Viability of Asyncio as Framework for Application Server Herd

Jesse Chen
cs131

1 Introduction

An application server herd is a type of architecture where multiple servers in a ”herd” communicate with each other to synchronize data without needing to communicate with a central database. Normally, implementing such a program is quite a difficult task, but using the Asyncio library in Python 3.6.3 dramatically simplifies the process. Python’s duck typing and reference-counting memory management scheme in conjunction with the Asyncio library’s event-driven, asynchronous nature significantly reduced the programmer workload needed to create a application server herd.

2 Application Design

The application was written in several files for the sake of simplicity. ServerClass.py and ClientRecord.py contain the code for the Server and ClientRecord classes, and server.py contains the main method. The Server class represents a server that listens at ’localhost’ on a specified port, and handles any messages received on said port. server.py instantiates the Server herd and calls each Server object’s `open_server()` function, which starts the server on Asyncio’s event loop. Asyncio’s `open_connection()` and `start_server()` functions offer a layer of abstraction that make writing server applications a breeze.

The majority of the work is done by the Server class, within the callback for each client connected. The client-callback function must correctly handle all messages received, ensure validity of received messages, produce proper response to client, and propagate any database changes across the network. At the same time, log files must also be correctly generated and written to accordingly. Each Server contains its own copy of the database and keeps it updated in sync with received IAMAT and UPDATE messages.

In order to increase readability and ease of maintenance, the client-callback function contains three main components:

1. Handle IAMAT message
2. Handle WHATSAT message
3. Handle UPDATE message

```
if word_list[0] == 'IAMAT':
     await self.handle_IAMAT(writer, word_list, message)
elif word_list[0] == 'WHATSAT':
     await self.handle_WHATSAT(writer, word_list, message)
elif word_list[0] == 'UPDATE':
     await self.handle_UPDATE(word_list, message)
```

2.1 `handle_IAMAT()`

2.1.1 Parameter Checking

A correctly formatted IAMAT message received by the Server should contain the client id from which it was sent from(hostname or ip addr), the client’s supposed GPS location, and the timestamp at which the message was sent from the client. The purpose of a IAMAT message is to update the Servers with a client’s coordinates.

Since the message was received as a string, care must be taken to ensure that the values passed in are valid. This is done by attempting to cast the parameter into type it should be within a try-except block. This method of simple parameter checking is made possible through Python’s use of duck typing: in which the interpreter doesn’t check whether the types match, but rather if the action is allowed on an object of that type. For example:

```
try:
    ts = float(word_list[3])
except:
    #the string in word_list[3] is not a valid float
```

In a way, this style of simple type-checking is an application-level static type-checking that is similar to the type-checking employed in languages like C and Java.

If this style of simple type-checking fails, the code will simply execute the except block and in this case, the except block would be used to handle the error.

As can be seen in the example code, the timestamp is checked by attempting to cast it to a float. Should the try block fail, it would indicate that the timestamp parameter is invalid. The GPS parameter is checked in a similar way as the timestamp; however, the GPS parameter has additional requirements that must be met. Specifically, the GPS parameter must also conform to ISO 6709 notation. This was accomplished by checking the positions of the unary positive and negative operators, using those positions to determine the locations of the numerical values of the coordinates, and checking whether the numerical values are valid floats using the simple checking described above.

If any of these parameters are invalid, the Server will simply echo back "? <invalid-message>" to the client, in which <invalid-message>is the message that was originally sent by the client and received by the Server.

2.1.2 Client Response

Assuming the received IAMAT message was valid, the Server generates an AT message to send back to the client. The AT message should be of the format:

```
AT <server_id> <time_diff> <client_id> <GPS> <ts>
```

2.1.3 Database Update/Propagation

Depending on whether a record for that client already exists in the database, the Server will now either create a new record, modify an existing record, or do nothing(if the record already exists). Not only is whether the record already exists a deciding factor in how the database should be updated, the database also is updated depending on whether the received timestamp is more current than the one (potentially)already in the database. Only the newest location updates are written to the database and propagated to the other Servers.

Assuming the received IAMAT results in a new/modified record, the Server floods all its neighbors with an UPDATE message in order to propagate this database change across the network Information about a Server's neighbors are stored within an internal data structure that keeps track of neighbors' hosts and ports. An UPDATE message contains all the information within an AT message, as well as an identifier for the server it was sent out of. Being able to identify which Server the UPDATE was sent out of is crucial to the flooding algorithm, as will be explained in the UPDATE section.

2.2 `handle_WHATSAT()`

A correctly formatted WHATSAT message received by the Server should contain the client id that it is querying about, a radius around the client's GPS coordinates, and the number of search results that are being requested. The purpose of a WHATSAT message is to query the Servers about locations near a client's coordinates.

2.2.1 Parameter Checking

Similar to the `handle_IAMAT()` function, the parameters of the WHATSAT message are also checked for validity. The client id is checked to be a valid id within the database, the radius is checked to be a number between 0 and 50, and the the number of search results is checked to be a number between 0 and 20. Invalid WHATSAT messages are handled the same way as invalid IAMAT messages.

2.2.2 Client Response

Assuming the received WHATSAT message was valid, the Server looks up the last AT message that was sent back to the client and sends another copy of the most recent AT message. The AT message should be of the format:

```
AT <server_id> <time_diff> <client_id> <GPS> <ts>
```

The Server should also respond with a JSON format message that is the response of a Google Places Nearby Search request.

2.2.3 Google Places Query

The Google Places Nearby Search parameters should be the GPS coordinates of the client on record, and the radius given in the WHATSAT message. Using these parameters to generate an endpoint URL, the Server makes an asynchronous call to open the connection and receive the webpage in JSON format.

2.3 `handle_UPDATE()`

A correct UPDATE message received by the Server should contain the client id to be updated, the Server that originally received the update, the client's new GPS coordinates, and the Server from which this UPDATE message originated from. The purpose of an UPDATE message is to inform the Server/network about updates to a client's record.

For the most part, an UPDATE message contains the same information as an AT message, with an extra bit of information to ensure correct forwarding.

2.3.1 Forwarding

Upon receiving an UPDATE message, the Server checks if its own records are up-to-date or need to be updated. If the record is already up to date, the Server assumes it has already forwarded the information and does nothing. On the other hand, if the record update is new, it will update its record, then forward the UPDATE message(modified to indicate itself as the origin) to all its neighbors except the one the message was received from.

3 Python vs Java

Compared to a Java-based approach, a Python-based approach offers significant advantages that allow for ease of writing, use, and maintenance of the code.

3.1 Typing

Python employs Duck Typing, a type of dynamic typing where it does not matter so much whether an object is of the right type, as long as the object is capable the action described in the code. Because of this, Python is extremely flexible in ways that statically typed languages(e.g. Java) are not. This flexibility leads to code that is easy to write and almost pseudocode-like in its simplicity. On the other hand, the extreme flexibility of Python can also be a downside, because when type errors do occur, they are not caught at compile time the same way they are caught by the Java or C compilers, but rather happen during execution of the program.

3.2 Memory Management

Python's memory management system uses a combination of reference-counting and mark-and-sweep to keep track of objects. In using a reference-counting memory management scheme, Python ensures that objects that are no longer referenced are quickly removed from memory, instead of having to wait for a garbage collector to free up memory. Python also employs a mark-and-sweep garbage collector to collect objects with cyclic references. By using the two in conjunction, Python gets the advantage of reference-counting while also making up for reference-counting's shortfalls with a mark-and-sweep garbage collector.

3.3 Multithreading

Asyncio's event-driven asynchronous, single-threaded model allows for a much more consistent performance metrics across a wide range of machines in comparison to Java's parallel multi-threaded model of execution. A Java implementation would depend more on the hardware of the machine than a Python implementation due to Java's multi-threaded reliance on CPU power. Asyncio's asynchronous nature is much more suitable for a server herd application than a Java parallel approach.

4 Asyncio vs Node.js

Asyncio and Node.js are both very similar given that they are both asynchronous, event-driven, single-threaded frameworks built for back-end network development. Node.js is JavaScript and runs on the V8 JavaScript Engine, while Python's Asyncio runs on the Python interpreter.

Both Asyncio and Node.js have an event loop from which callbacks are performed.

5 Conclusion

In conclusion, Asyncio has proven to be an effective and easy framework on which to build an application server herd. Asyncio provides an elegant layer of abstraction that significantly increases the ease with which server applications can be built. Python is also a clean and simple language that offers ease of writing, maintenance, and scalability over Java. I strong recommend the use of Python/Asyncio to build an application server herd. Although peak performance using Python and Asyncio might be less than a Java implementation of an application server herd, the Python/Asyncio approach offers robustness, stability, and significant simplification of the developer's(me, you, us, etc) task.