sjkm ★ 318 Last Edit: March 8, 2021 8:51 AM 12.3K VIEWS

273

Designing a distributed Job Scheduler

I am writing this post because I was not able to find any resources for the system design question “Design a job scheduler” that were satisfying (at least for me). I thought that sharing my approach with you in oder to start a discussion would be great. I also think that it touches and uses a lot of concepts involved in a distributed system and can be a good starting point for people studying for their System Design Interview.

Please don’t forget to UPVOTE if you like the post :)

Functional requirements (can vary but I assume the following):

- A job can be scheduled for one time or multiple executions (cron job) by other services/microservices
- For each job a class can be specified which inherits some interface like Job so that we can later call that interface method on the worker nodes when we execute the job. (That class can e.g. be present in a .jar file on the worker nodes).
- Results of job executions are stored and can be queried

Non-function requirements (again, can vary but I assume the following):

- Scalability: Thousands or even millions of jobs can be scheduled and run per day
- Durability: Jobs must not get lost -> we need to persist jobs
- Reliability: Jobs must not be executed much later than expected or dropped -> we need a fault-tolerant system
- Availability: It should always be possible to schedule and execute jobs -> (dynamical) horizontal scaling
- Jobs must not be executed multiple times (or such occurences should be kept to a minimum)

Domain Analysis: Concepts
We can define the Domain Model that can later be converted into a Data Model (Database Model for Schema or a Model for ZooKeeper):

Job:

- Represents a Job to be executed
- Properties:
  Id, Name, JobExecutorClass, Priority, Running, LastStartTime, LastEndTime, LastExecutor, Data (Parameters)

Trigger (based on the concept Quartz Scheduler uses):

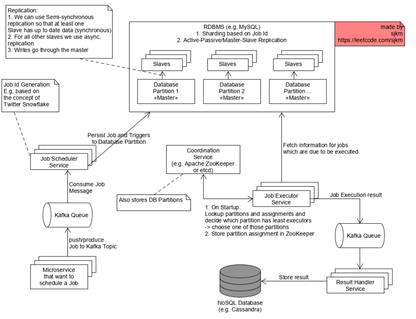
- Defines when a Job is executed
- We can define different Triggers like: OneTimeTrigger, CronTrigger
- Based on the type, we have properties like:
  Id, Type, StartTime, EndTime, OneTime, Cronjob, Interval

Executor:

- Is a single Job Executor/Worker Node
- Can have Properties like:
  Id (e.g. IP-based), LastHeartBeat

The multiplicity between Job and Trigger: Job-–1-------*-–Trigger (A Job can have multiple Triggers)

HLD (High Level Design)
Now, lets look at a high-level System Design diagram:



Microservices that want to schedule a non-/recuring Job: Can send a Message (or produce in Kafka terminology) to the corresponding Kafka Queue (precisely a Topic).

Job Scheduler Service: Will consume the Messages (requesting a Job enqueueing). They will generate a unique Id using e.g. the Snowflake ID Generation concept. Based on that ID (e.g. by hashing it) they decide into which Database Partition the Job will go. They create a Job and Trigger record according to the Message in the corresponding Database Partition.

RDBMS: I chose an RDBMS because we will later need ACID properties, meaning transactions. The database is sharded into an adequate number of shards to distribute the load and data. We use Active-Passive/Master-Slave Replication for each Partition in a semi-synchronous fashion. One Slave/Follower will follow synchronously while the others will receive the Replication Stream asynchronously. That way, we can be sure that at least one Slave holds up to date data in case the Master fails (due to network partitions, server outage, etc.) and that Slave will be promoted to be the new Leader.

Job Executor Service:
1. On Startup it will fetch the Database Partitioning info from ZooKeeper as well as the Partition Assignment between other instances and the Database Partitions.
2. It will choose a Database Partition which has the least number of Executors assigned to balance out the number of Executors that execute Jobs for a all the different Database Partitions.
3. It will send/store the Partition Assignment to ZooKeeper.
4. It constantly sends Heartbeats to ZooKeeper
5. It pulls information from the Database Partition and fights with other Executor instances assigned to the same Database Partition for Jobs that are due to execute. The fighting works by using Row Locks. That is why we need transactional properties (hence an RDBS that supports that, not all really do!)
6. Before an Executor Node executes a Job, it will update the Job record in the DB: Flagging it to be “Running”; store the “StartTime” and which Executor node (=itself) is executing the Job, etc.
7. When an Executor Node fails then other Nodes (assigned the same Partition) can detect it using ZooKeeper (due to the Heartbeating). They can then find all the Jobs that the failed Node was executing (Flag=Running and LastExecutor=Failed Node) and can fight for those Jobs to execute them (we could make a Job configurable to retry or not in case of execution failure).
8. Finally after successfully/unsuccessfully executing a Job, we send/publish/produce a Message to another Kafka Queue.

Regarding Point 7 (to expand the horizon on possibilities): We could also use Broadcast Messaging or a Gossip Protocol to detect Node failures. I’m excited to hear your argumentation.

Result Handler Service: Will consume Messages and store the Execution Result in a NoSQL database like Cassandra – Cassandra has a great write-throughput due to the fact that it is Leaderless (no time lost for fail overs for example), replicates asynchronously (can be configured) etc. We are also okay with Eventual Consistency because it is not crucial if we see a result with some delay.

Coordination Service, ZooKeeper: Stores the above mentioned information. Regarding the Database Partitioning information: We can e.g. load that info into ZooKeeper from another Service/Configuration file.

Message Queues, in general: We use Message Queues (here Kafka which is more of an (append) log than a Queue in the conventional sense, you can find great docs on the official website) in order to:

- Be able to scale the consumer and producer nodes independently (in Kafka we have Topics which can be horizontally partitioned and scale by that)
- We decouple the consumer and producer from each other
- Lower latency for the producer (doesn’t have to wait for a response)
- Durability and Reliability: When a Consumer Node crashes another Node can process the Message which otherwise would be lost (see Offset in Kafka). The Messages are persisted.
- We can throttle/limit the number of messages the consumers process (see Backpressure)
- Kafka offers message ordering

Other Issues and Concerns

- Unreliable Clocks/Time: In a distributed system we have unreliable clocks and time (due to unbounded delays when requesting NTP time because we are using packet-switched networks and usually not circuit-switched ones), unreliable NTP servers, Quartz Clocks that develop an offset, etc). When we want to schedule Jobs reliably and execute them at the right time, clocks and time play an important role. We therefore need to make sure that the times on the nodes are synchronized and don’t differ too much. One way to achieve that is to use multiple NTP servers and filter out those that deviate much. Another more reliable but costly way is to use Atomic Clocks in the Data Center(s).

Final Words:

- Of course this is a somewhat “simple” approach that would probably fit into a 45 minutes system design interview
- I’m aware that there are a lot of difficulties to actually build such a system, e.g.: Distributing Jobs correctly based on the load/work they have to do (and using the CPUs most efficiently), Exactly-once-delivery (/execution) and so on. Feel free to start a discussion :).
- I didn’t go deeply into things like how ZooKeeper or Apache Kafka works since that would blow the scope of this post. I assume that you know that those systems are already built in a distributed, fault-tolerant way.

Let me know what you think :)! I appreciate any input and feedback and would love to iterate on this approach with you!

system design | distributed systems | concepts

Comments: 22 Best | Most Votes | Newest to Oldest | Oldest to Newest

Qu33rsGambit ★ 29 February 26, 2021 4:20 AM

Dropbox has a blog about the design of their task scheduler, it follows a similar design as this one. It includes a decent discussion about idempotent tasks and configurable retries as well. https://dropbox.tech/infrastructure/asynchronous-task-scheduling-at-dropbox

21 Show 2 replies Reply

bakaiti002 ★ 69 February 28, 2021 3:21 AM

how are we querying db to figure out which tasks are due? do we query every minute/second? how to ensure job must not be executed much later than expected or dropped?

12 Reply

vanya203 ★ 6 February 26, 2021 7:29 PM

Read More

4 Show 3 replies Reply

sheepcoder1 ★ 37 February 26, 2021 6:35 PM

Brilliant explanation. Not too short, not too long.
One note, I feel this is intermediate level article, and i would mark it as such to help set expectations with beginners.

3 Reply

Notimplemented ★ 74 February 26, 2021 10:30 AM

Handling node failures (Regarding Point 7) can be processed as follows: every worker can send heartbeats to database and update last_active_timestamp. Other worker nodes can query for running=true and last_active_timestamp <= now - 2*heartbeat_period.

3 Show 2 replies Reply

rahul23111988 ★ 205 Last Edit: March 15, 2021 1:41 PM

Good write up.Following doubts.

1. why do we need to store results in No sql, results should go back to RDMS so that if something failed can be retried?Why do we need special result handler service?RDMS itself should maintain all info of previous runs?
2. Why do we have multiple executors fighting over same partitions? If there are multiple executors for same partitions, cant we distribute load based on job id ,to each executor?
3. I think there can be separate component that keeps on polling the database to get the jobs that needs to be run now and based on job id, It can push to some queue, and from that queue, each executor can start reading, with one queue per executor,as executor should only considering execute tasks, it should not bother how to get tasks.What do you say?One jobid goes one queue and hence one executor and you can live without db locks?

2 Reply

mayank12559 ★ 602 February 26, 2021 11:45 PM

I have used Airflow for task scheduler which is started by Airbnb and now is licenced by Apache. Airflow provides the platform to schedule jobs/tasks and a user-interface also for better visibility. It uses DAG (Directly Acyclic Graphs) to manage workflow so that dependent tasks do not form a cycle. There are multiple blogs for this. https://airflow.apache.org/blog/

2 Show 1 reply Reply

xanidex ★ 1 March 1, 2021 9:10 PM

A very Big Thank to you for sharing such a great article among us.

1 Reply

nadrane ★ 5 February 27, 2021 3:11 PM

Great write up! This one part had me asking question:

“One Slave/Follower will follow synchronously while the others will receive the Replication Stream asynchronously. That way, we can be sure that at least one Slave holds up to date data in case the Master fails (due to network partitions, server outage, etc.) and that Slave will be promoted to be the new Leader.”

Doesn’t a synchronous replication configuration imply that your system is no longer highly available? If there is a partition between these master and slave machines, users will no longer be able to perform writes to the database until failover occurs. How would you expect your system to behave under these circumstances? Also, what’s your take on the potential latency increase implications of this configuration, if there’s say a more tiny network blip?

1 Show 2 replies Reply

hyamath ★ 2 February 26, 2021 6:47 AM

Great Article!

1 Show 1 reply Reply