Table of Contents:

Common Terms:

- reliability
- faults
- RAID
- scalability
- load parameters
- throughput
- fan-out
- latency
- response time
- percentiles
- elasticity
- operability
- simplicity
- extensibility
- head-of-line blocking
- SLOs and SLAs
- tail latencies

Reliability

Definition

Reliability is the ability of a system to continue to work correctly even when things go wrong.

This means that the application:

- performs the functions that the user expects
- can tolerate user mistakes or unexpected usage of the software
- provides a good-enough performance under the expected load and data volume
- prevents unauthorised access and abuse

Faults

Things that can go wrong in a data system are called faults.

Systems that can survive different faults are described as *fault-tolerant* or *resilient*.

A system can experience:

Hardware Faults

These faults are usually: - random - usually indpendent from each other (one machine's failing disk doesn't necessarily cause another machine's disk to fail)

We usually counter these faults by imposing hardware redundancy:

- Setting up disks in RAID configuration
- Using servers with dual power supplies and hot-swappable CPUs

While this reduces the probability of hardware faults sufficiently for most use cases, applications with high demands for data volumes and computing need larger number of machines, which proportionally increases the rate of hardware faults.

For such system, it is good to design them in a way that tolerates entire machine loss.

Software Faults

These faults are usually:

- software bugs
- harder to anticipate
- systemetic errors
- usually correlated across nodes

There is no obvious solution for such errors, but they can be avoided by:

- thorough testing
- carefully thinking about assumptions and interactions in the system
- allowing processes to crash and restart
- process isolation
- monitoring system behaviour in production

Scalability

Definition

Scalability id the term used to describe a system's ability to cope with increased load.

A system can't be described as *absolutely scalable*, but we should rather indicate the load increase constraints with which system can cope.

Describing Load

Load can be described with a few numbers which we call *load parameters*.

The best choice of **load parameters** depends on the architecture of the system.

Examples of load parameters:

- requests per second to a web server
- ratio of reads to writes in a database
- number of simultaneously active users in a chat room
- hit rate on a cache

Working Example

Let's focus on two main features of twitter:

- Post tweet: a user can publish a new meesage to their followers
- Home timeline: a user can view tweets posted by the people they follow

Load parameters can be defined based on these 2 features:

- Post tweet:
  - 4.6k requests/sec on average
  - over 12k requests/sec at peak
- Home timeline:
  - 300k requests/sec

Handling 12000 writes per second at peak is easy to achieve. However, the scaling challenge here is not primarily due to the tweek volume, but rather due to fan-out; each user follows many people and each each user is followed by many people.

This poses a challenge in order to transmit a user's tweet to all the followers and/or to fetch all tweets of a user's followees when the user accesses the home timeline.

These operations can be implemented in 2 ways:

- **Compute home timeline tweets with database joins**

  - On user tweet: insert the new tweet into a global collection of tweets
  - On user home timeline request: look up all the people they follow, find all the tweets for each of them, merge and sort them by time.

- **Precompute Home Timeline Tweets in a Cache (Fan-Out)**

  - Maintain a cache for each user's home timeline.

    - On user tweet: lookup all the people who follow that user, and insert the new tweet into each of their home timeline caches.
    - On user home timeline request: fetch tweets from the home timeline cache.

Choosing the best approach

Looking at the load parameters, we see that the rate of home timeline requests per second is at least 2 orders of magnitude higher than the post tweet rate (300k requests/sec >>>> 4.6k requests/sec).

In other words, we have much more read operations than write operations.

Using database joins provides very fast write operations (inserting 1 record in a db table). But it provides slow read operations (need to join several tables).

In terms of read operations, using a home timeline cache seems promising because the result will be precomputed ahead of time, which makes read operations very fast. But write operations will become slower as the number of followers of the tweeting user increases.

On average, a user has 75 followers. This means that we need to perform an average of 345k (4.6k x 75) writes to home timeline caches per second.

However, relying only on the average number of followers is not enough because it hides the fact that different users can have different numbers of followers. Famous users, like celebrities, can have over 30 Million followers, which means that a single tweet from a famous celebrity can result in over 30 Million write operations to home timeline caches.

Twitter tries ti deliver tweets to followers within 5 seconds, which is significantly challenging to achieve with over 30 Million writes.

For this reason, we need to consider a new load parameter: **the distribution of followers per user** (we can also weight this by how often a user tweets). This is a key load parameter since it determines the fan-out load.

At the end, twitter adopted a hybrid approach where the latter approach is applied to most users' tweets (since most users have limited number of followers), and the former approach is applied to tweets of users with a significantly large number of followers (like celebrities). In other words, if a user is following a celebrity, the celebrity tweets aren't fanned out into the user's home timeline cache, but rather fetched seperately on read time using approach 1.

Describing Performance

Once we have described our system's load, we can investigate what happens when the load increases.

We can think about this in two ways:

- When you increase a load parameter and keep the system resources (CPU, memory, network bandwidth etc.) unchanged, how is the performance of your system affected?
- When you increase a load parameter, how much do you need to increase the resources if you want to keep performance unchanged?

These questions require performance numbers to be answered.

Examples of performance parameters include:

- throughput: the number of records we can process per second (applicable in Hadoop for example)
- response time: time between a client sending a request and receiving a response (applicable in webservers).

Note: **Response time** is not the same as **latency**:

- Response time is what the user sees: it is the time it takes to process the request (service time) + network delays + queueing delays.
- Latency is the duration that a request is waiting to be handled (during which it is *latent* awaiting service).

Measuring Response Time

Making the same request to the same system multiple times can yield different response times. This is why we need to measure response time as a **distribution** instead of a single number.

The mean response time is not enough because it doesn't give an idea about the variations of different samples. Outliers having large response times can be coming from the bigger customers (for example more data to process), so we should ensure these are taken into account as well.

For this reason, the best way to describe a system's response time is using percentiles:

- median (or p50): indicates that half users get a response time lower than this value.
- 95th percentile (or p95): indicates that 95% of users get a response time lower than this value.
- 99th percentile (or p99): indicates that 99% of users get a response time lower than this value.
- 99.9th percentile (or p99): indicates that 99.9% of users get a response time lower than this value.

High percentiles of response time are known as **tail latencies**, and they are impoerant because they directly affect users' experience of the service.

For example, Amazon dsecribes response time requirements for internal services in terms of the 99.9th percentile, even though it only affects 1 in 1000 requests. It is because such users have more data to process in their requests, which means they are valuable customers because they have made many purchases.

However, optimising for for p99.9 is too expensive. Reducing response times at very high percentiles are usually affected by random events outside of control and the benefits are diminishing.

SLOs and SLAs

We usually define SLOs and SLAs, which are contracts defining the expected performance and availability of a service.

SLA Example:

An SLA may state that the service is considered to be up if it has a median response time of less than 200 ms and 99th percentile under 1s.

Head-of-Line Blocking

Queueing delays account for a large part of the response time at high percentiles. A server can only process a small number of things at parallel.

A small number of slow requests can hold up the processing of subsequent requests, this is know as head-of-line blocking.

For this reason, when measuring response time, we should do it on the client side and also keep sending requests independently of the response time because if we wait for the previous request to be completed before sending the next one it will result in keeping the queues shorter in our test compared to what they will be in reality.

Coping with Load

We have several techniques to cope with load, including:

- Horizontal scaling / Scaling out
  - known as shared-nothing architecture
  - services should be stateless across replicas
- Vertical scaling / Scaling up
  - simpler but expensive

A system that can automatically cope with load is described as **elastic**.

Maintainability

Software should be built in a way that reduces the pain during maintenance and thus avoid creating legacy software ourselves.

There are three design principles that we should follow to achieve this:

- Operability: make it easy for ops teams to keep the system running smoothly
- Simplicity: make it easy for new engineers to understand the system, removing as uch complexity as possible from the system (use abstractions for example)
- Evolvability: make it easy for engineers to make changes to the system in the future, adapting it for new unanticipated cases such as requirements changes. (also known as extensibility).