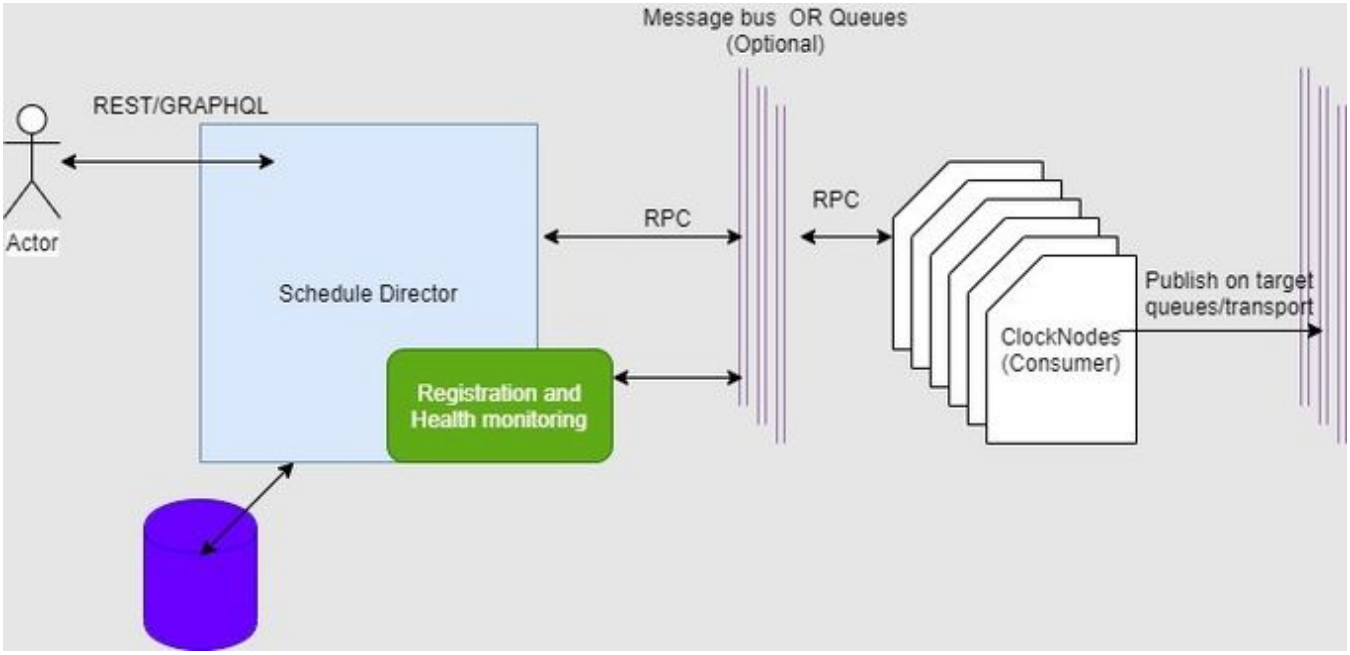
Developing a Scalable Scheduler service - Python + Microservices - Part 2

Cluster Overview:



- **Schedule director node:** The schedule director node is what we can call a “master” from Part -1 discussion. The primary responsibility of scheduler director is to provide CRUD on schedule, maintain the schedule state. Also monitor Clock Nodes. State of scheduler is maintained in persistent node such as PostgreSQL database. Scheduler director has following two more micro-component:

1. Registration service: This is a RPC server which provides APIs for clock nodes to get registered to given Director node. Registration process can be simply adding database entry to Perform complex state management calls.
2. Health Monitoring service: This service will check health of registered consumers periodically and generates alters.

- **ClockNode:** This should a light-weight node. It can or can not have it’s own state management or database specifically. This decision is totally up-to how your requirements are. If clock node choose to not to have state management then this node can simply store it's schedules in memory, Redis or any other in-memory store like Memcached. Primary responsibility of ClockNode is to check on all schedule it has received, when its time to trigger – put them on target channel (Defined in job definition or pre-configured on Director or ClockNode). It is also a responsibility of ClockNode to respond to health and control messages of Scheduler Director.

Partition Strategy examples:

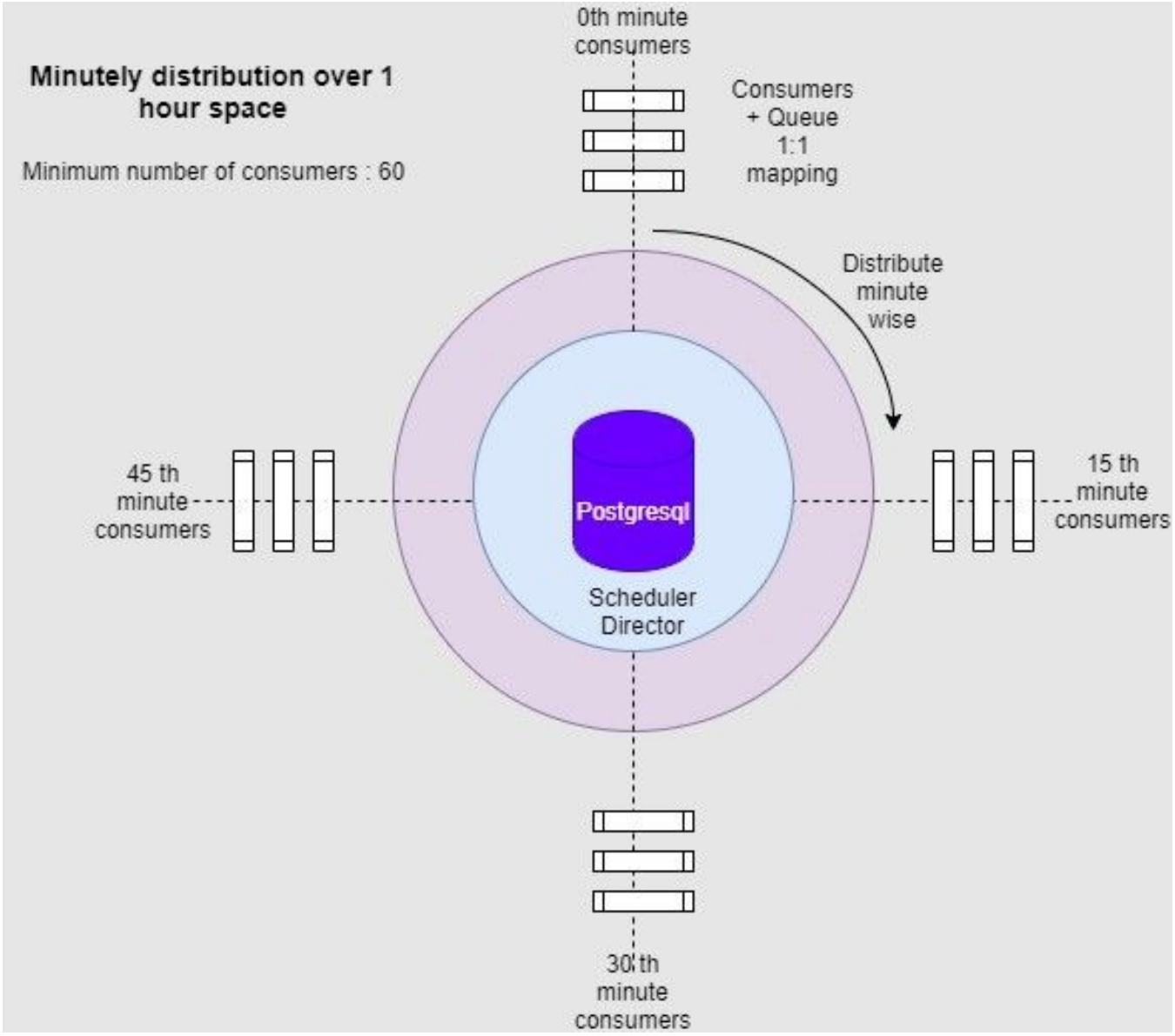
- One of the efficient way could be to distribute schedules based on it's schedule it-self. To achieve this we can have distribution of schedules as per there minutes offsets. This idea is inspired by a wall clock. We can say – This is subset of algorithm for consistent hashing.
- Minute Hand strategy with 1 Hour span or space: Basic idea is to have consumers (ClockNodes) re-presenting and responsible for scheduling schedules for particulate minute. For example in Cron mode:
- If a Schedule runs Every hour and on 15th minute will run 00:15, 01:15, 02:15 .. so on. This schedule can be routed to consumers attached to 15th minute queues.

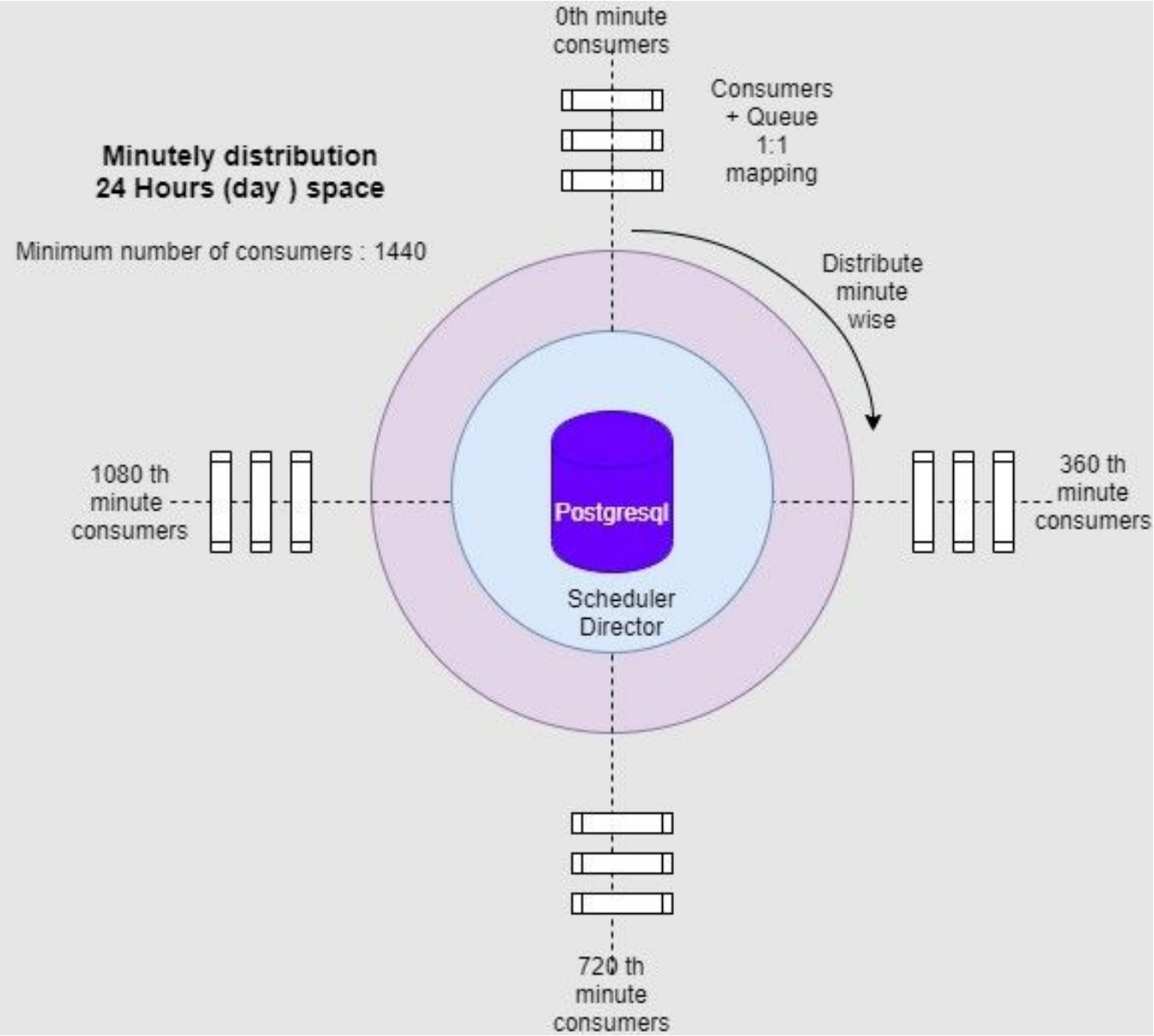
- Another strategy is do the same over 1 day or 24 hours span. Number of minute spots in this case are 1440.
- In generic way let's called these minute partitions as “Minute spots”.
- Time Space size will range from 1 to 1440. (1 minute to 1 day).
- Equation to calculate the minute spot: Minute spot = ((schedule hour * 60) + schedule minute) % time space.
- Consider following scenarios schedule hour minute and ts size.

	Schedule type	HH	MM	TS size in minutes	minute bucket
1	Every hour	-	0	60	0
2	Repeat on 30th minute of every hour	-	30	60	30
3	Repeat every day at	0	0	60	0
4	Repeat every day at	23	10	60	10
5	Repeat every day at	23	10	120	70
6	Repeat every day at	23	25	1440	1405

7	Repeat every week on Nth day	Same as daily			
8	Run once on	12	0	1440	0

So basic idea is to allocate minute bucket based on hour, minute and Space size.





What about every minute schedule? The every minute schedule can have dedicated point in space denoted by -1. OR alternate approach is it can be scattered over bucket 1 to 60.

What about every */X minute? for example */15 minute would run at 15, 30, 45 0. We can follow following strategy.

1. Scattered - We can pre-calculate all the interval and scatter its respective minute bucket. In above example Schedule will distributed to 15 30 45 0. Pros: Fine distribution within existing nodes. Cons: Performance issue, The expressions can range anywhere from */1 to */59 so minimum so such schedule can take additional RPC calls ranging from 1 - 59.
2. Dedicated Buckets: For such schedules we can add dedicated buckets ranging from -1 to -59. Pros: Easy to track, Fine distribution. Cons: Additional buckets. However, not every application requires that much granularity Many application can define there granularity and minimize buckets required. For example: Application can choose to provide granularity of 5 minutes only. In that case buckets required would be 60/5 => 12.

What about interval based schedule? The difference between interval and cron schedule -- The interval schedule states time gap between two consecutive jobs, here the next run time is usually calculated by looking at last run time, where as Cron based specifies exact trigger time on clock. For this series we would not consider interval Schedule because Cron can do what interval schedule can do but not a v.v.. Although easiest strategy to support Interval based schedule is to convert interval schedule in a cron job.

To summarize In this part we talked about overall architecture design, partitioning strategy etc. In next part we will go through scalability and performance consideration for our design.