**Upstream Resiliency – Protect against upstream pressure**

1. **Load Shedding:** OS has a connection queue per port, if that exhausts, new connection requests are dropped. Increase in response time could make server appear unavailable. Do not process new requests if the server is operating at capacity. How to measure overload – number of concurrent requests being processed. When the threshold is breached, fail the request fast by returning 503( Service Unavailable). Server might still pay the price for opening TLS connection. So load shedding is helpful only to an extent; if the load keeps on increasing, the cost of rejecting becomes too high degrading the service.
2. **Load Levelling:** If the clients are not expecting response immediately, load levelling can be used. Message channel. Allows the service to process packets at it’s own pace.

Diagram

Description automatically generated

Load levelling and Load shedding only protects the service from being overloaded. To address more load, the service needs to be autoscaled.

1. **Rate Limiting:** Reject a request when specific quota is breached. Quotas are applied to API Keys, users or IP addresses. Provide an appropriate error code (429 – Too many requests). Also include Retry-After in the response. Rate limiting can also be used to enforce pricing tiers. Doesn’t fully protect against DDoS because service pays the price to open a TLS connection and reads a part of the request to find out the API Key. But rate limiting helps reduce the impact. Load shedding rejects the requests based on the local state of the process while rate limiting rejects requests based on global state applicable to all service instances.
   1. **Single Process Implementation:** Buckets with sliding window. For example if the time unit for rate limiting is minute, we can divide the time into buckets of 1 min duration. A new request is normalized to the corresponding bucket. Then a sliding window of a similar length can be used that moves across time. For example in the following diagram, the requests could have come at times 12 h 1 sec, 12 h 2 s, 12 h 3 s and 12 h, 1 min 40 s. So final counter = 0.60 \* 3 + 0.4 \* 1. Sliding window ends at the last timestamp. Each bucket weight is proportional to it’s overlap with the sliding window. So we only end up storing two counters per API key.

**Diagram

Description automatically generated**

* 1. **Distributed implementation:** Need a global store because the rate limiting quota applies across all service instances. Race conditions because multiple processes would want to update the same bucket concurrently. Fetch, update and write needs to be part of the same transaction. One transaction per request so extremely slow and database needs to be scaled up linearly with the number of requests. Rather than updating DB on every request, a possible solution is to batch the updates in memory each process and flush after some time. Not accurate but reduces the pressure on DB.

**Diagram

Description automatically generated**

1. **Bulkhead –** Rate Limiting can only help with the number of requests, but what if the request is poisonous. How do you prevent attacks if the issue is intrinsic with the requests. Partition the service behind a load balancer and allow a user to only access specific partitions. So if the user is sending poisonous requests, it will only affect the specific partition and the requests can only utilize the resources of that partition. Only the other users who happen to use the same partition will be impacted.
2. **Health Endpoint:** Load shedding and rejecting incoming requests work only after the request has reached the server. What if we want to prevent sending requests to the degraded server in the first place? The service could be behind a load balancer and expose a health endpoint that returns 200 or some error code if its overloaded. The process could be taken out of the pool if the load balancer find it’s unhealthy in while querying periodically.