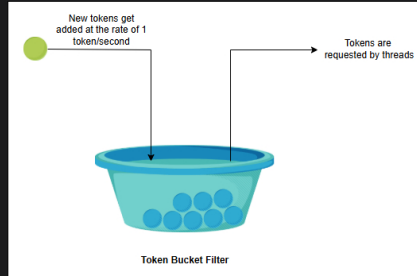


Rate Limiting Using Token Bucket Filter

Implementing rate limiting using a naive token bucket filter algorithm.

Problem Statement

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 `getToken` that various threads can call to get a token.



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 message to make judicious use of threads. This question tests a candidate's comprehension prowess as well as concurrency 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 `getToken()` 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

```
public class TokenBucketFilter {  
  
    private int MAX_TOKENS;  
    // variable to note down the latest token request.  
    private long lastRequestTime = System.currentTimeMillis();  
    long possibleTokens = 0;  
  
    public TokenBucketFilter(int maxTokens) {  
        MAX_TOKENS = maxTokens;  
    }  
  
    synchronized void getToken() throws InterruptedException {  
  
    }  
}
```

Note how `getToken()` doesn't return any token type ! The fact a thread can return from the `getToken` call would imply that the thread has the token, which is nothing more than a permission to undertake some action. Note we are using `synchronized` on our `getToken` method, this means that only a single thread can try to get a token, which makes sense since we'll be computing the available tokens in a critical section. We need to think about the following three cases to roll out our algorithm. Let's assume 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 monitor would still be held by the token-requesting thread and any new threads invoking `getToken` would get blocked, waiting for the monitor to become available.

The above logic is translated into code below

```
public class TokenBucketFilter {

    private int MAX_TOKENS;
    private long lastRequestTime = System.currentTimeMillis();
    long possibleTokens = 0;

    public TokenBucketFilter(int maxTokens) {
        MAX_TOKENS = maxTokens;
    }

    synchronized void getToken() throws InterruptedException {

        // Divide by a 1000 to get granularity at the second level.
        possibleTokens += (System.currentTimeMillis() - lastRequestTime) / 1000;

        if (possibleTokens > MAX_TOKENS) {
            possibleTokens = MAX_TOKENS;
        }

        if (possibleTokens == 0) {
            Thread.sleep(1000);
        } else {
            possibleTokens--;
        }

        lastRequestTime = System.currentTimeMillis();

        System.out.println(
            "Granting " + Thread.currentThread().getName() + " token at " + (System.currentTimeMillis() - lastRequestTime) / 1000
        );
    }
}
```

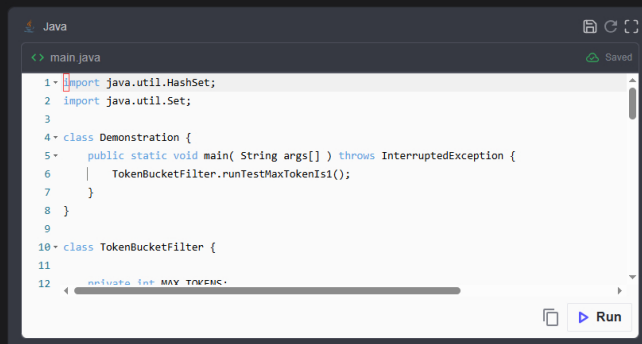
You can see the final solution comes out to be very trivial without the requirement for creating a bucket-filling thread of sorts, that runs perpetually and increments a counter every second to reflect the addition of a token to the bucket. Many candidates initially get off-track by taking this approach.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

Note we achieve thread-safety by simply adding synchronized to the `getToken` method. We can have finer grained synchronization inside the method, but that wouldn't help since the entire code snippet within the method is critical and would be guarded by a lock.

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.

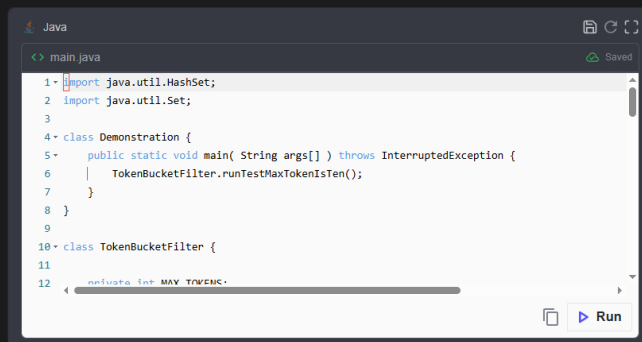
Complete Code

Below is the complete code for the problem:



```
1 import java.util.HashSet;
2 import java.util.Set;
3
4 class Demonstration {
5     public static void main( String args[] ) throws InterruptedException {
6         TokenBucketFilter.runTestMaxTokenIs1();
7     }
8 }
9
10 class TokenBucketFilter {
11
12     private int MAX_TOKENS;
```

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



```
1 import java.util.HashSet;
2 import java.util.Set;
3
4 class Demonstration {
5     public static void main( String args[] ) throws InterruptedException {
6         TokenBucketFilter.runTestMaxTokenIsTen();
7     }
8 }
9
10 class TokenBucketFilter {
11
12     private int MAX_TOKENS;
```

The output will show that the first five threads are granted tokens immediately at the same

The output will show that the inactive threads are granted tokens immediately at the same second granularity instant. After that, the subsequent threads are slowly given tokens at an interval of 1 second since one token gets generated every second. The astute reader would have noticed a problem or a deficiency in our solution. We wait an entire 1 second before we let a thread return with a token. Is that correct? Say we were 20 milliseconds away from getting the next token, but we ended up waiting a full 1000 milliseconds before declaring we have a token available. We can eliminate this inefficiency by maintaining more state however for an interview problem the given solution is sufficient.

Follow-up Questions

For the brave, we recommend implementing the following challenges.

1. Grant tokens to threads in a FIFO order
2. Generalize the solution for any rate of token generation

[← Back lesson](#)

[✔ Mark As Completed](#) [Next →](#)

... continued

... continued