Non-Abstract Large Scale Design Workbook

Introduction

This short workbook is a companion to the Non Abstract Large Scale Design distributed systems engineering exercise. This is not an exhaustive guide to distributed systems engineering - it's just a high-level methodology for attacking a large-scale design problem.

Do one thing and do it well

When you design your system try to split it into services that each do one thing and do it well. This gives different parts of your system isolation from each other, makes it easier to troubleshoot problems in the system, and can make it easier for the system to evolve over time.

It also simplifies performance analysis: your service does one thing, analyse that work. When capacity planning for that service or load testing you don't need to think about different traffic mixes.

Prefer simple, stateless services where possible

Your system is going to need to have state somewhere, in order to be useful. Try to keep the state in specific components and keep the rest stateless. Stateless components are those which do not need to store state that persists for longer than one request. Stateless components are easy to manage because they are trivial to scale horizontally.

Assumptions, constraints and SLOs

Service Level Objectives (SLOs) are the performance and reliability goals that your system is intended to meet. They should be specified in terms of metrics that you can collect or generate.

Before you start designing, take a careful look at the constraints and non-functional requirements for your system. Have they been stated? Are they comprehensive?

You should understand (at a minimum):

- What the system is supposed to do (business logic)
- What are the SLOs around performance, error rates/availability, data loss? If no SLOs have been defined you may need to define them, with the business owners.
- Where should it run? What is the budget, or what hardware is available? (you may have to make these decisions)
- Expected workload (this can be difficult to gauge)

Getting to an initial high-level design (or analysing an existing one)

Here we are designing systems from scratch, but much of this process is equally applicable to understanding a running system or a proposed design.

API

Figure out the API for the system. What operations are available externally? What data goes in, and what results come out? Does the operation change the system's state, or is it read-only? This is fundamental - you need to be able to do this in order to make progress on creating or understanding a design.

Each set of closely-related operations is likely to be a service in your system (each individual operation may be a service). Many systems have a stateless API layer that coordinates with other parts of the system to serve results.

Here's an example: You are asked to design a service that can store images for users. It provides a way to upload new images to the system. Users should be able to see their photos, organised as pages of thumbnails organised sequentially, and to view the original version of any photo. There are already-existing services you may use to deal with authentication and authorization, which allocate a UUID to each user in the system and deal with all security and privacy related functions.

The API could look like this:

- `store_image(uuid, image_data)` -> success/failure
- `get_thumbnails(uuid, start_index)` -> set of thumbnails, next_index
- `get_image(uuid, image_index)` -> original image data

get_thumbnails and get_image use an image index - we need to be able to page through a users' images in sequence. This is an ascending index that we'll maintain per-user. If end_index isn't specified, it'll return a set of thumbnails starting with the user's most recent image.

Follow the data

So you've got your system's external API. For each operation, figure out the data flow. What state does the system need to read and/or write in order to do the work? What do we know about this state?

In our example we need to store:

- original image data
- thumbnail versions of the image data
- the most recent image_index allocated for each user

Two of these are inherent to the problem, and the third is a property of the way we designed the API - we're already making design decisions.

Data consistency - critical data

Some applications require very consistent views of some critical state which if wrong or lost compromises the correctness or availability of the rest of the system.

Examples:

- What is the leader of the group of processes?
- Is a lock/lease held or not?
- Assigning sequential and globally-unique IDs
- Which processes are members of a group (for instance, the set of processes serving one shard of a datastore)

These can be tackled using services that use distributed consensus algorithms to manage state changes: run 3, 5 or more replicas in different failure domains, and as long as a majority (quorum) can communicate, they can make globally consistent updates and reads. These are expensive operations (in latency terms) and should be used only when required.

FastPaxos is an optimized consensus algorithm that uses a stable leader process to improve performance. Performance for FastPaxos is one round-trip between the leader and quorum of closest replicas to commit an update or a consistent read. As well as this, the client must communicate with the leader, so perceived latency for the client is RTT between client and leader plus RTT between the leader and nearest quorum . Stale reads require a roundtrip to the nearest healthy replica, but the data may not be consistent (that's useful, sometimes). You can do independent transactions on unrelated data; and you can batch several related modifications in one transaction.

In our example, the last allocated `image_index` for each user is critical metadata, because if lost we don't know how to access the user's photos. It's also a small set of data. It's appropriate to use a distributed consensus backed datastore for this kind of data.

Data consistency - less critical data

Often, the bulk of a dataset is valuable but less critical, and using these highly-replicated and consistent datastores is overkill. In this case consider replication and/or backups for safeguarding data, depending on SLO required and budget available. Replicating data means keeping more than one copy and is helpful for availability in the face of failure of storage system or network, for reducing latency when clients are in a variety of locations, and in order to reduce the likelihood of data loss.

In the image store example, the original image data is user data, and we should be careful not to lose it. However, losing one image does not compromise the rest of the system. This implies storing multiple copies of the data.

The thumbnails are generated data and can be regenerated from the originals if lost. Losing thumbnails might compromise our ability to serve requests until it can be regenerated. We should store as many copies as required to meet SLOs.

Data storage considerations

There are a lot of things to think about when storing data. One of the most fundamental is how to organise the data in RAM and disk (SSDs are also obviously useful, but for simplicity we're not using them in this exercise).

RAM is smaller and fast, it's good for caching frequently accessed data. It might make sense to keep most of your data on disk, but to keep an index of locations of your data in RAM.

Think about how data is read and written. Which is the most common case? Usually, it is reading. In the case of the images, they're relatively large - it probably makes sense to simply infer a file path from the uuid and image_index, and store and read it from there. The OS can do the heavy lifting of caching frequently accessed data.

In the case of the image thumbnails, what makes sense is probably to store them as they're accessed: in chronological order, and by user. Create files per-user, and concatenate thumbnail data there for many thumbnails, with some data about where each image begins/ends; impose maximum sizes on the files and rotate to new ones when the maximum size is exceeded. Keep a sparse index of the offsets in RAM to speed up access to particular ranges of `image_index` per user (a hashmap of the filename and offset for every nth image and seek forward and back). This means that thumbnail requests can be served without needing individual reads for each thumbnail image.

Sharding data

Datasets are often too large to be stored on one machine. They should be sharded, by some property of the key that is used to access the data. In our imageserver example, we could use the `uuid` for the user to shard the data, or a combination of the `uuid` and the `image_index`. Using the uuid is simple - but heavy users could cause some shards to be heavier (more data, more frequently accessed) than others, and it limits the maximum number of images stored for one user to the capacity of a shard. Be aware of this limitation before making this design choice.

Sometimes keys are not evenly distributed. It's always best to shard based on a hash of the key you are using.

The simplest way of sharding is this:

- n static shards on n server n
- shard = hash(key) modulo n
- shard n is on server n

This is bad. Resharding is terrible, shards are large, you can't do anything about a very busy shard.

A better way of sharding is this:

- m shards on n servers, m is at least 100x n
- shard = hash(key) modulo m
- we have a map of {shard -> server} that we look up to find the server
- the system can dynamically move shards and update the map in order to rebalance, as shards are relatively small (this does require care and is important state - strong consistency)

Many other more complex sharding schemes are possible.

Sharded and replicated datastores should have automatic mechanisms to cross-check state with other replicas and load lost state from peers - this is useful after downtime, or when new replicas are added. This needs to be rate-limited somehow to avoid thundering-herd problems where many replicas are out of sync.

Scaling and Performance

Do some back of the envelope calculations to estimate the potential performance of the system, any scaling limits created by bottlenecks in the system, and how many machines are likely to be needed to service the given workload.

These are the type of calculations to use:

- Disk - sanity check the volume of data read/written, disk seeks over time

- RAM - how much can we cache? Are we storing an index of some kind?
- Bandwidth - size of requests inwards and outwards - do we have enough bandwidth? Bandwidth between datacenters and bandwidth between machines in the same DC are both interesting
- CPU - consider this if it's a computationally-intensive service - hard to gauge
- What are the estimated concurrent transactions for the service (compute based on throughput per second and latency - e.g. 500 requests per second, 200ms average latency -> 500/(1000/200) -> 100 transactions in flight) - is this a reasonable number? Each requests needs some RAM to manage state and some CPU to process

You should have a handout called '[Latency numbers every programmer should know]' - this has some sample calculations.

Consider the set of operations for each request and add together the expected minimum latency for each operation: has this got a chance of meeting the SLO?

These calculations are a sanity check. Use them to validate a design, but load testing is still a necessity. These back-of-the-envelope calculations ignore traffic peaks and queuing theory, for instance.

Iterating and refining

You've now got a complete design. Well done! This is a great time to take another look. You now have a much deeper understanding of the problem and may well have unearthed new constraints or problems that you didn't initially consider.

Look at your solution:

- Does it work correctly?
- Are you meeting your SLOs and constraints?
- Is it as simple as it could be?
- Is it reliable enough?
- Is it making efficient use of hardware?
- What assumptions did you make and are there risks?

Are there places where you can improve by adding caching, indexing, lookup filters like Bloom filters? Can you degrade gracefully when overloaded by doing cheaper operations (return 10 image thumbnails instead of 20, for instance)?

If you're not happy with any aspect then refine and iterate.

Monitoring

Monitoring should be included in the design phase of any distributed system. Your monitoring should:

- detect outages (and alert)
- provide information on whether you're meeting your SLOs
- give indications of growth and system capacity to meet future needs
- assist in troubleshooting outages and less-critical problems

Design metrics for your system and indicate how you will collect data. What are your alert thresholds/conditions?

What could go wrong?

Answer these questions:

- What is the scariest thing that can happen to your system?
- How does your system defend against it?
- Will it continue to meet its SLO? How can you recover, if not?
- Can improvements be made to the system to prevent the problem, and what would they cost?

Ideas:

- Loss of a datacenter
- Loss of important state/data
- Bugs in a critical software component
- Load spikes
- Queries/requests of death which can crash your processes
- 10x growth in workload (and available computing resources) - will it scale?
- Change in data access patterns changes cache hit rate