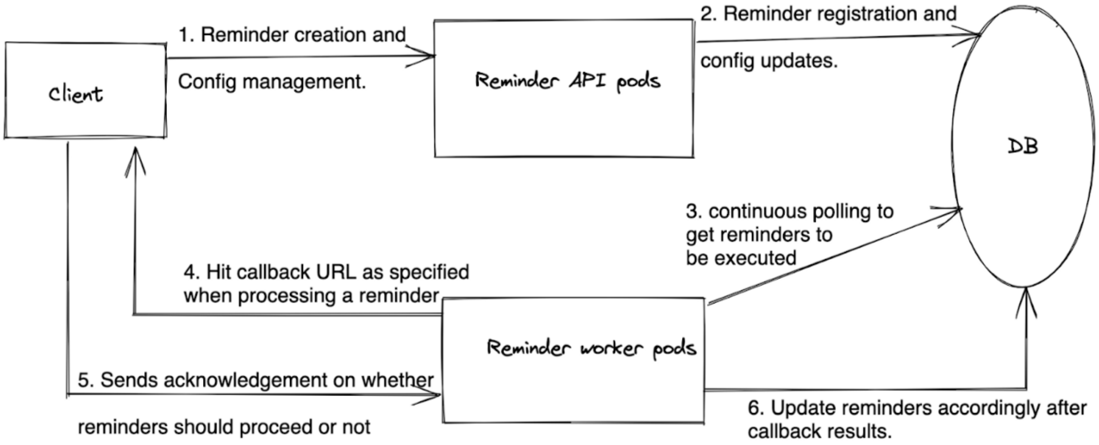
How we built in-house Circuit Breaker in Reminders

Problem Statement

The Reminders team which is part of platform engineering of Razorpay have a cron like service to notify clients on the configured time by hitting their configured callback URL. When the clients are overloaded or facing downtime, then they should be given some breathing space and time to recover by not sending in more requests which might lead to cascading failures. So, we needed a circuit breaker which would reduce the load on the clients and also to ensure a high response time and success metric.

Reminders Architecture :-


Client
1. Reminder creation and
Config management.
Reminder API pods
2. Reminder registration and
config updates.
DB
3. continuous polling to
get reminders to
be executed
4. Hit callback URL as specified
when processing a reminder
Reminder worker pods
5. Sends acknowledgement on whether
reminders should proceed or not
6. Update reminders accordingly after
callback results.

We have 2 components :

1. API Layer :-

Handle all the configurations related to users, reminders and create a reminder accordingly in the specified time.

2. Worker layer :-

This layer has a periodic cron running at fixed intervals. It fetches reminders scheduled to be executed at a fixed time, processes them by hitting the callbackURL and updates the database according to the response received.

After we hit the configured callbackURL, we continue with the reminder flow based on the http status returned -

- If the status is 200, we continue for next configured run.
- If the service returns 400, we stop the execution.
- If the service returns anything other than these 2, we retry 3 times with 1 minute time gap before stopping execution of that reminder item.

How circuit breaker fits in to our system :-

1. Each client onboards reminder service via a unique identifier for them called namespace(similar to client-ID).
2. When a client is facing degradation or service failure, our callback to them is blocked till we receive a response from their end.

3. When a lot of calls are blocked, the rate at which we process reminders goes down.
4. When a client’s service is down, our reminders to them also result in failure thus failing before solving its intended purpose.
5. Also, every failure causes a retry to happen which further increases the number of calls we make and also the error count.
6. Moreover, adding more traffic to a service undergoing downtime/issue may lead to a cascading failure leading to a failure of the entire system.
7. We can enable/disable reminders for a particular client by changing the namespace enabled flag to true/false.
8. We continuously monitor 2 important metrics for all of our namespaces — **latency and error count**.
9. We take a decision of whether to switch a circuit to on/off flow by comparing these metrics against a fixed threshold.
10. We settled to store these 2 metrics continuously for all namespaces in **Redis time series database** as we need to

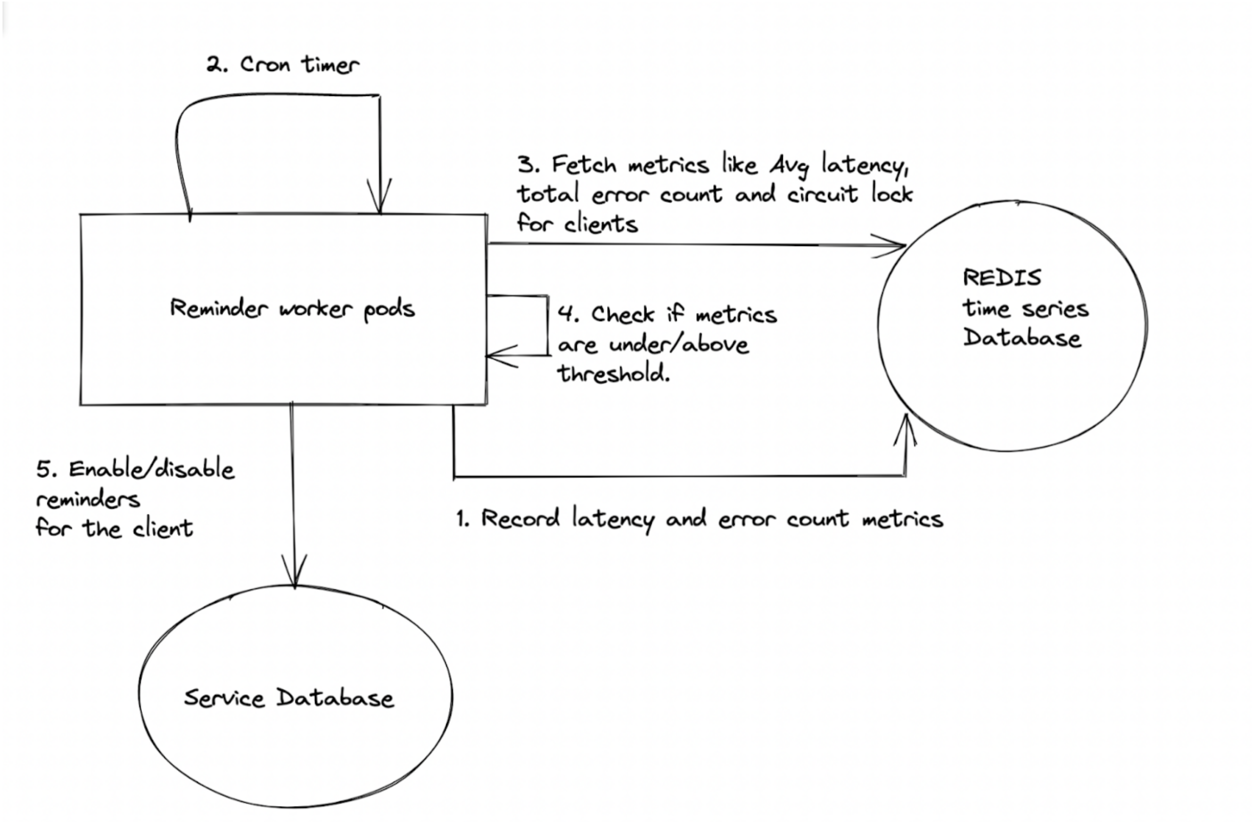
do computational operations over a given time interval in an easy and quick manner.

11. We can configure the thresholds for average latency, error count and duration for which circuit should be open with the help of APIs.
12. We have default values assigned for them initially for fallback.

Design(Inside the Circuit Breaker) :-

1. We will have an additional cron running for circuit breaker in our workers at fixed time intervals.
2. We check the average latency and total error counts in the fixed time window between current run and previous run.
3. We will compare the numbers achieved with the max threshold permitted for that client.
4. If the threshold has been breached and the circuit is in a closed state, we open the circuit for a fixed cool-off time before retrying again.

5. If the numbers are within threshold and the circuit is open, we close the circuit bringing everything to normalcy.
6. We should also wait till the configured circuit open time to cool off before doing any circuit related actions, we maintain this via a redis lock.


2. Cron timer
3. Fetch metrics like Avg latency, total error count and circuit lock for clients
Reminder worker pods
4. Check if metrics are under/above threshold.
REDIS time series Database
5. Enable/disable reminders for the client
1. Record latency and error count metrics
Service Database

We had 3 redis keys stored in our redis time series database.

1. **Average latency :-**

Used to record the average latency of a particular customer’s calls

Format :- service_name:{client_id}:latency:current_hour

Example :- reminders:{12xys12}:latency:13

Ttl :- 2 hrs from creation.

2. Total error count :-

Used to record total error count.

Format :- service_name:{client_id}:err_count:current_hour

Example :- reminders:{12xys12}:err_count:13

Ttl :- 2 hrs from creation.

3. Circuit open timer :-

Duration upto which a circuit should remain open before accepting traffic again.

Format :- service_name:{client_id}:lock

Example :- reminders:{12xys12}:lock

Ttl :- circuit open timer configured for that customer

Data storage in redis time series :-

Since we are a service which serves more than 1k TPS, we might receive multiple requests for even a given millisecond and hence there are high chances for data to have collision and get corrupted. So, we stored data according to the nanosecond timestamp at which the request came in. With redis time series we were easily able to obtain the accurate results within this time range once we scaled the start and end time as well to nanoseconds.

Sample queries :-

```
TS.CREATE reminders:{namespace}:err_count:20
```

```
TS.ADD reminders:{namespace}:err_count:20 1648046710276825010 40
```

```
TS.RANGE reminders:{terminals}:err_count:20 1648046710276825000 1648046830276825000 AGGREGATION “sum” 12000000000000
```

Why not use open source circuit breakers?

Initial thought process was to use Netflix’s Hystrix but we decided against it because of the following constraints :

1. Hystrix works on a defined fixed url. Here, since we have the callbackURL for a namespace changing according to the paymentID.

2. In case of Hystrix, there are call drops when the circuit breaker switches from open to half-open to close circuit. Here, we cannot afford to have any call drops.
3. Hystrix is not a centric mechanism, it has to be implemented on pod level. We needed a central mechanism to have data for all pods in sync. It would require a lot of effort to do this in hystrix.
4. We needed more control of the flow since this could potentially cause a downtime.

Logging and Monitoring :-

1. We have set up a critical alert when redis goes down as our entire circuit breaker depends on redis.
2. We have a metric dashboard citing the circuit open/close occurred across time periods.
3. Apart from this, we also log critical points in flow regarding circuit open/close and threshold breach.

Impact of Circuit Breaker in our system :-

1. After circuit breaker was released in production, we have never received any complaints so far regarding high traffic when our underlying services are going through degradation.
2. On an average today, circuit breaker acts and opens circuit 50 times a day approximately for 300 different namespaces saving us from around 250 minutes of high error rates and prolonged responses.
3. This in turn contributes to a more efficient processing and high throughput of reminders belonging to other unaffected namespaces.

Enhancements :-

1. We do not have a design similar to half-open mode in other open source circuit breakers.
2. We make a retry callback after a minute in case we receive a failure, we are currently able to live with this design. But, in systems without strong retry logic it might become problematic without half-open mode.

3. Instead of a cool off period circuit re-trigger, we can selectively send one http call per client to check if the service is up again before sending all our traffic again.
4. This can be easily achieved by having an event driven queuing mechanism. We can put a http call in queue with the callback url and finish consuming the message only when the service is up to restart the flow.

Conclusion :-

In this blog, we saw how we are able to build a simple in-house circuit breaker using redis time series database, how the data is stored and what are the few enhancements we could do in this design.

We at Razorpay platform engineering team keep developing a lot of cool features like this. We are actively hiring for our Engineering team and you can apply for us via our jobs page.