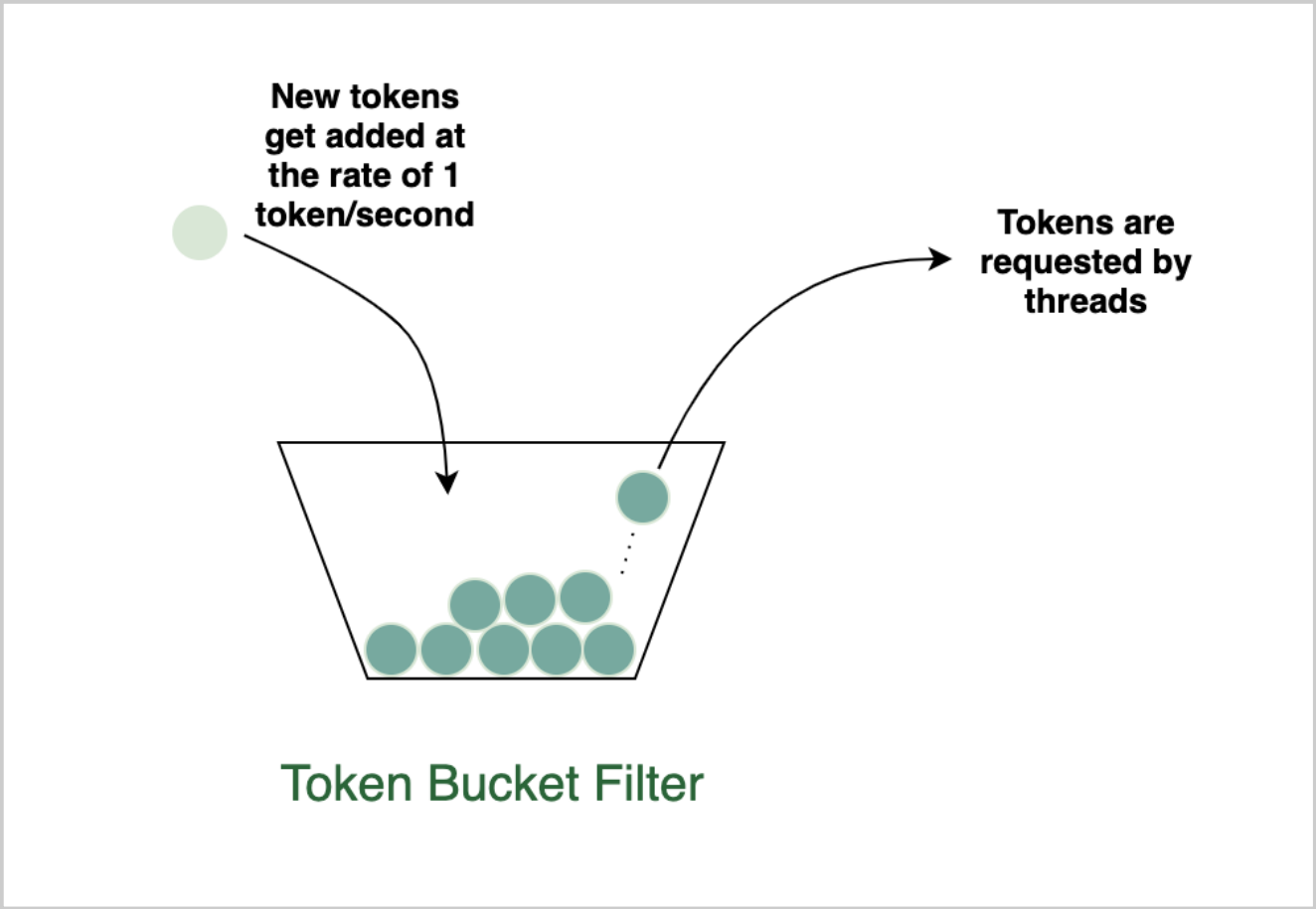
Rate Limiting Using Token Bucket Filter

Implementing rate limiting using a naive token bucket filter algorithm.

Rate Limiting Using Token Bucket Filter

> *This is an actual interview question asked at Uber and Oracle.*
>
> Imagine you have a bucket that gets filled with tokens at the rate of 1 token per second. The bucket can hold a maximum of N tokens. Implement a thread-safe class that lets threads get a token when one is available. If no token is available, then the token-requesting threads should block.
>
> The class should expose an API called **`get_token()`** that various threads can call to get a token.



Tocken Bucket Filter Problem

Solution

This problem is a naive form of a class of algorithms called the "token bucket" algorithms. A complimentary set of algorithms is called "leaky bucket" algorithms. One application of these algorithms is shaping network traffic flows. This particular problem is interesting because the majority of candidates incorrectly start with a multithreaded approach when taking a stab at the problem. One is tempted to create a background thread to fill the bucket with tokens at regular intervals but there is a far simpler solution devoid of threads and a lesson in making judicious use of threads. This question tests a candidate's comprehension prowess as well as concurrency knowledge.

comprehension prowess as well as scalability knowledge.

The key to the problem is to find a way to track the number of available tokens when a consumer requests for a token. Note the rate at which the tokens are being generated is constant. So if we know when the token bucket was instantiated and when a consumer called **`get_token()`** we can take the difference of the two instants and know the number of possible tokens we would have collected so far. However, we'll need to tweak our solution to account for the max number of tokens the bucket can hold. Let's start with the skeleton of our class:

```
class TokenBucketFilter:

    def __init__(self, MAX_TOKENS):
        self.MAX_TOKENS = MAX_TOKENS
        self.last_request_time = time.time()
        self.possible_tokens = 0
        self.lock = Lock()

    def get_token(self):
        pass
```

Note how **`get_token()`** doesn't return any token type! The fact that a thread can return from the **`get_token()`** call would imply that the thread *has the token*, which is nothing more than a permission to undertake some action.

We can envelope the logic of the **`get_token()`** with the lock we define in the **`__init__()`** method. Since we'll be manipulating the token count we must ensure that only a single thread is inside the **`get_token()`** method at any given time. Given the GIL restriction in Python, we may erroneously reason that we don't need to use a lock since only one thread can be active within the **`get_token()`** method. However, not using a lock can wreak havoc as the thread can be context switched at any time allowing mutation of shared state in a thread unsafe manner.

Moving on, we need to think about the following three cases to roll out our algorithm. Let's assume that the maximum allowed tokens our bucket can hold is 5.

- The last request for token was more than 5 seconds ago: In this scenario, each elapsed second would have generated one token which may total more than five tokens since the last request was more than 5 seconds ago. We simply need to set the maximum tokens available to 5 since that is the most the bucket will hold and return one token out of those 5.
- The last request for token was within a window of 5 seconds: In this scenario, we need to calculate the new tokens generated since the last request and add them to the unused tokens we already have. We then return 1 token from the count.
- The last request was within a 5-second window and all the tokens are used up: In this scenario, there's no option but to sleep for a whole second to guarantee that a token would become available and then let the thread return. While we `sleep()`, the lock would still be held by the token-requesting thread and any new threads (processes) invoking `get_token()` would get blocked.

The above logic is translated into code below:

```
class TokenBucketFilter:

    def __init__(self, MAX_TOKENS):
        self.MAX_TOKENS = MAX_TOKENS
        self.last_request_time = time.time()
        self.possible_tokens = 0
        self.lock = Lock()

    def get_token(self):

        with self.lock:
            self.possible_tokens += int((time.time() - self.la
st_request_time))

            if self.possible_tokens > self.MAX_TOKENS:
                self.possible_tokens = self.MAX_TOKENS

            if self.possible_tokens == 0:
                time.sleep(1)
            else:
                self.possible_tokens -= 1

            self.last_request_time = time.time()

            print("Granting {0} token at {1} ".format(current_
thread().getName(), int(time.time())));
```

You can see the final solution comes out to be very trivial without the requirement for creating a bucket-filling thread of sorts that runs perpetually and increments a counter every second to reflect the addition of a token to the bucket. Many candidates initially get off-track by taking this approach. Though you might be able to solve this problem using the mentioned approach, the code would unnecessarily be complex and unwieldy.

If you execute the code below, you'll see we create a token bucket with

max tokens set to 1 and have ten threads request for a token. The threads are shown being granted tokens at exactly 1-second intervals instead of all at once. The program output displays the timestamps at which each thread gets the token and we can verify the timestamps are 1 second apart.


```
            if self.possible_tokens > self.MAX_TOKENS:
                self.possible_tokens = self.MAX_TOKENS

            if self.possible_tokens == 0:
                time.sleep(1)
            else:
                self.possible_tokens -= 1

            self.last_request_time = time.time()

            print("Granting {0} token at {1} ".format(current_thread().getNa


if __name__ == "__main__":

    token_bucket_filter = TokenBucketFilter(1)

    threads = list()
    for _ in range(0, 10):
        threads.append(Thread(target=token_bucket_filter.get_token))

    for thread in threads:
        thread.start()

    for thread in threads:
        thread.join()

```

Below is a more involved test where we let the token bucket filter object receive no token requests for the first 10 seconds.

```python
from threading import Thread
from threading import current_thread
from threading import Semaphore
from threading import current_thread
from threading import Lock
from threading import Barrier
import random
import time


class TokenBucketFilter:

    def __init__(self, MAX_TOKENS):
        self.MAX_TOKENS = MAX_TOKENS
        self.last_request_time = time.time()
        self.possible_tokens = 0
        self.lock = Lock()

    def get_token(self):

        with self.lock:
            self.possible_tokens += int((time.time() - self.last_request_tim

            if self.possible_tokens > self.MAX_TOKENS:
                self.possible_tokens = self.MAX_TOKENS

            if self.possible_tokens == 0:
                time.sleep(1)
```

Back

Non-Blocking Queue

Next

... continued

Mark as Completed

Report an Issue

Ask a Question (https://discuss.educative.io/tag/rate-limiting-using-token-bucket-filter__interview-practise-problems__python-concurrency-for-senior-engineering-interviews)