CS 131 Project Report

Siddharth Joshi

Abstract

This report evaluates the suitability of implementing a Wikimedia style news service (where (1) updates to articles will happen far more often, (2) access will be required via various protocols, not just HTTP, and (3) clients will tend to be more mobile) as an application server herd with Python's *asyncio* asynchronous networking library. The report also compares Python's *asyncio* implementation to a possible implentation of an identical application in Java with explicit reference to the key concerns of Type Checking, Memory Management and Multi-threading amonst other things. Moreover, the report also attempts to briefly compare the Python architecture to a potential implementation using Node.js. The conclusion presented argues that Python's asynchronous networking library *asyncio* can be thought to combine performance efficiency rivaling that of Java, with functionality that is nearly on par with a framework like Node.js, and thus presents the conclusion that it is infact very well-suited for this kind of application.

1 Introduction

The Wikimedia platform currently uses a particular web stack architecture called the LAMP-stack architecture that is based on GNU/Linux, Apache, MySQL and PHP. LAMP uses multiple, redundant web servers behind a load-balancing virtual router for reliability and performance. While LAMP has been fairly successful for Wikipedia, there are 3 key differences in the type of application we wish to design:

- updates to articles will happen far more often
- access will be required via various protocols not just HTTP
- clients will tend to be more mobile

These critical differences would obviously not be best suited for the LAMP stack as, the application server would end up being a bottleneck. Since the clients are mobile, adding new servers would not be the most convenient and since the updates will happen far more often, response time is of prime concern and would be rather slow with such a central bottleneck. Thus, an application server head is suggested as an alternative, an organization where multiple application servers communicate directly to each other as well as through databases, shared caches etc. The main idea being to be able to adapt to the frequent and mobile requests, the servers should be able to quickly communicate certain rapidly evolving data (e.g. GPS-based locations or ephemeral video data) that doesn't need to be accessed very often or doesn't need to be submitted to the rigorous transactional semantics of a database entry. Thus, inter-server communication offers an effective alternative to deal with the common case of such 'less stable' data (data that is prone to frequent changes from mobile clients).

In this report, a model application server herd is designed to simulate an application with the desired properties using Python. The *asyncio* asynchronous communication library in particular is chosen to do so due to its potential to satisfy sufficiently the 3 additional key constraints as it offers a fast way of client-server and inter-server communication and would allow access via a diverse set of protocols rather than being limited solely to HTTP. Thus in the following sections, the report first discusses the specifics of the design and implementation, then moves to an evaluation of the efficacy of the *asyncio* asynchronous networking library, after which a detailed comparison of the implementation in Python is compared to a prospective one in Java, followed by a brief but effective comparison with a potential implementation in Node.js, finally concluding with the final verdict on implementing such an appication (one with such an application server herd) using Python's *asyncio* asynchronous netowrking library.

2 Design and Implementation specifics of Project Server Herd

The application chosen for this project is a simple one, that offers users the functionality to send their location to one of the servers in the application server herd and query what establishments are in proximity of a certain user. This application will only implement processing for the so-called 'unstable data' in order to implement the minimal product needed to demonstrate the effectiveness of such a stack for an application herd. The application server herd will 'flood information' received by any one server to all, in order to allow the user to freely query any server for the data. There will be 5 servers in the server herd, with the following ids: Goloman, Hands, Holiday, Wilkes and Welsh. The bidirectional communication links set up amongst the servers will be such:

- Goloman talks with Hands, Holiday and Wilkes
- Hands talks with Wilkes
- Holiday talks with Welsh and Wilkes
- Wilkes talks with Goloman, Hands and Holiday
- Welsh talks with Holiday

The two types of requests will be *IAMAT* requests with which users can inform the application of their location and *WHATSAT* requests with which users can query information about establishments in close proximity to the location of one of the users.

The goal is to evaluate efficiency but also robustness, and hence the implementation would have to guarantee that the application server herd is able to continue functioning even when individual servers in the herd fail (i.e. the remaining servers must gracefully handle the exception that may result from being unable to flood a received message to one of the servers.)

The *asyncio* asynchronous communiction library is used to start servers as desired that handle connections by reading the message carrying out one of the following actions or responding appropriately with a question mark and the invalid command. The servers are also intended to be as robust as possible, hence statements involving writing back a response to the client amongst other asynchronous networking tasks are wrapped in try except blocks with appropriate exception handling to prevent a client crashing before receiving the response or other such issues from affecting the servers.

IAMAT message processing
The IAMAT message format is:
IAMAT [client id] [coordinates] [time request was sent]
and the application server herd attempts to cascade the information a user sends to one of the servers in this format across all, while also passing along which server received the original message (the exact design and implementation of this is discussed in the Flooding Algorithm subsection). The IAMAT messages are processed by checking for valid first. This includes checking if the desired parameters have been passed, followed by checks to ensure that the parameters themselves are correct (e.g. co-ordinates in correct form). Finally, after confirming that a IAMAT request is indeed valid, the server in question stores the clients location, its own name (that of the receving server) along with the time the request was sent, difference in received time and sent time and co-ordinates in a dictionary. This is followed by the server responding to the client with an acknowledgement of the form:
AT [receiving server name] [difference in time sent and received] [client id] [coordinates] [time request was sent]
that attempts to confirm for the client that it's message was received correctly by the desired server and also convey the resulting latency. After this, the server in question triggers the flooding algorithm with the same response message it sent to the client to cascade all the relevant information.

WHATSAT message processing
The WHATSAT message format is:
WHATSAT [client-id] [radius] [number of resutls] where radius refers to the radius within which the client desires results and number of resutls specifies the upper limit on the number of results desired. This particular functionality is implemeted using HTTP requests to communicate with Google's Places API that allows users to search establishments within a certain radius of any coordinates. Naturally the constraints of the Places API are thus inherited by this application as well and thus the maximum radius within which such searches can be made is set to 50 km and the maximum number of results must also be bounded by 20. These constraints are checked for while checking validity of the request in general (in a manner similar to that of the IAMAT request). An additional validity concern here is checking whether the client being refered to has informed the system of his location earlier using an IAMAT request, in particular if this is not true, it is impossible to handle this request thus it is as such treated as invalid. If the request is valid, the server first responds with the original *AT* acknowledgement message that was sent to the client when it informed the application of its location. The client's request is then translated into an appropriate API request for Google's API using the stored location information, and the results are manually limited to the desired amount and then returned to the user. The HTTP requests are implemented using the *aiohttp* library.

Flooding Algorithm
Perhaps the most critical component of the entire project, the flooding algorithm is what enables all servers to share this

ephemeral data using inter-server communication rather than database accesses and stores. The key concern is deciding when to flood as flooding requires not only the original receiving server to send the message to all its peers but also for all peers (and transitive peers) to forward the message in a similar manner. However, while this may appear easy, a concern that appears almost immediately due to the bi-directional nature of the links is one of infinite flooding. If the servers naively forward all messages received the system will flood messages infinitely and cause the application to crash. Hence, the check implemented in the flooding algorithm is to ensure that the information being flooded is relevant, in particular that it provides new information. The assumption being that if the flooding algorithm uses this approach consistently, when it chances upon information that is not novel, it knows that said information was received earlier and flooded to all neighbours and thus there is no need to do so again. However, if the information is new (i.e. information about a new client or new location information from a client), it is desired that all servers be made privy to the information and hence it will be flooded. This simple check allows flooding to be finite and fairly efficient. There is of couse, some amount of redundancy still in the way this implemented but it is nearly inconsequential. It is possible to design more efficient flooding algorithms and in the final application, an efficient flooding algorithm may well be a prime concern due to the size of the ephemeral data being shared amongst servers, however, in this project, the size of the messages is nearly negligible and thus small multiplicative factors of inefficiency are benign.

3 Evaluation of *asyncio* Library

It is easy to see from the project application developed that the *asyncio* networking library makes implementing such an asynchronous application server herd rather seamless. The asynchrony allows servers to process multiple requests simultaneously as the server need not wait to completely deal with one request before it can start processing another (in some synchronous manner). Moreover, access to lower level tools like the event loops allows for fine-tuning the performance of the server herd by switching tasks effectively when certain I/O events are not "ready" yet. Thus, the control on performance that *asyncio* is actually rather comprehensive.

The same asynchrony that is an advantage of the *asyncio* networking library is also a possible pitfall for applications where sequential consistency is of importance. This problem is elucidated by the redundancies in the flooding algorithm where messages may be received more than once at times. It is possible also that since the *asyncio* networking library doesn't process requests in the order it received that an IAMAT request sent by one client followed by a WHATSAT request by another client referring to the same information may be deemed invalid, even though it was technically in the correct order due to the way that messages are processed by the *asyncio* library.

4 Python vs Java in Server Herd

The implemetation of the application server herd in Python with *asyncio* asynchronous networking library appears to be the ideal implemetation of such an application in Python, however, it remains to see how such an implementation compares up to possible alternatives implemented in Java. In particular, the concerns mentioned earlier regarding Type Checking, Memory Management and Multithreading are dealt with in the subsections that follow.

4.1 Type Checking

A key difference in the type checking of Java and Python is that Java is statically typed whereas Python uses dynamic type checking. There are inherently several trade-offs that result from this distinction. In particular, while static type checking provides benefits of reliability and potentially readability of code, dynamic checking makes writing the code far more convenient and helps abstract away some details, as well as simplify some concerns that prevent compilation in statically typed languages.

However, there are drawbacks to this convenience. Consider the example of the the function call to *asyncio.start-server()* that is arguably one of the most crucial function calls in the entire applicaion, however, a developer need not know what type it returns to be able to implement any arbitrary complicated application that uses said function. However, this does obscure the debugging process as in the case of incorrect execution or runtime/compile time errors: it is possible that reason is that the object returned is being passed to a reference of an incompatible type, but the programmer unaware of types struggles to catch this. A similar constraint faced personally, was while dealing with the time() function as the type it returns is float and the + operator cannot be used to concatenate arbitrary floats to strings (as is desired when constructing the reply message), however the lack of types made catching this particular bug far more challenging than it would have been in a dynamically typed language. Nonetheless, all in all, this particular concern is not that alarming and in fact the convenience and simplicity of dynamically typed languages may well be prefered by most in this case.

A potentially more dangerous drawback of dynamically typed langauges is run-time errors resulting from incompatible types that cause the servers to crash. This would be particularly disastrous and is a major concern to any developer intending to provide a reliable service. In this particular implementation, the potentially disastrous resutls of such a potential error were

entirely circumvented using try except blocks that implement effective exception handling, providing reliability in all situations, however, it is definitely desirable to catch all such bugs at compile time. Nonetheless, for most applications, effective exception handling could deal with this issue effectively, and in the suggested application where efficiency rather than robustness is the key concern, it is very likely that this convenience of the dynamic checking is worth the extra effort in setting up exception handling to avoid such unfortunate crashes.

4.2 Memory Management

Java and Python both use garbage collectors to deal with memory management, rather than having the developer manually deal with such issues, which immediately guarantees higher reliability since a majority of crashes while using languages that require custom memory management are often due to dereferencing of deallocated memory or not allocating enough memory etc. In fact, the use of garbage collectors eliminates a whole class of potential problems. Nonetheless, there are significant differences in the approaches to garbage collection taken by Python and Java. Java uses a mark and sweep algorithm while Python relies on reference counting.

In Java's mark and sweep, the memory is swept periodically from the root to detect reachable blocks that are so marked. After this, blocks that remain unmarked are evidently not needed as there is no way to refer to them and hence they are "swept up" by the garbage collector.

On the other hand, Python's approach deals with memory management more promply. Whenever a new refernce to an object is created, the reference count is incremented and when one is lost, the reference count is decremented. If the reference count of a block were to reach 0, the block is deleted. The blocks are also organized in a hierarchy of data strucutres called pools, blocks and arenas with the strategy being to re-use freed memory first before asking the system for more memory. The grouping of new memory requests into size classes is also intended to ensure that the memory is tightly packed and used completely.
Overall, what appears to be the conclusion is that memory management in Java is more efficient in terms of runtime with the simple algorithm of mark and sweep that doesn't seem to have an excessive overhead, however, it has the downside of memory of the system being held up for longer than it needs to be. On the other hand, Python, while less efficient in terms of runtime, is more space efficient and frees up memory as soon as it can while also ensuring that no new memory is request unless absolutely needed. In particular, the choice of language depends on the system and application being implemented which will in turn clarify whether performance or space is the primary concern.

4.3 Multithreading

Python uses a Global Interpreter Lock which essentially locks all Python objects and causes essentially all multi-threading to be lose any parallelism it may have, while Java has perhaps one of the most effective multithreading frameworks there exists. However, this problem too can be circumvented by the use of mutli-processing to essentially achieve the parallelism desired in Python. There are drawbacks to this, in particular, loss of access to shared memory etc. things that threads of one process can intrinsically share, however, they can be worked around using inter process communication and most applications can have their multi-threading needs satisfied by using multiprocesses.

5 asyncio vs Node.js in Server Herd

Node.js and *asyncio* are event-driven asynchronous approaches that use an event loop loop that provides fine grained control to scheduling according to the completion of I/O events. JavaScript is also dynamically typed lke Python, however, Node.js is built on top of the v8 engine whick makes it have slightly faster performance and also provides greater scalability. Moreover, the Node.js framework is designed to handle asynchronous networking whereas *asyncio* is simply a library in Python - a deviation from standard operation.

Node.js is also probably prefered by developers as front end of web stacks often involve JavaScript anyway and this results in a greater familiarity with this and easier integrations all over.
Finally, by virtue of being a newer development tool Node.js inherently is more attuned to the modern development environment's needs and desires, however, it would be unfair to say that Python with the *asyncio* asynchronous networking library doesn't in many ways provide nearly all the same functionality for most applications.

6 Conclusion

To conclude, it is easy to see that the *asyncio* asynchronous networking library is an extremely powerful, simple and convenient tool to use and allows developers to harness the intrinsic power and simplicity that is characteristic of Python to build complex server architectures.

While it is true, that Java may have better performance across several parameters in subtle ways and that Node.js too may provide greater functionality and power, it can be said that the ease and simplicity of Python along with the nearly negligible manifestations of these differences for most standard applications make Python an extremely viable candidate for such applications. Python, in some ways, seems to combine the

best of both worlds while providing the ease and simplicity it is known for as always.

References

[1] "Asyncio - Asynchronous I/O, Event Loop, Coroutines and Tasks." *18.5. Asyncio - Asynchronous I/O, Event Loop, Coroutines and Tasks - Python 3.6.4 Documentation*, Python Software Foundation, docs.python.org/3/library/asyncio.html.

[2] "Memory Management." *16.2. Threading - Higher-Level Threading Interface - Python 2.7.15 Documentation*, Python Software Foundation, docs.python.org/3/c-api/memory.html.

[3] Artem Golubin. "Garbage Collection in Python: Things You Need to Know." *Artem Golubin*, Artem Golubin, 7 Jan. 2019, rushter.com/blog/python-garbage-collector/.

[4] Wouters, Thomas. "Global Interpreter Lock." *Global Interpreter Lock*, Python Software Foundation, wiki.python.org/moin/GlobalInterpreterLock.

[5] "Python vs Node.js: Which Is Better for Your Project." *7 Best Social Network Software. How Much Does It Cost to Build Own?* | DA-14, da-14.com/blog/python-vs-nodejs-which-better-your-project.