Payment Statistics Implementation

- **Syam Malla**

Summary

Create a system to calculate real-time statistics. Implement APIs to register transaction amount within the last 60 seconds and get the collective statistics on the transactions within last 60 seconds. Implement the GET api in O(1) time and space.

I will be using code snippets for reference. You can find the source code at below URL:

https://github.com/shyamkrishna8/payment-statistics

Solution

Overview

As the GET api should be O(1), there cannot be any operations we can do at the time of fetching the statistics. Hence we need to have an object precomputed which will always reflects the statistics within last 60 seconds. So when a GET api is invoked, we will simply check if there are no updates happening on the object and return the same as response. STATISTICS_DATA in com.syam.paymentstatistics.pojo.StatisticsData is the object which always reflects the statistics in last 60 seconds accurately except when updates are happening.

For every transaction POST api, we register the amount and update the STATISTICS_DATA real time. The real problem is that these transactions data needs to removed from the STATISTICS_DATA at the time of expiry which is 60 seconds after the timestamp provided in the request. So for a N number of requests that come in variable timestamps, we need to have a system which updates the STATISTICS_DATA exactly at the time of expiry of the Transaction. If it is done early or late, it could potentially make the GET consider data older than 60 seconds or vica-versa.

Events driven model vs Orchestrator

The problem of executing multiple requests at exactly various time stamps can be achieved by both Event driven model and Orchestrator. Event driven model is where you have one task for each request which is scheduled based on the execution time stamp. So each request acts on their own and it's their duty to get executed at the execution time stamp.

The other approach would be an Orchestrator, where one thread orchestrates the entire execution. Like polling from a channel continuously to identify if there are any requests that need to be executed. The only downside is we will be polling at every specific interval which probably makes the execution delayed.

Recursive scheduling model

In this solution, we will explore the idea of Recursive scheduling model. As stated in the earlier, orchestrator model comes with a downside of having a delay maximum of the polling interval. At the same time the Event driven model has the downside of having a new task for every request which simply isn't scalable although that is the right approach compared to the orchestrator. So for a 5000 requests at the same time, we will probably have 5000 scheduled tasks created which would stay alive for 60 seconds. Although the approach is correct, this is simply not a scalable solution

So we try to solve the problem of Event driven model. The problem is having all the future tasks scheduled at the time of receiving the request. So explore the idea of having a recursive scheduling tasks. If there is a task **T1** scheduled at 5 seconds from current time and there is a task **T2** scheduled at 10 seconds from current time. Instead of scheduled two tasks we can only schedule **T1** and inform that there is one task that need to scheduled after 10 seconds. So post execution of **T1,** it will schedule a task **T2** 5 seconds(10s - 5s) after the current time. If we generalize this, in a way we can have only one task to be executed in future and that task can schedule the further tasks.

So i have created a TreeMap to track the remaining requests to be executed with the execution time as the key. Although treemap is not thread safe, we are maintaining the thread safety by having a lock over any access to that object. TreeMap is used so that can get next immediately task to be executed directly. For every request, we add the details to REMAINING_REQUESTS and schedule a task if:

1. If no task is already present.
2. If the task scheduled is after the time of expiry of the first request in REMAINING_REQUESTS.

So this will ensure the expiration happens real time.

Min-Max map

Update the min and max for the new requests is pretty straightforward. We just check if the new amount is < or > for the min-max values respectively. The problem is when you are removing a min or max value. You need to identify the next min/max value to update the STATISTICS_DATA object directly. So we maintain another TreeMap to store the transaction entries sorted by the amount value. For calculating the min, we fetch the first amount in the map and for max, we fetch the last amount in the map.

We do not actually store all the requests in the Min-Max map. We only store the values which can potentially be Min-Max. For every new request, we check if there is already a Min and Max value after the timestamp of the new request. If there are, then there is no need to store these values.

Locks

For STATISTICS_DATA object, no multiple threads can write simultaneously and no thread can read when there is a write happening. So we are using ReadWriteLock for STATISTICS_DATA object access.

For REMAINING_REQUESTS object, we have two places where REMAINING_REQUESTS is accessed -

1. While scheduling, we add the request to REMAINING_REQUESTS.
2. While execution, we remove from REMAINING_REQUESTS.

The method for scheduling the task is synchronized at the class level(static method). So no two methods can schedule it simultaneously. And at the time of execution, as we have only one task executing at any time, no multiple writes can happen from the execution threads.

The only problem would be when thread is trying to modify REMAINING_REQUESTS for scheduling and at the same time, another thread is also modifying at the time of execution. So it is important to have a mutually exclusive lock for both these operations. Hence i have introduced ReadWriteLock REMAINING_REQUESTS_LOCK. ReadWrite lock is overkill as a simple lock would be sufficient though.