Join

Back To Course Home

Grokking Modern System Design Interview for Engineers & Managers

0% completed

Search Course

Key-value Store

□

Content Delivery Network (CDN)

□

Sequencer

□

Distributed Monitoring

□

Monitor Server-side Errors

□

Monitor Client-side Errors

□

Distributed Cache

□

Distributed Messaging Queue

□

Pub-sub

□

Rate Limiter

□

Blob Store

□

Distributed Search

□

Distributed Logging

□

Course Certificate

Mark Course as Completed

Design Considerations of a Distributed Task Scheduler

Learn about the design considerations for the distributed task scheduler.

We'll cover the following □

Queueing#

A distributed queue is a major building block used by a scheduler. The simplest scheduling approach is to push the task into the queue on a **first-come, first-served basis**. If there are 10,000 nodes (resources) in a cluster (cloud), the task scheduler quickly extracts tasks from the queue and schedules them on the nodes. But, if all the resources are currently busy, then tasks will need to wait in the queue, and small tasks might need to wait longer.

This scheduling mechanism can affect the reliability of the system, availability of the system, and priority of tasks. There could be cases where we want urgent execution of a task—for example, a task that notifies a user that their account was accessed from an unrecognized device. So, we can't rely only on the first-come, first-serve to schedule tasks. Instead, we categorize the tasks and set appropriate priorities. We have the following three categories for our tasks:

- Tasks that can’t be delayed.
- Tasks that can be delayed.
- Tasks that need to be executed periodically (for example, every 5 minutes, or every hour, or every day).

Multiple queues based on the task categories

Our system ensures that tasks in non-urgent queues are not starved. As soon as some task’s delay limit is about to be reached, it is moved to the urgent tasks queue so that it gets service. We’ll see how the task scheduler implements priorities later in this lesson.

Let’s explore some parameters that help the scheduler efficiently utilize resources and provide reliable service to the users.

Execution cap#

Some tasks take very long to execute and occupy the resource blocking other tasks. The **execution cap** is an important parameter to consider while scheduling tasks. If we completely allocate a resource to a single task and wait for that task’s completion, some tasks might not halt because of a bug in the task script that doesn’t let it finish its execution. We let the clients set the execution cap for their tasks. After that specified time, we should stop task execution, release the resource, and allocate it to the next task in the queue. If the task execution stops due to the execution cap limit, our system notifies the respective clients of these instances. The client needs to do appropriate remedial actions for such cases.

If clients don’t set the execution cap, the scheduler uses its default upper bound on the maximum allowed time to kill the tasks. Suppose a task actually takes longer—for example, if we are training a machine learning model. In that case, the scheduler might need to pause and resume a task many times to accommodate other tasks. It wouldn't be fair to the short task that has to wait for two days to use a resource for two seconds.

Cloud providers can't let a task execute for an unlimited time for a basic (free) account, because using their resources costs a certain fee to the providers. To handle such cases, clients are informed about maximum usage limits so that they can handle long task execution. For example, clients may design their task in such a way that they checkpoint after some time and load from that state to resume progress in case resources are taken from the client due to usage limit.

Point to Ponder

Question

What if a long task is 90% executed, but before it completes, the machine that was executing this task fails?

Show Answer

Prioritization#

There are tasks that need urgent execution. For example, in a social application like Facebook, the users can mark themselves safe during an emergency situation, such as an earthquake. The tasks that carry out this activity should be executed in a timely manner, otherwise this feature would be useless to Facebook users. Sending an email notification to the customers that their account was debited a certain amount of money is another example of tasks that require urgent execution.

To prioritize the tasks, the task scheduler maintains a **delay tolerance** parameter for each task and executes the task close to its delay tolerance. **Delay tolerance** is the maximum amount of time a task execution could be delayed. The task that has the shortest delay tolerance time is executed first. By using a delay tolerance parameter, we can postpone the tasks with longer delay tolerance values to make room for urgent tasks during peak times.

Point to Ponder

Question

How do we determine the value of delay tolerance?

Show Answer

Resource capacity optimization#

There could be a time when resources are close to the overload threshold (for example, above 80% utilization). This is called **peak time**. The same resource may be idle during off-peak times. So, we have to think about better utilization of the resources during off-peak times and how to keep resources available during peak times.

There are tasks that don't need urgent execution. For example, in a social application like Facebook, suggesting friends is not an urgent task. We can make a separate queue for tasks like this and execute them in off-peak times. If we consistently have more work to do than the available resources, we might have a capacity problem, and to solve that, we should commission more resources. A cloud provider needs to have a target **resources-to-demand** ratio. When demand starts increasing, the ratio will move towards 0. If the ratio starts changing over time, the provider might decide to commission more or fewer resources.

Task idempotency#

If the task executes successfully, but for some reason the machine fails to send an acknowledgement, the scheduler will schedule the task again. The task is executed again, and we end up with the wrong result, which means the task was non-idempotent. An example of non-idempotence is shown in the following illustration:

A has 20 dollars. It makes a request to send 10 dollars to B, which has 0 dollars. A is the sender, and B is the receiver

1 of 11

We don’t want the final result to change when executing the task again. This is critical in financial applications while transferring money. We require that tasks are idempotent. An idempotent task produces the same result, no matter how many times we execute it. The execution of an idempotent task is shown in the following illustration:

A makes a request to send 10 dollars to B with the idempotence key that is added by the application

1 of 10

Let’s make the task of uploading a video to the database an idempotent operation. We don’t want the video to be duplicated in the database in case the uploader didn’t receive the acknowledgment. Idempotency ensures that the video is not duplicated. This property is added in the implementation by the developers where they identify the video by something (for example, its name) and overwrite the old one. This way, no matter how many times someone uploads it, the final result is the same. Idempotency enables us to simply re-execute a failed task.

Point to Ponder

Question

How should we handle task execution that can never be completed because of an infinite loop in the payload of that task?

Show Answer

Schedule and execute untrusted tasks#

Before proceeding, let’s ask ourselves a question: What are the untrusted tasks, and how should we manage them?

If you’re unsure about the answer, click the “Show Hint” button below:

Programs have latent bugs and might have malicious intent. When using task schedulers, we should be careful that one task does not impact other tasks negatively. If we provide infrastructure as a service, security is an essential component. This is because it becomes easier for tenants to harm each other’s tasks by executing malicious code in the shared environment. Execution of malicious code can also damage our infrastructure. So, we need to keep the following considerations in mind:

- Use appropriate authentication and resource authorization.
- Consider code sandboxing using dockers or virtual machines.
- Use performance isolation between tasks by monitoring tasks’ resource utilization and capping (or terminating) badly behaving tasks.

Point to Ponder

Question

What happens when the same task fails multiple times?

Show Answer

Now, let’s evaluate the design of our distributed task scheduler in the next lesson.

Design of a Distributed Task Scheduler

Next

Evaluation of a Distributed Task Sche…

Mark as Completed

Report an Issue