Viability of Twisted and Python for Implementing an Application Server Herd

Viet T. Nguyen
University of California, Los Angeles

Abstract

This paper provides a cursory look into the use of the Twisted framework, built on top of the Python language, to build application server herds with particular interest to language considerations such as type checking, memory management, and multithreading.

1 Introduction

The motivation for this research comes from a desire at our company to build a "new Wikimedia-style service designed for news." Due to differing requirements we sought an alternative to Wikimedia's LAMP platform (GNU/Linux, Apache, MySQL, and PHP). Our service intends to provide much more frequent updates, access via a variety of protocols, and a robustness towards increasing mobile traffic share.

The requirements outlined for the service result in a number of high level technical desires. To avoid application server bottlenecks we intend to distribute the application servers. With distributed servers we require rapid diffusion of unstructured data via a flooding algorithm. Beyond data diffusion there are a number of "-ility's" we hope to achieve: maintainability, reliability, scalability, parallelizability. Of course from a cost perspective we would also like ease of implementation.

2 Twisted

One of the many options available to implement such a framework is Twisted. Twisted is an event-driven networking engine written in Python [10]. Twisted shows promise because of its event driven. It also contains means to spawn processes beyond using multiple threads, allowing us to overcome the Global Interpreter Lock which will be discussed later in Section 5.

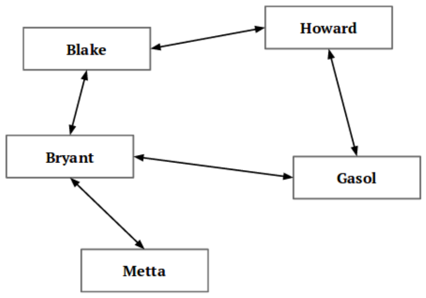

Figure 1: Topology of the network used to test the prototype.

3 Prototype

To assist in evaluating Twisted and Python a simple prototype was rapidly developed. The prototype implements a single application run in multiple instances to mimic multiple servers each running the application. They communicate with each other via TCP. The servers as a whole comprise a service. A client expects to be able to send data to the service and then request something based on the data it provided the service. Since a client does not necssarily interact with the same server the servers must propagate the client's data to the other servers in the service. The topology of prototype servers is given by Figure 1. More information about the design of the prototype are given by [5].

Handling client input was straightforward in thanks to the event driven structure of Twisted. Polling or thread handling is unnecessary as the `twisted.internet.reactor` handles much of it. Implementation simply requires the inheritance of prototypical classes representing protocols (e.g. connections), a means of spawning protocols (factories), and then the definition of desired behavior encapsulated in a method whose arguments are the incoming data from the client.

From there Python provides numerous means for parsing data, including built-in string and list manipulation. Here the use of Twisted shines in quickly getting development to the actual desired functionality with little boiler plate code. Python's maturity and flexibility also prove advantageous here with its numerous capabilities via built-in structures and standard packages. Also, Python's object model, where everything is an object and type handling is done with duck typing, makes it easy to both prototype code and develop abstract code that can be reused.

3.1 Simple Flooding Algorithm

Part of the prototype functionality requires that servers communicate in an intra-service manner to keep client data consistent across servers. To achieve this a simple flooding algorithm was implemented. Upon receiving of new client data the receiving server updates its internal store of client data (in memory) before forwarding the data to all of its known peers. When a peer receives this data it then updates its internal store of client data before proceding to forward the data to all of its known peers sans the peer it received the data from. To prevent infinite recursive forwarding a peer checks the time stamp of the data and proceeds only if the time stamp is newer than the data in its local store.

Implementation is relatively straightforward except for having to understand the multi-layered callbacks required to establish a temporary connection to send data from server to server. Receiving and parsing of data from another server follows the same steps as receiving and parsing client data. In communicating with another server, however, and end point needs to be instantiated (e.g. `TCP4ClientEndpoint`), a connection made via the end point, and a call back provided to the connection. The connection then executes the callback providing the callback with the protocol that can then be used to actually send the data. As a result the method that performs data communication is separated from the location of desired communication by two object instantiations and a callback that must be executed by the reactor. When implemented the process works fine, but maintainability and readability suffers. However, in an event driven framework this problem is common.

3.2 Post-Mortem

The Twisted framework and Python very much succeeded in allowing us to quickly develop the prototype. It is our feeling that this setup can be used to develop our product in a scalable fashion.

Access via a variety of protocols is readily available due to the object-oriented design of the framework and its built-in implementation of numerous other protocols that can, more or less, be swapped with the instantiations used in the prototype. It appears to require little additional work to have servers communicate over `ssh` or UDP. The same applies for client communication. The design of the server herd addresses the concern for handling mobile traffic.

However, there are caveats to using Twisted and Python. An understanding of Python's multithreading implementation and memory management are important to avoiding development of software that scales and performs poorly. Some of these concerns are discussed in following sections.

3.3 Source Code

Source code for the prototype can be found at the following URL:

```
https://github.com/vietjtnguyen/
     ucla-fall12-cs131-pr
```

Unfortunately the prototype lives up to its name. The prototype has not been designed for robust execution or even functionality. Instead it simply proves the ability to run multiple servers using Python and Twisted that can communicate with each other. It does not strictly adhere to good object-oriented design, don't-repeat-yourself principles (DRY), or other best practices.

4 Python Memory Management

Python memory allocation and deallocation is automatic using two strategies: reference counting and garbage collection. The important aspect of Python memory management in our application is that, despite the automatic methods, the onus is on the developer to ensure memory is handled properly. Python does not automatically garbage collect when out of memory so such errors must be handled. The developer should avoid cyclic references which are not handled by the reference counting. For long running programs, garbage collection can be manually invoked and the garbage collector can be tuned. [3]

5 CPython and the Global Interpreter Lock

When discussing Python it is implicitly assumed that the discussion is in regards to CPython. This paper is no different, but an issue particular to CPython is relevant. Alternative implementations of Python exist and are discussed in Section 8 so we will explicitly refer to CPython here.

Part of our technical goals in selecting Twisted and CPython is parallelizability. Parallel software is generally executed using either multiple threads, multiple processes, or both. Threads can be considered light-weight processes that operate within a process, possibly under the control of the operating system [13]. Threads can operate in multiple ways. They may very well be executed in parallel on a multi-core system such as a server, but they may also be executed in an interleaved fashion on a single-core system.

In CPython threads are unfortunately forced to operate in an interleaved fashion due to the global interpreter lock (GIL) [1]. The GIL forces the interpreter to operate on only one thread at a time. Even though Python threads are true threads in an operating system sense, they are byte-code interpreted by the interpreter. As a result, the interpreter itself can be seen like a shared resource. A thread must acquire the GIL before it can be interpreted itself. Due to this implementation true parallel processing is unavailable via threads. Multiple processes may still run in parallel as each process can run an interpreter, but parallel processes are more difficult to manage compared to parallel threads due to inter-process communication.

The GIL limitation results in increased complexity if a server application is to take advantage of a multi-core system which is a commonplace arrangement in today's server architectures.

6 Twisted Revisited

Fortunately the application server herd construction allows a way around the limitations of the GIL. Applications running on separate servers can just as well operate as independent processes on the same server. Communication between processes can then be abstracted as communication between servers allowing for reuse of pre-built communication systems. Twisted's object-oriented design already achieves this communication abstraction.

An unfortunate side-effect of treating separate processes as separate servers, if minimal rewrite is desired, is the duplication of data in memory. If memory bottlenecks become an issue then additional complexity will have to be added to the application to handle shared data between processes which would result in special cases for interaction between processes versus between servers.

Beyond its networking capabilities Twisted is also used by developers as a replacement for Python's standard `subprocess` library. Python's GIL affects only threads in the same process as the interpreter. Spawned subprocesses are still capable of taking full advantage of a multi-core system. Twisted provides its own means of handling subprocess spawning, particularly via `twisted.internet.reactor.spawnProcess`. It also provides implementations for remote procedure calls between processes and process interaction over communication protocols via `twisted.internet.protocol.ProcessProtocol`. [11, 4]

7 Alternative Event Driven Python Frameworks

Aside from the Twisted framework there are a number of other event based frameworks available for Python. These include gevent and Eventlet.

Both gevent and Eventlet operate based on green threads (greenlets in the case of gevent). These green threads are psuedo-threads in that they are threads implemented on the virtual machine rather than the operating system itself. Although these threads are lightweight, they appear to offer only a locking abstraction on top of regular threads [2, 7, 6]. Regardless, both are based on threads rather than processes meaning they would not take full advantage of a multi-core server for parallel processing due to the GIL limitations as previously discussed.

8 Alternative Python Implementations

Aside from alternatives to Twisted there are actually alternative implementations of Python. The standard implementation of Python is known as CPython. CPython, as its name suggests, is the Python language implemented in C as a bytecode interpreter. To address shortcomings of this implementation there are a number of alternative Python implementations such as IronPython, Jython, PyPy, and Stackless. Notable alternatives are discussed below. Despite the existence of these Python implementation alternatives, it should be clearly noted that frameworks such as Twisted are not immediately compatible with them due to various factors including changes to the standard library.

8.1 Stackless

Stackless is a Python implementation that is very thread-centric. It opts to make threaded programming in Python simpler while also providing cheap and lightweight threads [12]. Although at face value this appears to provide for true parallel execution with Python, it is instead a trade off between GIL with its automatic cooperative threading and manually controlled threading. The serial execution on one CPU limitation is still present in Stackless. Instead Stackless makes threading very explicit which encourages more maintainable and scalable code.

8.2 Jython

Jython is a Python implementation built to run on the Java Platform. It compiles Python source code to Java bytecode that can be run on a Java Virtual Machine JVM [9]. Due to its direct usage of the Java framework Jython's concurrency model is essentially Java's. Threads in Jython are mapped to Java threads and Java classes such as `ConcurrentHashMap` and `CopyOnWriteArrayList` are available. As a result, there is no GIL in Jython. [8]

9 Conclusion

For the creation of an application server herd the Twisted framework and Python look to be excellent candidates. Twisted provides excellent networking capability and, using object-oriented design, supports a variety of protocols. Python itself makes code prototyping and generalizing easy while making available a large variety of functionality through its standard library. However, there are limitations in the domain of multi-core utilization for high parallelizability due to CPython's global interpreter lock (GIL). The most important point is that there are a variety of directions to take starting at Twisted and Python. Alternative Python implementations such as Jython and IronPython can overcome the GIL limitation. Similarly, using Twisted to spawn and control subprocesses can take advantage of scalability across CPUs but also machines. With further research a particular design can be chosen with confidence.

References

[1] BEAZLEY, D. Understanding the python gil. `http://www.dabeaz.com/GIL/`, 2010.

[2] BILENKO, D. gevent. `http://www.gevent.org/`, November 2012.

[3] DIGI. Python garbage collection. `http://www.digi.com/wiki/developer/index.php/Python_Garbage_Collection#Introduction_to_Python_Memory_Management`, December 2010.

[4] ECKEL, B. Concurrency with python, twisted, and flex. `http://www.artima.com/weblogs/viewpost.jsp?thread=230001`, May 2008.

[5] EGGERT, P. Project. twisted twitter proxy herd. `http://cs.ucla.edu/classes/fall12/cs131/hw/pr.html`, November 2012.

[6] EVENTLET. Eventlet. `http://eventlet.net/`, November 2012.

[7] GREENLET. greenlet. `http://codespeak.net/py/0.9.2/greenlet.html`, November 2012.

[8] JOSH JUNEAU, FRANK WIERZBICKI, J. B. L. S. V. N. Chapter 19: Concurrency. `http://www.jython.org/jythonbook/en/1.0/Concurrency.html`, November 2012.

[9] JYTHON. General information. `http://wiki.python.org/jython/JythonFaq/GeneralInfo`, November 2012.

[10] LABS, T. M. Twisted. `http://twistedmatrix.com/trac/`, November 2012.

[11] LABS, T. M. Using processes. `http://twistedmatrix.com/documents/current/core/howto/process.html`, November 2012.

[12] STACKLESS. Stackless. `http://www.stackless.com/`, November 2012.

[13] WIKIPEDIA. Thread (computing). `http://en.wikipedia.org/wiki/Thread_(computing)`, November 2012.