Rate Limiting Using Token Bucket Filter

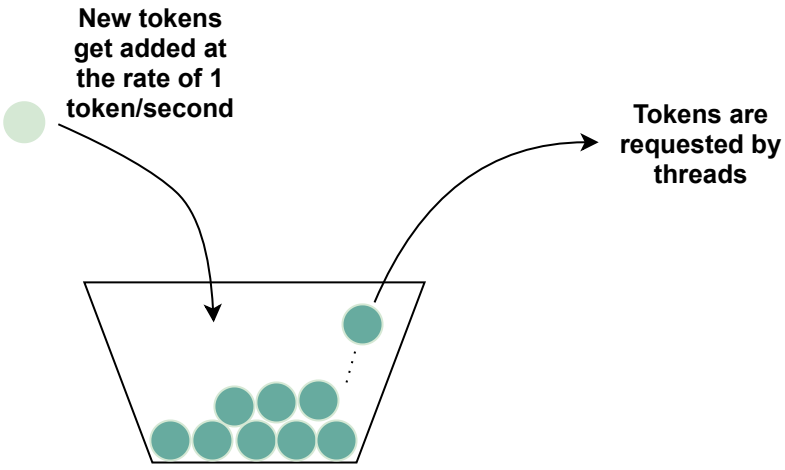
In this lesson, we will be implementing rate limiting using a naive token bucket filter algorithm.

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



Token Bucket Filter

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.

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:

```
class TokenBucketFilter

  def initialize(maxTokens)
    @maxTokens = maxTokens
    @lastRequestTime = DateTime.now.strftime('%s').to_f
    @possibleTokens = 0.0
    @mutex = Mutex.new
  end

  def getToken
  end
```

The class `TokenBucketFilter` contains a constructor method `initialize()` and the `getToken()` method. The bucket can hold N number of tokens at a given time represented by the `maxTokens` variable. There are two more variables, `lastRequestTime` and the current number of tokens represented by the variable `possibleTokens` in the bucket.

```
class TokenBucketFilter

  def initialize(maxTokens)
```

```
  def initialize(maxTokens)
    @max_tokens = maxTokens
    @lastRequestTime = DateTime.now.strftime('%Q').to_f
    @possibleTokens = 0.0
    @mutex = Mutex.new
  end

  def getToken

  end

end
```

Note how `getToken()` doesn't return any token type! The fact that 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.

We can envelope the logic of the `getToken()` with the mutex we define in the `initialize()` method. Since we'll be manipulating the token count, we must ensure that only a single thread is inside the `getToken()` method at any given time. Given the GIL restriction in Ruby, we may erroneously reason that we don't need to use a lock since only one thread can be active within the `getToken()` 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 are 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 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 mutex would still be held by the token-requesting thread and any new threads (processes) invoking `getToken()` would get blocked.

The above logic is translated into code below:

```
class TokenBucketFilter

  # initializing the bucket with maximum number of tokens
  def initialize(maxTokens)
    @maxTokens = maxTokens
    @lastRequestTime = DateTime.now.strftime('%s').to_f
    @possibleTokens = 0.0
    @mutex = Mutex.new
  end

  # the method where token is granted and the bucket is updated accor
dingly
  def getToken
    @mutex.synchronize do
      @possibleTokens += ((DateTime.now.strftime('%s').to_f) - @lastR
equestTime)
      if (@possibleTokens > @maxTokens)
        @possibleTokens = @maxTokens
      end
      if (@possibleTokens == 0)
        sleep 1
      else
        @possibleTokens = @possibleTokens - 1
      end
      @lastRequestTime = DateTime.now.strftime('%s').to_f
      puts "Granting " + (Thread.current[:name]).to_s + " token at "
 + (DateTime.now.strftime('%s')).to_s
    end
  end
```

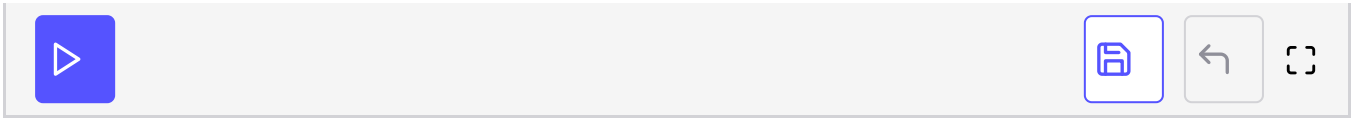
You can see the final solution comes out to be very trivial without the requirement for creating a bucket-filling thread of sorts that runs

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.

```
require 'date'
# A token bucket filter class with getToken and initialize methods
class TokenBucketFilter

        # initializing the bucket with maximum number of tokens
    def initialize(maxTokens)
        @maxTokens = maxTokens
        @lastRequestTime = DateTime.now.strftime('%s').to_f
        @possibleTokens = 0.0
        @mutex = Mutex.new
    end

        # the method where token is granted and the bucket is updated accordingly
    def getToken
        @mutex.synchronize do
                @possibleTokens += ((DateTime.now.strftime('%s').to_f) - @lastRequestTime)
                        if (@possibleTokens > @maxTokens)
                    @possibleTokens = @maxTokens
                end
                if (@possibleTokens == 0)
                          sleep 1
                else
                    @possibleTokens = @possibleTokens - 1
                end
                @lastRequestTime = DateTime.now.strftime('%s').to_f
                puts "Granting " + (Thread.current[:name]).to_s + " token at " + (DateTime.no
            end
    end
end

# creating a bucket with max capacity of 1 token/s
tokenBucketFilter = TokenBucketFilter.new(1)

threads = []
for i in 1..10 do
        t = Thread.new do
                tokenBucketFilter.getToken
        end
        t[:name]= "Thread " + i.to_s
```

```
        threads << t
end
threads.each(&:join)
```



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

```
require 'date'
# A token bucket filter class with getToken and initialize methods
class TokenBucketFilter

        # initializing the bucket with maximum number of tokens
    def initialize(maxTokens)
        @maxTokens = maxTokens
        @lastRequestTime = DateTime.now.strftime('%s').to_f
        @possibleTokens = 0.0
        @mutex = Mutex.new
        puts "Iniited"
    end

        # the method where token is granted and the bucket is updated accordingly
    def getToken
        @mutex.synchronize do
                @possibleTokens += ((DateTime.now.strftime('%s').to_f) - @lastRequestTime)
                        if (@possibleTokens > @maxTokens)
                    @possibleTokens = @maxTokens
                end
                if (@possibleTokens == 0)
                          sleep 1
                else
                    @possibleTokens = @possibleTokens - 1
                end
                @lastRequestTime = DateTime.now.strftime('%s').to_f
                puts "Granting " + (Thread.current[:name]).to_s + " token at " + (DateTime.no
            end
    end
end

# creating a bucket with max capacity of 1 token/s
tokenBucketFilter = TokenBucketFilter.new(5)

sleep(10)

threads = []
for i in 1..12 do
        t = Thread.new do
                tokenBucketFilter.getToken
        end
        t[:name]= "Thread_" + i.to_s
        threads << t
end
```

```
end
threads.each(&:join)
```