System Design — Distributed Job Scheduler

source: quanta magazine

Design a system for distributed job scheduling.

Functional Requirements

- Users can create jobs to be run at a specified time, and at specific intervals if the job requires to be run periodically.
- Users can delete or update jobs.
- Users can specify priority of a job. Priority is used to resolve ties for jobs with same start time.
- Jobs are asynchronous i.e. users do not have to wait for the job to get completed and results are updated in the backend.

Non Functional Requirements

- **Durable**. No jobs and their results should get lost due to system failures.

- **Reliable**. All jobs created should be either taken to completion or update status with appropriate message and status codes and jobs should be started as much close to the specified start time as possible.
- **Available**. It should be always possible to create jobs and query results for jobs.
- **Scalability**. Users can add more jobs without breaking the reliability and availability of the system.

Traffic and Throughput Requirements

- 10 million jobs created per day
- Average size of data per job = 100KB
- Number of requests per second for job results = 10K
- P99 latency for querying results and logs = 5 seconds

The "On My Computer" Approach

Sorted Set+HashMap+Multiprocessing Queues

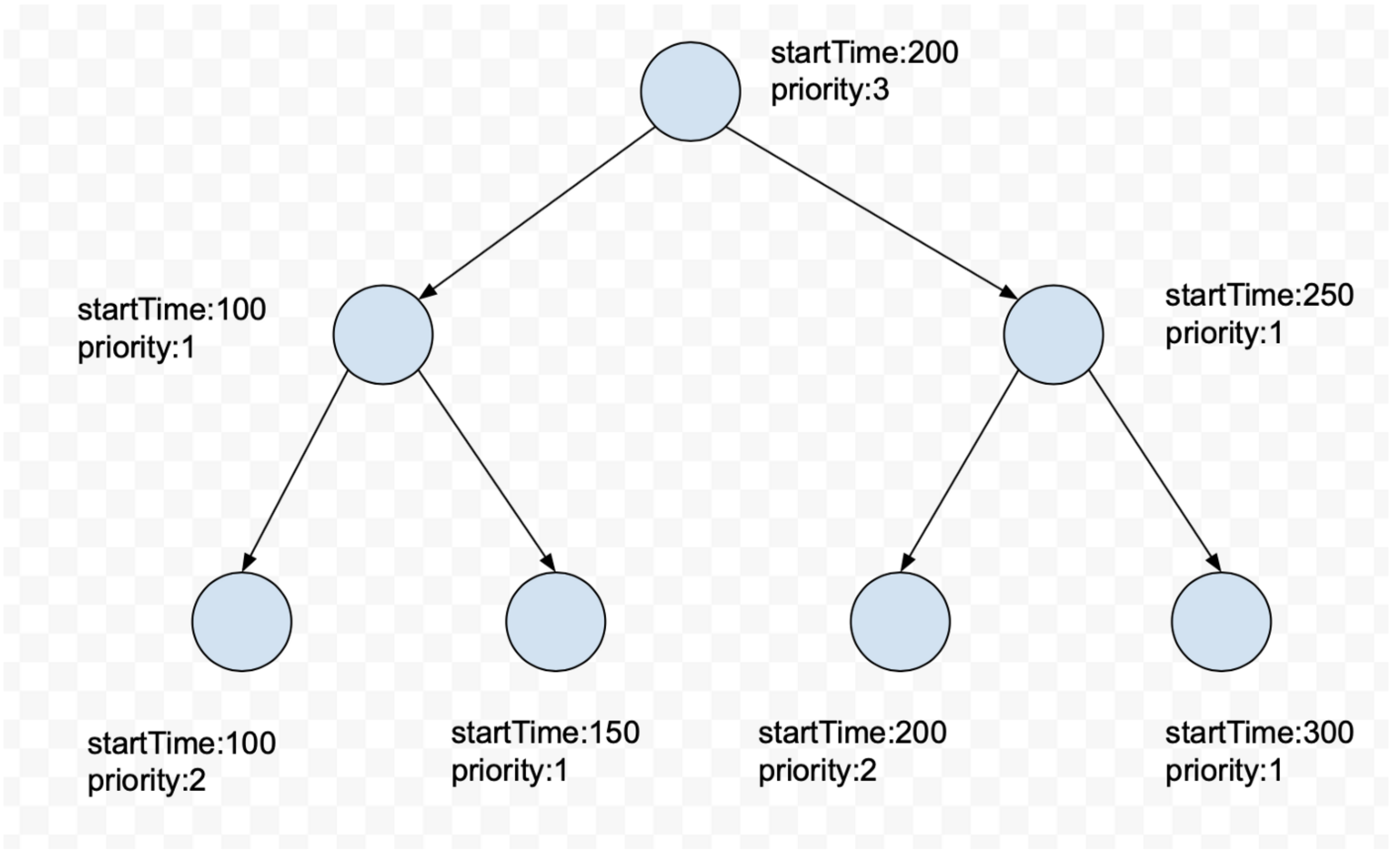
Each job is represented as a struct or a class with the following properties:

```
Id
JobName
JobExecutable
StartTime
Period
Priority
Data
```

'JobExecutable' is either a script or a zipped file with multiple classes or a docker image.

The jobs are then added to a **Sorted Set, implemented using either AVL Tree (balanced BST) or Skip List**. **The jobs are ordered first by the StartTime and for same StartTime, they are ordered by the Priority.**

For e.g. To add a new job in a balanced BST, if the start time of the new job is lower than the start time of root or if they are equal but priority of new job is higher than that of root *then go to left else go right and so on at each level.*



Return error if two jobs exist with the same StartTime and same Priority.

Maintain a HashMap from the 'Id' to the corresponding node in the balanced BST. This is required in order to access a node directly without searching the entire tree because nodes are not indexed on the job Id.

Time complexity to add, delete and update jobs is **O(logN) where N is the number of unprocessed jobs**.

To maintain the status of the jobs, we can use another Hashmap.

Whenever a job is created and added to BST we will update the status to 'CREATED' against the job Id in this Hashmap. Similarly when the job is being executed will update the status to 'PROCESSING' and whenever it has been completed, we update the status to 'COMPLETED'.

Create a Pool of parallel processes — P1, P2, … PN and a shared multiprocessing Queue Q.

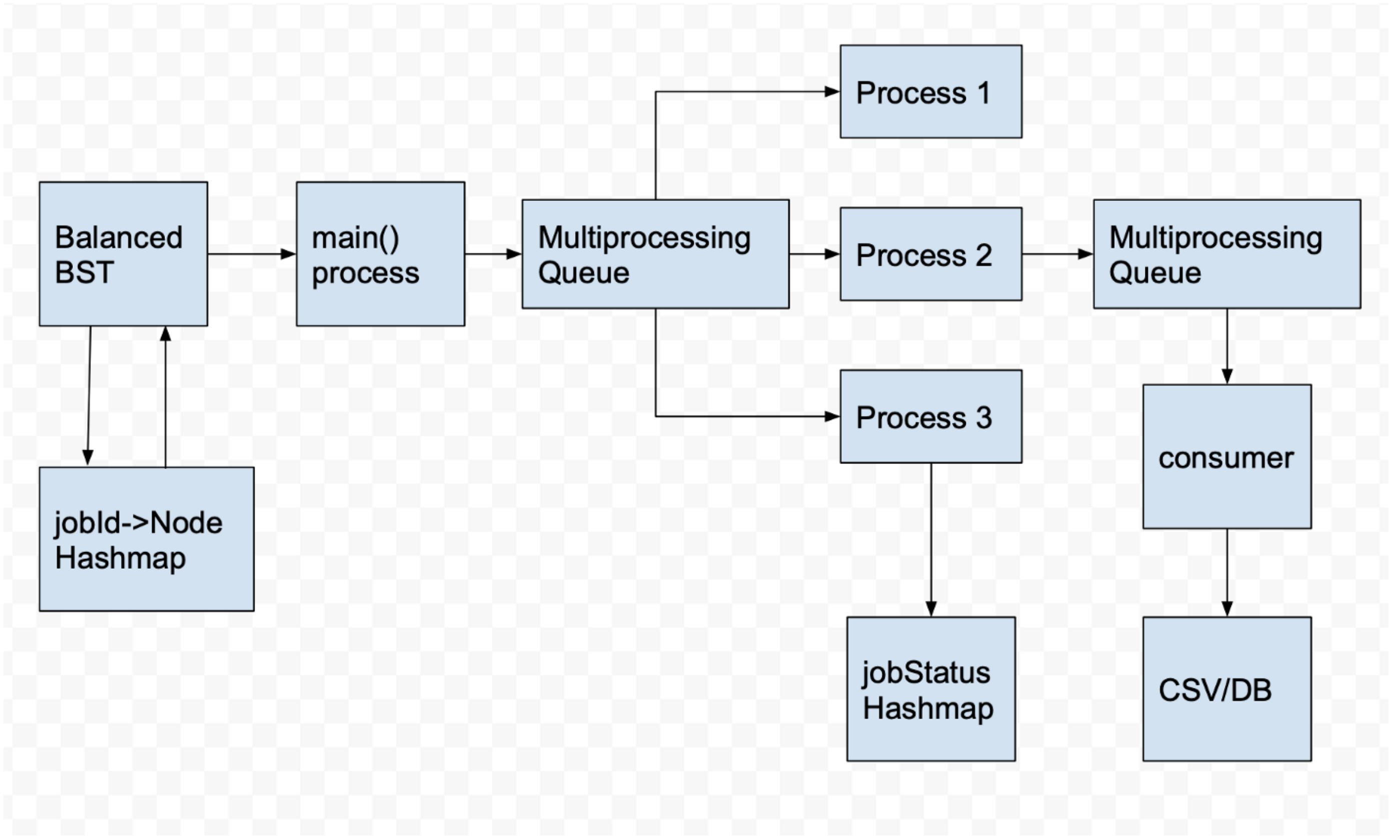
The number of processes N is dependent on the available RAM and CPU cores. Assuming that total RAM on machine is 16GB and each job has data size of 100KB, then approximately we can run 168K parallel processes. But there are only 8 CPU cores which means we can run only 8 processes in parallel.

> *In the main process, continuously (or after every 1 sec) check if the size of the balanced BST is non-zero. If it is non-zero, then extract the leftmost node of the tree and check if the **StartTime ≤ current timestamp**. If it is then, **add the job to the Q** and **delete the node**.*

Each parallel process Pi whenever free, continuously polls Q and executes the job by executing 'JobExecutable', **writes the result to another shared Queue** if successfully completed. Also if the Period for the job is non empty, then a **new job is re-ingested** into the balanced BST but the StartTime is updated to **StartTime+Period**

If the job fails for certain reasons, then also a result with an error message is written to the output shared Queue.

> *A process to poll the results from the output queue fetches the results from the queue and updates the results in a database or a CSV file.*



The problems with this approach are:

- We can a run a **maximum of 8 processes only**. Thus many jobs will be sitting idle in the queue even after well past their start times. This **violates reliability and scalability**.
- The balanced BST is an in-memory data structure. On system crash or restart, the data for the jobs are lost **violating durability**.
- Assuming that we are ingesting 10 million jobs each of size 100KB per day, total size of in-memory data for 30 days = 28TB which is way greater than 16GB (available RAM in most systems). Thus it **violates scalability**.

Using Distributed Queue and MySQL

Whenever a job is submitted, it is pushed to Kafka queue.

Kafka consumers will poll the latest messages from the queue and send the job metadata to a MySQL database and the actual data and executables to S3.

MySQL table schema:

(jobId, startTime, period, priority, status, result_location, max_retries, retry_num)

MySQL is used because it has ACID properties which allows transactions and row based locking.

This is required so that multiple worker processes reading the pending jobs and taking them up for execution, will not see inconsistent view for a job i.e. a process while updating status of a job from PENDING to PROCESSING, another process is trying to read it will read the status as PENDING, thus 2 processes will be taking up the same job.

Assuming jobs for 5 years, total number of jobs = 10M*365*5 = 19 Billion.

Number of bits to store an unique jobId = log2(19 Billion) = 35 bits

Assuming period is in minutes and it can be anything between 0 and 1 year (525600 minutes). Number of bits required = 19 bits

Assume that for a given start time there can maximum of 1000 jobs with the same start time. Then number of bits required for priority = 10 bits

Also we can use integers to represent statuses. Since there could 4 different statuses (PENDING, PROCESSING, COMPLETED, ERROR) we need 2 bits to represent them.

The size of job metadata can be calculated as follows:

jobId (35bits) + startTime (32 bits) + period (19 bits) + priority (10 bits) + status (2 bits) + result_location(10 bytes, S3 bucket name) + max_retries (2 bits) + retry_num (2 bits) = 182 bits.

Total size of data in MySQL for 5 years = 19billion*182bits = 400GB.

This much data is manageable on a single MySQL instance.

If the MySQL instance goes down, metadata for jobs will be lost. Thus in order to satisfy availability and reliability of the system, maintain replicas. There will be a master instance where all writes will happen and 2 read replicas from where all queries for job statuses will happen.

The choice of master-slave replication instead of master-master was pretty obvious as we want consistency on writes. Workers should see the same value before updating.

Controller

There will be a Controller service which will doing the following functions:

- One process to read from the MySQL table — **SELECT jobId FROM jobs WHERE (status='PENDING' OR status='ERROR') AND startTime ≤ NOW() AND retry_num < max_retries ORDER BY startTime, priority DESC**
- Based on the number of processes, CPU and RAM usage in the worker instances, it will **select a worker instance to execute a job** from the above query.

The controller service will maintain an in-memory map of instance_id to the (instance_host, instance_port, number of processes, CPU usage, available RAM, last heartbeat timestamp) tuple.

These data is updated along with heartbeat messages from the worker instances every 2 seconds.

If the last heartbeat timestamp is **more than 2 minutes**, then an worker instance is assumed to be dead and that worker is not assigned any job. Whenever a new worker joins, it again starts sending heartbeat messages along with the system data.

First filter all worker instances with CPU usage and RAM less than 90%, then based on the minimum number of running job processes, it selects a worker instance.

If there are no worker instance with CPU usage and RAM less than 90%, then it will spin up a new worker instance.

Once selected, the controller calls an API along with **data and executable location in S3** for the job, on instance_host and instance_port which starts a new process to execute the job.

Since the expected number of jobs per day is 10M, and assuming that each job on average takes around 5 minutes to get completed.

Number of jobs that can be run on a single worker instance = 8 cores * 24*60/5 = 2300 jobs per day. Thus number of worker instances required is = 10M/2300 = 4.5K instances to start with.

What if the controller service goes down ?

We will maintain 2 backup controller instances. Have a coordinator such as zookeeper to detect heartbeat messages from the controller instance and if no heartbeat is detected for 2 minutes then assume it is dead and make another backup instance as controller.

Now we also need to update the controller details in the worker instances.

To handle this problem, instead of directly sending the heartbeat messages from the workers to controller, let the coordinater handle them. In the coordinator we will have a table of instances to host and port. When a worker instance or controller is updated, the table in the coordinator is also updated.

Worker Instances

There will be several worker processes for executing the jobs. Whenever a new process is created for a job, the data and executables are downloaded from S3 then the executable is run on the given data.

If the job successfully completes, then the **logs and results are uploaded to S3** and **status and location of logs+results in S3 is pushed to another Kafka queue** for asynchronous workflow.

A **JobUpdate Kafka consumer** will pull the latest messages from Kafka and then update the MySQL database for the status of the job and result location. Also if the period for the job is not null, then a new entry is made in MySQL table with startTime+period and status='PENDING'.

If the job encounters an error, then status is updated to 'ERROR' and update the MySQL table for jobId by increasing the retry_num.

JobResults

To get the status and results of jobs, there would be a JobResults service running behind a **load-balancer**.

The service would query the MySQL table to get the status and results_location, then it would download the logs+results from S3.

Number of CPU cores required for JobResults service = 10K RPS*5 seconds latency = 5000 cores which is approximately 625 instances with 8 cores.

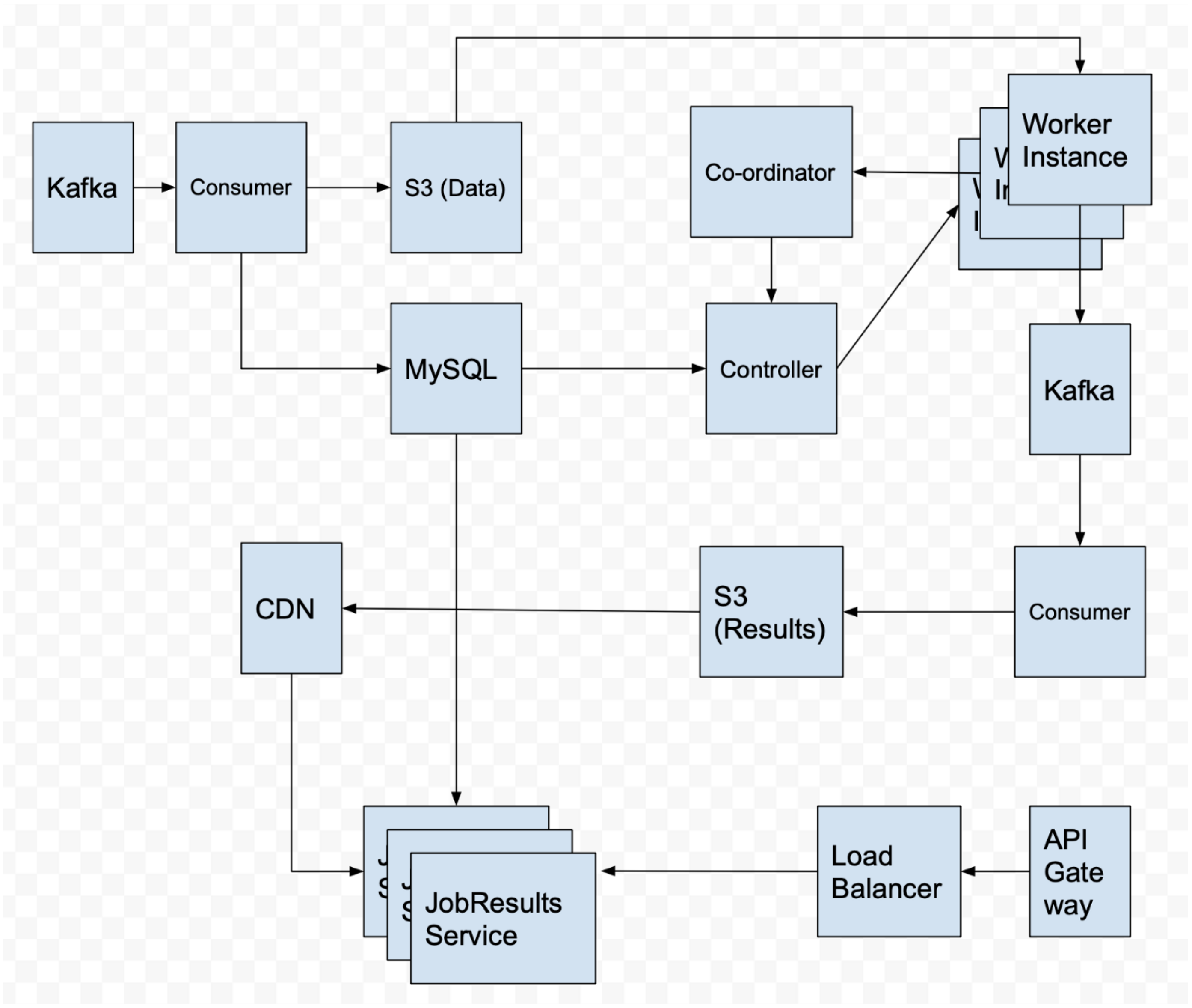
Assuming the size of results+logs is on average 10KB, the read bandwidth required is 10KB*10K RPS = 100MB/s.

To handle download failures use a **retry logic with exponential backoff strategy**.

To deal with loss of data due to ***datacenter crash*** *and to enable fast access to S3 from globally distributed regions, we replicate the same set of files across multiple datacenters across the world. Request for a file is sent to the nearest datacenter from the user.*

In order to reduce the load on S3 for downloading the results, we can use a **CDN** like caching service to cache the most recently used files.

The Pipeline



Useful Resources

- https://leetcode.com/discuss/general-discussion/1082786/System-Design%3A-Designing-a-distributed-Job-Scheduler-or-Many-interesting-concepts-to-learn
- https://dropbox.tech/infrastructure/asynchronous-task-scheduling-at-dropbox
- https://sre.google/sre-book/distributed-periodic-scheduling/
- https://bowenli86.github.io/2016/07/14/system%20design/task%20queue/System-Design-Design-Task-Scheduling-System-in-Distributed-System/

- https://medium.com/walmartglobaltech/an-approach-to-designing-distributed-fault-tolerant-horizontally-scalable-event-scheduler-278c9c380637
- https://stackoverflow.com/questions/26890312/how-to-design-a-distributed-job-scheduler