Jahong Liu

System Design Basics 2: Design Requirements

Design Requirements

The requirements that need to met to satisfy the quality standards of a large, and effective distributed systems.

We don't need to know the nitty gritty details of every single component – it is more about the throught process and analysis that goes behind creating an effective distributed system. As long as you have a solid idea of what direction we are going in, and can discuss trade-offs among different choices, it is acceptable

---

Thinking in System Design

Whenever we are designing a big enterprise system, it can be boiled down to three points:

Moving Data

As we discussed earlier in the chapter on computer architecture, data is moved between the disk, RAM and CPU. However, when designing large systems, our focus shifts to moving data between different clients and servers, which may be geographically dispersed across the world. This is significantly more challenging compared to local data movement.

Storing Data

We already discussed how storing data in RAM is different to storing data on disk. When designing large distributed systems, it is guarantee that we will need to store data. The question is, how do we store the data? Databases? Blob Stores? File Systems? Distributed file systems? This might remind you of picking between different data structures to find the optimal solution. For example, choosing an array over a BST to store data doesn't mean that an array is better than a BST in all scenarios, but rather it depends on the use case. By the same token, we have to choose how we want to store data and which way will be the most efficient, given the scenario.

Transforming Data

We also want to transform data. It wouldn't be very fun if all we were doing was moving and storing data. If we were given a bunch of server logs, one way to transform this data would be to output the % of successful requests vs % of failed requests. This is sometimes

handled by a monitoring service. Perhaps, we are given some medical records, and we want to filter the patients by age. These are just two basic examples, and there are countless ways to transform data. Regardless of how complex or how simple the process, the fundamental question is: what is the most efficient way to transform the given data.

Generally, in large systems, we prefer horizontal scaling, as it is much more powerful, and can be achieved with commodity hardware (i.e., relatively inexpensive, standard hardware). However, it also requires much more engineering effort, as we need to ensure that the servers are communicating with each other, and that the user requests are being distributed evenly.

Note

In the case of data structures and algorithms, if we picked a bad algorithm, changing the code wouldn't be that challenging. For example, if you write bad code, you can always fix that code without many adverse effects.

Bad design choices in the application architecture can be very costly. Choosing the wrong database will have more severe consequences. You will have to migrate data from one database to another database and at the same time, rewrite portions of the application.

What is good design?

The next question is: what constitutes a good design? Well, there are a number of factors, performance measures, and certain metrics are deemed as good design. These factors are good starting points when you start to compare and contrast, i.e. figuring out tradeoffs of design choices.

Avaliability

Availability is at the heart of an effective system. Availability refers to the percentage of time the system is available, as in, up and running for a given period of time. Traditionally, the availability requirement was Monday-Friday, 9am to 5pm(typical workday hours). Nowadays, the systems we design need to have global connectivity, a near 27/7 operation, where requests to access the system can be made concurrently, from different timezones. Mathematically, availability is calculated by:

Availability = uptime/(uptime + downtime), where uptime refers to the total amount of time the system is up and downtime refers to the amount of time the system was not available to the users.

Downtime can be planned or unplanned. For example, downtime because of a software update, verification, or back-up is planned. But, what if there is a hardware or software failure? Perhaps a natural disaster? It's hard to predict unplanned downtime, but the chances of it occurring can be minimized.

Using the equation above, let's say that we had an uptime of 2323 hours out of 2424 hours. This would result in a total availability of 96%96%. From a system design and a business perspective, this is rather poor, because we are losing money and users for 1 hour a day.

Of course, it is ideal to have 100%100% availability, but it just simply is not possible due to unplanned downtimes. Therefore, companies will aim for *at least* 99%99% availability. However, even 99%99% uptime means that out of 365365 days, the system would be down for 3.653.65 days. If we want to reduce the downtime by a factor of 1010, this would take our uptime to 99.9%99.9% and downtime to 0.1%0.1%. This is a big jump. Ultimately, availability is measured in terms of 99s.

A good target for companies to have is 99.999%99.999% availability, which is 55 minutes of downtime in 365365 days. This can be hard to achieve, but is important for mission critical systems.

The measure of availability is used to define SLOs (service level objectives) and SLAs(service level agreements).

SLA refers to an agreement aa company makes with their clients or users to provide a certain metric of uptime, responsiveness, and responsibilities. SLO refers to an objective your team must hit to meet the SLA requirements. For example, AWS’s monthly SLA is99,99% and if not met, they refund a percentage of service credit.

Reliability, Fault Tolerance, and Redundancy

Reliability refers to the system's ability to perform its intended functions without failure or errors over a specified period of time. When requests are made from our server, the server might be available, but when discussing reliability, we are talking about the probability that the server won't fail. If thousands of users are making requests, or if there are DDoS attacks, how easily does our server go down? This brings us to fault tolerance.

If one portion of our system has a fault, i.e. it fails, and we have another server, it means that our server is somewhat fault tolerant. Fault-tolerance refers to how well the system can detect and heal itself from a problem, i.e. disable a function, revert to a different mode, switch to a different server.

To have fault tolerance, we can have a redundant server. Redundancy in english refers to something that is unessential. This redundancy is provided by our backup server which essentially "shadows" the contents of a server. We don't need this server, but it only comes into play if our primary server fails. Only by having this redundancy, are we able to have fault tolerance.

In this case, the second server is simply a backup. But, what if we had two servers that were both active? This would be called active-active redundancy.

Another measure of throughput is queries/second, which is a measure of the number of requests made by the user to a server or database. Another one is bytes/second. This refers to the maximum amount of data that can be sent over a network at any given time.

Throughput

Throughput refers to the amount of data or operations we can handle over some period of time. The throughput of a client making requests to a server would be measured through the number of requests per second. If we want to improve the throughput, we can perform vertical scaling. This makes sense, but as we discussed earlier, there are limitations to vertical scaling. This is where horizontal scaling would come in.

Another measure of throughtput is queries/second, which is a measure of the number of requests made by the user to a server or databse. Another one is bytes/second. This refers to the maximum amount of data that can be sent over a network at any given time.

Queries per second makes more sense when we are discussing design in terms of users. But what if we had a data pipeline where we are required to process given data in a different format? Here, the data isn’t related to a single user, per se, so bytes/second would make more sense here.

Latency

Latency refers to the delay between the client making the request and the server responding to that request. So, while throughput refers to how many requests can be sent over a newwork per second, latency refers to the amount of time it makes for each individual request to be completed. Latency is not exclusive to networks, however. Recall that latency even exists within a computer's internal components, such as the RAM and the cache making requests from the CPU.

Closing Notes

We want to design effective systems that can handle failures, have a high throughput, high availability, and low latency.