Lecture 8 - Topics in Distributed systems

CAP theorem

- It stands for consistency - availability - partition tolerance, pick two of the three properties
- A couple of examples to illustrate cap theorem. One of the key aspect to understand about a distributed system is that it's distributed, aka machines are connected through some sort of network that are prone to failure and therefore "partitions" may form from time to time. Partitions form when connections between either machines or clusters of machines are severed, and you can imagine islands of machines that cannot communicate with each other
- You are actually fairly familiar with problems this will cause. Imagine you are working on code with a bunch of other people. You don't always communicate, and within your own mind, you have a consistent view of the whole code base and there aren't (or at least shouldn't) be any conflicts.
- Now imagine collaborating with someone else through git, and they have their own version of the main/master branch. As soon as you establish contact, there will be conflicts that result from the interaction, and reconciling the differences will be the main task. To a certain extent, automatic merging is possible, but oftentimes both of you will make conflicting changes to the same files and the only resolution is establishing a shared goal and resolving the conflicts together.
- So in this case, when someone asks you about what's the latest state of a file on master, you may have a different answer than another engineer working on the same file. Therefore, if the answer is different, then the system is inconsistent.
- However, if you resolve this inconsistency by just telling the person to wait or refuse to tell the person an answer, then you are unavailable.
- Notice that you can either tell someone an answer or not tell that person an answer, with partition, there are only two choices you can make, thereby making the trade off between having a consistent view of the world or answer when asked about the world.
- Note the trivial example of having no partition, aka you and your friend is sitting side by side, and every update made by you is immediately mentioned to your friend and vice versa. If a third party asks either one of you, both of you will have a consistent view of the world and can tell them about what you know immediately.
- How does it apply to software engineering and distributed systems? Typically when encountering scaling issues, most people tend to scale horizontally. That is, getting more servers to do the work. However, because all the computers are connected through a network, a lot of failures can happen which will result in clusters of servers not able to communicate with each other. In this particular case, we are mostly concerned about one particular type of consistency model called read after write. Aka, when something is written, then all readers on all available nodes should be able to read what was written immediately.

Load balancing - Consistent hashing

- Imagine you have n cache servers, and you wish to balance requests coming in to the n servers. A common way to do so is hash(key) % n, where n is the number of servers you have available.
- In order to fetch the key from the right server, you would have to locate the server through the operation above.
- This approach works well when the number of servers is fixed, however in real life the number of servers are rarely fixed. What if you get more traffic than expected and need to scale up? What happens if a server breaks and falls out of rotation?
- In this case, when clients try to hit the cache server, they will be met with a bunch of cache misses (since now the server is different), and if you modulo each number by 3 vs 4, the resulting number will likely be on a different server.
- Therefore, you will be hit with a wave of cache misses and the result is very much undesirable.
- Consistent hashing will help you resolve this issue. Key to understanding consistent hashing is to arrange your servers on a ring based on the hashing space. For example, if the hash algorithm you use maps input to an output from x0 to xn (SHA-1 goes from 0 to 2^160-1), you tie x0 and xn together, thereby forming a ring
- You then place your servers evenly around the ring at specific points, now whenever an input is hashed, it'll be placed somewhere along the ring. To see which server it should be taken from, we go from clockwise from the hashed string to a server and place the key into that server.
- Now the key magic happens when a server is added or removed, you follow the same process as above to find the new server for a subset of the keys, then only the impacted keys will be moved.

Bloom Filters

- It's a very interesting data structure to answer the question, is an element in the set, where there are two properties:
    - If the answer is no, then it definitely does not exist in the set
    - If the answer is maybe, then it has an quantifiable rate of false positives
- It's also an in-memory data structure that's very space efficient, where for each item, we store k bits of data (where k is configurable based on different properties of the data structure).
- You might be thinking, that's a random collection of properties for a data structure, why is it useful? In general, it's useful when you can save a bunch of time or work if something is not in the set, and if you know something is in the set, then you can do further processing as needed.
- For example, a famous example is Akamai, a CDN provider, which uses bloom filters to avoid saving "one hit wonder" links. They discovered 70%+ of the link is only ever visited once, and therefore not worth caching in CDN, and only if a url is hit a second time then it will try to cache it. In this case, it can use a bloom filter as an approximation, if the filter

returns a no, then it can safely ignore the link until the next time when it returns a maybe. In this case, a false positive is not super impactful, because it's slightly wasted resource to store an extra page.

- Another example, it's used in some database to get a quick answer for whether a particular row or column is in the database before making a round trip to the harddrive itself to search through
- Final example, Chrome used to use a bloom filter to check URLs that are potentially malicious, saving a round trip to the server to double check in the case of no. Phased out later due to the growth in the number of urls and frequency of updates.
- So how does this work? Basically, it relies on hash algorithms, and each hash algorithm hashes to a number. Depends on how many items you wish to store in the bloom filter and % of false positives there are, you pick the number of hash algorithms you would like to use and how big to make the bloom filter. There are k hash algorithms you use, and hash the word k times with each algorithm, then you would mark the necessary field in the array with 1. When looking up, you hash the target word k times, then see if all of those fields light up to decide the result.
- https://llimllib.github.io/bloomfilter-tutorial/ -> good example