System Design Day 3: Load Balancing and Caching

🌐 Load Balancing Algorithms – Summary

🔎 What is Load Balancing?

Load Balancing is the process of distributing incoming network traffic or application requests across multiple servers (or instances) to ensure:

- **High availability**
- **Optimal performance**
- **Scalability**
- **Fault tolerance**

It ensures no single server bears too much load, thereby avoiding bottlenecks and improving overall system reliability.

📚 Categories of Load Balancing Algorithms

1. **Static Algorithms**

- Do **not** use real-time server health/performance data.
- Ideal for **predictable, evenly distributed workloads**.

🔁 Round Robin

- Requests are distributed in a rotating order: A → B → C → A...
- **Real-Life Example:**

  A simple blog website with 3 identical stateless servers behind an NGINX load balancer.

✅ Pros	❌ Cons
Easy to implement	Doesn't adapt to load or server performance

---

🧷 Sticky Round Robin

- Like round robin but keeps client-session affinity (stickiness).
- **Real-Life Example:**

  An online shopping site where the cart state is stored in server memory.

✅ Pros	❌ Cons
Supports session persistence	Uneven distribution if users vary in activity

---

⚖️ Weighted Round Robin

- Assigns **more requests to powerful servers** using weights.
- **Real-Life Example:**

  A system with a mix of virtual and physical servers where physical machines get more traffic.

✅ Pros	❌ Cons
Adjusts for hardware differences	Manual tuning, not responsive to real-time changes

🧮 Hash-Based

- Maps request (e.g. by IP or URL) to server using a hash function.
- **Real-Life Example:**

  DNS-based load balancing that consistently routes a user's IP to the same CDN edge server.

✅ Pros	❌ Cons
Ensures repeat request consistency	Poor distribution if hash function is flawed

2. **Dynamic Algorithms**

- Adapt based on **real-time metrics** like active connections or response time.
- Best for **highly dynamic, large-scale systems**.

📉 Least Connections

- Routes new requests to the server with the **fewest active sessions**.
- **Real-Life Example:**

  A video conferencing platform like Zoom, where connections last for long durations.

✅ Pros	❌ Cons
Real-time adaptability	May overload servers with long-lived connections

⏱️ Least Response Time

- Sends requests to the server with the **fastest average response**.
- **Real-Life Example:**

  An API gateway routing traffic to different microservices based on current latency.

✅ Pros	❌ Cons
Optimizes for latency	Requires continuous monitoring and health checks

🧠 Summary Table

Algorithm	Type	Real-Time Adaptive	Session Affinity	Use Case
Round Robin	Static	❌	❌	Stateless web apps
Sticky Round Robin	Static	❌	✅	Session-based apps
Weighted Round Robin	Static	❌	❌	Mixed-capacity servers
Hash-Based	Static	❌	✅	Consistent routing (e.g. CDN)
Least Connections	Dynamic	✅	❌	Long-lived connections
Least Response Time	Dynamic	✅	❌	Latency-sensitive APIs

🎯 Choosing the Right Algorithm

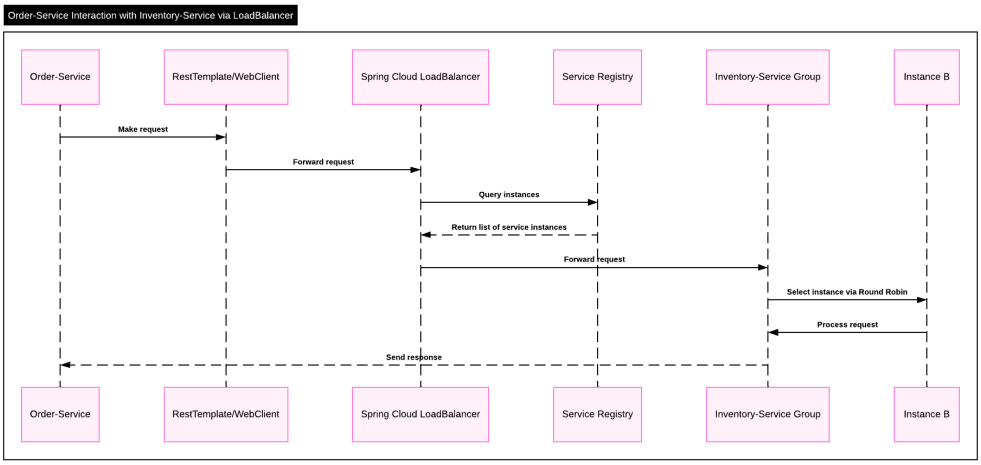
Consider these factors:

- 🔄 **Session stickiness** required?
- 📊 **Real-time monitoring** available?

- 💻 **Server heterogeneity**?
- 📈 **Traffic pattern** predictable or bursty?

> Hybrid approaches (e.g., Weighted Least Connections) are common in production systems.

How Spring Cloud Load Balancer Works



Caching Fundamentals

🔷 What is Caching?

Caching is a performance optimization technique where frequently accessed or expensive-to-compute data is stored temporarily for faster future access.

It helps to:

- ⚡ Reduce latency
- 📉 Minimize backend/database load
- 📈 Improve scalability

Cache Strategies

1️⃣ Cache-Aside (Lazy-Load)

- App checks cache → Miss → Reads from DB → Stores in cache
- App controls caching logic

✅ Ideal for: **Read-heavy** systems

Supported in: Redis, Memcached, Apache Ignite

Example:

```
data = cache.get(key);
if (data == null) {
   data = db.fetch(key);
```

```
    cache.put(key, data);
}
```

🧱 **Architecture:**

App → [Cache] → [DB]

Manual sync logic

2️⃣ Read-Through

- App reads cache → Cache automatically fetches from DB on miss
- Less app logic required

✅ Ideal for: Consistent fallback behavior

Supported in: Apache Ignite only (via `CacheLoader`)

Example:

```
cache.get(key); // auto loads from DB if not found
```

🧱 **Architecture:**

App → [Cache ⇄ DB]

3️⃣ Write-Through

- App writes to cache → Cache **synchronously** writes to DB
- Ensures consistency

✅ Ideal for: **Write consistency critical apps**

Supported in: Apache Ignite (`CacheStore`)

Example:

```
cache.put(key, value); // cache writes to DB
```

🧱 **Architecture:**

App → [Cache → DB]

4️⃣ Write-Behind

- App writes to cache → Cache writes to DB **asynchronously**
- Batch DB writes for performance

✅ Ideal for: **High write throughput**

Supported in: Apache Ignite (`writeBehindEnabled`)

Example:

```
cache.put(key, value); // DB write deferred
```

🧱 **Architecture:**

App → [Cache → (delayed write) → DB]

5️⃣ Write-Around

- App writes directly to DB
- Cache is updated on next read (if needed)

✅ Ideal for: Data not frequently read after write

Supported in: Redis, Memcached (custom logic), limited support in Ignite

Example:

```
db.save(data); // write directly
data = cache.get(key);
if (data == null) {
    data = db.get(key);
    cache.set(key, data);
}
```

🧱 **Architecture:**

Writes: App → DB

Reads: App → Cache → DB (if miss)

🔃 Cache Invalidation

Invalidation ensures stale data is removed from cache.

Common Strategies:

Method	Description
TTL (Time-to-Live)	Expire cache entry after set time. *(e.g., 10 mins)*
Explicit Invalidation	App manually removes cache after DB update.
Write-Through / Write-Behind	Automatically keeps DB and cache in sync.
LRU (Least Recently Used)	Evict based on usage pattern. Built-in in Redis.

🏛️ Cache Layers

Layer	Description	Tools
Local Cache	In-memory cache on each node (fastest)	Java HashMap, Caffeine
Distributed Cache	Shared cache across cluster or services	Redis Cluster, Apache Ignite, Memcached

💡 **Best Practice:**

Use local cache for ultra-low latency → fallback to distributed cache → fallback to DB.

📊 Comparison Table Summary

Strategy	Read Path	Write Path	Best Use Case	Redis	Ignite	Memcached
Cache-Aside	App → Cache → DB	App → DB → Cache	Read-heavy apps	✅	✅	✅
Read-Through	App → Cache	App → DB	Simplified app logic	❌	✅	❌
Write-Through	App → Cache → DB	App → Cache → DB	High consistency writes	❌	✅	❌
Write-Behind	App → Cache → DB*	App → Cache	High write throughput	❌	✅	❌

Strategy	Read Path	Write Path	Best Use Case	Redis	Ignite	Memcached
Write-Around	App → DB → Cache	App → DB	Infrequent post-write reads	✅	⚠️	✅

✅ Summary Recommendations

Scenario	Recommended Strategy	Tools
High Read, Low Write	**Cache-Aside**	Redis, Ignite
Reads with minimal app logic	**Read-Through**	Apache Ignite
Strong consistency on writes	**Write-Through**	Apache Ignite
Bulk writes, high performance	**Write-Behind**	Apache Ignite
Rarely accessed after write	**Write-Around**	Redis, Memcached
TTL-based invalidation needs	Any + TTL	Redis, Ignite
Multi-node scalability	Distributed Cache	Redis Cluster, Ignite