Numbers server application (python)

Albert Nadal Garriga
18/02/2022

In this document I try to explain some thoughs and decisions regarding the implementation I made to solve the task. I divided this document in different sections, taking in cosideration only the topics that are relevant to better understand the code and the decisions I made that achieve a good performance.

Installation and usage

I strongly recommend to run the application with versions 3.7 or 3.8 of the Python interpreter, as those are the versions I used during the development. First download the code by cloning the git repository to you computer.

git clone https://github.com/albertnadal/numbers_server

I tried to use only the built-in standard libraries as much as I could. However I used two external packages to avoid spending time reimplementing the wheel. Those external packages are aiofiles and bintrees, later I will explain in depth why I'm using that dependencies. Please, install the dependencies with the command below.

pip3 install -r requirements.txt

Now you are able to run the application just running the server.py script as described below. Note that no parameters are needed to run the application.

python3.8 server.py

I decided to avoid the use of parameters from the command line for the sake of simplicity. You are able to manualy edit some code variables, like the address, port, max connections, log filename or the report periodicity. Below you can see the default values.

ADDR = 'localhost'
PORT = 4000
MAX_CONNECTIONS = 5
LOG_FILENAME = 'numbers.log'
REPORT_PERIODICITY = 10

There are also available two simple scripts I implemented in order to interact with the server. Open a new tab in the terminal and run the ***client_numbers.py*** script if you want to start sending valid number sequences. Run this script in a new tabs to run more instances concurrently.

python3.8 client_numbers.py

You can send a 'terminate' sequence to the server just by running the ***client_terminate.py*** script.

python3.8 client_terminate.py

In order to run the tests follow the instructions below.

cd tests
python3.8 -m unittest -v Server_Test.Integration

Some initial considerations

There are some things you should take in consideration. The LF (\n) is the escape sequence of my choice, so the server will only accept number sequences with the '\n' newline character as valid. The source code has a large amount of comments explaining in depth some topics that are roughly detailed in this document. I encourage you to take in consideration inline comment in the code.

Concurrence and parallel processing

I use the built-in '**asyncio**' library in order to add the desired concurrency behaviour to the application. Unfotunatelly Im not enough familiar with the '**multiprocessing**' library that Python provides.

Due to the lack of time to better understand the documentation I decided to follow the aproach of using only one asyncio loop in a single process. When dealing with parallel computing I'm much more familiar with other programming languages natively built around concurrency and parallelism. Being said that, the fact that Im not taking profit of all the available virtual CPUs of the computer have an impact in the final performance. I hope I had more time.

Data structures

I used a BST (binary search tree) as a data storage structure to manage all the numbers the server receives. I also used a simple python list as temporary memory buffer storage for writting unique numbers to the numbers.log file in a performant way. And I also use an array of length 3 to store the counts for the current report.

A binary search tree is a very simple data structure that provides search, read and write operations with an average computation cost of O(log(n)). Instead of making a new implementation of a BST I used the '**bintrees**' package. I could port to Python an implementation of a BST I made in C++ some time ago (https://github.com/albertnadal/binary-search-tree), unfortunately I have not enough time.

With the BST I can quickly check if a number has been already received in the past and, also check if the number is unique or not. New numbers are also inserted into the tree really fast.

However, BSTs have some pros and cons. The average computation cost of an operation is O(log(n)) as long as we receive the numbers unordered. The worst case scenario comes when we receive and add ordered numbers into the BST. Adding ordered sequences of numbers generates an unbalanced tree and that implies that the cost of any operation is O(n), wich may potentialy cause a denial of service of the application. In order to mitigate this we can use other variants of BSTs that are autobalanced, or balance the tree periodically (during the report generation for example).

I use an array of 3 integers to store the data that should be reported every 10 seconds. Every time a consumer worker receives a unique or a duplicate number then the values of the array are properly updated.

Synchronization

Running multiple socket connections that perform read and write operation on shared resources may cause potential race conditions and unexpected behaviours. The use of synchronization primitives is the proper way to deal with shared resources in a concurrent environment.

All the data structures mentioned in the previous section require the use of *Lock* (https://docs.python.org/3/library/asyncio-sync.html#asyncio.Lock) and *Event* (https://docs.python.org/3/library/asyncio-sync.html#asyncio.Event) synchonization primitives. It means that every data structure that is subject to read and write operations must be locked every time a concurrent task can perform a read or write operation.

In some way synchronization among concurrent tasks has an impact on the global performance. There are some techniques and tricks to mitigate or avoid synchronization overheads, for example, provide individual BSTs for each concurrent task and, later, during the report generation, process all the BSTs to update the master BST. This should increase the speed during the 10 seconds and extend the time to generate the report.

There is another synchronizing operation that must be performed during the life of the application. Every 10 seconds all the concurrent stream workers should be immediatelly paused in order to avoid updating the array containing the current report values. Once the report has been generated all the stream works should be resumed.

In order to lock all the workers right before running the report generation the application send an asyncio Event synchronization statement. All the stream workers stay locked until the report generation task unlocks the Event to resume all workers.

Writing logs to disk

Performig massive IO operations is really expensive,whatever the programming language you use. Continuous writting operations of small chunks of data is extremely unperformant. The time needed to write a small chunk of data is practically the same that writting a huge amount of data.

According to this fact I decided to store all the numbers that should be written in the numbers.log in a memory buffer (append operations in a python list is faster than writing to disk, despite the mem allocs). This memory buffer is also a shared resource and thus protected with the *Lock*.

Right before printing the current report to the stdout I join all the strings of the list buffer and write the result at the end of the file in a unique writing operation. Joining a list of strings is really fast in python and writting contents to disk is also really fast if you perform only a single disk IO operation.

I used the '**aiofiles**' package to manage files.

Testing

I'm familiar with the build-in unittest framework that comes with Python, so I decided to write some integration tests using this aproach. I know there is a lot of corner cases that can be tested but I just tested only a bunch of scenarios, so I have not enough free time out of my daily work. I can implement a large suite of test cases but my intention here is to show a quick, simple and reusable aproach fer testing the application.

Each individual test launches an OS process running the ***server.py*** in a shell using the built-in '**subprocess**' package. The *setUp()* method launches server process prior to the execution of each individual test. The *tearDown()* method kills the server process when an individual test finishes (even if the test failed or succeeded).

The bunch of individual tests I implemented are self explanatory.

```
test_GivenServerIsRunning_WhenConnectingToTheServerAddress_ThenConnectionSucceeds
test_GivenServerIsRunning_ThenTheServerAllowMultipleLiveConnections
test_GivenServerIsRunning_WhenReachingTheMaxAllowedActiveConnections_ThenNewConnectionsAreAutomat
icallyClosed
test_GivenServerIsRunning_WhenClientSendAnInvalidNumberSequence_ThenTheServerDisconnectsTheClient
test_GivenServerIsRunning_WhenClientSendValidNumberSequences_ThenTheServerKeepsTheClientConnected
```

I hope I have had more time to implement more individual tests and performance improvements.

To run the tests follow the instructions below.

> ***cd tests***
> ***python3.8 -m unittest -v Server_Test.Integration***

Results