DISTRIBUTED SYSTEMS PRIMER

I. Challenges, Goals, and Approaches

Distribution is hard for many reasons (facts of life). Distributed systems (DS) aim to provide the core mechanisms and protocols that address the challenges and hide them under convenient, easier to use abstractions that others can use. Unfortunately, not all challenges can be hidden under clever abstractions, and they creep up whenever one pushes a distributed system to its limits. So anyone wishing to develop or even use distributed systems must understand these fundamental challenges and general approaches to address them when they creep up.

Below are a few **facts of life** that make distribution hard, the corresponding **goals** of distributed system design, and the main **approaches** that distributed systems take to address them.

FACT OF LIFE 1: Data is big. Users are many. Requests are even more.
No single machine can store or process all data efficiently. Supercomputers can do a lot, but they haven't been the final answer to scaling for a long time. The primary goal of distributed systems is to enable distribution, i.e., to *make multiple independent machines interconnected through a network coordinate in a coherent way to achieve a common goal (e.g., efficiently process a lot of data, store it, or serve it to a lot of users).* The preceding sentence is, btw, an accepted definition of a distributed system.

However, effective processing at scale is hard. An arbitrarily application may simply not scale:
- coordination is expensive (networks are expensive).
- the application may not exhibit sufficient parallelism.
- bottlenecks may inhibit parallelism. Sometimes bottlenecks hide in the very low levels if those are not used correctly (e.g., a network hub, a logging server, a database, a coordinator, etc.).

Goal 1: Scalability. Effective coordination at scale. The more resource you add, the more data you should be able to store/process, and the more users you can serve. This implies programming models and abstractions that are known to scale. Examples that you'll learn about: the map/reduce model, RDDs, etc. These are all examples of programming models that make an application scalable. But, never hope for perfect scalability: add one machine, increase your capacity proportionally forever. Most often, the scalability curve tapers off as various components of the system start to reach their capacity. Sometimes these can be very hidden components (e.g., a monitoring system, a network router).

Approach 1: Sharding. The primary mechanism for scalability is called *sharding*: launch multiple processes (a.k.a., *workers*), split up your load into pieces (a.k.a., *shards*), and assign different shards to different workers. For example, you can split your dataset into pieces, split your user base into subsets of users, or distribute your incoming requests to different workers. The workers should be designed to coordinate to achieve a common, coherent service despite the sharding (e.g., compute a global statistic over the dataset, perform each user's actions in a consistent way with respect to the other users, etc.). We'll see that sharding raises substantial semantic challenges (called *consistency* challenges), especially in the context of failures, as next described.

FACT OF LIFE 2: At scale, failures are inevitable.
Many types of failures exist at all levels of a system:
- network failures
- machine failures (software, hardware, flipped bits in memory, overheating, etc.)
- datacenter failures
- software failures
… other…

They are of many types: some are small and isolated others are major failures, some are persistent others are temporary, some resolve themselves others require human intervention, some result in crashes others result in small, detectable corruptions. What they all have in common: most failures are very unpredictable! They can occur at any time, and at scale they are guaranteed ALL THE TIME! And they greatly challenge coordination between the machines of a distributed systems (e.g., a machine tells another machine to do something but it doesn't know if it's done it, how can it proceed?)! Or, imagine that two machines need to coordinate (e.g., to compute a global statistic over a sharded dataset) but they cannot talk to each other. What are they supposed to do? Can they go on with their processing and make progress only among those processes that are up and running? When is it OK to do that? For example, if the statistic we're computing over a sharded dataset is a rough average, then it may be OK to report the average over n-1 workers if one of the workers is down. However, if the statistic needs to be exact and is sensitive to the data (e.g., we need an exact maximum/minimum), then n-1 live workers cannot continue until the n-th comes back with its own value.

Goal 2: Fault tolerance. The goal is to hide as much as of the failures as possible and provide a service that e.g., finishes the computation fast despite failures, stores some data reliably despite failures, provides its users with continued and meaningful service despite failures. Coordination needs to take failures into account and recover from them.

Approach 2: Replication. The primary mechanism used for fault tolerance is *replication*: have multiple replicas execute/store the same shard; if one replica dies, another replica can provide the data/computation. For example, if you're computing a maximum over a sharded dataset across multiple workers, have the maximum over each shard be computed by two or three replicas in parallel; if one replica dies, another one can report the maximum to the other n-1 worker sets.

FACT 3: Consistency in sharded and replicated systems is HARD.
Both sharding and replication raise substantial consistency/semantics challenges in the context of failures. Consider the case of computing an exact average over a sharded and replicated dataset: how do we make sure that we incorporate the average over each shard only once if the average over each shard is computed and reported by three replicas? Assigning each shard a unique ID may help address this particular problem, but the challenge can become a lot harder if the faulty replica does not die, but instead spews up a faulty average value due to, e.g., a memory error.

Goal 3: Meaningful consistency semantics. Dealing with these issues is hard for both the programmers who build distributed applications and the users who use these applications. So the key thing is to build infrastructure systems (such as storage systems, computation frameworks,

etc.) that provide clear semantics that hold in the face of failures, and to express those semantics clearly in the APIs of these systems. E.g., if you decide that in case of failure your distributed computation system will return the results of a partial computation, then you need to communicate that through your API so the programmer/user of the results is aware of the situation. You may also want to provide error bounds for the results you are reporting.

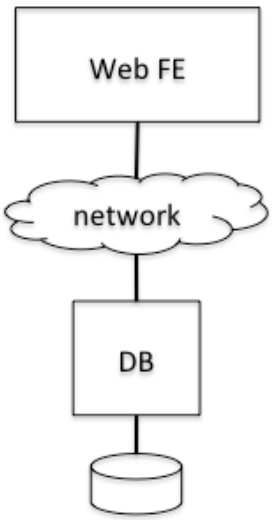
Approach 3: Rigorous protocols, such as agreement protocols. The general approach is to develop rigorous protocols, which we will generically call here *agreement protocols*, that allow workers and replicas to coordinate in a consistent way despite failures. Agreement protocols often rely on the notion of *majorities*: as long as a majority agrees on a value, the idea is that it can be safe to continue making that action. Different protocols exist for different consistency challenges, and often the protocols can be composed to address bigger, more realistic challenges.

In this module, we will look at two types of protocols that address consistency challenges in distributed databases: (1) commitment protocols, for implementing transactional semantics in a sharded database; and (2) consensus protocols, for implementing consistent updates in a replicated database. After we look at these separately, we will look at how one particular distributed database, Google's Spanner, combines the two mechanisms to build a strongly consistent database of global scale. At the module's end, you should be able to understand some pretty complex design of distributed systems!

II. Example: Web Service Architecture

Before looking at real designs, let's understand the challenges a bit more with a simple example: we'll build up a web service architecture! We'll start with a basic architecture that's not scalable or fault tolerant (or particularly efficient) and we'll add DS components to make it more so. Unfortunately, every time we add a new component to solve a problem, we'll see that we'll open up a bunch more problems...

1. Basic Architecture



Architecture:
- web front end (FE): creates and serves pages in response to users' requests; accesses the database to get the data necessary to populate the response pages.
- database server (DB) with disk attached: stores all the data (user data, even session data).
- network that connects web server with database server.

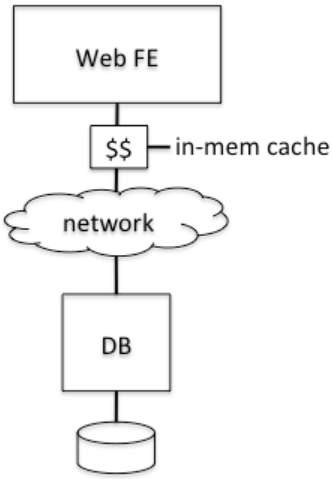
Advantages:
- FE is stateless and can restart on failures, all data is in DB, which gives good durability and consistency properties.

Problems:

1) Latency: network, DB (disk, transactions, contention)
2) Throughput: limited by DB most likely
3) Fault tolerant: not very. DB is not replicated => single point of failure for both availability and durability.
4) Scalable: not very. 1 FE, 1 DB.

Let's deal with problem 1).

2. Goal: Reduce Latency



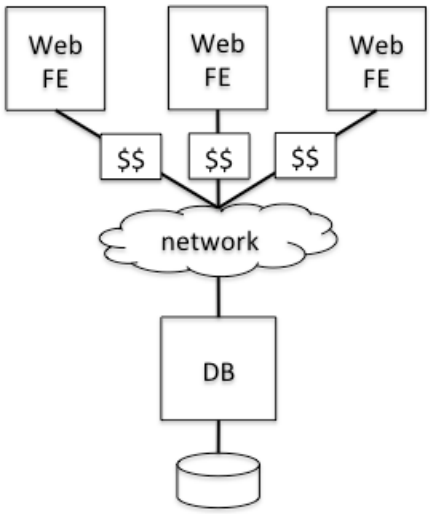
Architecture: Web FE's reads/writes go through in-memory cache ($$). The $$ saves the latest values of written data and responds with them when the FE asks for something in the cache.

Properties:

- Read performance: improved if working set fits in memory. If not, read performance doesn't improve (may even degrade).
- Durability: depends on cache: write-through vs. write-back:
  - Write-through cache = writes go through the cache, where they are saved for future reads, but then they go all the way to the DB. The cache waits for the DB to ack the write before it returns to FE.
  - Write-back cache = writes go through the cache, where they are buffered. The cache sends them asynchronously to the database from time to time.
  - Durability is good with write-through $$, poor with write-back $$.
- Write performance is opposite: good with write-back cache, poor with write-through cache.
- Are there any consistency issues? No. Only one server accesses DB, and it goes through one $$.

Problems 3), 4) from above still exist. Let's deal with part of 4) next.

3. Goal: Scale Out the FE (and get some fault tolerance for it)



Architecture: Launch multiple replicas of the front end. Each has its own local cache, which we'll assume is write-through.

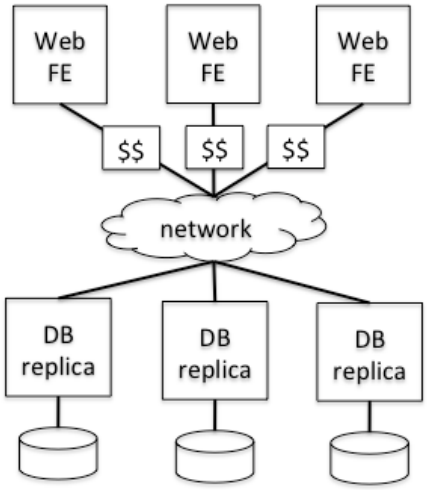
Benefit: If an FE dies, another one can take over. If the caches are write-through, there are no recovery issues. So this architecture gives scalability and fault tolerance for the FE part.

Problems:

- Consistency now becomes a problem. Entries in the caches can become stale. To address this, we need to keep the caches in sync, or invalidate them whenever a write occurs on a cached entry. Writes (or invalidation messages) thus need to propagate to the DB plus all the caches.

- So, write performance can be affected.
- We also need some concurrency control on the DB now because multiple FEs can update the DB at the same time. E.g., the DB server needs to lock entries while updating them.

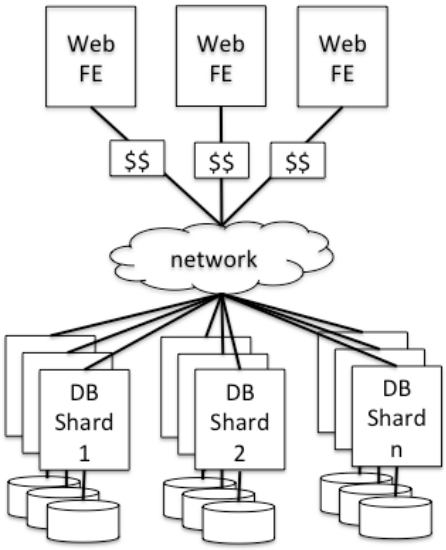
4. Goal: Fault Tolerance for DB



Architecture: Launch identical replicas of the DB server, each with its disk. All replicas hold all data, writes go to all.

Problems:
- Writes now need to propagate to all replicas. So they are much slower! Even if done in parallel, because FE now needs to wait for the slowest of DB replicas to commit (assuming write-through cache, which offers the best durability).
- Moreover all replicas need to see all writes IN THE SAME ORDER! If order is exchanged, they can quickly go "out of sync"! So lots more consistency issues.
- There are also availability issues. If you require all the replicas to be available when a write is satisfied (for durability), availability goes DOWN! Consensus protocols, which work on a majority of replicas, address this.
- Another consistency challenge: how are reads handled? If you read from one replica, which one should you read given that updates to the item you're interested in might be in flight as you attempt to read it? We'll see how to address these issues in future lectures by structuring the replica set in particular ways.

5. Goal: Scalability for DB



Architecture: Partition the database into multiple shards, replicate each multiple times for fault tolerance. Requests for different shards go to different replica groups.

Problems:
- How should data be shared? Based on users, based on some property of the data?
- How should different partitions get assigned to replica groups? How do clients know which servers serve/store which shards?
- If the FE wants to write/read multiple entries in the DB, how can it do that atomically if they span multiple shards? If different replica groups need to coordinate to implement atomic updates across shards, won't that hinder scalability?

III. CONCLUSION

Scalability, fault tolerance, and consistency are difficult goals to achieve. Solving them requires rigorous protocols and architectures. We'll learn about basic protocols and architectures through the course of this module.