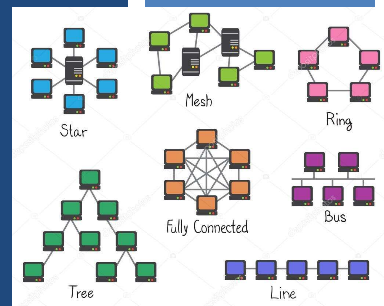
TCSS 558: APPLIED DISTRIBUTED COMPUTING



Consensus, Consistency, Replication

Wes J. Lloyd
Institute of Technology
University of Washington - Tacoma

OBJECTIVES

- Assignment #2 Questions
- Assignment #3 Questions
- Review Quiz #2
- Assignment #1 Feedback
- Feedback from 12/5

- Raft Consensus Algorithm
- Ch. 7 – Consistency and Replication
  - Introduction
  - Data centric consistency models

ASSIGNMENT #1 FEEDBACK

- UDP "store" command
  - For the LARGE test file, since UDP does not automatically split messages into multiple packets, it is easy to exceed a statically defined byte array size
  - Many folks used [1024] bytes
  - Two strategies to address this:
  - (1 – CHEAP SOLUTION) *(instructor did this)*
    Extend to the largest allowable UDP packet size
    - Set to ~65,000 bytes
    - Append a "message truncated" message at the end

December 7, 2017	TCSS558: Applied Distributed Computing [Fall 2017] Institute of Technology, University of Washington - Tacoma	L19.3

ASSIGNMENT #1 - FEEDBACK

- (2 – THE RIGHT WAY)
  Break message into multiple numbered packets
  - Start UDP communication with client by sending total number of messages (packets = total size / 1024)
  - Wait until client ***echoes back*** this number
  - Send messages of 1024 bytes each
  - Begin each with a monotonically increasing ID
  - Client knows how many messages it should receive
  - If any message is lost, client gets an opportunity to ask for messages to be replayed at end
  - Client assembles "store" results from multiple packets
    - *UDP messages could be out of order*

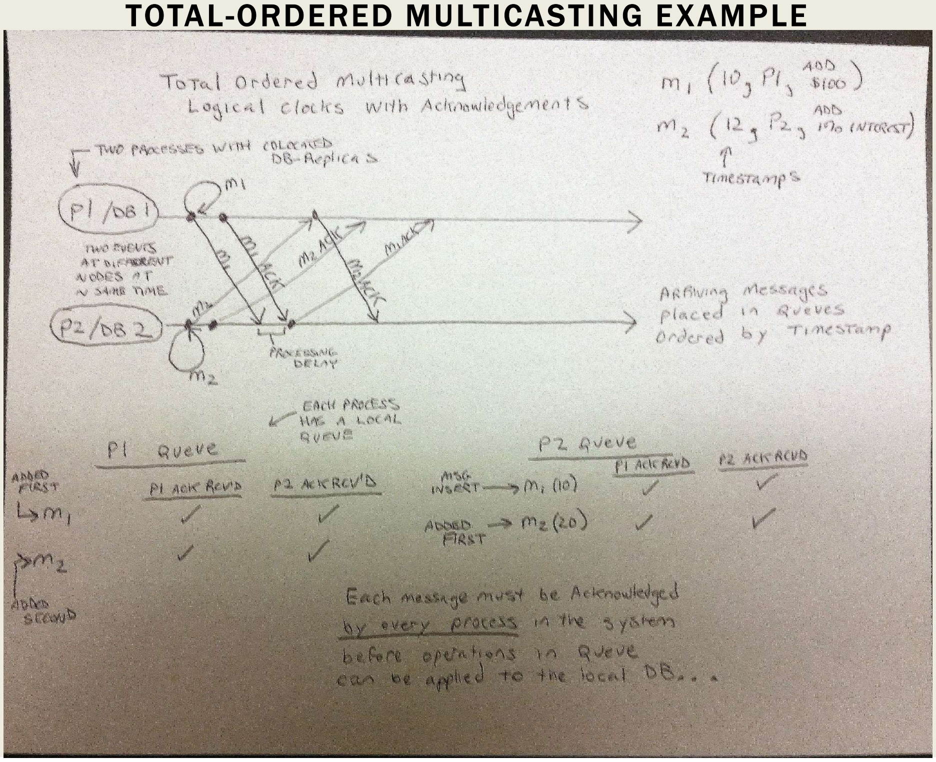
December 7, 2017	TCSS558: Applied Distributed Computing [Fall 2017] Institute of Technology, University of Washington - Tacoma	L19.4

FEEDBACK FROM 12/5

- **From Quiz #2:**
- **Question #3**
- For total ordered multicasting if there are two processes, both sharing data element X, and initially X=10.
- (a) How many messages does P1 receive, when the only operation is *by* P1: X=X+100 ?
- (b) If P1 performs X=X+100 at Lamport Clock (20), and P2 performs X=X*2 at Lamport Clock (10), what is X's value with total ordered multicasting?

December 7, 2017	TCSS558: Applied Distributed Computing [Fall 2017] Institute of Technology, University of Washington - Tacoma	L19.5

TOTAL-ORDERED MULTICASTING EXAMPLE



FEEDBACK – 2

- (c) Using total ordered multicasting, how many messages are exchanged by P1 and P2 to perform:
- P1 (clock=20) X=X+100
- P2 (clock=10) X=X*2
- Recall the whiteboard…

FEEDBACK - 3

- What does it mean, "ways logs can diverge"
- RAFT, by using a leader, limits the number of ways logs (across the nodes) can become out of sync
- The leader's log is always assumed to be the "master" copy.
- **Ways logs can diverge**
- (a) **Follower** may be missing entries present on **leader**
- (b) **Follower** may have extra entries not present on the **leader**
- (c) Both A and B
- Disagreements are resolved by overwriting follower's logs with the leader's
- The election safety property ensures that the leader will always have an up-to-date log.
- **Majority rules** in RAFT elections (and log certification)

FEEDBACK - 4

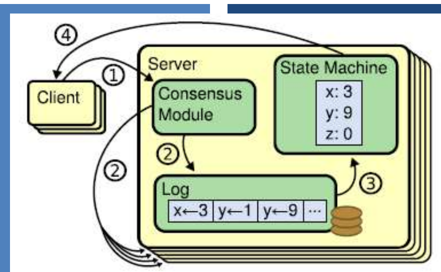
- **Where should the it be Intermediate concurrent hash table in assignment 2 deployed?**
- Each node should maintain a list of keys which are presently involved in put or del transactions
- Just one transaction is allowed at any time on the same key
- If a node finds a key is already involved in another transaction (*by checking the concurrent hash table*) it REJECTS the **dput1** request
  - The transaction originator then sends **dputabort** instead of **dput2**
- If servers are multi-threaded, there could be multiple concurrent transactions to alter many keys simultaneously
- **Improvement:** the originator, after failing the transaction across the nodes, could retry the transaction, perhaps up to 10x
  - Not a requirement for Assignment 2

FEEDBACK - 5

- I'm confused about port mapping when using SWARM mode

```
docker service create --name kvservice --
replicas=5 --network overnet --publish
1234:1234 kvstore
```

- Publishing port makes the service available from ***any*** docker-machine in the swarm by accessing its IP and port
- Syntax is: --publish <external port>:<container port>
- Access to the external port of ***any*** docker-machine in the swarm will be routed to the internal port on any service container (*Presumably in round-robin fashion*)
- Feature is similar to load balancing; provided by docker swarm



RAFT CONSENSUS

L19.11

State

Persistent state on all servers:
(Updated on stable storage before responding to RPCs)

currentTerm	latest term server has seen (initialized to 0 on first boot, increases monotonically)
votedFor	candidateId that received vote in current term (or null if none)
log[]	log entries; each entry contains command for state machine, and term when entry was received by leader (first index is 1)

Volatile state on all servers:

commitIndex	index of highest log entry known to be committed (initialized to 0, increases monotonically)
lastApplied	index of highest log entry applied to state machine (initialized to 0, increases monotonically)

Volatile state on leaders:
(Reinitialized after election)

nextIndex[]	for each server, index of the next log entry to send to that server (initialized to leader last log index + 1)
matchIndex[]	for each server, index of highest log entry known to be replicated on server (initialized to 0, increases monotonically)

AppendEntries RPC

Invoked by leader to replicate log entries (§5.3); also used as heartbeat (§5.2).

Arguments:

term	leader's term
leaderId	so follower can redirect clients
prevLogIndex	index of log entry immediately preceding new ones
prevLogTerm	term of prevLogIndex entry
entries[]	log entries to store (empty for heartbeat; may send more than one for efficiency)
leaderCommit	leader's commitIndex

Results:

term	currentTerm, for leader to update itself
success	true if follower contained entry matching prevLogIndex and prevLogTerm

Receiver implementation:

1. Reply false if term < currentTerm (§5.1)
2. Reply false if log doesn't contain an entry at prevLogIndex whose term matches prevLogTerm (§5.3)
3. If an existing entry conflicts with a new one (same index but different terms), delete the existing entry and all that follow it (§5.3)
4. Append any new entries not already in the log
5. If leaderCommit > commitIndex, set commitIndex = min(leaderCommit, index of last new entry)

RequestVote RPC

Invoked by candidates to gather votes (§5.2).

Arguments:

term	candidate's term
candidateId	candidate requesting vote
lastLogIndex	index of candidate's last log entry (§5.4)
lastLogTerm	term of candidate's last log entry (§5.4)

Results:

term	currentTerm, for candidate to update itself
voteGranted	true means candidate received vote

Receiver implementation:

1. Reply false if term < currentTerm (§5.1)
2. If votedFor is null or candidateId, and candidate's log is at least as up-to-date as receiver's log, grant vote (§5.2, §5.4)

Rules for Servers

All Servers:

- If commitIndex > lastApplied: increment lastApplied, apply log[lastApplied] to state machine (§5.3)
- If RPC request or response contains term T > currentTerm: set currentTerm = T, convert to follower (§5.1)

Followers (§5.2):

- Respond to RPCs from candidates and leaders
- If election timeout elapses without receiving AppendEntries RPC from current leader or granting vote to candidate: convert to candidate

Candidates (§5.2):

- On conversion to candidate, start election:
  - Increment currentTerm
  - Vote for self
  - Reset election timer
  - Send RequestVote RPCs to all other servers
- If votes received from majority of servers: become leader
- If AppendEntries RPC received from new leader: convert to follower
- If election timeout elapses: start new election

Leaders:

- Upon election: send initial empty AppendEntries RPCs (heartbeat) to each server; repeat during idle periods to prevent election timeouts (§5.2)
- If command received from client: append entry to local log, respond after entry applied to state machine (§5.3)
- If last log index ≥ nextIndex for a follower: send AppendEntries RPC with log entries starting at nextIndex
  - If successful: update nextIndex and matchIndex for follower (§5.3)
  - If AppendEntries fails because of log inconsistency: decrement nextIndex and retry (§5.3)
- If there exists an N such that N > commitIndex, a majority of matchIndex[i] ≥ N, and log[N].term == currentTerm: set commitIndex = N (§5.3, §5.4).

LOG REPLICATION

- Leader receives commands forwarded from followers
- Ways logs can diverge
- (a) Follower may be missing entries present on leader
- (b) Follower may have extra entries not present on the leader
- (c) Both A and B
- Because raft uses a “coordinator” node to achieve consensus the number of possible ways logs can diverge is limited
- Raft leaders FORCE followers logs to match its own
- Conflicting entries in follower logs are overwritten

LOG REPLICATION - 2

- FOR THE WHOLE SYSTEM THERE IS JUST ONE MONOTONICALLY INCREASING LOG INDEX
  - Akin to Lamport’s Clocks
- Possible *follower* states at start of new term
- (a) Missing entries
- (b) Extra uncommitted entries
- (c) Both

RAFT - LOG REPLICATION ALGORITHM

- **Leader**:
1. Receives command(s)
2. Appends commands to local log (concurrent hash table)
3. Sends AppendEntries() to **followers**

- **Leader** tracks index of its highest committed log entry
- Provides this index to **followers** in AppendEntries() RPC

- **Leader commit to state machine:**
- (1) When log entries replicated at a majority of the **followers**, **leader** commits to its state machine (KV-store)

LOG REPLICATION ALGORITHM - 2

- **Synchronizing follower logs**
- (2) If **follower** rejects AppendEntries() then **leader** decrements its "follower-nextIndex" by one, and *retries* AppendEntries().
  - "follower-nextIndex" tracks which logs entries are sent to the follower for each AppendEntries() RPC call
- Loop continues until **leader** ***walks back*** its "follower-nextIndex" until it **matches** what is committed at the **follower**
  - **Follower** has a **commitIndex**
  - Tracks 1st phase of a "two-phase" commit
  - **Follower** has a **lastApplied** index
  - Tracks 2^{nd} phase of "two-phase" commit
- Once **leader** matches follower-nextIndex, the **follower** accepts the AppendEntries() RPC, and writes data to its log
  - Conflicting log entries are overwritten

LOG REPLICATION ALGORITHM - 3

- Leader based consensus algorithms require the leader to "eventually store" all committed log entries
- Raft handles follower node failure by retrying communication indefinitely
  - If crashed server restarts, the log will be resurrected, and the follower's state machine will be restored (kv-store)

COMMITTING LOG ENTRIES

- Each node keeps a **commitIndex** and **lastApplied** index variable
- **PHASE I**
- Leader: when log message replicated at a majority of follower logs (not state machines) ****- described next slide***
- Leader increments its commitIndex
- Followers set commitIndex to Min (leader-commitIndex , index of last new log entry)

> If leaderCommit > commitIndex, set commitIndex = min(leaderCommit, index of last new entry)

- **PHASE II**
- For any node (follower, leader):
- If commitIndex > lastApplied
  - Increment lastApplied by 1
  - commit log[lastApplied] to **state machine** (kv-store)

> If commitIndex > lastApplied: increment lastApplied, apply log[lastApplied] to state machine (§5.3)

UPDATING COMMIT-INDEX OF LEADER

- If there exists an N such that N > commitIndex, a majority of matchIndex[i] ≥ N, and log[N].term == currentTerm: set commitIndex = N (§5.3, §5.4).

- ***How leader determines when to update it's commitIndex***
- Use a ***majority consensus*** of what has been committed at follower logs
- **Leader** maintains follower state arrays:
- **nextIndex[]:** index of next log entry to send to follower
- **matchIndex[]:** index of highest log entry known to be replicated (to log) at follower
- Find N, such that N > $\text{commitIndex}_{leader}$
- **and** a majority of matchIndex[i] ≥ N *(from followers)*
- **and** $\text{log\_entry}_{leader}$[N].term == $\text{currentTerm}_{leader}$
- **then** set $\text{commitIndex}_{leader}$ = N

RAFT CLUSTER MEMBERSHIP – A3

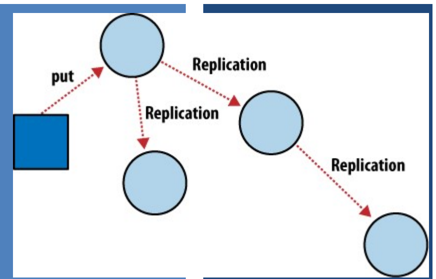
- Cluster discovery performed at startup
- Use any method:
  - Static file, UDP discovery (kv-store), TCP discovery (kv-store)
- Once membership is discovered, it can remain static/fixed
- Nodes can go offline, come back online
- Once a common configuration is propagated across the system, it can not be changed without restarting
- RAFT specifies a configuration change protocol where the system does a "hand-off" between an old and new configuration (section 6 of the paper)

A3 RAFT SIMPLIFICATIONS

- RequestVote() can be single threaded
  - AppendEntries() probably should have one thread per **follower**
- TCP client catch exceptions:
  - IOExcpetion – newSocket()
  - IOException – getOutputStream()
  - IOException – getInputStream()
  - **Leader** should catch exceptions, and retry requests indefinitely
  - Use socket method .setSoTimeout() to set a socket timeout in MS
- Node directory should generate and track nodeIDs
  - E.g. 1, 2, 3, 4, … n
- Node directory should retrieve a node by ID, or IP/PORT

A3 RAFT SIMPLIFICATIONS - 2

- **Leader** election: if using a single thread for election **candidate** should retry RequestVote() up to 10 times for a **follower** then give-up and move to next **follower**
- Instead of pushing data to **followers** when put() or del() is received by **leader**, can wait until next scheduled heartbeat to **follower**



CONSISTENCY AND REPLICATION

L19.23

WHY REPLICATE DATA?

(1) Fault tolerance: continue working after one replica crashes

(2) Provide better protection against corrupted data

(3) Performance

(3a) Scaling up systems (***scalability***)

- Replicate server, load balance workload across replicas

(3b) For providing geographically close replicas

- Replicas at the edge
- **MOVE DATA TO THE COMPUTATION**
- Performance ***perceived*** at the edge increases
- **But what is the cost of localized replication?**

December 7, 2017	TCSS558: Applied Distributed Computing [Fall 2017] Institute of Technology, University of Washington - Tacoma	L19.24

DATA REPLICATION COSTS

- Network bandwidth consumed maintaining replicas
  - Updates must be sent out and coordinated
- Maintaining consistency may be difficult
- All copies must be updated to ensure consistency
- **WHEN** and **HOW** updates need to be performed determines the prices of data replication…
- **Web caching example**
- Web browser caches local content to improve performance
- Doesn't know when content is "stale"
- **Solution:** Place server in charge of replication not browser
- Server invalidates and updates client cached copies
- Track how current copies are
- Degrades server performance → overhead from tracking, etc.

REPLICATION TRADEOFF EXAMPLE

- **Process P** accesses a local replica N times per second
- Replica is updated M times per second
- Updates involve complete refreshes of the data
- If N << M (very low access rate) many updates M are never accessed by P.
- Network communication overhead for most updates is useless.
- **TRADEOFFS:**
- Either move the replica away from P
  - *So the total number of accesses from multiple processes is higher*
- Or, apply a different strategy for updating the replica
  - *i.e. less frequent updates, possibly need based*
- **BALANCE TRADEOFF BETWEEN REPLICA ACCESS FREQUENCY AND COSTS OF REPLICATION** *(communication overhead)*

REPLICATION: SCALABILITY ISSUES

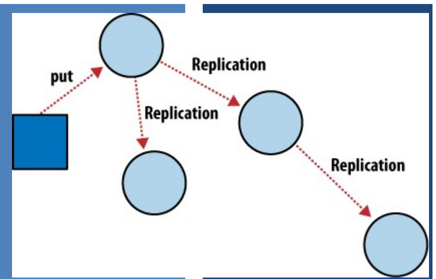
- **TIGHT CONSISTENCY**
- Reads must return same result
- Replication must occur after an update, before a read
- Provided by synchronous replication
- Update is performed across all copies as a single atomic operation (or transaction)
- **Assignment 2 replication is with tight consistency.**
- Keeping multiple copies consistent is subject to scalability problems
- May need global ordering of operations (e.g. Lamport clocks), or the use of a coordinator to assign order
- Global synchronization across a wide area network is time consuming (network latency)

December 7, 2017	TCSS558: Applied Distributed Computing [Fall 2017] Institute of Technology, University of Washington - Tacoma	L19.27

REPLICATION SCALABILITY - 2

- Only solution is often to ***relax*** the consistency constraints
- Updates do not need to be executed as atomic operations
- Try to avoid instantaneous global synchronizations
- TRADEOFF: consistency
  - Not all copies may always be the same everywhere
- Whether consistency requirements can be relaxed depends on:
  - Access and update patterns
  - Use cases of the data
- Range of consistency models exist
- Implemented with distribution and consistency protocols

December 7, 2017	TCSS558: Applied Distributed Computing [Fall 2017] Institute of Technology, University of Washington - Tacoma	L19.28

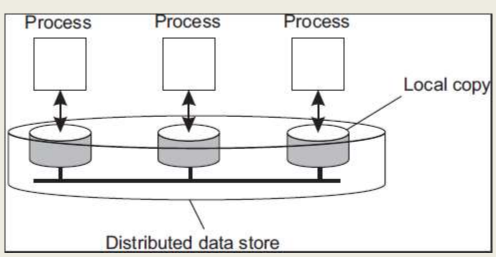


DATA CENTRIC CONSISTENCY MODELS

L19.29

DATA-CONSISTENCY MODELS

- Data consistency is discussed in the context of
  - Distributed shared memory
  - Distributed shared database
  - Distributed shared file system
- Generically referred to as a ***"data store"***
- Each process has a nearby replica:



December 7, 2017	TCSS558: Applied Distributed Computing [Fall 2017] Institute of Technology, University of Washington - Tacoma	L19.30

DATA-CONSISTENCY MODELS

- **CONSISTENCY MODEL**
- Rules that must be followed to ensure consistency
- Represents a contract between processes and data store
- If processes agree to obey certain rules, store promises to work correctly

- No general rules for loosening consistency
- What can be tolerated is highly application dependent

- **Three types of inconsistencies**
- Data variation
- Staleness
- Ordering of update operations

CONTINUOUS CONSISTENCY

- Ranges assigned to "what is allowed" for these deviations:
  - How much data variation?
  - How old/stale can the data be?
  - How much can ordering of update operations vary?

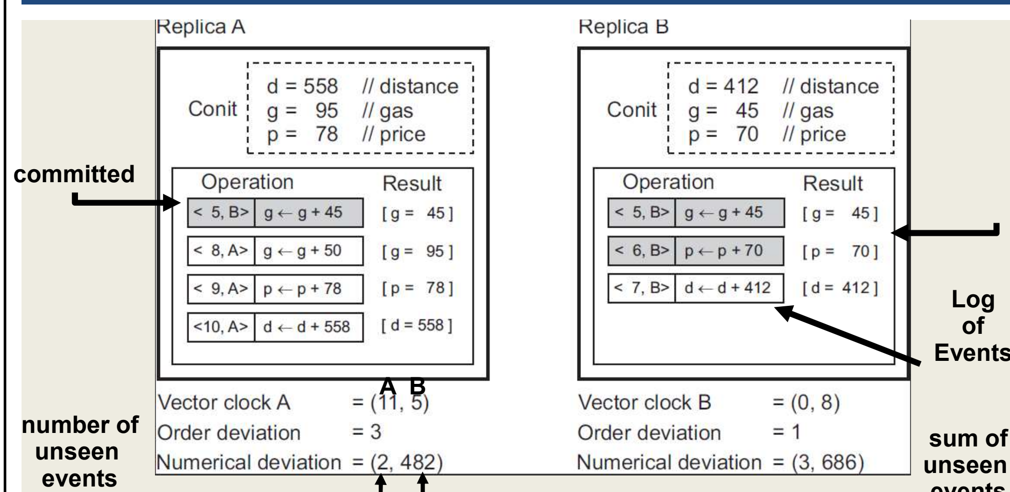
- Idea is to specify bounds for numeric deviation:
- **Relative numeric deviation**: 2% (percent)
- **Absolute numeric deviation**: .2 (implies a particular scale)

- **Numeric deviation**: may also refer to the number of updates applied to a replica
- **Staleness**: specifies bounds relative to time, e.g. how old?
- **Ordering of updates**: updates applied tentatively to local copy; may later be rolled back and applied in different order before becoming permanent

CONSISTENCY UNITS (CONIT)

- Abbreviated as "Conit"
- Specified the unit to measure consistency
- **Example**: Tracking fleet of rental cars
- Variables for a "conit":
- (g) gasoline consumed
- (p) price paid for gasoline
- (d) distance traveled
- Server keep conit consistently replicated

CONSISTENCY UNIT (CONIT)



- Each process has vector clock (known time @A, known time @B)

SEQUENTIAL CONSISTENCY

- Result of any execution is the same as if the operations of all processes were executed in some sequential order, and the operations of each individual process appear ***in this sequence*** in the order specified by its program.

Sequentially Consistent

P1: W(x)a				
P2:	W(x)b			
P3:		R(x)b		R(x)a
P4:			R(x)b	R(x)a

NOT Sequentially Consistent

P1: W(x)a				
P2:	W(x)b			
P3:		R(x)b		R(x)a
P4:			R(x)a	R(x)b

- Exact order seen by processes **DOES NOT MATTER**
- As long as they all agree
- Processes here must see: R(x)b, then R(x)a

CAUSAL CONSISTENCY

- Writes that are potentially causally related ***must be seen*** by all processes ***in the same order***.
- ***Concurrent writes*** may be seen ***in a different order*** by different processes.
- Concurrent writes happen with no READS in between
  - Events can be seen as "concurrent events"
- **Which writes are concurrent?**

P1: W(x)a				W(x)c		
P2:		R(x)a	W(x)b			
P3:		R(x)a			R(x)c	R(x)b
P4:	R(x)a				R(x)b	R(x)c

- Note how the reads after the concurrent write for P3 and P4 are ***in a different order.***
- This is ok with causal consistency

CAUSAL CONSISTENCY - 2

- **Which timing graphs uphold causal consistency?**
- (A)

P1: W(x)a				
P2:	W(x)b			
P3:			R(x)b	R(x)a
P4:			R(x)a	R(x)b

- (B)

P1: W(x)a				
P2:	R(x)a	W(x)b		
P3:			R(x)b	R(x)a
P4:			R(x)a	R(x)b

- **Which writes are concurrent?**
- For (B), since R(x)a can influence W(x)b, the subsequent reads by P3 and P4 **must be in the same order** . . .

ENTRY CONSISTENCY

- Locks can be used to control access to data members
- Releasing a lock tells the distributed system that a variable needs to be synchronized / updated.
- A simple read without obtaining a lock may result in a stale value

P1:	L(x)	W(x)a	L(y)	W(y)b	U(x)	U(y)			
P2:							L(x)	R(x)a	R(y) NIL
P3:								L(y)	R(y)b

- Here P2 does not obtain L(y) before reading y R(y)
  - P2 receives a stale/old value

CONSISTENCY VS. COHERENCE

- **Consistency models** define what to expect when processes concurrently operate on distributed data
- Data is consistent, if it adheres to the rules of the model
- **Coherence models**: describe what can be expected for only ***a single data item***
- Data item is replicated
- Data item is coherent when copies adhere to consistency model rules
- Coherence often uses **sequential consistency** applied to a single data item
- For concurrent writes, all processes eventually see the same order of updates

EVENTUAL CONSISTENCY

- If no new updates are made to a given data item, eventually all accesses to that item will return the last updated value.
- System must reconcile differences between multiple distributed copies of data
- Servers must exchange data updates
- Servers must reconcile updates to agree on final state
  - Read repair: correction done when read finds inconsistency
  - Write repair: correct done on write operation
  - Asynchronous repair: correction done independently from read and write

EVENTUAL CONSISTENCY - 2

- Most processes mainly read from data store
  - Rarely update data
- How fast should updates be made to read-only processes?
- Example: Content Delivery Networks (video streaming)
  - Updates are propagated slowly
- Conflicts: write-write and read-write (most common)
- Often acceptable to propagate updates in a lazy manner when most processes perform only READ-ONLY access
- All replica gradually (eventually) become consistent

QUESTIONS

EXTRA SLIDES

43