

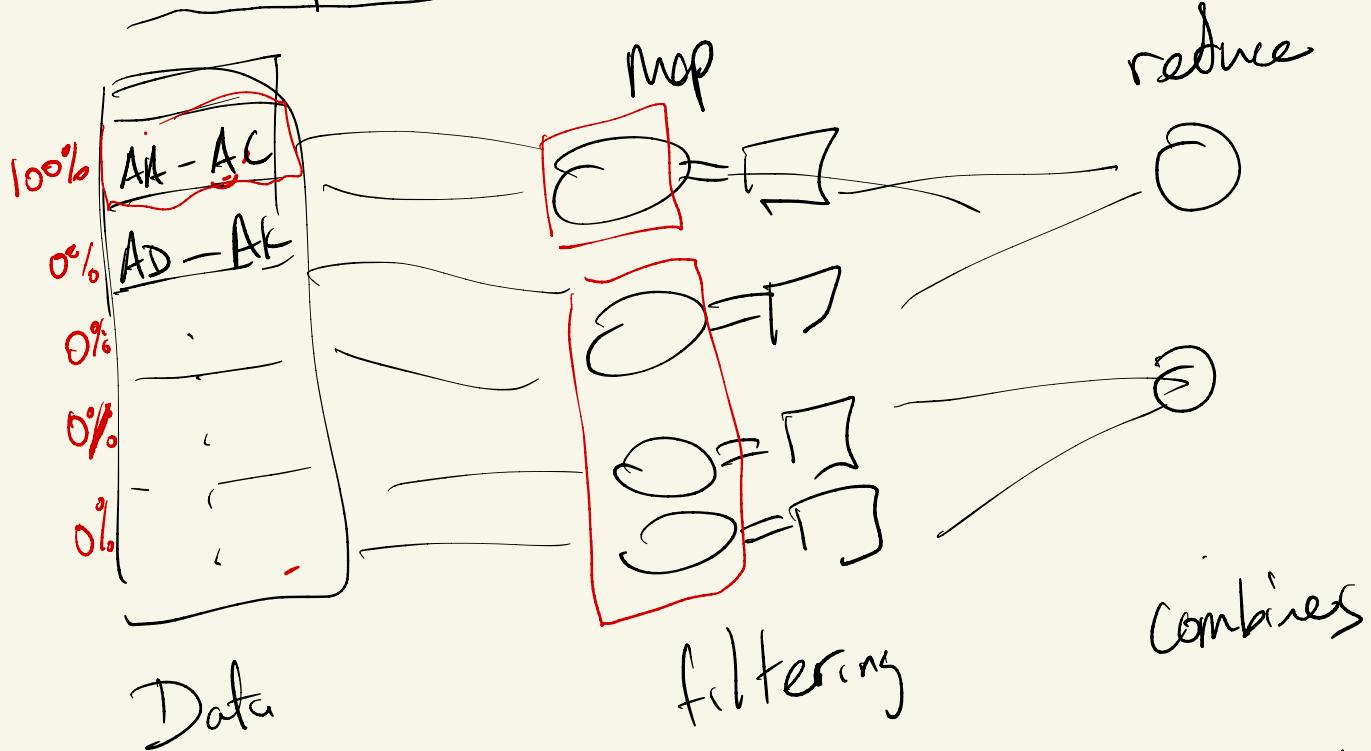
CSCI 1330

Review Session



Map Reduce (failure / perf + Big Data)

Map Reduce



Map Reduce = Big data frame that **manages servers** + data orchestration

user heartbeats to detect failures ↛ failures

 ↛ performance

 ↛ progress reports to detect fast/slow servers

- (1) why are servers slow?
- (2) How to recover/improve performance?

Which of these signals is the best method for detecting such a slow task?

- Progress reports
- Heartbeats
- CPU Utilization
- None of the above

If the slow task is due to data-imbalance, which of these approaches will best address the problem?

- Cancel and retry the task on a different server
 - Duplicate (or clone) the task to a different server
 - Quarantine the server (which is currently running the task), and never use it again.
 - None of the above.
- servers failing
slow nodes
Bad H/w

Networking (TCP/UDP/RPC)

TCP V. UDP

	<ul style="list-style-type: none"> Connection oriented ↳ Start up latency Good features ↳ retransmit/retry ↳ reorder packets 	<p><u>You get Nothing</u></p> <ul style="list-style-type: none"> ↳ Very fast ↳ has no overhead
when to use?	<ul style="list-style-type: none"> ① large bulk transfer ② web traffic 	<ul style="list-style-type: none"> ① small msg (RPC) ② live streaming

RPC

① Semantic : at-most-once

or 1 time

requires complex tech.

at-least-once

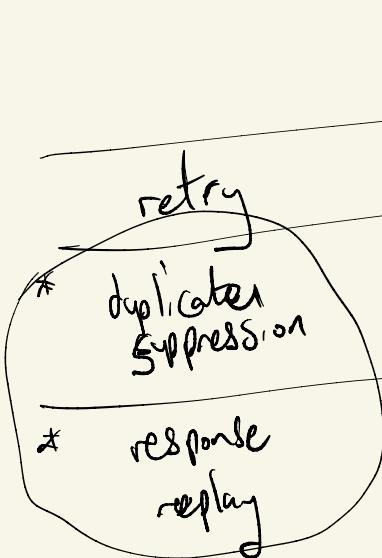
1 or more times

Idempotent

exactly once

the func call happens once

This is not practical

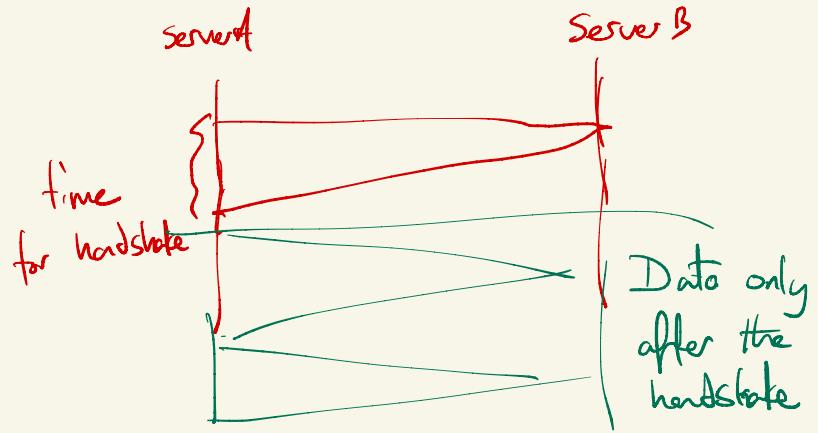


For short RPC messages

- () TCP
 UDP

For live streaming events

- () TCP
 UDP



Which of these is the main reason why UDP is used for RPC messages

- UDP provides amazing encoding that reduces data size
 TCP has a latency overhead but UDP does not
 TCP attempts to provide fairness which is bad in data centers
 UDP uses special sockets that are optimized for distributed systems

Which of these filesystem calls do you expect to be idempotent? (Select all that apply)

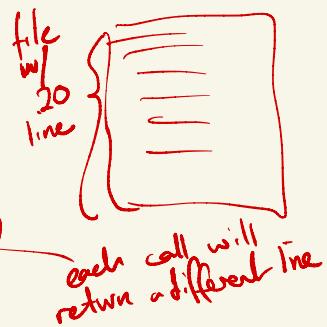
Note: these call do not exactly correspond to C calls and I have provided a description of their semantics.

[X] Set(key, value) // this function store the "value" in memory under the identifier "key"

[] char *fgets(char *str, int n, FILE *stream) // this function reads a line from the specified stream and stores it into the string pointed to by str. It stops when either (n-1) characters are read, the newline character is read, or the end-of-file is reached, whichever comes first.

[X] Delete(key) // This function deletes the data that was stored under the "key".

property of a function:
* if you call it once or N times you get the same result



You want to design a RPC framework that provides at-least-once semantics. Your team informs you that you only need to implement the 'request-retry' feature and you do not need to implement either 'duplicate suppression' or 'response replay'. Which of these best explain why you can ignore those two?

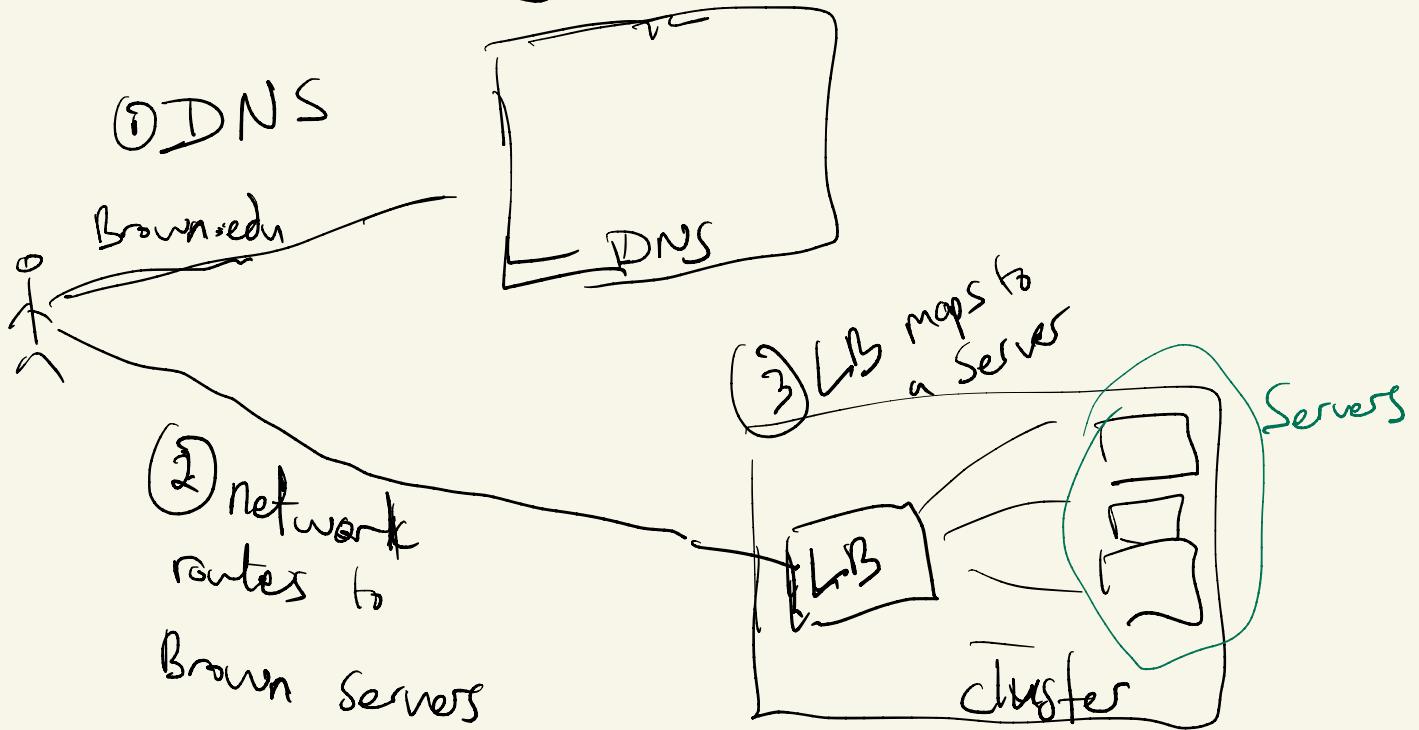
understanding semantics
helps you decide which features you need

() 'request-retry' by default provides 'duplicate suppression' and 'response replay'.

(X) Neither 'duplicate suppression' or 'response replay' are needed because at-least-once call can support multiple invocations.

() The team is wrong and you need to implement all three features.

Load Balancing (Global / local)



(1) DNS : name to ip mapping

(2) Network to get to cluster
(BGP)

(3) LB maps to server
(local LB) \Rightarrow distribute load
btw servers

Global LB

① reduce latency
(get you a close cluster)

② data policy

	Global LB	Local LB
Goals	<ul style="list-style-type: none"> (1) reduce latency to client / cluster (2) respect data policy 	<ul style="list-style-type: none"> (1) distribute load btw servers
Techniques	<ul style="list-style-type: none"> (1) DNS (2) BGP 	<p style="color: red;">FB</p> <p style="color: red;">Azure</p> <ul style="list-style-type: none"> (1) Round Robin (2) random (3) Modulo hash (4) Consistent hashing <p style="color: green; margin-left: 20px;">Amazon EC2</p> <p style="color: red; margin-left: 20px;">Google FB</p>

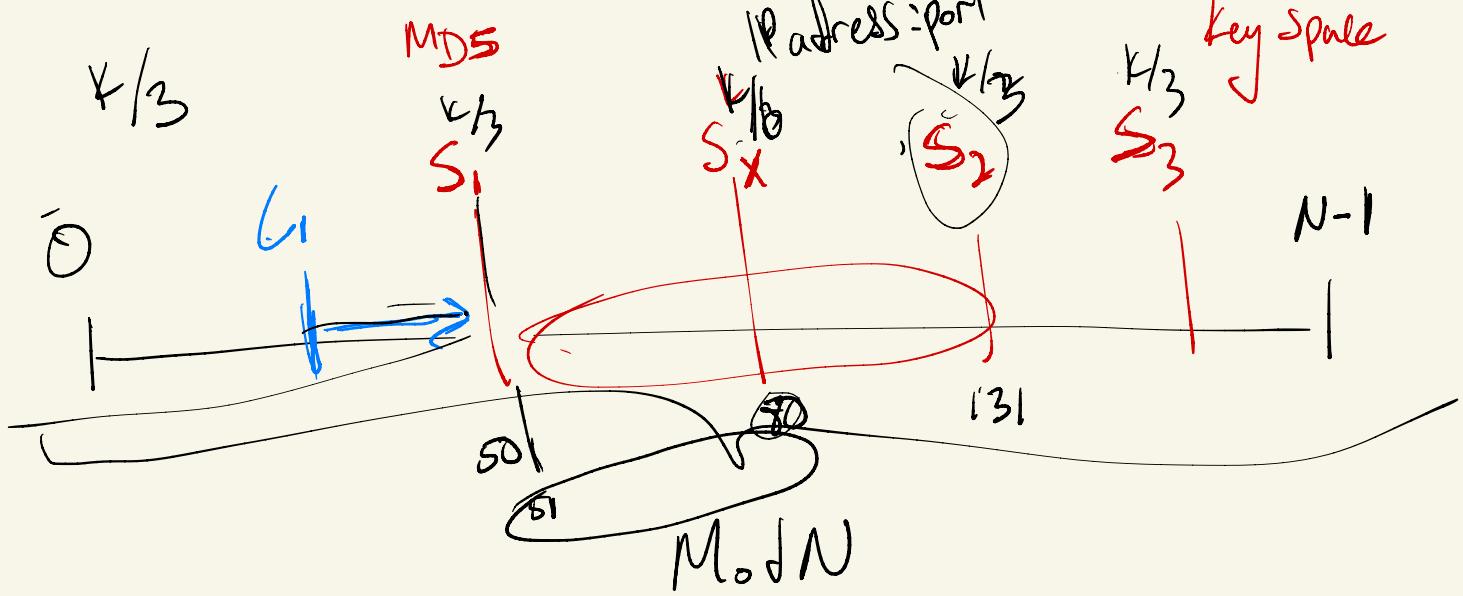
Local LB

key issue with Round Robin / Random is that they send connection to different servers. This is bad because we want all connection from a user to go to same server (TCP overhead / session info)

Consistent hashing → consistently assigns client/user to the same server

$ID = \text{hash}(\text{string}) \bmod N$

size of the key space



In this system if one of the N server crashes, how many clients will need to be moved?

- K (all clients)
- K/N (only the clients on a server)
- 0 (no client will need to be moved)
- K/(2*N) (half of the clients on a server)
- N (one client from each of the servers)

In this system, if a new server is added to the system crashes (now there are N+1 server), how many clients will need to be moved?

- K (all clients)
- K/N (the num of clients on a server)
- 0 (no client will need to be moved)
- K/(2*N) (half of the clients on a server)
- N (one client from each of the servers)

Which of these enables the global DNS infrastructure to scale?

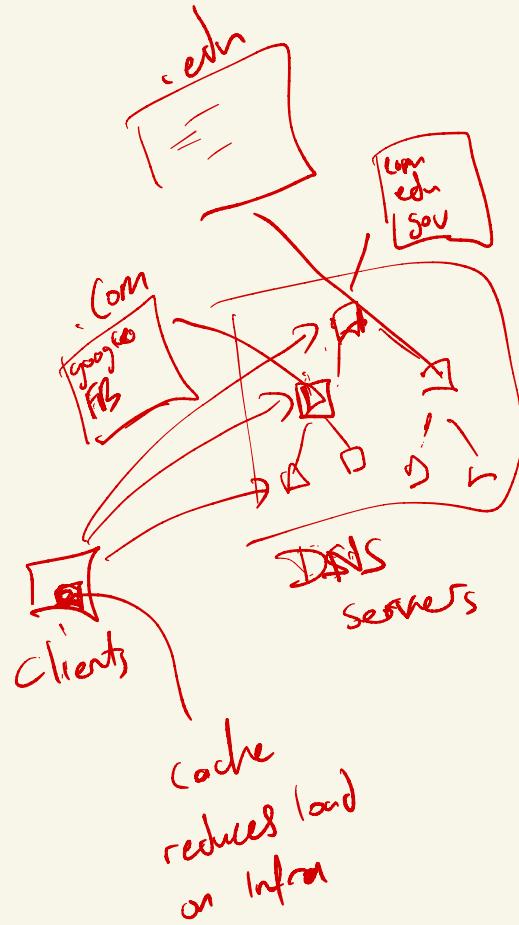
- Client side caching of responses
- The use of TCP
- Hierarchical partitioning of the Name space
- Automated partitioning of the IP space (Divide IP addresses and allocate different IP addresses to different cities)

Which of these are potential goals for a load balancer within a cluster?

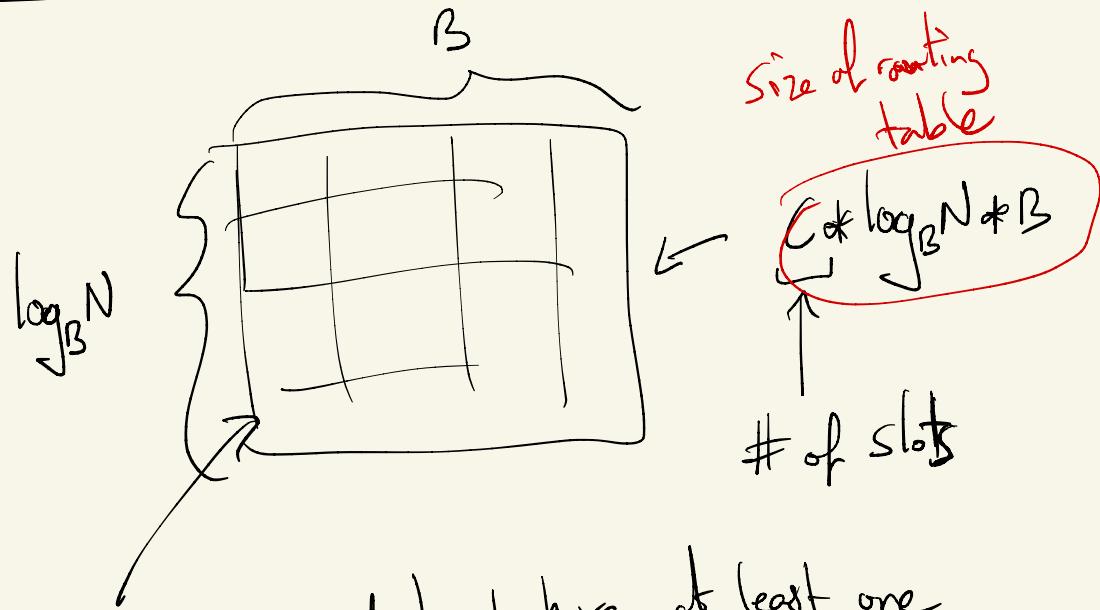
- evenly distribute work
- ensure consistent map of client to server
- assign client to server based on location
- provide local RPC semantics

$$K = \# \text{ of clients}$$
$$N = \# \text{ of servers}$$
$$\frac{\text{each server}}{N} = \frac{K}{N}$$

(Clients are evenly distributed)



Tapestry



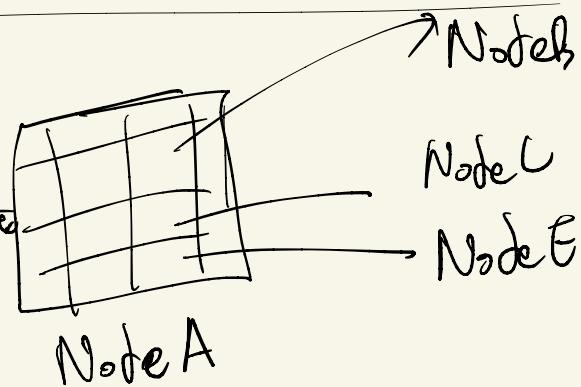
each row guaranteed to have at least one entry : the node itself

The max # of hops in any lookup is $O(\log_B N)$

of DIGIT in ID; every hop takes you one digit closer to destination

Backpointers

- ① Grateful exit: use backpointers to determine which nodes to update



Exit

Graceful

Un Graceful

① update routing tables
of nodes in your backpointer
structure

① heartbeats

② bad nodes detected
via RPC calls

② move objects to
new root locations

For Graceful failures, what is the main technique that tapestry provides to ensure this guarantee:

- () Clients republish object
- (X) Nodes (i.e., server) transfer object store before failure
- () Nodes (i.e., servers) use backpointers to find objects
- () Nodes (i.e., servers) use route table to find clients

For NON-Graceful failures, what is the main technique that tapestry provides to ensure this guarantee:

- (X) Clients republish object
- () Nodes (i.e., server) transfer object store before failure
- () Nodes (i.e., servers) use backpointers to find objects
- () Nodes (i.e., servers) use route table to find clients

Chord

client ID → next largest server ID

Time + Global State

Real Time is Bad (it is not monotonic / hard to synchronize servers)

What you care about is ordering of events

Total

every server agrees on
the order

FIFO

Servers don't agree on
all events: events need
to order events based
on clock / ID from
individual servers

Causal

Vector Clocks

"happens after"
relationship

logical clocks

Vector
clocks

Global State (distributed snapshots)

Time-based

time synchronization

manual

time to
go to all
servers

continuous

only valid
snapshot maybe
① at the beginning
of the system

ChandyLamport

- very strong assumptions
- ① no server will fail
 - ② the n/w never reorders msgs

Consistent snapshots : iff for every "recv" event in the distributed snapshot you also have the "Send" event.

Use of Vector Clocks to detect inconsistencies

→ Compare VC for the latest events in the Snapshot & check to some Server A has higher clock for server B than Server B has for itself

$$VC[(A, B)]$$

$$VC_A[B] > VC_B[B]$$

[10, 10] Snapshot

A

[5, 6] Snapshot

[5, 10]

B

Consider the following system with two servers S1 and S2.

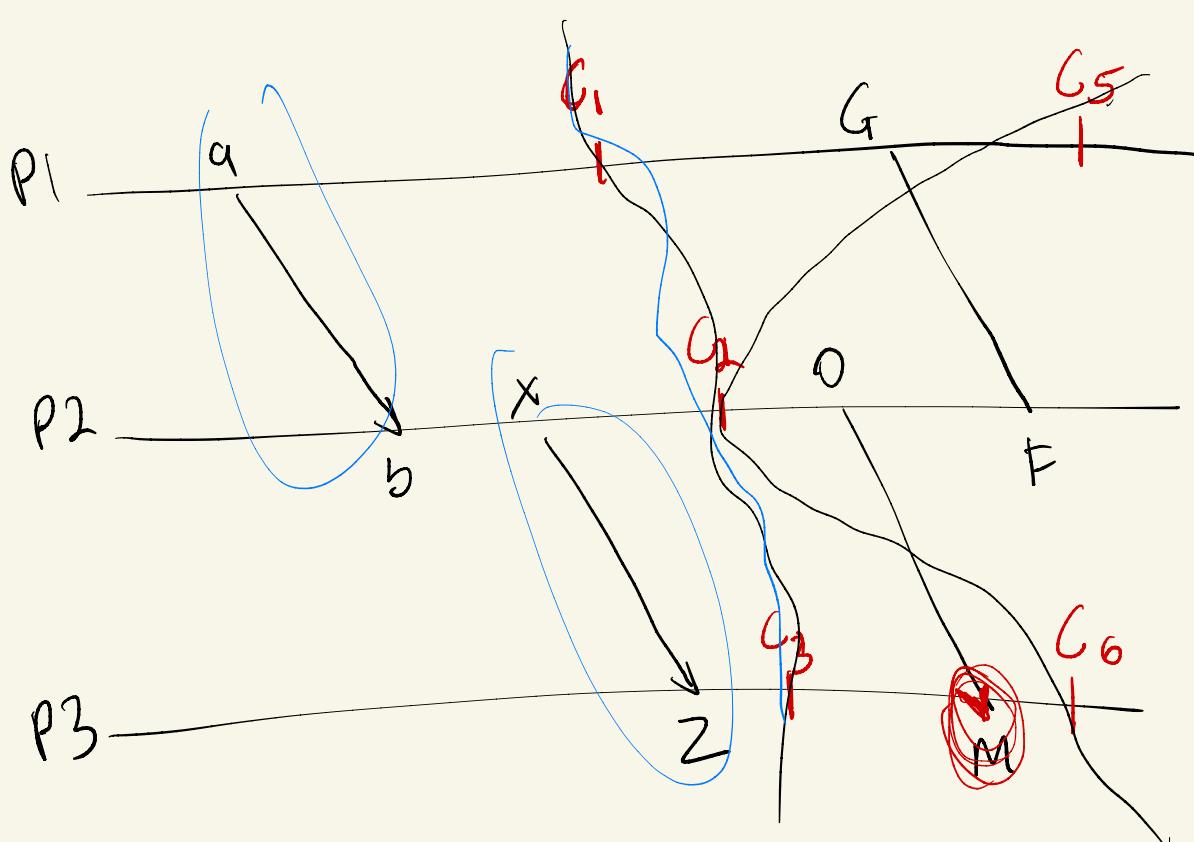
S1 receives and process events in the following order: X1, X2.
S2 receives and process events in the following order: Z1, Z2.

Which of these ordering of events is total but not FIFO
ordered:

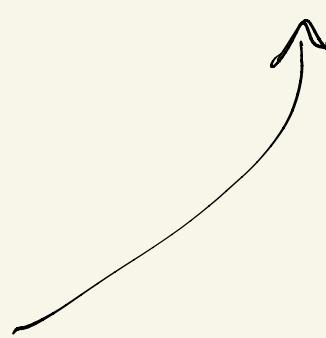
- total and FIFO
- not total
- () S1: X1, Z1, X2, Z2 S2: X1, Z1, X2, Z2
 - () S1: X1, X2, Z1, Z2 S2: X1, Z1, X2, Z2
 - () S1: X1, X2, Z1, Z2 S2: Z1, Z2, X1, X2
 - S1: X1, Z2, X2, Z1 S2: X1, Z2, X2, Z1

Which of these ordering of events is both total and FIFO
ordered:

- S1: X1, Z1, X2, Z2 S2: X1, Z1, X2, Z2
- () S1: X1, X2, Z1, Z2 S2: X1, Z1, X2, Z2
- () S1: X1, X2, Z1, Z2 S2: Z1, Z2, X1, X2
- () S1: X1, Z2, X2, Z1 S2: X1, Z2, X2, Z1



Consistent



This is
not
consistent

Snapshot @ P2

doesn't include
the sent event
associated w/ "M"

Topics

Map Reduce (Litimer)

Networking (TCP/UDP, P2P) ↗ chord ↗ Tapestry

L/B (Local V. Global / Consistent hash)

Time (logical / Vector / Snapshots)