THE UNIVERSITY *of* EDINBURGH

Distributed Systems Fall 2024

Yuvraj Patel

Today's Agenda

Exam Format

Exam Preparation

Revision

Exam Format

Exam Preparation

Lecture 1 – What is a Distributed System?

A distributed system

- Multiple computers (or nodes) communicate via a network
- Work together to achieve some task together/collectively
- Appears as a single coherent system to the users
- Failure of a node you didn't even know existed can render your own node unusable

Examples

- Distributed Systems are ubiquitous

Why study Distributed Systems?

Inherently distributed

- Either necessarily or sufficiently

For better reliability

- If one node fails, the system continues to work

For better performance

- Get things done faster; for example, due to replication (from a nearby datacenter)

To solve bigger and complex problems

- Single node cannot handle all the data/processing capacity, etc.
- Efficiently solve a problem

Distributed Systems Architectural Styles

A style is formulated in terms of

- (Replaceable) components with well-defined interfaces
- The way that components are connected to each other
- The data exchanged between components
- How these components and connectors are jointly configured into a system

Connector

- A mechanism that mediates communication, coordination, or cooperation among components

Different styles

- Layered – Components organized in a layered style
- Service-Oriented – Multiple types – object-based, microservices, resource-based
- Publish-Subscribe – Strong separation between processing and coordination; event based and shared data-space

Distributed System Architecture

Organize system based on where the software components are placed

Three main types

- Centralized
  - Basic Client-server model
  - Clients – Processes using services; Servers – Processes offering services
- Decentralized
  - Peer-to-peer systems; Client and server physically split up into logical equivalent parts
  - Two types of topology – Structured (deterministic) and unstructured (random)
- Hybrid
  - Mix of centralized and decentralized – Cloud Computing, Edge Computing

Lecture 2– Remote Procedure Call

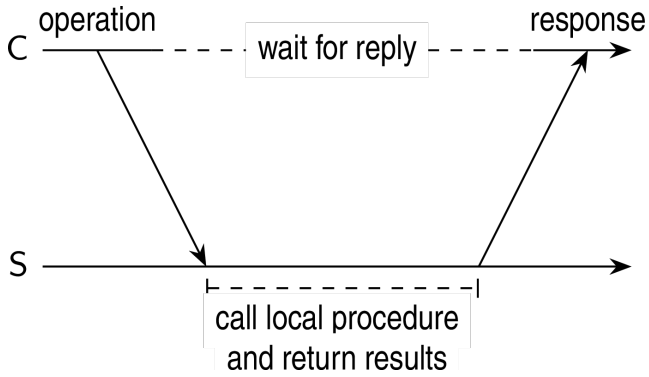
Allow remote services to be called as procedures

- Transparency with respect to location, implementation, language, etc.

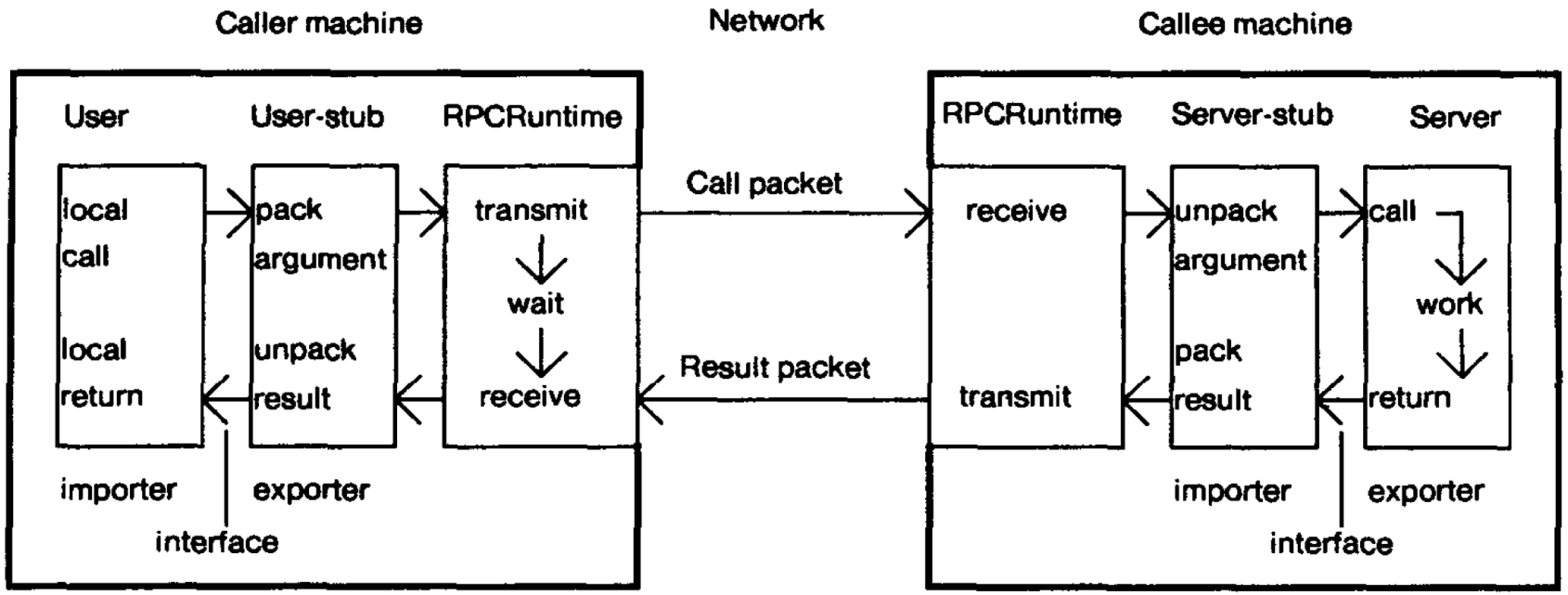
Goal is to make distributed computing look like centralized computing

Basic Idea

- Programs can call procedures on other machines
- When process A calls a procedure foo() on machine B, A is suspended
- Execution of foo() takes place on machine B
- After execution of foo(), the result is sent back to A, which resumes execution



RPC Overall Flow



Lecture 3 – System Models

Two thought experiments

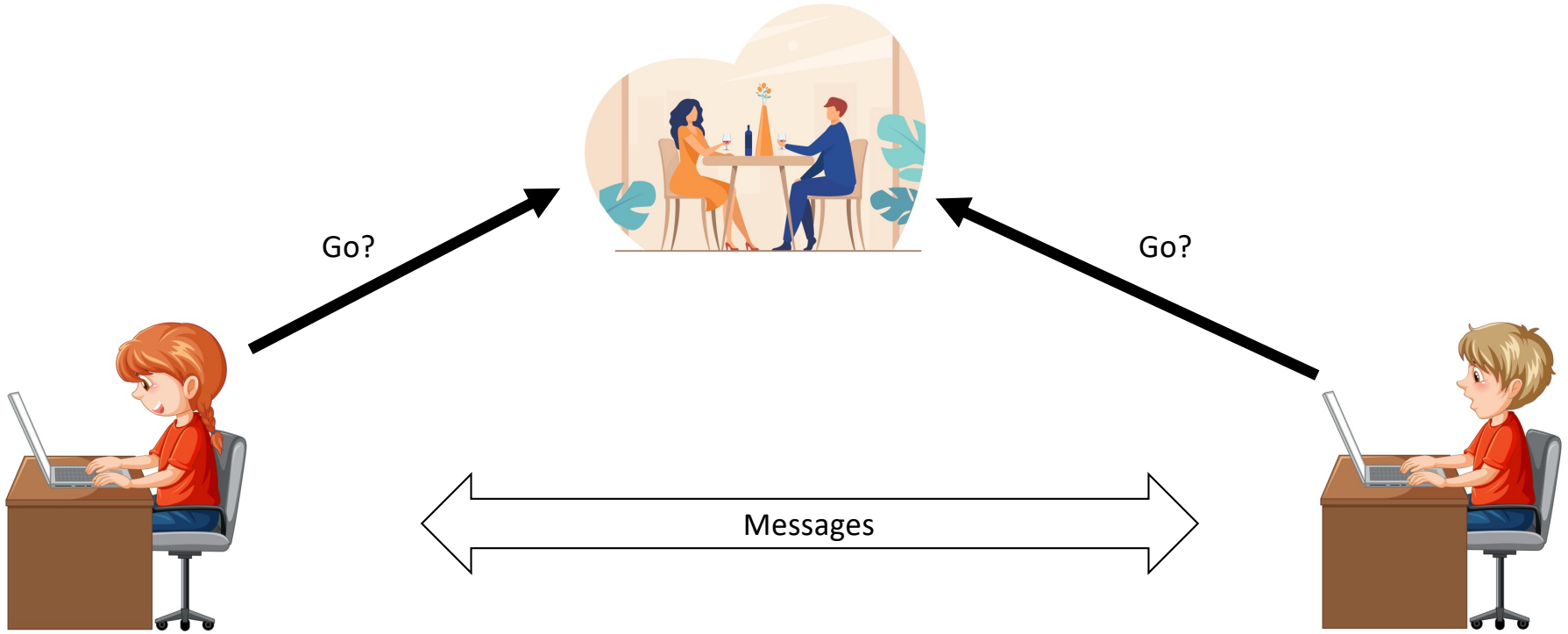
- Two generals problem – a model of networks
- Byzantine generals problem – a model of nodes

Nodes and network both can be faulty

System model captures our assumptions about nodes and the network behavior

- Abstract description of the properties
- Implementation may vary depending on the technology/language used
- Network, Node, Timing behavior

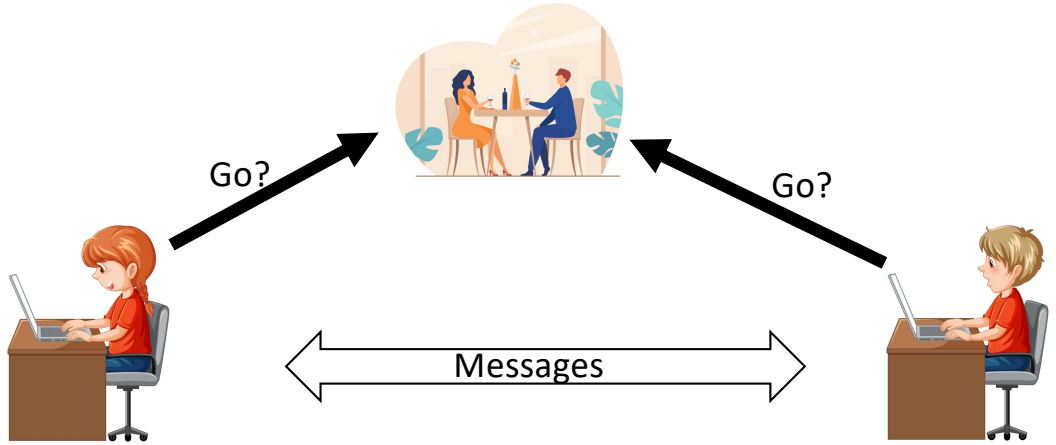
Thought experiment 1: A romantic date…



Designed by Freepik

Thought experiment 1: A romantic date…

Desired outcome – Boy goes if and if only Girl goes, or vice versa



Boy	Girl	Outcome
Do not go	Do not go	Nothing happens
Goes	Does not go	Poor boy 💔
Does not go	Goes	Poor girl 💔
Goes	Goes	Great romantic date 💞

Designed by Freepik

Thought Experiment 2: Byzantine General Problem

Problem – Byzantine general's problem

- Up to f nodes (friends) might behave maliciously
- Honest nodes (friends) don't know who the malicious ones are
- The malicious nodes (friends) may collude
- Nevertheless, honest nodes (friends) must agree on a plan

Theorem

- Need 3f + 1 nodes (friends) in total to tolerate f malicious nodes (friends)
- Cryptography may help

Key Message

- How do you make sure that multiple entities, which are separated by distance, are in absolute full agreement before an action is taken?

Nodes & Network Behavior

Network behavior

- Reliable – Message received if it is sent; Messages can be re-ordered
- Fair-loss – Messages may be lost, duplicated, or reordered
- Arbitrary – Malicious adversary may eavesdrop, modify, drop, spoof, replay

Node Behavior – Halting behavior

- Fail-stop – Crash failures, can reliably detect failures
- Fail-noisy – Crash failures; eventually reliably detect failure; noisy behavior
- Fail-silent – Omission or crash failures; clients cannot tell what went wrong
- Fail-safe – Arbitrary; yet benign failures (no harm)
- Fail-arbitrary (Byzantine) – Arbitrary, with malicious failures

Lecture 4 – Ordering of Events

Clock synchronization is one approach to order events

Avoid absolute time as it may not be accurate (and less important)

For ordering, we need causality to be determined

Happens-before relation

- An event is something happening at one node
- Event A happens before Event B (written as A→B) iff:
  - A and B occurred on the same node, and A occurred before B in that node's local execution order
  - Event A is the sending of some message M, and event B is the receipt of that same message M
- There exists an event C such that A→C and C→B

Happens-before relation is a partial order

- A→B or B→A is not possible
- In that case, A and B are concurrent (written A || B)

Logical Clocks

Assign logical timestamp to each event where timestamp obeys causality

Rules/Algorithm

- Each process maintains a counter value; On init, counter value is 0
- Each process increments its counter when a send or an instruction is executed.
- A send message event carries the counter (timestamp)
- For a receive message event, the recipient process will update its local counter max(local counter, message counter (timestamp)) + 1

Properties

- If a → b then value-of-counter(a) < value-of-counter(b)
- However, if value-of-counter(a) < value-of-counter(b), does not imply a→b

Lamport clock not guaranteed to be ordered or unequal for concurrent events

Logical Clocks Example

Vector Clocks

Assume n nodes in the system, N = <N1, N2… Nn>

Vector timestamp of an event a is V(a) = <t1, t2…tn>

ti is the number of events observed on node Ni

Each node has a current vector timestamp T

On event at node Ni, increment vector element T[i]

Attach current vector timestamp to each send message

Recipient merges message vector into its local vector

Vector Clocks (contd…)

Each node has a current vector timestamp T having n elements for n nodes

On event at node Ni, increment vector element T[i]

Rules/Algorithm

- On initialization at node Ni,
  - do T = <0, 0, …0>
- On any event occurring at node Ni,
  - do T[i] = T[i] + 1
- On Send event while sending message M at node Ni,
  - do T[i] = T[i] + 1;
  - Send (T, M)
- On Receiving event message M at node Nj, do
  - T[j] = T[j] + 1
  - T[k] = max(T[k], T'[k]), for every $k \neq j$ and $k \in \{1,\ldots n\}$

Vector Clocks Example

Detecting Global Properties – Global Snapshots

Sometimes it is necessary to have a global view of the system

- Checkpointing to support restarting the system post failure
- Garbage collection of objects
- Deadlock detection
- Termination of computation
- Debugging in general

Global Snapshot = Global State

Global state comprises

- Individual state of each process in the system
- Individual state of each communication channel in the system

Capture instantaneous state of each process and the communication channels

Consistent Cut

A cut is a set of cut events (or snapshot), one per node, each of which captures the state of the node on which it occurs

A cut C = {c1, c2, … cn} is consistent, iff

- for (each pair of events e, f in the system)
- event e is in the cut C, and if f --> e, then event f is also in the cut C

The cut events in a consistent cut are not causally related

- The cut is a set of concurrent events, and a set of concurrent events is a cut

Lecture 5 – Consensus with Paxos

Two phases – Prepare & Accept

Prepare Phase

- Find out any chosen values so far
- Block older and uncompleted proposals

Accept Phase

- Inform acceptors to accept a specific value

Algorithm – Prepare Phase

Proposer

- Choose proposal number n, send <prepare, n> to acceptors

Acceptor

- Only receiving a prepare message
  - If $n > n_h$, where n_h is the highest proposal seen so far by the acceptor

    $n_h = n$. (Promise to not accept older proposals)

    If no prior proposal accepted,

    reply <promise, n, NULL>

    Else

    reply <promise, n, (n_a, v_a)>
  - Else

    Reply <prepare-failed>

Algorithm – Accept Phase

Proposer

- If receive promise from majority of the acceptors,
  Determine any earlier chosen value v_a for n_a and choose latest value or any value v selected by the proposer
  send <accept, n, v> to acceptors

Acceptors

- If n >= n_h
  $n_a = n_h = n$
  $v_a = v$
- reply <accept, n_h>

Proposer

- When responses received from the majority
  If any $n_h > n$
  Start from prepare phase again
  Else
  Value is chosen

Paxos flow

Raft Basics

A node can be either follower, candidate, or leader

Raft divides time into terms of arbitrary length; terms are numbered consecutive integers

Each term begins with an election, where one or more candidates attempts to become a leader

- Two possible outcomes of an election – leader elected or split vote

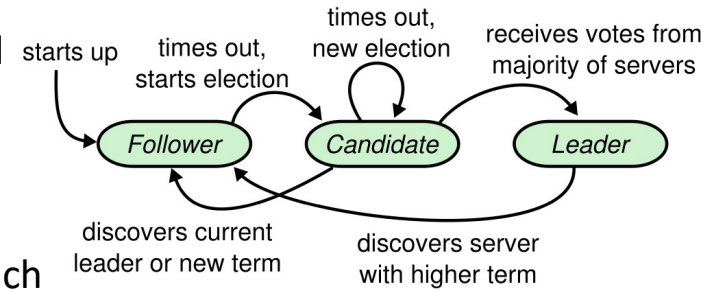
Term acts as a logical clock and helps detect obsolete information such as stale leaders

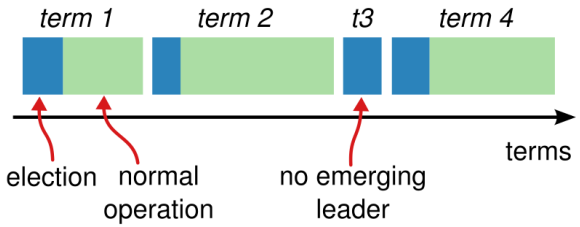
Each node stores a current term number, increases monotonically

Current terms exchanged while normal communication

- One node's current term smaller than others, it updates it term to larger value
- If leader/candidate discovers its term is out of date; revert to follower role

If node receives a request with a stale term number, reject the request





High-Level Understanding

Log entries over time

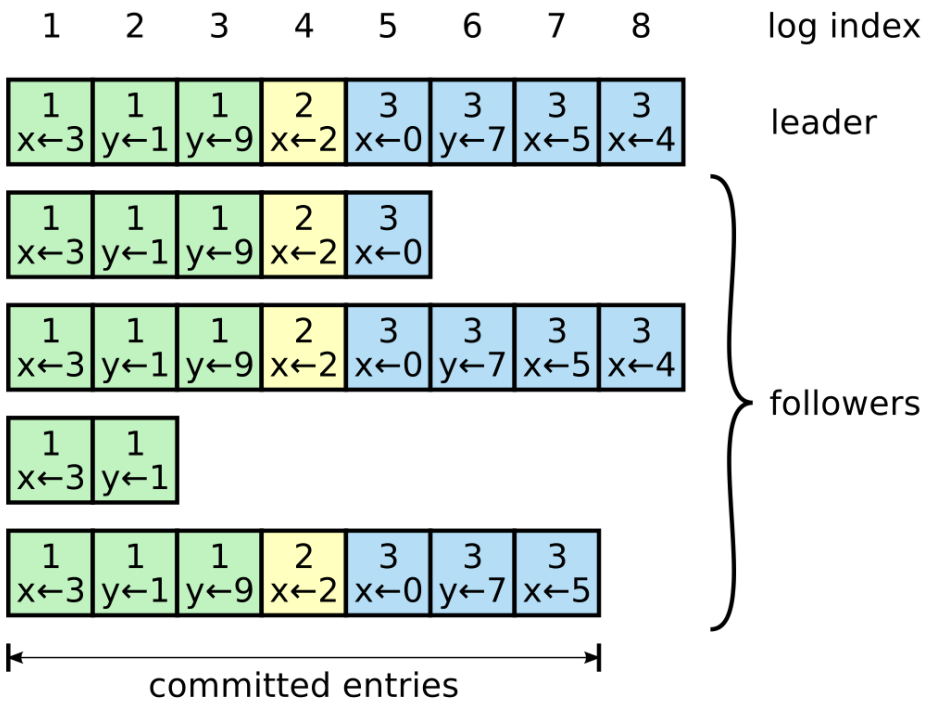
A leader's log is the ultimate truth

While election, ensure that the leader has all committed entries

Leader keeps track of each follower's log

Leader ensures all followers are up to date

- Either remove uncommitted log entries or append to log entries



Lecture 6 – Mutual Exclusion & Concurrency

Concurrency leads to non-deterministic behavior

- Different results even with same inputs

Race conditions: Specific type of bug

- Sometimes program works fine, sometimes it doesn't; depends on timing

Want to execute instructions as an uninterruptable group

- Want them to be atomic; appears that all execute at once, or none execute

Uninterruptable group of code is called critical section

Mutual exclusion for critical sections

- If thread A is in critical section C, thread B isn't
- It is fine if other threads do unrelated work

Distributed Locks

Cannot share local lock variables

Mutual exclusion in a distributed system

Central Solution

- Elect a central leader using election algorithm
- Leader keeps a queue of waiting requests from nodes who wish to access

Decentralized approach

- All nodes involved in the decision making of who should access the resource
- Ricart-Agrawala Algorithm – Use the notion of causality – rely on logical timestamps
- Token Ring Algorithm -- All nodes arranged in a ring fashion; Use token as a means of ownership

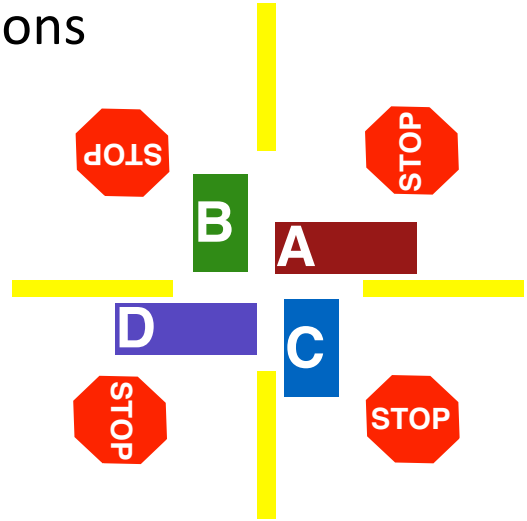
Deadlocks

No progress can be made because two or more nodes are each waiting for another to take some action and thus each never does

Deadlocks can only happen with these four conditions

1. Mutual Exclusion
2. Hold-and-wait
3. No preemption
4. Circular Wait

Can eliminate deadlock by eliminating any one condition



Lecture 7 – Replication

Replicate data at one or more sites can help with

- Availability & Fault Tolerance
  - If primary server crashes, secondary can takeover => Highly available service
  - Mask node crashes => Transparency
- Performance
  - Concurrent Reads can be served from multiple servers improving performance
- Scaling
  - Size scalability – Prevent overloading a single server

Having multiple copies, means that when any copy changes, the change needs to be propagated to all other copies

- Need replicas to have same data, i.e., they should be kept consistent

Efficiently synchronize all replicas a challenging problem

Consistency Models

A consistency model is a contract between the programmer and a system

- The system guarantees that if the programmer follows the rules for operations on data, data will be consistent
- Result of the reading, writing, updating data will be predictable

Two consistency models

- Data-centric consistency models – Defines consistency as experienced by all the clients; provides a system wide consistent view on the data store
- Client-centric consistency models – Defines consistency of the data store only from one client's perspective; Different clients might see different sequences of operations at their replicas

Strong Data-Centric Consistency

Strong Consistency Models

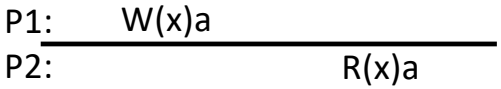
- Operations on shared data are synchronized without synchronization operations

Few options

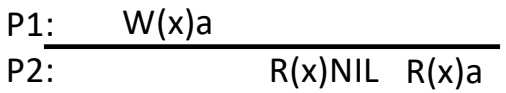
- Strict Consistency – Absolute time ordering of all shared accesses matters
- Sequential Consistency – All processes see all shared accesses in the same order
- Linearizability – Sequential Consistency + Operations are ordered according to a global time

Strong Data-Centric Consistency (contd…)

Strict Consistency

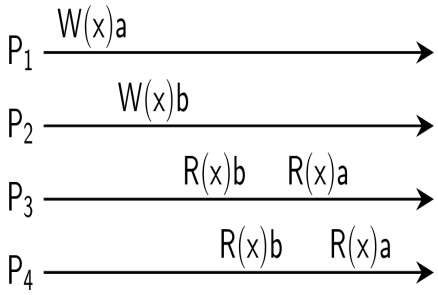


Strictly Consistent Data Store

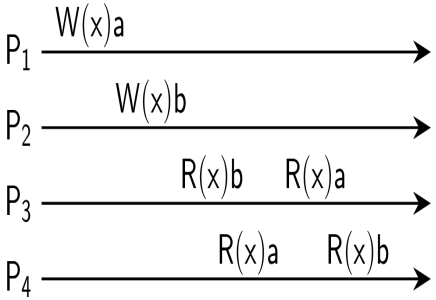


Not Strictly Consistent Data Store

Sequential Consistency

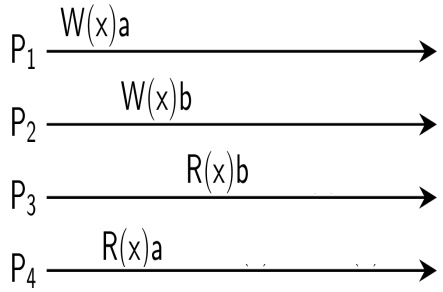


Sequentially Consistent Data Store

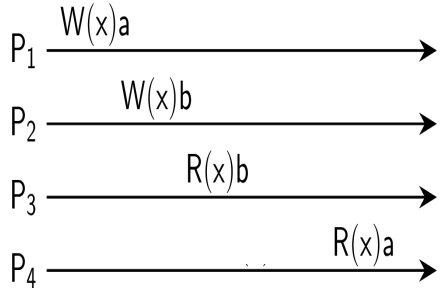


Not Sequentially Consistent Data Store

Linearizable Consistency



Linearizable Consistent Data Store



Not Linearizable Consistent Data Store

Lecture 8 – Client-Side Consistency Models

Strong Consistency – See all previous writes
- Possible Values – 2-5

Eventual Consistency – See subset of previous writes
- Possible Values – 0-0, 0-1, 0-2, 0-3, 0-4, 0-5, 1-0, 1-1, 1-2, 1-3, 1-4, 1-5, 2-0, 2-1, 2-2, 2-3, 2-4, 2-5

Consistent Prefix – See initial sequence of writes
- Possible Values – 0-0, 0-1, 1-1, 1-2, 1-3, 2-3, 2-4, 2-5

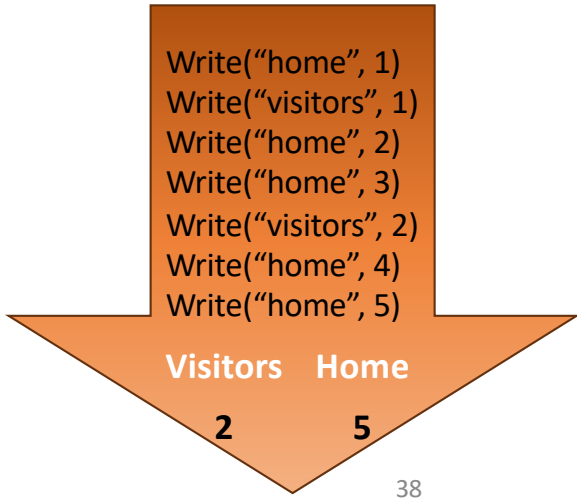
Bounded Staleness – See all "old" writes
- Possible Values – 2-3, 2-4, 2-5 (At most one innings out-of-date)

Monotonic Reads – See increasing subset of writes
- Possible Values – After reading 1-3, 1-3, 1-4, 1-5, 2-3, 2-4, 2-5

Read My Writes – See all writes performed by the reader
- Possible Values – 2-5 (Writer); 0-0, 0-1, 0-2, 0-3, 0-4, 0-5, 1-0, 1-1, 1-2, 1-3, 1-4, 1-5, 2-0, 2-1, 2-2, 2-3, 2-4, 2-5 (Everyone else)

	1	2	3	4	5	6	7	8	9	RUNS
Visitors	0	0	1	0	1	0	0			2
Home	1	0	1	1	0	2				5

Existing Score



Lecture 9 – Distributed File-Systems

Network File System (NFS)

- Stateless: Servers do not remember clients or open files
- Different behavior for non-idempotent operations like rmdir, mkdir
- NFS handles client and server crashes very well
- Caching Update Visibility Problem – Server doesn't have latest version
  - Flush data on client on close() or at other times (optionally)
  - Odd consistency model – File flushes not atomic; Mix of updates within file possible
- Caching Stale Cache – All clients do not have the latest version from server
  - Clients recheck if cached copy is current before using data
  - Clients could read data that is up to 3 seconds old

Lecture 9 – Distributed File-Systems

Andrew File System (AFS)

- Stateful design: Servers remembers clients and open files
- Caching Update Visibility Problem
  - Cache whole file instead of blocks
  - Last writer wins (i.e., the last file close wins); no data mix up like NFS
- Caching Stale Cache Problem
  - On open() by a client, ask for callback from server if file changes
  - Server tells clients when data is overwritten

END OF LECTURES
Good luck with the exam

Chandy-Lamport Snapshot Algorithm

Initiator Pi records it own state

Initiator process creates special messages called “Marker” messages

For j = 1 to N except i

- Pi sends out a Marker message on outgoing channel Cij
- Starts recording the incoming messages on each of the incoming channels at Pi: Cji (for j = 1 to N except i)

Chandy-Lamport Snapshot Algorithm

Whenever a process Pi receives a Marker message on an incoming channel Cki

- If (this is the first Marker Pi is seeing)
  - Pi records its own state first
  - Marks the state of the channel Cki as “empty”
  - for j = 1 to N except I
    - Pi sends out a Marker message on outgoing channel Cij
  - Starts recording the incoming messages on each of the incoming channels at Pi: Cji (for j = 1 to N except i and k)
- Else (Marker already seen)
  - Marj the state of the channel Cki as all the messages that have arrived on it since recording was turned on for Cki

Chandy-Lamport Snapshot Algorithm

Algorithm terminates when

- All the processes have received a Marker to record their own state
- All the process have received a Marker on all the (N – 1) incoming channels to record the state of all the channels

A central authority(server/entity) may collect all the local state of the processes and the communication channels to obtain the full global snapshot

Any run of the algorithm creates a consistent cut

Chandy-Lamport Snapshot Algorithm Example