Thundering Herd Problem -Paypal

https://medium.com/paypal-tech/thundering-herd-jitter-63a57b38919d

The **thundering herd problem** occurs when a large number of processes or threads waiting for an event are awakened when that event occurs, but only one process is able to handle the event. When the processes wake up, they will each try to handle the event, but only one will win. All processes will compete for resources, possibly freezing the computer, until the herd is calmed down again.

Let us say each server can handle a certain number of requests(say 500) per second and the load balancer distributes the requests to handle the load. But what if the server 1 goes down while handling 500 requests (after completing say 200 requests). So the rest of the 300 requests would be sent to server 2 . Let's say server 2 comparatively has lesser computing power or even if it has the same computing power it is handling more than it's designated capacity (QPS). So the server 2 crashes after a while (after handling 200 requests). Now the first 200 + requests directed at server 2 goes to server 3. Now remember server 3 may additionally also have to handle new requests. So there we have the thundering herd problem . By now let's say if you are using JVM based languages there is a high probability of out of memory error and server 3 crashes now. So there you have the scenario of "**cascading failure"** .

In the thundering herd problem, a great many processes (jobs in our case) get queued in parallel, they hit a common service and trample it down. Then, our same jobs would retry on a *static* interval and trample it again. The cycle kept repeating until we exhausted our retries and eventually DLQ'd(Dead Letter Queued).

While we had autoscale policies in place for this, our timing could improve. We would hammer the processor service which would eventually crash it. Then our jobs would go back into the queue to retry N times. The processor service would scale out, but some of our retry intervals were so long, the processor service would inevitably scale back in before the jobs retried.

While we were using exponential backoff it doesn't exactly stop the thundering herd problem. What we need is to introduce randomness into the retry interval so the future jobs are staggered.
The gist of Jitter is that if you have 100 people run to a doorway, the doorway might come crashing down. If instead everyone ran at different speeds and arrived at random intervals, the doorway is still usable and the queue pressure is significantly lessened.

If it is okay for us to deny requests then we can also use standard Rate Limiting techniques like Leaky Bucket and Token Bucket which will reduce load on the server.

Could also use caching