Block chain / Bitcoin

(connect to some neighbors) gossip

Transactions grouped into a block that gets added to the chain (history of transaction) by the leader of that block" (="closet to 0") lottery LE.

① Choosing the random number ② Using a seed(H) seed||X

③ Iterated Hashing / proof-of-work

Longest chain Rule (Majority decision)

Incentives for Logging: Transaction fees and Mining rewards

Set Logging Speed by hardware speed/num of particants (block mined per h)

Transaction

ACID: ① Atomic: all-or-nothing; ② Consistent: rules maintained; ③ Isolation: multiple transactions do not interface with each other; ④ Durability: values preserved even after crashes.

① Commit transaction to make tentative updates permanent; Abort to roll back to pre; ② Checked at commit time, abort if not satisfied

③:

1. Lost Update Problem

2. Inconsistent Retrieval Problem

Increase concurrency by targetting serial equivalence

Interleaving ↑ Two op. are conflicting operations if their combined effect depends on their orders

Sol: 1. Pessimistic: assume the worst and prevent: Locking

2. Optimistic: assume the best and allow but check (e.g. at commit time)

1.

① → Read-write Locks (two modes)

Two-phase Locking: A transaction can not acquire (or promote) any locks after it has started realeasing locks. (SE)

Two phases: ① Growing phase: only acquire locks;

② Shrinking phase: only realease locks; "Strict..." release only at commit.

Combating Deadlock: 1. Lock all objects in the beginning in a single atomic step; 2. Lock timeout (abort if not within); 3. Deadlock detection: keep track of wait-for graph and find any cycle => abort one/more to break the cycle.

2.

Transaction id determine by serialization order;

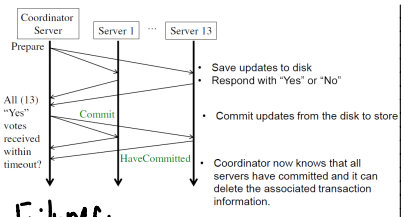
T's write to object O is allowed only if transactions that have read/written O had lower ids than T;

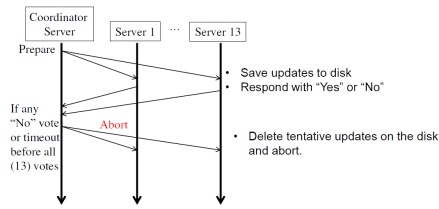
T's read to Object O is allowed only if O was last written by a transaction with a lower id than T;

Rule	T_c	T_i	
1.	write	read	T_c must not *write* an object that has been *read* by any T_i where $T_i > T_c$ This requires that $T_c \geq$ maximum read timestamp of the object.
2.	write	write	T_c must not *write* an object that has been *written* by any T_i where $T_i > T_c$ This requires that $T_c >$ write timestamp of the committed object.
3.	read	write	T_c must not *read* an object that has been *written* by any T_i where $T_i > T_c$ This requires that $T_c >$ write timestamp of the committed object.

Distributed Transactions

Two-phase commit:





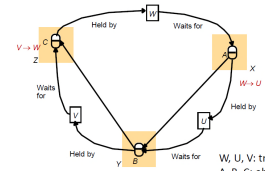
Failures:

- If server voted Yes, it cannot commit unilaterally before receiving Commit message.
  - Does not know if other servers voted Yes.
- If server voted No, can abort right away.
  - Knows that the transaction cannot be committed.
- To deal with server crashes
  - Each server saves tentative updates into permanent storage, right before replying Yes/No in the first phase, so it is retrievable during crash recovery.
- To deal with coordinator crashes
  - Coordinator logs decisions and received/sent messages on disk.
  - After crash recovery, can retrieve the logged (saved) state.
- To deal with Prepare message loss
  - The server may decide to abort unilaterally after a timeout during the first phase (server will vote No, and so coordinator will also eventually abort).
- To deal with Yes/No message loss
  - coordinator aborts the transaction after a timeout (pessimistic!).
  - It must announce Abort message to all.
- To deal with Commit or Abort message loss
  - Server can poll coordinator (repeatedly).

Handle Deadlocks:

① Decentralized Deadlock Detection

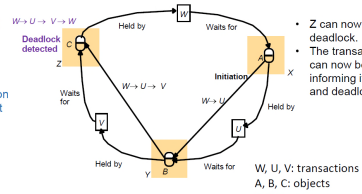
- Edge chasing: Forward "probe" messages to servers in the edges of wait-for graph, pushing the graph forward, until a cycle is found.



All servers know local wait-for relationships.

Coordinator for each transaction knows whether the transaction is waiting on an object lock, and at which server.

W, U, V: transactions
A, B, C: objects
X, Y, Z: servers



- Z can now detect a deadlock.
- The transaction in the cycle can now be aborted (by informing its coordinator), and deadlock breaks.

W, U, V: transactions
A, B, C: objects
X, Y, Z: servers

② Edge Chasing: Phases

- Initiation: When a server S_1 notices that a transaction T starts waiting for another transaction U, where U is waiting to access an object at another server S_2, it initiates detection by sending <T→U> to S_2.
- Detection: Servers receive probes and decide whether deadlock has occurred and whether to forward the probes.
- Resolution: When a cycle is detected, one or more transactions in the cycle is/are aborted to break the deadlock.

Phantom Deadlocks

Phantom deadlocks = false detection of deadlocks that don't actually exist due to messages containing stale data

- Edges may have disappeared in meantime
- Edges in a "detected" cycle may not have been present at the same time

This leads to spurious aborts of transaction.

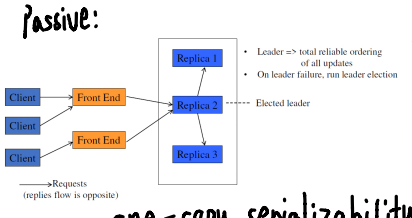
Replication: holds identical copies: $1-f^k$ with k replicas

Replication challenges: ① Transparency ② Consistency

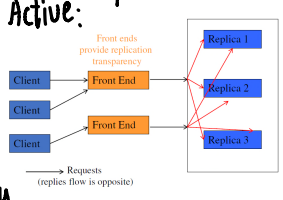
↓

passive replication / Active replica. "Replicated state Machines"

Passive:



Active:



one-copy serializability

HW6_mysol - Notepad

File Edit Format View Help

Q1

a) Availability and partition-tolerance are held

b) Availability and consistency are held

c) Availability and consistency are held

Q2

a) No. The distribution of R = 1,W= N is great for read-heavy workloads.
The configure of R = 4, W = 8 is great for write-heavy workloads.

b) Two conflicting writes may occur at the same time if the W that is smaller or equal to N/2

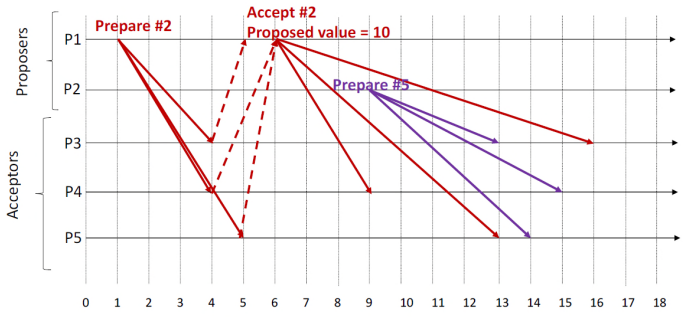
c) No, because W + R = N instead of larger than N.

d)Yes, because there is a overlap and the write operation that could not lead to a conflicting write. The overlap ensures that any read operation that follows a write operation will see the most recent completed write operation.

Q3

With the change of phase 2 and new phase 3 in Three-Phase Commit, the nodes that have voted to the prepare message will know whether the transaction will be committed or aborted. When the coordinator or any other participants fail, the remaining participants continue on dealing with the preCommit request or doAbort request instead of waiting to the failed nodes being recovered. By this buffer phase, the delay is decreased.
continue on dealing with the preCommit request or doAbort request instead of waiting to the failed nodes being recovered. By this buffer phase, the delay is decreased.

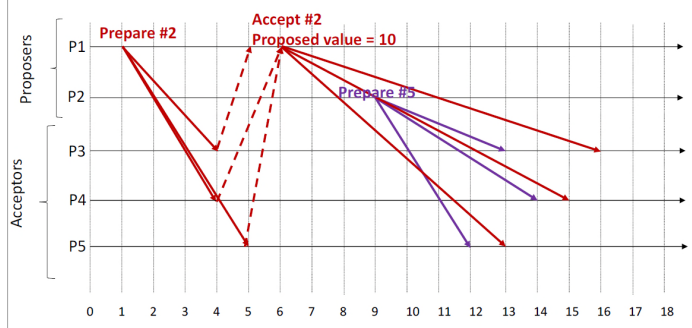
Paxos



P2 intends to propose a value 15.
Will P2 send an accept request? What value will it propose?

yes, 10

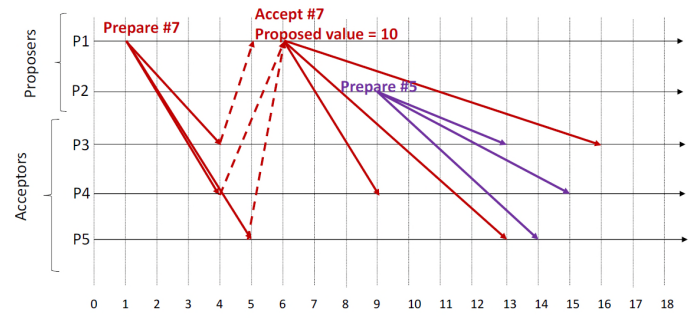
Paxos



How about now?

yes, 15

Paxos



How about now?

no

T1	T2
read A	
read B	
write A	
	read D
	write C
read D	
write B	
	read A
	write B
	write E

- What about this? serially equivalent
- Can it be achieved with strict two-phase locking? yes
- Can it be achieved with timestamp ordering? yes

T1	T2
read A	
read B	
write A	
	read D
	write C
	read A
read D	
write B	
	write B
	write E

- What about this? (serial equivalent) yes
- Can it be achieved with strict two-phase locking? No
- Can it be achieved with timestamp ordering? No

HW#5. Q1.

ECE428 hw5 solution

Question 1

a): 4ms.

2ms for leader sending AppendEntries.

2ms for follower replying.

b): The event when the leader of each cluster receives the message.

c): 32ms totally.

5ms for sending the update to participant clusters.

4ms for consensus in participants.

5ms for replying the request.

4ms for consensus in coordinator.

5ms for sending commit command.

4ms for Raft consensus in participants.

5ms for replying the HaveCommitted message.

d): Since there is no communication or node failures, the transaction must be committed for participant after the coordinator making a consensus.

Thus 5 + 4 + 5 = 14ms.

e): After making a consensus on the commit message from coordinator, each participant can delete the associated transaction information and release the transaction lock. Thus 27ms.

f): 8ms. Since there are 4 messages transmit during the Paxos.

g): The majority of consensus is make after the leader receive the local follower's reply.

new c: 4*2 + 3*10 = 38ms. 10ms for delay of two-phase commit.

new d: 2 + 10 + 2= 14ms.

new e: 38 - 2 = 36ms.

ECE428 HW3

Q1.

a) elif message.contents == "Election":
    # fill in the rest
    self.leader = self.pid # Assign the new leader
    for pid in self.group:
        if pid <self.pid:
        #Send Coor msg to all lower-number processes
        unicast (self.pid, "Coordinator")

b) P4 will send election msg to all the process with second highest PID P7, and P7 sends Coordinator msg to all of the lower-numbered processes(P1-P6). Hence, totally 1+6=7 msg

c) P4 send election to P7 first within T, then P7 send to P1-P6 Coordinator msg simultaneously within T. Hence, totally T+T=2T

d) P4 will still send election msg to P7 since the failure of P7 has not been detected. Within time out P4 does not get response from P7 so it sends election msg to P6. Then P6 send Coordinator msg to P1-P5. Hence totally 2+5 = 7

e) P4 send election msg to P7 and does not get response within T+T = 2T (Time out should be 2T for a round of msg transmission). Then P4 send election msg to P6 within T. And P6 send Coordinator msg to P1-P5 simultaneously within T. Hence, totally 2T+T+T=4T

f) Best case: exactly P7 detect P8's failure and send Coordinator msg to the lower with T.
Worst case: exactly P1 detect P8's failure while P7-P2 failed while P1 initiate the election. Then 2T * 6 = 12T is needed.

Q3.

a) B-multicast. We have assumed that unicast channels are reliable and considered Time out. We do not care on the failure in the multicast of Query msg, so pair-to-pair B-multicast should be used.

b) 2T

c) R-multicast. In the multicast of Decision msg, the phenomenon that the send may fail after it send the Decision msg could exist, which ask the reliability of multicasting Decision msg. And we do not care on the future input so pair-to-pair B-multicast is not needed.

d) Query msg from Pi to Pj within T

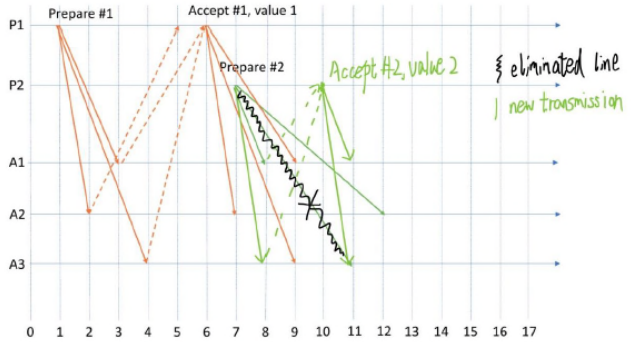
Reply msg with Xi from Each Pj to Pi within T
Decision yi from pi to each pj within T
Totally, 3T.

e) The process Pi may fail after multicasting msg to the group. And no failure detection is used to check the failure of Pi. Hence the future part of the algorithm could be meaningless and the safety could not be guaranteed for not reaching a consensus in the group.

Q4.

a) i. A1, A2, A3
ii. A2, A3
iii. P1 will send Accept msg after the Promise msg is taken after getting voted by the majority of acceptors (after receiving promise msg from Acceptor A1). Hence 3T+2T = 5T. P2 will send Accept msg after the Promise msg is taken by Acceptor A3. Hence 7T+2T = 9T.
iv. Yes, by iii we know P1 could send Accept msg to A1 and A2 at Time 5 and it take one time unit. So at Time 6 the P1 could be accepted by A1 and A2 while at Time 7 the Prepare msg just arrive at A3 (6<7). Hence, P1 could be accepted by a majority of acceptors. (2/3>1/2)

b) i. A1, A2
ii. A1, A2, A3
iii. P2 will send Accept msg after getting the Promise msg by the majority of acceptors (after A3). Hence 11T+2T =13T
iv. All of the acceptors: A1, A2, A3.
v. value 1. Since P2 has not start sending Accept msg.
vi. Change the arrived Time of Prepare msg to A3 from 11T to 8T hence P1 has not been accepted by all of the acceptors (at 9T, 8<9).



Q2.

a) i) No. A counterexample could be that there exist two processes carry the same interger zinput (i.e. xk = xk+1) and in this scenario it will mistakenly take xk as the max value since it think that xk is pass through the whole loop while actually it is just passed to the next process.

ii) The judgement that xk is pass through a whole loop can not just be the value of it. Instead, there should be something record whether process pi has been checked. i.e Initialize a sent p_check as a blank set, when pi pass msg (PROPOSAL, X_K, pi), p_check.append(pi). Then change the judgement else #b==X_K into elif pi in p_check. By doing this the safety could be guaranteed.

ECE428 HW4

Q1

a) 1/7 conflict on A; 5/4 conflict on B; 6/3 conflict on C; 2/9 conflict on D

b) Yes, it is interleaving serially equivalent(T1, T2).

c) Yes, it is possible for both. (T3, T1) ∥ for B; (T2, T3) ∥ for D

d) A=5; B=5; C=6; D=1; E=3;

e) (T1, T2, T3): A=5; B=5; C=3; D=1; E=3;
(T1, T3, T2): A=2; B=5; C=3; D=1; E=0;
(T2, T1, T3): A=5; B=9; C=7; D=5; E=3;
(T2, T3, T1): A=5; B=5; C=10; D=1; E=3;
(T3, T1, T2): A=2; B=5; C=6; D=1; E=0;
(T3, T2, T1): A=2; B=5; C=7; D=1; E=0;

f) If we have (T1,T2) and (T2,T1), the transaction will forms a cross and this cannot be serially equivalent.

g) Where there is a cycle or a cross, the second side should have a lock to wait till the first side release the lock. (i.e. If we have (T1,T2),(T2,T3),(T3,T1), in the transaction of (T3,T1), there should be a lock on T1).

h) T1 was not available since it was writing a reading value.

i) Take write(A,z+1) in T2 above and move x=read(A) in T1 down to make it in serial T2->T1.
Thus,we would have T2− > T3− > T1 in total serial order and not one end even all started

Q2

a) According to the recurrence Nt+1 = |Nt + Nt(N − Nt)/(N − 1)|,
t=0, N = 1;
t=1, N = 2;
t=2, N = 3;
t=3. N = 3+2 = 5;
...
t=11, N = 100;
The round we need is 11.

c) yuhangc3

```
import os
for num in range(1,100000):
    ret = os.popen(" echo sc54 "  + str(num) + "  | sha256sum" )
    rel = ret.readlines()
    if (rel[0][0:5] == " 00000"):
        print("the number is",num)
        break
```

Q3

a) T1 should be committed first; then T2 would be aborted since WITHDRAW C is violated; T3 would be committed and finally T4 would be aborted.

b) A:0; B:30; C:40

P4:

a):

484408336:

```
0: 487574807 1: 487574807 2: 487574807 3: 487574807 4: 487574807 5: 487574807 6:
487574807 7: 487574807 8: 487574807 9: 487574807 10: 487574807 11: 487574807 12:
487574807 13: 487574807 14: 487574807 15: 487574807 16: 487574807 17: 487574807 18:
487574807 19: 487574807 20: 487574807 21: 487574807 22: 623671120 23: 623671120 24:
623671120 25: 623671120 26: 623671120 27: 623671120 28: 764379973 29: 1063543501 30:
1560975388 31: 2659741832
```

1095619456

```
0: 1098796346 1: 1098796346 2: 1098796346 3: 1098796346 4: 1098796346 5: 1098796346
6: 1098796346 7: 1098796346 8: 1098796346 9: 1098796346 10: 1098796346 11:
1098796346 12: 1098796346 13: 1098796346 14: 1098796346 15: 1098796346 16:
1098796346 17: 1098796346 18: 1098796346 19: 1098796346 20: 1098796346 21:
1098796346 22: 1111608456 23: 1111608456 24: 1120746458 25: 1151137960 26:
1181992062 27: 1252080068 28: 1381160637 29: 1656324640 30: 2172985745 31:
3358105298
```

3500380522

```
0: 3520503822 1: 3520503822 2: 3520503822 3: 3520503822 4: 3520503822 5: 3520503822
6: 3520503822 7: 3520503822 8: 3520503822 9: 3520503822 10: 3520503822 11:
3520503822 12: 3520503822 13: 3520503822 14: 3520503822 15: 3520503822 16:
3520503822 17: 3520503822 18: 3520503822 19: 3520503822 20: 3520503822 21:
3520503822 22: 3520503822 23: 3520503822 24: 3520503822 25: 3549309559 26:
3577574332 27: 3675748542 28: 3796694723 29: 4056579626 30: 340980054 31: 1381160637
```

b):

The minimum number of distinct fingers is 6 and maximux is 10. Approximately, the nodes are equally spaced.

c):

484408336 stores the least keys.

3500380522 stores the most keys.

d):

The key 0x12345678 is stored in the node 340980054(0x1452f156), which is the 30th finger of 3500380522.

(340980054)

e):

Now the key is stored in the node 357445747.

Q5

a) read(A)T1, write(B)T1, write(A)T1, read(C)T2, write(D)T2, read(C)T1, write(E)T1, read(A)T2, read(E)T2, write(B)T2

b) T2 starts until it readlock(A) and read(A).
T1 is available to readlock(A) and read(A). Then writelock(B) write(B).
Then the first lock is needing T2 to release readlock(A) for T1 to writelock(A) and write(A).
While in T2, it need T1 to release writelock(B) to write(B).

c) read(A)T1, write(B)T1, write(A)T1, read(C)T2, write(D)T2, read(A)T2, read(C)T1, write(E)T1, read(E)T2, write(B)T2;
When in read(A) T2, it start to release locks from T1 of writelock(A). But in T1, later it requires locks for read(C) and write(E).
Thus, it still requires the locks, which is not a 2-phase locking condition.

d) We focus on A in this situation. Let T1 to read(A) first then T2 read(A) then T1 write(A).
Thus, for A.RTS, it is [1,2]. But when T1 is trying to write A. Its id is 1 which is lower than the maximum id of A.RTS, which is 2. Thus, it would call T1' s abort.

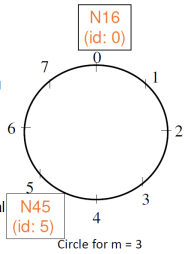
e) read(A)T1, write(B)T1, write(A)T1, read(C)T2, write(D)T2, read(C)T1, write(E)T1, read(A)T2, read(E)T2, write(B)T2;
Timestamp as following:
A.RTS = [1]; then B.TW = 1; then A.TW = 1; then move to T2;
C.RTS = [2]; then D.TW = 2; then move to T1;
C.RTS = [2,1]; E.TW = 1; then T1 all commited and go to T2;
We can process the following 2 process since all needed in T1 is commited. Thus, A.RTS = [1,2], E.RTS = [2]; And since it was in T2, 2 is the largest number so the final process write(B) can be committed

Distributed Hash Tables

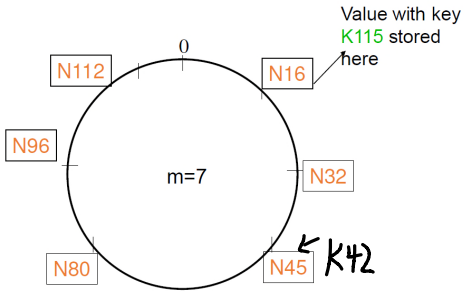
- Peer-to-peer protocol for efficient insertion and retrieval of key-value pairs.

Chord: Consistent Hashing

- Uses Consistent Hashing on node's (peer's) address
  - SHA-1(ip_address,port) → 160 bit string
  - Truncated to m bits (modulo 2^m)
  - Called peer id (number between 0 and 2^m-1)
  - Not unique but id conflicts very unlikely
  - Can then map peers to one of 2^m logical points on a circle



Ring of Peers: Running Example

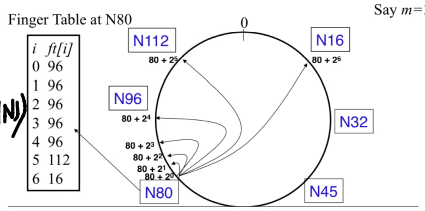


Performing Lookups

- Chord chooses a middle-ground.
  - Each node is aware of m other nodes.
  - Maintains a *finger table* with m entries.
  - The i-th entry of node n's finger table = successor(n + 2^i)
    - i ranges from 0 to m-1

The search for key takes O(log(N)) routing building block

Finger Tables



If a node fail:

One solution: maintain r multiple successor entries. In case of failure, use another successor entry.

One solution: replicate key-value at r successors

Chord Summary

- Consistent hashing for load balancing.
- O(log N) lookups via correct finger tables.
- *Correctness* of lookups requires correctly maintaining ring successors.
- As nodes join and leave a Chord network, runs stabilization protocol to periodically update ring successors and finger table entries.
- Fault tolerance: Maintain r ring successors and r key replicas.

Search under node failures

- If every node fails with probability 0.5, choosing r=2log(N) suffices to maintain lookup correctness (i.e. keep the ring connected) with a high probability.
  - Intuition:

Pr(at given node, at least one successor alive)=

$$1-(\frac{1}{2})^{2\log N}=1-\frac{1}{N^2}$$

Pr(the above is true at all alive nodes)=

$$(1-\frac{1}{N^2})^{N/2}=e^{-\frac{1}{2N}}\approx 1$$

New node joins

- A new peer affects O(log(N)) other finger entries in the system, on average.
- Number of messages per a node join (to initialize the new node's finger table) = O(log(N)*log(N))
- Proof in Chord's extended TechReport.

New node joins

New node contacts an existing Chord node (introducer).

Introducer informs N40 of N45.

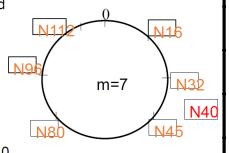
N40 initializes its ring successor to N45.

N40 notifies N45, and N45 initializes its ring predecessor to N40.

N32 realizes its new successor is N40 when it runs stabilization.

N32 notifies N40, and N40 initializes its ring predecessor to N32.

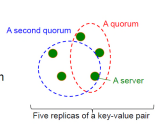
Periodically and eventually, each node update their finger table entries.



Quorums?

In a nutshell:

- Quorum = (typically) majority
- Any two quorums intersect
  - Client 1 does a write in red quorum
  - Client 2 does read in blue quorum
- At least one server in blue quorum returns latest write
- Quorums faster than ALL, but still ensure strong consistency
- Several key-value/NoSQL stores (e.g., Riak and Cassandra) use quorums.



Quorums in Detail

- R = read replica count, W = write replica count
- Necessary conditions for consistency:
  1. W+R > N
     - Write and read intersect at a replica. Read returns latest write.
  2. W > N/2
     - Two conflicting writes on a data item don't occur at the same time.
- Select values based on application
  - (W=N, R=1):
    - great for read-heavy workloads
  - (W=1, R=N):
    - great for write-heavy workloads with no conflicting writes.
  - (W=N/2+1, R=N/2+1):
    - great for write-heavy workloads with potential for write conflicts.
  - (W=1, R=1):
    - very few writes and reads / high availability requirement.

Cloud Computing

Features of cloud

I. Massive scale.
- Tens of thousands of servers and cloud tenants, and hundreds of thousands of VMs.

II. On-demand access:
- Pay-as-you-go, no upfront commitment, access to anyone.

III. Data-intensive nature:
- What was MBs has now become TBs, PBs and XBs.
  - Daily logs, forensics, Web data, etc.

Map/Reduce in LISP

Sum of squares:

- (map square '(1 2 3 4)) — Unary operator
  - Output: (1 4 9 16)

[processes each record sequentially and independently]

- (reduce + 0 '(1 4 9 16)) — Binary operator
  - (+ 16 (+ 9 (+ 4 (+1 0))))
  - Output: 30

[processes set of *all records* in batches]

MapReduce

MapReduce Architecture

- *MapReduce programming abstraction:*
  - Easy to program distributed computing tasks.
- MapReduce programming abstraction offered by multiple open-source *application frameworks*:
  - Handle creation of "map" and "reduce" tasks.
  - *Hadoop: one of earliest map-reduce frameworks.*
  - *Spark: easier API and performance optimizations.*
- Application frameworks use *resource managers*.
  - Deal with hassle of distributed cluster management.
  - *e.g. Kubernetes, YARN, Mesos, etc.*

Slow Servers

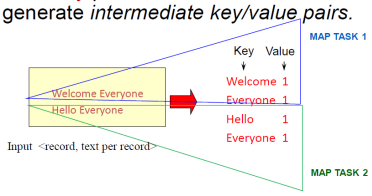
Barrier at the end of Map phase!

Slow tasks are called Stragglers.
- The slowest task slows the entire job down (why?)
- Due to bad disk, network bandwidth, CPU, or memory
- Keep track of "progress" of each task (% done)
- Perform proactive (replicated) execution of some straggler tasks
  - A task considered done when its first replica completes (other replicas can then be killed)
  - This approach is called Speculative Execution.
- Straggler mitigation has been a very active area of research.

- Input: a set of key/value pairs (record, list of words)
- User supplies two functions:
  - map(k,v) → list(k1,v1)
  - reduce(k1, list(v1)) → v2
- (k1,v1) is an intermediate key/value pair. (word, 1)
- Output is the set of (k1,v2) pairs. (word, count)

Map:

- Parallelly process individual records to generate *intermediate key/value pairs.*



Reduce:

- Each key assigned to one Reduce task.
- Parallelly process and merge all intermediate values partitioned per key.

- Popular: *Hash partitioning, i.e.*, key is assigned to
  - reduce # = hash(key)%number of reduce tasks

Key-value/NoSQL Data Model

- NoSQL = "Not Only SQL"
- Necessary API operations:
  get(key) and put(key, value)
  - And some extended operations, e.g., "CQL" in Cassandra key-value store
- Tables
  - Like RDBMS tables, but …
  - May be unstructured: May not have schemas
    - Some columns may be missing from some rows
  - Don't always support joins or have foreign keys
  - Can have index tables, just like RDBMSs

Partitioner

- The component responsible for a key to server mapping (a hash function).
- Two types:
  - *Chord-like hash partitioning*
    - *Murmer3Partitioner* (default): uses *murmer3* hash function.
    - *RandomPartitioner*: uses MD5 hash function.
  - *ByteOrderedPartitioner*: Assigns ranges of keys to servers.
    - Easier for *range queries* (e.g., get me all twitter users starting with [a-d])
- Determines the primary replica for a key.

Writes at a replica node

On receiving a write

1. Log it in disk commit log (for failure recovery)
2. Make changes to corresponding Memtables
   - **Memtable** = In-memory representation of key-value pairs
   - Cache which can be searched by a key
   - Write-back cache as opposed to write-through
3. When Memtable is full or old, flush it to the disk
   - Data file: **SSTable** (Sorted String Table, list of key-value pairs, sorted by the key)
   - Index file: SSTable of (key, position in data SSTable) pairs
   - And a Bloom filter (for efficient search) – next slide.

Deletes

Delete: don't delete item right away
- Write a tombstone for the key.
- Eventually, when compaction encounters the tombstone, it will delete the item

Reads

- Coordinator contacts X replicas (e.g., in same rack)
  - Coordinator sends read to replicas that have responded quickest in the past.
  - When X replicas respond, Coordinator returns the latest-timestamped value from among those.
  - X = based on consistency spectrum (more later).
- Coordinator also fetches value from other replicas
  - Checks consistency in background, and initiate a read repair if any two values are different.
  - This seeks to eventually bring all replicas up to date.
- At a replica
  - Read looks at Memtables first, then at SSTables.
  - A row may be split across multiple SSTables => reads need to touch multiple SSTables (reads slower than writes, but still fast)

CAP Theorem

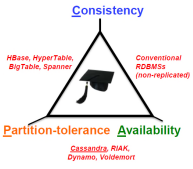
- Consistency: reads return latest written value by any client (all nodes see same data at any time).
- Availability: every request received by a non-failing node must result in a (quick) response
- Partition-tolerance: the system continues to work in spite of network partitions.
- In any distributed system, we can only guarantee at most 2 out of the above 3 properties.
  - Proposed by Eric Brewer (UC Berkeley)
  - Subsequently proved by Gilbert and Lynch (NUS and MIT)

CAP Theorem

- Data replicated across both networks N1 and N2.
- If network is partitioned, N1 can no longer talk to N2.
- Consistency + availability require that N1 and N2 can talk.
  - no partition-tolerance.
- Partition-tolerance + consistency:
  - only respond to requests at N1 (no availability at N2).
- Partition-tolerance + availability:
  - writes at N1 will not be captured by reads at N2 (no consistency).

CAP Tradeoff

- Starting point for NoSQL Revolution
- A distributed storage system can achieve at most two of C-A-P.
- When partition-tolerance is important, you have to choose between consistency and availability.
- Check *Spanner CAP-theorem* white paper on Blackboard



Cassandra write and read recap

- Writes
  - Client sends write request to *Coordinator*.
  - Coordinator writes to all replicas.
  - Waits for X replicas to respond before returning acknowledgement to the client.
  - Hinted handoff: if a replica is down, it receives the write request once it comes back up.
- Reads
  - Client sends read request to *Coordinator*.
  - Coordinator contacts X replicas, and returns the latest returned value.
  - Read repair: After returning a response, Coordinator continues with fetching values from other replicas, and initiates repairs to outdated values.