Developing a Scalable Scheduler service - Python + Microservices - Part 1

Problem statement:

A time based scheduler is always a need for any buisness driven application. It is also important to have microservice driven architecture for such mission critical component. One of core responsibility of scheduler as service is to trigger schedule at it's scheduled time. However, as your number schedules /concurrent/near concurrent schedule will increase it may become bottleneck. This blog is an attempt to solve scalability and performance issues by stating a example architecture.

Few bullet points before we begin:

Abstraction : Scheduler service should be a re-usable component/service however, imo, it shouldn't be exposed to UI directly an independent service. Other services can use/proxy it -- Service to service communication. Althoughproxying is not good for performance. But in microservices world it's a common choice.

Cluster per service: Every service should have right to spin its own scheduler instances/cluster -- may be there own implementation or just reuse common scheduler if requirements suffice. although, common/shared service is also good idea -- here the scalability needs to be considered. And service to service communication cost and complexity should be considered.For example: A service can spin it's own scheduler cluster if workload is going to be high. This scheduler will user thar service's db. Service can simply provide scheduler REST/Graphql APIs by interfacing a re-usable django app. As a standard practice -- every service/component should be very fine re-usable app. For other light workload service deploy a common cluster so other services can use. Important point to highlight -- how you code/package is totally different that how you going to deploy.R

Reusability with isolation: It's good to not to mix scheduler with job management. I think schedulers responsibility can be limited to CRUD schedule in persistent storage, tick the clock logic and when clock meets -- just put job definition it on destination (a queue or async call), that's it! Rest of things is the responsibility of individual service. E.g. Job management, send sms, email, call a phone number etc. This will keep Scheduler lightweight, loose coupled, isolated. It also provides flexibility to consumer service. for example -- if a consumer services wants to have more granular and customised task status execution status(es), it can freely implement it. If consumer needs to apply retry logic based on its own criterias it can do so. Here jobs db will not be overloaded soon and become bottleneck because jobs will be stored in Service db, scheduler service db also won't be overloaded. If re-usability is concern then we can develope a common job/worker management framework -- which shall be easily extensible as per needs of consumer.F

Flexibility: Have flexibility to support cross language, framework independent scheduling consumers.

Scalability and performance: Consider scalability by either load distribution of schedule(aka sharding or partitioning) or load balancing with replica -- with distributed locking (like redis) to avoid duplication. Prefer (in-memory?) stateless/brainless scheduler clocks nodes (slaves) which actually tick the time just send events to master. Use any simple library like Apscheduler/quartz. Seperate out schedule management node (master/brain) which manages the state and other things. Keep some partitioning strategy, for example -- per service based, per tenant based, recurrence based or best case plain even distribution (round robin) scheme (like a consistent hash ring). Even mix of it? Have a solid and flexible strategy. This can be configured per cluster.The service which consume scheduler can act as master (by deploying master node as part of it) in itself and choose best fitted strategy. In short it's all above sums up simply distribution of the load.

Reliability: Having master and state less slaves will also give flexibility to recreate and distribute redistribute schedule in case of crash.

Registration and health monitoring: Good to have slave registry mechanism and health monitoring. Assign slave a unique id which lives life long with it. Everytime slave is started it must first registered with master.

Auto-scaling: This shouldn't be initial aim but Based on number of records/load threshold/memory/cpu etc etc health monitoring service can generate alerts which later on could be consumed by infra scaling service.

Simplicity and standard interface: Most widely used interface is -- simple/advanced cron based schedules. I believe cron is sufficient for the most of scheduler needs-- offcourse we can make cron human readable.

Fault tolerance: I think simple way is supervision of clocks as restart them as they are dead. Whenever clock node resigster to master shall recreate /resend all the jobs. If we need replica we can have multiple replica (slave cluster) but one active other passive, this is bit more complex though.

I will detail the demo architecture with more details… stay tuned!