Use of Python ‘s asyncio Library to Program Servers in an Application Server Herd

Morgan Madjukie
UCLA Student, COMSCI 131

Abstract

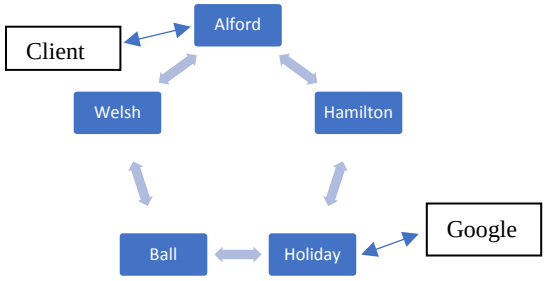
The asyncio library is a great candidate for use in a lightweight, simple server in an application server farm due to it’s single threaded nature and ability to schedule asynchronous tasks to prevent blocking. Particularly, the asyncio library is able to abstract the notions of asynchronous programming in order create a simple interface on which a server can be easily coded. However, care must be taken to ensure that asynchronously scheduled tasks do not create data races when accessing local data. In addition, the library is a poor candidate for multithreading as the main event loop is not thread-safe. Python’s duck typing requires additional high-level type checks to be performed as part of the asyncio library and the server program to ensure program safety, further slowing down performance.

1. Header

The asyncio library was introduced in Python 3.4 toto replace the Twisted framework popularly used in Python 2. Unlike the Twisted framework, the Asyncio library is part of the Python 3 Standard Library. This not only allows it to take advantage of Python 3’s improvements over Python 2, but allows gives it support from the Python Software Foundation and cohesion with the rest of the Python library.

To test how well asyncio can be used as a server foundation in an application server farm, a network consisting of 5 servers was built. Within these 5 servers, attention was focus on how simply messages can be processed, servers can communicate, and asynchronous tasks be scheduled. An analysis of the asyncio library source code was also performed in order to determine additional performance characteristics of the asyncio library.

2. Network Setup



The test network consists of 5 servers named Alford, Hamilton, Welsh, Ball, and Holiday communicating with each other as shown above. Each server should share similar architecture, with the only difference being their names and their locations. Each server must be able to receive information about the client and then propagate this information to all other servers using a flooding algorithm. Once this has happened, a client can ask any server about locations around another client that has reported its location to the network. The server that receives this request must be able to query the Google Places API directly for this information and not ask any other client for information.

3. Implementation

3 Message formats are used to implement the aforementioned behavior. IAMAT, which takes a clients information and propagates it to all other servers, WHATSAT, to query the Google Places API for information, and AT, to facilitate propagation of client information. In an invalid command is sent to the server, it is echoed back to the client with the format ‘? <message>’. The server itself integrated within an event loop. The infrastructures of both the event loop and the server are provided by asyncio.

3.1 Foreword

Some portions of the asyncio library, such as parts of the BaseEventLoop class and the Transport. class no not include a library defined implementation, even though they may have behavior defined by the specifications. This means that performance for the library can vary by interpreter implementation. That being said, this does open the option of modifying the library in order to optimize performance or achieve a certain desired event loop/ connection behavior, possibly at the cost of reduced reliability. One option is to remove semantic type-checks that

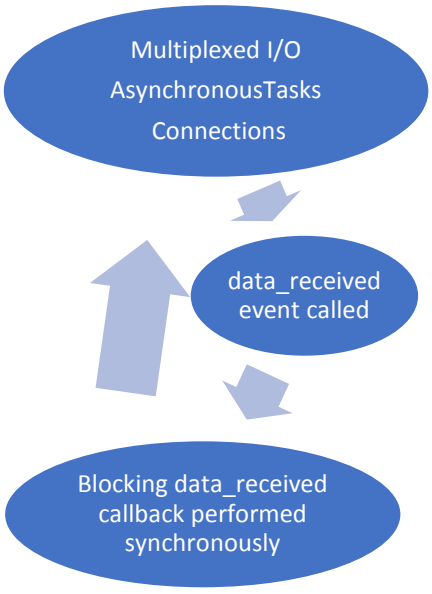
ensure reliability. However, modifications to the asyncio implementation are out of the scope of this test.

3.2. Network Implementation

3 Message formats are used to implement the aforementioned behavior. IAMAT, which takes a clients information and propagates it to all other servers, WHATSAT, to query the Google Places API for information, and AT, to facilitate propagation of client information. In an invalid command is sent to the server, it is echoed back to the client with the format '? <message>'. The server itself integrated within an event loop. The infrastructures of both the event loop and the server are provided by asyncio.

3.3. Server Basis: Protocols & Event Loops

The server is built using the Protocol interface as defined by the asyncio.Protocol base class. The class defines a set of callbacks which are called when an event occurs on the server, the most important of which is the data_received(self,data) method. My definition of the Protocol class is passed into the loop.create_method as a callable. For every connection made to the server, a new Protocol instance is created. The intent is that the Protocol is used to parse incoming data received by the connection, which is abstracted by the Transport class. In turn, the Transport handles any network I/O required by the Protocol.



The concept of the event loop as used with the Protocol is as follows: When a connection is made, a Protocol instance is created to serve that connection. After this, the event loop reads data from that connection. This is asynchronously performed with reads from other connections and coroutines that have been scheduled. (See 3.5 WHATSAT and 3.6 Information Propagation (AT)). Once all data from that connection has been received, the reader readies the associated callback. This is the reaction caused by the associated 'event' of a read finish. The event loop then executes the callback synchronously its next available iteration. Any Tasks scheduled follow a similar pattern: a portion of the Task is completed asynchronously, and then sends an event signal that informs the scheduler that it is ready to execute the next step of the task.

The use of "event-driven" to describe asyncio's event loop implementation is actually somewhat misleading. What actually happens is that a scheduled callback records the time it finishes. During a single iteration of the loop, the event loop collects all finished callbacks and executes any callbacks associated with the finished callbacks. This type of implementation is optimized for multiple asynchronous tasks with many order-independent callbacks as 1 loop iteration can perform many callbacks. However, this results in poorer performance for small groups of asynchronous tasks with deep coroutine calls (i.e many instances of coroutines within coroutines), due to one loop iteration required to perform each callback.

3.4. IAMAT

The IAMAT message is used by clients to inform the network of theirselves. It has the following format:

IAMAT <ClientName> <Geocoordinate> <Time>

Where Geocoordinate is given in ISO 6709 notation and Time is the time at which the message is sent by the client expressed in POSIX time with nanoseconds.

The IAMAT command serves as a basis for understanding how a callback is synchronously performed. After all data has been received by the client, the data_received callback is performed, which parses the message and replies to the client. The synchronous, uninterrupted nature of the data_received callback is shown by the following logs:

2017-12-03 19:57:09,582 - Alford - INFO - Is IAMAT Message
2017-12-03 19:57:09,583 - Alford - INFO - Valid AT message
2017-12-03 19:57:09,583 - Alford - INFO - Updating self and network

2017-12-03 19:57:09,583 - Alford - INFO - Flooding network with AT Message
2017-12-03 19:57:09,583 - Alford - INFO - Answering Client at: 57548
2017-12-03 19:57:09,583 - Alford - INFO - Parsing finished for Client at: 57548

A valid IAMAT message returns an AT message to the corresponding client. More information on how server information is propogated can be found in 3.6. Information Propogation (AT).

3.5. WHATSAT

The WHATSAT message simulates a server's connection to an external network. In this case, it is a query to the Google Places API. It has the following format:

WHATSAT <ClientName> <Radius> <Places>

Where Radius is the Radius of the search area around ClientName's position and Places is the number of places returned by the command. The server will reply to the client with an AT message with ClientName's information and a json formatted Google Places API result.

The async.open_connection method is used to connect to the Google Places API. This is essentially a wrapper for AbstractEventLoop.create_connection that returns a reader, writer pair that can be easily used to communicate with the server. The issue with opening a connection is that the server may need to wait for the connection to be established and for results to arrive from the server. This will block event loop execution if executed synchronously. The solution to this is to parse the message in the data_received callback, but to place the Google APIs connection and client response inside a coroutine. This coroutine is scheduled in the event loop using asyncio.ensure_future() as a Future, a type of Task. As a result, the data_received callback can finish completely without blocking the event loop. The following log file shows this:

2017-12-03 19:57:18,310 - Hamilton - INFO - Is WHATSAT Message
2017-12-03 19:57:18,310 - Hamilton - INFO - WHATSAT is valid
2017-12-03 19:57:18,310 - Hamilton - INFO - Creating Task to request to google with Details 10 +34.068930,-118.445127
2017-12-03 19:57:18,310 - Hamilton - INFO - Parsing finished for Client at: 32854
2017-12-03 19:57:18,310 - Hamilton - INFO - Connecting to Google Places API for Client at 32854
2017-12-03 19:57:18,413 - Hamilton - INFO - Successful Google Places API connection for Client at: 32854
2017-12-03 19:57:18,413 - Hamilton - INFO - Sending Google GET request for Client at: 32854
2017-12-03 19:57:18,413 - Hamilton - INFO - Rceiving HTTP response for Client at: 32854
2017-12-03 19:57:18,575 - Hamilton - INFO - Parsing HTTP respose for client at: 32854
2017-12-03 19:57:18,575 - Hamilton - INFO - Writing json to Client at: 32854
2017-12-03 19:57:18,575 - Hamilton - INFO - Closing Client Connection at: 32854

However, there still lies the problem of waiting for the Google Places API connection to be established, as well as waiting for messages to be written to the API and received from the API. The solution to this is the 'yield from' keyword. This is implemented as a Python feature (See PEP 380) and has an opcode, to be used as follows:

yield from <coroutine>

The open_connection method and reading methods are both coroutines. When they are called using yield from, the main coroutine containing the yield from is paused at its location and transfers control to the called coroutine. This point is also used as a callback point for the event loop. In a sense, it yields to the event loop and transfers program execution to the callee. It is the called coroutine's responsibility to complete its asynchronous execution in the event loop and call the next segment of the main coroutine after creating a result. This allows the connections, reads, and writes to be performed asynchronously in a non blocking manner. In theory, were another client to send a message to the server, the message can be parsed between the connection to the Google Places API, the HTTP request to the API, and the HTTP response to the API, as well as right after it. As the 'yield from' keyword is a part of the Python 3.4+ , a high degree of performance is expected.

Of note is that the coroutine has to take on the duty of closing the connection to a client. This is because the Protocol is unable to determine at what point the coroutine finishes. As such, it is unable to (and dangerous) to close the connection at the end of the synchronous data_received callback as messages may still be sent over the server.

3.6. Information Propagation (AT)

The AT message is used to return information to clients, as well as relay information to other servers. It is formatted as follows:

AT <RecipientServer> <ClientName> <Geocoordinate> <Time>

The contents of the AT message depend on the message sent to the server. For an IAMAT message, the AT message to the client has RecepientServer as the Server that receives the IAMAT, and Geocoordinate and Time fields as reported by the IAMAT message. For a WHATSAT message, the AT reply has the RcipientSErver field as the Server that received the IAMAT from the requested client, ClientName as the client to be searched around, and the Geocoordinate and Time fields contain the latest information available about the requested client.

The AT message is also used to propagate information about clients. A server only propagates Client Information if it receives an IAMAT message or an AT message from another server that has a client timestamp newer than what is already on the server. In both cases, the RecipientServer field contains the Server that received the intial IAMAT message.

Similar to the Google Places request performed by the WHATSAT message, the connections for propagating client information must be performed asynchronously to prevent event loop blocking. There are two issues that are specific to client information propagation. The first is that the destination server that the AT message is being sent to is inactive, in which case we should skip sending the message. This is done using a try-except block as open_connection raises an exception when a connection is denied. The second issue is that no writing to local server data allowed inside coroutines as unintended program order may occur. For instance:

```
#say the first read is 45 and var a is some program global
a = yield from reader.read()
b = 3
w.write ((b +’’).encode())
#warning: value of a can change between before w.drain() and end w.drain() as it is asynchronous.
yield from w.drain()
print(a+b) # is it 453? Or some other value?
```

In order to combat potential data races, it is imperative that all accesses and writes to local server data must occur synchronously. In the case of AT message propagation, all checks and local data updates are performed before a pre-formulated message is passed to the coroutine and sent to each server. If the message was formulated inside the coroutine, there is the possibility that the message sent may differ from what was expected. The following log files showcase this behavior for the IAMAT recipient:

```
2017-12-03 19:57:09,583 - Alford - INFO - Updating self and network
2017-12-03 19:57:09,583 - Alford - INFO - Flooding network with AT Message
2017-12-03 19:57:09,583 - Alford - INFO - Answering Client at: 57548
2017-12-03 19:57:09,583 - Alford - INFO - Parsing finished for Client at: 57548
2017-12-03 19:57:09,583 - Alford - INFO - Propogating info to server 'Hamilton' at port:19626
2017-12-03 19:57:09,585 - Alford - INFO - Successful Propogation to 'Hamilton'
2017-12-03 19:57:09,585 - Alford - INFO - Propogating info to server 'Welsh' at port:19627
2017-12-03 19:57:09,586 - Alford - INFO - Server: 'Welsh' is unavailable
```

And the AT message recipient:

```
2017-12-03 19:57:09,585 - Hamilton - INFO - Message received: AT Alford 32945945.190711737 kiwi.cs.ucla.edu +34.068930-118.445127 1479413884.392014450
2017-12-03 19:57:09,585 - Hamilton - INFO - Is AT Message
2017-12-03 19:57:09,586 - Hamilton - INFO - Flooding network with AT Message
2017-12-03 19:57:09,586 - Hamilton - INFO - Parsing finished for Client at: 32846
2017-12-03 19:57:09,586 - Hamilton - INFO - Propogating info to server 'Alford' at port:19625
2017-12-03 19:57:09,587 - Hamilton - INFO - Successful Propogation to 'Alford'
2017-12-03 19:57:09,588 - Hamilton - INFO - Propogating info to server 'Holiday' at port:19629
2017-12-03 19:57:09,589 - Hamilton - INFO - Server: 'Holiday' is unavailable
```

4. Conclusions and Considerations

From building this server infrastructure for a simple network, I can conclude that the asyncio library should be suitable for use in a lightweight serve within an application server herd in which servers receive and echo gps locations. This is because of strong loop performance in what is perceived to be many similar asynchronous tasks which do not depend on each other or perform deep coroutine yield froms. It is also easily expandable as new message parsing can be added by simply adding another branch within the data_received callback of the server protocol. Last but not least, Python features native support for generators and coroutines, which should allow for faster performance as opposed to other non-python asyncio implementations. Memory wise, Python

refence count based garbage collection should ensure that unused objects are immediately made available as memory space. This is especially helpful as Protocol instances are constantly being created and destroyed. New Protocol instances can occupy spaces left by dead Protocol objects without much respacing required.

However, precautions must be taken to ensure high performance and lack of data-races. In particular, the performance of an unmodified asyncio library may slightly suffer. The first reason for this is that python's duck-typing means that high-level sematic checking is included in asyncio functions to ensure the argument passed has the correct type. Of course, these may be removed to increase performance at the cost of reliability. Secondly, asyncio in its base form is mostly incompatible with multithreading, as specified by the documentation itself. Even though the library may be locally modified to run callbacks on other threads, the main event loop, as well as reading and writing global variables/mutable objects is not thread-safe. As such, coroutines are still restricted from reading and writing to globals.

A possible alternative framework that can be used to implement a server which is a part of an application server herd is the Node.js Javascript runtime. Unlike Python's asyncio, which is designed to schedule asynchronous local tasks as well, Node.js is built specifically for web applications. It provides an even more abstract event-driven interface for building a server. In addition, it is truly event-driven as opposed to asyncio: an actual event signal is propagated throughout the Node.js environment when a method finishes. This means that callbacks occur as soon as possible after the event is called.

However, this purely event-driven methodology has some severe performance and usage drawbacks. First, event signal propagation is likely to be expensive, which slows down performance in all cases except on low number of events. Secondly, you cannot schedule your own asynchromous tasks as conveniently in Node.js. Doing so requires the understanding of the TimeOut method, which is far more complicated and not as efficient as Python's event loop scheduling due to potential gaps of doing nothing. This, along with Node.js's poor support of multithreading, makes multithreading bery difficult to utilize. Last but not least, the Javascript library itself, including Node.js, is very sparse. Doing anything complicated, such as implementing coroutines, will require some degree of dependency hell. In short, Node.js specializes in servers with only synchronous callbacks, which is unsuitable for the task at hand.

References

[1] asyncio — Asynchronous I/O, event loop, coroutines and tasks, https://docs.python.org/3/library/asyncio.html

[2] CPython asyncio Source Code, https://github.com/python/cpython/tree/3.6/Lib/asyncio/

[3] Python 3.6.3 Memory Management, https://docs.python.org/3/c-api/memory.html

[4] Node.js API Documentation, https://nodejs.org/api/