Distributed Task Scheduler

Author: Justin Barnyak

Version 1.0

6/17/24

Introduction

This is a take-home assignment for DeepOrigin. It's a scale-ready durable task scheduler that handles one-off and reoccurring events with a cron syntax.

System Overview

Objectives:

- Handle one-time tasks. Specified for a certain time and that won't be continuously executed.
- Handle recurring tasks. Specified with a cron syntax and are executed each time the cron syntax is matched.
- Despite any component failing, data should be durable.
- Tasks must be executed within 10 seconds of being registered.
- Components must be considered highly available.
- The system should be cost-effective
- The system should scale up and down.

Components:

- Task scheduler
- Task runner
- Pool watcher
- Task execution queue
- Main task database storing the at-rest states of every task in the system.

Overview:

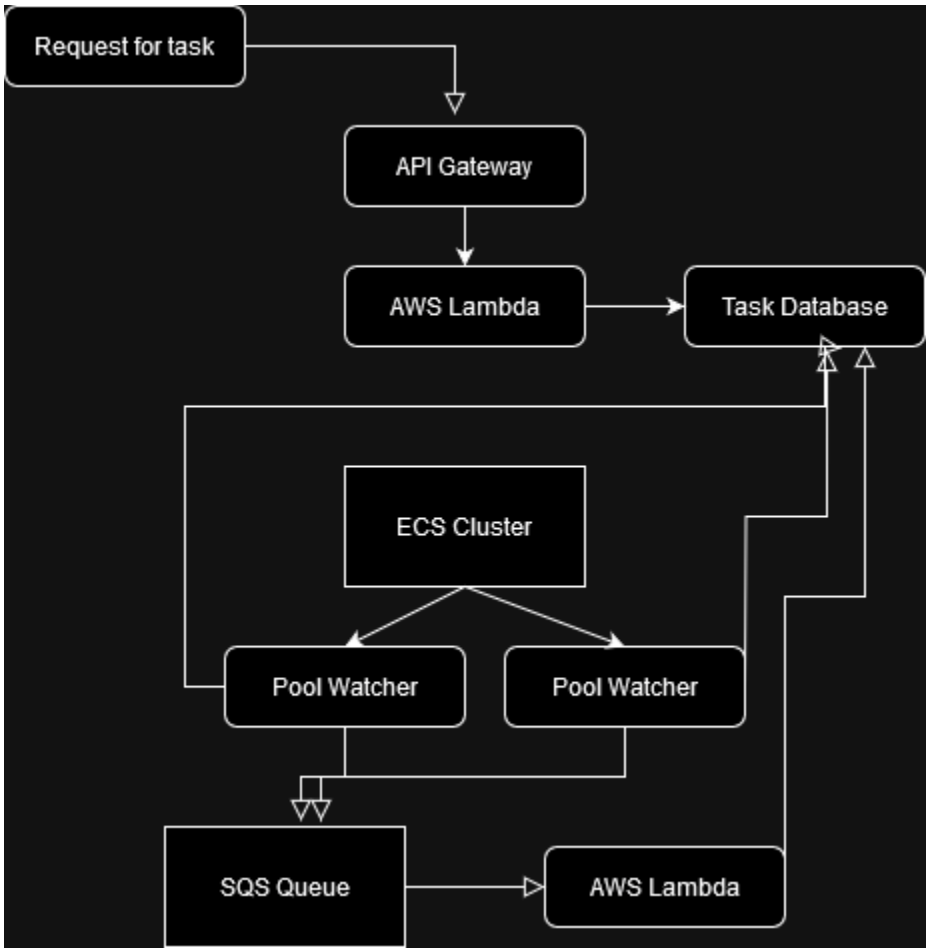
As an overview, the system can be described as receiving requests for new tasks via the task scheduler. Those tasks persisted with a new task ID in the main database. In parallel pool watchers are continuously checking the pending tasks with no executions attached to them. When a pool watcher finds this task, it creates a lock and then submits the task to a queue. That queue will instantly fire off a task runner in AWS lambda to handle the processing of the task and report back its results to the poll watcher. By configuring the pool watchers to be pooling the database with the particular keys that it has visibility for it should be able to execute its monitoring process every 5 seconds. Allowing 2 windows where tasks can be assigned and submitted for execution within the requirements of the overall system.

Architectural Design

Components:

- Task Scheduler
    - AWS Lambda Function behind an API Gateway
    - API Gateway requests are authenticated and then parsed with the parameters for the new task
    - The new task details are given a UUID key and stored persistently in an underlying Database
    - When created the schema of the database will ensure that poolers can tell nothing has looked at this task yet.
- Task Runner
    - AWS Lambda Function behind an SQS Queue
    - SQS queue contents are just the UUID of an event that needs to be executed.
    - On receiving the UUID it looks at the persistent database to get details of what needs to be executed for the task.
    - When execution is done it performs a clearing of the locks associated with it and updates the event to be marked as completed.
    - Updates an SNS queue that can allow downstream resources to act in the event of an event being completed (For example a front-end notification service).
- Pool Watcher
    - Nodes running in ECS.
    - Communication to the ECS cluster for ingress and egress is handled by VPC.
    - When a node fires up it generates a new UUID that it uses for locks.
    - The Locking Mechanism:
        - A pre-configured range of events can be calculated that can be grokked in the 5-second processing window of the pooler
        - Every 5 minutes that range of events can be reserved through MySQL and a combination of the pool's UUID that it uses for locking
        - Every 5 seconds it pulls down the configurations for the events within its scope
        - It then goes through and checks for matches of execution. If the execution occurs a separate per-event lock is created to indicate it's running and prevent the watcher from duplicating work
        - That even details are submitted to the SQS queue for processing.

- A log of how recently an action has been completed will be created in MySQL so that, for instance, for a trigger configured down to the minute that's processed in the first 5-second window available it won't re-process if its execution is completed within that minute.
- This historical logic allows us to find stale executions that should have been triggered by now but haven't. This could be the case in a scale-down scenario where a stale lock took time to be released from a pool watcher that's no longer active but has been picked up by a new pool watcher.
- If the watcher has "stale" locks that have never received the callback that they've been completed, they can be re-run after a pre-configured amount of time.



Performance Considerations

Trade-Offs

The pool watcher is both the bottleneck and primary performance consideration. Considering that AWS Lambda's attached to API Gateways and SQS queues will have managed out-of-box performance scaling with limits that can be increased into the millions of requests per second. Our ability to monitor how many tasks we have persisted to the database and analyze all their cron tasks potentially on a per-second basis is a tremendous undertaking when considering the upper limits of event requests that can be submitted.

In this case, the tradeoff of submitting every event to an SQS pool and using their pooling mechanism points to the fact that we'd be wasting resources continuously resubmitting a cron job that runs once a month, for instance. This would eventually bump up against the standing in-flight message limits of the queues themselves. We could take a hybrid approach and only submit jobs to the SQS pool that "should" be processable that day. However, this still leaves us with limitations in the daily events that the system can handle. Beyond the increased complexity of having 2 classes of poolers running, we'd still need a main type of pooler to handle adding and removing items from that queue if they become stale and roll over into another day's worth of requests. We are adding yet another moving window to consider when trying to calculate the total number of events the system can process.

Another potential tradeoff is lowering the performance impact on the database. We could offload the details of the event to a key-based storage solution like Redis. Or use another AWS service like RDS to spin up read replicas of our database so that the lambdas can access critical information for running the event data without impacting the nodes that are handling updating and creating the locks for the pooler's attempting to scan events as fast as possible.

Analysis

- Task Scheduler
    - Its performance is the least impactful on the entire system. Individual tasks should be almost instantaneous only needing to handle public authentication and storing the initial task in the database for the pool watcher to pick up later.
- Task Runner
    - By having this process asynchronously to the rest of the system executions can take as long as they need up to the 15-minute Lambda execution window.

    - The size and memory of the Lambda configuration can be tweaked to ensure a snappy execution of each task.
    - We're trading off potentially faster execution times and longer maximum duration of tasks for the cost-saving benefits of AWS Lambda here. But alternatively, it could be moved to execute within ECS in the future.
- Pool Watcher
    - Potentially the largest bottleneck of the system
    - Needs to be able to process a set number of records every 5 seconds to keep up with its rolling windows and ensure every record can have its cron jobs assessed.
    - The ability to scan the keys will be critical to scaling up and down in a reliable manner.
    - It's also important to note an offloading mechanism when a node is shutting down to cleanly release keys. While not critical to durability it will ensure the system has the least amount of downtime when stale keys roll over to a new pool watcher.

To calculate the max throughput of the system we need to calculate how many keys each pooler can reliably get a lock on and continuously read every 5 seconds if we assume this number to be 10,000. Then we can say that with 10 nodes running in the ECS cluster, we can begin the execution within 10 seconds of up to 100,000 registered events in any given second. Assuming they all match and are submitted to the SQS queue in that time. There is some overhead in how long it takes an SQS queue to begin executing a lambda function. This is also a moving window as it depends on the architecture of the lambda function itself. However, no matter how it's architected a warming function provided by another lambda function can serve to keep them "ready" at a given threshold that's calculated off the system's traffic.

The max connections on the read replicas will set a limit to the maximum number of AWS Lambda executions that can occur in parallel despite how much they scale underneath. For this, an RDS connection proxy layer provided by AWS can be utilized to mitigate this headache when scaling up.

Scaling

When scaling up we need to make sure that each additional pool watcher coming online isn't looking at the same execution set when processing it for each cron run. To do this designing queries to find the next block of executions with no lock on them should allow nodes to progressively get farther and farther into the execution set and eventually cover every execution entry so that they're all analyzed in parallel in each 5-second window.

When scaling up and down we can assume that there are not enough watchers to cover all the outstanding tasks in the database or too many and some have nothing to do. When scaling down since we'll be doing it at pseudo-random some tasks may become stale and will need to wait for their locks to expire until they can be looked at again. In this period, it's possible, if the scaling down is not "clean" and doesn't allow for the application code to release the locks, that these tasks will miss their deadlines for the duration of when an overseer application can determine the locks are stale and make them available for other watchers to process. However, this is still durable as those tasks will eventually be processed, just not on time.

Durability

There are multiple points of data being lost. If the pooler closes unexpectedly. If the event fails to be processed by the worker. If the data is not returned to the SQS queue in a set amount of time. All of these are handled by re-checking for data that has a lock, but never received a "well processed" flag at the end of the Lambda worker's routine. By checking for these regularly and re-processing requests we can ensure at least one execution of each task once its schedule requirements have been met.

The Lambda workers will be releasing running locks for items entered the SQS queue once the processing of those items is complete. While they have this lock present, they won't be re-entered into the SQS queue and shouldn't be reprocessed.

We also have the durability of the MySQL cluster itself. This can be handled by the relevant data teams managing the MySQL server to ensure backups are regularly created.

Cost Considerations

- Task Scheduler
    - Alternative is using ALB's and EC2 instances. Or to use ECS here as well. In both scenarios, we'll be paying for the underlying resources. Whereas the Lambda approach can scale to zero and incur significantly less usage when user registration usage is down.
- Task Runner
    - An alternative to AWS Lamdba functions for the workers is having another ECS cluster with nodes that pool the SQS queue and do the work here as well. It should be cheaper to use managed infrastructure with AWS Lambda as this will incur less operational costs, especially since the overhead of the ECS pricing (which includes all the underlying resources that it uses) is built into the execution time model of

Lambda. However, if Lambda ends up being a bottleneck this could be an opportunity to squeeze more performance out of the design.

- Pool Watcher
    - While costs can be incurred using ECS instead of manually spinning up watchers in EC2 the automation of setting the desired number of nodes based on the total unclaimed tasks in the database needs to stay highly available at all times.

Alternatives to RDS could be investigated such as PlanetScale to save on database costs. As the performance of the pool watchers could be impacted by lagging MySQL servers this could increase costs of ownership.

SQS can be compared to hosting our queue infrastructure as its per-request billing. At the same time, hosting something like Apache Kafka will incur the cost of the underlying resources to scale and host it on its own. We should also consider the operations cost of running our queue infrastructure as well. However, this may become necessary if the running limits of SQS become a bottleneck in terms of scaling to millions of requests.