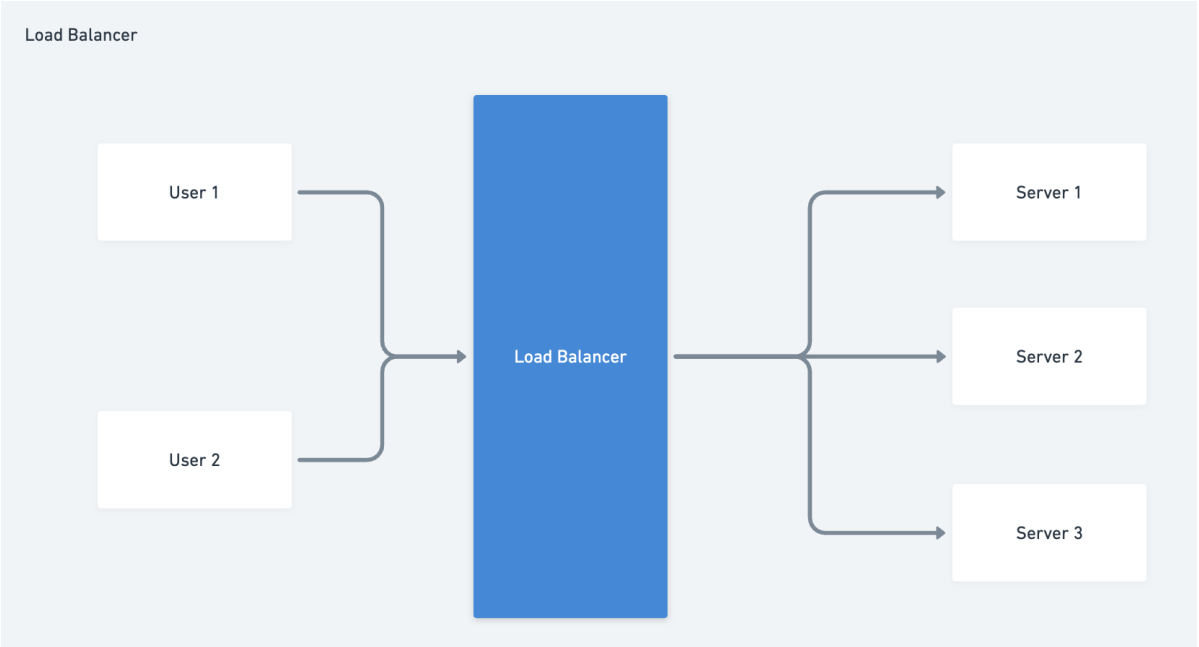
Notes - Session 8 - Load Balancers, Circuit Breakers

Load Balancers (LBs):

Load balancer is a device / service which distributes incoming network traffic across multiple backend servers.

Load Balancers (LBs) are one of the most frequently used system design components, as they are used in every distributed and scaled system.



For an external user, Load Balancer is the only point of direct contact

Why should we use a LB?

- **Distribute Traffic:** Prevent any single server from getting overloaded, thus maintaining performance.
- **Scalability:** Easily add or remove servers based on traffic demands.
- **Fault Tolerance & High Availability:** If one server fails, the LB redirects traffic to the remaining operational servers.
- **Maintain Session Persistence:** Some applications require a user's session to be processed by the same server they initially connected to. Load balancers can be configured for "sticky sessions" to handle this.

Types of LB:

- **Layer 4 Load Balancing (Transport Layer):** Based on IP address and port. It's fast and efficient since it doesn't inspect packet content.

- **Layer 7 Load Balancing (Application Layer):** Distributes requests based on content type, URL, or other HTTP header information.
  It allows for more complex and customizable distribution strategies, like directing traffic based on URL patterns or the type of device making the request.

Load Balancing Algorithms:

- **Round Robin:** Requests are distributed sequentially to each server. Suitable for servers with similar specifications.

- **Weighted Load Balancing:** Assigns a weight to each server based on its capacity and performance. Servers with higher weights receive more requests than those with less weight.

- **Least Connections:** Directs traffic to the server with the fewest active connections. Useful when servers have varying capabilities.

- **Least Response Time:** Directs traffic to the server with the fastest response time for a new connection.

- **IP Hash:** Determines the server to send a request based on the IP address of the client. Useful for maintaining client (or session) stickiness.

Popular LBs:

- Nginx
- HAProxy
- AWS Elastic Load Balancing (ELB)

Challenges and Considerations

- **Session Persistence:** As mentioned, some applications need all requests from a user session to be directed to the same server.

- **SSL Termination:** Decrypting SSL traffic at the load balancer before sending it to backend servers can offload the decryption overhead from the servers. However, this might raise security concerns in some scenarios.

- **Distributed Load Balancing:** In globally distributed systems, directing users to the nearest data center can reduce latency.

- **Scaling the Load Balancer:** In very high-traffic scenarios, even the load balancer can become a bottleneck. Solutions include DNS-level load balancing or using a tiered load balancing approach.

Circuit Breakers

Why Do We Need Circuit Breaker

In a distributed system, services often depend on other services. If a dependent service fails or becomes slow, it can cause the calling service to become slow or fail as well, leading to cascading failures throughout the system.

Circuit breakers help prevent cascading / recurring failures. Let's understand with an example (on whimsical and recorded video)

Circuit Breaker States:

A circuit breaker has three primary states:

- **Closed:** Everything is functioning normally. Requests to the service are allowed.
- **Open:** The circuit breaker has detected failures and blocks all calls to the failing service.
- **Half-Open:** After a certain time, the circuit breaker allows a limited number of test requests to pass through. If those requests succeed, the circuit breaker closes; otherwise, it remains open.

Transition Triggers:

- **From Closed to Open:** When failure rates cross a predefined threshold.
- **From Open to Half-Open:** After a predefined "reset" interval.
- **From Half-Open to Closed or Open:** Based on the success or failure of the test requests.

Advantages:

- **Fail Fast:** The system can quickly detect and isolate failures, ensuring that clients aren't stuck waiting for requests that are bound to time out or fail.
- **Resilience:** Provides a mechanism to let a failing service recover without being overwhelmed with incoming requests.

- **Feedback:** Monitoring the circuit breaker states can provide essential feedback on the system's health.

Implementation Considerations:

- **Thresholds and Timing:** You need to define thresholds (e.g., 50% failure rate) to transition from a closed to open state. Similarly, determine how long the circuit breaker should stay open before transitioning to half-open.

- **Fallback Strategies:** When a circuit breaker trips, you need a strategy. This could be serving a cached version of the data, using default values, or returning an error.

- **Network Failures vs. Application Failures:** Decide whether the circuit breaker should trip for both network and application-level errors or just one of them.

- **External Configuration and Monitoring:** Implementing circuit breakers using external libraries or services like Hystrix (from Netflix) can provide better control and monitoring capabilities.

Use-Cases:

- **Microservices:** Particularly beneficial in microservices architectures where a failure in one service shouldn't cascade to others.

- **Third-party Service Integrations:** If your system depends on third-party services, you don't control those services' uptime. A circuit breaker can prevent issues in these third-party services from affecting your system