Back To Module Home

Basic Building Blocks for Modern System Design

0% completed

Search Module

Introduction to Building Blocks

Domain Name System

Load balancers

Cache

Databases

Key-value Store

Content Delivery Network (CDN)

Sequencer

Distributed Monitoring

Distributed Cache

Distributed Messaging Queue

Pub-sub

Rate Limiter

Blob Store

Distributed Search

Distributed Task Scheduler

System Design: The Distributed Task Scheduler

Requirements of a Distributed Task Scheduler's Design

Design of a Distributed Task Scheduler

Design Considerations of a Distributed Task Scheduler

Evaluation of a Distributed Task Scheduler's Design

Design of a Distributed Task Scheduler

Explore and connect the design components of the distributed task scheduler.

We'll cover the following

Let’s identify the components used in this design:

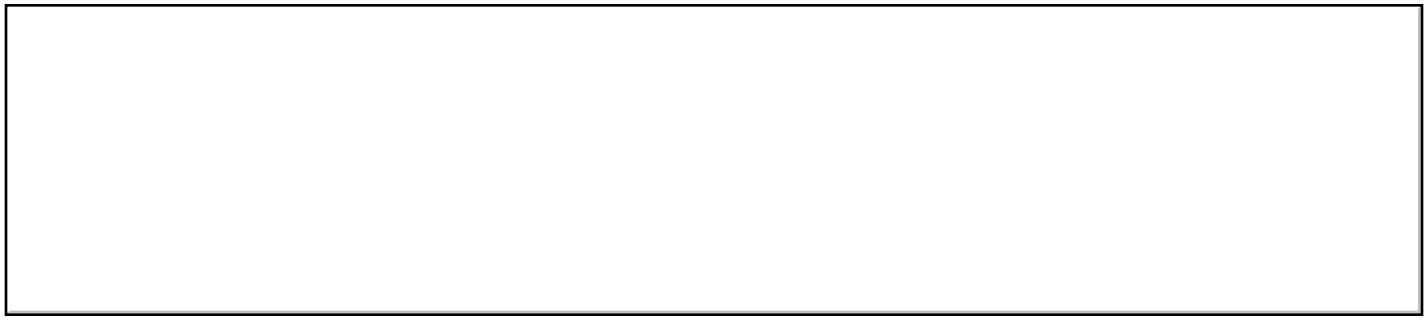
Components#

We can consider scheduling at many levels. We could be asked to design scheduling that is done internally by an organization to run tasks on their own cluster of machines. There, they have to find ample resources and need to decide which task to run first.

On the other hand, we could also be asked to design scheduling that a cloud provider uses to schedule tasks coming from multiple clients. Cloud providers need to decide which task to run first and which clients to handle first to provide appropriate isolation between different tenants.

So, in general, the big components of our system are:

- **Clients:** They initiate the task execution.
- **Resources:** The task is executed on these components.
- **Scheduler**: A scheduler performs processes between clients and resources and decides which task should get resources first.



Scheduler putting tasks into a queue for resource allocation

As shown in the above illustration, it is necessary to put the incoming tasks into a **queue**. It is because of the following reasons:

- We might not have sufficient resources available right now.
- There is task dependency, and some tasks need to wait for others.
- We need to decouple the clients from the task execution so that they can hand off work to our system. Our system then queues it for execution.

Let’s design a task scheduling system that should be able to schedule any task. Often, many tasks are relatively short-lived—from seconds to minutes. For long-running tasks, we might need the ability of periodic checksumming and restoration at the application level to recover from possible failures.

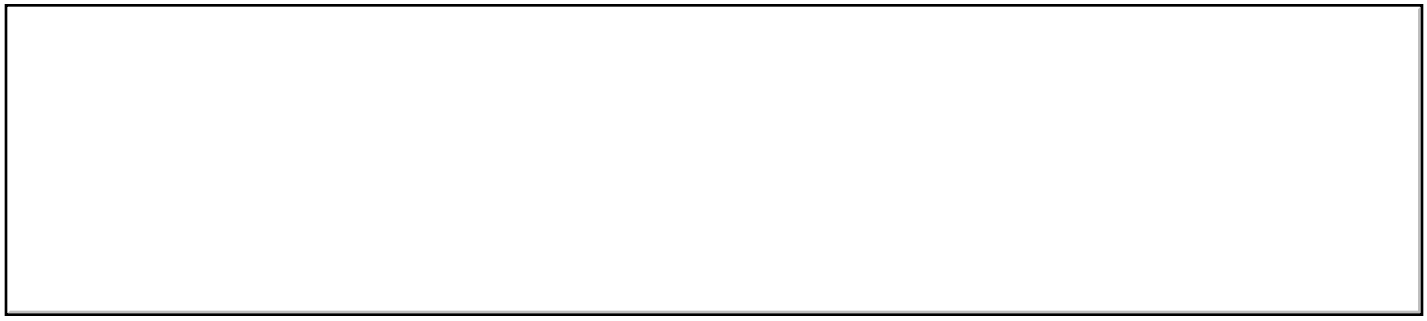
Let’s assume that some single server in our fleet can meet the computational needs of each task. For tasks that need many servers, either the application would need to break them down into smaller tasks for our system or employ long-term resource acquisition from the cluster manager.

Design#

When a task comes for scheduling, it should contain the following information with it:

- **Resource requirements:** The requirements include how many CPU cores it needs, how much RAM is required to execute this task, how much disk space is required, what should the disk access rate be (input/output rate per second, or IOPS), and how many TCP ports the task needs for the execution, and so on. But, it is difficult for the clients to quantify these requirements. To remedy this situation, we have different tiers of resources like basic, regular, and premium. The client can specify the requirement in terms of these tiers.
- **Dependency:** Broadly, tasks can be of two types: dependent and independent.
    - **Dependent tasks** require executing one or more additional tasks for their complete execution. These tasks must run in a sequence. For a dependent task, the client should provide a list of the tasks on which a given task is dependent.
    - **Independent task**s don't depend on the execution of any other task. Independent tasks can run in parallel. We should know whether a task is dependent or independent. The dependency information helps to execute both dependent tasks in order and independent tasks in parallel for efficient utilization of resources.

The design of the task scheduler is shown in the following illustration:



The design of task scheduler

- **Clients**: The clients of the cloud providers are individuals or organizations from small to large businesses who want to execute their tasks.
- **Rate limiter**: The resources available for a client depend on the cost they pay. It is important to limit the number of tasks for the reliability of our service. For instance,

X number of tasks per hour are allowed to enter the system. Others will get a message like “Limit exceeded” instead of accepting the task and responding late. A rate limiter limits the number of tasks the client schedules based on its subscription. If the limit is exceeded, it returns an error message to the client that the rate limit has been exceeded.

- **Task submitter**: The task submitter admits the task if it successfully passes through the rate limiter. There isn’t a single task submitter. Instead, we have a cluster of nodes that admit the increasing number of tasks.
- **Unique ID generator**: It assigns unique IDs to the newly admitted tasks.
- **Database**: All of the tasks taken by the task submitter are stored in a distributed database. For each task, we have some attributes, and all of the attributes except one are stored in the relational database.
    - **Relational database (RDB)**: A relational database stores task IDs, user IDs, required resources, execution caps, the total number of attempts made by the client, delay tolerance, and so on, as shown in the following table. We can find the details on the RDB here.
    - **Graph database (GDB)**: This is a non-relational database that uses the graph data structure to store data. We use it to build and store a directed acyclic graph (DAG) of dependent tasks, topologically sorted by the task submitter, so that we can schedule tasks according to that DAG. We can find more details of the graph DB here.

Database Schema

Column Name	Datatype	Description
TaskID	Integer	Uniquely identifies each task
UserID	Integer	This is the ID of the task owner
SchedulingType	VarChar	This can be either once, daily, weekly, monthly, or annually
TotalAttempts	Integer	This is the maximum number of retries in case a task execution

ResourceRequirements	VarChar	Clients have to specify the typ the offered resource categori such as Basic, Regular, or Premium. The specified resou category is saved in the form string in the RDB.
ExecutionCap	Time	This is the maximum time allo for the task execution. (This ti starts when a resource is alloc to the task.)
Status	VarChar	This can be waiting, in progre done, or failed.
DelayTolerance	Time	This indicates how much delay can sustain before starting a ta
ScriptPath	VarChar	The path of the script that nee be executed. The script is a f placed in a file system. The f should be made accessible so t can be executed, just like how mount Google Drive in the Goo Colaboratory and then execute code files there.

> **Note:** If we use geo-replicated data stores, we can run multiple instances of our task scheduling system in different data centers to achieve even larger scale and higher resource utilization.

- **Batching and prioritization**: After we store the tasks in the RDB, the tasks are grouped into batches. Prioritization is based on the attributes of the tasks, such as delay tolerance or the tasks with short execution cap, and so on. The top K priority tasks are pushed into the distributed queue, where K limits the number of elements we can push into the queue. The value of K depends t on many factors, such as currently available resources, the client or task priority, and subscription level.

Point to Ponder

Question

Why do we store tasks in a database? Why should we not push the tasks directly to the queue?

Show Answer

- **Distributed queue**: It consists of a queue and a queue manager. The queue manager adds, updates, or deletes tasks in the queue. It keeps track of the types of queues we use. It is also responsible for keeping the task in the queue until it executes successfully. In case a task execution fails, that task is made visible in the queue again. The queue manager knows which queue to run during the peak time and which queue to run during the off-peak time.
- **Queue manager**: The queue manager deletes a task from the queue if it executes successfully. It also makes the task visible if its previous execution failed. It retries for the allowed number of attempts for a task in case of a failed execution.
- **Resource manager**: The resource manager knows which of the resources are free. It pulls the tasks from the distributed queue and assigns them resources. The resource manager keeps track of the execution of each task and sends back their statuses to the queue manager. If a task goes beyond its promised or required resource use, that task will be terminated, and the status is sent back to the task submitter, which will notify the client about the termination of the task through an error message.
- **Monitoring service**: It is responsible for checking the health of the resource manager and the resources. If some resource fails, it alerts the administrators to repair the resource or add new resources if required. If resources are not being used, it alerts the administrators to remove them or power them off. Here's a detailed discussion on the design of monitoring services.

Task submitter#

As we have seen above, every component we use in the design of the distributed task scheduler is distributed and therefore scalable and available. But, the task submitter could be a single point of failure. So, to handle this, we use a cluster of nodes. Each node must admit the tasks, send the tasks to a unique ID generator for ID assignment, and then store the task along with the task ID in the distributed database.

There is a cluster manager to which each node sends a heartbeat that indicates the node is working correctly. Each node updates the cluster manager about the admitted tasks. The cluster manager maintains a list of tasks and the node ID that admitted that task. In case a node fails to execute a task, the cluster manager hands over that task to another node in the cluster. The cluster manager is itself replicated.

Above, we designed a task scheduling system. We'll discuss the design considerations of our task scheduler in the next lesson.

Requirements of a Distributed Task S…

Next

Design Considerations of a Distribute…

Mark as Completed

Report an Issue