CS 188 Notes

Lecture Notes

4/1 - Week 1

- Distributed system is where you have a bunch of connected computers that cooperate and communicate with each other to provide some service. The main goal is to unify several machines to provide shared service.
    - One application should be able to run on the different machines.
- Distributed systems are helpful because they allow for services and applications that are resistant to failure (building reliable systems with lots of unreliable components), have low latency, handle geographic separation of the clients across the world, and can handle a large load. You are also able to customize computers for specific tasks called microservices.
    - To achieve these things, you need to have distributed databases to store redundant data, a load balancer, need more machines (preferably in different geographic places)
- Geo-distributed services refer to having servers or systems that are spread out across the world in attempts to have low latencies for people in different parts of the world.
- Data centers spread services and data storage and processing across thousands of machines.
    - Cooling is a huge concern with these data centers in order for them to run properly.
    - When you have multiple data centers you need to figure out how to communicate between them and what information each data center has.
- Challenges with distributed systems
    - Partial failures - Because one machine in the system fails, you might not be able to get your work done.
        - Software should be able to handle these failures but you can also do it irl through just swapping disks or getting new ones.
    - Ambiguous failures - If a server doesn't reply, hard to tell whether the server or network or both or something else failed or the communication is just being slow.
    - Concurrency - If a bunch of users use your application concurrently. Solution is to partition machines to different users (M1 is assigned to U1 - U10).
        - Problem with that is if the application needs a shared state across the different machines aka your servers need to talk to each other about the actions that the users that are assigned to them are taking.
        - The basic goal is to ensure the consistency of distributed state in the face of concurrent operations.

4/3 - Week 1

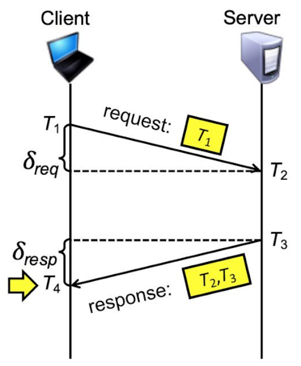
- MapReduce is a distributed data processing framework. Basically, when you have a very large amount of data that needs to be processed and that processing cannot be done on one computer and therefore you need a bunch of machines that are connected.
- Common theme across companies that use things like MapReduce is that they have petabytes and soon exabytes of data.
    - With that scale of data we can't store it on one server and access and processing of that data would take forever.
    - Solution: Distributed storage and processing
- Desirable properties for distributed processing
    - Scale - Performance should grow as the number of machines increase.
    - Fault tolerance - Be able to make progress on the task if one machine in your set of machines fails since as the number of those machines grow, the likelihood of a failure of at least one machine gets larger as well.
    - Simple - Programmers should be able to use the framework as quickly as possible.
    - Expressive - Programmers should be able to do a wide range of tasks.
    - Flexible - Minimal restrictions and assumptions on the type of processing that the system will be used for.
- In distributed processing, it is almost always the case that each machine will not have all of the possible data and thus we need to figure out how we can have that machine still be able to do useful work even with a subset of the data.
    - Sometimes this is difficult because computation will need to have a look at the whole dataset rather than just a subset. This is due to the inter-data dependencies.
- Hadoop is a framework and it uses MapReduce which is a distributed data processing paradigm.
- MapReduce represents both:
    - Programming interface for users to be able to define the map and reduce functions.
    - Distributed execution framework that takes care of actually running the functions across the machines that we have. It should be scalable and fault tolerant.
- MapReduce Execution
    - Data that you're going to execute on should be stored in key value pairs.
    - Then, split dataset into a bunch of partitions with some k number of key-value pairs in each of them.
    - Then want to invoke the map function (that the developer wrote) that takes the pairs as input and then outputs a set of new key-value pairs.
    - Then coalesce phase groups the pairs so that all the keys are in the same partition.
    - Finally, run the reduce function on each key and you get a single key-value pair for each key.
- Running the map and reduce functions on each partition are independent of each other.
    - Map on partition 1 can be done at the same time as partition 2.

- Reduce is able to start running only after coalesce finishes which only happens when all the partitions have gone through the map stage.
    - Synchronization barriers before starting the coalescing stage and the reduce stage.
- The problem with this barrier requirement is when a machine fails, which mean that the reduce stage cannot start until that machine finishes its work.
    - The master node is in charge of keeping track of the workers, the status of the tasks, and what to do in case of any machine failures.
        - However, if the master node fails, then all that info gets lost and so we can't really just assign master status to another node and ask it to take over. Solution is to save master state at certain times
    - In the case where you have multiple masters, each master needs to send the others info about what it is assigning, but you could run into synchronizations and timing issues and have different workers working on the same task.
        - Solution of making sure that all replicas are in sync is to make sure that each replica executes updates in the exact same order by using a state machine.
- Replicated state machine is to run copies of the same state machine across many servers.
    - Replicate over time: Start new replicate after one goes down. This is in case of a crash failure.
    - Replicate over space: Run multiple replicas at the same time. This is more preventative and fail stop.
    - These are two ways of addressing failures.
- One issue with replication is that the clocks are not in sync across machines. You can try to leverage GPS broadcasts, but this doesn't work indoors and is power hungry. You can also try NTP (Network Time Protocol) which is where you sync clock with dedicated servers around the world.
- Christians algorithm is where the client tries to sync its local clock with the server's clock.
    - • Goal: at T_4, client sets clock $\leftarrow T_3 + \delta_{resp}$

      • But the client knows δ, not δ_{resp}

      • Client measures round trip time $\delta = \delta_{req} + \delta_{resp} = (T_4 - T_1) - (T_3 - T_2)$

      • Current "solution": assume $\delta_{req} \approx \delta_{resp}$
        • Client sets clock $\leftarrow T_3 + 0.5\,\delta$



4/8 - Week 2

- The main thing that we need to ensure with the replicated state machines is to make sure that the order of operations on the different machines is preserved. All replicas apply all the updates in the exact same order. Since they all have the same starting point and they all use a finite state machine, if the operation order is the same, then they'd all have the same final state.

  - The simple approach is to order the events based on the timestamp but this is hard because of clock skew and drift, as well as synchronizing clocks. The hard part is that you don't really know the exact time of a particular task and the time for sending/receiving the message.
  - The thing, though, is we want a correct relevant ordering and we don't necessarily need the exact time, which is what the timestamp gives us. The solution is to use logical clocks. Disregard the precise clock time and look more at the relationships between events.

- From the client's POV, they don't need to know that we have multiple replicas for the master. Each of the clients will be contacting one replica. The replicas need to communicate behind the scenes to make sure that the information is consistent.
- Logical clocks work by associating each event with a logical time (think about it as an integer) that preserves a happens before relationship
  - If a happens before b, then Clock(a) < Clock(b). In a way, it's similar to the timestamps, but the numbers themselves don't have any relation to the actual wall clock time.
- Defining "happens before"
  - Single process: If a occurs before b, then a -> b
  - Multiple process: If c is the message receipt of b, then b -> c
  - Transitive properties hold.
- Lamport Clock Approach
  - The Lamport clock says that if there is a -> b, then Clock(a) < Clock(b). Each processor will have their own local clock. Each will have a counter that will have the value that the next event will have its clock value assigned.
  - When there is an event a, C(a) = 1, increment the local clock value, and then event b, C(b) = 2. This is what's happening for events on a single processor. Then, when you're sending a machine to another machine, include your local clock in that message. When the other process gets the message, the receiving event c will get the value
    - = max(local clock of the second process, clock value that got sent) + 1
  - The ordering that we have rn is a partial ordering because we cannot really order the events happening on different machines. However, we need a total ordering of updates. That total ordering preserves the idea that C(a) < C(b). There can be multiple ways of ordering and multiple final orders. The place of ambiguity happens when you're looking at events that happen in different processes. You will break ties between two values with the same clock value by looking at the process/machine identifier (MAC address, etc). It needs to be a unique ID. With this deterministic way of breaking ties, we have a guaranteed total ordering.

- One of the limitations of Lamport Clocks is that not all events are related by the what "happens before" relationship. You can't also take any two events and be able to determine which happened before.
- If a causally led to b, then Clock(a) < Clock(b)
- One of the hard parts with communicating between replicas is making the assumption that the order in which one machine sends info is the order in which another machine will process the info. So this is the problem of one message getting delayed and another later operation not getting delayed and the other machine receiving that message first.
- There is another issue in that all messages from one machine to another may get delayed.
- Inside each server you have an application and state machine component. The RSMs of each server will talk to each other to determine the correct order of updates that should happen and then it will pass on those ordered updates to the application component which will display/apply that correct order of operations and reflect the right changes to the all the users, which could be talking to different servers.
- To reduce the time between receiving a message from another machine and applying the update is to:
    - Periodically ping to the other nodes to sync clock
    - Tolerate temporary consistency by applying the update immediately and then rolling back if needed.
- Vector clocks give you a way to label each event with a vector where each component will be a clock value associated with one of the machines/processes in the system.
    - Each element is a count of the events in that process that causally precede the element.
- All vectors start as a zero vector. 2 rules for updating a vector
    - For each local event on the process, increment the local entry at that process index by 1.
    - If another process receives a message with a vector D, then
        - For each index, set the local to max(current entry at the index, entry of vector D at the index).
        - Increment the local entry at that process index by 1.
- When you compare the vector clocks of two events a and b, V(a) < V(b) when the entry for a for all indices <= value for b for all indices and they are not just equal vectors.
- When you're using vector clocks then C(a) < C(b) if and only if a -> b. The happens before and clock value being less are valid in both directions.
- Two events are concurrent when for some index, a's value < b's value and for another index, b's value < a's value
- V(a) < V(z) if and only if there is a chain of events linked by -> between a and z.

<u>**4/10 - Week 2**</u>

- With RSMs and logical clocks, any replica can execute an update only after confirming the clock on all other replicas is higher than the update's clock. There is a problem that you need to hear from all of the workers.
  - What happens if one of the workers die? How do you know if they died and it's not just a slow network.
- There are two types of failures
  - Crash failures - When a node dies, resume by taking the saved state as a starting state.
  - Fail stop - All state is lost upon failure. Cannot recover from this.
- In order to have those aforementioned saved states, we need to have checkpoints so that we can save different versions of the state, and then when a node fails, it can look to the newest checkpoint.
- The checkpoint should get stored in persistent (over volatile) memory and probably in a remote (over local) machine. However, all 4 configurations are valid.
  - Pros for storing remotely is that any local crash in your code won't affect anything. Also, it is protected from any external danger on your local machine.
  - Where and how to store this stuff depends on tradeoffs with the type of failures you're dealing with and the performance overheads.
- When one node rolls back to one checkpoint, then you might have to roll back the other nodes to that checkpoint so that there is no inconsistent state.
  - Problem is that each node might have different times at which they did their checkpoint. This can create uncoordinated checkpointing across processes.
- The solution would be to use coordinated checkpointing with is the Chandy Lamport snapshot which is a global snapshot of a distributed system. So, instead of having the nodes individually do checkpoints at random times, take a snapshot of the entire system which would have info on all the nodes.
  - It has to be a snapshot of all the nodes at a single time. Hard to do since you cannot really tell each node to take a pic at a certain time because of clock differences between nodes.
- Chandy Lamport snapshot is way to recover from failures as well as for deadlock detection, garbage collection, etc.
- OS's often use a token ring which rotates a ring across the different processes and when a process has the token, it can do some critical operation like a save a checkpoint. It's used in OS to avoid deadlocks.
  - If you're taking snapshots of the system, the problem is that you're still getting an uneven cut of the space-time diagram.
  - That uneven cut causes the problem below.
- The global snapshot must capture the "happens before" rule. If event b is in the snapshot and a -> b, then event a must also be in the snapshot.
  - If the receiving message event is in the snapshot and the sender message is not, then the snapshot isn't consistent.
  - If the sending message event is in the snapshot and the receiving message is not, then the snapshot still is consistent.

- Goals of the global snapshot is to capture the current state of every process and capture the messages in transit. The distributed system's state encompasses the state of the processes and the state of every communication channel.
- Chandy Lamport snapshot algo
    - Let's say N processes in the system. There is unidirectional channel between each pair of processes.
        - The processes won't fail while the action of taking a snapshot.
        - All channels have FIFO delivery and network is lossless.
    - Let's say you have 2 processes. P1 sends m1 to P2 and P2 sends m2 to P1. The idea is that when one process wants to take a checkpoint, it will take its local snapshot, and then it will send a message to all the other processes telling them to take snapshots as well.
        - Let's say P1 starts the snapshot, sends message to P2 to take checkpoint. P2 is guaranteed to get that message before any other message from P1 that takes place after P1 sends snapshot command to P2 (b/c FIFO characteristic).
    - The snapshot command is a broadcasted marker message to all other processes. It is a distinct message that is different from the application's messages.
        - Whenever process receives this, it knows to take a snapshot.
        - Serves as a barrier that makes sure that the checkpoints include all messages received before the marker, but none after the marker.
    - Then, when the process receives that marker, it will take its local snapshot, and then it will send back its snapshot back to the original (as well as to the other processes) which gives the original process a way to check/log all the messages that may have been received between the sending of the marker and the receiving. Once the marker is received, then no more logging takes place between those two processes.
    - If a process receives its first marker message from some process, record the local state and send a marker message to all other processes. If there is a duplicate marker message received, then we stop recording from that channel and we record the state of the channel as all the messages received since the marker.
- The frequency that global snapshots should be initiated with is _.
    - Important to note that the external environment does not checkpoint.
- In conjunction with global snapshots, between checkpoints you can log the sources of nondeterminism (which means you can try to determine when exactly to take the next snapshot).

<u>4/12 - Week 2</u>

- In vector clocks, when an event occurs, the process will only increment its own clock. When an event occurs and a process receives a vector from another process, it will copy all the clock values for the processes except the index for its process, which it will just increment.

4/15 - Week 3

- What we've learned: So if we have a bunch of replicated state machines, logical clocks give you a way to execute operations in the same order.
    - However, you can't handle replica failures.
- Crash failures (ones that you can recover from saved state) are handled by global snapshots (through Chandy Lamport algo) and by logging nondeterminism between checkpoints for efficiency.
- P2 will still log that M1 was in flight and was received, even though that info isn't in the snapshot. When the recovery happens, P1 won't resend the M1 message, and so when P2 goes to its checkpoint, it looks at its log of messages and sees/executes M1.
- Fail stop failures are ones where you don't have any state to recover from. All the state is lost.
    - The solution is that we need to replicate the state proactively by copying/syncing state from one machine to another.
- The simplest way is Primary Backup Replication. We have 2 replicas in the system, one that is the primary and one is the backup. The client will interact with the primary. Backup stores a copy of the primary's state.
    - The client is abstracted away from all of this. From the client's POV, it has no idea that there is some backup that stores of a copy of the primary's state.
- When the primary fails, then promote the backup, and if the backup fails, then find a new machine that can be the new backup(s).
- In order for this to work, the primary and the backup copy need to sync with each other. The servers should sync with each other prior to making anything externally visible. So basically before telling the outside world that you've done something, you need to sync first.
    - The primary and backup are allowed to be out of sync until the change is externally visible. They can be out of sync because nothing can know about the inconsistency anyway.
    - External consistency is the idea that the primary backup is in sync to the external world.
    - In other words, before the primary sends any output information to the clients, primary should sync with the backup. Primary will send message to backup, wait for the backup to respond, and then primary will send the responses to the clients.
- For the client operations that don't update any state in the primary, you don't need to have the primary consult the backup.
- If the condition that the backup is externally consistent with the primary, if the backup gets promoted, it will act in the same way that the primary does.

- During the syncs, you can send the operations that primary receives to the backup (instead of sending the whole state of the primary).
    - However, you do need to send the whole state when you have a new machine as the backup. This would actually be easier than having to send all the instructions from the past.
    - One issue with the passing of every operation is when the operations are nondeterministic. This would create the chance that the states of the primary and the backup are different. Solution is to send as much info as you can to get back the property of determinism.
- Applications normally rely on a library to keep primaries and backups in sync.
- Virtual machine monitor is above the hardware layer, and the actual VM is on top of that and it consists of the OS and applications running on it.
- Primary and backups are running in two different VMs on top of a VM monitor. Any inputs the primary receives (as well as outputs the primary produces) will get noticed by the VM monitor and the message is relayed through a logging channel to the backup for it to apply those messages.
    - The VMM will be sure to log the results of nondeterministic instructions and that info will get transferred from the primary to the backup so that the backup is able to execute those instructions and get the same output as primary did.
        - More specifically, the primary VMM sends log entries to the backup VMM and the backup VMM is in charge of replaying the log entries with the nondeterministic arguments that the primary VMM sends over.
- The inputs into the VM are incoming network packets, disk reads, keyboard/mouse events, interrupt events, and results of nondeterministic instructions. VM outputs are the outgoing network packets and disk writes.
- One issue that we have is that when the backup is executing some input that went into the primary, there could be some interrupt and the execution of that interrupt handler may affect the output causing the output of the backup to be inconsistent with the output of the primary (or there could be some state change too).
    - Solution is to have backup wait a little for the primary to produce an output and for the primary to inform it if it needs to suppress any sort of interrupts. The primary must also wait and buffer its output until acknowledgement from the backup.
    - In other words, primary gets message from client, sends it over to backup, primary gets some additional info, sends it to backup, primary gets final output, sends it to backup, backup sends the acknowledgment and then primary sends the final output back to client.

<u>4/17 - Week 3</u>

- As a review, Even though clocks are great for maintaining the order of operations, they can't handle any sort of failure, which means that you need some sort of primary backup replication. The rules for primary backups replication is to promote the backup when the primary fails, and replace the backup with a new machine if it fails. Primary should sync with the backup and be consistent with each other before changes are externally visible. When bringing a new machine as the backup, sending the whole state would be good, but if you're just keeping the backup up to date, then just forwarding the operations would be enough.

- Handling timer interrupts does not necessarily produce any output, but when the backup is doing operations sent from the primary, we have to make sure it is the same order.
    - In other words, we can't do anything with the receive the deposit request until its timer interrupt fires.
- The green arrows show the information sent from primary to backup over the logging channel from the VMs.
- On the 2nd timer interrupt, the backup will suppress the handler until the show new balance happens. Once it is sent up, then the wake up AddInterest is invoked.
- Receive deposit first is shown at the top of the backup because that is the first operation that the backup can do anything with. The first two things that were sent are just supplementary non-deterministic info that can/will be used later.
- Client needs to know which server is the primary so that it knows where to send the operations.
    - That primary server isn't really permanent, so the client needs to try to learn about the current primary. In the assignment, this is basically the client contacted the viewservice to learn about the identities of the primary and the backup.
- Viewservice will change views when Primary or Backup fails. It has to do this by sending heartbeat and ping messages to keep track of which servers are alive.
- When the primary fails, you need to make sure the backup should have been consistent with the primary before it failed, so that when the backup is promoted to primary, it is up to date. A machine that isn't a backup can't really be primary immediately since it wouldn't have any information.
- Two scenarios for failure
    - Primary applies the operation before fails before syncing
    - Primary fails before new backup is initialized.
- These are okay because the way we set up the server, the primary won't send a confirmation to the client until it makes sure that the backup is in sync with it.
- If viewservice detects a new view change, it will broadcast message to all the servers, the primary will sync with the backups, and the primary must send back an acknowledgement.
    - You could get stuck here if the primary dies during the sync and thus it will never send the ack.
- Client can cache the results from the viewservice response, but client will need to check in with viewservice if it figures out that the server it thought was the primary doesn't respond.
- Split brain scenario is when multiple servers think they are the primary. The problem arises when the connection between the primary and the viewservice goes down. A

client will try to contact it, but the viewservice will have already updated the view with a new primary.

    - Solution is to have the primary forward all the operations to the backups, and get ACKs from them. If the backup has a new view, then the backup can let the supposed primary know that it isn't the primary anymore.

4/22 - Week 4

- The primary acts as a serializer for all the client updates. Everything the client wants to do basically needs to go through the primary, and the primary needs to contact/sync with the backups before responding to the client.
- Can backups be able to serve reads from clients? On a first thought, they should right, because the state between primary and backups should be the same. It would also really help reduce the load on the primary. Two possible problems:
    - Primary’s state ahead of the backup?
        - Okay, since no external client is aware that the primary has executed the operation, backup will return the same thing the primary will return.
    - Backup’s state ahead of the primary?
        - Not okay, since primary could be still in the process of sending updates to backups, and depending on which backups server the client contacts, the client will get a different result. Basically you could have the situation where the client contacts the primary, primary forwards operation to backup, backup does the operation, another client sends a get to that backup which returns the new value, when in reality if it had contacted the primary it would have gotten the old value since the primary didn’t make the change locally since it didn’t finish the sync with all the backups.
        - Also, in the slides, a side issue is in the case of B2 getting promoted to primary before it is able to get the info from P will mean that that operation never gets applied.
- The properties of a system we want are that there is an ordering of all the writes, and when there is a read, then all future reads should return the same value, unless there is a subsequent write. Also, once a write is complete, then all future reads should reflect that value.
    - Linearizability is a property that reflects those ^ properties + single copy semantics (externally visible effects of writes and reads are equivalent to if there was a single copy) which mean the users aren’t aware that we’re using multiple machines.
        - FB News Feed doesn’t have this since you might see one post during one time on your laptop, but not see it on your phone.
        - Google Drive should have this though.
- The minute a write is complete is when the primary has made the changes and they are externally visible (which is basically the point at which any server can contact the primary and it will get returned the new value).

    - We don’t really care when the client you sent the first operation gets the acknowledgement message from the primary. The amount the primary makes the local changes on its local data structure is the moment at which the write is complete.
- There is a consistency spectrum (Eventual -> ReadAfterWrite -> Causal -> Sequential -> Linearizability)
    - Linearizability is the best property of consistency. It is pretty much the ordering of all the writes and reads/writes reflecting the correct values the second the operations are applied.
    - Eventual consistency doesn’t guarantee the linearizable properties but it’s still good and what S3 uses.
    - Causal is where the order of causally related writes must be preserved.
        - Won't see the effect of W2 without seeing the effect of W1 if W1 happens before W2. An example would be that you’ll always see the comment before the post.
        - Lazy sync means that for every operation, you don't have to block and sync, but rather you only have to do that for causally ordered stuff.
    - Sequential is similar to causal but it makes sure that there is a total ordering of all writes (whereas casual only orders the related ones). Also, if a read sees the effect of a write then all the future reads will too, which I guess is not guaranteed into causal.
    - Sequential is less consistent than linearizability because it doesn’t have the quality that once a write happens, the read will reflect it right away.
- The stronger the consistency, the easier to use in conjunction with other things (aka ease of programming).
- Weakening the consistency will allow you to get better performance. There is a tradeoff between latency and consistency.
    - FB News Feed doesn’t have great consistency, but it doesn’t matter because users would more likely want the feed to load faster.
- Instead of having a primary to serialize the operations, you could try to have the replicas perform the write with a lock. There is a problem for if the replica fails, the replica will still have the lock.
    - The solution would be to have a timeout through a lease.
    - Setting the timeout value means making sure the lease value is longer than the time expected for the operation the client wants to do, but also not too long because everyone else is blocked.
- Issues with leases are with the time validity of the locks. The clocks of the leaseholder and the service may have different views and the messages between the two can have network delays.
- Google File System is their distributed storage system.
    - The files are split into chunks, the chunk is replicated on randomly selected machines, and a central chunkmaster server (acts like the viewservice) knows where the replicas are stored.

    - Client will begin by sending instructions to the chunkmaster. The instructions will contain info on which file and chunk within the file the client wants. The chunk server will look at its data structures, determine which chunk server has the appropriate data, and contact that server. The server will return its state which contains into on where exactly that chunk is located on the server. The master will then forward that info over to the client and the client will then send a direct request to the chunkserver for its data, and the chunkserver will return it.
    - The client will send data to the closest replica first, then the primary, and then the secondary backup.

__4/24 - Week 4__

- Today will talk about limitations of primary-backup replication and Paxos, which is a solution.
- When are RSMs unavailable?
    - 1st: Nobody can really be the primary rn
    - 2nd: If backup isn't connected to the primary, then primary isn't able to sync with it and therefore not able to return the response to the client. Solution then could be to change the view. From viewservice POV, it won't really know to change the view because its connection with backup and its connection with primary are both up and so viewservice wouldn't know to change the view.
    - 3rd: Viewservice down so everything goes to shit
    - The main issue is not being able to move onto the new view.
- We want to figure out ways to keep this RSM system working in the case where replicas fail.
    - One solution could be RSM via consensus where the primary will apply the update if the majority of the replicas sync with the update.
- Another issue is with keeping the replicas in sync with each other in the case where you have concurrent new updates and all the replicas have to apply them in the same order.
- Paxos is one algorithm that is a solution.
- We want system where system doesn't reach a bad state (safety) and the system makes progress eventually (liveness).
    - To get safety, you want to only accept a value if it is accepted by a majority and it was proposed by some client.
    - To get liveness, if values are proposed, then eventually one of them should be accepted and the replicas should know that it got chosen.
- Roles of a process
    - Proposers are generally the clients and they propose values
        - Clients in the bank account example
    - Acceptors accept values and chose the value if there is a majority.
        - Replica servers in the bank account example
    - Learners learn the chosen value.
        - Also replica servers in the bank account example

    - Machines/processes could play any of the roles
- Paxos works in multiple rounds to figure out which proposal will be accepted when there is a majority.
- 3 phases in each round of Paxos
    - Prepare: Any proposer will pick a proposal number and sends to all the acceptors to figure out who is around to accept. There is no actual value being proposed rn, the proposer is just trying to figure out if there is a majority of acceptors who can accept.
    - Accept: Proposer sends a value to those aforementioned acceptors and proposer will wait for the proposal to get accepted by the majority.
    - Learn: Learners discover the value
- Every acceptor will have info about
    - Highest proposal number the acceptor is currently promised to accept (n_p)
    - Highest proposal number the acceptor has accepted so far (n_a)
    - The value that was accepted for that number ^ (v_a)
- In prepare, if the proposal number n each acceptor gets is greater than n_p, then n_p = n and reply with <promise, n, (n_a, v_a)> which is saying that the acceptor won't accept any new proposal with a number x where x < n (Means that it can accept a proposal from another proposer that sends a higher value). It's also letting the proposer know about the highest value the acceptor has accepted so far. If that original n is less then n_p, then that acceptor will reject (reply with <prepare-failed> and might also let the proposer know about its n_p so that it can create a future proposal number that is higher).
- In accept, the proposer will look at the promises from the majority of acceptors and will choose the value v associated with the highest n_a. Then, proposer will send <accept, (n, v)> to the acceptors. From the acceptor's POV, it will get a message with (n, v) and if the proposal number n each acceptor gets is greater than n_p, then you accept the proposal and notify the learners (There is a possibility that the acceptor will reject the value that the proposer sends). Then set n_a = n_p = n and v_a = v.
- One issue is that we need every proposer to make sure that it picks a unique proposal number.

<u>**4/29 - Week 5**</u>

- The major goal of Paxos is to be able to apply an update if a majority of replicas commit to it.
    - Primary backup RSM was better than logical clocks because you could deal with failures (aka if a replica is down), and Paxos based RSM is better than primary backup RSM because primary backup is dependent on viewservice as well as only a single primary. Also primary backup is unavailable until the failed replica is replaced.
- One issue is when one proposer gets an acceptor to commit to it in the prepare phase but before the proposer gets to the accept phase, that acceptor commits to another

proposer with a higher value, and thus when P1 gets to the accept phase, the acceptor will reject it since it will have committed to another proposer. That proposer will send another proposal with a higher number, and thus the acceptor will commit to it, and subsequently reject P2.
    - In other words, this is an indication to a proposer that there are other proposals currently being sent to acceptors.
    - Solutions are to:
        - When a proposal fails, back off for some random amount of time before retrying
        - Use a predetermined ordering of proposers. Higher precedence for some proposers will indicate how long they should back off for.
- Synonym for the previous problem is that between a proposer's prepare and accept phase, their n_p is updated by another proposer.
- Client is serving as a proposer and a learner while replicas each server as an acceptor and a learner.
- In the Learn phase, we want to have all the learners discover if any value was accepted by the majority.
    - The learners will send prepare messages to the acceptors to see what each acceptor accepted. If the majority of them have accepted a certain value, then you do the same update.
- In order to make sure all the replicas are applying operations in the same order, we need to have a log of updates, and Paxos makes sure the appropriate updates are placed in the appropriate slots in the log.
- Important note is that majority refers to the majority number at the beginning of the system where all the replicas are up. It does not refer to the majority at any time.
- The benefit of Paxos is that only a majority of replicas need to be up. A downside is that the client needs to deal with lots of communication overhead and the client loses the abstraction of just being able to communicate with the primary. You need at least 2 rounds of inter-replica communication.
    - To fix the downside, you could have a leader based Paxos where one acceptor is the leader, the clients submit proposals to the leader, the leader skips to the Accept phase and then does the Paxos algo with the other replicas.

<u>5/1 - Week 5</u>

Chubby

- Chubby is an internal service at Google and it is a highly available coordination service. It serves as a lock service (so Chubby stores the locks and records of who has the locks at any particular time) and a file system for small files.
- Nodes will try to obtain the chunkmaster lock through try-locks (return with an error if you can't get the lock). If you get it then you are the chunkmaster, and if not then you are a normal node and you want to find out who the chunkmaster is.

    - On a path name, only one node can hold the lock and thus only one node can be the chunkmaster.
- Locks granted by Chubby are used by 3rd party programs. Chubby doesn’t care what happens after, it only deals with giving away locks. Chubby isn't running within any services, it's a separate one that just takes care of locks.
- So GFS cluster will use Chubby as a way of getting locks (I guess to replicate Paxos behavior).
    - There isn't a library that implements Paxos because Paxos isn't very fast, it's not the simplest protocol, developers don’t know how to use it (but they know how to use locks), and you need to export the result of coordination outside the system.
- There will be 5 servers (one is designated as the master) in a Chubby cell and client applications will use library RPC calls to contact those servers in order to create a replicated service that implements fault tolerant logs.
    - Since there is a master, Chubby is a leader based version of Paxos. In the case of master failure, then another replica has to be the master and it must catch up in terms of knowing the updates at slots.
    - Clients will submit reads and writes to the master. Thus, the master is the performance bottleneck.
- There could be inconsistencies with reads in Paxos RSMs because there may be a gap between Accept and Learn so replicas could have differences in understanding.
    - So this means that the linearizability property is not achieved since a read must see the effect of all the accepted writes. So we want future reads to be able to immediately see the latest write.
- In order to get the above property, we need a Paxos replicated log where the replica executes commands in a slot only after executing the commands in the prior slots.
    - The operations that will be reading state will be looking at this log.
    - In other words, this is a global log that orders reads and writes.
    - All reads should get accepted to one of the slots in the log.
- One other problem is the speed of Paxos and the performance at scale because the master is the performance bottleneck.
    - Clients should cache the data that they read and should try to read from the local cache when they need to read. This creates a potential issue of reading old value and thus violating linearizability. Helpful in the case where writes are infrequent.
    - To solve ^, the master can invalidate cached copies upon update. The library will receive a notification to update the cache.
- To help performance, there are also proxy servers in between the clients and the 5 chubby servers. It helps make invalidations and keeping a local cache at the proxies easier.
- In the case where a client gets a lock and then fails, the client will exchange keep-alives with the master and if the master detects the client is not sending messages, then the lock is revoked.
    - Problem could be that if the client isn't sending messages anymore, it does not necessarily mean that it is down, it could be another reason like a network delay.

Zookeeper

- Goal of Zookeeper is to do coordination for other services (aka wait-free coordination). It addresses the need for polling in Chubby (if you don't succeed in getting a lock, then you should do other things that don't require the lock and check periodically for the lock).
- Clients can register to watch a file, and Zookeeper will notify the client (application is notified specifically) when the file is updated and when the lock gets released.
    - The problem is if everyone is watching the file, then everyone will try to get the lock when it is released.

5/6 - Week 6

- 3 ways to do RSMs: Lamport clocks, primary backup replication, and Paxos based RSM.
- FLP Impossibility Result: In an asynchronous model, the distributed consensus is impossible if there is even a chance that one replica may fail. Aka because there is a possibility that one node may fail, you might not be able to have a consensus.
    - Holds for even small cases where you have weak consensus where only some process must learn.
    - The underlying issue is that we can't distinguish network being down from slowness. You have to choose safety or liveness.
        - If you haven't heard from a node and you want to make a decision asap (within some bounded time), then you have to give up safety.
    - In order to get both safety and liveness, we need failure detectors and need to make assumptions about the network delay you're willing to tolerate.
- CAP Theorem is where in a distributed system you choose any two between consistency, availability, and partition tolerance (system can handle partitions in network where nodes can't talk to each other).
    - Most of the time choose between CP and AP.
    - A case where you where you would sacrifice availability for consistency is the bank account example or Google docs where the order of operations is very important.
    - A case where you where you would sacrifice consistency for availability is when you want to respond quickly and push off having to sync up beforehand. A good example would be the ordering of the FB news feed. It's okay if the order is wrong or if you don't see posts for a while.
- PACELC says that if there is a partition, then we choose between availability and consistency and if not partition, then we choose between latency and consistency (situation where you want to respond quickly and skipping having to sync up immediately).
- When a replica gets a new read and write, in order to not violate linearizability, you have to wait until you get in a majority partition.

- In one example scenario, you have a shared calendar application and there is a local copy of the calendar on every phone and there is no master copy. You only get to sync with a peer when you have WiFi (bluetooth is low range and cellular data is expensive).
    - The goal is that we have automatic conflict resolution when the replicas with their own local copies of the calendar finally get to sync with each other.
    - The issue is that if you have two users that try to schedule a meeting at the same place/time.
        - One possible solution is to have users submit backup times/places that would also work. This helps with conflict resolution a little.
        - Situations where that wouldn't work are where two events with the exact same time/place (and has same options for backups). Now, depending on which replica the user syncs with first, the event timings could be different.
            - If you want to ensure consistency, you want to make sure that each user syncs with the replicas in the exact same order.
    - Getting the consistent ordering (even with intermittent connectivity) is possible through Lamport clocks.
    - Good to note that in this application, we are accepting events when they come in, and then syncing afterward when we have the chance. Therefore, we have to save snapshots of updates (aka maintain a log of updates) since when you do eventually sync, you might have to revert some operations and then apply the updates in order.
        - As an effect, the users may see a local change, and then after a sync, then those changes may be reverted and other changes could take precedence.
- There is eventual consistency in the above example which means that if there are no new updates, then all replicas will eventually converge to the same state, but there aren't any guarantees about the consistency at any arbitrary time.
- One issue is that if there is one replica that is almost always disconnected, then this prevents others from declaring that their updates are stable and forces them to keep a pretty large list of updates and doesn't let it garbage collect since we will need to send those updates over to the disconnected node when it decides to sync up.
    - Solution is to pick one of the replicas as the primary and have it determine the order of the updates.
- At each of the replicas, we have to store the stable state as well as a log of tentatively ordered updates. When you sync with the primary, then receive the ordered updates and apply the updates and then prune the log of the updates you don't need to keep track of anymore. At this point, the updates are final and not tentative anymore.

<u>**5/8 - Week 6**</u>

- One of the issues with the RSMs talked about in this class is that it isn't really scalable. As you add more servers, every replica has to see each operation since they all have to be synced up with the correct info. Also, idle servers will be underutilized.
    - Also, important to note that it is easier to do horizontal scaling (adding more servers) than vertical scaling (getting better servers), so we need distributed systems that are able to scale better.
- The assumption that we'd had is that all replicas need to have the entire state. However, this does not have to be the case. For example, one FB server doesn't have the entire information of users in its machine. Instead, we can split up the state between different machines and when you need to access some piece of state, then contact the machine that has the state.
- In order to partition state (e.g splitting a dictionary across machines), you could apply the hash function to the key, then mod by the number of servers, and then store that KV pair into that server.
    - One issue comes up when the number of servers changes (one dies or one gets added). Your hash function will get changed, and thus data will have to be moved from server to server since it is likely the new hash function will change the location a data point will get mapped to. This data shuffle is not that scalable especially since the number of servers at any point is likely to change.
    - The solution is to consistent hashing where you represent the hash space as a circle with all the possible hash values. Assign every server an ID, hash that ID which will get it mapped to something between 0 and N where N is the # of servers, and the server is responsible for the keys between itself and the predecessor.
        - Then, to map the key to a server, hash the key so it can fall into a spot in the circle, find the successor server, and query that server for the value pair.
        - When you add a server, part of the data from one shard will move to another server, but nothing else changes. Whatever data they're storing, they will continue storing it.
            - You're splitting an existing shard into two.
        - When a server is removed, the data that was stored on that server will get added to the successor server.
            - You're moving a shard to another server.
- Each server can also act as being a set of virtual nodes. Each server gets a set of random IDs. So each virtual node will own 1 / v*N hash space. Each server will need to make sure it has all the info assigned to its virtual nodes.
    - The idea is that even if you have N servers, you're splitting the load and subsequently the output of the hash function into more buckets.
    - Aka helps with the distributing of keys.
    - It also allows for data to get split across lots of machines since one server doesn't just have one contiguous segment in the hash space.

- In the case where each server doesn't have information about every single server on the system (because in that case you could just apply the hash, find the server responsible and then forward the operation to that server), success pointers allow requests to get forwarded from successor to successor until it eventually gets to the server that has the key value.
    - The plus side is that each node doesn't need to know info about all the nodes in the system.
    - The problem is that you're doing an O(N) lookup to figure out what server stores a particular key.
- So, it's not enough for a node to have a pointer to the successor node, but rather they need a finger table which has log N entries where N is the size of the hash space. So you have log N of these pointers to other nodes in the system. The pointers will be ½-way across the space, ¼-way across the space, etc.
    - You want to forward the request to the server that has the largest hash value but that is smaller than the key.
    - So, lookups will take O(log N) hops.

5/13 - Week 7

- Amazon Dynamo is the underpinning of the S3 architecture.
    - Dynamo gives you eventual consistency and thus is able to have a very high level of availability.
- Dynamo is necessary because you have tens of millions of customers and data is spread across tens of thousands of servers.
    - Ex) Keep track of information on users' shopping carts.
- The main goals are high availability (system is operational in the face of lots of possible failures) and low latency (shopping operations should get done very quickly - customers buy less if it takes more time).
- Dynamo mainly uses consistent hashing where each node/server has an ID, and you assign a piece of data to a place on the hash space and thus one node will have to store it.
    - Instead of having just one node store that piece of data, you replicate the value for every key at N clockwise successors of the key.
- Each write is still submitted to the first successor, it acts as the coordinator, and then it will forward to the next N-1 successors.
- Normally, clients shouldn't have to know about the other servers in the system. That's why there is a front end server in between the client and the backend servers.
    - In Dynamo though, there isn't really a distinction between the frontend and backend servers.
    - Thus, each server will need to contact a randomly chosen server and exchange their lists of known servers, so that eventually all the servers will know of all the other ones. However, when a new node is added, it may take a while for all servers to learn about it. Also, there may be differences in the view that each

node has of the servers that are alive. This is okay though since the system is eventual consistency and thus it's okay to have periodic inconsistencies.

- Overall process for writing/reading
    - Coordinator will forward the request to the N-1 successors.
    - Wait for response from R or W replicas where R is the number of nodes that need to respond in order to execute the read, and W is for write requests.
    - In order to maximize availability and to minimize latency, we should make R and W equal to 1 since you only need one node to respond (basically the original node that the request got sent to).
        - These are system and application dependent hyperparameters.
    - In order to ensure that the read will see the last committed write, you need to set R + W > N in order to make sure that there is an intersection between the nodes that have seen a particular write (W) and the nodes that are required to execute a read (R).
        - Low read latency will mean R -> 0 and W -> N.
        - Low write latency will mean R -> N and W -> 0.
    - There could be a case where you have a get request, and two nodes have different values. How do you handle this?
        - Associate each piece of data with a vector clock in addition to the value. Problem is that if you have N servers, then you need a vector with N numbers for each KV pair.
        - Better approach is to have a list of (coordinator node, counter) pairs.
            - Whenever a node gets a request and it is the coordinator then it increments/adds a pair to the vector clock that is associated with a particular key.
    - Therefore, when you have a get request, and the nodes return different values, then you compare the vector clocks associated with that key at different nodes, and return the value that has the greater clock value.
        - However, sometimes you get concurrent events in the clock comparison. An example is a node with VC of [(A,1)] and another with a VC of [(C,1)] Just make sure the way you choose between these concurrent events is deterministic (i.e look at process IDs or something). You can also take the union (and this would be helpful in that the client will get both A and C. Useful in the case where A and C are items to be placed in a cart)
- VC’s can also help with garbage collection in the following way.
    - When responding to a GET, then Dynamo should return the VC. When the client sends a later PUT, then it should send that VC, and then when Dynamo gets it, it can compare that VC with its current VC, and if the current one is greater than the sent VC, then we can garbage collect the previous value.
- In Dynamo’s client interface, along with a client PUT, the client sends what it currently knows about the version vector. And then the server, in response to a client GET, will send a list of values with their contexts instead of just one value.

- We need to be able to trim the VCs since they can get really long. One approach is to drop the least recently used node when the VC gets above a certain size.
- One issue is that, in the case of nodes dying and there being partitions, it is possible if there is no overlap between the W nodes written to and the R nodes read from.
    - Basically the R + W > N may not be helpful in the case of failures.
- 2 kinds of failures to deal with
    - Permanent Failures: Need to mark manually and needs to be discovered via gossip. Human pretty much has to come in and intervene.
    - Temporary Failures: Being able to read from R and write from W handles this. However, if you have so many failures that you don't have W machines, then you could get stuck. Also, the nodes need to be able to catch up when they go down.
        - E should basically inform C when C comes back up. Possible problem is if E fails before C can come back up. The solution there is to periodically gossip in that nodes will compare the KV pairs they hold, and copy any missing keys the other node has.

<u>5/15 - Week 7</u>

- Main concepts with Dynamo are that it uses consistent hashing. You use a one hop DHT that means you don't have to use the log n approach to find the correct node. Data is also replicated across N different nodes (referred to as sloppy quorums). Vector clocks also used to identify precedence when two servers are not agreeing with each other on the value to be returned.
- Efficient synchronization of KV maps
    - Just to identify where they are different. The leaves are the hashes of the keys and the parent of the two is a hash of the concatenation. When comparing two trees, start at the top and see where the hashes are different and then go down to see what the actual key is that is different between the two.
- When you are initializing a new server, it needs to make some RPCs to the successor and the successor needs to send over the KV information and delete the information on itself.
    - The problem is with the scale of Dynamo, you have lots of nodes coming in and going out at the same time. This will mean you will have to scan your map a lot when you're trying to make a change. You also need to recompute the Merkle trees.
- Basically when you have a new node or a node goes down, you need to change how the data is partitioned and where the partitions are placed.
    - One solution is to partition the hash space into a fixed number of equal sized shards. When you add/remove nodes, you're not changing how the data is split (since each server is still in charge of the same section of data), you're changing where the data is stored. Now, we have a Merkle tree per shard, but I guess the advantage is that they're smaller.

- Partitioning of state was originally important because we don't want to have ever a node store a full copy of the data. The problems with the hashing and DHT is if you have an operation that touches multiple partitions.
    - Ex) Add meeting to the calendars of two participants and the account data is stored on different shards.
    - The solution is to ensure atomicity and execute one transaction at a time.

5/22 - Week 8

- Reason for distributed transactions requires the partitioning of state among different machines.
    - Creates need for atomic execution of operations that need to access data across shards.
- Concurrency + Serializability means that the transactions should take place concurrently so that we get the effects of parallelism but we want serializability in that the externally visible effects should have some serial order of execution.
    - We want situations where we have T1 then T2 or T2 then T1, but we don't want the two operations to interleave.
- Serializability is guarantee on set of transactions while linearizability is guarantee on single operations.
- To get serializability and concurrency, we are able to do so if the data written and read is disjoint.
- Splitting up the data into shards means that there is a larger probability that there will be some shard that will fail which means that you won't be able to do your operation. There are basically more points of failures which can cause the operation to stall. Partitioning often causes availability to go down.
    - Replicated state machines is one possible solution.

5/29 - Week 9

- Disjoint transactions can execute at the same time, but operations that have some overlap have to be done sequentially.
- One solution to try to improve concurrency in the latter situation is to only acquire locks for data transactions that will write or modify the data. However, you need to make sure that if T1 and T2 are fine running parallel even if they aren't disjoint, then T3 can't be running at the same time where T3 modifies the data that T1 and T2 are reading from.
- Two phase locking is pessimistic and they acquire locks because they assume that conflict will occur.
    - However, you can have optimistic concurrency control where you execute the transaction assuming no conflict and then you should verify that no conflict took place later on. Read -> Compute transaction -> Validate -> Write
    - This ^ is preferable because a lot of times we want throughput to be high.
- Validation is what lets you make sure that you don't violate the serializability property.

    - The TC will send the transaction to the validation server and the server will response with a yes you're good to go or no you should reread and re-do the operation.

6/3 - Week 10

- Difference between Azure and S3 is that Azure is strongly consistent while S3 is eventually consistent.
- The goals for Azure are durability, strong consistency, high availability, and scalability.
- Azure can only provide storage within a single geographic datacenter.
- Primary backup replication is the main technique used to create strong consistency.
    - Main issue is that the viewservice might not know to change views when it can communicate with everyone but it won't know if there is a situation where the primary can't communicate with the backups.
- The key idea in Azure is that the client triggers the view change (instead of viewservice doing that change) by contacting the view service when it's not getting a response from primary.
    - Main issue with this is that changing the view is expensive because that requires primary to send the whole KV store to another server to get it synced. We don't want to do this if there is a small error like a packet drop (client will sense a failure and will trigger a view change). There would be too many view changes.

6/5 - Week 10

- Google Spanner is Google's globally distributed database
- Spanner is an actual database while Azure and S3 are just key value stores.
    - This is a database that is spread across data centers
- Spanner wants the transactions to be serial (externally visible effects must be equivalent to some serial order).
    - This is different from linearizability in that there is no guarantee of real time ordering. Serializability just has to match some serial ordering
- Spanner offers strict serializability which has both linearizability and serializability.
- Each transaction has a commit timestamp and when a row is modified by a transaction, a new version is created with a new timestamp.
    - When read only transactions come in, it will look for the latest version of the data committed before the transaction timestamp.
    - Since we're dealing with timestamps, the commit timestamps should be monotonically increasing even across all of the different machines. This is an issue we faced before. Lamport clocks were the answer back then, but using them would be too slow.
- The answer is to the monotonically increasing timestamps is to try to do clock synchronization. This is able to be done because of more precise GPS, atomic clocks, and signaling methods.
    - GPS even has issues if you're in cities or if you have weather constraints.

        - Solution to this is to have server specialization in that some are focused on GPS and some are focused on storage, computation, etc. All servers will periodically sync with time masters that will have specialized atomic clocks.
- Given that the servers will be syncing periodically (every 30 seconds), TT.now() call will return a range [earliest, latest] and the current time will be guaranteed to be somewhere in this range.
    - There is some bound on the error of the time.