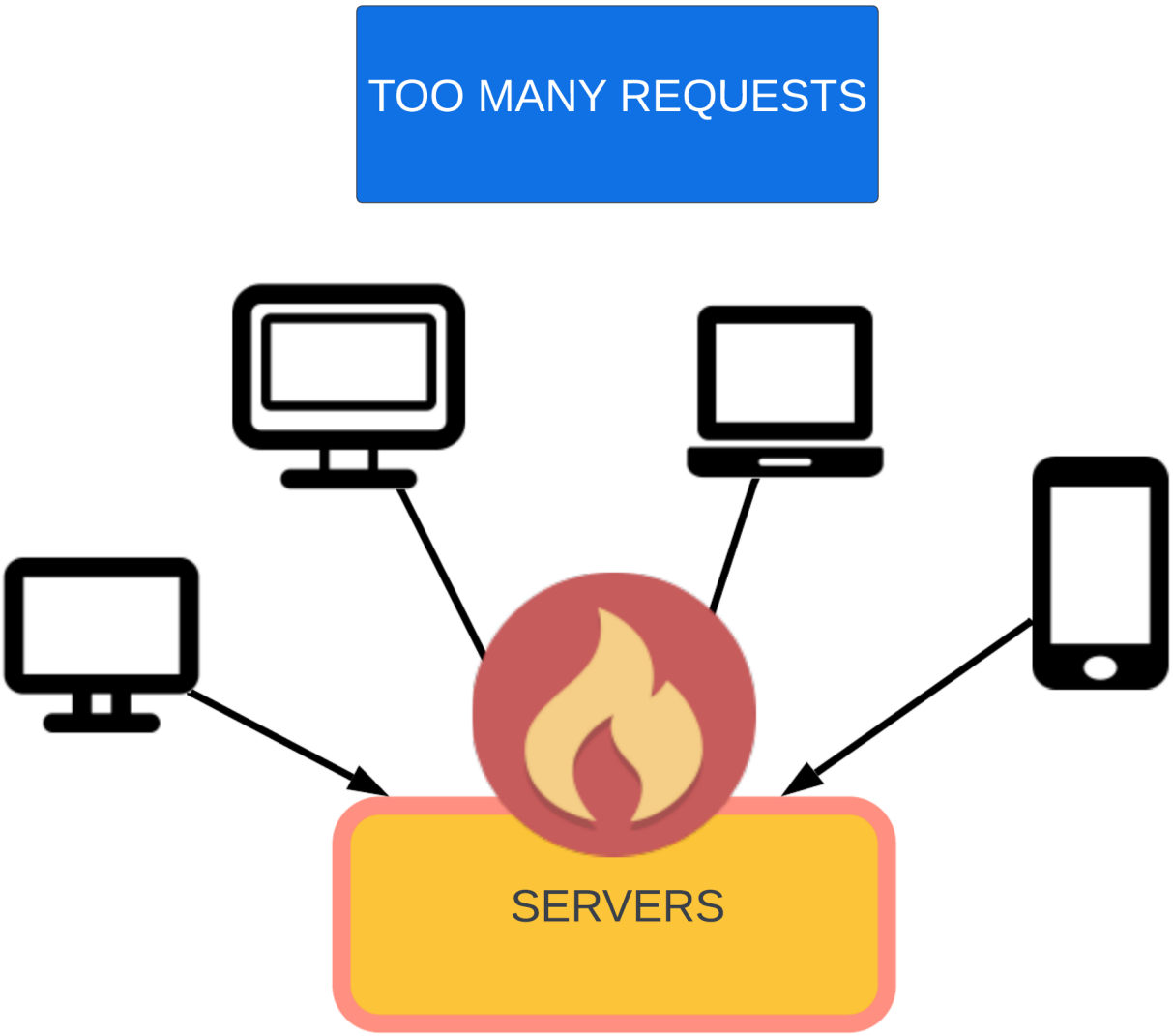
INTERVIEW READY

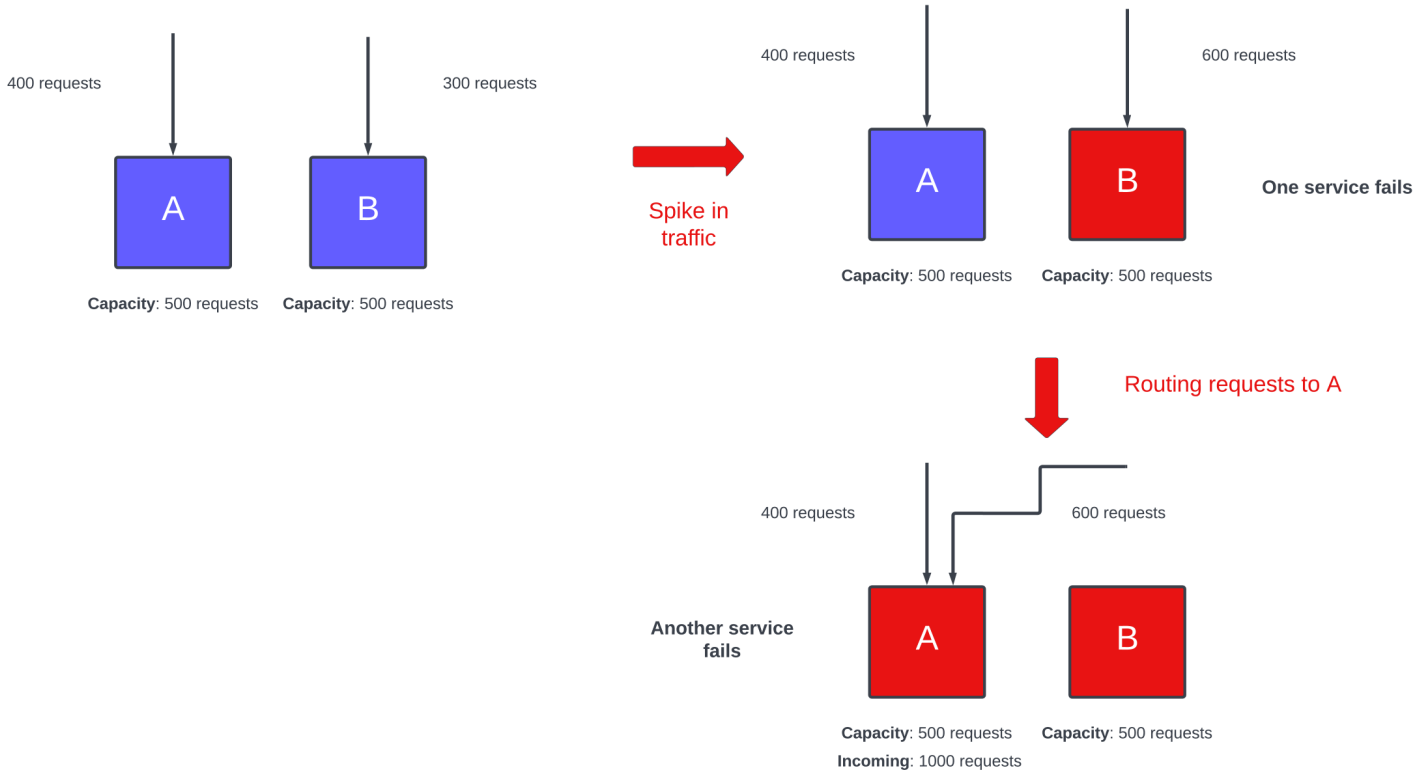
Distributed Rate Limiting

In simple terms, rate-limiting means **accepting the number of requests in proportion to how many requests your system can serve**.



Why is rate-limiting important?

- If our services are overloaded with requests then the **time required to send a response will increase** resulting in a bad user experience.
- If the number of requests keeps on increasing then our servers might crash due to **Out of Memory** exceptions.
- We want to avoid **cascading failure.**
  A cascading failure is a failure that grows over time due to positive feedback. It can occur when one service of a distributed system fails, increasing the probability that other services fail.
  Let's understand this with an example. We have two servers A and B. Both can handle at most **500 requests**. Currently, A is serving **400 requests** and B is serving **300 requests**. Now suddenly B is getting **600 requests**. It cannot handle that many requests and crashes. Now we need to route all the requests to the other server i.e., A. But A cannot handle **400 + 600** requests so it also crashes. This is known as **cascading failure**.



All the above situations can be avoided if we have a **rate liming** in our system.

Short term solutions

- **Vertical scaling**
  We can increase the number of requests each system can handle. *It can be very expensive though*

- **Efficient messaging protocol** (*something like gRPC*)
  One of the problems with traditional messaging protocols is **head of line blocking**. Suppose there are many requests in a queue and the first request is taking a lot of time to be processed then all other requests have to wait. But gRPC implements multiplexing that allows asynchronous processing of multiple requests.
- **Message compression**
  If we reduce the size of messages then our servers can process more requests.
- **Client connections**
  Creating and closing connections are expensive operations. Creating new connections for every request might be a bad idea. Instead, we can keep the connection alive for a particular time interval.
- **Pull/Push hybrid**
  If we need to send a response to a lot of users, then pushing the response to every user will become very expensive. Instead, we can ask the users to pull the response from our system.
- **Graceful degradation**
  It means our system degrades but users are not too upset about it. If our system is under tremendous load then we can ignore some features (that are not critical) temporarily.

Now let's discuss the fun part

Designing our system

Components required

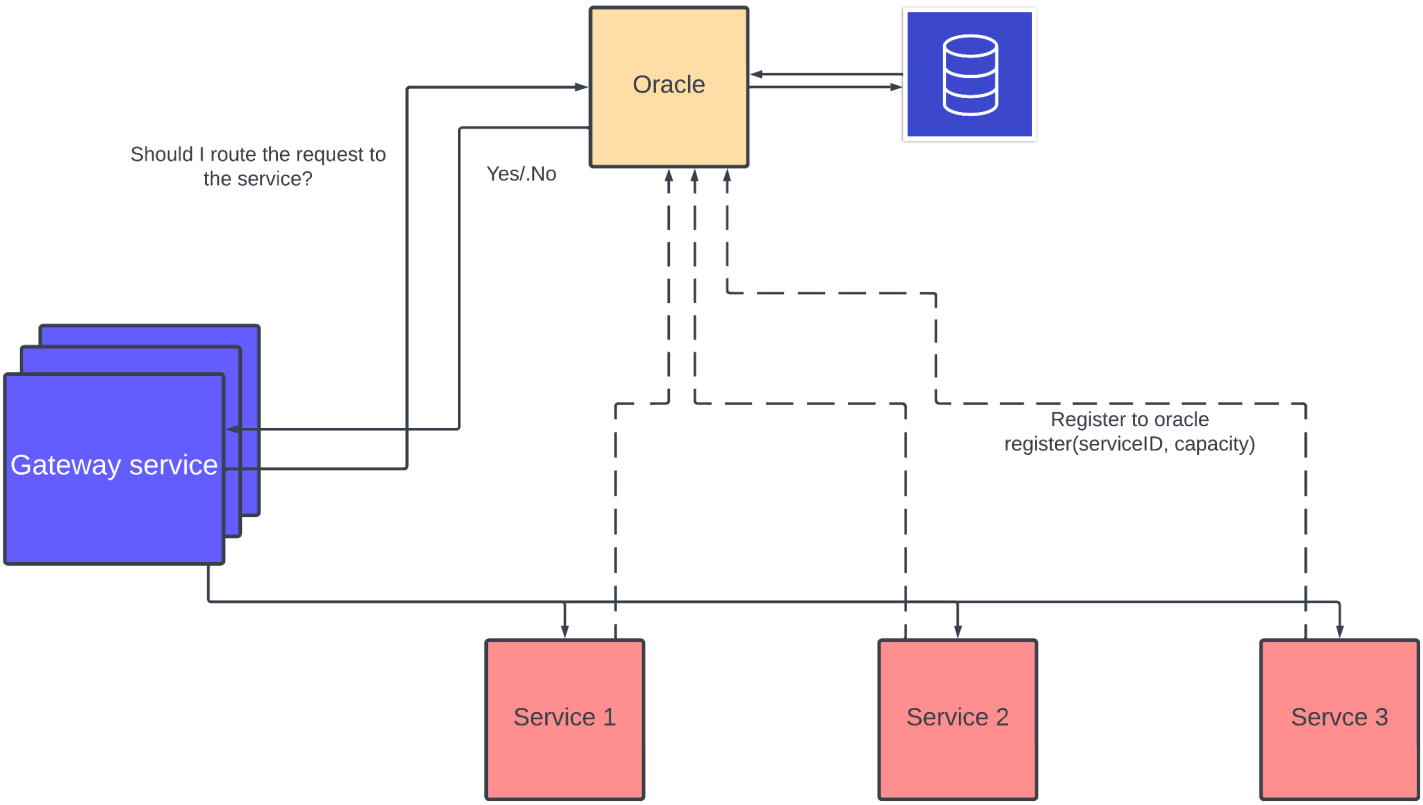
- **Oracle**
  Whenever there is a request, first the gateway sends a request asking oracle **should I process this?** and expects a boolean answer. If the oracle says yes then the gateway routes it to the required service else sends a *service unavailable* response to the user.

  But you must be thinking, **how does the oracle decide whether to route the request to the service or not?**
  Well, every time a service starts it registers itself to the oracle and sends its capacity (number of requests it can handle).

- **Gateway service**
  It asks the oracle whether to route the request to services or not. If yes then it routes requests

else sends a response of *503 service unavailable*.

- **Other services**



Rate limiting algorithm

Sliding Window Rate Limiting

In the sliding window approach, we will only allow a certain number of requests in a time interval. If the number of requests in the time interval exceeds then we won't be processing them.

Advantages of this approach

- **It is simple to implement**

However, this approach has two major issues:

- **Memory footprint**
  If our sliding window is of size **N** then we need to keep **N** requests in the memory.
- **Garbage collection**
  Every time we get a request we need to find all the requests that are no longer valid so we need to check at most **N** slots.

So is there any **better approach? Yes**

Timer Wheel

In a timer wheel,

- Size of the wheel (Number of buckets) = Timeout of the incoming request
- Each bucket is numbered from **0** to **Timeout -1**
- Each bucket can store a limited number of requests
- Every time we add a request to the bucket number **Time % Number of buckets**
- Our system can pull requests from this queue one by one.
- Before inserting a new request into a bucket it deletes all the existing requests (These requests have not been processed by the system).

Note: If there are a lot of slots then the distance between any two requests can be very large. In such cases, we will be wasting a lot of system resources and time. To tackle this issue we can use **Hierarchal Timer Wheel**.

Well now we have rate limited external requests but our internal services are also talking to each other. One service sending a lot of requests to another service might cause the other service to fail. So let's discuss how we are going to solve this issue.

Rate Limiting Internal Requests

If we are optimistic we can have a **Service Level Agreement** which states the number of requests one service can handle.

But a better approach is to have rate limiters in the service itself.

But how does a **service know that it cannot handle any more requests?**

1. **Average Response Time**
   If the average response time is increasing then either the service is not functioning or it is under too much load and cannot process requests fast enough.
2. **Age of requests**
   Let's say the service takes at most **1s** to process each request but many requests are present in the queue for more than **2s**. That means the **number of incoming requests >> Number of requests the service can process.**
3. **Dead letter queue**
   Every time our service is unable to process a request it sends the request to the dead letter

queue. If the number of requests keeps on increasing in the dead letter queue then we know that our system is failing to process requests.

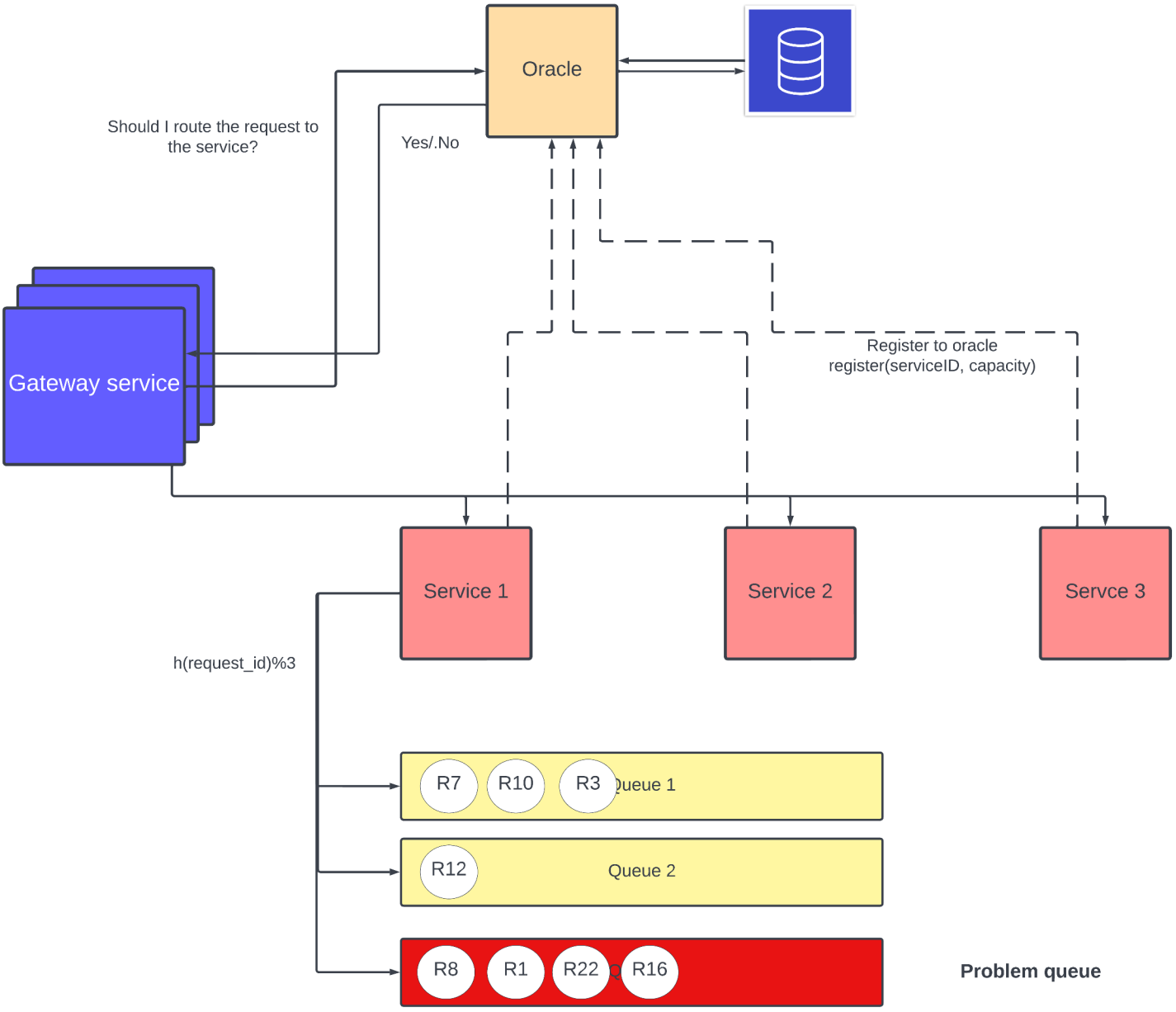
Solution

We have a queue that has all our requests. If we have a bad request in the queue (let's say **R1**) then it blocks all other requests. So we must separate **R1** from the rest of the requests.

We can do this by **partitioning the queue into smaller queues.**

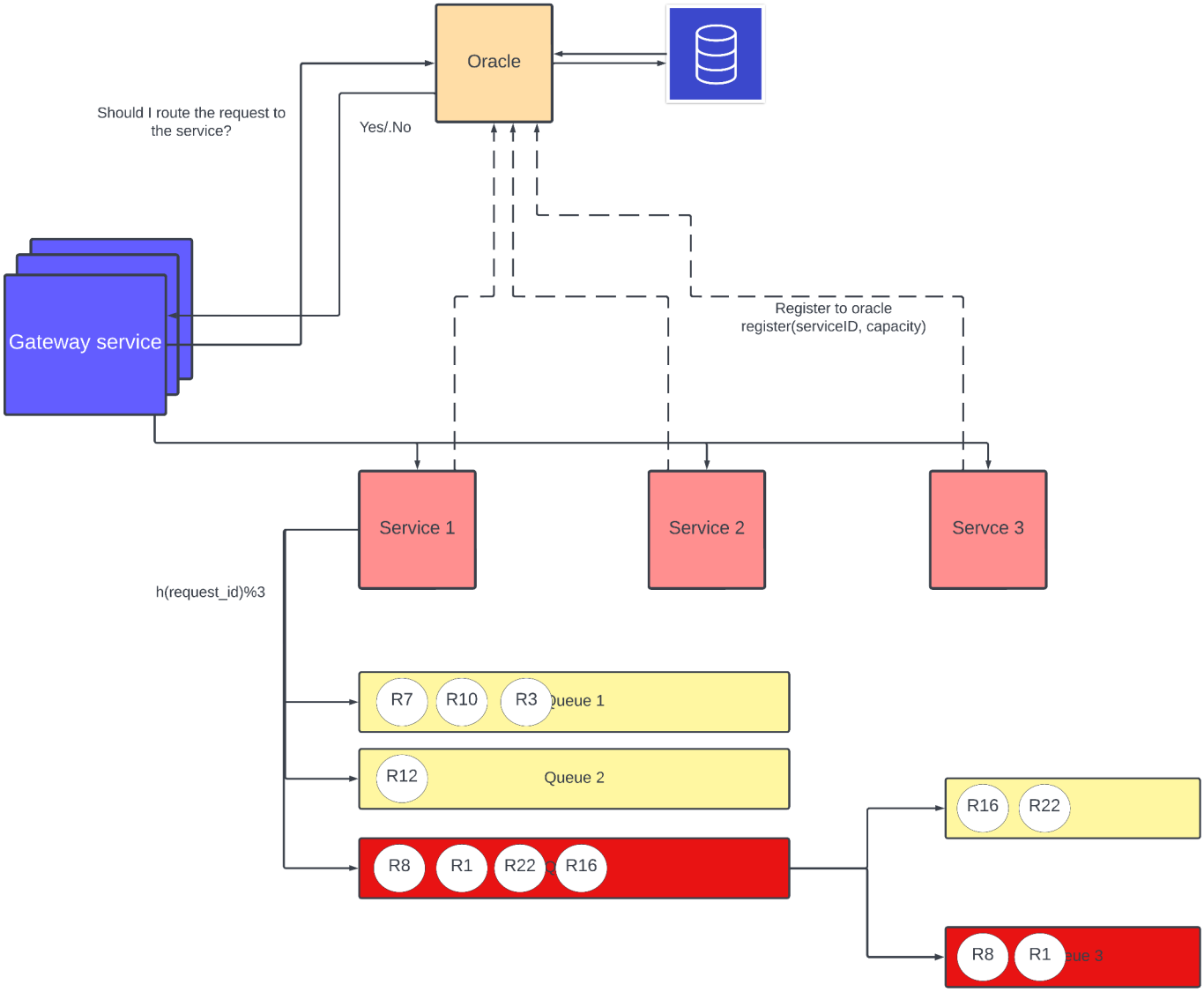
Let's say we partition it into **x** queues numbered from **0 to x-1**. Now we can use an hash function **q_num = h(request_id)%x** and send the request to the queue number **q_num**. We can see we have reduced the number of requests that were affected by **R1**.

In the worst case every queue might contain one bad request so all queues will be blocked.



Notice that at most **N/x** requests can still be blocked by **R1** (where N = the number of requests). We can further improve this process by

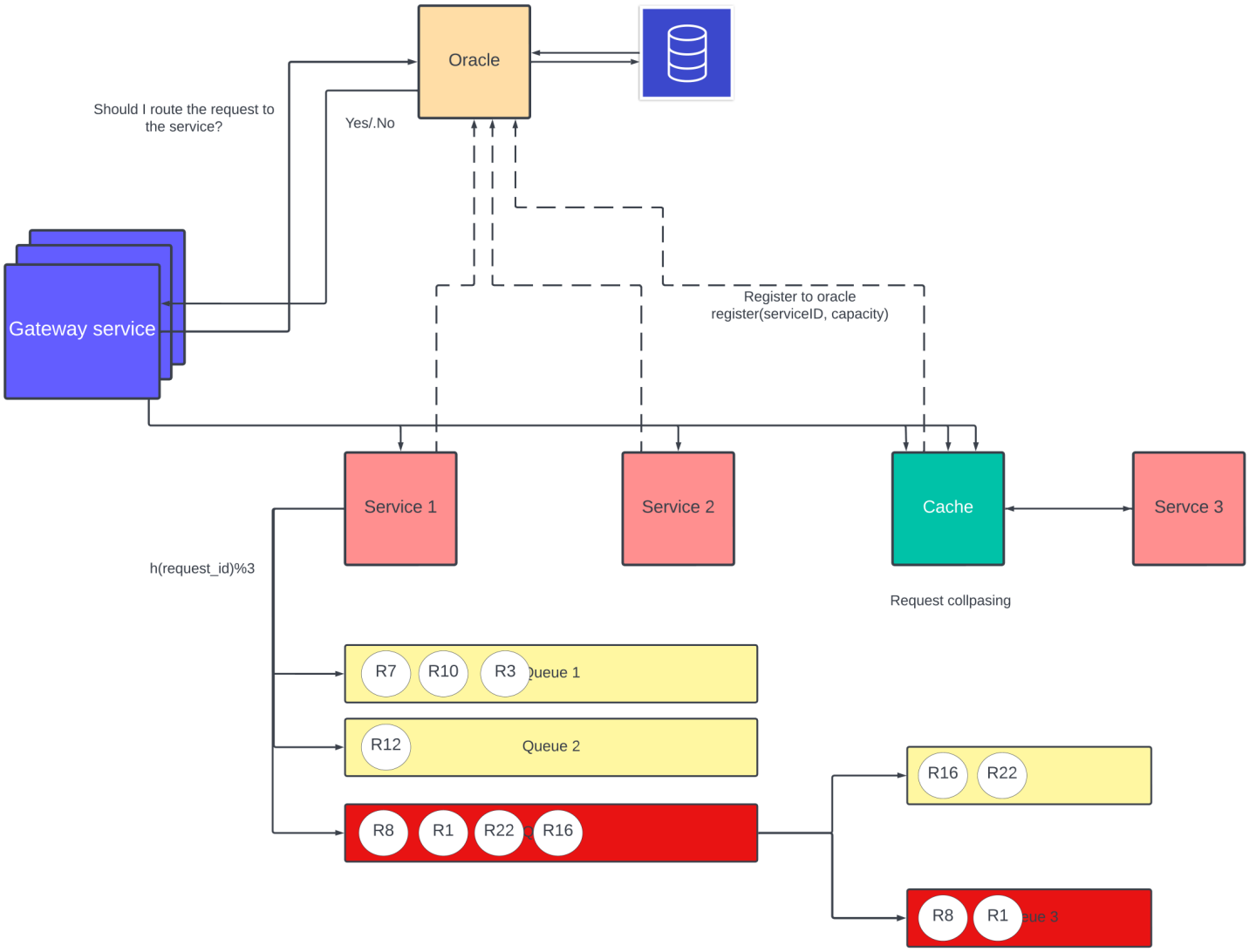
- **Increasing the number of queues**
- **Increase the size of each queue**
- **Further partitioning the overloaded queues**
  If let's say **Q3** is blocked then we can further partition **Q3** into smaller queues, and use another hash function to push requests into smaller queues. **Some requests will still suffer however we have reduced their numbers significantly.**



Some Real Life Optimizations

Request collapsing

When multiple clients are asking for the same thing. Instead of computing and sending the response we store the **request in the cache itself**. We then combine all the requests and send only one request to the server and then use the response for satisfying all the pending requests.



Client Side Rate Limiting

If our system sends an error response then the client must **check the type of error.**

If the error is permanent then the request is invalid and the client should not send the same request.

If the error is temporary then it means that there was an issue with the server and we want the client to retry.

However, we also do not want to hammer our system by the client's request so we can use an algorithm called **Exponential backoff**.

In exponential backoff, we keep increasing the time interval (Time interval increases exponentially) between subsequent requests until we reach **Max retries** after which we ask the client to stop retrying.

Capacity estimation for our system

Assumptions

- Total number of services: **100**
- Number of APIs each service is handling: **30**
- Request timeout for each API: **the 60s**
- Size of each request: **1 kB**
- We receive **10 requests** per second per API.

Memory required by the oracle

Number of buckets required by timer wheel = **Timeout for API** = **60 buckets**

Total number of API requests for all services = **30** * **10**(Number of API request per second per API) * **100** (Number of services) = **30,000**

So every second we have 30,000 requests. But we only need to store ID in the timer wheel.

Assuming size of each ID is **8 byte**. Total memory required = **8 byte * 300000** = **240 kB**

But we have 60 buckets, so the memory requirement of the timer wheel is = 240 kB * 60 ~ **15 MB**

Maximum queue wait time

Assumptions

- API has a timeout of **10s**
- Processing time of each request **1s**
- Time is taken to travel to and from client **100ms**

Maximum wait time = **API Timeout - (Time taken to travel from client to server + Time taken to travel from server to client + processing time)**
= 10s - 2*100ms - 1s
= **8.8s**

Maximum queue size required

Assumptions

- API has a timeout of **10s**
- Processing time of each request **1s**
- Time is taken to travel to and from client **100ms**
- Number of requests per second **1000 requests**

We need to process 1000 * 1kb = **1MB of data per second**.

Since the waiting time of the queue is **8.8s**, the maximum queue size = **8.8 MB**

That's it for now!

You can check out more designs on our video course at InterviewReady.