single main loop. This approach allows for the implementation of massively concurrent servers which would require prohibitive amounts of memory under a traditional preemptive multi-threading or multi-process model. However, nonblocking I/O libraries such as asyncore or twisted can be difficult for programmers to use because they require the use of continuation passing style. This means code must be broken up into functions which initiate operations, and functions which will be called when the operation is complete, known as "callbacks."

Eventlet avoids these difficulties by using a coroutine library called greenlet. Coroutines allow Eventlet to cooperatively reenter the main loop whenever an I/O operation is initiated, switching back to the original coroutine only when the operating system indicates the operation has completed. This means code written using Eventlet looks just like code written using the traditional multi-threading or multi-process model, while avoiding the locking problems associated with preemption and requiring very little memory.

```
from eventlet import api

participants = [ ]

def read_chat_forever(writer, reader):
    line = reader.readline()
    while line:
        print "Chat:", line.strip()
        for p in participants:
            if p is not writer: # Don’t echo
                p.write(line)
        line = reader.readline()
    participants.remove(writer)
    print "Participant left chat."

try:
    print "ChatServer starting up on port 3000"
    server = api.tcp_listener(('0.0.0.0', 3000))
    while True:
        new_connection, address = server.accept()
        print "Participant joined chat.
```

Listing 1: chatserver.py

Example Chat Server

Let's look at a simple example: a chat server.

The server shown in *Listing 1* is very easy to understand. If it was written using Python’s threading module instead of eventlet, the control flow and code layout would be exactly the same. The call to api.tcp_listener would be replaced with the appropriate calls to Python’s built-in socket module, and the call to api.spawn would be replaced with the appropriate call to the thread module. However, if implemented using the thread module, each new connection would require the operating system to allocate another 8 MB stack, meaning this simple program would consume all of the RAM on a machine with 1 GB of memory with only 128 users connected, without even taking into account memory used by any objects on the heap! Using eventlet, this simple program should be able to accommodate thousands and thousands of simultaneous users, consuming very little RAM and very little CPU.

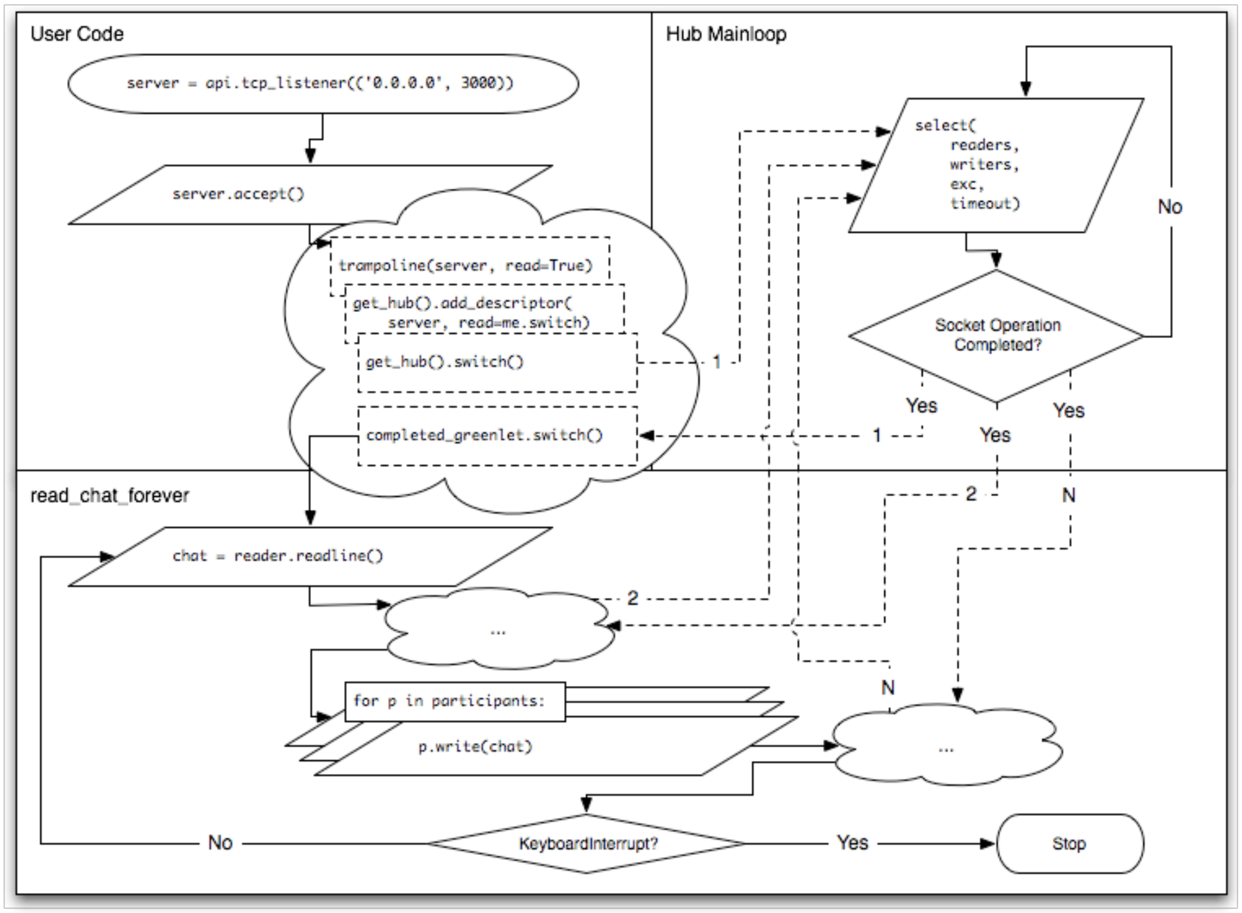

Figure 1: chatserver.py Flowchart

What sort of servers would require concurrency like this? A typical Web server might measure traffic on the order of 10 requests per second; at any given moment, the server might only have a handful of HTTP connections open simultaneously. However, a chat server, instant messenger server, or multiplayer game server will need to maintain one connection per connected user to be able to send messages to them as other users chat or make moves in the game. Also, as advanced Web development techniques such as Ajax, Ajax polling, and Comet (the “Long Poll”) become more popular, Web servers will need to be able to deal with many more simultaneous requests. In fact, since the Comet technique involves the client making a new request as soon as the server closes an old one, a Web server servicing Comet clients has the same characteristics as a chat or game server: one connection per connected user.

Basic usage

Most of the APIs required for basic eventlet usage are exported by the eventlet.api module. We have already seen two of these in listing one: api.tcp_listener, for creating a TCP server socket, and api.spawn, for spawning a new coroutine and executing multiple blocks of code conceptually in parallel. There are only a few more basic APIs: connect_tcp for creating a TCP client socket, ssl_listener and connect_ssl for creating encrypted SSL sockets, and sleep, call_after, and exc_after to arrange for code to be called after a delay. Let's look at these in detail.

spawn(function, *args, **keyword)

Create a new coroutine, or cooperative thread of control, within which to execute function. The function will be called with the given args and keyword arguments and will remain in control unless it cooperatively yields by calling a socket method or sleep. spawn returns control to the caller immediately, and function will be called in a future main loop iteration.

sleep(time)

Yield control to another eligible coroutine until at least time seconds have elapsed. time may be specified as an integer, or a float if fractional seconds are desired. Calling sleep with a time of 0 is the canonical way of expressing a cooperative yield. For example, if one is looping over a large list performing an expensive calculation without calling any socket methods, it's a good idea to call sleep(0) occasionally; otherwise nothing else will run.

call_after(time, function, *args, **keyword)

Schedule function to be called after time seconds have elapsed. time may be specified as an integer, or a float if fractional seconds are desired. The function will be called with the given args and keyword arguments, and will be executed within the main loop's coroutine.

exc_after(time, exception_object)

Schedule exception_object to be raised into the current coroutine after time seconds have elapsed. This only works if the current coroutine is yielding, and

```
from eventlet import api, httpc

def read_with_timeout():
    cancel = api.exc_after(30, api.TimeoutError())
    try:
        httpc.get('http://www.google.com/')
    except api.TimeoutError:
        print "Timed out!"
    else:
        cancel.cancel()
```

Listing 2: exc_after.py

is generally used to set timeouts after which a network operation or series of operations will be canceled. Returns a timer object with a cancel method which should be used to prevent the exception if the operation completes successfully. See *Listing 2*.

named(name)

Return an object given its dotted module path, name. For example, passing the string “os.path.join” will return the join function object from the os.path module, “eventlet.api” will return the api module object from the eventlet package, and “mulib.mu.Resource” will return the Resource class from the mulib.mu module.

Socket Functions

Eventlet’s socket objects have the same interface as the standard library socket.socket object, except they will automatically cooperatively yield control to other eligible coroutines instead of blocking. Eventlet also has the ability to monkey patch the standard library socket.socket object so that code which uses it will also automatically cooperatively yield; see *Using the Standard Library with Eventlet*.

tcp_listener(address)

Listen on the given address, a tuple of (ip, port), with a TCP socket. Returns a socket object on which one should call accept() to accept a connection on the newly bound socket.

connect_tcp(address)

Create a TCP connection to address, a tuple of (ip, port), and return the socket.

ssl_listener(address, certificate, private_key)

Listen on the given address, a tuple of (ip, port), with a TCP socket that can do SSL. certificate and private_key should be the filename of the appropriate certificate and private key files to use with the SSL socket.

connect_ssl(address)

TODO: Not implemented yet. The standard library method of wrapping a socket.socket object with a socket.ssl object works, but see *Using the Standard Library with Eventlet* to learn about using wrap_socket_with_coroutine_socket.

Using the Standard Library with Eventlet

Eventlet’s socket object, whose implementation can be found in the eventlet.greenio module, is designed to match the interface of the standard library socket.socket object. However, it is often useful to be able to use existing code which uses socket.socket directly without modifying it to use the eventlet apis. To do this, one must call wrap_socket_with_coroutine_socket. It is only necessary to do this once, at the beginning of the program, and it should be done before any socket objects which will be used are created. At some point we may decide to do this automatically upon import of eventlet; if you have an opinion about whether this is a good or a bad idea, please let us know.

Some code which is written in a multithreaded style may perform some tricks, such as calling select with only one file descriptor and a timeout to prevent the operation from being unbounded. For this specific situation there is wrap_select_with_coroutine_select; however it's always a good idea when trying any new library with eventlet to perform some tests to ensure eventlet is properly able to multiplex the operations. If you find a library which appears not to work, please mention it on the mailing list to find out whether someone has already experienced this and worked around it, or whether the library needs to be investigated and accommodated. One idea which could be implemented would add a file mapping between common module names and corresponding wrapper functions, so that eventlet could automatically execute monkey patch functions based on the modules that are imported.

TODO: We need to monkey patch os.pipe, stdin and stdout. Support for non-blocking pipes is done, but no monkey patching yet.

Communication Between Coroutines

channel
event
CoroutinePool
Actor
TODO

Using the Backdoor

TODO

Integrating Blocking Code with Threads

In the language of programs which use nonblocking I/O, code which takes longer than some very small interval to execute without yielding is said to "block."

tpool
TODO

Database Access

Most of the existing DB API implementations, especially MySQLdb, block in C. Therefore, eventlet's monkey patching of the socket module is not enough; since the database adapter does not use the python socket methods, calling them will block the entire process. Thus, any usage of them must call these blocking methods in the thread pool. To facilitate this, eventlet's DB pool module provides some convenient objects.

dbpool
TODO

Hub and Coroutine Library Negotiation

Eventlet performs multiplexing using the standard library select.select, select.poll, or the third-party libevent module, which in turn supports kqueue on FreeBSD and OS X and epoll on Linux. Eventlet will try to automatically determine what is installed and use what it thinks will provide best performance. Eventlet can also run inside of the nginx Web server using the mod_wsgi package for nginx. Using the nginx mod_wsgi package provides by far the best performance for serving HTTP; it almost seems impossible that Python should be able to achieve this level of performance, but it's true. See *Using Eventlet with Nginx*.

get_default_hub
use_hub
get_hub
TODO

Eventlet can also use several different coroutine implementations; the original is the greenlet package from py.lib. Eventlet can also use the cheese shop's packaging of greenlet; easy_install greenlet is generally the easiest way to get greenlet installed and running. Eventlet can also run inside of stackless-pypy without threads, and on Stackless Python 2.5.1, although it runs with an inferior emulation of greenlet implemented using tasklets, and is slower than Eventlet running on a plain python with the greenlet package installed. Another future candidate for experimentation is the libCoroutine package from the Io language, although it would need to be wrapped for Python first.

Advanced APIs

TODO

trampoline(fd, read=None, write=None, timeout=None)

Suspend the current coroutine until the given socket object or file descriptor is ready to read, ready to write, or the specified timeout elapses, depending on keyword arguments specified. To wait for fd to be ready to read, pass read=True;

ready to write, pass write=True. To specify a timeout, pass the timeout argument in seconds. If the specified timeout elapses before the socket is ready to read or write, TimeoutError will be raised instead of trampoline returning.

Using hub.switch() (you have to arrange to have your coroutine called back, otherwise it will never run again and just leak garbage)

Using tracked_greenlet and subclassing GreenletContext (need to do an example)

Using Eventlet with Nginx

TODO instructions for installing Nginx and mod_wsgi

In the nginx.conf file, set up a location with the wsgi_pass directive pointing to the nginx_mod_wsgi support module. TODO how to configure which wsgi application runs inside this? An env variable which specifies the name of the app?

```
location / {
    wsgi_pass /path/to/eventlet/support/nginx_mod_wsgi.py;
}
```