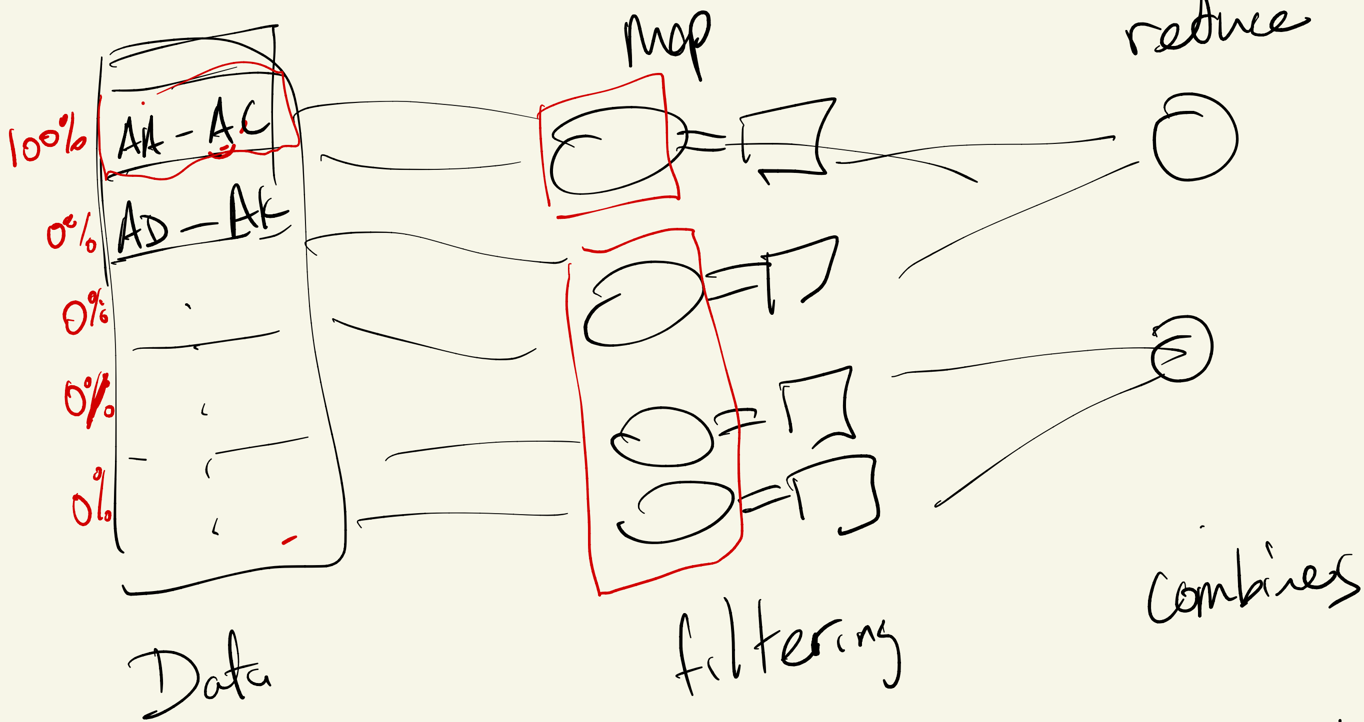
# CSCI 1380

# Review Session

# MapReduce (failure/perf + Big Data)

# Map Reduce




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

failures ⟸ Uses heartbeats to detect 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?
(X) 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 servers failing
( ) Duplicate (or clone) the task to a different server slow nodes
( ) Quarantine the server (which is currently running the task), and never use it again. Bad H/W
(X) None of the above.

# Networkin (TCP / UDP / RPC)

## TCP v. UDP

| | TCP | UDP |
|---|---|---|
| | Connection oriented<br>↳ startup latency<br>Good features<br>↳ retransmit / retry<br>↳ reorder packets | You get Nothing<br>↳ Very fast<br>↳ has no overhead |
| when to use? | (1) large bulk transfer<br>(2) web traffic | (1) small msg (RPC)<br>(2) live streaming |

## RPC

(1) Semantics : at-most-once | at-least-once, | exactly once

| at-most-once | at-least-once | exactly once |
|---|---|---|
| ⇓ 0 or 1 time | ⇓ 1 or more times | ⇓ the func call happens once |
| ⇓ requires complex tech. | ⇓ Idempotent | This is not practical |

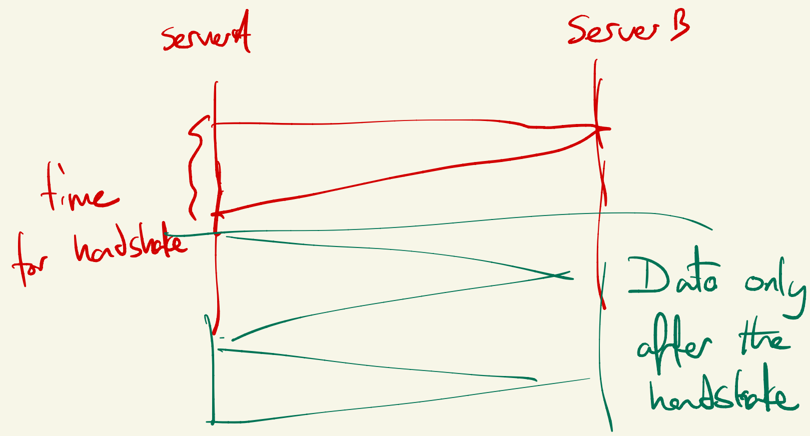
| | | |
|---|---|---|
| retry | ✓ | ✓ |
| * duplicates suppression | ✓ | |
| * response replay | ✓ | |

For short RPC messages

( ) TCP
(X) UDP

For live streaming events

( ) TCP
(X) UDP




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

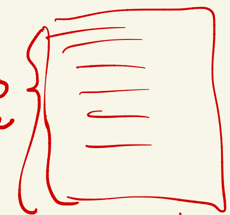
( ) UDP provides amazing encoding that reduces data size
(X) TCP has a latency overhead but UDP does not
( ) TCP attempts to provide fairness which in bad in data centers
( ) UDP uses special sockets that are optimized for distributed systems

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.

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

[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".

file w/ 20 line



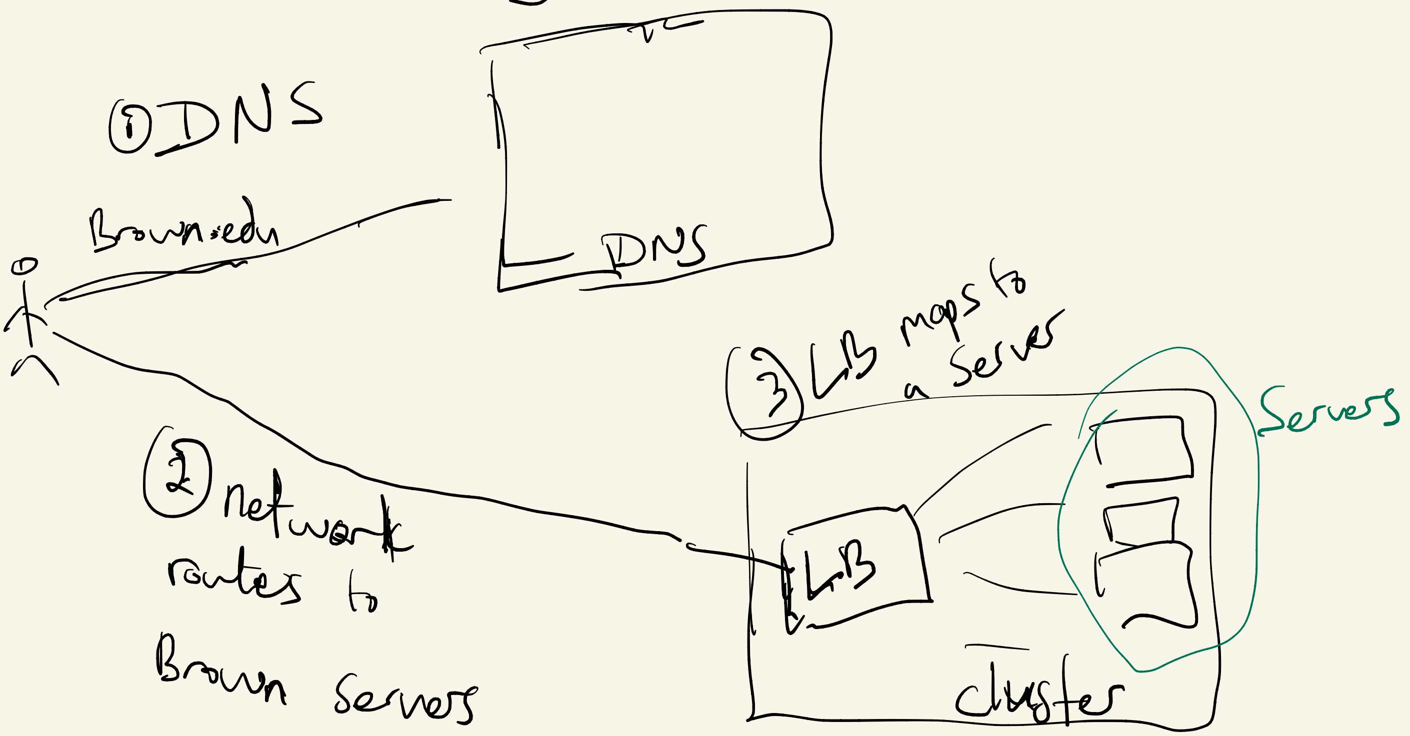
each call will return a different line

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) ⇒ distribute load btw servers

**Global LB**

① reduce latency (get you a close cluster)

② data policy

| | Global LB | Local LB |
|---|---|---|
| Goals | ① reduce latency to client /cluster<br>② respect data policy | ① distribute load btw servers |
| techniques | ① DNS FB<br>② BGP Azure | ① Round Robin<br>② Random Amazon EC2<br>~~③ Modulo hash~~ Google/FB<br>④ Consistent hashing |

## 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)

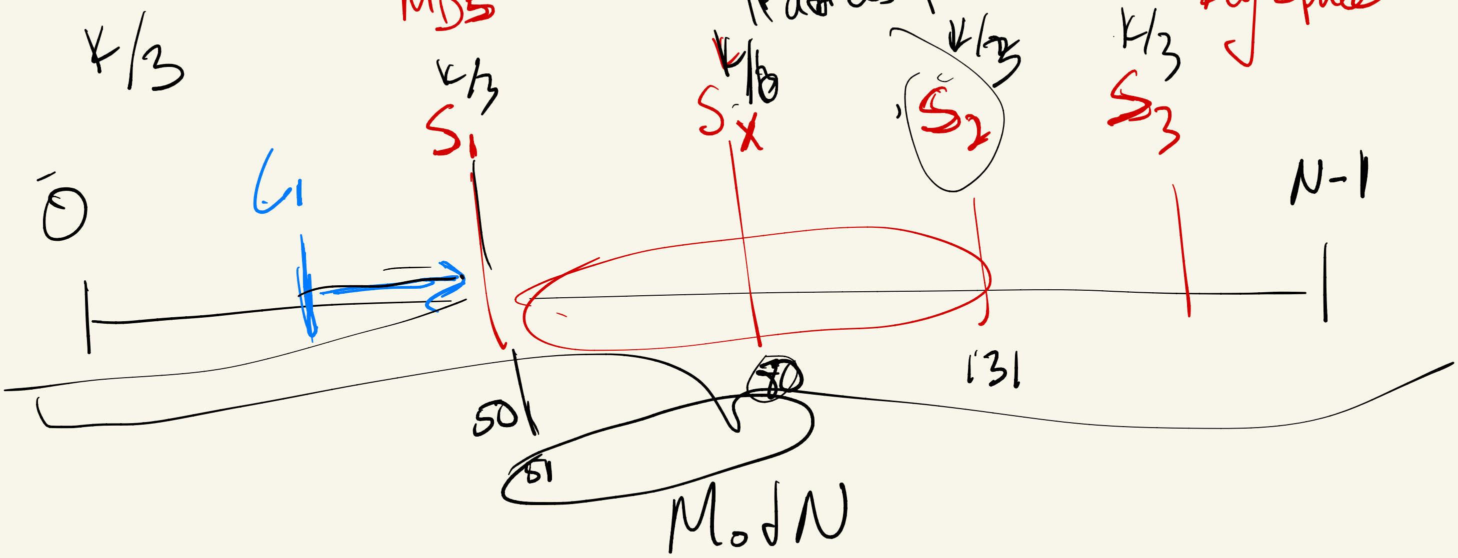
Consistent hashing → consistently assigns client/user to the same server

ID = hash(string) mod N

MD5

IPadress:port

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)
(X) 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)

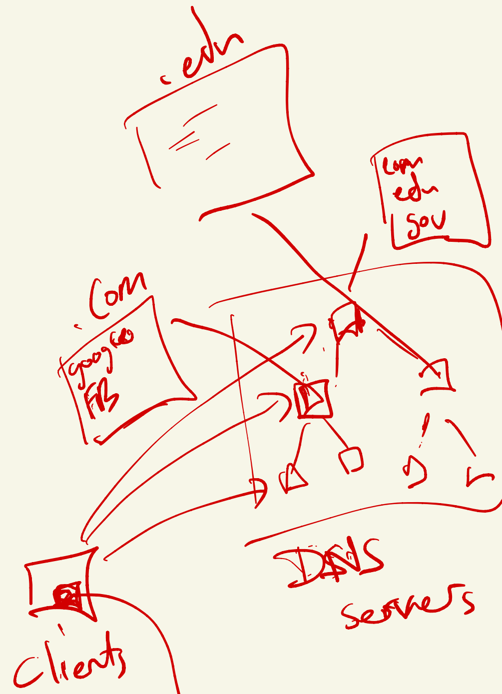
K = # of clients
N = # of server
each server = K/N (clients are evenly distributed)

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)
(X) 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?

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

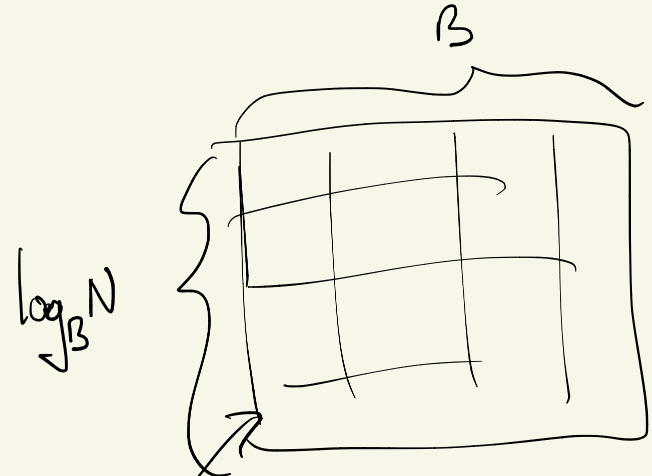



cache reduces load on infra

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

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

# Tapestry




Size of routing table

$C * \log_B N * B$

# of slots

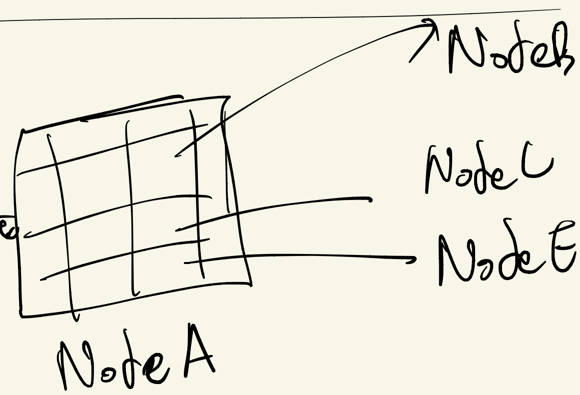
each row guaranteed to have at least one entry: the node itself

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

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

---

## Backpointers




① Grateful exit: use backpointer to determine which nodes to update

# Exit

| Graceful | UnGraceful |
| --- | --- |
| 1 update routing tables of nodes in your backpointer structure | 1 heartbeats |
| 2 move objects to new root locations | 2 bad nodes detected via RPC calls |

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 $\longrightarrow$ 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

Total, FIFO → logical clocks

Causal → Vector clocks

## Global state (distributed snapshots)

- Time based → time synchronization
- manual → time to go to all servers
- continuous → only valid snapshot maybe @ the beginning of the system
- ~~Chandy Lamport~~ → very strong assumptions
  1. no server will fail
  2. the n/w never reorders msgs

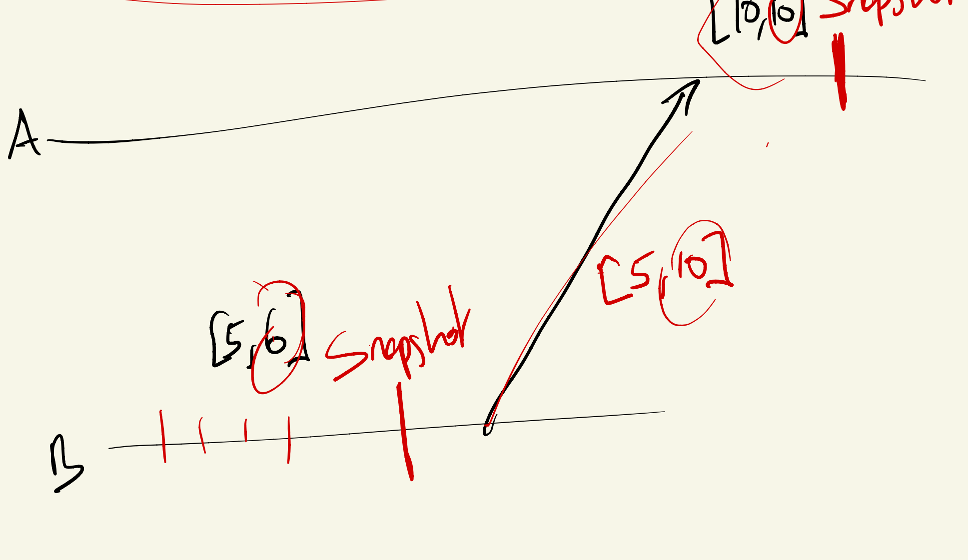
Consistent snapshots : iff for every "recv" event in the distributed snapshot you also have the "send" event.

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

$$VC[A, B]$$

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



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:

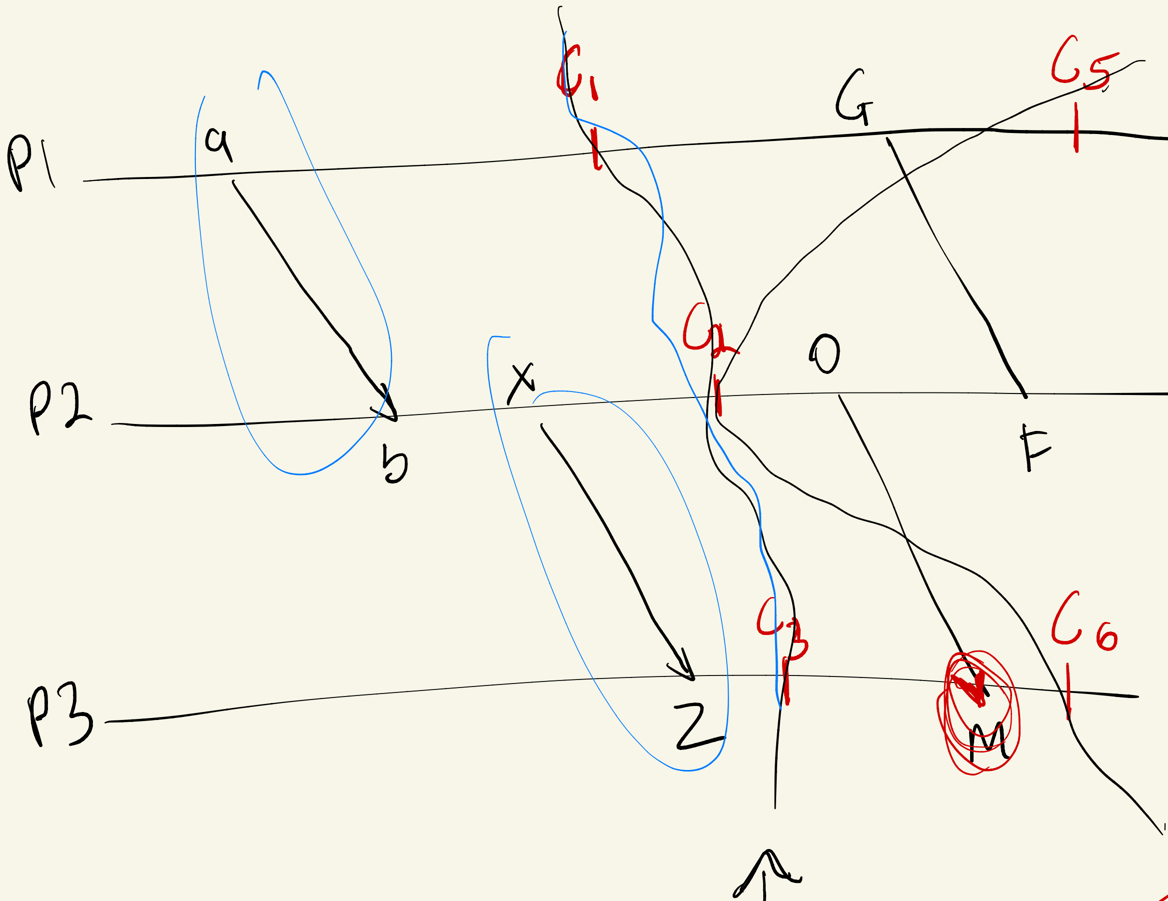
total and FIFO

( ) 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~~
(X) S1: X1, Z2, X2, Z1   S2: X1, Z2, X2, Z1

not total

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

(X) 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

MapReduce (Liteminer)

Networking (TCP/UDP, RPC)

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

Time (logical / Vector / snapshots)