# Designing Data-Intensive Applications

## Chapter 8: The Trouble with Distributed Systems

**Chapter Introduction**

- In this chapter looking on the pessimistic side of things
- Assuming anything that can go wrong will go wrong
- Working with distributed systems there are even more modes of failure

**Faults and Partial Faults**

- In single-system applications with proper code an application either works or it doesn't
    - If hardware fails the system will fully crash
- With distributed systems things can be a bit more gray
    - We know individual systems will fail or groups of systems will fail but we need the overall system to continue working
    - Sometimes it is not obvious exactly which systems have failed, maybe there has only been a partial failure, one piece works but others don't or they are unreliable

**Cloud Computing and Supercomputing**

- High-performance computers (Super-computers) operate closely networked and integrated and function more closely to a single computer
    - Will typically checkpoint work and on failure will simply stop
    - Once systems are repaired then the checkpoint will be resumed and computing will continue
- Cloud computing is more loosely networked systems that can be distributed across many different kinds of systems and data centers and countries
    - Typical of global web applications, not acceptable to stop the whole system for repair
    - Requires low latency
- The bigger a system is the more parts there are to break. With a sufficiently large system it is safe to assume something is always broken
- Must assume partial failures will occur and build in fault tolerance to the application

**Unreliable Networks**

- Networks are slow and unreliable and it is hard to share significant amounts of information
    - Have to build things with shared-nothing in mind because of this limitation
- Most networks are asynchronous packet networks. There is no guarantee on the transmission of information
    - Requests may be lost
    - Request might just be delayed because of other traffic
    - Receiver of request may have died
    - Receiver may just be temporarily unresponsive
    - Request could have been received but response got lost
    - Request received and response sent but delayed
- Impossible to tell exactly why no response
    - Possible to use a timeout to determine what is a reasonable amount of time before retrying or considering something has failed

**Network Faults in Practice**

- Network hardware has become increasingly reliable but human error is always present
- Even EC2 and the best run datacenters in the world still have hiccups
- If your application allows it it is perfectly fine to simply show errors messages in case of temporary outage. Dont always need to be completely fault tolerant

**Detecting Faults**

- Need to be able to detect failed systems or else they will accumulate and requests will continue to go to downed systems and slow down overall network and increase latency to actual responses
    - Load balancer needs to remove downed nodes
    - In a single leader system if the leader goes down need to be able to assign a new one
- If the system is still up but application has run into a problem then possible to get a return from the system saying it is present but non-operational
    - No way of knowing what requests it may have received and left unanswered or what data was sent to it and lost
    - Possible for OS to directly announce it has failed so other systems dont have to wait for timeout
- Possible to use remote management tools to check on systems if you are closely networked

**Timeouts and Unbounded Delays**

- It is difficult to determine the correct timeout duration
    - Don’t want to prematurely timeout systems and pronounce them dead simply because of slowdowns
    - Don’t want systems waiting excessive amounts of time for downed nodes
- We have unbounded delays. There is no guarantee for timing of information
- Often delays are caused by queuing of some sort
    - Multiple systems trying to send info to the same system, all have to be handled in whatever order they are received, but some will have to wait
    - If all CPU cores of a system are busy it will have to wait for cycles to address the information
    - In virtual systems they can be temporarily frozen while other VMs are doing things
- Possible to have a noisy neighbor that is causing delays for you by using up networking capacity
- Have to determine appropriate timeout lengths experimentally

**Synchronous Versus Asynchronous Networks**

- Systems like a fixed phone line work synchronously
    - Fixed and constant amount of traffic delivers just the right amount of bandwidth for communication and guarantees a certain latency
    - Delay is bounded
- Would be nice to have these guarantees in our networking but there are also pitfalls
- Ethernet and IP are packet-switched protocols, require queuing and this leads to unbounded delays
    - These are better for bursty and flexible traffic
    - With QoS and admission control it is possible to get near bounded delays

**Unreliable Clocks**

- Timing is exceptionally important to distributed applications for several reasons
    - Checking if a request has timed out
    - 99th percentile response time
    - How many queries have been received in last 5 minutes
    - Etc.
- Sometimes it is important to measure duration of things (5 seconds since x happened), but others the absolute point in time (Saturday at 12) is what matters
- Time is tricky in distributed systems because likely not everyone agrees
- Each system likely has quartz crystal oscillator to keep track of time, but not particularly accurate
- Network time protocol (NTP) syncs with a sort of time authority to try to get agreement, but still subject to latency and other issues

**Monotonic Versus Time-of-Day Clocks**

- Time of day clocks/wall clock time - aligned to be the amount of time since epoch to roughly tell the date/time
    - Can be resynced so not good for relative tasks. Might appear to be time travelling
- Monotonic clocks - basically constant counters that tell how much time has elapsed
    - Good for keeping track of ordering of things
    - Better for measuring duration of things, but still imperfect

**Clock Synchronization and Accuracy**

- Quartz crystal based timekeeping can drift as much as 17 seconds a day
- With sufficient drift syncing may be refused or a reset may be forced causing a jump in time
- NTP syncronization only as good as network delay
- Possible for NTP to be wrong or slightly off
- Virtual machines may essentially be frozen and have jumps in time
- People might intentionally change computer clocks

**Relying on Synchronize Clocks**

- Sometimes small discrepancies in timing can be very important
- Hard to tell desync in time because not usually a catastrophic failure until it is
- Useful to timestamps events to make sure requests happen in order, but if time is not agreed upon it can cause lots of issues
- In last write win scenario the last write might not actually be the last write and newer information can be overwritten or corrupted
- Timekeeping cannot be easily made accurate enough to make this go away
- Clock readings can have confidence intervals which is good for determining if a request truly precedes another or if there is some uncertainty about which came first
- Might use time to backup all information until a certain point in time, but information can be lost if systems disagree on what time it actually is

**Process Pauses**

- Possible to use leases to assure who the leader is
    - Leader must periodically renew lease to remain leader
    - System might pause for a bit and miss lease renewal, but still assume it is leader during that period and perform problematic writes

- Possible to create systems with real time guarantees but much more expensive to develop and requires specific operating systems and programming languages
- Long garbage collection calls can cause considerable pauses

**Knowledge, Truth, and Lies**
- Difficult to determine what is reality based on senses that might be faulty

**The Truth is Defined by the Majority**
- In distributed systems some things need to be agreed upon. A majority or some other threshold is required in order to make certain actions
- One node might think it is still leader but all other nodes disagree and have moved to a new leader
- In order to alleviate this can use something like lease or lock but timing can still cause issues
- Leader might check out lease, pause for a bit, not renew lease and other system takes over, then believe they are still the leader
- Good to use a fencing token, incrementing counter that blocks old leader from trying to write when it is using an old token

**Byzantine Faults**
- Not always required to build this kind of tolerance in, but good to think about
- Might have bad actors in your midst and need to use majority and agreement to fight against this
- Systems might accidentally lie rather than intentionally sending bad information

**System Model and Reality**
- Models
  - Synchronous - bounded network delay and time drift
  - Partially synchronous - sometimes asynchronous properties when network delays and time drift occur
  - Asynchronous - no timing assumptions
- Faults
  - Crash-stop - when node faults it just permanently stops
  - Crash-recovery - may fault and appear dead, but may recover at a later time
  - Byzantine - nodes may do anything, even intentionally trick and deceive
- Typically partially synchronous and crash recovery is where most systems fall on the model/fault matrix

Lots of good theory and research but many things can happen in the messy world and have to be prepared for any number of different scenarios