PyconSG 2013 Documentation

Release 0.1.0

Calvin Cheng

April 20, 2015

Contents

Contents:

CHAPTER 1

General concepts: concurrency, parallelism, threads and processes

In this section, we want to set the fundamentals knowledge required to understand how greenlets, pthreads (python threading for multithreading) and processes (python's multiprocessing) module work, so we can better understand the details involved in implementing python gevent.

1.1 What's the difference between concurrency and parallelism?

When we talk about implementing threads (whether greenlets or pthreads) and processes, what we are really trying to achieve is concurrency and/or parallelism.

So what's the difference, you ask?

Concurrency and parallelism are distinct concepts. Concurrency is concerned with managing access to shared state from different threads, whereas parallelism is concerned with utilizing multiple processors/cores to improve the performance of a computation.

1.2 What's a coroutine?

It's a computer program that generalize subroutines to allow multiple entry points for suspending and resuming execution at certain locations. Coroutines are suitable for implementing cooperative tasks, iterators, infinite lists and pipes.

1.3 What is a thread?

A thread is a basic unit of CPU utilization. It is also referred to as a "lightweight process".

A thread is a sequence of instructions within a process and it behaves like "a process within a process". It differs from a process because it does not have its own Process Control Block (collection of information about the processes). Usually, multiple threads are created within a process. Threads execute within a process and processes execute within the operating system kernel.

A thread comprises:

- thread ID
- program counter
- register set

- stack

A thread shares resources with its peer threads (all other threads in a particular task), such as:

- code section
- data section
- any operating resources which are available to the task

In a multi-threaded task, one server thread may be blocked and waiting for something and another thread in the same task may be running. If a process blocks, then the whole process stops. But a multithreaded process is useful in programs such as web browsers when we want to download a file, view an animation and print something all at the same time. When multiple threads cooperate in a single job, there is a higher throughput. If one thread must wait, the whole process does not stop because another thread may still run.

1.4 What is a process?

In computing, a process is an instance of a computer program that is being executed. It contains the program code and its current activity. Depending on the operating system, a process may be made up of multiple threads of execution that execute instructions concurrently.

Most modern operating systems prevent direct communication between independent processes, providing strictly mediated and controlled inter-process communication (IPC).

A process typically contains the following resources:

- an image of the executable machine code associated with the program
- memory, which includes:
  - executable code
  - process-specific data (input and output)
  - call stack that keeps track of active subroutines and/or other events
  - heap which holds intermediate computation data during run time
- operating system descriptors of resources that are allocated to the process such as file descriptors (unix/linux) and handles (windows), dat sources and sinks
- security attributes (process owner and set of permissions, e.g. allowable operations)
- processor state (context) such as registers and physical memory addressing

The operating system holds most of this information about active processes in data structures called process control blocks (PCB).

The operating system keeps its processes separated and allocates the resources they need, so that they are less likely to interfere with each other and cause system failures (e.g. deadlock or thrasing). The operating system may provide mechanisms for inter-process communication to enable processes in safe and predictable ways.

1.5 What's the difference between threads and processes?

A process is an executing instance of an application. What does that mean? When we launch a python shell or executing a python script, we start a process that runs our python shell or our python script. The operating system creates a process in response to us starting the python shell or python script and the *primary thread* of our process begins executing.

A thread is simply a path of execution within a process and in the case of python programs, our (python) process can contain multiple threads - implemented via the python threading module for example.

On a single processor, multithreading typically happens by time-division multiplexing (also referred to as multitasking), i.e. the single processor switches between different threads. This context switching happens fast enough so that we perceive the threads to be running at the same time.

On a multiprocessor and a multi-core system, threads can be truly concurrent, with every processor or CPU core executing a separate thread simultaneously (concurrently *and* in parallel).

1.6 What does that mean in the context of a python application?

python's (CPython) Global Interpretor Lock (GIL) prevents parallel threads of execution on multiple cores and as such, threading implementation on python is useful only for concurrent thread implementation for webservers.

This is what it means when people say "python manage.py runserver" development server for django is a *single-threaded server (process)*. There's only one thread running inside the "runserver" program. To solve the limitation of I/O and http network bottlenecks, third party source code (such as django-devserver by David Cramer) implements multiple threads to override the standard django runserver. Because our python server primarily deals with http traffic, our network I/O is the bottleneck and having multiple threads will improve its (data transfer) throughput.

However, if our python application is not a webserver and it bottlenecks due to CPU-intensive computation instead of network I/O, having multiple threads will not help at all (and in fact, such a CPU-bound python application will perform badly if we attempt to implement multiple threads). This is because of python's Global Interpreter Lock (GIL). There are some python interpreter implementation (such as Jython and IronPython) that do not have a GIL and so multithreaded execution for a CPU-bound python application will work well but the typical python interpreters that we use - CPython - is not appropriate for multithreaded CPU execution.

1.7 If CPython python has GIL, why do we still use it?

We know that the java implementation of Python (Jython) supports true threading (concurrent and parallel) by taking advantage of the underlying JVM. We also know that the IronPython port (running on Microsoft's CLR) do not have GIL. We could use them if we want to run code that has true threading capabilities.

The problem is that these platforms are always playing catch-up with new language features or library features, so unfortunately, it boils down to a trade-off between being able to use updated python features and python library features versus being able to run true threading code on Jython/IronPython.

1.8 So we cannot execute in parallel with python?

Actually, we can. But generally not by using threads but by using processes (with one exception which allows for parallel threads!).

Using the threading module on standard python (CPython interpreter), we **cannot** execute parallel CPU computation and we cannot execute parallel I/O operation because of GIL. The threading module is *still useful* for implementing I/O concurrency (e.g. webserver implementation) but causes more harm than good for CPU-intensive operations.

However, we **can** execute parallel CPU computation and parallel I/O operation in python with python's multiprocessing module, or subprocess module or a 3rd party library called parallel python - http://www.parallelpython.com/. Each approach has its own features and limitations but note that none of them use threads to achieve parallelism.

The exception - cython is able to support native thread parallelism through the **cython.parallel** module by releasing the GIL (http://docs.cython.org/src/userguide/parallelism.html?highlight=nogil). The backend for executing parallel

threads is OpenMP which is a feature available in the gcc compiler but not yet available in clang/llvm compiler. It is expected that the clang/llvm compiler will support OpenMP in the near future.

1.9 Advanced distributed, parallel computing with python

Beyond some of the solutions offered in the previous paragraph, large scale data processing tools include discoproject (python with erlang and includes map/reduce capabilities) and PySpark on top of the spark framework (scala based).

For data analysis which can become compute-intensive, augustus is an open source system for building and scoring scalable data mining and statistical algorithms.

For GPU computing, numbapro and pycuda are the emerging players.

1.9.1 Useful references

- http://doughellmann.com/2007/10/multiprocessing.html
- http://eli.thegreenplace.net/2011/12/27/python-threads-communication-and-stopping/
- http://eli.thegreenplace.net/2012/01/16/python-parallelizing-cpu-bound-tasks-with-multiprocessing/

CHAPTER 2

What is gevent?

Gevent is the use of simple, sequential programming in python to achieve scalability provided by asynchronous IO and lightweight multi-threading (as opposed to the callback-style of programming using Twisted's Deferred).

It is built on top of libevent/libev (for asynchronous I/O) and *greenlets* (lightweight cooperative multi-threading).

The job of libevent/libev is to handle event loops. As we will learn in the SocketIO sections later on, our SocketIOServer is an event loop which can *emit* specific results, *on* the occurrence of specific events. This is essentially how our SocketIOServer instance will know when to send a message to the client, hence real-time streaming of data from the server to the client, *on* the occurrence of specific events.

As we have understood from *general concepts* relating to *processes* and *threads (pthreads)* and concurrency and parallelism in the previous section, we want to be able to handle concurrency in python (for I/O benefits) and this is where *greenlets* fits into the picture.

Pre-1.0 version, gevent is based on libevent; and from 1.0 onwards, gevent is based on libev.

Once we understand what each of the building blocks of gevent do -

- *libevent*,
- *libev* and
- *greenlets*

we will have a clear idea of what it means to implement gevent in our python projects.

2.1 libevent

Written in C, this library provides asynchronous event notification. The libevent API provides a mechanism to execute a callback function when a specific event occurs on a file descriptor or after a timeout has been reached. It also supports callbacks triggered by signals and regular timeouts.

2.1.1 Implementation

- /dev/poll
- kqueue
- select
- poll
- epoll

- Solaris' event ports
- experimental support for real-time signals

2.1.2 Notable Applications that use libevent

- Chrome web browser (linux and mac version)
- Memcached
- tmux
- tor

etc etc

2.2 libev

A forked version of libevent with a view to improve on some (problematic) architectural decisions made in libevent, for instance:

- the global variable usage in libevent made it hard to use libevent safely in multithreaded environments.
- watcher structures are big because they combine I/O, time and signal handlers in one
- extra components such as the http and dns servers may not be implemented well, resulting in security issues

libev attempts to resolve some of these problems by not using global variables and use a loop context for all functions. The http and dns related components were completely removed and focuses on doing only one specific thing - POSIX event library

gevent 1.0 onwards has been refactored to use libev instead of libevent. Details of the rationale for this decision is explained by gevent author Denis Bilenko.

The c-ares library is used to replace libevent-dns since libev does not handle http and dns functionality as explained above.

2.2.1 Implementation

Exactly the same as libevent's

- /dev/poll
- kqueue
- select
- poll
- epoll
- Solaris' event ports
- experimental support for real-time signals

libevent has better windows-support implementation since libevent accepts windows handles while libev needs to convert windows handles into C runtime handles.

2.3 greenlets

Greenlets are a lightweight cooperative threads - which is different from our conventional understanding of POSIX threads (pthreads).

It is a spin-off of Stackless, a version of CPython which supports microthreads called "tasklets". Tasklets (Stackless) run pseudo-concurrently (in a single or a few OS-level threads) and are synchronized with data exchanged in "channels". Greenlet is more primitive compared to these "tasklet" microthreads and is more accurately described as a "coroutines" - cooperative routines. Meaning that greenlets has no implicit scheduling like "tasklets" and we can control exactly when our code runs.

The greenlet source code can be found here and is provided as a C extension module for python.

We dive into further details about greenlets *here*.

2.4 gevent API design

gevent's interface follows the conventions set by python standard modules

- gevent.event.Event has the same interface and the same semantics as python's built-in modules threading.Event and multiprocessing.Event.
- wait() does not raise an exception
- get() can raise an exception or return a value
- join() is like wait() but for units of execution

Having consistent code interfaces like these helps programmers read and reason with the code in a much more efficient manner.

2.5 gevent with other python extensions

If some kind of transaction involves I/O, the greenlet might get switched away waiting for a write-acknowledgement (or other kinds of I/O block), we have to explicitly lock the transaction. If our code ever gets back to the old blocking I/O style, our entire application will fail. To prevent this from happening, only use extensions that make use of the built-in python socket module.

2.6 gevent's monkey patch

A monkey patch is a way to extend or modify the run-time code of dynamic languages without altering the original source code. Monkey patching as a programming technique is very powerful but can result in hard-to-debug code in the wrong hands. Jeff Atwood wrote a good post about these issues here - http://www.codinghorror.com/blog/2008/07/monkeypatching-for-humans.html.

> Monkey patching is the new black [in the Ruby community]. It's what all the hip kids are doing. To the point that smart, experienced hackers reach for a monkey patch as their tool of first resort, even when a simpler, more traditional solution is possible.
>
> I don't believe this situation to be sustainable. Where I work, we are already seeing subtle, difficult-to-debug problems crop up as the result of monkey patching in plugins. Patches interact in unpredictable, combinatoric ways. And by their nature, bugs caused by monkey patches are more difficult to track down than those introduced by more traditional classes and methods. As just one example: on one project, it

> was a known caveat that we could not rely on class inheritable attributes as provided by ActiveSupport. No one knew why. Every Model we wrote had to use awkward workarounds. Eventually we tracked it down in a plugin that generated admin consoles. It was overwriting Class.inherited(). It took us months to find this out.
>
> This is just going to get worse if we don't do something about it. And the "something" is going to have to be a cultural shift, not a technical fix. I believe it is time for experienced Ruby programmers to wean ourselves off of monkey patching, and start demonstrating more robust techniques.

Whenever we decide to use a library which uses a monkey patch approach, it is important that we read the source code and documentation fully and understand how that library's monkey patch affects our standard source code, modules and libraries.

One of gevent's most important features is monkey patching, so we will need to understand what monkey patching actually does - http://www.gevent.org/gevent.monkey.html

> The functions in this module patch parts of the standard library with compatible cooperative counterparts from gevent package.
>
> To patch an individual module call the corresponding patch_* function. For example, to patch socket module only, call patch_socket(). To patch all default modules, call gevent.monkey.patch_all().
>
> Monkey can also patch thread and threading to become greenlet-based. So thread.start_new_thread() starts a new greenlet instead and threading.local becomes a greenlet-local storage.

2.6.1 Examples

This works:-

```
import gevent.monkey; gevent.monkey.patch_thread()
import threading
```

This explodes (try it):-

```
import threading
import gevent.monkey; gevent.monkey.patch_thread()
```

When the threading module is imported, it uses the main thread ID as a key in a module-level thread dictionary. When the program exits, the threading module tries to obtain the thread instance from the dictionary (using the current thread ID) to perform clean up.

However, because of *gevent.monkey.patch_thread()*, the ID of the main thread is no longer the same! Stackoverflow question and answer here with all the gory details.

Long story short, the *order in which we monkey patch gevent is important*. Always execute the monkey patch first before running your python code, particularly if your code uses threading at some point. Note that the *logging* module also uses *threading* so when *logging* your application, monkey patch first!

2.7 gevent with webservers

Most web application accept requests via http. Since gevent allows us to work seamlessly with python's socket APIs, there will be no blocking call. However, as mentioned above in *gevent with other python extensions*, be careful when adding dependencies with C-Extensions that might circumvent python sockets.

2.8 gevent with databases

Our python application typically sits between a webserver (as mentioned above) and a database. Now that we are sure that our gevent-powered python app is not affected by code or dependencies with C-Extensions that circumvent python sockets, we want to be sure that we are using the appropriate database drivers.

Database drivers that work with python gevent apps are:

- mysql-connector
- pymongo
- redis-py
- psycopg

We cannot use the standard MySQLdb driver because it is C-based.

How we design our database-connection depends on how our http-interface works. If we use *greenlet-pool* for example, it spawns a new greenlet per request. On the database side, for *redis-py*, every *redis.Connection* instance has one socket attached to it. The *redis-client* uses a pool of these connections. Every command gets a connection from the pool and releases it afterwards. This is a good design pattern for use with gevent because we cannot afford to create one connection per greenlet - since databases often handle every established connection with a thread, this can cause our machine to run out of resources on the database side very quickly!

Using a single connection on the other hand, will create a huge bottleneck. Connection pools witha limited number of connections can hinder performance so on a production application, we will need to carefully decide on the connection limit as our app usage pattern evolves.

pymongo can ensure that it uses one connection for one greenlet through its whole lifetime so we have read-write consistency.

2.9 gevent with I/O operations

Because of GIL, python threads are **not parallel** (at least in the CPython implementation). gevent's greenlet does not give us magical powers to suddenly achieve parallelism. There will only be one greenlet running in a particular process at any time. Because of this, CPU-bound apps do not gain any performance gain from using gevent (or python's standard threading).

gevent is only useful for solving I/O bottlenecks. Because our gevent python application is trapped between a http connection, a database and perhaps a cache and/or messaging server, gevent is useful for us.

2.9.1 Exceptions to I/O operations advantage

However (well, you know that was coming right? :-)), gevent does not handle regular file read-write (I/O) well.

POSIX says:

> File descriptors associated with regular files shall always select true for ready to read, ready to write, and error conditions. the linux read man-page says:
>
> Many file systems and disks were considered to be fast enough that the implementation of O_NONBLOCK was deemed unnecessary. So, O_NONBLOCK may not be available on files and/or disks.

The *libev-documentation* says:

> [...] you get a readiness notification as soon as the kernel knows whether and how much data is there, and in the case of open files, that's always the case, so you always get a readiness notification instantly, and your read (or possibly write) will still block on the disk I/O.

File I/O does not really work the asynchronous way. It blocks! Expect your application to block on file I/O, so load every file you need up front before handling requests or do file I/O in a separate process (Pipes support non-blocking I/O).

2.10 gevent code example

Here's a simple example of how we can make use of gevent's I/O performance advantage in our code. In a typical web request-respond cycle, we may want to run concurrent jobs that

- retrieve data source from a particular database,
- make a get request to a 3rd party (or even in-house) API on a different application that returns us json,
- instantiates an SMTP connection to send out an email,
- or more

We can of course execute these tasks one-by-one, in a sequential manner. But being the experts that we are, we would like to execute them in a concurrent way (where the tasks will switch away if it encounters an I/O bottleneck in one of the above I/O jobs).

So we can write:-

```
def handle_view(request):
    jobs = []
    jobs.append(gevent.spawn(orm_call, 'Andy'))
    jobs.append(gevent.spawn(call_facebook_graph_api, 14213))
    jobs.append(gevent.spawn(email, 'me@mysite.com'))
    gevent.joinall()
```

This allows us to handle all 3 tasks concurrently.

2.11 Summary

- gevent helps us to reduce the overheads associated with threading to a minium. (greenlets)
- gevent helps us avoid resource wastage during I/O by using asynchronous, event-based I/O. (libevent/libev depending on which version of gevent we use)
- gevent is exceptionally suited for concurrency implementation with webservers, databases, caches and messaging frameworks because these are I/O-bound operations
- The exception to I/O performance gain is file I/O. To deal with that, load file upfront or execute file I/O in a separate process
- gevent is not a solution for multicore CPU-bound programs. To deal with that, delegate your CPU-intensive code to a queue or to another program and return the results from a message queue.

CHAPTER 3

What are greenlets?

Greenlets are lightweight thread-like structures that are scheduled and managed inside the process. They are references to the part of the stack that is used by the thread. Compared to POSIX threads (pthreads), there is no stack allocated up front and there is only as much stack as is actually used by the greenlet

In python, we implement greenlets via the gevent package and we implement pthreads via python's built-in threading module.

Both green threads (greenlets) and POSIX threads (pthreads) are mechanisms to support multithreaded execution of programs.

POSIX threads use the operating system's native ability to manage multithreaded processes. When we run pthreads, the kernel schedules and manages the various threads that make up the process.

Green threads emulate multithreaded environments without relying on any native operating system capabilities. Green threads run code in user space that manages and schedules threads.

The key differences between greenlets and pthreads can be summarized as such:

pthreads	greenlets
pthreads can switch between threads pre-emptively, switching control from a running thread to a non-running thread at any time	greenlets only switch when control is explicitly given up by a thread - when using yield() or wait() - or when a thread performs a I/O blocking operation such as read or write
On multicore machines, pthreads can run more than one thread. However python's Global Interpreter Lock (CPython Intepreter) prevents parallelism and concurrency is only effective for I/O-bound programs	greenlets can only run on one single CPU and is useful for I/O-bound programs
Race conditions can occur when implementing multi-threading code. Use locks to manage mutex to avoid race conditions.	There's no possibility of two threads of control accessing the same shared memory at the same time for greenlets so there will not be any race conditions.

3.1 Installation

When we install gevent, greenlet as a dependency will be downloaded, compiled (since it is a c extension) and installed.

We can also install greenlet directly:

```
pip install greenlet

Downloading/unpacking greenlet
  Downloading greenlet-0.4.1.zip (75kB): 75kB downloaded
  Running setup.py egg_info for package greenlet
```

```
Installing collected packages: greenlet
  Running setup.py install for greenlet
    /usr/bin/clang -fno-strict-aliasing -fno-common -dynamic -pipe -O2 -fwrapv -arch x86_64 -DND
    clang: warning: argument unused during compilation: '-fno-tree-dominator-opts'
    building 'greenlet' extension
    /usr/bin/clang -fno-strict-aliasing -fno-common -dynamic -pipe -O2 -fwrapv -arch x86_64 -DND
    clang: warning: argument unused during compilation: '-fno-tree-dominator-opts'
    In file included from greenlet.c:416:
    In file included from ./slp_platformselect.h:12:
    ./platform/switch_amd64_unix.h:40:26: warning: unknown attribute 'noclone' ignored [-Wattrib
    __attribute__((noinline, noclone)) int fancy_return_zero(void);
                             ^
    ./platform/switch_amd64_unix.h:41:26: warning: unknown attribute 'noclone' ignored [-Wattrib
    __attribute__((noinline, noclone)) int
                             ^
    2 warnings generated.
    /usr/bin/clang -bundle -undefined dynamic_lookup -L/opt/local/lib -L/opt/local/lib/db46 -arc
    Linking /Users/calvin/.virtualenvs/pyconsg2013/build/greenlet/build/lib.macosx-10.7-x86_64-2

Successfully installed greenlet
Cleaning up...
```

Notice that the c code gets built and the shared object file *greenlet.so* is now available for us to import inside our python code.

3.2 Example

Here's a simple example extracted from greenlet docs that explains the nature of greenlet execution:

```
from greenlet import greenlet


def test1():
    print 12
    gr2.switch()
    print 34


def test2():
    print 56
    gr1.switch()
    print 78

gr1 = greenlet(test1)
gr2 = greenlet(test2)
gr1.switch()
```

CHAPTER 4

Spawning greenlets via gevent

Gevent provides a wrapper *Greenlet* class around base *greenlet* library. Reference https://github.com/surfly/gevent/blob/master/gevent/greenlet.py

In our own source code, we will therefore depend on

```
import gevent
from gevent import Greenlet
```

to implement our custom logic.

1. Via the base *Greenlet* class:

   ```
   import gevent
   from gevent import Greenlet

   def foo(message, n):
       """
       Each thread will be passed the message, and n arguments
       in its initialization.
       """
       gevent.sleep(n)
       print(message)

   # Initialize a new Greenlet instance running the named function
   # foo
   thread1 = Greenlet.spawn(foo, "Hello", 1)

   # Wrapper for creating and runing a new Greenlet from the named
   # function foo, with the passed arguments
   thread2 = gevent.spawn(foo, "I live!", 2)

   # Lambda expressions
   thread3 = gevent.spawn(lambda x: (x+1), 2)

   threads = [thread1, thread2, thread3]

   # Block until all threads complete.
   gevent.joinall(threads)
   ```

2. Subclassing the base *Greenlet* class and using internal method *_run*

   ```
   import gevent
   from gevent import Greenlet
   ```

```
class MyGreenlet(Greenlet):

    def __init__(self, message, n):
        Greenlet.__init__(self)
        self.message = message
        self.n = n

    def _run(self):
        print(self.message)
        gevent.sleep(self.n)

g = MyGreenlet("Hi there!", 3)
g.start()
g.join()
```

4.1 Full Tutorial on gevent

See http://sdiehl.github.io/gevent-tutorial/ for all the detailed explanations of gevent functionalities.

CHAPTER 5

General concepts: what are sockets?

Written in C, Berkeley sockets (BSD sockets) is a computing library with an API for internet sockets and other unix domain sockets used for inter-process communication. The API has not changed much in its POSIX equivalent, so POSIX sockets are basically Berkeley sockets.

All modern operating systems come with some implementation of the Berkeley socket interface because it has become the standard interface for connecting to the internet.

Various programming languages provide similar interfaces and are essentially wrappers around the BSD socket C API.

5.1 Python sockets

Gordon McMillan wrote a great overview of sockets and how python's standard socket module can easily be used to create a socket for IPC (Inter-Process Communication).

- python2: http://docs.python.org/2/howto/sockets.html
- python3: http://docs.python.org/3/howto/sockets.html

When a socket is created, an endpoint for communication becomes available and a corresponding file descriptor is returned. A file descriptor is simply an abstract indicator for accessing a file and has integer values of 0, 1, 2 corresponding to standard input (stdin), standard output (stdout) and standard error (stderr).

A simple example illustrates how a server socket and a client socket can be created to send data to each other.

```
# server.py
import socket

serversocket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
serversocket.bind(('localhost', 8089))
serversocket.listen(5)

while True:
    connection, address = serversocket.accept()
    buf = connection.recv(64)
    if len(buf)>0:
        print buf

# client.py
import socket

clientsocket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
```

```
clientsocket.connect(('localhost', 8089))
clientsocket.send('hello')
```

After we run server.py

```
python server.py
```

We run client.py

```
python client.py
```

And "hello" gets printed out on the *python server.py* process stdout (standard output).

Our webservers can be implemented in python or C or golang or any other languages but the basis from which data is passed between each other via the HTTP (TCP) protocol rest upon sockets. Sockets are the fundamental building block from which HTTP, HTTPS, FTP, SMTP protocols (all of these are TCP-type protocols) are defined.

DNS, DHCP, TFTP, SNMP, RIP, VOIP protocols are UDP protocols but they too are built on top of sockets.

5.2 So what's a WebSocket?

WebSocket is a full-duplex communication channel over one single TCP-type connection. It is an independent TCP-type protocol and its only association to HTTP is that its handshake is intepreted by HTTP servers as an Upgrade request. HTTP 1.1 introduced an "Upgrade" header field and this connection "Upgrade" must be sent by the client (in other words, this "Upgrade" header is sent by SocketIO javascript client to tell the server that this is a WebSocket connection). The server can also enforce an upgrade by sending a "426 upgrade required" response back to the client and the client will have to handle this response accordingly - either upgrade or fail the websocket connection attempt.

This is how our WebSocket can work seamlessly with a HTTP server.

WebSocket is a browser/client feature and only works on browsers (or custom clients, if you are writing your custom native app) that support it. Socket.IO client library intelligently determines if the browser it is loaded up on supports WebSocket or not. If it does, it will use WebSocket to communicate with the server-side SocketIO server. If it does not, it will attempt to use one of the fallback transport mechanisms.

WebSocket differs from TCP protocols like HTTP in that it enables a stream of messages instead of a stream of bytes. Before WebSocket, port 80 full-duplex communication was attainable using Comet. However, compared to WebSocket, comet implementation is non-trivial and is inefficient for small messages because of TCP handshake and HTTP header overheads.

A WebSocket protocol handshake looks like this:

Client sends a WebSocket handshake request.

```
GET /mychat HTTP/1.1
Host: server.example.com
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Key: x3JJHMbDL1EzLkh9GBhXDw==
Sec-WebSocket-Protocol: chat
Sec-WebSocket-Version: 13
Origin: http://example.com
```

Server returns a WebSocket handshake response.

```
HTTP/1.1 101 Switching Protocols
Upgrade: websocket
Connection: Upgrade
```

```
Sec-WebSocket-Accept: HSmrc0sMlYUkAGmm5OPpG2HaGWk=
Sec-WebSocket-Protocol: chat
```

A protocol like HTTP uses a (BSD socket) socket for only one transfer. The client sends the request, then reads the reply and the socket is discarded. This means that a HTTP client can detect the end of the reply by receiving 0 bytes.

For WebSocket, once a connection is established, the client and server can send WebSocket data or text frames back and forth in full-duplex mode. The data itself is minimally framed, containing a small header and the payload. WebSocket transmissions are described as "messages" where a single message can optionally be splitted across several data frames. This can allow for sending of messages where initial data is available but the complete length of the message is unknown (it sends one data frame after another until the end is reached and marked with the FIN bit). With extensions to the protocol, this can also be used for multiplexing several streams simultaneously (for instance to avoid monopolizing use of a socket for a single large payload).

CHAPTER 6

What is Socket.IO?

Socket.IO is a javascript library for real-time web applications. It has two parts

- a client side library that runs in the browser; and
- a server-side library for node.js.

Both components have identitical API and are event-driven.

There are implementations for the **server-side library in other languages**.

In python land, we use the gevent-socketio library.

```
pip install -e "git://github.com/abourget/gevent-socketio.git#egg=gevent-socketio"
```

Whichever server-side language implementation we so choose, the following 6 transports are supported:

- WebSocket
- Adobe® Flash® Socket
- AJAX long polling
- AJAX multipart streaming
- Forever Iframe
- JSONP Polling

Socket.IO selects the most capable transport at runtime without affecting its APIs so that we can have realtime connectivity on every browser.

6.1 Various Language Implementations

- Python
  - gevent-socketio - https://github.com/abourget/gevent-socketio
- Perl
  - PocketIO - https://github.com/vti/pocketio
- Java
  - Atmosphere - https://github.com/Atmosphere/atmosphere
  - Netty-socketio - https://github.com/mrniko/netty-socketio
  - Gisio for Google's GWT Framework - https://bitbucket.org/c58/gnisio

  - Socket.IO for Vert.x - https://github.com/keesun/mod-socket-io
- Golang
  - go-socket.io (supports up to socket.io client 0.6) - https://github.com/davies/go-socket.io
  - go-socket.io (supports up to socket.io client 0.8) - https://github.com/murz/go-socket.io
- Erlang
  - Socket.IO-Erlang (supports up to socket.io client 0.6) - https://github.com/yrashk/socket.io-erlang
  - erlang Socket.IO (supports up to socket.io client 1.0) - https://code.google.com/p/erlang-scoketio

6.2 ServerIOServer example

The simplest way to launch a *SocketIOServer*:-

```
from gevent import monkey
from socketio.server import SocketIOServer


monkey.patch_all()

PORT = 5000

if __name__ == '__main__':
    print 'Listening on http://127.0.0.1:%s and on port 10843 (flash policy server)' % PORT
    SocketIOServer(('', PORT), app, resource="socket.io").serve_forever()
```

SocketIOServer is our python implementation of the original Socket.IO server-side nodejs library.

The full source code can be referred to here - https://github.com/abourget/gevent-socketio/blob/master/socketio/server.py

And we can see that it accepts a *(host, port)* argument, the *app* instance argument, a *resource* argument and optionally a list of *transports*, the flash *policy_server* as a boolean and an arbitrary list of keyword arguments.

6.2.1 SocketIOServer host and port

The *host and port tuple* argument is straightforward - provide the IP address and the port that we would like to run our *SocketIOServer* on.

6.2.2 SocketIOServer app

The *app* argument is simply an instance of the python application that we will run.

Here's a *django* example:

```
# runserver_socketio.py

#!/usr/bin/env python
from gevent import monkey
from socketio.server import SocketIOServer
import django.core.handlers.wsgi
import os
import sys
```

```
monkey.patch_all()

try:
    import settings
except ImportError:
    sys.stderr.write("Error: Can't find the file 'settings.py' in the directory containing %r. It ap
    sys.exit(1)

PORT = 9000

os.environ['DJANGO_SETTINGS_MODULE'] = 'settings'

app = django.core.handlers.wsgi.WSGIHandler()

sys.path.insert(0, os.path.join(settings.PROJECT_ROOT, "apps"))

if __name__ == '__main__':
    print 'Listening on http://127.0.0.1:%s and on port 10843 (flash policy server)' % PORT
    SocketIOServer(('', PORT), app, resource="socket.io").serve_forever()
```

6.2.3 SocketIOServer resource

The *resource* argument is where we will have to define our actual python application's Socket.IO *url*.

For a django application, we define in our *urls.py*, like this

```
# urls.py

from django.conf.urls.defaults import patterns, include, url
import socketio.sdjango

socketio.sdjango.autodiscover()

urlpatterns = patterns("chat.views",
    url("^socket\.io", include(socketio.sdjango.urls)),
```

sdjango is a pre-written integration module available in *gevent-socketio* library and it contains the following definition for *sdjango.urls*:

```
SOCKETIO_NS = {}

class namespace(object):
    def __init__(self, name=''):
        self.name = name

    def __call__(self, handler):
        SOCKETIO_NS[self.name] = handler
        return handler

@csrf_exempt
def socketio(request):
    try:
        socketio_manage(request.environ, SOCKETIO_NS, request)
    except:
        logging.getLogger("socketio").error("Exception while handling socketio connection", exc_info=
    return HttpResponse("")
```

```
urls = patterns("", (r'', socketio))
```

6.3 SocketIO Namespace example

A simple example of implementing a namespace on the client (javascript) side is:

```
var socket = io.connect("/chat");
```

A namespace can be confused as a "url" for people new to SocketIO. It is actually not a *url* (*router* in MVC design pattern speak) but in fact a *controller*.

On the server side, our namespaces are implemented via the *BaseNamespace* class:

```
from socketio.namespace import BaseNamespace
from socketio import socketio_manage

class ChatNamespace(BaseNamespace):

    def on_user_msg(self, msg):
        self.emit('user_msg', msg)

def socketio_service(request):
    socketio_manage(request.environ, {'/chat': ChatNamespace}, request)
    return 'out'
```

In this example, the *user_msg* event will be in the */chat* namespace. So we can say that the */chat* namespace contains the *on_user_msg* method.

socketio_manage() is the method that runs when the *SocketIOServer* gets started and the real-time communication between the client and the server happens through that method.

The *socketio_manage()* function is going to be called only once per socket opening, even though we are using a long polling mechanism. The subsequent calls (for long polling) will be hooked directly at the server-level, to interact with the active Socket instance. This means we will not get access to the future request or environ objects. This is of particular importance regarding sessions. The session will be opened once at the opening of the Socket, and not closed until the socket is closed. We are responsible for opening and closing the cookie-based session ourselves if we want to keep its data in sync with the rest of our GET/POST calls.

A slightly more complex *django* example here':

```
# sockets.py

import logging

from socketio.namespace import BaseNamespace
from socketio.mixins import RoomsMixin, BroadcastMixin
from socketio.sdjango import namespace

@namespace('/chat')
class ChatNamespace(BaseNamespace, RoomsMixin, BroadcastMixin):
    nicknames = []

    def initialize(self):
        self.logger = logging.getLogger("socketio.chat")
        self.log("Socketio session started")

    def log(self, message):
```

```
        self.logger.info("[{0}] {1}".format(self.socket.sessid, message))

    def on_join(self, room):
        self.room = room
        self.join(room)
        return True

    def on_nickname(self, nickname):
        self.log('Nickname: {0}'.format(nickname))
        self.nicknames.append(nickname)
        self.socket.session['nickname'] = nickname
        self.broadcast_event('announcement', '%s has connected' % nickname)
        self.broadcast_event('nicknames', self.nicknames)
        return True, nickname

    def recv_disconnect(self):
        # Remove nickname from the list.
        self.log('Disconnected')
        nickname = self.socket.session['nickname']
        self.nicknames.remove(nickname)
        self.broadcast_event('announcement', '%s has disconnected' % nickname)
        self.broadcast_event('nicknames', self.nicknames)
        self.disconnect(silent=True)
        return True

    def on_user_message(self, msg):
        self.log('User message: {0}'.format(msg))
        self.emit_to_room(self.room, 'msg_to_room',
            self.socket.session['nickname'], msg)
        return True
```

The *sdjango* module has defined a nice namespace class which accepts the name of our namespace (*/chat* in this case) which we can use as a decorator corresponding to our fully defined *ChatNamespace* subclass (subclass of *BaseNamespace*). All our event handling methods are implemented in this class and will work with a javascript client that connects via

```
var socket = io.connect("/chat");`
```

, the *io.connect("/chat")* call.

6.4 Summary of gevent-socketio API

The key concepts and usage that we have covered are:

- **socketio.socketio_manage** (usage seen in the *sdjango.py* module)
- **socketio.namespace** (usage seen in by the implementation of the *BaseNamespace* parent class and the *@namespace* decorator in django)
- **socketio.server** (usage seen in the instantiation of a *SocketIOServer* instance)

In the django example above, we also notice the use of **socketio.mixins** to pass in (specifically *RoomsMixin* and *BroadcastMixin*) pre-written classes that contain methods useful for a typical chat project.

Other APIs include:

- **socketio.virtsocket**
- **socketio.packet**

- **socketio.handler**
- **socketio.transports**

Reference document - https://gevent-socketio.readthedocs.org/en/latest/#api-docs

6.5 From SocketIO client to SocketIOServer logic and back

Here's an example django *chat* app layout (inside a django project):

```
chat
-- __init__.py
-- admin.py
-- management
|   -- __init__.py
|   -- commands
|       -- __init__.py
|       -- runserver_socketio.py
-- models.py
-- sockets.py
-- static
|   -- css
|   |   -- chat.css
|   -- flashsocket
|   |   -- WebSocketMain.swf
|   -- js
|       -- chat.js
|       -- socket.io.js
-- templates
|   -- base.html
|   -- room.html
|   -- rooms.html
-- tests.py
-- urls.py
-- views.py
```

Our client-side logic resides in *chat.js*. This can be thought of as our client-side "controller logic on controller named /chat". If we so desire, we can always instantiate the second or more socket(s) with different controller names.

```
// chat.js

var socket = io.connect("/chat");

socket.on('connect', function () {
    $('#chat').addClass('connected');
    socket.emit('join', window.room);
});

socket.on('announcement', function (msg) {
    $('#lines').append($('<p>').append($('<em>').text(msg)));
});

socket.on('nicknames', function (nicknames) {
    console.log("nicknames: " + nicknames);
    $('#nicknames').empty().append($('<span>Online: </span>'));
    for (var i in nicknames) {
        $('#nicknames').append($('<b>').text(nicknames[i]));
    }
```

```
});

socket.on('msg_to_room', message);

socket.on('reconnect', function () {
    $('#lines').remove();
    message('System', 'Reconnected to the server');
});

socket.on('reconnecting', function () {
    message('System', 'Attempting to re-connect to the server');
});

socket.on('error', function (e) {
    message('System', e ? e : 'A unknown error occurred');
});

function message (from, msg) {
    $('#lines').append($('<p>').append($('<b>').text(from), msg));
}

// DOM manipulation
$(function () {
    $('#set-nickname').submit(function (ev) {
        socket.emit('nickname', $('#nick').val(), function (set) {
            if (set) {
                clear();
                return $('#chat').addClass('nickname-set');
            }
            $('#nickname-err').css('visibility', 'visible');
        });
        return false;
    });

    $('#send-message').submit(function () {
        //message('me', "Fake it first: " + $('#message').val());
        socket.emit('user message', $('#message').val());
        clear();
        $('#lines').get(0).scrollTop = 10000000;
        return false;
    });

    function clear () {
        $('#message').val('').focus();
    }
});
```

The client side SocketIO library is straightforward to use:

- **socket.on** receives 2 arguments, the *event_name* (which the server side code will emit to) as the first argument and the *event callback function* as the second argument. When an event happens, the (callback) function gets triggered.
- **socket.emit** also receives 2 arguments, the first being the *event_name* and the 2nd being the *message*. **emit('<event_name>', <message>)** sends a message to the server - the python method **on_<event_name>(<message>)** is waiting for the client side **emit()** call.

Here's the corresponding server-side code in our **ChatNamespace** class. This server-side controller named */chat* corresponds to our client-side controller also named */chat*, which means that any **on** or **emit** refers to its counterpart.

```
# sockets.py

@namespace('/chat')
class ChatNamespace(BaseNamespace, LonelyRoomMixin, BroadcastMixin):
    nicknames = []

    def initialize(self):
        self.logger = logging.getLogger("socketio.chat")
        self.log("Socketio session started")

    def log(self, message):
        self.logger.info("[{0}] {1}".format(self.socket.sessid, message))

    def on_join(self, room):
        self.room = room
        self.join(room)
        return True

    def on_nickname(self, nickname):
        print("Creating the nickname: " + nickname)
        self.log('Nickname: {0}'.format(nickname))
        self.socket.session['nickname'] = nickname
        self.nicknames.append(nickname)
        self.broadcast_event('announcement', '%s has connected' % nickname)
        self.broadcast_event('nicknames', self.nicknames)
        return True, nickname

    def recv_disconnect(self):
        self.log('Disconnected')
        nickname = self.socket.session['nickname']
        self.nicknames.remove(nickname)
        self.broadcast_event('announcement', '%s has disconnected' % nickname)
        self.broadcast_event('nicknames', self.nicknames)
        self.disconnect(silent=True)
        return True

    def on_user_message(self, msg):
        self.log('User message: {0}'.format(msg))
        # TODO: dig into the logic of emit_to_room
        self.emit_to_room(self.room, 'msg_to_room',
                          self.socket.session['nickname'], msg)
        return True
```

CHAPTER 7

A gevent-socketio example

Full example here - https://github.com/calvinchengx/learnsocketio

CHAPTER 8

Indices and tables

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