CS 131: An Examination of Twisted Performance

Jonathan Woong
804205763
CS 131 – F16

Abstract

This project aims to explore the benefits and downfalls of Twisted Python in the setting of proxy server herding. More specifically, the fields of implementability, maintainability, and memory management are considered. A prototype is developed that implements server-to-server communication to test the feasibility of using Twisted in a large-scale setting. Node.js is considered to be a viable alternative to Twisted.

1 Introduction

Twisted is a framework written in Python that is focused on event-driven networking. The framework defines classes that make constructing and operating both clients and servers easily: *ClientFactory, ServerFactory, LineReceiver*. The goal of the project is to determine whether Twisted is a viable candidate to replace Wikimedia's backend. The servers implemented for this project communicate with their neighbors (flooding), eliminating the need to contact a database for client information.

2 My Server Herd Implementation

My implementation of the server herd can be run in BASH using the following:

```
$ python server.py SERVER_NAME
```

where *SERVER_NAME* is one of Alford, Ball, Hamilton, Holiday, or Welsh. One can run all of the servers at once using the *startserver.sh* script in the following way (after granting executable permissions):

```
$ ./startservers.sh
```

A client may then connect to a server with the following:

```
$ telnet localhost PORT_NUMBER
```

where *PORT_NUMBER* is the corresponding port number for the server.

The python script *server.py* defines the *server* class as a twisted *LineReceiver*. The *connectionMade* and *connectionLost* methods simply log the event that a client connects or disconnects to the server. The bulk of event handling occurs in the *lineReceived* method. The *lineReceived* method first checks for a non-zero argument line, then parses a message received from a client (or neighboring server) to determine the type of command being requested: *IAMAT, WHATSAT, AT*, or an unknown command. For every valid command, *lineReceived* first checks for the correct length of arguments and then checks for the validity of the arguments.

For handling *IAMAT,* the client's POSIX time is deemed valid if each character in its string is either a digit or decimal point, and that there is only one decimal point in the string. I chose this low-level validation method over using a *try-exception* block to decrease the amount of overhead necessary to detect invalid client inputs. This type of error checking makes appearances throughout the entire python script and is preferred for its speed. Python's *Design and History FAQ*[1] states that *try/except* blocks are "extremely efficient if no exceptions are raised. Actually catching an exception is expensive" so I opted to avoid *try/except* with the intention that Seunghyun will purposely test the code with incorrect client-to-server messages. If the server herd is used on a large scale with many buggy clients, this optimization may make a noticeable difference. However, for the case where the server code is lengthy and requires lots of validity checking, this may bloat the code and make it less readable. The client's coordinate position is checked in the same way. The client's identifier is then recorded and the time difference between the client time and server time is calculated. The *AT* response message is then constructed depending on the time difference and sent to the client. The server saves the client identifier, *AT* response, and client time in an list *clientData* to be referenced later. The server then floods the *AT* response to its neighbors using the method *updateNeighbor* (explained later).

For handling *WHATSAT*, the number of arguments, radius, and upper bound are all checked in the same manner as mentioned before (low level as opposed to *try*). The server then checks if the *WHATSAT* message refers to a client that the server already knows about, if the server does not know about the client, this is invalid and logged. If the server does have prior information for the client, it refers to *clientData* and extracts the corresponding *AT* message for this client. The *AT* message is then parsed to obtain the client position *clientCoord* and radius. The *clientCoord*, radius, and my personal Google API key are used to build a JSON query that is resolved using twisted *getPage*, *addCallBack*, and user defined method *handleQuery* (explained later).

For handling *AT*, the number of arguments is checked in the same way as before, and there is no further checking necessary because all *AT* messages received from a server must be formatted correctly at this point. The server parses the *AT* message and checks whether the client identifier and client time in the *AT* message is already known, and in this way avoids an infinite loop of neighbors flooding. If the client data was not known before, the client identifier, *AT* message, and client time are saved and passed on to neighbors using *updateNeighbor*, but not sent to the neighbor who originally sent the *AT* message.

The method *handleQuery* decodes the JSON response from google using *json.loads()* and then extracts the results field by subscripting into the decoded JSON response. The results are then bounded using *upperBound* as an upper subscript limit. The response message to the client is then built using *json.dumps()* and send to the client and logged.

The *updateNeighbor* method simply connects the server to its neighbors by connecting as a client that constructs with a message (the *AT* message to be sent to the neighbor). The connection invokes a *sendLine()* and *loseConnection()*. This implementation of connection, sending message, and disconnection is likely the bottleneck of the entire script, as it requires connection overhead to simply send one message. This can be remedied by establishing a persistent TCP connection between servers. The benefit of doing this is reduction of network resources and overhead. The downside of doing this is complicating the code, which is not necessary for a prototype.

The *serverFactory*, *client*, and *clientFactory* classes are responsible for defining the constructors for the server and client. Nothing particularly interesting or unusual lies in these blocks except for the fact that *connectionMade* in the client class invokes *sendLine()* and *loseConnection()*.

The main method simply checks for proper server usage and constructs the server using *serverFactory()* and *listenTCP()* in the usual twisted way.

3 Writing Twisted Programs

Writing Twisted programs, for this use case in particular, is extremely easy compared to other languages. Compared to C/C++, the need for memory management is eliminated. In C/C++, one is often required to create and bind to sockets and define a loop to listen to connections and handle them properly. This is overly complicated compared to Twisted, which only requires definition of two classes: one for constructing a server and one for server behavior. Client behavior is also just as simple. The Twisted framework also makes it easy to run a server, as it only requires *listenTCP()* and *run()*. Compared to Java, Twisted (or Python in general) requires fewer characters and requires less demanding hardware to run effectively.

4 Twisted Performance Implications

Although Twisted provides a powerful and simple framework to implement webservers, its performance is not as great as some other languages. In particular, Scala outperforms Twisted in a synthetic testing environment of 100,000 HTTP *GET* requests:

Server	Mean requests/second	Time per request (ms)	Percentage of requests served within a certain time (ms)			
			50%	75%	90%	100%
Scala	6,220	160.8	81	119	152	9,087
Twisted TCP	3,173	315.1	39	51	53	22,673

http://brizzled.clapper.org/blog/2009/03/09/scala-and-python-an-informal-tcp-performance-benchmark/

This makes sense, as Twisted is simply a framework written in Python that is tailored to provide developers with an easy way to implement servers whereas Scala is built from the ground up to be a scalable networking language. In other words, Twisted adds a layer between the developer and traditional Python by adding overhead of its predefined methods such as *lineReceived(), connectionMade(), connectionLost(), etc*.

When compared to Web Server Gateway Interface (WSGI) Python servers, Twisted commonly ranks among the lowest in performace[2]. When benchmarked against the other WSGI Python frameworks (aspen, cherrypy, eventlet, fapws3, gevent, gunicorn, modwsgi, tornado, uwsgi, gunicown-3w), Twisted ranked amongst the lowest for reply rate and response time of HTTP *GET* requests. Twisted also ranked highest for error rates in these benchmarks. These statements hold truth for both HTTP 1.0 and HTTP 1.1.

Implementing Twisted for the Wikimedia platform comes with a few considerations. One must decide whether the ease of writing Twisted servers is worth the performance penalty compared to other Python frameworks or even other programming languages.

5 Python's Type Checking

The fact that Python relies on duck typing may lead to issues down the road, especially if Twisted is applied to larger applications. Duck typing may harm the maintainability of large scale applications because it makes understanding the relationship between caller and callee more difficult. A developer who reads the code may not fully understand the types of the parameters being passed from one function to another until the code is actually run[3]. Another downfall of duck typing is not knowing whether a program will work correctly until one actually tries to run it. There is no compiler to do static type checking and alert of type mismatch.

The benefits of duck typing are simple: faster development cycles and cleaner code. These strengths are beneficial for prototyping and developing code that need not be 100% reliable[3].

Static type checking would alleviate much of the confusion when reading the source code, not to mention it would allow for (arguably) easier and quicker debugging. It would also give assurance to developers about the overall behavior of functions, and therefore the overall behavior of the system.

6 Python's Memory Management

The fact that Python memory management is handled by the Python Memory Manager (PMM) and cannot be controlled by the developer may lead to issues with integration in other systems. For example, attempting to manipulate Python objects with C functions such as *malloc(), calloc(), realloc(),* and *free()* will cause memory corruption[4]. In this sense, choosing Twisted to be used in a large application that interacts with other languages (especially those that allow developer memory management) may be unreliable and dangerous. Another flaw of the PMM is that a long running process, after reaching peak memory usage, will maintain the same amount of peak memory usage until its termination. This means that even if the average memory usage of a process is much less than its peak, the process will hog memory, which is wasteful and harms overall system performance[5]. Python also faces the issue of circular reference, but is remedied in CPython with a period, separate garbage collection routine.

Once again, the benefit of such an automated system is the reduction of the amount of code necessary to get a system running. This is beneficial for small applications and prototypes, or even systems where memory use is not a concern.

Since Python uses reference counting, there is a cost every time an object is referenced and dereferenced. The interpreter must check if the reference count for that object is 0, then deallocate the object if it is. Java's garbage collection avoids this problem, ultimately performing better in memory management. Additionally, garbage collection in Java can be delegated to a separate thread, increasing its performance in this region. The downfall of this approach is the complexity in implementing a separate thread as a garbage collector.

7 Python's Multithreading

In Python, the Global Interpreter Lock (GIL) prevents multiple threads from executing Python bytecodes at once, with the reason being that Python's memory management is not thread-safe. This leads to poorer performance in multithreaded Python applications[6]. Due to the fact that every variable in Python is mutable, there is no way to force immutability reliably. In this way, multithreading is not safe.

However, in the case of Twisted for our purposes, multithreading may be applicable. This is due to the fact that each server does not share data in the sense of shared memory. One server will not alter the readable/writeable data of another server, but instead simply make copies of the data received from other servers *AT* messages. It would be beneficial, then, to have one thread running for each server.

The only instance where a client may receive old *AT* data after sending a *WHATSAT* message is when the server that received the *WHATSAT* message receives an updated *AT* message at the same time that it responds to the *WHATSAT* message. Due to the asynchronous nature of Twisted, this event is not likely to occur, making multithreading a probable solution. The obvious benefit of multithreading is an increase in performance. The only drawback would be further complicating the code.

8 Twisted vs. Node.js

Node.js is an event-driven, non-blocking I/O runtime model built on Chrome's V8 JavaScript engine[7]. Twisted is similar to Node.js in that both are asynchronous and relatively easy to develop in. Another similarity is that both are supported by package managers, *pip* for Python and *npm* for Node.js.

One advantage that Node.js has over Twisted is that it generally requires fewer resources to run, allowing for more systems to utilize it[8]. Aside from the fact that Node.js is likely to have better performance than Twisted, there are no downsides to switching from Twisted to Node.js for the server herd application, as both languages have the same model and are relatively easy to code in.

9 Problems Encountered

The biggest issue I faced during this project was finding adequate benchmarking data for Twisted. Many of the articles I found were anecdotal and supplied little to no evidence for their claims about Twisted performance. The same could be said about Node.js. Even more difficult was finding performance data that compared Twisted and Node.js directly.

References

1: **https://docs.python.org/2/faq/design.html#how-fast-are-exceptions**

2: **http://nichol.as/benchmark-of-python-web-servers**

3: **http://beust.com/weblog/2005/04/15/the-perils-of-duck-typing/**

4: **https://docs.python.org/2/c-api/memory.html**

5: **http://www.evanjones.ca/memoryallocator/**

6: **https://wiki.python.org/moin/GlobalInterpreterLock**

7: **https://nodejs.org/en/**

8: **https://blog.geekforbrains.com/why-im-switching-from-python-to-nodejs-1fbc17dc797a#.ovqgusbbd**