9/28/17

disorderly labs.github.io

Distributed Systems → fundamentally characterized by uncertainty.

☒ DISTRIBUTED SYSTEM:

WHAT

* A bunch of computers Asynchronously communication → cannot make assumptions on upper limit of message time.
* Many Computers appearing as one.

really hard to detect failed connection on an asynchronously distributed System.

* Parts are communicating over an inconsistent network.

WHY

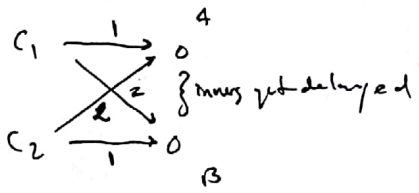
* REDUCE LATENCY (due to locality) – replication redundancy
* Scaleability
* Alternate routes – multiple choice Same task redundancy

☒ ARE NON-DETERMINISTIC. uncertain.

non-determinism & delivery order.

Uncertainty

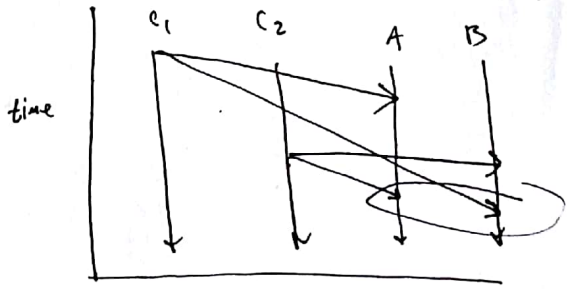
1. Concurrent / Asynchrony
   ↳ Message ordering.



If B is a replica of A.
And some delay occurs,
A & B will have input different messages.

2. FAILURE (PARTIAL FAILURE)
   ↳ components can fail, without entire system failing.

SPACE TIME DIAGRAM (LAMPORT)



PHYSICAL SPACE

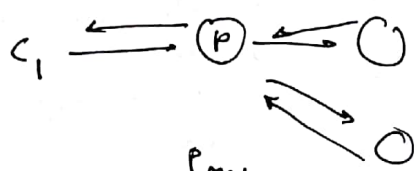
THEME 1 : time & Asynchrony.

no total Order in OS.

Partial Orders

reasoning about time & Partial Orders.

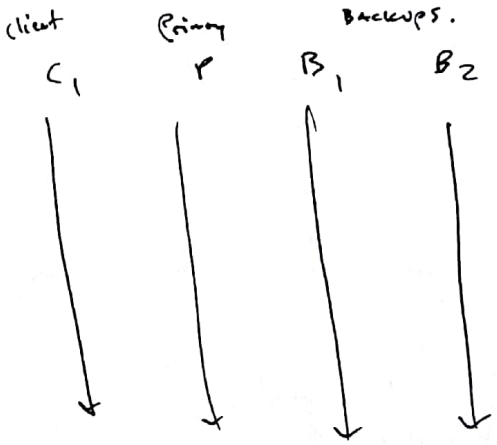
THEME 2 : FAULT TOLERANCE



P makes replicas & waits until they signal they've stored data to respond to client.

Big concern is with primary failing.
client thinks success, no data stored.

THEME 3 : Consistency.
↳ IF I can trick you into thinking I'm not a collection of computers, I'm strongly consistent.



THEME 4: Scaleout & Parallelism.

10/3/17

TIME, CAUSALITY, "HAPPENS BEFORE" RELATION, LOGICAL CLOCKS, PARTIAL ORDERS.

come up w/ QUESTIONS ABOUT RESEARCH PAPER.

① ②

DS?

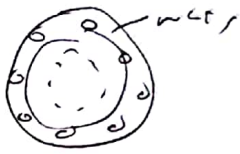
1. Scale-up / Scale-out } most DS want both.
2. FAULT-TOLERANCE } REPLICATION
3. REDUCED LATENCY → LOCALITY ...

↳ Extrinsically Distributed Systems.
- Nothing about algorithm is Distributed.
- Mass is in center, pretending to be a single system.

- correctness = transparent distribution.

→ LESS-COMMON, MORE CUTTING EDGE
- Intrinsically Distributed.

p2p systems.
→ Mass is on edge

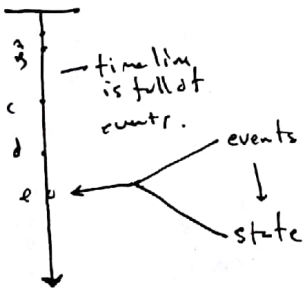


→ IOT.

need to invent models of correctness.

LAMPORT DIAGRAMS

TIME:



e is the set (A, B, C, D)

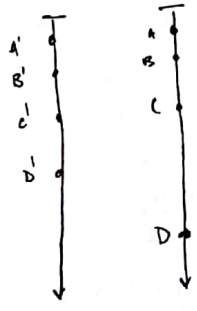
arguing events, state are the same.

- state is a ~~glossy~~ lossy representation by a chain of events.

USUALLY THE CASE, given a history (set of events) we can figure out state.

(a particular time is the same as history of previously occured events.
↳ can't go state → event.

causal ordering — accounting for time —

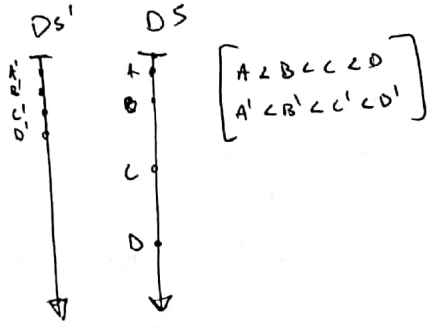


- can't distinguish lines, due to fact that the same events occured on each line.
- consequence of Asynchronous Network Model.

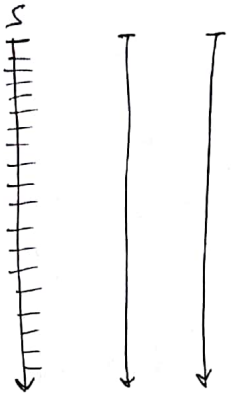
NETWORK MODELS

Synchronous — upper bound on the latency of computation & communication.
— can only take so long: for message delivery, task completion ...

Asynchronous — No bounds

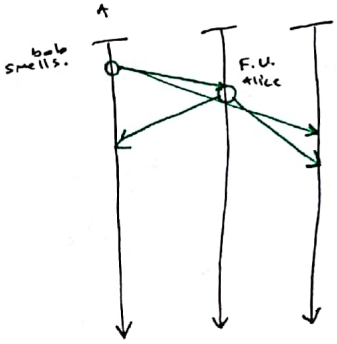


Same Results.
Ds' gets job done faster.

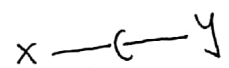


[what is order for & why do we need it?]

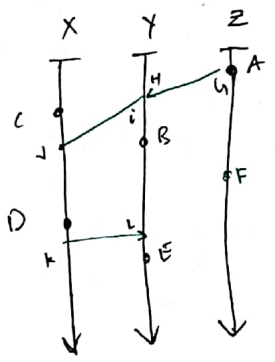
Consistency
- exhibits behaviors that look the same way.
- implies something about anomalies that do not occur.



CAUSALITY



Potential Causality



"HAPPENS BEFORE" RELATION

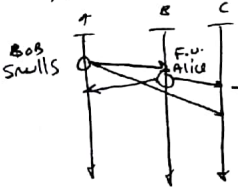
- Event B CAUSED A?
- B could've caused E
- A can't cause an event
  → nothing is reachable from *.

E is SET of EVENTS {A < B < C < B}

→ B → E | G → H, K → L
C → D | I → J, A → D
J → K

CAUSALITY IS REACHABILITY IN SPACE-TIME.

Alice broadcasts message



— carol gets f.u. alice before bob smells...

look no further
messages entangle causality.

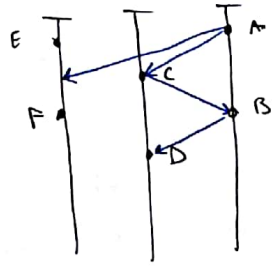
"Happens Before" → CAUSALITY

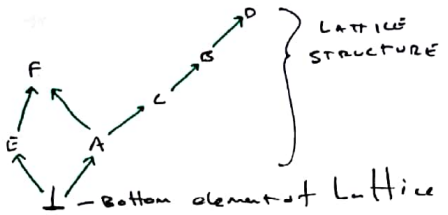
given two events, (a, b)
a happened before b if any of following are true. LAMPORT

① a could've caused b if (a, b) occur in same process & a occurs first on a single process line.
② if a is a send event & B is corresponding Receive event
   a happened before b.
③ if there exists a C, such that a happen before C & C happened before b
   then a happened before b. ∃C s.t. a → C ∧ C → B we say a → B.

⑤

PARTIAL ORDERS





1. TRANSITIVE $((a \to B, B \to c) \Rightarrow (a \to c)$
2. ~~irreflexive~~ Asymmetric $(a \to b) \Rightarrow (b \not\to a)$
3. ~~Asymmetric~~ irreflexive $\neg a \to a$

$\bot$ – Bottom element of Lattice.

↳ bot. element/child of lattice such that no element is less than.

↳ Diagram to enter all ordered pairs.

TOTAL ORDERING

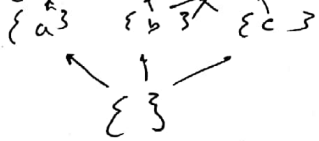
PARTIAL ORDERING w/ EXTRA RULE

- for all (a, b) $\quad a \to b \vee b \to a$

PARTIAL ORDERING: (subset example)

$\{A, B, C\}$ consider all subsets of powerset (A, B, C)

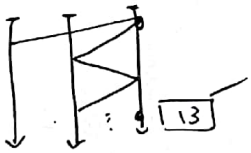
$\{B\}$ is smaller than $\{B, C\}$ because b is contained is set $\{B, C\}$



elements are comparable, if a path through the lattice exists.

Logical Clock

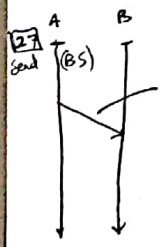
1) Lamport clock
2) Vector clock



abstract accounting of events.

A) Mutable **state** containing clock value.
e.g. lamport clock → integer
vector of integers.

B) **Interpose** – get in between – on communication, such that, all messages contain clock value.



message sent contains payload & metadata (clock) ((current estimate # relative events)

(BS, 27)

C) SET OF RULES DESCRIBING HOW & WHEN TO MODIFY CLOCK.

10/5/17

1. $\neg(a \rightarrow a)$
2. $(a \rightarrow b) \not\Rightarrow b \not\rightarrow a$
3. $a \rightarrow b \wedge b \rightarrow c \Rightarrow a \rightarrow c$
4. $\forall a, b,\ a \rightarrow b$ or $b \rightarrow a$

PARTIAL ORDER — TOTAL ORDER

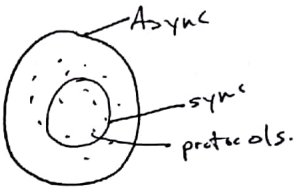
→ BEST WE CAN GET IN D.S.

MODELS

1. Sync → CPUs CAN'T BE ARBITRARILY SLOW
2. Async →

proofs

1. existence / constructive / correctness
2. Impossibility.



strongest correctness proof → Async Model proves Sync model. → Proves so in Async world, assuming worst-case.

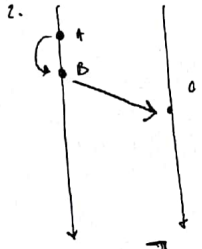
prove impossibility. → do so in a best case scenario

HAPPENS BEFORE A "happened before" B = $A \rightarrow B$

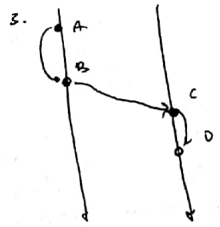
$1 \rightarrow 2,\ 2 \rightarrow 3,\ 1 \rightarrow 3,\ 2 \rightarrow 4,\ 1 \rightarrow 4$

1. A, B in same process, A happens B before B.

2.



3.



not transitive!

CONCURRENCY

if $a \not\Rightarrow b \wedge b \not\rightarrow a$

then $a \| b$

a is concurrent to b

"no causal relation"

- when two events are concurrent, they can float down lines freely.

if $a \| b \wedge b \| c \not\Rightarrow a \| c$

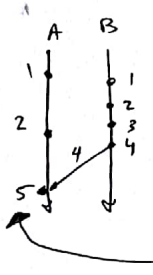
"Causality is reachability in space time"

Cause of 4? → dots that can reach 4.

LOGICAL CLOCKS

LAMPORT CLOCKS → consistent with causality.

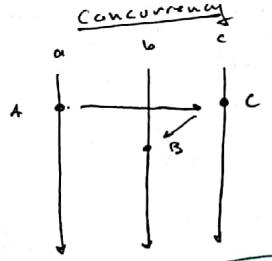
- Associates events with integers
- gives following property: if $a \Rightarrow b$ then $L(a) < L(b)$ (a happened before b)
- not strong enough to prove causality.
- can only check if b did not happen before A.

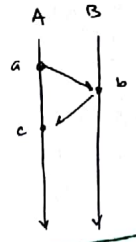


LAMPORT CLOCK RULES

1. EVERY PROCESS MAINTAINS AN INTEGER COUNTER INITIALIZED TO 0.
2. LOCAL EVENTS INCREMENT PROCESS' COUNTER.
3. PROCESSES ATTACH COUNTER TO EVERY MESSAGE.
4. WHEN PROCESSES RECEIVE A MESSAGE, THEY UPDATE THEIR COUNTER TO MAX (local counter, msg counter) + 1

CONCURRENCY





"consistent with causality"

compare : $<$ over integers

merge : MAX ( ) over integers

VECTOR CLOCKS

$a \rightarrow b \iff V(a) < V(b)$

↳ stronger than lamport clock

"characterize causality"

equivalent to happens before.

1) $a \rightarrow b$
2) $b \rightarrow a$
3) $a \parallel b$

compare: $VC(a) < VC(b)$ if $\forall i: VC(a_i) <= VC(b_i)$ every element of b is greater than or equal to element in a & exists j s.t. $VC(a_j) < VC(b_j)$

3 numbers

[A B C]

↳ represents (integer value) events committed on each timeline.

Compare

[1 7 2] < [2 7 2]

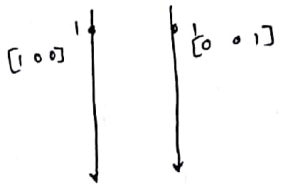
later in "causal time"

> [0 6 1]

|| [2 7 1]

$VC(a) < VC(b)$ iff $\forall i: VC(a_i) <= VC(b_i)$

$\exists_j : VC(a_j) < VC(b_j)$



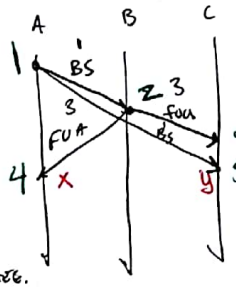
Merge

ELEMENT-WISE MAX.

VC

1. every vector is a vector of ints
2. increment "our" integer
3. PROCESSES ATTACH COUNTER TO EVERY MESSAGE
4. ELEMENT-WISE MAX FOR UPDATING

LC

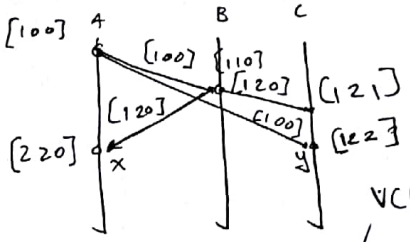


2 is the event bob learning of event from Alice.

$\max(1,4)+1 = 5$

$x \parallel y$

VC



$VC(x) \parallel VC(y)$

↳ $x \parallel y$ characterise causality

GREAT GUARANTEE.

CAUSAL TOTAL ORDER

↑

FIFO TOTAL ORDER

↑

TOTAL ORDER

DELIVERY PROTOCOLS

IF A SENT M_1 before M_2

B DELIVERS M_1 before M_2

delays 2 until safe.

FIFO Anomalies

CAUSAL D

IF $A \rightarrow B$

A IS DELIVERED BEFORE B.

Any FIFO violation is a violation of causal, but no opposite.

CAUSAL is STRONGER THAN FIFO.

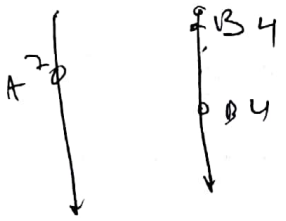
10/12/17

CAUSALITY -

Any pair of events are comparable.

$C_a = 7$, $C_b = 4$

$a \not\rightarrow b$ a doesn't happen before b is only thing we can conclude.



ORDERING GUARANTEES (LATTICE)

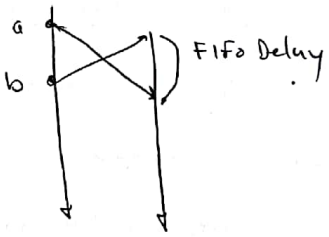
FIFO - network acts as a queue

 one-to-one

 broadcast to many

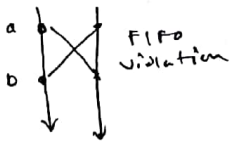
 many - to - one

 many - to - many

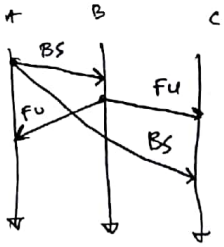


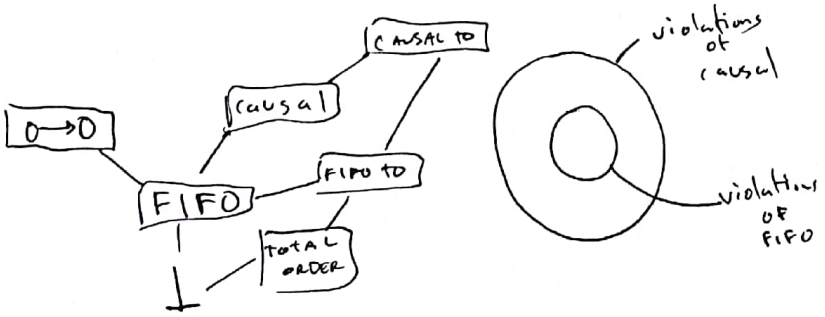
CAUSAL = FIFO (violation)

↳ violations of causal ~~are~~ that are not violations of FIFO, because causal is stronger

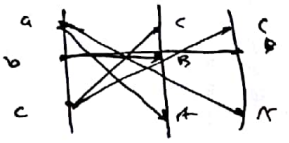


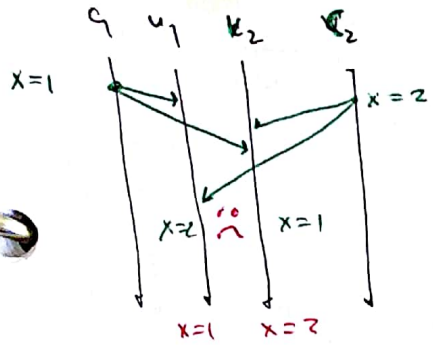
causal violation





TOTAL ORDER





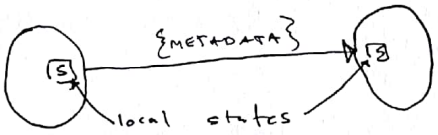
should be.
So, this a violation of total order.

not causal TO violation, because a & b are concurrent.
it a happened before b
deliver a before b.

DELIVERY PROTOCOLS

- A SET OF RULES
  - → COMMUNICATION → NEED MORE THAN 1 PERSON TO NEED A PROTOCOL.
  - → CONTRACT BETWEEN COMMUNICATING PROCESSES.

PAIR OF COMM. ENTITIES



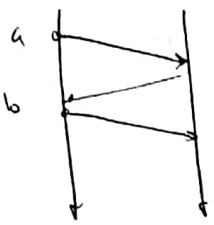
DELIVERY ORDER PROTOCOLS

1. LOCAL STATE.
2. RULES FOR SENDING MESSAGES
3. RULES FOR RECEIVING MESSAGES
4. WHEN IT'S OK TO DELIVER A MESSAGE
   - ↳ MIGHT RECEIVE MESSAGE, BUT DELIVERY CAN BE DELAYED.

IN FACT, THIS IS ALL D.O.P's DO.

(13)

blocking Sends implementation.



don't send b until ack, a has been received.

autonomous vehicles comm. via Radio.

non-blocking algorithm

- can send messages w/o waiting
- RECEIVER SORTS OUT FIFO.
- Senders have counter
  - b included
- receivers have counter for every sender
  - delivers message if it is expected
- eventually all messages are fifo delivered.

causal

sending - everything appended in thread.

State:

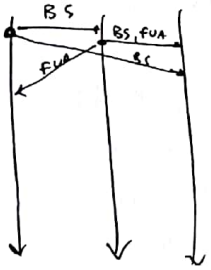
Sender: Every Message I have ever sent

receiver: ∅

~~Others~~:

rules: ① When sending a message b attach all messages a, such that $a \rightarrow b$.

②

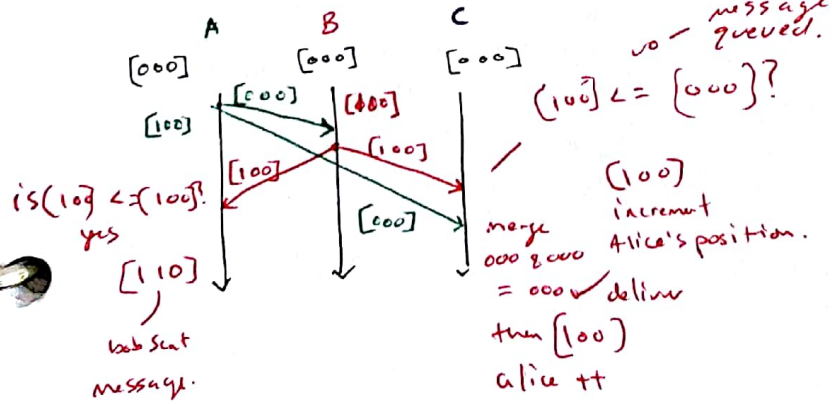


uniquely identify inputs & remember.

BROADCAST REGIME

Receive.	BROADCAST.
Blocking	Causal ~~Delivery~~
using Vector clocks.	

Vector Clocks characterise causality.



n positions in V.C.

state:

Sender: V.C.

receiver: V.C., queue

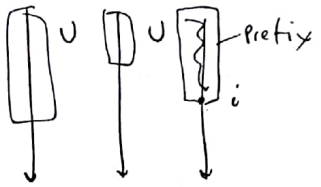
RULES:

1. after message send, increment my position in V.C.
2. on receipt, merge V.C. w/ local V.C., then increment senders position.
3. on send, include local V.C.
4. Deliver message with V.C. v WHEN v <= local V.C.

10/19/17

Omission model
- Dropped messages act as crash

CONSISTENT GLOBAL SNAPSHOTS
- Local Snapshot → all previous events leading up to event. (prefix) — set of points
  - its point is completely characterized by events before.
- GLOBAL SNAPSHOT → UNION OF SET OF LOCAL SNAPSHOTS
  - if $a \to b$ & $b \in \zeta$ $\Rightarrow a \in \zeta$



ALGORITHM FOR OBTAINING GLOBAL SNAPSHOT

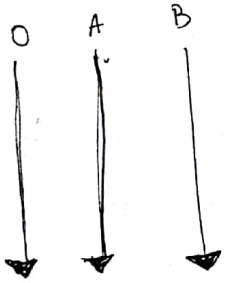
Chandy / Lamport Snapshots

not reasonable

ASSUMPTIONS
1. No faults
2. FIFO delivery
3. All-to-all communication

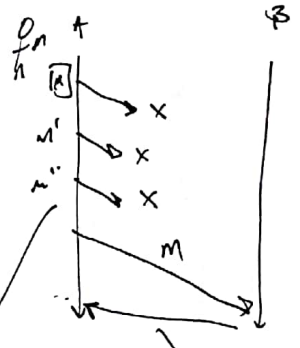
Can an external witness

1. An observer process sends a token to every other process
2. When a process receives a token, it records its local state, & sends to observer, then sends a token to all processes (for the first time), & begins recording in-bound messages.



send infinitely many times.
Reliable Delivery
- omission model

Reliable Broadcast
- crash model
- upgrade to omission model



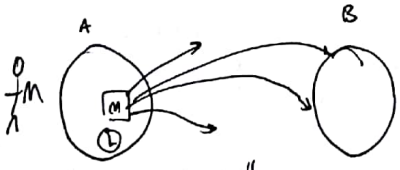
each gets its own buffer

infinite state & behavior

once message B → A saying M was received, A's local state changes.

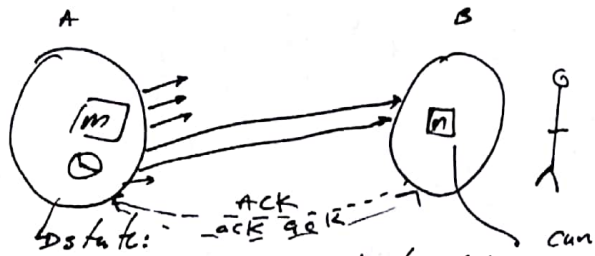
Note: everyone sends and receives a token from everyone.

Token always pushes relevant messages in snapshot.



put message in buffer — stimulates redundant behavior

we have a clock



fm

Dstate: can have buffer on receiving end.
once ack was implemented
we got rid of state & could
also buffer.
now, ack ack clears up B's state.

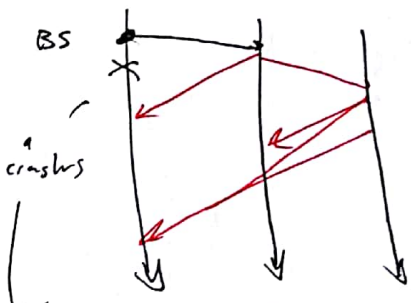
eventually, if not ever, if not all messgs were dropped, RD will work.

'Exactly once' delivery algorithm/process

2^{64} bit number for sequencing is how TCP handles two generals

Reliable broadcast - crash model

All-to-All reliable broadcast for crash model



how to fix it?

if a correct process delivers a message, then all processes deliver it.

if A then B $A \Rightarrow B$

$\neg A$

$A \wedge B$

↳ when a process receives a message, it relays the message to every process it can touch.

chunky / unport.

Independent to each other WRT FAILURE

FAULT TOLERANT Protocols

- [x] FAULT TOLERANCE IS REDUNDANCY

~~Alt~~

SOME FAULTS WON'T AFFECT ALL RESULTS

- ASSUME INDEPENDENCE
- REDUNDANT BEHAVIOR, COMPUTATION, STATE, etc.

TIME: RETRY

SPACE: REPLICATION

MATH: Error-Correcting Codes.

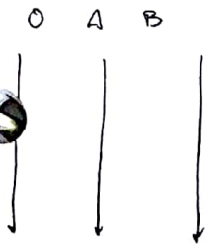
Representation: DATA PAGES / write ahead log / SHADOW PAGES.

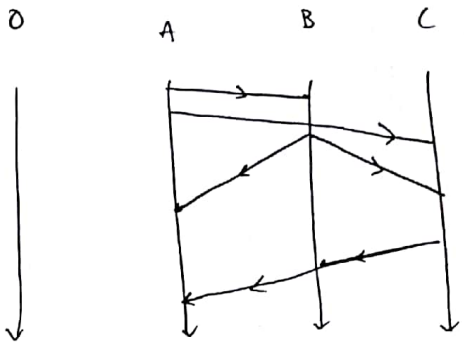
Program: N-version Programming ⟶ LIFE CRITICAL SYSTEM.

Two copies of state asynchronously updated ~~change~~ raise caution about synchronization & which data values are correct..

10/19/17

SECTION

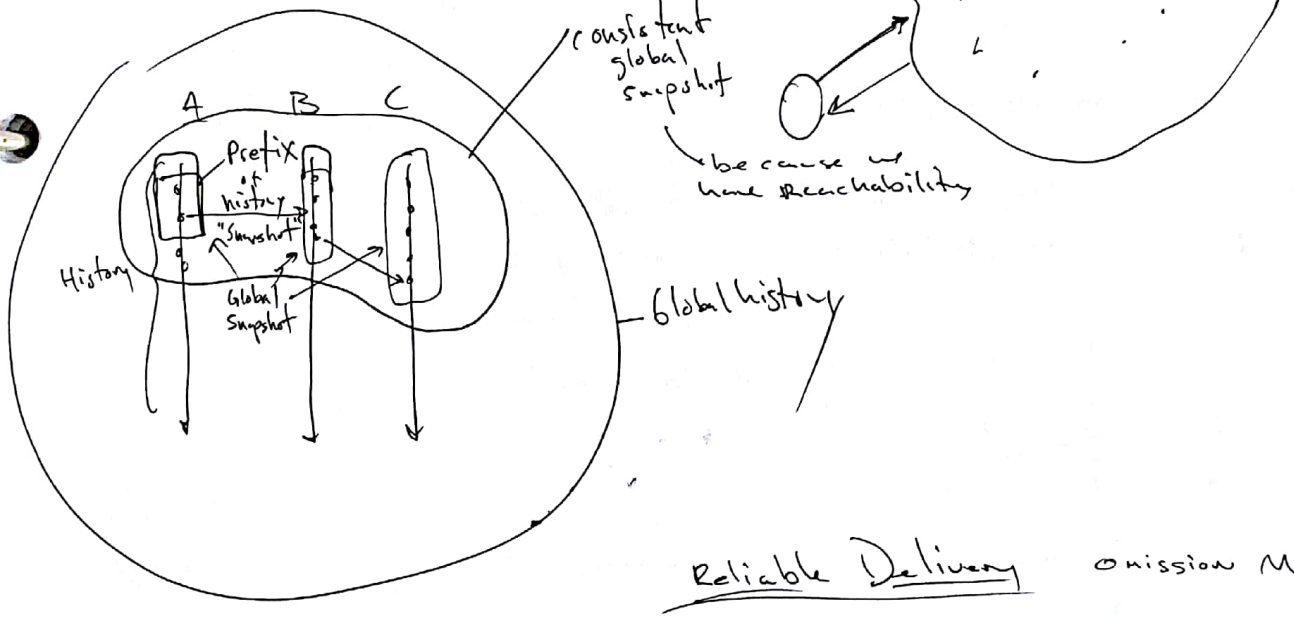




each has a single queue for each process other than itself.

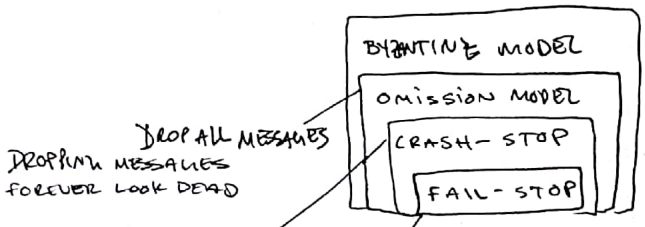
VC characterize causality

History
global history



Reliable Delivery omission Model

FAIL-STOP MODEL



could be messag takes too long or is lost
don't know they die

know process failed
know they die

DELIVERY : eat cake

SEND : Pass cake

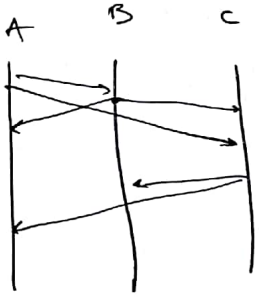
USE RELIABILITY DELIVERY

RELIABLE BROADCAST

if one correct process delivers, ~~all do~~. will send message.

Send first, then open

Short ~~if~~ ask?'s later.



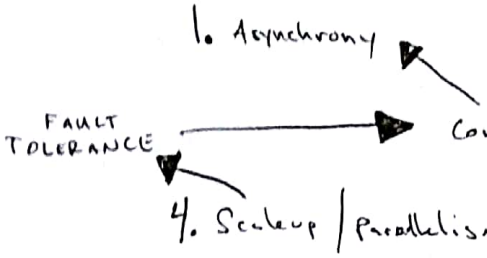
DYNAMO

FAULT-TOLERANCE is ∈ REDUNDANCY.

Time Partial-failure in space

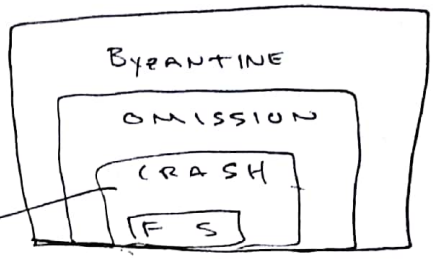
Asynchrony - message doesn't arrive "in time" - keep trying

10/24/17



replicas see messages in non-deterministic orders.

DASHBOARD OF ALL THINGS THAT CAN GO WRONG IN D.S.

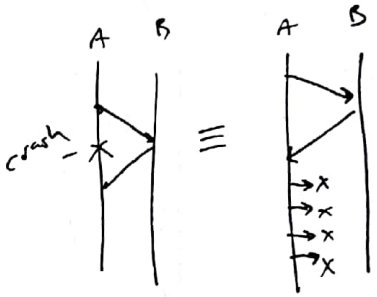


$p_{fail} = \frac{1}{1000}$

if we scaleup to 1,000 computers we start to approach the apex of the parabola.

MESSAGES DON'T GET DROPPED, SERVERS FAIL.

- CAN BE MODELED BY SUBTRACTING EVENTS.
- REDUNDANCY IN TIME OVERCOMES OMISSION.



$1000p = 1 = fail$

well, if we have 3 cpu's acting in a O.S. as / cpu.

we get $\underbrace{3000}_{\text{linear scale}} \cdot \underbrace{p^3}_{\text{by exponentially better probability of failure.}} =$

- if it doesn't pass the fluffy higher model, it won't pass smaller ones.

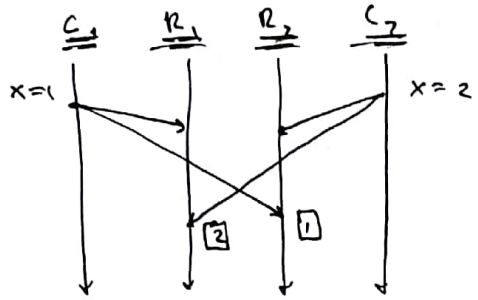
Replicate data & computation

we want to go fast (scaleup), but we also want to go (fault tolerance) so we make replicas (consistency), but we need to make sure they're in sync, due to unordered Network (Async).

simplest.

Primary	Backup

– CAUSAL DELIVERY USING VECTOR CLOCKS.



- Don't permit clients to query backups
- don't ack client until all backups ack.

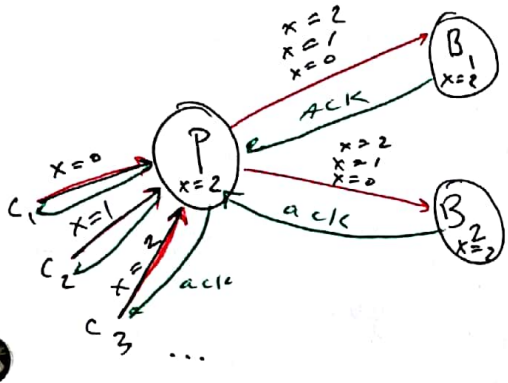
$R_1 \; != \; R_2$ → Inconsistent.

PROBLEM: CAUSAL ORDER NOT TOTAL.

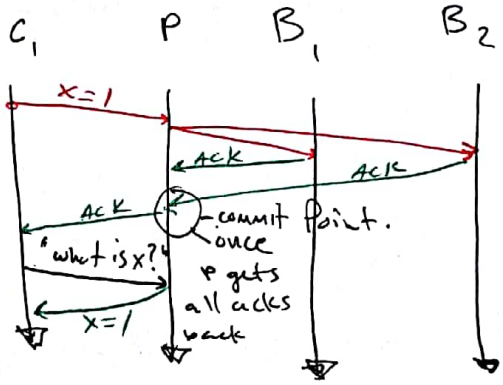
ESTABLISH TOTAL ORDER

Primary – LEADER (Computer)

Backup – followers



AFTER (P) receives Ack from ALL! backups, it will ack client.

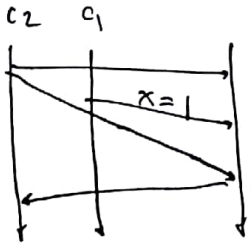


pull primary to get value of x.

Strong Consistency

There is no way to tell the system is distributed.

→ REPLICATED SYSTEM is S. Con. if there can be no witness of the fact it's replicated.



$P_{(x=0)}$

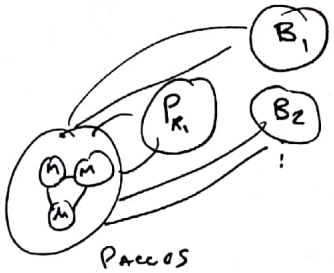
1) x=1
   x=1

2) x=0
   x=1

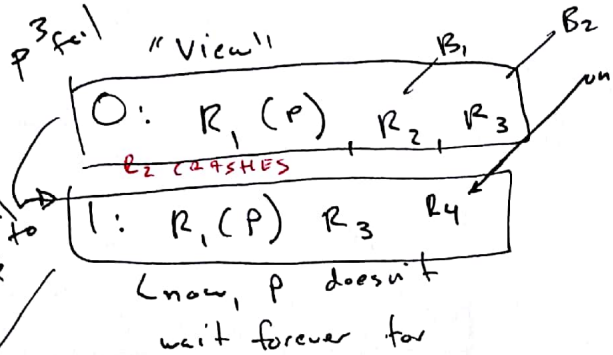
FAULT-TOLERANCE

cluster of computers → Configuration Manager

cluster of C.M. circle, talking using consensus algorithm.



Paxos
raft
zab.
Zookeeper nodes.



know, P doesn't wait forever for crashed replica.

BACKUP CRASHES

CM. has not heard back after reasonable amt. of Time.

Set of views atomically propagated to all nodes/replicas.

what happens when we crash?

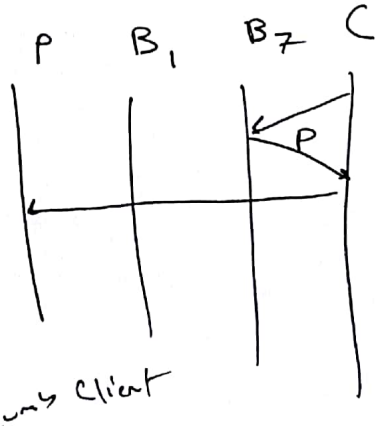
want to keep $P_{fail} = P^3$.

do this by promoting already existing replica node waiting for a time like this.

→ note: it's uninitialized, needs to be demarshalled.

R_4

2: R_3 (P) R_4 R_5 — view change needs to be visible to clients as well

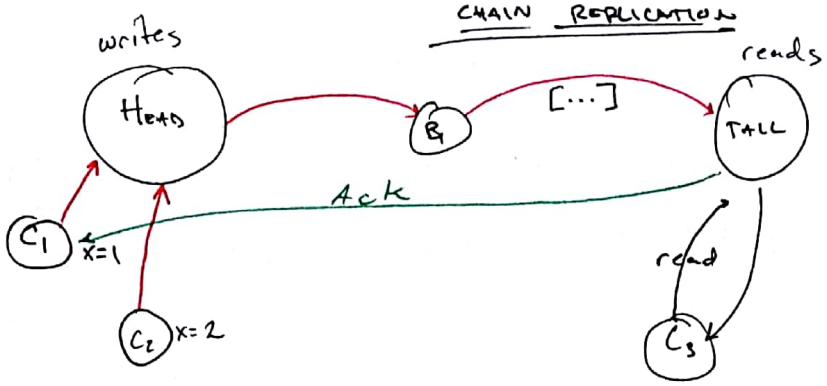


dumb Client

Backup can proxy

smart Backup says — talk to P.

CHAIN REPLICATION



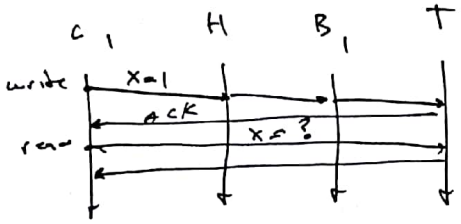
last value is newest value of x.

WRITE

- DATA ISN'T REPLICATED UNTIL DATA HITS TAIL.
- TAIL RESPONDS TO C_1, C_2...
- need to anticipate being acked from tail.

READ

- if client wants to read.
→ read from tail.



LATENCY — client perceived latency is king

- Primary / Backup — Max Round trip time to the (slowest) backup.
- Chain Replication — Sum of latency of RTT's among nodes in chain.
  - → better network utilization
  - → less work, Primary doesn't handle dissemination. or does.
  - → Pipeline writes
  - → shed the load of read to tail

more bytes through network.

MEASURING FAULT-TOLERANCE

RELIABILITY — likelihood it will be functioning at a given time. (likelihood of a component)

AVAILABILITY — likelihood a system is functioning correctly at any time.

$$A = \frac{\text{uptime}}{(\text{uptime} + \text{downtime})}$$

uptime — how long ran before crash

.9999

4 nines of available.

MTBF - MEAN TIME BETWEEN FAULTS

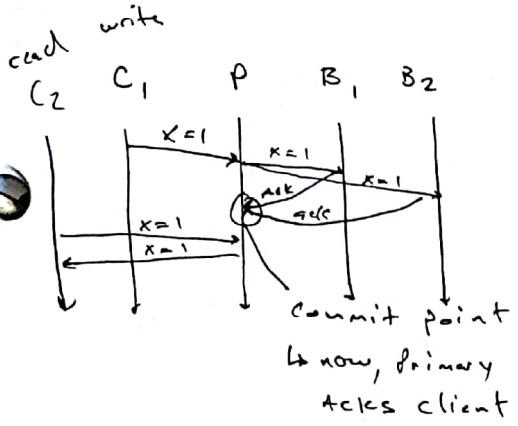
$$A = \frac{MTBF}{MTBF + MTTR}$$

MTTR - MEAN TIME TO RECOVER

↳ cut this, increases availability.

$$\left(\frac{1}{1 + \frac{MTTR}{MTBF}}\right)$$

10/26/17



latency = max latency edge.

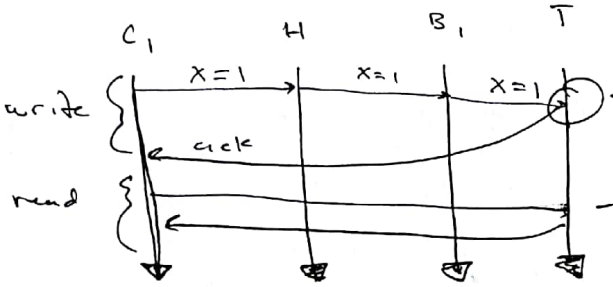
replicated systems

- Primary Backup
  - ↳ decides order based on arrival time
- Total order over operations

CHAIN REPLICATION

latency = sum of edges.

when C_1 writes x=1 to H, H immediately passes ~~not~~ message to b_1, b_2 ... & that ↳ acks C_1.

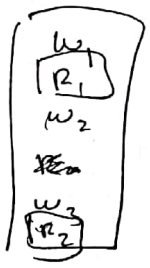


Total order is Madden

↳ Head chooses order based off delivery order of writes

tail

↳ chooses order of reads.



DELIVERY PROTOCOLS – Correctness Properties

Safety & Liveness

any correctness property is one and or both.

SAFETY: "A bad thing never happens"

always have finite counter examples.

Liveness: "a good thing eventually happens"

infinite execution.

(if you can draw it, safety property)

SAFETY — A B S — LIVELESS

CAUSAL DELIVERY – bob smells

FIFO

RELIABLE DELIVERY

R.D. eventually a message will get delivered

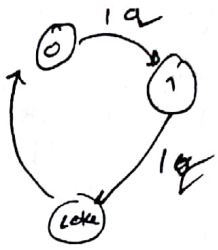
Dynamo paper

SMR — STATE MACHINE REPLICATION

1. STATE MACHINE
   - Processing is deterministic

$f : (S \times I) \rightarrow S'$

S = states
I = input

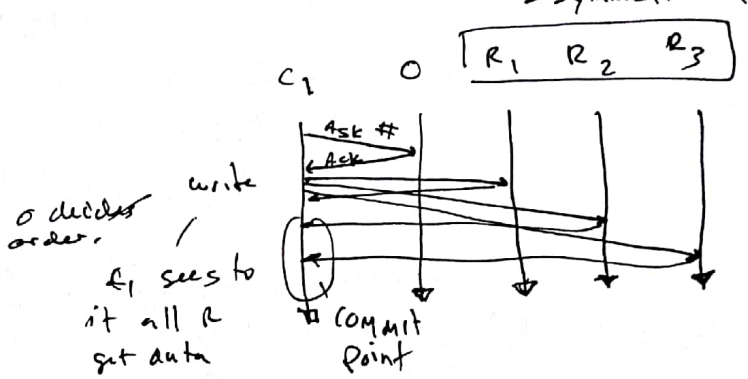


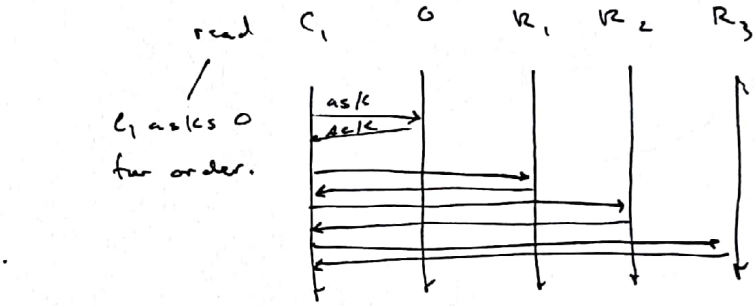
2. Assume Ordering Oracle

client asks O for ordering,

client plays Role as primary, SMR ASSUMES SMART CLIENT

client disseminates Data.





Implement O via consensus.

token from O to C_1 gets attached to messages for everyone to compare.

QUORUM REPLICATIONS

1) • clients send operations to a set of replica.

Note: Set is not necessarily all replicas.

2) Three key Parameters

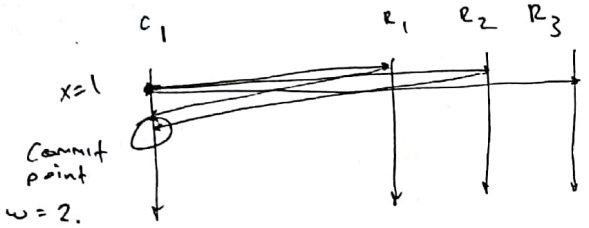
FAULT TOLERANT

N - number of replicas in system

W - number of acknowledgments required for a write

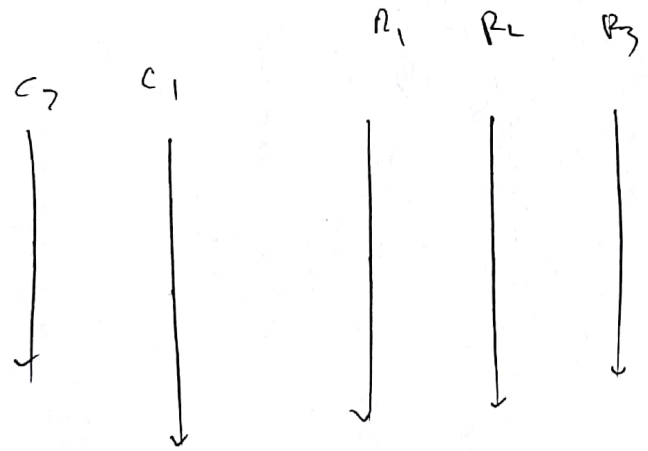
R - number of Acks " " read

ex: <N, W, R> = <3, 2, 2> - FASTEST REPLICA Determins latency



Strongly Consistent

$$W = \left(\frac{R}{2} + 1\right)$$



$\frac{n}{2} + 1$

N = 3

1. R = 3 W = 1
2. R = 1 W = 3
3. R = 1 W = 2

ROWA read once write all good for read heavy work load but, not FT wrt writes

Quorum Restraints

1. Every read quorum should intersect with every write quorum
   $R + W > N$
2. Every 2 write quorums must intersect
   $W > \frac{N}{2}$

Equates to No concurrent writes.

CAP Theorem

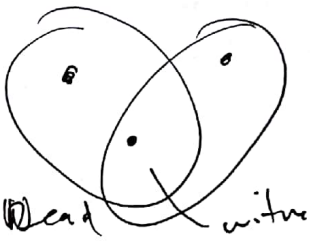
Consistency - Single copy consistency

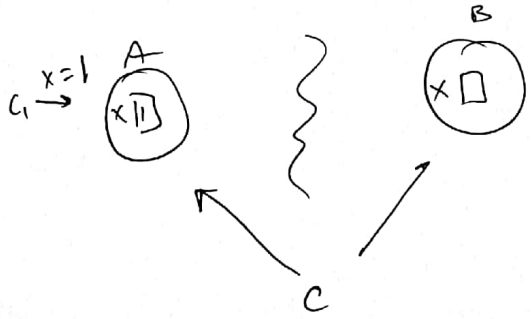
R=1
W=2 **A**vailable - as long as a node exists, it should provide a meaningful response

Partition tolerant

pick 2 out of 3

every D.S. is CP or AP. No D.S. is C.A.P.





if C queries B and B responds I don't Know. CP.

Satellite - AP

Any situation where we have majority quorums → not ~~CP~~. partition tolerant

Google

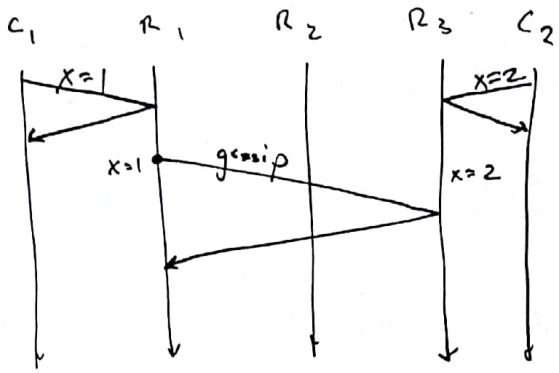
total order → Dynamically decided by execution

by node/nodes in intersection

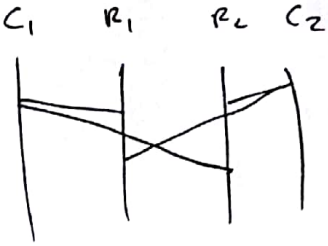
floats around, latency dependent

10/01/12

Inconsistent quorum Replication



W=1
R=1
FAST!



Total order anomaly
FIFO CORRECT

need Vector Clocks to handle gossip protocol.

can't rely on clients to follow protocol.

if disjoint,
merge
if not,
~~to~~pic breaker

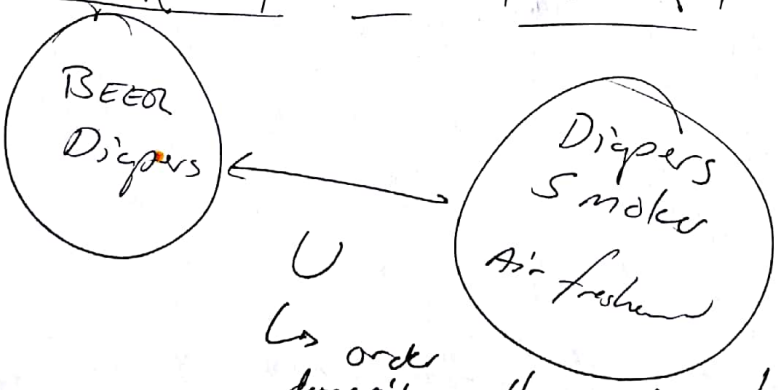
gossip
- PAIRWISE
↳ Anti-entropy
~~READ REPAIR~~
↳ Done by V.C.

concurrent
(data goes both ways

~~not~~ one process has a lot of data over the other.
(data goes one way

$f(x) = ff x = fff x$
↳ Idempotence
↳ can retry with impunity.

Shopping Cart Analogy



∪
↳ order doesn't matter due to commutative property of sets.

Set deletion doesn't have the ↗

10/31/12

Ex: PROTOCOL

P(I)

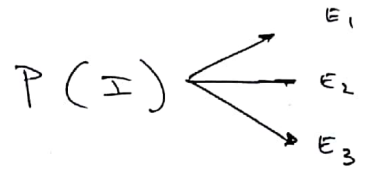
P/B

C_1 = X = 1

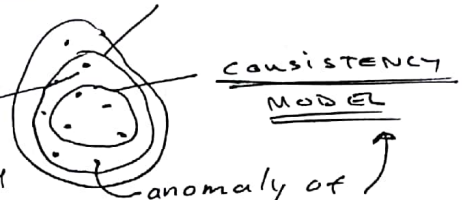
C_2 = X = 2

C_3 R(X)

a given protocol gives rise to set of EXECUTIONS



EXECUTIONS



One of the possibly executions.

anomaly of execution which is not permitted.

LC's ARE CONSISTENT WITH CAUSALITY

$a \rightarrow b \quad LC(a) < LC(b)$

$b \not\rightarrow a$

VC's characterize causality "Potential"

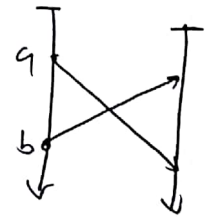
$a \rightarrow b \iff VC(a) < VC(b)$ why?

or

$b \rightarrow a$

$a \,||\, b$

FIFO



a → b

but b is delivered

FIFO total

FIFO

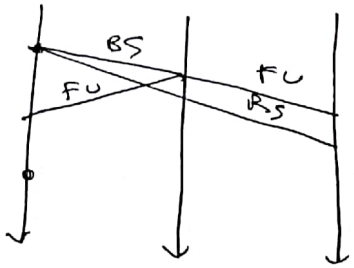
CAUSAL

CAUSAL TOTAL

Anomaly

CAUSAL

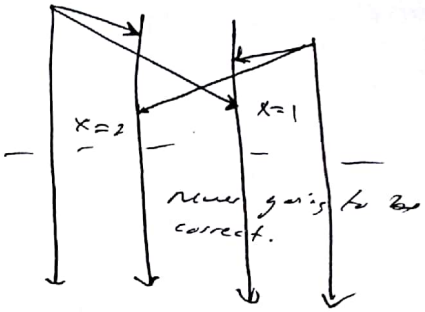
Bob Smells



CAUSAL violation

not FIFO

TOTAL



Split brain

T.O. violation

FIFO allows more executions than FIFO anomaly is ~~a~~ a causal anomaly because FIFO has all executions causal has, plus more

17/7/17

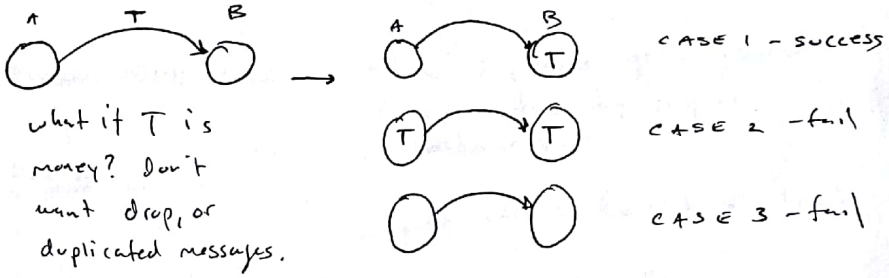
Agreement Protocols

POWERFUL PRIMITIVE

- Consensus
- Atomic Commit
- Leader Election
- Totally-ordered broadcast
- Two-phase commit

SAME problem

1. Two generals problem
2. Leader Election → failed primary, need to boost a new node
3. Flight control
4. Deciding a total order
5. Choosing a president
6. Failure detection
7. Group membership → configuration manager
8. Atomic changes across multiple nodes



what if T is money? don't want drop, or duplicated messages.

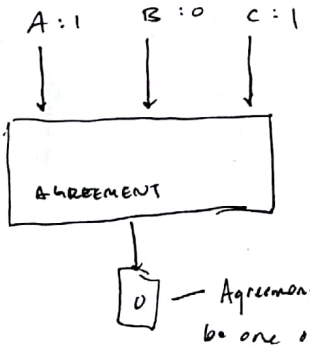
Agreement

Validity (Safety) → "If a node decides x, then x was proposed"

Agreement (safety) → "If a node decides x, then all nodes decide x" ← most important

Termination (Liveness) → "If a node proposes x, eventually some value is decided"

NOTE: Validity is non-triviality.



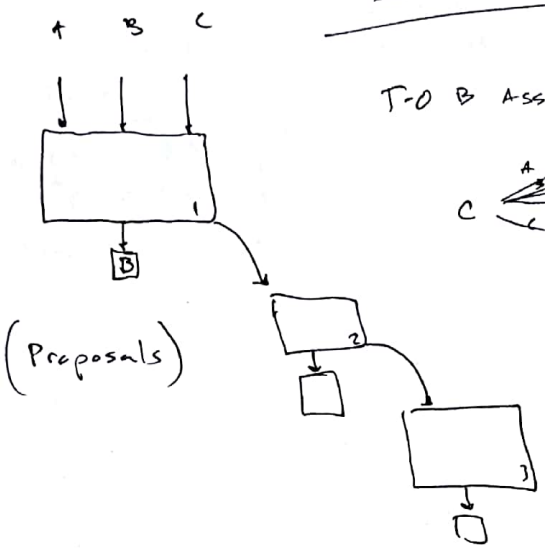
Agreement has to be one of the proposed messages.

- Consider 3 instances of the above, x, y, z. All 3 instances need to agree on the output.

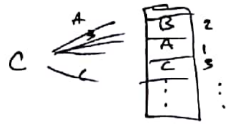
~~of one~~

. Similar to CAP theorem, we typically choose two. Validity & Agreement.

Totally-Ordered Broadcast (A.K.A. Consensus)



T-O B Assigns messages to slots (from client)



- USE CONSENSUS AS MANY TIMES AS THERE ARE SLOT.

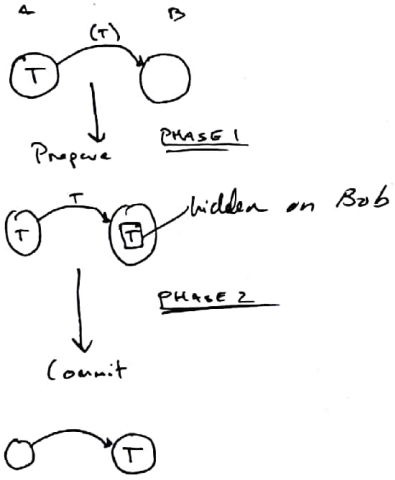
REDUCTION

- CREATE AN ALGORITHM that can solve A & Algo A to solve B & ALG B to solve A

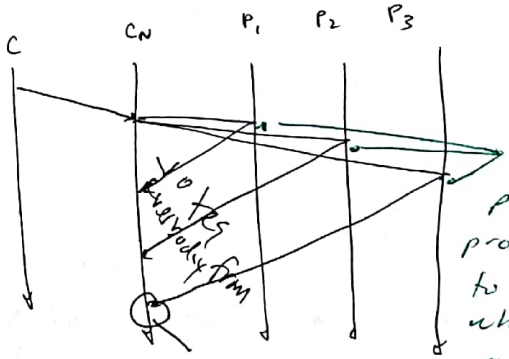
Two-Phase Commit

think 2 people getting wed & an authority saying I pronounce you...

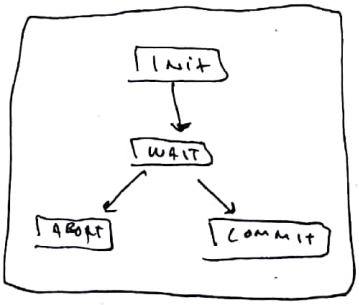
PROTOCOL:



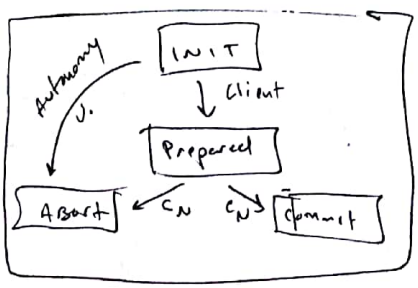
Coordinator node = C_N — has clock.

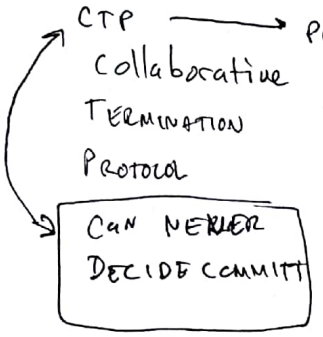


Now, C_N has to send Commit or abort.



- PARTICIPANT CAN go from prepare to abort.
- Commit gets sent on unanimous vote (or not) C_N chooses. It is solely about the participants being ready to commit after responding.
  - Seems like primary back up.
  - if a node crashes, we could abort.



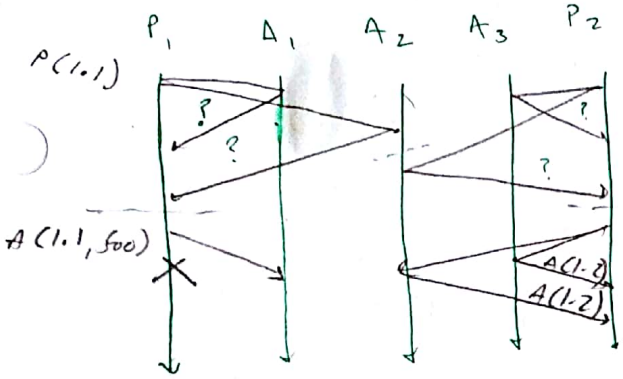


violation of agreement
if C_N writes to disk (abort)

Pick A node at random & Start to gossip, eventually system will be good.

~~if in abt~~

~~could → check if~~
rather than set value to none, lets keep a bool dict tracking if we can talk to to nodes or not...



- cut out prepare in multipaxos

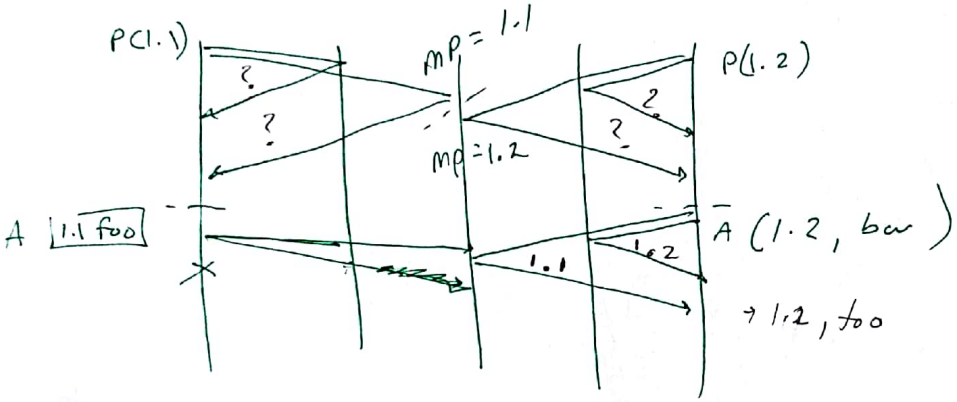
returns unconstrained value until Accept phase

—— it Returned min proposal 1.2, bar It matches our sent?

→ we're all good. bar was accepted

↳ else

→ send P (larger # than min-pro)



write-write conflict
2 xacts overwrite
same key

write-read: xact write variable that a later xact reads

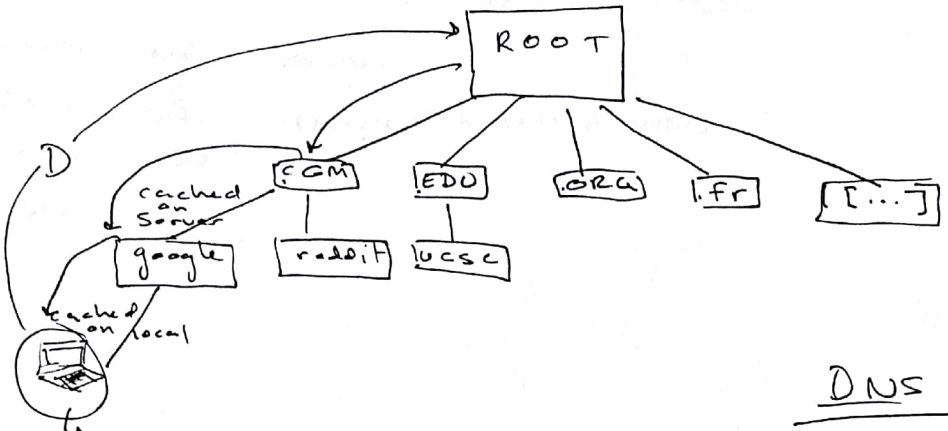
valid Shuffling
↳ x = 100

Serial Execution
- one stream execution
- everything in order
- x = 100

non-serializable Schedules
histories — invalid Shuffling cycles (wr, ww)
- stale read,
- divergence

scheduling (list of transactions)
transaction (operations)
list of options
: f (r/w)
: k
: v

dep



DNS ENTRIES:

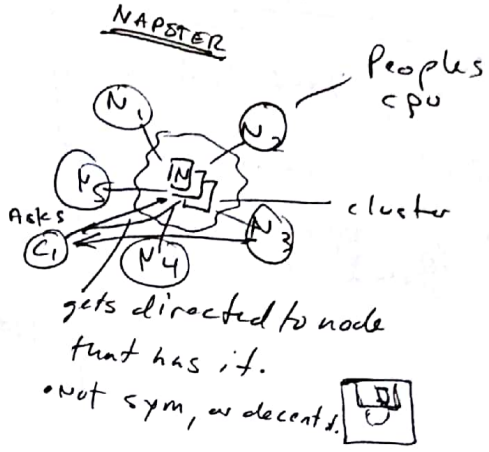
- A google.com → 128.76.2.112, 178.76.2.113, 197.2.2.10
  - 'name' → 'place'
- TTL (aggressive Caching)
  - time to live
  - Short ttl – cache clears frequently.
    - if Updating often
    - small ttl is better, less stale data.
  - each domain server decides ttl for data underneath it.
- Primitive
  - LOAD BALANCING
  - SERVICE Discovery

11/10/17

Peer To Peer

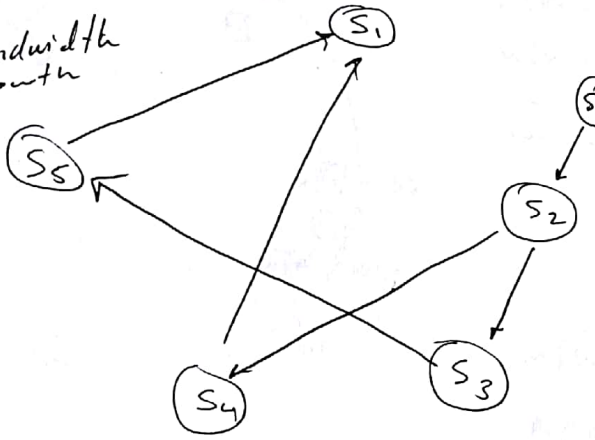
- Users, Not Servers
- No Distinction between Clients & Nodes
- Decentralized
- Symmetrical

Napster



Scaleability:

- minimize cost
  - storage & bandwidth
  - want log. growth



Gnutella

- Decentralized
- Symmetrical

who can you see?

eventually, the chatting will result in everyone knowing.

Flooding

retrieval

↳ do you have this?

CHORD - DHT

GOALS: (weakly consistent)

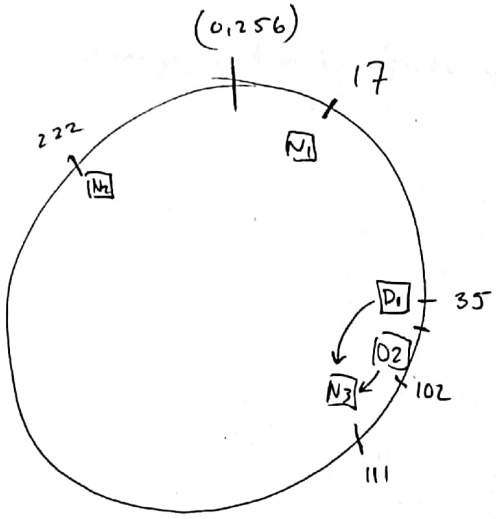
- Massive Scale (100K nodes / 10^6 nodes)
- RESILIENT TO CHURN

map names to nodes (computational Resources)

- SCALEABLE (Logarithmic ~~Scale~~ storage & network costs)
- Efficient (name → node in log time) "(Lookups)"

CHORD TRICKS

- consistent hashing
- "Finger table"
- Link List stuff (Doubly Linked List)



h ⇒ hash function
h(data) = hash
h(node) = hash

imagine we have 3 nodes

$h(N_1) = 17$
$h(N_2) = 222$
$h(N_3) = 111$

$h(D_1) = 35$

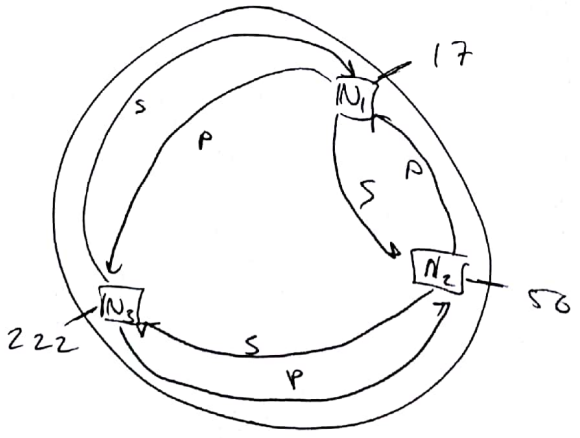
NOTE: if $h(N_4) = 45$, D_1 would remap to N_4.

DATA ITEM BELONGS TO SUCCESSOR HASH VALUE.

$h(D1) = 35$ → hashes/links to $N_3 = 111$

N: NODES
K: KEYS

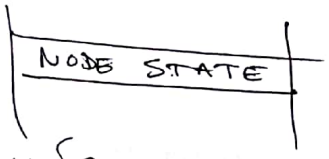
∞ H.P. if a node disappears, need to remap $\frac{K}{N}$



STORAGE: $O(1)$ → doubly linked list

NETWORK: $O(N)$ — have to go all around circle to find place

EFFICIENCY: Dog Slow



maintaining { Successor (next r)
Predecessor

FINGER TABLE

(i) Power	hash	ADDRESS
0	$n+2^i$	10.0.0.1
1		
2		
3		
4		
⋮		

$n+2^i$ →

FINGER TABLE – Lives in $\log_2(n)$ for lookups.

Points to nodes powers of 2^i distance from me.

↳ has $\log_2(n)$ storage pointers

So... storage, network, efficiency are all $\log_2(n)$.

11/10/17

More Agreement Protocols

FAMILE

TWO-PHASE COMMITT

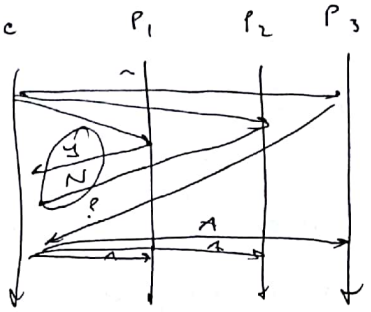
correctness Properties for agreement

safety { 1. Validity
2. Agreement

liveness 3. Termination

- one-node has power ← decider/coordinator
  - decides → COMMIT
  - → ABORT

C/A



- 2PC to ensure all nodes are on board to C/A
  - → after votes are in, $P_1, P_2, \ldots, P_n$ have no choice but to commit. (or abort, really after the votes come in yes, the $P_1 - P_n$ do as C_n says)
  - → have to get Yes vote from every node.
  - → but ultimately, the Coordinator makes the C/A decision

CONSENSUS

1. Make progress if only a bare-majority of nodes are correct.

2. To tolerate f-failures/faults, we need $\boxed{2f+1}$ nodes.

"collection of nods, satisfy 3 agreements (well, 2 & mostly 3) and the two rules above.

When is consensus hard?
when two opposing ideas are trying to be decided at the same time.

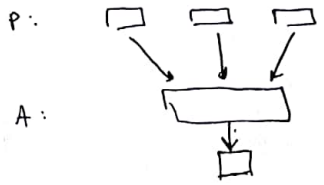
Paxos

All of the computers play all of the roles

1. Proposer → node → clients may communicate → looks similar to coordinator in 2PC
2. Acceptor → Passive (relatively) → listen to accept values, make decisions, send votes back regarding decision to accept values.
3. Listener → remember which decisions were chosen.

ex: Simplest Algorithm

one acceptor
could be many Proposers



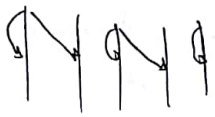
1. NEED MULTIPLE ACCEPTORS

but, we may run into two concurrent proposals with no causality
→ total order anomaly

→ to accept, we need majority multiple acceptors to accept in order to decide what to do.

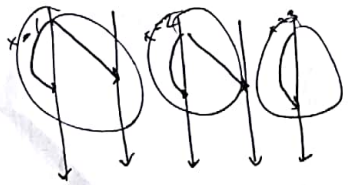
2. Acceptors need to accept multiple values.

majority anomaly



try: (1) ~~1 ACCEPTOR~~ X didn't work

(2) ~~Acceptors accept first value they see.~~ didn't work



- no majority vote. so we cannot decide.

- each node is a Proposer & an acceptor.
- we can send accept to ourselves

why? acceptors sometimes need to accept multiple values.

3. NEED A TWO-PHASE PROTOCOL

4. NEED TO ORDER OUR PROPOSALS

~~highest~~ extrema numbered gets chosen.

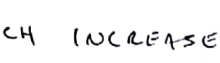
(3) Last value?

doesn't work!

NUMBERING PROPOSALS

1. EACH NUMBER MUST BE UNIQUE
2. MUCH INCREASE MONATOMICALLY

lamport clock with identity → sufficient to implement Paxos.



typically done by bit-shifting a big number.

want capability new proposer to come along & unlock system with dead node.

PAXOS

Proposers

1. Choose a new proposal number
2. BROADCAST A PREPARE MESSAGE CONTAINING (N) TO ALL ACCEPTORS (SIMILAR TO CMP IN 2PC)
4. WHEN A MAJORITY OF ACCEPTORS RESPOND, PROPOSER CHOOSES VALUE TO BE ACCEPTED.
   - → HER OWN VALUE, IF RESPONSES ARE UNRESTRAINED.
   - → THE HIGHEST NUMBERED PROPOSAL, OTHERWISE
5. BROADCAST ACCEPT(N, VAL)
7. WAIT FOR MAJORITY

   If min-proposal = N from every response. val is chosen.

   else GOTO 1.

Acceptors →

Variable: min-proposal
Accepted Proposals
Accepted Values

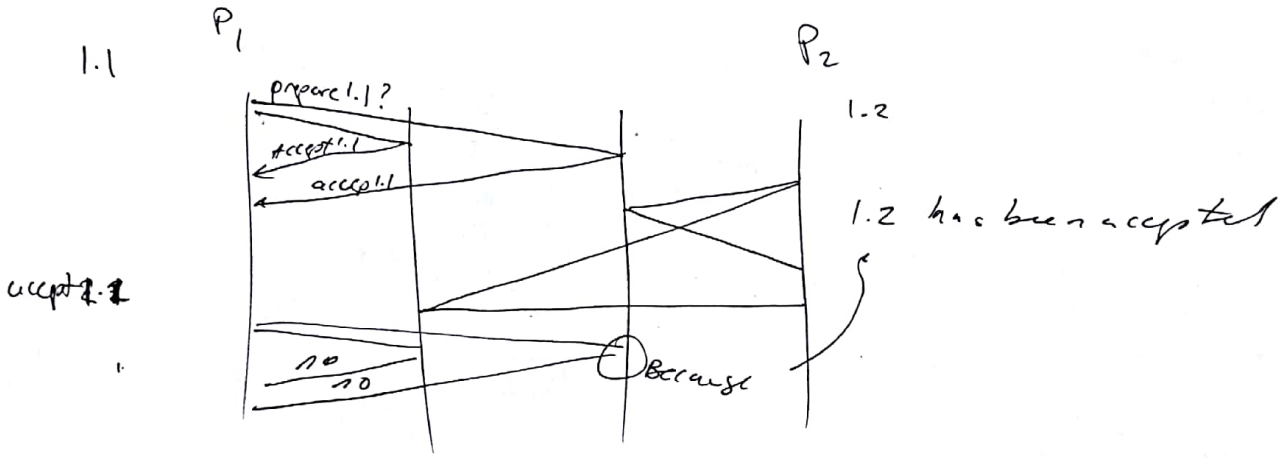
3. IF N IS GREATER THAN MIN-PROPOSAL
   - → SET MIN-PROPOSAL TO N. REGARDLESS OF CHANGE TO MIN-PROPOSAL
   - Return (accepted proposal, accepted value)
5. ~~BROADCAST~~ If N >= min-proposal then,
   - → accepted proposals & min-proposal are N.
   - ACCEPTED VALUE = VALUE.
   - RETURN (MIN-PROPOSAL)

fisher lynch Patterson Result (FLP)

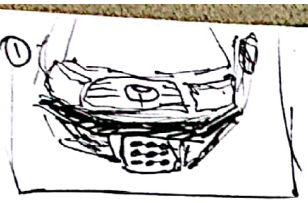
FLP IMPOSSIBILITY RESULT:

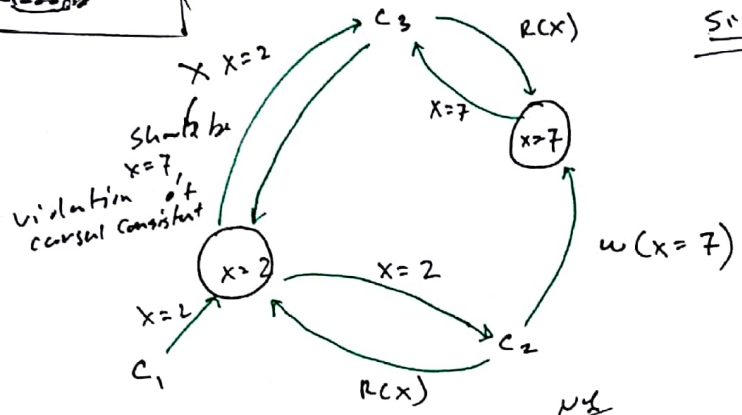
In an async system, any deterministic consensus protocol has executions that fail to terminate.

AKA: "consensus is impossible"



11/14/17

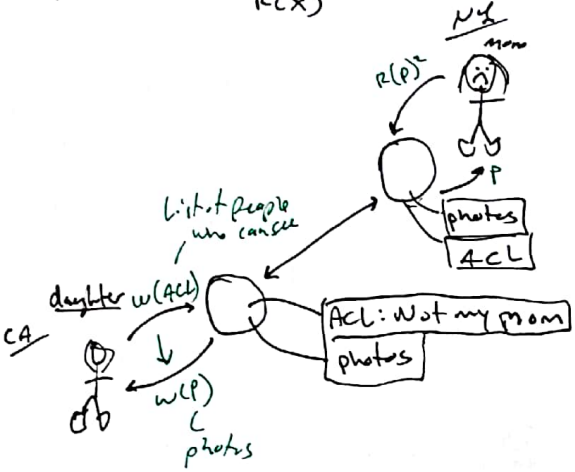




Single Variable example

NOTE: C_1 w(x=7)
w(y=2
C_2 r(y=7)

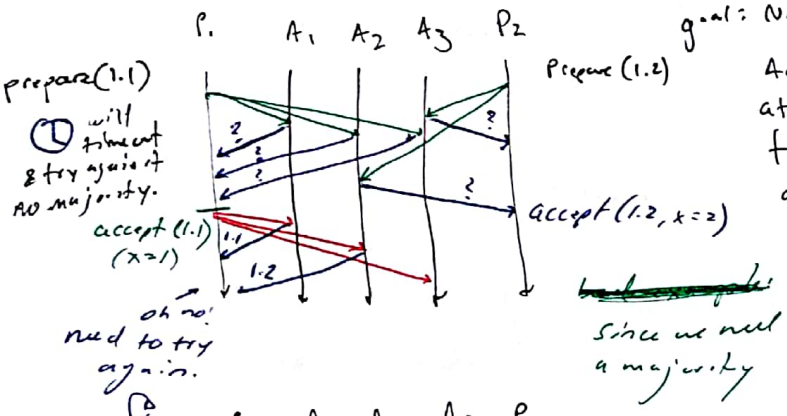
we know if C_2 reads y=7, somewhere in time w(y=7) occured.
suppose:
C_2 w(z=2)
C_2 r(z=2)
if C_2 r(z=2) we can see all events which happened before.



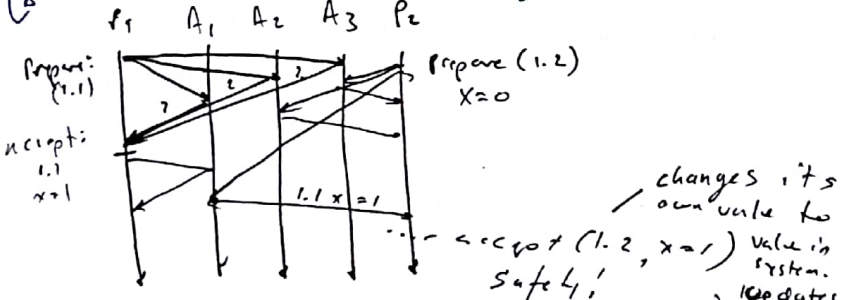
Spring-break photos

MIDTERM #2	
Median:	70.5
mode:	68.5
mean:	78
STDDEV:	15.6

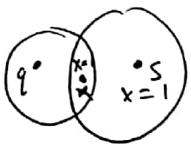
PAXOS



goal: No matter the order Acceptor nodes get writes, at the end, they're all friends who agree on the value.

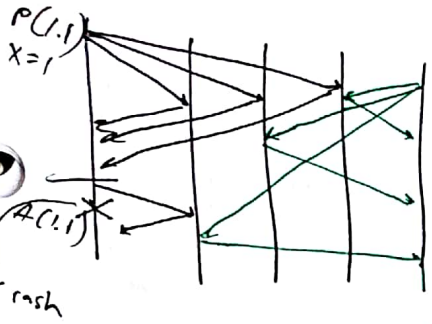


is over cautious, but necessary for correctness

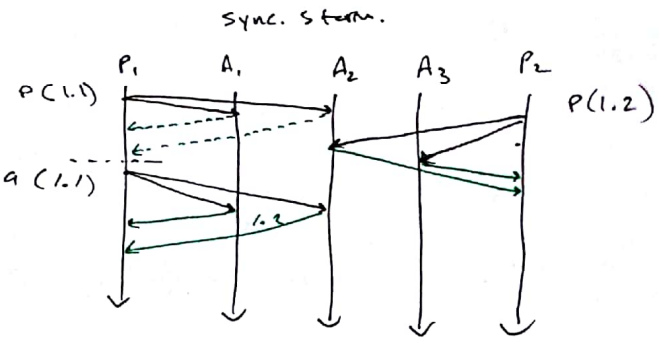


x=1 is chosen due to two majorities may overlap on a single node.

- must assume if we see a single constrained value, we use that value as a safety net. choose higher number to satisfy Liveness

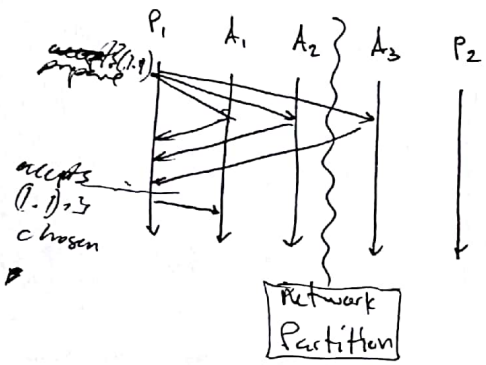


accept(1.2, x=0) due to fact that A crashes.



- John osterhaut – stanford Paxos
- need to re-read paxos made simple.
- Need to know how to run paxos by hand for final

network partition example:



P_1: finishes prepare phase but P_2 finishes prepare phase before P_1 accept phase completes, and we're stuck in a bind.

→ To solve: introduce timeout to solve issue of this loop. timeout is ~~randomized~~ gives other proposer chance to finish, this allows me to do what I want. We need to chose random timeouts to break out of synchronization storm. pick a number 0–10 ms sleep, if we get blocked again → 0–100ms, every block, we increase exponentially to make condition timeout more unlikely. exponential backoff

so, we can implement Paxos

Random timeout
exponential backoff
Leader election

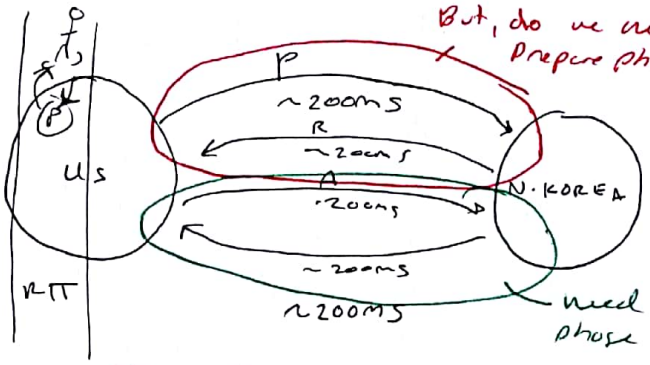
LEADER ELECTION

1. ALL NODES PERIODICALLY BROADCAST THEIR ID TO ALL NODES
   → EVERY T MS
2. IF I HAVEN'T HEARD FROM A HIGHER ID (IN 5 MS)
   → ACT AS A PROPOSER
3. IF I HEAR FROM SOMEONE WITH A HIGHER ID,
   → ACT AS AN ACCEPTOR.

Prepares are there to block earlier proposals in which they shouldn't accept.

TERMINATION
Message Complexity
Config changes
but wait!!!

MULTI PAXOS

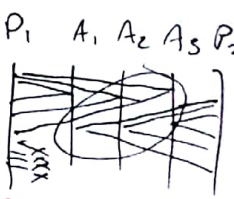


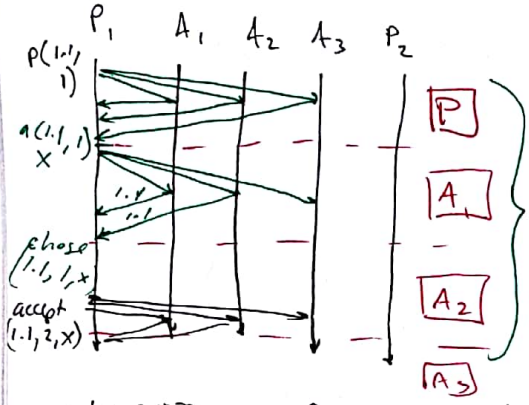
2 WAN

But, do we need Prepare phase?

need this Accept phase obviously.

note: delays over 250ms will be visible to user.





ONLY NEED A PREPARE FIRST TIME WE ARE IN CHARGE.

ONLY NEED TO Prepare one time each time we're the leader.

- Steps down if a higher round number is seen by us.
  - → In which case, we drops back to prepare phase & chooses an extreme.

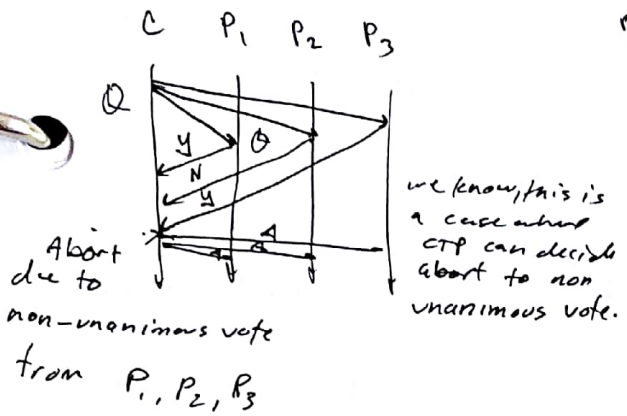
(W)IDE (A)REA (N)ETWORK (R)OUND (T)RIP (T)IMES.

- → need accept responses to ensure outcome.
- → REDUCE Message Complexity from 2 RTT TO 1 RTT.
- → WE CAN AVOID THE PREPARE PHASE AND GET AWAY FROM IT.
- Pass round number & slot number

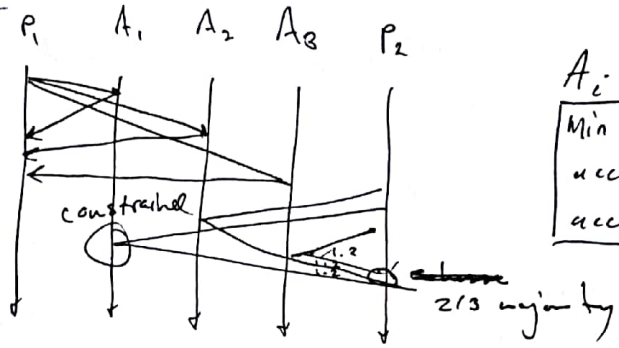
Prepare (1.1, slot 1) ← Should be called/referred to as phase. round. which round of prepare

2 RTTS
for worse case
1 RTT
for NORMAL/AVERAGE CASE

2PC + CTP.



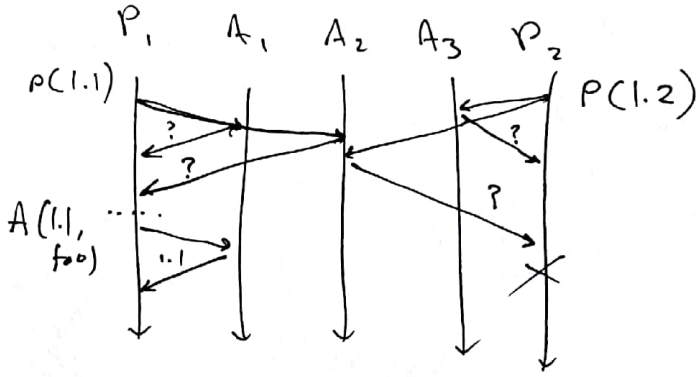
Paxos



A_i

Min Proposal accepted P accepted V

Since Coordinator node crashes,



WEAK CONSISTENCY
- SESSION GUARANTEES

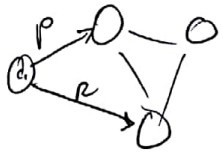
~~FAILURES~~

"what could possibly go wrong?"

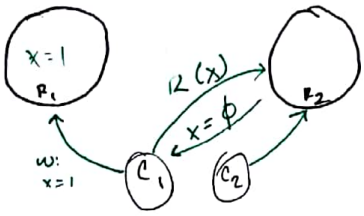
→ Ruling Out Anomalies

each Replica needs to maintain a view of the world.

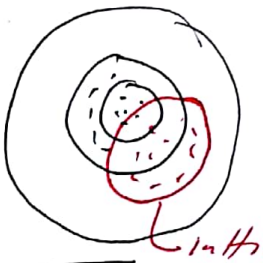
if a client writes to a node & reads from another.



what if we read from a node & it doesn't have that key?
block or forward.

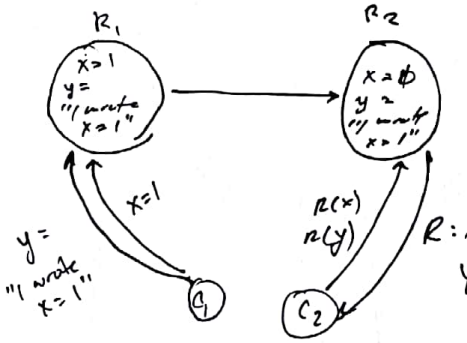


Anomaly with respect to "READ your writes"



Inflicted

what could go wrong
1. STALE READS
writes & reads conflict
2. WRITE CONFLICTS



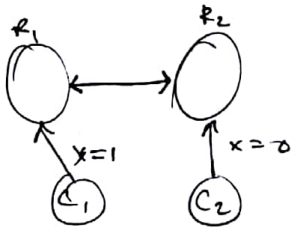
could use stickiness
client → R_1 for session.

Easy Trick:

KEEP VALUE ON CLIENT.

CACHE WRITES

PERFORMS GREAT.



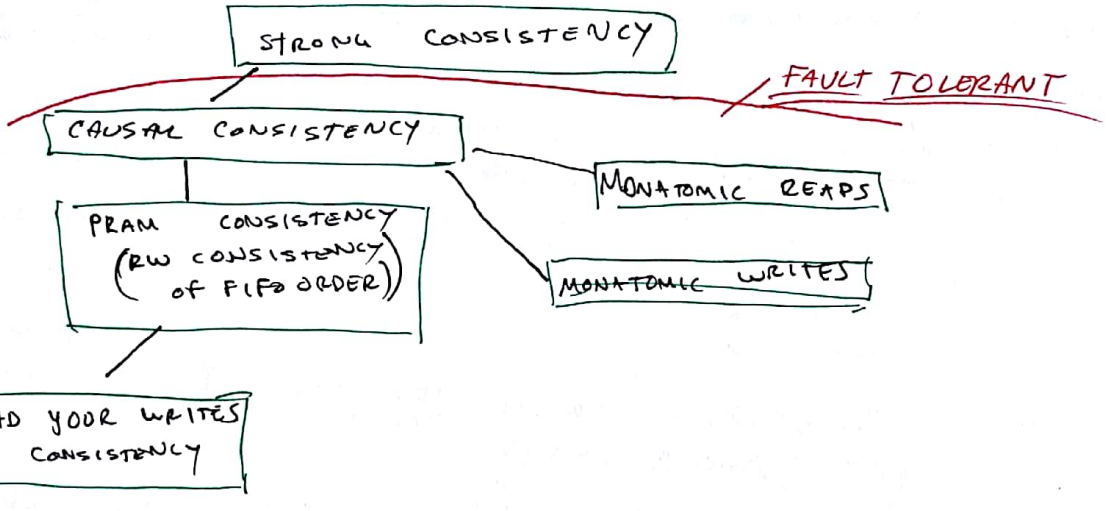
- what if we have an echoing reliable broadcast?

SESSION GUARANTEES

1. READ YOUR WRITES.
   →
2. MONOTOMIC READS
   ⟹ CAN'T READ OLDER VALUE
3. WRITES FOLLOW READS
   - RESPECTS "HAPPENS BEFORE"
   →
4. MONOTOMIC WRITES → "who cares dog?" - P. ALVARO

NOTE but I guess our Asg 3 use causal consistency yet still allow stale reads?

"Cache Rules Everything Around Me"



11/21/17

TODAYS TOPICS: PERFORMANCE

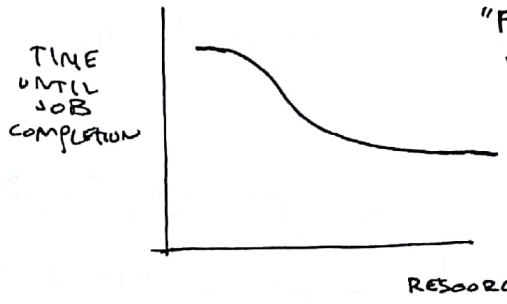
TRANSACTIONS &
- Concurrency Control
  - conflicts
  - schedules
  - serializability
  - Deadlocks
  - 2PL, OCC
  - Recovery
  - ACID

PARALLELISM & SCALEUP
- SPEEDUP - SCALEUP
- SCALEOUT - AMDAHL'S LAW

PLACEMENT OF DATA & COMPUTATION
- PARTITIONING SCHEMES

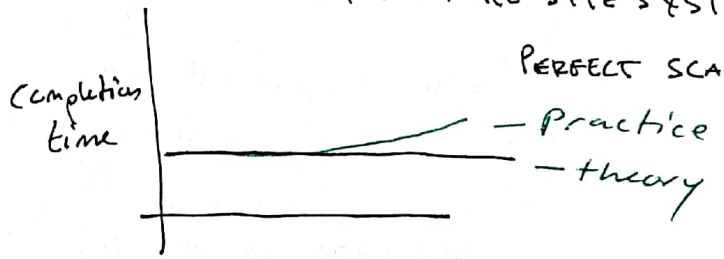
HW4 — USING LOTS OF CPUS TO GO FAST.

SPEEDUP: MEASURES THE DIFFERENCE IN RUNTIMES WITH OPTIMIZED UPGRADES



"FIXED WORKLOAD" LOOKING AT EFFECT OF OPTIMIZATION DUE TO INCLUSION OF RESOURCES.

SCALEUP: DESCRIBE ~~SYSTEM~~ SINGLE SITE SYSTEM — ADDING CORES TO A BOX.



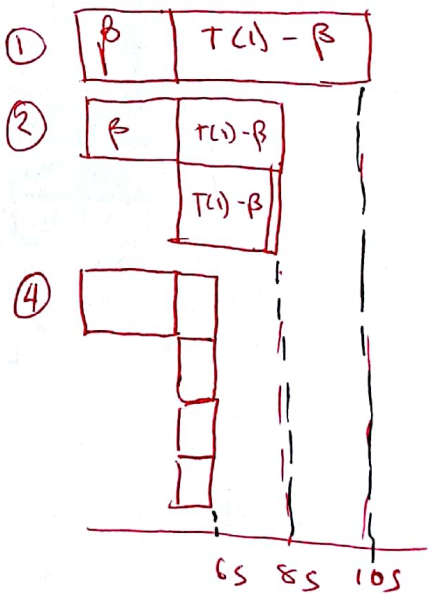
SCALEOUT: INCREASING SIZE OF CLUSTER (ACTUALLY ADDING CPU'S)

AMDAHL'S LAW:

$$T(N) = \beta + \frac{(T(1) - \beta)}{N}$$

- β: SERIAL EXECUTION (CAN'T MAKE FASTER. THIS IS THE BOTTLENECK)
- $T(1)$: TIME TO RUN ON ONE MACHINE
- N: RESOURCES
- $(T(1) - \beta)$: PARALLEL EXECUTION

$$T(N) = \frac{\beta}{S} + \frac{T(1)+\beta}{P}$$



BIGGEST WINS → TURN OF CONSISTENCY MECHANISMS THAT INCREASE LATENCY.

- OPTIMIZE β → SERIAL EXECUTION.

DATA PLACEMENT

- Computation → easy.
- DATA → non-trivial, (map reduce, PROCESSING LARGE AMOUNTS of DATA)
- PARTITIONING:

PROBLEM OF DATA PLACEMENT

1. **UNIFORMITY**: AS WE INGEST DATA, WE DISSEMINATE DATA EVENLY. (HASHING?)
2. **SCALEABILITY**: ~~AS N → ∞~~, WHEN POSSIBLE, WE WANT LOCALIZED AS MUCH AS WE CAN.
3. **WRITE**: SUPER FAST
4. **READ**: SUPER FAST

} HARD TO DO BOTH IN PRACTICE

5. **STABILITY**:

- DIVIDE DATA COMPLETELY EVENLY
  = QUICKEST SOLUTION
  → TOO EASY!!

CLUSTER FILE SYSTEM DESIGNS
UCSC CEPH SEPH

	uniformity	scalability	write fast	read fast	stability
RANDOM:	✓	✓+	✓+	✗	✓
HASHING:	✓	✓	✓	✓	✗-
RANGE:	✓	✗	✓	✓++	✓-
RR:	✓+	✓	✓	✗	✓
DIR:	✓+	✗	✓-	✓-	✓
CONHASH:	✓	✓	✓	✓	✓+

A. RANDOMLY → NOT A BAD PLACE TO START.
LOOKS ASS IF YOU NEVER NEED TO READ DATA

B. HASHING → JUST AS MUCH UNIFORMITY AS RANDOM & ITS DETERMINISTIC!!
h(x) % K gives us bucket

C. RANGE → READS ARE REALLY FAST, SUPPORTS RICHER QUERY.

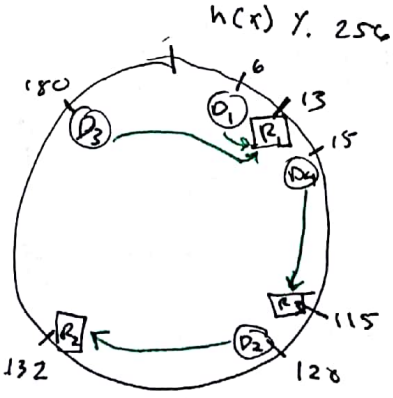
D. ROUND-ROBIN → ~~RAND~~ GREEDY → ALMOST STATELESS

E. DIRECTORY

F. CONSISTENT HASHING

given a cluster map, we can give it to a Function & find out where the data (reliably) needs to be stored.

BIP MAP INDEX OVER POINTER



it R3 crashes
Data will look for Successor on circle, now it just need to ramp one key.

$D_1, D_2, D_3, [\dots]$

$R_1, R_2, R_3, \dots$

$h(R_i) \% 256 = 13$

$h(D_1) \% 256 = 6$

③ CMPS128

Concurrency Control

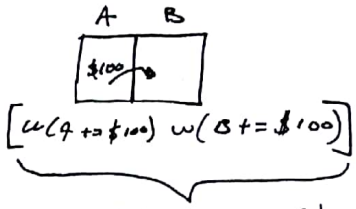
1. Establishing total order

A - Atomic

C - Consistent (mumble mumble)

I - Isolated

D - Durable



ok for none or both to happen, but not one or the other.

REALLY, because no-one has defined consistency is DB.

ACID

T_1	T_2
X=R(c)	X=R(c)
X=X+1	X=X+1
W(c=X)	
(X=2)	W(c=X)
	(c=2)

"Jim Gray" → Lost Transaction

Two transactions happened, but we only incremented the counter once.

NOTE: In a concurrent schedule have to think about what happens between reads & writes computations.

Serializability → Simultaneously Stronger & Weaker than total-ordering.

Conflicts:

write-write conflict
2 xacts wrote to same variable

T_1	T_2
w(x=1)	
	w(x=2)

write-read conflict
read-write

T_1	T_2
w(x=1)	
	w(x=2)
.....	.....
	r(x)

Schedule - AN INTERLEAVING OF PRIMITIVE OPERATIONS.

Equivalence

1. View Equivalence — TWO SCHEDULES ARE VIEW EQUIVALENT IF THE RESULT IN THE SAME END STATE OF DB & ALL READS RETURN THE SAME VALUE.

NP HARD

w(x=2)	w(x=bread)	ok
R(x=2)	R(x=2)	ok

2. Conflict Equivalence — implies View equivalence but not the other way around

IF CONFLICTING OPERATIONS OCCUR IN SAME ORDER IN TWO SCHEDULES.

Have to be over same unit or range one has to be a write

Terms:

conflict: two xact conflict if they operate over same variable & one of them is a write operation.

A SERIAL SCHEDULE → ALL TRANSACTIONS RUN TO COMPLETION BEFORE THE NEXT BEGINS.

4

CMPS128

operational transform
google docs
uses google
wave
-granularity

SERIALIZABILITY

- A schedule is serializable if it is equivalent to some serial schedule – duh?!.....
  - → ORDER IN which conflicting Xacts happen is equivalent to

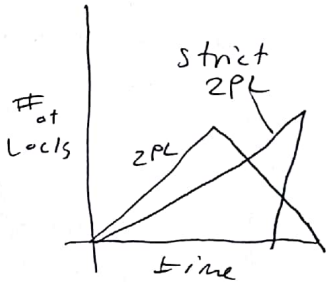
conflicts? waits

TWO-PHASE LOCKING → "strict 2PL"

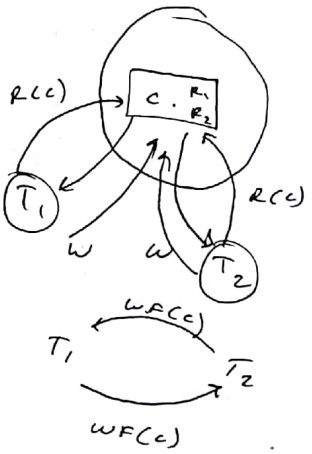
all you need to do:
worry solely about Xacts
2 place order on them.

1. BEFORE WRITING/READING A VARIABLE, Acquire a w/R lock
2. Hold all locks until end of Xact
3. RxR – are compatible
   - WR – conflict
   - WW – conflict

T_1	T_2
x = R(x)	
x = x+1	
W(x = x)	
	x = R(x)
	W(x = x)



1. DEADLOCK DETECTION
2. DEADLOCK REMEDIATION
   - → STOP 'EM → Abort/Pick a victim. – never guy or Xact. Lo should n't penalize older guy.
   - → PREVENT 'EM → (111)
     - ↳ USING A TOTAL ORDER



now we're in a deadlock.

Assumes infinite resources

"Ask for forgiveness"

conflicts – abort, throwing away work.

assume conflicts are rare.

Optimization Concurrency Control – someone dies & has to restart.

"You down with OCC?"

2PL → Ask for Permission

OCC → Ask for forgiveness

① ALL TRANSACTIONS RUN WITHOUT SYNCRONIZATION

② Xacts write to 'local' copies of variables

③ On commit → Validation occurs

example:
- T_1: Read-set {x y} write-set {x}
- T_2: Read-set {x} write-set {z}

compare R, W sets to all other concurrent Xacts.

if $R_{S_m} \cap W_{S_o} = \emptyset$ and $W_{S_m} \cap R_{S_o} = \emptyset$ and $W_{S_m} \cap W_{S_o} = \emptyset$

else abort

quick look

④ commit local w, to store.

11/28/17

Distributed naming

- DNS
- DHT's & P2P
  - CHORD

RECAP:

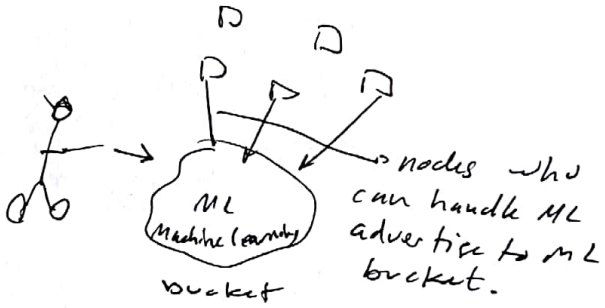
TRANSACTIONS
- Concurrency Control
  - 2PL: Ask for Permission
  - OCC: Ask for forgiveness
- Serializability
  - T_1 T_2 T_3

- MAKES SURE END STATE OF DB IS SAME
  - "VIEW SERIALIZABILITY" - affects are indistinguishable
  - "Conflict Serializability" - [illegible],
    - conflicting operations

DISTRIBUTED NAMING

why we name things?

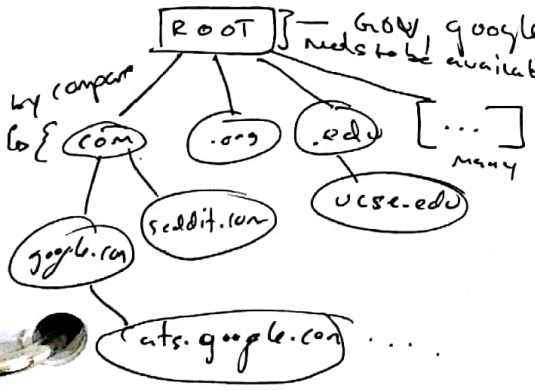
- So humans can read/tell things apart
- Identity - disambiguate
- Addressing / Retrival / data content.
  - names behave as indices.
- SERVICE DISCOVERY... (name is task)
  - Indexing
  - Decoding
  - machine learning



DNS - Your cpu is a replica!

APPROACH - handles case A, B

- HIERARCHICAL NAMESPACE



A SCALEABLE

B MULTIPLE ADMINISTRATIVE DOMAINS

C LOW LATENCY (Aggressively cached)(even on local cpu)

D AVAILABILITY is King.
- DNS is weakly consistent
- "imagine if every laptop had to ack"
- Thats what a strong consistent DNS system and that would suck my ass!" - Patrick

E Geo-Scale

11/30/17

BIG DATA

- DATA PARALLELISM
  - make stuff go fast

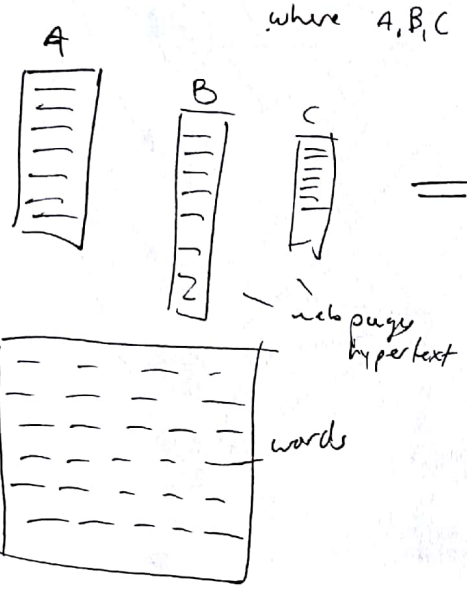
INVERTED INDEX

Searched term: Ted Dorsch

map(Ted) ∩ map(Dorsh)

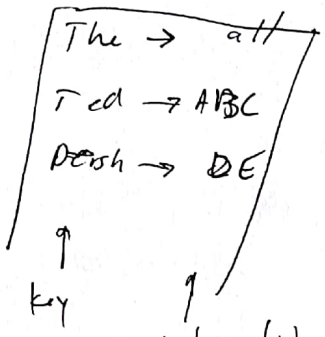
ABCADE = A.

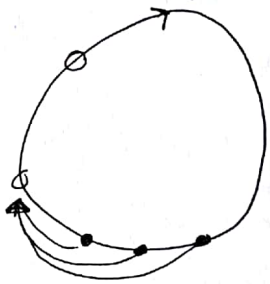
where A, B, C are urls



Map

word → list (URLS)





or:

a single machine:

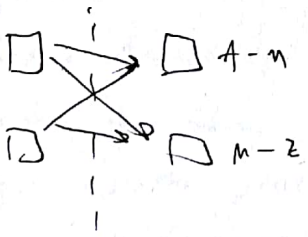
for every url

if word in url.doc

append url onto values list for word

many machines: Asg4

all-to-all communication



MapReduce

filtering, transformation phase — map

Combination — reduce

map: for each Doc

for each word

emit <word [Doc]>

reduce: ~~for k, v~~ for every pair <k, [L1]>

<k, [L2]>

emit <w, [L1 + L2]>

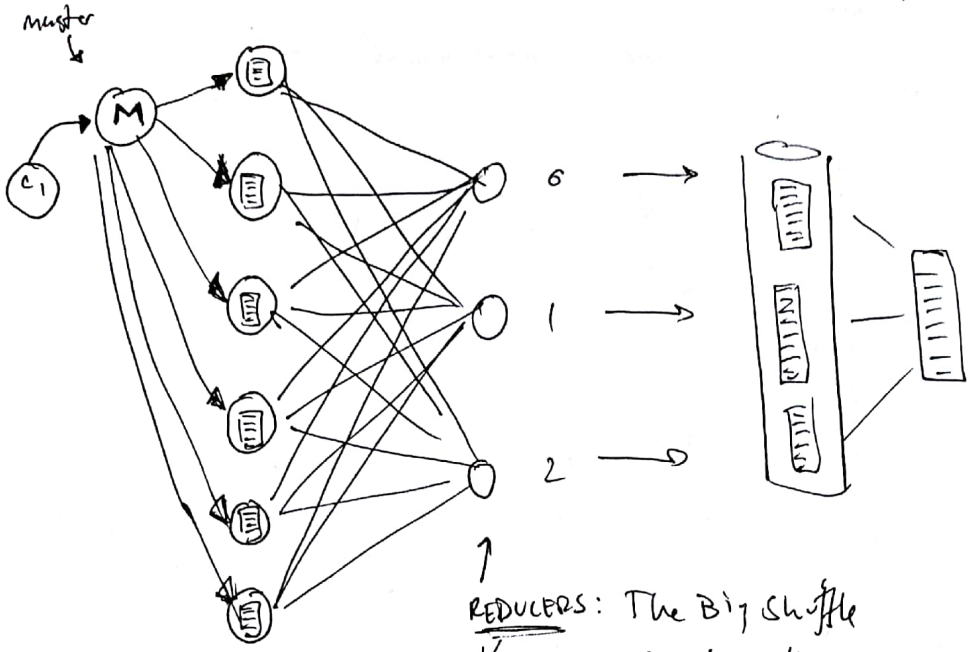
map(k1, v1) → list(k2, v2)

Reduce(k2, list[v2]) → list[v2]

(2)

MAPREDUCE

$map(k_1, v_1) \rightarrow list(k_2, v_2) \rightarrow reduce(k_2, list(v_2)) \rightarrow list(v_2)$



+100k mapper
50K Reducer

REDUCERS: The Big Shuffle
Bottleneck! All to all

mappers: run map locally (parallel to other mappers) eventually

still to deal with:
- ✓ fault tolerance
- ✓ Performance
  - limping hardware
  - Stragglers...
  - network Bandwidth
  - last gasp.
  - data skew

☒ PERFORMANCE → PROBLEM OF LOCALITY

1) SCHEDULING COMPUTATION CLOSE TO DATA
(RATHER SEND A FUNCTION THAN 1TB DATA)

☒ STRAGGLERS

INSIGHT: limping hardware

Speculation: Predictive Branching.

if a machine is taking a long time to complete a task, tell another process to run task.
takes care of old computers/slow processes.

bonus: gets rid of stragglers
neg: do a little more (x2) work.

TRADEOFF: throughput for latency.
"resources for time"

☒ FAULT-TOLERANCE

NOTE: mappers ack master
timeout is sufficient

mappers — promote new
if comes back
tell it to fuck off

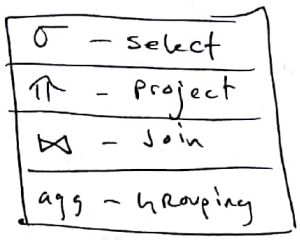
Reducer — Promote new & give it same bucket number
since mappers store file upon completion (NFS lookup) to get new reducer up to speed.

master — Boom Analytics
masters get smr with Paxos.

③

we, databasers, have been doing this for ages, we just didn't have a fancy name for it even we didn't work at google.

1. RESTRICTED PROGRAMMING MODEL
   LOSQL
2. DATA PARTITIONING
3. DATA SHUFFLING




```
select   x, y, count(z)
from     R, S
where Rq = Sq
and   Rx < 10
group by x, y
agg(σ x<0 R ⋈ S)
```

12/3/17

SECTION

ROLES

MAPPERS:

PARTITIONING
- DATA
  - fault tolerance
- TASKS
  - consistency

LOCALITY
- distance to data needed.

REPLICATION
- OF DATA
- OF COMPUTATION (STORE WORK TO LOCAL DISK)

REDUCERS:

INSERT DATA FROM MAPPERS

AGGREGATE USING JOIN

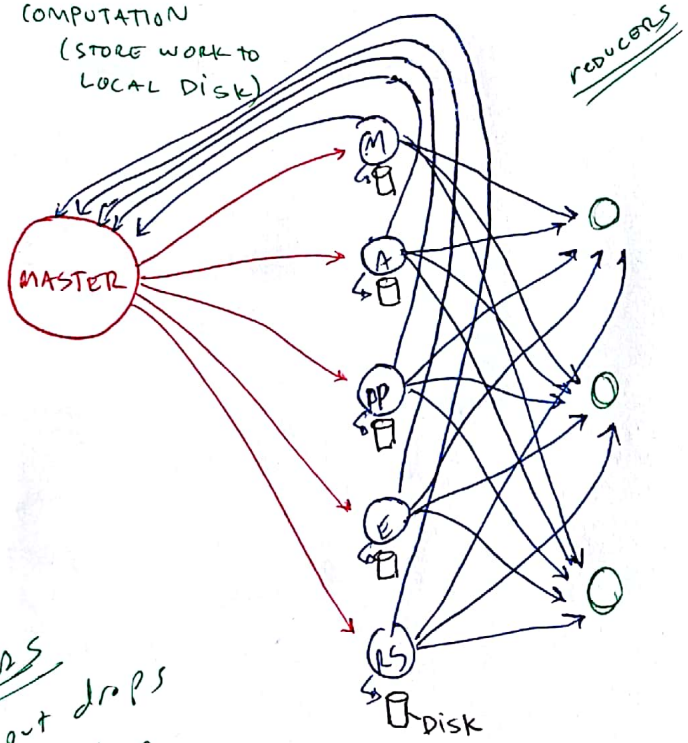
MASTER:

(MAPPERS)

DETERMINES WHO GETS WHAT WORK

DETERMINE WHICH REDUCERS MAPPERS TALK TO.

ALL-TO-ALL COMMUNICATION



MAPPER — REDUCER IS ALL-TO-ALL

Information Disseminated by Master

- which mappers get which work?
- REDUCERS LEARN WHERE TO SOURCE THEIR DATA
- KNOWLEDGE OF WHO IS UP.

FAULT TOLERANCE

- Master: replicated, Paxos
- Mapper: detect: timeout
  remedy: allocate node, give it the same mapping values
- REDUCERS: detect: timeout
  remedy: new reducer at same bucket & same sources.

STRAGGLERS

Throughput drops due to more than one node on a single job.

double + allocation of work tasks.

Latency drops avoid timeouts