Design and Engineering of Computer Systems

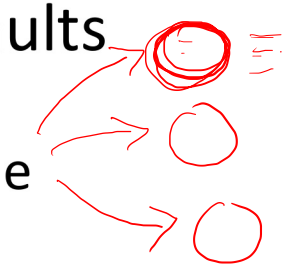
Lecture 39: Distributed Transactions

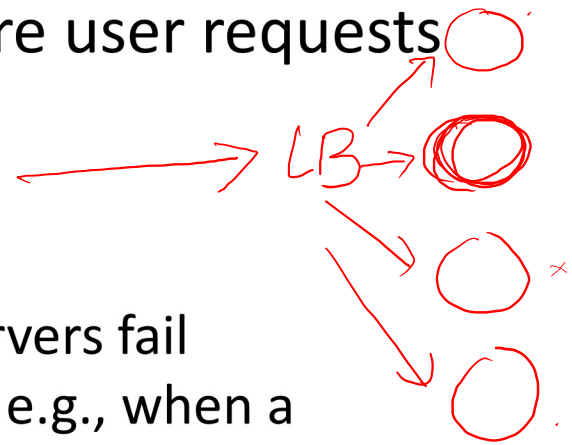
Mythili Vutukuru

IIT Bombay

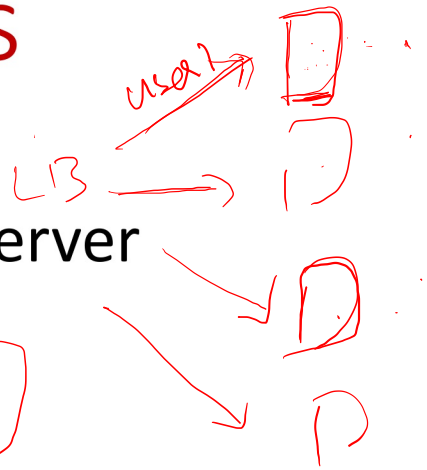
Reliability engineering

- Story so far this week: how to make systems more reliable by masking faults before they turn into failures
  - Have multiple replicas of app servers and app data, so that one can take over if the other fails, app server functions correctly
  - Within each server, ensure all-or-nothing atomicity of operations in a transaction, even if execution is interrupted by a fault, using write-ahead logging
- Another reason for multiple replicas: horizontal scaling, where user requests and app data partitioned across multiple shards of a server
  - Each shard may be further replicated for fault tolerance
- This lecture: how to handle faults in a partitioned system
  - How to partition data such that partitions stay stable even when servers fail
  - How to guarantee all-or-nothing atomicity across partitions/shards, e.g., when a transaction spans multiple shards (distributed transactions)





How to partition application into shards

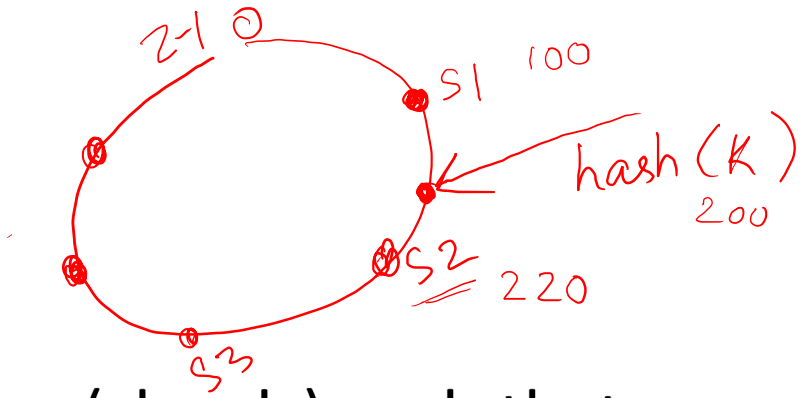


- K application data items/keys (e.g., user shopping carts) and N server shards/replicas: how to assign keys to servers?
- Simple strategies: round robin, least loaded
  - Round robin: assign key to servers in round robin manner when key seen first time
  - Least loaded: assign key to least loaded server when key seen first time
  - Problem: need to maintain lookup table mapping keys to assigned servers, to redirect future requests from same key to same server, causes high overhead
- Technique to easily derive key-server mappings: hashing
  - Map each key to a number using some hash function
  - Assign key K to server number [hash(K) mod N]
  - Problem: If N changes due to server failures, mapping of all keys to servers changes

$42 \bmod 10 = 2$

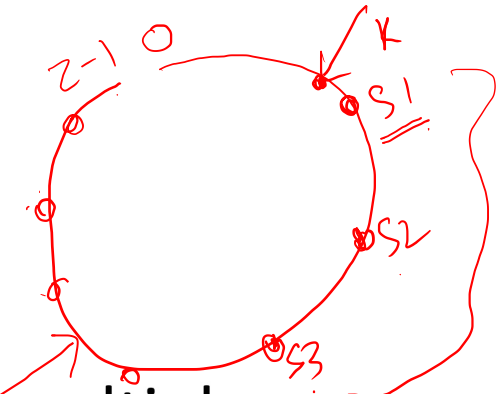
$\bmod 9 = 6$

Consistent hashing



- A technique to partition keys (app data) to servers (shards) such that
  - No need to remember mappings for every key to assigned server
  - Very few mappings change if servers go down or come up
- How consistent hashing works
  - Server identifiers are hashed to a value in a circular range, say [0, Z-1], i.e., each server occupies a certain position on a circular ring / number space
  - Application data / key is also hashed to same range, and stored at a server immediately succeeding it in the ring
  - No need to remember mapping for each key, only need to remember positions of server shards on the ring, can compute server storing any key
  - If a server shard fails or joins, only keys at its neighbors change, minimal disruption to key-server mappings

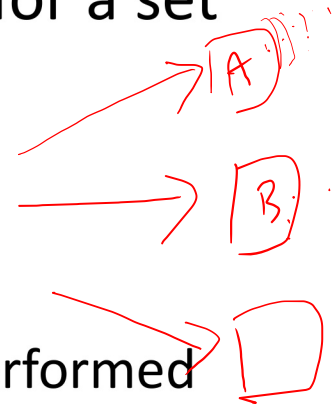
Horizontal scaling and replication



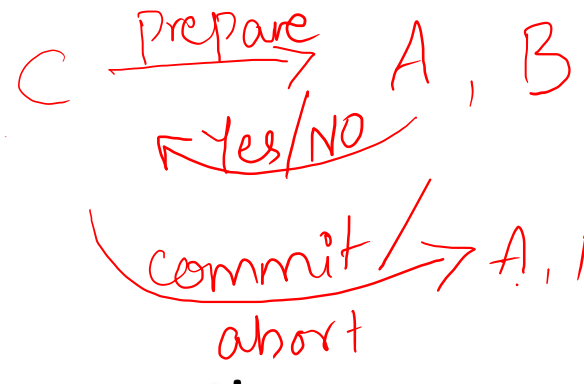
- For fault tolerance/replication, a key can be stored at multiple servers succeeding the key on the ring, not just the first one
  - Multiple servers at which a key is stored can run some consensus protocol, e.g., Raft, in order to store app data pertaining to a key consistently
- Understand the difference between replication and partitioning
  - We partition keys (app data) to servers using consistent hashing for horizontal scaling of performance, each shard/partition stores different sets of keys, handles different user requests
  - We replicate the same key at multiple server replicas for fault tolerance, so that the application processing can proceed correctly even with failures
  - Usually, application servers need to do both partitioning and replication: partition data into shards, replicate each shard at multiple replicas
  - Same server can be part of multiple shards as a primary/backup replica

Distributed transactions

- Atomicity discussion so far: how to achieve all-or-nothing semantics for a set of operations within a transaction in one server, in spite of failures
  - E.g., checkout shopping cart (billing + shipping), file system changes
- Sometimes, operations in a transaction need to be performed across multiple partitions/shards: distributed transaction
  - E.g., transfer money from one account to another involves two steps to be performed atomically: debit from one account, credit to another account
  - What if both accounts stored in different shards/partitions? How to ensure transaction executes atomically across both shards?
- Two phase commit: protocol for all-or-nothing atomicity across multiple nodes in a distributed transaction processing
  - Coordinates amongst multiple nodes to ensure that all nodes commit or abort the transaction together

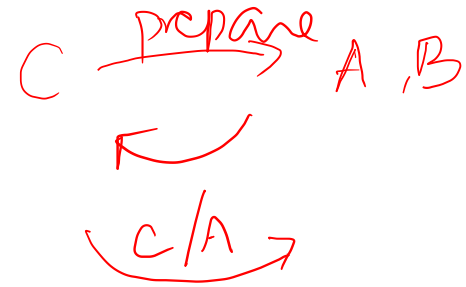


Two phase commit: overview

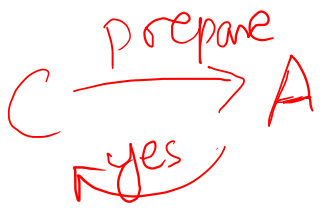


- Two phase commit: assume a node C coordinates the transaction between nodes/servers/shards A and B (can be more than two also)
  - Phase 1: C sends prepare messages to A and B. A and B can either reply Yes or No to agree/disagree to proposed transaction
  - Phase 2: If both A and B reply Yes, then C sends a message to A and B asking them to commit. If any one said No, C sends a message to abort
- Ensures that distributed transaction will commit only if all parties are willing to commit, else it will abort
- What about failures? What if one of A,B,C fail in between the protocol and forget what they have said? For example, what if B said Yes in phase 1, fails, doesn't commit in phase 2?

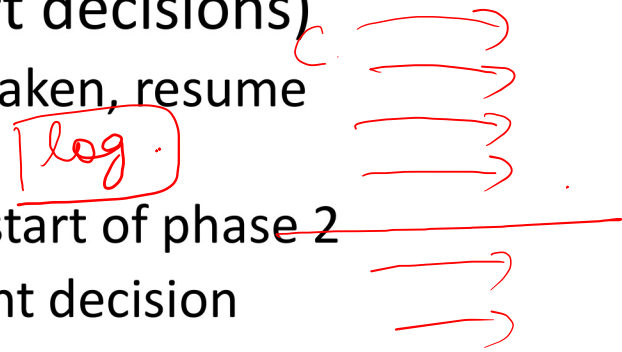
Two phase commit: coordinator failure



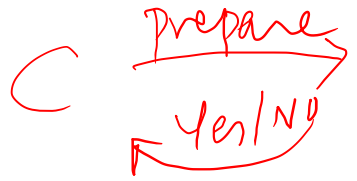
- If coordinator C fails in phase 1 before sending prepare message, A and B will detect failure, abort transaction
- If coordinator fails during phase 1 (A does not hear back after its vote)
  - If A voted no, A can simply ignore transaction since it will abort anyways
  - If A voted yes, A can neither commit nor abort, must repeatedly contact C to know whether to commit or abort



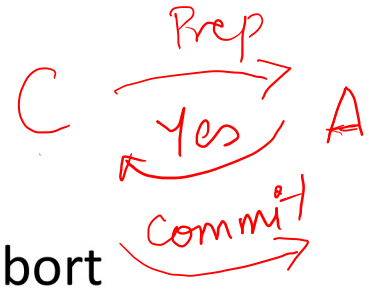
- If coordinator fails during phase 2 (while sending commit/abort decisions)
  - When coordinator resumes, must remember commit/abort decision taken, resume execution and complete conveying decision to all servers
  - Coordinator must log commit/abort decision to persistent storage at start of phase 2
  - Even if coordinator crashes and resumes, all nodes see same consistent decision



Two phase commit: node failure



- What happens when one of the nodes/servers/shards fails?
  - If node fails before voting, or after voting No in phase 1, coordinator aborts transaction anyway, so no harm done
  - If node fails after voting yes in phase 1, must wait for coordinator's commit/abort decision after restart, and commit transaction if required
  - To ensure node commits after voting Yes in phase 1, node should log all changes needed to commit transaction before replying Yes, so that transaction can be installed correctly if coordinator asks to commit in phase 2
  - If coordinator decides to commit transaction, must necessarily contact all nodes in phase 2 and ensure they receive the commit decision, in spite of failures
- Practical issues with 2 phase commit: nodes/coordinator have to block in certain cases, have to repeatedly retry to establish communication



Summary

- In this lecture:
  - Consistent hashing to partition application data to shards
  - Two phase commit for all-or-nothing atomicity in distributed transactions
- Revisit the Dynamo paper to read more about the concept of consistent hashing
  - "Dynamo: Amazon's Highly Available Key-value Store", DeCandia et al.