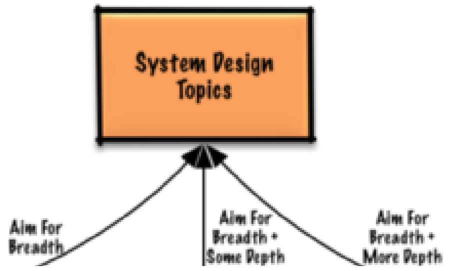
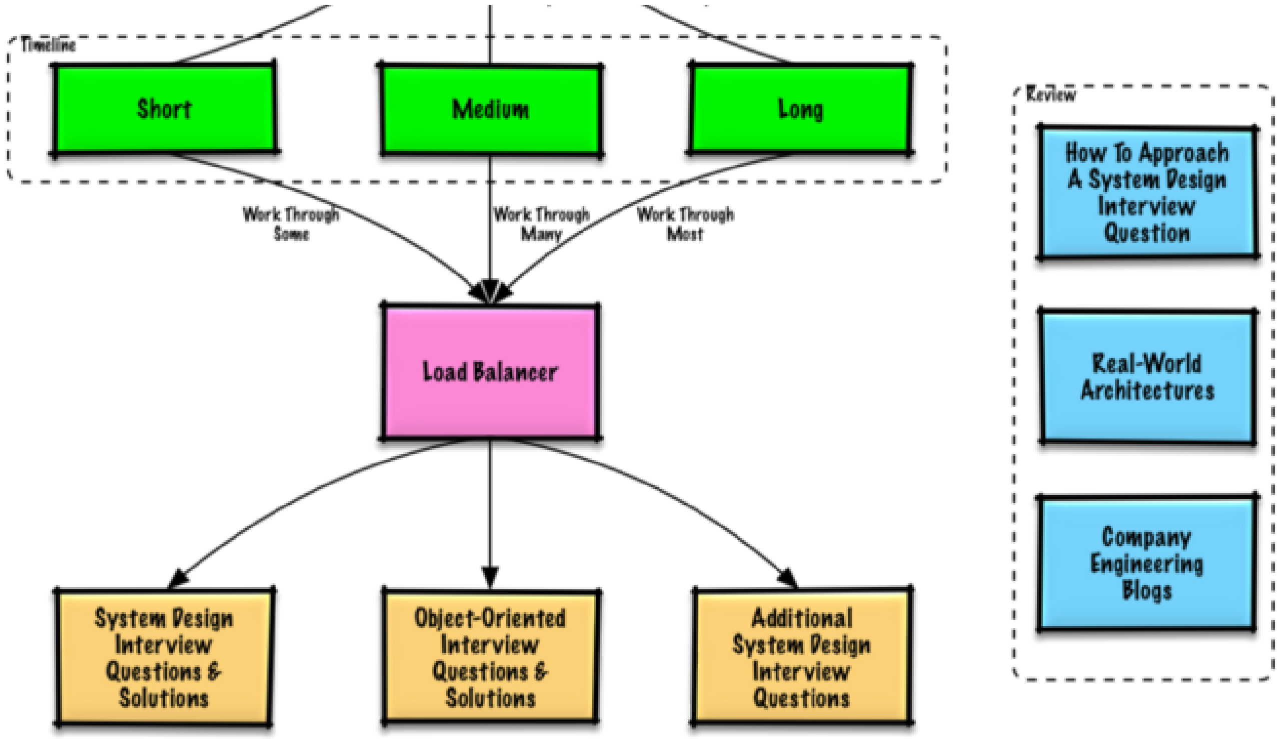
Basic Concepts You Need to Know about Building Large Scale Distributed System

Peng Yang Follow
Jul 20, 2019 · 5 min read ★




Timeline
Short
Medium
Long
Work Through Some
Work Through Many
Work Through Most
Load Balancer
System Design Interview Questions & Solutions
Object-Oriented Interview Questions & Solutions
Additional System Design Interview Questions
Review
How To Approach A System Design Interview Question
Real-World Architectures
Company Engineering Blogs

- *This article aims to share the concepts of building/designing large scale distributed system for every software engineer or architect.*
- *References of this article include but not limit to: Designing Data-Intensive Applications, educative, github and github.*

Table of Contents

1. Key Characteristics
2. Loading balancing
3. Caching
4. Sharding / Data Partitioning
5. Indexes
6. Proxies
7. Queues
8. Redundancy and Replication
9. SQL vs. NoSQL
10. CAP Theorem
11. Consistent Hashing
12. Client-Server Communication
13. Other Topics

1. Key Characteristics

1.1 Reliability

Concepts

The system should continue to work *correctly* (performing the correct function at the desired level of performance) even in the face of *adversity* (hardware or software faults, and even human error). In simple words, “continuing to work correctly, even when things go wrong.”

- **Hardware Faults**: Our first response is usually to add redundancy to the individual hardware components in order to reduce the failure rate of the system. Disks may be set up in a **RAID configuration**, servers may have **dual power supplies** and **hot-swappable CPUs**, and datacenters may have **batteries and diesel generators** for backup power. When one component dies, the redundant component can take its place while the broken component is replaced.
- **Software Errors**: Carefully think about assumptions and interactions in the system; thorough testing; process isolation; allowing processes to crash and restart; measuring, monitoring, and analyzing system behavior in production.
- **Human Errors:** Minimize opportunities for errors.

How to achieve

- Reliability is achieved through redundancy of components and data (remove every single point of failure).

1.2 Scalability

Concepts

- As the system *grows* (in data volume, traffic volume, or complexity), there should be reasonable ways of dealing with that growth.
- The capability of a system to grow and manage increased demand.
- A system that can continuously evolve to support growing amount of work is scalable.

How to achieve

- Horizontal scaling: by adding more servers into the pool of resources.
- Vertical scaling: by adding more resource (CPU, RAM, storage, etc) to an existing server. This approach comes with downtime and an upper limit.

Four golden rules for scalability (Clones, Database, Cache, Asynchronism**)**

Clones (horizontal scaling)

- Public servers are hidden behind a load balancer, it evenly distributes load (requests from your users) onto your group/cluster of application servers.
- Every server contains exactly the same codebase and does not store any user-related data, like sessions or profile pictures, on local disc or memory.
- Sessions need to be stored in a centralized data store which is accessible to all your application servers. It can be an external database or an external persistent cache, like Redis.
- Use tools such as Capistrano to deploy code to each servers.

Database

- Master-slave replication (read from slaves, write to master) and upgrade your master server by adding RAM, RAM and more RAM.

- Sharding, denormalization, and SQL tuning
- Denormalize right from the beginning and include no more Joins in any database query. You can stay with MySQL, and use it like a NoSQL database, or you can switch to a better and easier to scale NoSQL database like MongoDB or CouchDB. Joins will now need to be done in your application code.

Cache

- Cache here usually means in-memory caches like Memcached or Redis (Redis includes extra database-features like persistence and the built-in data structures like lists and sets). Please **never do file-based caching**, it makes cloning and auto-scaling of your servers just a pain.
- A cache is a simple key-value store and it should reside as a buffering layer between your application and your data storage. Whenever your application has to read data it should at first try to retrieve the data from your cache. Redis can do several ***hundreds of thousands*** of read operations per second when being hosted on a standard server.
- 2 patterns of caching data:
  1. **Cached Database Queries:** Whenever you do a query to your

database, you store the result dataset in cache. A hashed version of your query is the cache key.
2. Cached Objects: see your data as an object like you already do in your code (classes, instances, etc.). Let your class assemble a dataset from your database and then store the complete instance of the class or the assembled dataset in the cache. Some ideas of objects to cache:

```
1. user sessions (never use the database!)
2. fully rendered blog articles
3. activity streams
4. user<->friend relationships
```

Asynchronism

- Async #1: **doing the time-consuming work in advance and serving the finished work with a low request time**. It is to turn dynamic content into static content. Pages of a website, maybe built with a massive framework or CMS, are pre-rendered and locally stored as static HTML files on every change. Often these computing tasks are done on a regular basis.

- Async #2: **handle tasks asynchronously.** RabbitMQ is one of many systems which help to implement async processing. You could also use ActiveMQ or a simple Redis list. The basic idea is to have a queue of tasks or jobs that a worker can process.

1.3 Availability

- Availability is the time a system remains operational to perform its required function in a specific period.
- Measured by the percentage of time that a system remains operational under normal conditions.
- A reliable system is available.
- An available system is not necessarily reliable.
- A system with a security hole is available when there is no security attack.

1.4 Efficiency

- Latency: response time, the delay to obtain the first piece of data.

- Bandwidth: throughput, amount of data delivered in a given time. Check this link to see how to calculate bandwidth requirements.

1.5 Maintainability / Serviceability / Manageability

- Over time, many different people will work on the system (engineering and operations, both maintaining current behavior and adapting the system to new use cases), and they should all be able to work on it *productively*.
- Easiness to operate and maintain the system.
- Simplicity and spend with which a system can be repaired or maintained.

Thank you for reading! if you liked this blog, please also check my other blogs of System Design series.

- System Design — Load Balancing
- System Design — Caching
- System Design — Sharding / Data Partitioning
- System Design — Indexes

- System Design — Proxies
- System Design — Message Queues
- System Design — Redundancy and Replication
- System Design — SQL vs. NoSQL
- System Design — CAP Problem
- System Design — Consistent Hashing
- System Design — Client-Server Communication
- System Design — Storage
- System Design — Other Topics

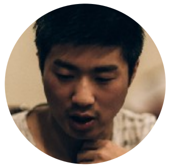
Web Development System Design

6 claps

WRITTEN BY

Peng Yang

Follow



Software/Infrastructure Engineer. Aim to become an engineer who understands computer science very well. https://www.linkedin.com/in/peng-larry-yang-9a794561/

Computer Science Fundamentals

Follow

Computer Science Fundamentals

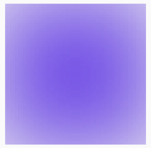
Write the first response

Exploring NestJS Workspace

Santosh Yadav in Better Programming

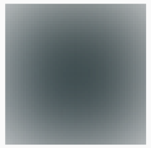
What we are looking forward to in Terraform 0.13

Marko Bevc in The Scale Factory



Adding References In Rails Using The Command Line

Jake Bartlow



A Tutorial on Modern Multithreading and Concurrency in C++

The Educative Team in educative